Usage of a Encoder Based Transformer model, a BiLSTM model and an Hybrid model to predict stocks prices

Smaldini Raffaele and Ardillo Michele

August/September 2023

Contents

Abstract

This projects starts with the idea to find the best deep-learning model to predict stock prices in three period of time (day, two days and a week). To accomplish this goal three models architecture are created, with one of them being an hybrid of the two. To train and test these models different approaches were tempted

1 Introduction

The greatest efforts were concentrated on finding the right combination between "light models" (in parameters), good metrics with a few days of trend (lags) of the Open, High, Low, Close values and models that were able to best capture the trend performance (not just MSE and MAE). The architectures were designed exclusively to predict the day immediately following the sequence of lags, as it has been decided to use an autoregressive approach to predict up to the 7th day. Because the models architectures, 2 different tests were carried out:

- The first test was carried out immediately after training, to define which models had good metrics (MAE, MSE) in predicting the day immediately following the lags.

- The second testing was carried out using an autoregressive approach, so as to be able to predict up to the 7th day.

These models can also predict for many more days, but it is advisable not to exceed lags/2 days as the accuracy (in trend and metrics) of the model could decrease significantly. The strengths of this approach are the following:

- the same pipeline and preprocess was used for all architectures;

- using an autoregressive test is possible to predict every day up to the seventh, so consequently is possible to trace a trend of the predictions to understand if the model is actually learning the trends in the data.

- predicting a single day instead of 3 (1st, 2nd and 7th day) for 4 different values is simpler and allows the model to focus on cyclical patterns and not just on the numerical values to be predicted.

All the previous considerations were made to respect both the project requests and to obtain good metrics for light models (in parameters) that were able to capture trends decently. During the training phase it was decided to use a number of lags equal to 25 (recommended as a starting point); it was also tested with a number of lags equal to 45, allowing the model to better capture the trends although slightly losing precision in predicting the correct value and increasing the training time, but keeping the architecture and hyperparameters almost completely unchanged. This project aim to describe different machine learning models and their performance metrics in predicting stock prices for various companies.

The passage discusses several machine learning models for stock price prediction, each trained on different amounts of historical data.

BiLPET Large Model: This model is trained on a substantial amount of data, sometimes spanning up to forty years of stock data for each company. It is not described as a great tool for autonomous prediction but may perform well for ”autobegging.”

BiLPET Small Model: Similar to the BiLPET Large model, but with a smaller dataset, typically covering ten years of stock data.

EBiL Large Model: This model provides predictions for stock prices over several days, with both predicted and real values provided. Metrics such as mean absolute error (mae), mean squared error (mse), and percentage deviation from actual values (pda) are given.

EBiL Small Model: Similar to EBiL Large but with less data.

Bagging Metrics: These metrics represent the performance of the models when used for bagging. Specific metrics are provided for different companies, including Apple (APPL), Amazon (AMZN), Google (GOOGL), and Microsoft (MSFT).

In general, these models are evaluated based on their ability to predict stock prices, with metrics such as mae, mse, and pda indicating their accuracy and performance. The choice between large and small models appears to depend on the amount of historical data available and the specific needs of the prediction task.

2 Data Pre-process

In this section all the preprocess and everything outside the main models are discussed.

2.1 Utils.py

In the **utils.py** file there is a collection of those function that are useful for the aim of this project. For doing so other external pyhton libraries are used such as Pandas[1] in order to load the CSV files needed and Matplotlib[2] to visualize all the data. The *load_csv* function is used to load more CSV file in a python dictionary. Eg: if there are 4 CSV files it loads them in a python dictionary with the name of four firm as key and the respective dataframe as value. *Show_trend* has two mode: ”s” (single mode); it shows one single pandas dataframe by splitting the 4 values in 4 different curves, meanwhile ”m” (multiple mode); shows trends for all dataframes for all firms. *Trend_plot* function is used to visualize data in single mode, this function is suited for *show_trend* function. *Clean_data* function helps to erase all those useless data from the dataset, such as: date, volume and adj Close. Also a function to create a cyclic encoder (*cyclical_encoder*) by calculating a sine and cosine transform. So instead of using its original values the model uses the transform to preserve the seasonality. To get an easier and quicker way to get the t(n+1) prediction time-lags are an amazing toll. So that the *generate_time_lags* function creates time lags given data, the feature (in this case columns) and the no of lags. In this case, after some testing, for this purpose, 25 is a valid no of lags.

