PREDICTING THE DAILY CLOSING PRICES OF S&P 500 STOCKS VIA NN ARCHITECTURES TRAINED ON TIME SERIES DATA

Student: Gun Kaan Aygen

ABSTRACT

In this project, we compare the accuracy of three distinct regression models, leveraging three distinct deep Neural Network architectures, in predicting the daily closing price of the stock tickers enlisted in the S&P 500 index. The three competing NN architectures are as follows: a Feed-Forward Deep Neural Network, a Convolutional Neural Network (CNN), and finally, a CNN-LSTM model that hybridizes the CNN architecture with consequent Long-Short-Term-Memory layers. The three NN models are trained and tested on the same dataset, consisting of daily stock market data - for 502 out of the total 503 stock tickers enlisted in the S&P 500 index - and the daily closing price of the S&P 500 Index in the last 10 years. Per each input to the model, the predicted label is a scalar float value, which represents the closing price of a given S&P 500 stock during a chosen trading day in the last 10 years. The Feed-Forward Neural Network model measures the baseline accuracy of a Deep NN architecture in predicting the closing prices of the 502 tickers. Furthermore, the CNN model first utilizes convolutional layers to extract feature maps from the input, which are then fed into fully connected layers that predict the scalar label. Finally, the third model first utilizes convolutional kernels to extract feature maps; the output of the convolutional layers are sequentially inputted through LSTM layers, and then, the output of the LSTM layers is sequentially fed through fully connected layers, which make the scalar label prediction.

1 DATA

1.1 DATASET COLLECTION:

The dataset is obtained from:

`https://www.kaggle.com/datasets/andrewmvd/sp-500-stocks`

The Kaggle dataset is updated daily and consists of three tables.

Table 1: It is the primary table enlisting 'Stock Prices.' It is 2-dimensional, with columns [Date, Symbol/Ticker, Open, High, Low, Volume, Adjacent Close, Close]. Each row in this table represents a given S&P 500 stock ticker's [Open, High, Low, Volume, Adjacent Close, Close] values, while being traded in the stock market, on any given trading day in the last 14 years.

Table 2: The second table in the dataset is the "S&P 500 Index Price" table, which is 2-dimensional, with columns [Date, S&P 500 Index Price] for every trading day in the last 10 years.

Table 3: The third and final table in the dataset, the 'S&P 500 Companies' table, is a 2-dimensional table with 503 rows, one for each of the 503 stock tickers listed in the S&P 500 index. It includes columns [Exchange, Symbol, Name, Sector, Industry, Market Cap, Ebitda, Revenue Growth].

1.2 DATA PREPROCESSING

The data is loaded from Kaggle and stored in a Google Drive folder. The data preprocessing in the script begins by using Pandas to read the three data tables and converting the 'Date' field in **Table 1** and **Table 2** to Panda's date-time format. Then, **Table 1** is outer (or left) merged with **Table 2**

on the 'Date' column, such that the resultant table is **Table 1** including the S&P 500 Index price for each date. Then, the ['Symbol/Ticker,' 'Exchange,' 'Sector,' 'Industry'] columns of **Table 3** are outer-merged with the expanded **Table 1** on the [Symbol/Ticker] column. Thus, we create one 2-dimensional dataset out of the three 2-dimensional data tables; this yet raw dataset is shaped and structured after the above-explained operations as follows,

Date	**Exch.**	**Sect.**	**Indust.**	**Open**	**High**	**Low**	**Close**	**Adj Close**	**Volume**	**S&P 500**	**Ticker**
datetime	*string*	*string*	*string*	*float*	*float*	*float*	*float*	*float*	*float*	*float*	*string*

Table 1: Initial Merged Data

The number of unique values in each of the above categorical fields are as follows:

Unique Tickers: 502, Unique Sectors: 11, Unique Industries: 116, Unique Exchanges: 4