2.2 Preprocess.py

Here is where the pre-processing is done using not just the function from the **utils.py** file but also from the Scikit-Learn[3] library, to be more precise: the train_test_split for the model selection and the MinMaxScaler for the preprocessing, and the PyTorch[4] library. The first function is the *dataframe_fragmentation*. Here since the models is focused on Open, Closed, High and Low prices of each stocks a fragmentation of all datasets is needed. Ence as input it takes data, the number of lag dimensions. Here the *generate_time_lags* from utils.py is used, in days to divide the data and returns a list of dataframe, one for each label (in addition to the four feature explained before there will also be two more if cyclic_encoding is true: "Embed" and "year"). The *merge_feature* function takes as input the lagged list created in the latter function and each label so that all frame with the same days are merged in one. it creates a vector of lags for all four values. It helps to apply lag functions. All the following function (feature_label_split, train_val_test_split or train_val_split) are used to divide the data in train, test and validation. Normalize function is designed to normalize all data to accomplish this goal both MinMaxScaler and StandardScaler might be used. However MinMaxScaler is more suited since it gave better result. The next function (data_loader) it's meant to load the data on some tensor. Here PyTorch serves the purpose to create and load data to tensors. *Noise_sample_gen* creates a Gaussian Noise given a duplicate probability using a specific Gaussian STD. It duplicate a dataset row, add noise for all data except for the firm label. This function even if in the files was not used for the final purpose since it ruined the validation loss. In *Data_cyclilc_encoding* all time stamps from the dataset are transformed in data-time. It extracts the most relevant features eg: week of the year, year. Those data are then given to the *cyclic_encoding_adapter* function, it transform 8 of the 9 extracted features (4 for the cosine and 4 for the sine) in a unique scalar value for a specific day of the year. The Year column will be later used to train each model, too.

3 Main

Main.py starts of course, with the importing of all dependencies. From PyTorch[4] the NN and the optimizer are imported. From BiL, Encoder+BiLSTM(BiLPET and EBiL) and Encoder Based Tansfomer (ToM) the proper Modesls are imported. From **utils.py** and **preprocess.py** all the methods useful for the data preprocess. **OptimzedTraining.py** is the training method class.

3.1 Starting the models and prerpocess

If the machines supports Cuda Cores from Nvidia GPU the model will be trained by using them. Now data are loaded thanks to *load_csv* function and *show_trend* is used to plot all data graphs. As shown here:

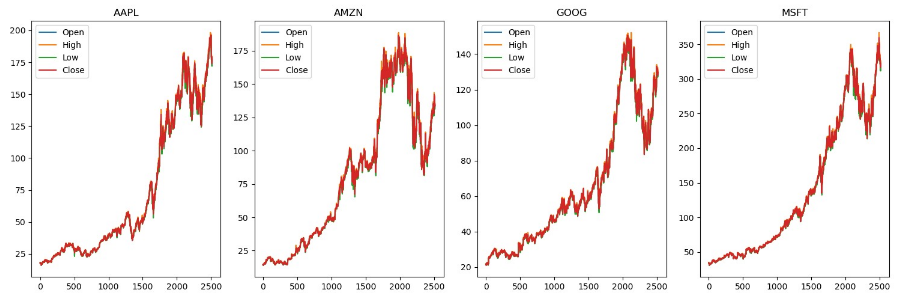
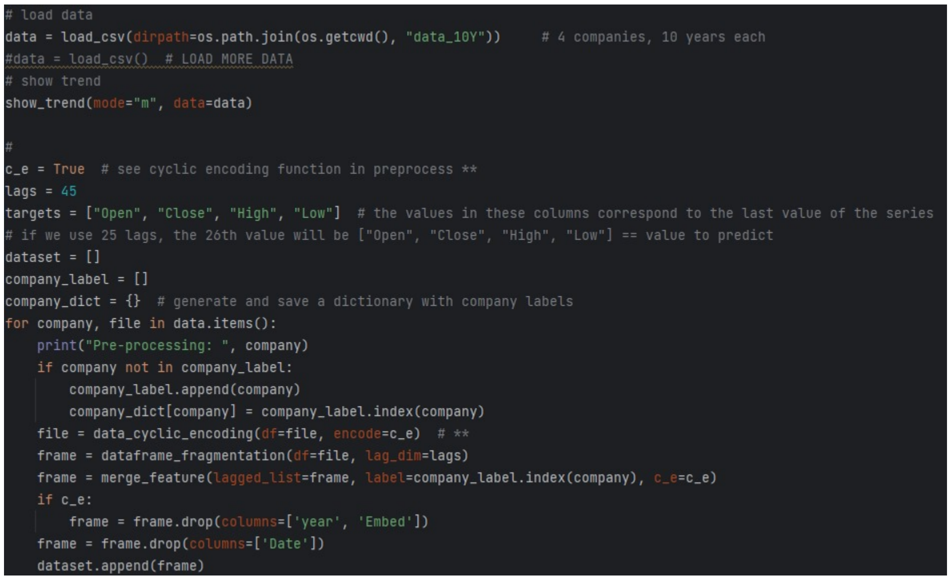


Figure 1: Trend of all four companies(Open, High, Low, Close)

Then a dataset variable is initialized as a list that will contain different firms preprocessed datasets. Later the company dictionary is initialized, its purpose is to save the firm name as key and labels as value. In the for cycle shown here



```
# load data
data = load_csv(dirpath=os.path.join(os.getcwd(), "data_10Y"))     # 4 companies, 10 years each
#data = load_csv()  # LOAD MORE DATA
# show trend
show_trend(mode="m", data=data)

#
c_e = True  # see cyclic encoding function in preprocess **
lags = 45
targets = ["Open", "Close", "High", "Low"]  # the values in these columns correspond to the last value of the series
# if we use 25 lags, the 26th value will be ["Open", "Close", "High", "Low"] == value to predict
dataset = []
company_label = []
company_dict = {}  # generate and save a dictionary with company labels
for company, file in data.items():
    print("Pre-processing: ", company)
    if company not in company_label:
        company_label.append(company)
        company_dict[company] = company_label.index(company)
    file = data_cyclic_encoding(df=file, encode=c_e)  # **
    frame = dataframe_fragmentation(df=file, lag_dim=lags)
    frame = merge_feature(lagged_list=frame, label=company_label.index(company), c_e=c_e)
    if c_e:
        frame = frame.drop(columns=['year', 'Embed'])
    frame = frame.drop(columns=['Date'])
    dataset.append(frame)
```

Figure 2: For Cycle

it iterates over a dictionary in which a company variable and a file, that is a Pandas dataframe different from all the others. If the firm is not in the company_dict dictionary it will be added. If the cyclic encoder is true the *data_cyclic_encoding* will be called and two more feature are added: *Embed, year*. Embed represents an unique singular value of the year given by the product of temporal features, meanwhile year is mandatory to check the trend over the year (as a period of time). Now a Pandas dataframe is an input of *dataframe_fragmentation*, this function splits features in different dataframes and generates time lags. *Split_values* divides the dataframe in a list of dataframe of features. Then lag are generated thanks to *generate_time_lags* called in *dataframe_fragmentation*, pay attention to the fact that lags are created backwards, however they will be rearranged by "reversed_df = lag_df.iloc[:, ::-1]". The output from *dataframe_fragmentation* will be a list of lagged feature dataframes. The sub sequential line of code: "merged_feature" will have the lagged list, c_e and the company label as input. If the cyclic encoding (c_e) is true it will initialize six dataframes (Open, Close, High, Low, year, Embed) otherwise just four (Open, Close, High, Low). In the end it takes in account all lagged dataframes and merged them by "Date" creating a new one. Then the company label will be added as a vector multiplied by the row shape of the later dataframe. If c_e is true "year" and "Embed" columns are dropped. This is done because those columns are linked to the values that have to be predicted (Open, Closed, High and Low). Suppose to choose 25 as no of lags, the model input (x) is a vector of lagged values like here:

lag25_Open	....	lag1_Open	lag25_High	....	lag1_High	lag25_Low	....	lag1_Low	lag25_Close	....	lag1_Close	lag25_year	....	lag1_year	lag25_Embed	....	lag1_Embed	company_label

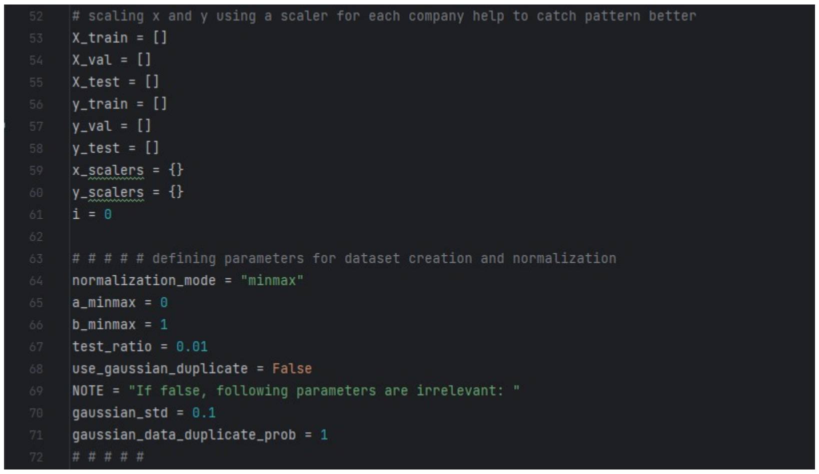
Meanwhile the output (Y) is a vector of four elements:

Open	High	Low	Close

The splitting operation of X and Y is done later, the tables are just an example. If the "year" and "Embed" columns weren't dropped the output should also have their prediction. In the first table is easy to spot that the lags are reversed (reversed_df) from lagN to lag1.