The first strategy we implemented was one-hot encoding all the categorical fields with binary integer values, thus creating $502 + 11 + 116 + 4 = 633$ new binary one-hot encoded columns to replace the original string-valued categorical columns. Nonetheless, we gave up on that strategy since the resultant data matrix was very sparse, since for each row strictly only 4 out of the 633 binary one-hot encoded columns would have non-zero entries, and the remaining 629 positions would be strictly zero-valued. Given the sample complexity of the dataset for the task, the sparsity induced by one-hot encoding categorical features ended up as a problem when all three models were trained, and the data was preprocessed as such. That is, the training loss would virtually not decrease after 50 epochs. Hence, we decided that one-hot encoding categorical fields with binary-valued columns was not a good strategy, considering the task and the dataset.

Therefore, we implemented the following strategy: We integer-encoded [Sector, Industry, Symbol/Ticker] columns and dropped these categorical columns from the Initial Merged dataset (Table 1) above. Furthermore, the [Exchange] column of Table 1 is one-hot encoded with binary-valued columns; thus, it is dropped from Table 1 and instead replaced by four binary integer-valued columns [BTS, NGM, NMS, NYQ], one for each of the four unique exchange values in the dataset. As all categorical columns are, as such, dropped in place from Table 1, the table is hereon denoted as the **numerical features matrix**, as it no longer contains any categorical variables. Then, the three distinct integer-encoding maps of [Sector, Industry, Symbol/Ticker] columns are converted into input tensors to be fed as input into embedding layers. As will be explained later, all three of our models contain three embedding layers before the head of their respective defining architecture. The input to these three embedding layers is the three integer-encoding tensors of the [Sector, Industry, Symbol/Ticker] columns. The outputs of the three embedding layers are the learned embedding representations from the integer encoding maps of these three categorical columns. The outputs of the three embedding layers are then concatenated with the remaining numerical features in the dataset in each model; the concatenated vectors are then inputted to each respective model architecture; this is the case with all three models we implemented.

That said, after the dropping of the [Sector, Industry, Symbol/Ticker] columns and one-hot encoding the [Exchange] column, the resultant numerical features matrix is preprocessed further with the following steps: The "Close" column is split from the numerical features matrix and shaped into a column vector of ground-truth labels; the single [Date] column in the numerical features matrix originally in the format of "Year-Month-Day" is dropped and replaced by three integer-valued columns: [Year, Month, Day]. Finally, all columns of the numerical features matrix are normalized.

Thus, at the end of preprocessing, we create 5 tensors from the dataset above, that is, 3 integer-encoding vectors and, 1 matrix of numerical features, and 1 vector of labels. The matrix of numerical features (dataset cardinality) contains 1232675 rows; each row represents a stock ticker during a given trading day in the last ten years. Hence, the integer-encoding and label tensors are shaped as follows,

Labels Tensor: $\mathbf{y} \in \mathbb{R}^{1232675}$; Ticker Encoding Tensor: $\mathbf{t_e} \in \mathbb{R}^{502}$;

Sector Encoding Tensor: $\mathbf{s_e} \in \mathbb{R}^{11}$; Industry Encoding Tensor: $\mathbf{i_e} \in \mathbb{R}^{116}$

The preprocessed numerical features matrix is structures and shaped as follows in Table 2,

Year	MonthDay	BTS	NGM	NMS	NYQ	Open	High	Low	Adj Close	S&P 500	Volume
int	*int*	*int*	*int*	*int*	*int*	*int*	*float*	*float*	*float*	*float*	*float*	*float*

Table 2: Preprocessed Numerical Features Matrix of Dim. 1232675×12

2 MODELS

2.1 MULTI-LAYER PERCEPTRON / FEED FORWARD NN ARCHITECTURE

Embedding Layers: The embedding layers are inputted $\mathbf{t_e}$, the 1-d ticker encoding tensor, $\mathbf{s_e}$, the 1-d sector encoding tensor, and $\mathbf{i_e}$, the 1-d industry encoding tensor; the output of the embedding layers, that is the learned embeddings, are as shaped follows,