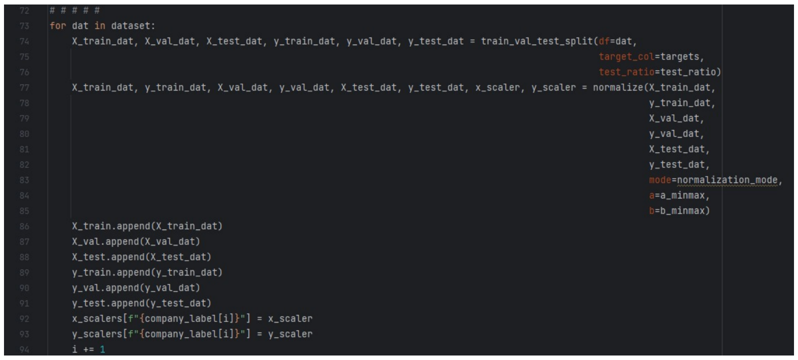
3.2 TrainValTest and Normalizations Steps

Picture that sums up the Train, Test, Val Parameters. Dataframes of the dataset variable are now divided and normalized on their own in a for-cycle. The normalization is done by firm with the same range(0-1) so that the models is focused more on the pattern rather than the value of the firm. The output is now Numpy arrays. From the Normalization function *normalize* x and y scaler are returned too and saved in a dictionary. These will be useful to normalize values during autoregressive testing. All the now Numpy arrays are later append in the lists and then concatenated in a single unique Numpy array.



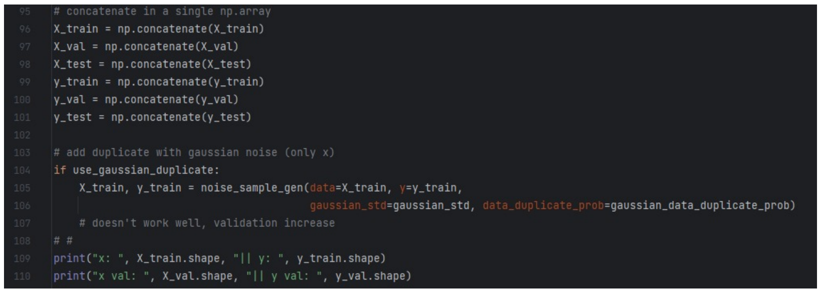
```python
# scaling x and y using a scaler for each company help to catch pattern better
X_train = []
X_val = []
X_test = []
y_train = []
y_val = []
y_test = []
x_scalers = {}
y_scalers = {}
i = 0

# # # # # defining parameters for dataset creation and normalization
normalization_mode = "minmax"
a_minmax = 0
b_minmax = 1
test_ratio = 0.01
use_gaussian_duplicate = False
NOTE = "If false, following parameters are irrelevant: "
gaussian_std = 0.1
gaussian_data_duplicate_prob = 1
# # # # #
```

Figure 3: Initialization of variables for the Train Test Val Split and Norm



```python
# # # # #
for dat in dataset:
    X_train_dat, X_val_dat, X_test_dat, y_train_dat, y_val_dat, y_test_dat = train_val_test_split(df=dat,
                                                                                                  target_col=targets,
                                                                                                  test_ratio=test_ratio)
    X_train_dat, y_train_dat, X_val_dat, y_val_dat, X_test_dat, y_test_dat, x_scaler, y_scaler = normalize(X_train_dat,
                                                                                                           y_train_dat,
                                                                                                           X_val_dat,
                                                                                                           y_val_dat,
                                                                                                           X_test_dat,
                                                                                                           y_test_dat,
                                                                                                           mode=normalization_mode,
                                                                                                           a=a_minmax,
                                                                                                           b=b_minmax)
    X_train.append(X_train_dat)
    X_val.append(X_val_dat)
    X_test.append(X_test_dat)
    y_train.append(y_train_dat)
    y_val.append(y_val_dat)
    y_test.append(y_test_dat)
    x_scalers[f"{company_label[i]}"] = x_scaler
    y_scalers[f"{company_label[i]}"] = y_scaler
    i += 1
```

Figure 4: Split Steps and Norm Steps for each firm dataset



```python
# concatenate in a single np.array
X_train = np.concatenate(X_train)
X_val = np.concatenate(X_val)
X_test = np.concatenate(X_test)
y_train = np.concatenate(y_train)
y_val = np.concatenate(y_val)
y_test = np.concatenate(y_test)

# add duplicate with gaussian noise (only x)
if use_gaussian_duplicate:
    X_train, y_train = noise_sample_gen(data=X_train, y=y_train,
                                        gaussian_std=gaussian_std, data_duplicate_prob=gaussian_data_duplicate_prob)
    # doesn't work well, validation increase
# #
print("x: ", X_train.shape, "|| y: ", y_train.shape)
print("x val: ", X_val.shape, "|| y val: ", y_val.shape)
```

Figure 5: Concatenation of all firm dataset in one single dataset and addition of Gaussian Noisy duplicate

4 Training

In **OptimizedTraining.py** is where the model is trained. Of course, as always the first thing to do is to start with the dependencies as Pandas[1], Matplotlib[2] for plotting, Scikit-Learn[3] for all the metrics and PyTorch[4], Numpy[5] and OS[6] to menage OS directory path.

4.1 Training model

In the first function *train_step* is easy to understand that given and X input calculates y_hat, computes the loss and then computes the gradients backwarding the loss. Usually used for forecasting task models, "MSE Loss" is used (typically with reduction="mean"). In this project L1-smooth Loss has been chosen. It uses the L2 loss function if the element-by-element absolute error is less than 1 and the L1 loss function otherwise. One of the reasons why it was chosen for data training is because this loss is less sensitive to outliers rather than MSELoss. During training, MSELoss was also tested, but with the previously set hyperparameters, models did not perform well; instead L1-smooth allowed the models to function even with a smaller number of parameters. Inside the project models folder there is an example of MSELoss training; Good results were detected with the same pre-established parameters only for the "ToM" architecture. The *train* function simply trains the models. Takes as input: batch size, the *train_loader*, *val_loader*, *n_epochs*, *n_feature* and model name (so that it can creates a dir). Then it prints all the losses: Training Loss and Validation Loss for each of the four labels. In the end saves all the parameters in a dir created in *save_utils*.

4.2 Saving Parameters

In the functions *save_utils* and *save_test_metrics* the OS [6] library is used to store some key information about the models and the training parameters. Eg: scaler, prep_parameters and also which of the models is currently running.

4.3 Evaluation and plot loss

The *evaluate* function has been done just after the training step. It simply computes all the metrics for the current training model and predicts the next lag day. Takes as input the data, size of the batch and the no of feature. Without the usage of gradients it calculates the MeanAbsoluteError and the MeanSquareError. As input both the error take a Numpy Arrays of true_values and predicted_values. These values are lists of values given by the first steps of the *evaluate* function. All the metrics are then saved in the directories. Last function *plot_loss* just print all the graph losses.

4.4 Training Examples

Since the training fase is crucial to catch and understand the best parameters for the architectures a lot of training was done, mainly to understand: best batch dimension, best learning rates, best no of steps and best weight decay. The base ideas for these models was to use shared parameters for every models so that they are easy to compare with each other. As an example: here the BiLSTM model is discussed. By using a very low learning rate,

$$10e^{-}5 \tag{1}$$

the validation losses is to big to be useful. In an other case, with parameters

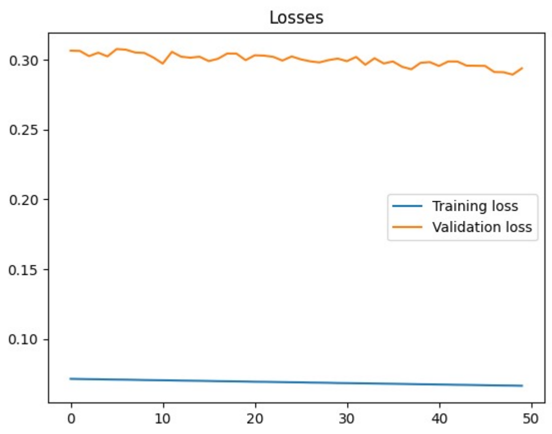


Figure 6: BiLSTM with low learning rate

like:

"cyclic_encoding": true, "lags": "25", "learning_rate": "1e-05", "batch": "256", "epochs": "70", "input_dim": "151", "weight_decay": "0.1", "normalization_mode": "minimax", "a_minmax": "0", "b_minmax": "1", "test_ratio": "0.1", "use_gaussian_duplicate": true, "NOTE: ": "If false, following parameters are irrelevant: ", "gaussian_std": "0.1", "gaussian_data_duplicate_prob": "0.5"

we would have high errors:

"Open_mae": "2.2303843", "Open_mse": "6.9178963", "High_mae": "3.201395", "High_mse": "12.187611", "Low_mae": "2.2558298", "Low_mse": "7.044762", "Close_mae": "2.2565515", "Close_mse": "7.0540624"

and the model will be underfitted. Meanwhile the learning rate:

$$10e^{-}4 \tag{2}$$

batch size of 128, validation and test size is set to 0.05 for less data models (ten years period of time of data) and 0.01 for more data models (more than ten years period of time of data) so that there are five hundreds sample of testing for both model type. With these parameters the Validation loss is very low and acceptable:

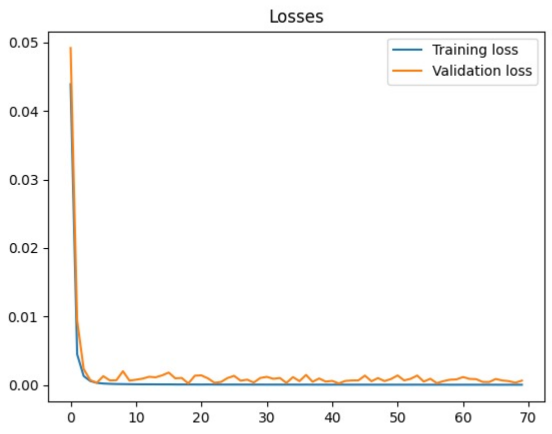


Figure 7: BiLSTM with the best learning rate

5 Models Architecture

Each model used in the project, tends to use a sequence of initial layers that allows to capture invariant features by exploiting the bottleneck technique:

- Linear + ReLU (linear expantion): This part involves a linear (dense) layer followed by a Rectified Linear Unit (ReLU) activation. This is a typical component in many neural network architectures and is used to introduce non-linearity and expand the dimensionality of the data.

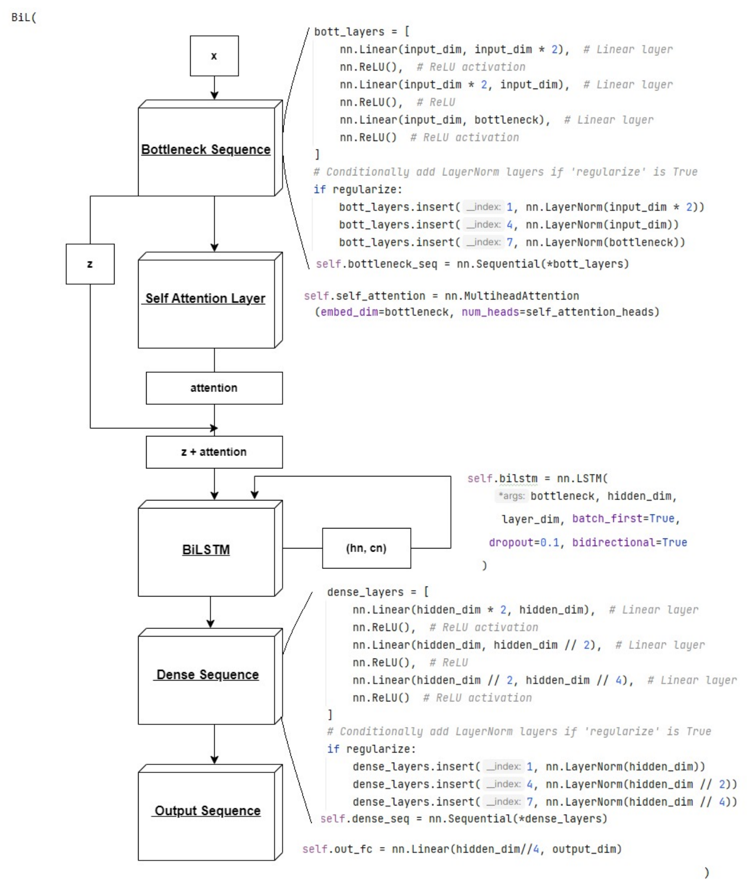
- Linear + ReLU (bottleneck layer): In this part, there's another linear layer followed by a ReLU activation, which is the "bottleneck layer". This combination can be used for dimensionality reduction, often seen in architectures like autoencoders or certain types of neural networks designed to capture lower-dimensional representations.

This architecture can be effective in learning hierarchical features, where the initial linear + ReLU layers capture complex patterns in the data and the subsequent linear + ReLU layers reduce the dimensionality while preserving important information.

All models use a "regularize" parameter to define whether or not to add normalization layers that help in regularizing architectures. Some models like ToM and BiLPET use *PositionalEncoding* class to sum linear embedding, generated from a linear layer sequence, with positional encoding embedding generated from the latter class. All classes present an *init_weigths* functions even tough normalization layers are used in the model architecture. As explained before, this project include three kind of models: a BiLSTM, and Encoder Based Transformer Model and a Hybrid model of the two.