Layer	Mapping
Ticker Embedding	$\mathbf{t_e} \to \mathbb{R}^{30}$
Sector Embedding	$\mathbf{s_e} \to \mathbb{R}^{10}$
Industry Embedding	$\mathbf{i_e} \to \mathbb{R}^{15}$

Concatenation and Input Processing: The outputs of the embedding layers are concatenated with rows of the Numerical Features matrix (Table 2), creating an input of size $(\textit{batch size}) \times (30 + 10 + 15 + 13) = (\textit{batch size} \times 68)$

Fully Connected Layers: The concatenated input is passed through eighteen fully connected layers with LeakyReLU activation and batch normalization:

Layer Size	Activation	Normalization	Dropout Rate
10112	LeakyReLU ($\alpha = 0.01$)	Yes	0.3
5046	LeakyReLU ($\alpha = 0.01$)	Yes	0.3
2048	LeakyReLU ($\alpha = 0.01$)	Yes	0.3
1792	LeakyReLU ($\alpha = 0.01$)	Yes	0.3
1536	LeakyReLU ($\alpha = 0.01$)	Yes	0.3
1152	LeakyReLU ($\alpha = 0.01$)	Yes	0.3
1024	LeakyReLU ($\alpha = 0.01$)	Yes	0.1
896	LeakyReLU ($\alpha = 0.01$)	Yes	0.1
768	LeakyReLU ($\alpha = 0.01$)	Yes	0.1
640	LeakyReLU ($\alpha = 0.01$)	Yes	0.1
512	LeakyReLU ($\alpha = 0.01$)	Yes	0.1
384	LeakyReLU ($\alpha = 0.01$)	Yes	None
256	LeakyReLU ($\alpha = 0.01$)	Yes	None
128	LeakyReLU ($\alpha = 0.01$)	Yes	None
64	LeakyReLU ($\alpha = 0.01$)	Yes	None
32	LeakyReLU ($\alpha = 0.01$)	Yes	None
16	LeakyReLU ($\alpha = 0.01$)	Yes	None
1	None	None	None

Output Layer: The final output is $\mathbf{y}' \in \mathbb{R}^{batchsize}$ for every batch processed and $\mathbf{y}' \in \mathbb{R}^{1232675}$ for the whole dataset. That is, the network outputs a scalar value $y \in \mathbb{R}$ (float) for every row that is processed in training and testing, as the task is regression.

Loss Function: Mean Square Error with learning rate $= 0.01$ and weight decay $= 10^{-5}$

Learning Rate Scheduler: *ReduceLROnPlateau* (mode='min', factor=0.9, patience = 4)

That is, if the MSE loss is not decreasing for 4 consecutive epochs, then the running learning rate is multiplied by a factor of 0.9. The initial learning rate beginning the training in epoch 0 is 0.01

Optimizer: Adam; **Batch Size:** 1024 and **Training Epochs:** 500

2.2 CONVOLUTIONAL NEURAL NETWORK MODEL

The CNN architecture begins with the identical embedding layers as in the MLP; the learned embeddings are concatenated with the numerical features to be fed into the CNN. Hence, in the beginning of the CNN, the embedding layers and the concatenation of numerical inputs columns with the learned embeddings are identical as in the MLP. The difference is that the CNN model has convolutional layers following the concatenation operation; the concatenated input is passed as input to the first convolutional layer, and there on sequentially through the deeper convolutional layers, creating feature maps at each depth of the convolution. The last convolutional layer's output is flattened and sequentially inputted through five dense layers (as demonstrated below) which yields the final scalar prediction of the network for any given row in the dataset.

Embedding Layers: The embedding layers are inputted $\mathbf{t_e}$, the 1-d ticker encoding tensor, $\mathbf{s_e}$, the 1-d sector encoding tensor, and $\mathbf{i_e}$, the 1-d industry encoding tensor; the output of the embedding layers, that is the learned embeddings, are as follows,

Layer	Mapping
Ticker Embedding	$\mathbf{t_e} \to \mathbb{R}^{30}$
Sector Embedding	$\mathbf{s_e} \to \mathbb{R}^{10}$
Industry Embedding	$\mathbf{i_e} \to \mathbb{R}^{15}$

Concatenation and Input Processing: The outputs of the embedding layers are concatenated with rows of the Numerical Features matrix (Table 2), creating an input of size $(\textit{batch size}) \times (30 + 10 + 15 + 13) = (\textit{batch size} \times 68)$

Convolutional Layers:

Convolutional layers are sequentially applied. Conv1 layer processes the concatenated input of the learned embeddings with the numerical features:

Layer	Channels In/Out	Kernel Size	Stride
Conv1	1 / 16	3	1
Conv2	16 / 32	3	1
Conv3	32 / 64	3	1
Conv4	64 / 128	3	1

Flattening and Fully Connected Layers:

The output of the last convolutional layer is flattened, and the flattened tensor is sequentially processed through several fully connected layers all with ReLU activation:

Layer	Output Size
Fully Connected 1	512
Fully Connected 2	256
Fully Connected 3	128
Fully Connected 4	64
Fully Connected 5	1

Output The final output is $\mathbf{y}' \in \mathbb{R}^{batchsize}$ for every batch processed and $\mathbf{y}' \in \mathbb{R}^{1232675}$ for the whole dataset.

Loss Function: Mean Square Error with learning rate $= 0.01$

Learning Rate Scheduler: *ReduceLROnPlateau* (mode='min', factor=0.9, patience = 2)

That is, if the MSE loss is not decreasing for 2 consecutive epochs, then the running learning rate is multiplied by a factor of 0.9. The initial learning rate beginning the training in epoch 0 is 0.01

Optimizer: Adam; **Batch Size:** 1024 and **Training Epochs:** 500

2.3 CNN & LSTM ARCHITECTURE

This third architecture has the same exact three embedding layers as in the head of the CNN and MLP models. Identical to the MLP and CNN model, the learned embeddings are concatenated with the numerical features after the embedding layers. Then, the concatenated input is passed through a 3-layered CNN, and the output of the 3-layered CNN is inputted to sequential LSTM layers. The output of the final LSTM layer is then inputted into a 5-layered fully connected perceptron network, which yields the final scalar prediction of the model.

Embedding Layers The embedding layers are inputted $\mathbf{t_e}$, the 1-d ticker encoding tensor, $\mathbf{s_e}$, the 1-d sector encoding tensor, and $\mathbf{i_e}$, the 1-d industry encoding tensor; the output of the embedding layers, that is the learned embeddings, are as follows,

Layer	Mapping
Ticker Embedding	$\mathbf{t_e} \to \mathbb{R}^{30}$
Sector Embedding	$\mathbf{s_e} \to \mathbb{R}^{10}$
Industry Embedding	$\mathbf{i_e} \to \mathbb{R}^{15}$

Concatenation and Input Processing: The outputs of the embedding layers are concatenated with rows of the Numerical Features matrix (Table 2), creating an input of size $(\textit{batch size}) \times (30 + 10 + 15 + 13) = (\textit{batch size} \times 68)$

Convolutional Layers:

Layer	Channels In/Out	Kernel Size	Stride
Conv1	1 / 16	3	1
Conv2	16 / 32	3	1
Conv3	32 / 64	3	1

LSTM Layers

The feature maps extracted from the final Convolutional layer is flattened and inputted into the first layer of LSTM.