5.1 BiLSTM

The BiLSTM (Bi direction Long Short Term Memory) model is used to predict the t(n+1) value, in this case, stock by having information (long-term dependencies) from the past stocks. It is a Recursive Neural Network (RNN) model and its usefulness is on the fact that classic RNN tend to loose information during the training: if the object to predict is a word in a long phrase the first words tend to loose importance. The BiLSTM fix this lack. In this specific scenario PyTorch [4] is used to build the main Network. However the main PyTorch "nn" module is adjusted to accomplish the main goal. The "bottleneck" value (used to define the layers for the invariant feature extraction) is given as input of the function. The ReLU was choose as activation function (this is used to every model). If the regularization term is set True, some normalization layers will be added. This will help the model to be more robust and helps to generalize data. Now the BiLSTM is created thank to PyTorch [4]. *Dense_layers* follow the same structure of the bottom layers: created, with *hidden_dim* dimension given as an input, and the if reg term is set True is regularized. In the forward function the bottleneck, self attention and the first cell state for the BiLSTM are set. In the end the output is a PyTorch[4] tensor of shape [batch_size, output_dim], output_dim is set to four(Open, Close, High, Low). The *init_weights* function simply initialize the weights for the model. BiL is a model designed to make BiLSTM work correctly by paying attention to multiple parts of the input: The input of all models is a concatenation of lags of Open, High, Low, Close + any Embed and year, if c_e = True ; therefore a multi head self attention mechanism was chosen so that the input data to the bilstm had first paid attention to the different parts of the input, given that the four stock values could have dependencies on each other and with other features.



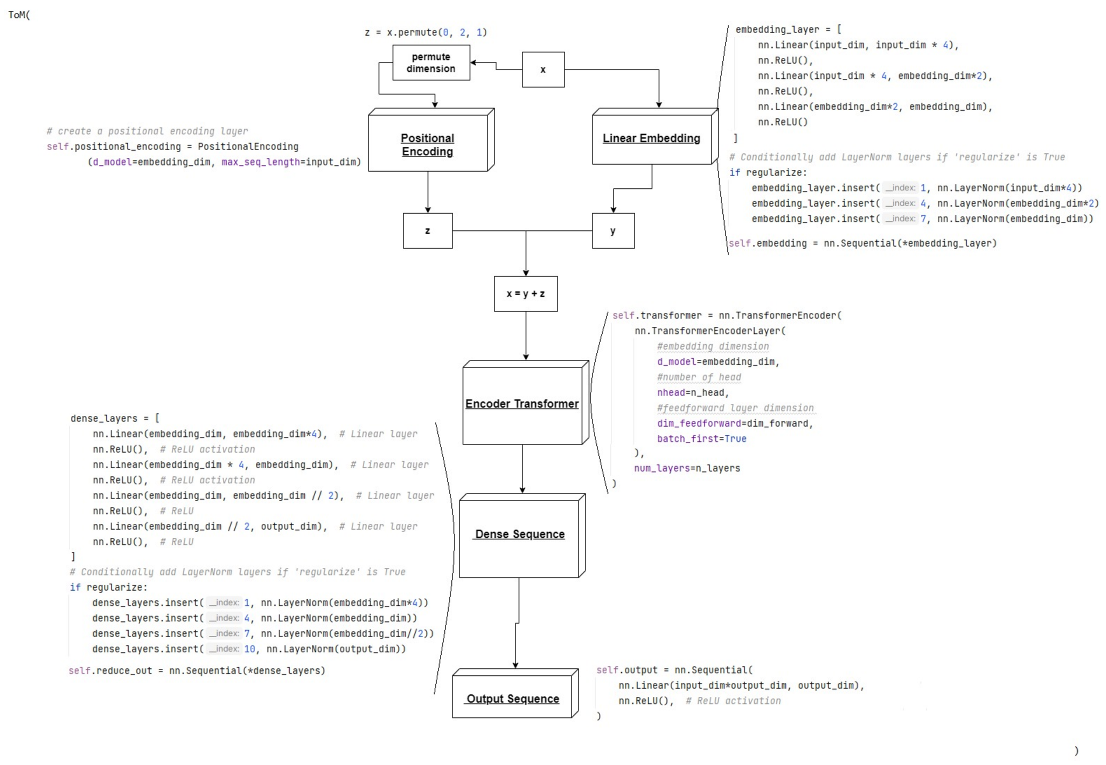
```
BiL(
    bott_layers = [
        nn.Linear(input_dim, input_dim * 2),  # Linear layer
        nn.ReLU(),  # ReLU activation
        nn.Linear(input_dim * 2, input_dim),  # Linear layer
        nn.ReLU(),  # ReLU
        nn.Linear(input_dim, bottleneck),  # Linear layer
        nn.ReLU()  # ReLU activation
    ]
    # Conditionally add LayerNorm layers if 'regularize' is True
    if regularize:
        bott_layers.insert( __index: 1, nn.LayerNorm(input_dim * 2))
        bott_layers.insert( __index: 4, nn.LayerNorm(input_dim))
        bott_layers.insert( __index: 7, nn.LayerNorm(bottleneck))
    self.bottleneck_seq = nn.Sequential(*bott_layers)

    self.self_attention = nn.MultiheadAttention
     (embed_dim=bottleneck, num_heads=self_attention_heads)

    self.bilstm = nn.LSTM(
        *args: bottleneck, hidden_dim,
         layer_dim, batch_first=True,
       dropout=0.1, bidirectional=True
      )
    dense_layers = [
        nn.Linear(hidden_dim * 2, hidden_dim),  # Linear layer
        nn.ReLU(),  # ReLU activation
        nn.Linear(hidden_dim, hidden_dim // 2),  # Linear layer
        nn.ReLU(),  # ReLU
        nn.Linear(hidden_dim // 2, hidden_dim // 4),  # Linear layer
        nn.ReLU()  # ReLU activation
    ]
    # Conditionally add LayerNorm layers if 'regularize' is True
    if regularize:
        dense_layers.insert( __index: 1, nn.LayerNorm(hidden_dim))
        dense_layers.insert( __index: 4, nn.LayerNorm(hidden_dim // 2))
        dense_layers.insert( __index: 7, nn.LayerNorm(hidden_dim // 4))
    self.dense_seq = nn.Sequential(*dense_layers)

    self.out_fc = nn.Linear(hidden_dim//4, output_dim)
)
```