LSTM Layer	Hidden Size	Num Layers
LSTM1	128	8
LSTM2	256	4
LSTM3	512	2

Fully Connected and Normalization Layers

The output of the final LSTM layer is inputted through five fully connected layers with ReLU activation and batch normalization:

FC Layer	Output Size	Normalization
FC1	512	Yes
FC2	256	Yes
FC3	128	Yes
FC4	64	Yes
FC5	1	No

Output: The final output is $\mathbf{y}' \in \mathbb{R}^{batchsize}$ for every batch processed and $\mathbf{y}' \in \mathbb{R}^{1232675}$ for the whole dataset.

Loss Function: Mean Square Error with learning rate $= 0.01$

Learning Rate Scheduler: *ReduceLROnPlateau* (mode='min', factor=0.9, patience = 3)

That is, if the MSE loss is not decreasing for 3 consecutive epochs, then the running learning rate is multiplied by a factor of 0.9. The initial learning rate beginning the training in epoch 0 is 0.01

Optimizer: Adam; **Batch Size:** 1024 and **Training Epochs:** 500

3 Training and Testing Findings:

The three models were each trained for 500 epochs, and the MSE per epoch was logged for each model for each epoch.

3.1 Model Loss Comparison

The following table demonstrates the Test Loss after the training phase, as well as the Training Loss of the three models during epochs 1, 100, 200, 300, 400, 500.

Model	Epoch 1	Epoch 100	Epoch 200	Epoch 300	Epoch 400	Epoch 500	Testing Loss
MLP	1.35×10^4	8.37×10^2	2.56×10^2	1.95×10^2	8.54×10^1	5.67×10^1	5.88×10^1
CNN	3.20×10^4	3.84	0.412	0.181	0.134	0.118	0.116999070017840
CNN-LSTM	2.04×10^3	5.51	0.967	0.341	0.197	0.139	0.143999448882426

Table 3: Loss values at every 100 epochs and testing loss after training, for all three models

4 Discussion of Procedure and Findings

All three models have been implemented with PyTorch, and data preprocessing has been done using Pandas. The dataset was stored in a Google Drive folder, and all three models were trained and tested on a Google Collab Pro+ Notebook with TP4 GPU (High RAM). The CNN model has demonstrated the best performance among the three models, followed by the CNN-LSTM and MLP models. We also experimented with many pure LSTM architectures that do not incorporate preliminary convolutional layers before the LSTM layers. We observed that it did not perform better than the CNN or the CNN-LSTM. That said, the use of convolutional layers to extract feature maps has been proven effective for regression on closing prices given our dataset and the task.

The relatively poor performance of the MLP model is a finding worth analyzing. Even though we trained many MLP architectures for the task, incrementally increasing the model complexity of the MLP architecture, we could never drive the training loss down to the order of magnitude of 10^1 with any MLP model we implemented in the process. The MLP model reported in this analysis, Model 3, is the most complex MLP model with which we have experimented. We have gradually increased the model complexity of the MLP model, adding more hidden layers and increasing the width of the hidden layers, eventually obtaining the architecture of Model 3. Nonetheless, contrary to our expectations, increasing model complexity has not resulted in the training loss to converge. Instead, increasing the model complexity of the MLP resulted in a slower convergence of the training loss to a higher loss value than observed with the less complex MLP models. We expect that increasing the MLP architecture's complexity would strictly decrease the training loss, given that less complex MLP models were observed to underfit the training data as evident by the non-converging training loss. Hence, the anticipation was to increase the MLP model complexity to decrease underfitting incrementally, yet the opposite ended up happening, where training loss increased instead with higher MLP model complexity. This is strange behavior as one would expect increasing model complexity to surely yield lower training loss, where training loss would converge to 0 if the model were too complex fully overfitting the training data. It would have been a natural outcome for increasing model complexity to yield higher testing loss in the case of overfitting, yet the training loss is expected to decrease.

Moreover, the use of the three embedding layers to learn embeddings for the categorical (integer-encoded) [Symbol, Industry, Sector] columns and then concatenating the learned embeddings with the numerical features matrix, has allowed the training loss to converge much faster to a much smaller value and yielded much lower testing loss, compared to the strategy of one-hot encoding the categorical columns with binary-valued columns. Thus, all three models performed significantly worse without learning the embeddings and concatenating the embeddings with the numerical features. In that regard, one improvement to the pipeline and models that could further increase training

and testing prediction accuracy is learning embeddings for the [Exchange] column and concatenating the learned embeddings with the numerical features matrix rather than one-hot encoding the [Exchange] variable with binary-valued columns. In our data preprocessing pipeline, we do not integer-encode the [Exchange] column, a categorical variable. The [Exchange] column with four unique values is the only categorical feature that we one-hot encode by replacing it with four binary feature-valued columns in the numerical features matrix. Given that dropping the other categorical variables from the numerical features matrix instead of one-hot encoding them has significantly improved learning, it would be a viable attempt also to learn an embedding for the integer-encoding vector of the [Exchange] column and then concatenate the learned embeddings with numerical features, as implemented with the other categorical variables, rather than one-hot encoding this categorical field.

5 Discussion of Prior Work

It would not be logically sound to compare our model's performance to a benchmark performance achieved by an arbitrary regression model that does the same task, as the dataset we used in training and testing has yet to be used for the same task in a published paper available online. However, predicting the closing prices of stocks is a typical application, and we can still compare our model's performance to regression models trained on different datasets for stock closing price prediction. However, the comparison of this accuracy of models trained on different datasets for different stock tickers cannot be directly reduced to the comparison of model performance since the dataset, and the stocks for which regression is done differ between the applications. That said, Roondiwala et al. have achieved a lowest of 7.4×10^{-5} MSE for predicting the daily High / Low / Open / Close prices of stocks listed in the National Stock Exchange of India. Given that our lowest testing error is 0.116999070017840 in terms of MSE, we have that Roondiwala et al. have their best model surpass our best model's performance by a factor of 1.6×10^3. Even though the dataset and the stocks being represented are entirely different in our study and the study conducted by Roondiwala et al., we still can compare the performance as the ultimate task is the same, predicting float-valued price points of a certain number of stock tickers with regression. In that regard, we observe that our best-performing model's Root-MSE testing loss is 0.342. This indicates that our best model, the CNN, on average, is off by 0.342 in terms of predicting closing prices when tested on the test set. Hence, it goes without saying that if the task is to leverage NN models to make closing prices for predictions for S$P 500 stocks to inform daily investment decisions, then a Root-MSE testing loss of 0.342 is not low enough to make the learned model practically successful. Intuitively, the RMSE loss should be driven down at least to the order of magnitude of 10^{-2}. That implies that our best model's RMSE testing error must be further decreased by a factor of at least 3.4×10^0 to achieve an RMSE loss in the order of magnitude of 10^{-2}.

Furthermore, Vijh et al. have achieved the lowest testing RMSE of 0.42 in predicting the daily closing price of the 'Pfizer Inc. stock ticker using an Artificial Neural Network (ANN). Given that our CNN model's testing loss is 0.116999 in terms of MSE, and therefore 0.342 in terms of RMSE, we have that our CNN model's testing accuracy surpasses that of the most successful model developed by Vijh et al. singularly trained and tested on the 'Pfizer Inc.' stock. It is worth bearing in mind that the ANN developed by Vijh et al. that is being referred to has been trained on a different dataset, and the model is trained on and makes predictions for the closing prices of individual stocks. However, the most accurate closing price prediction model by Vijh et al., on average, was less accurate in testing with its best performance for the 'Pfizer Inc. stock, compared to our CNN model's average testing performance making predictions for any of the 502-many S&P500 tickers.

6 Conclusion

It is observed that the CNN and CNN-LSTM models are successfully learning from the data, given that the models generalize well to the testing data, and the training loss consistently decreases over 500 epochs. To further improve the prediction performance of the models, the hyper-parameters should be fine-tuned by utilizing model selection curves and plotting the validation loss across different settings of hyper-parameters to identify the most optimal setting that minimizes the testing loss. It would be reasonable to expect that increasing the architectural complexity (the number of parameters) in the CNN and CNN-LSTM models would allow the training loss to further converge

to a smaller value, as the models are still relatively underfitting but not yet overfitting the training data, given the current model complexities of Models 2 and 3. Furthermore, the reason causing Model 1, the MLP, to perform poorly compared to the other two models should be further inspected. One reason why increasing the model complexity did not result in lower training loss with the MLP could be because the MLP architecture was already excessively complex, and therefore, the MLP could be possibly overfitting the noise in the data with the level of its excessive complexity rather than overfitting to the actual trend in the data, which would have yielded a training loss converging to zero.

Furthermore, the prediction accuracy of all three models could be improved using a dataset with more numerical features. All three models we have implemented only consider the [Year, Month, Day, Open, High, Low, Adjacent Close, S$P500 Index Price, Volume]numerical features for predicting the closing price for each row in the dataset. However, if more numerical financial indicators, such as daily Price-to-Earnings, Debt-to-Equity, or Dividend Payouts, were included for each row in the dataset, all three models would be expected to make much more accurate predictions. Hence, the data is highly limited in terms of the features it allows our models to consider and learn from to make predictions for the regression task.

It is worth noting that we have indeed experimented with building our pipeline with two alternative datasets containing a greater cardinality of numerical features for each row vector in the dataset. Nonetheless, the data integrity of these two alternative datasets was discovered to be highly compromised, with most of the data points missing in each alternative. Therefore, we were not able to identify a better-suited dataset for the task than the one being used, as the dataset we have used is updated daily, and less than $\%0.03$ of the entire dataset is corrupted due to missing fields. We have mitigated the corrupted rows with missing fields by dropping them from the dataset, yet a more efficient approach for optimizing the learning of the models could have been filling the missing fields with the correct values.

Another limitation of the utilized dataset stemmed from being unable to incorporate **Table 3**, as mentioned in the 'Data Collection' section, advantageously in the numerical features matrix.**Table 3** consisted of [Exchange, Symbol, Name, Sector, Industry, Market Cap, Ebitda, Revenue Growth] columns for each of the 503 stocks enlisted in the S&P 500 index. Nonetheless, **Table 3** is not a time-series dataset; it consists of only 503 rows with the current values of [Market Cap, Ebitda, Revenue Growth] for each stock. However, since our stock price data goes back 10 years, it would not have made logical sense to include these static features demonstrating current statistics in our dataset of time series data. If we had the daily time series data for these [Market Cap, EBITDA, Revenue Growth] for each stock during all trading days in the last ten years, as with the S&P 500 Index Price, we would then have incorporated these numerical features in the numerical features matrix and, thus, most possibly improve learning.

Finally, although a total of 503 stock tickers are enlisted in the S&P 500 index, we only have 502 tickers represented in the dataset. After rows with empty fields in the dataset were dropped, we observed that the stock with the 'GEV' ticker value no longer contained any representative samples in the dataset. That is because all the rows representing the performance of the 'GEV' tickered stock had at least one empty field. Hence, in a better implementation, we would not drop the corrupted rows for the 'GEV' ticker. Instead, ideally, we would have populated the missing fields for the 'GEV' ticker rows with correct values and other missing entries in the dataset that we have indeed dropped.

7 GitHub Repo and the Code:

The code for the project is in the following Github repo, `https://github.com/kaanaygen/StockPredict.git`

8 REFERENCES

Roondiwala, Murtaza & Patel, Harshal & Varma, Shraddha. (2017). Predicting Stock Prices Using LSTM. International Journal of Science and Research (IJSR). 6. 10.21275/ART20172755.

Vijh, M., Chandola, D., Tikkiwal, V. A., & Kumar, A. (2020). Stock closing price prediction using Machine Learning Techniques. Procedia Computer Science, 167, 599–606. https://doi.org/10.1016/j.procs.2020.03.326