Figure 8: BiL Architecture

5.2 Encoder Based Transformer

The Transformer model is probably the most known model in the deep learning family thanks to Generative Pretrained Transformer (GPT) used in ChatGPT chat-bot. Despite so, it is not the only one and also its structure is different from the classic Transformer model. As a matter of fact Transformer are composed of two main part: an Encoder and a Decoder. The latter is used in the GPT model, meanwhile the first is the one used in this model. The Transformer model *ToM.py (Transformer only Model)* works based on some NN of the PyTorch [4] library. The init function initialize the embedding layer (input is given), that encode the input in a vector, the positional encoding(explained in a later function), the normalization layers for the embedding and dense layers, the dense layers and the transformer encoder. *init_weights* initialize the layers weights used in the *forward* function. Named before is the *PositionalEncoding* class here the encoding is done so that the model pays a lot of attention on the position in the tensor of the n(th) element. This approach has been chosen given the high importance of, in the finance world, the time position of each stocks. ToM is a surprisingly good model even on its own: it exploits the layer architecture of a transformer's Encoder Layer to generate embeddings after processing from simple Linear Layers + ReLU and any Normalization Layers. Probably what makes ToM a better model than BiL and EBiL is the presence of Positional Encoding and an architecture that remains simple overall.

There will be two ToM model: Big and Small. The differ for just the amount of data used in training.



```
ToM(
```

```
z = x.permute(0, 2, 1)
```

```
# create a positional encoding layer
self.positional_encoding = PositionalEncoding
      (d_model=embedding_dim, max_seq_length=input_dim)
```

```
embedding_layer = [
    nn.Linear(input_dim, input_dim * 4),
    nn.ReLU(),
    nn.Linear(input_dim * 4, embedding_dim*2),
    nn.ReLU(),
    nn.Linear(embedding_dim*2, embedding_dim),
    nn.ReLU()
]
# Conditionally add LayerNorm layers if 'regularize' is True
if regularize:
    embedding_layer.insert( __index: 1, nn.LayerNorm(input_dim*4))
    embedding_layer.insert( __index: 4, nn.LayerNorm(embedding_dim*2))
    embedding_layer.insert( __index: 7, nn.LayerNorm(embedding_dim))

self.embedding = nn.Sequential(*embedding_layer)
```

```
self.transformer = nn.TransformerEncoder(
    nn.TransformerEncoderLayer(
        #embedding dimension
        d_model=embedding_dim,
        #number of head
        nhead=n_head,
        #feedforward layer dimension
        dim_feedforward=dim_forward,
        batch_first=True
    ),
    num_layers=n_layers
)
```

```
dense_layers = [
    nn.Linear(embedding_dim, embedding_dim*4),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(embedding_dim * 4, embedding_dim),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(embedding_dim, embedding_dim // 2),  # Linear layer
    nn.ReLU(),  # ReLU
    nn.Linear(embedding_dim // 2, output_dim),  # Linear layer
    nn.ReLU(),  # ReLU
]
# Conditionally add LayerNorm layers if 'regularize' is True
if regularize:
    dense_layers.insert( __index: 1, nn.LayerNorm(embedding_dim*4))
    dense_layers.insert( __index: 4, nn.LayerNorm(embedding_dim))
    dense_layers.insert( __index: 7, nn.LayerNorm(embedding_dim//2))
    dense_layers.insert( __index: 10, nn.LayerNorm(output_dim))

self.reduce_out = nn.Sequential(*dense_layers)
```

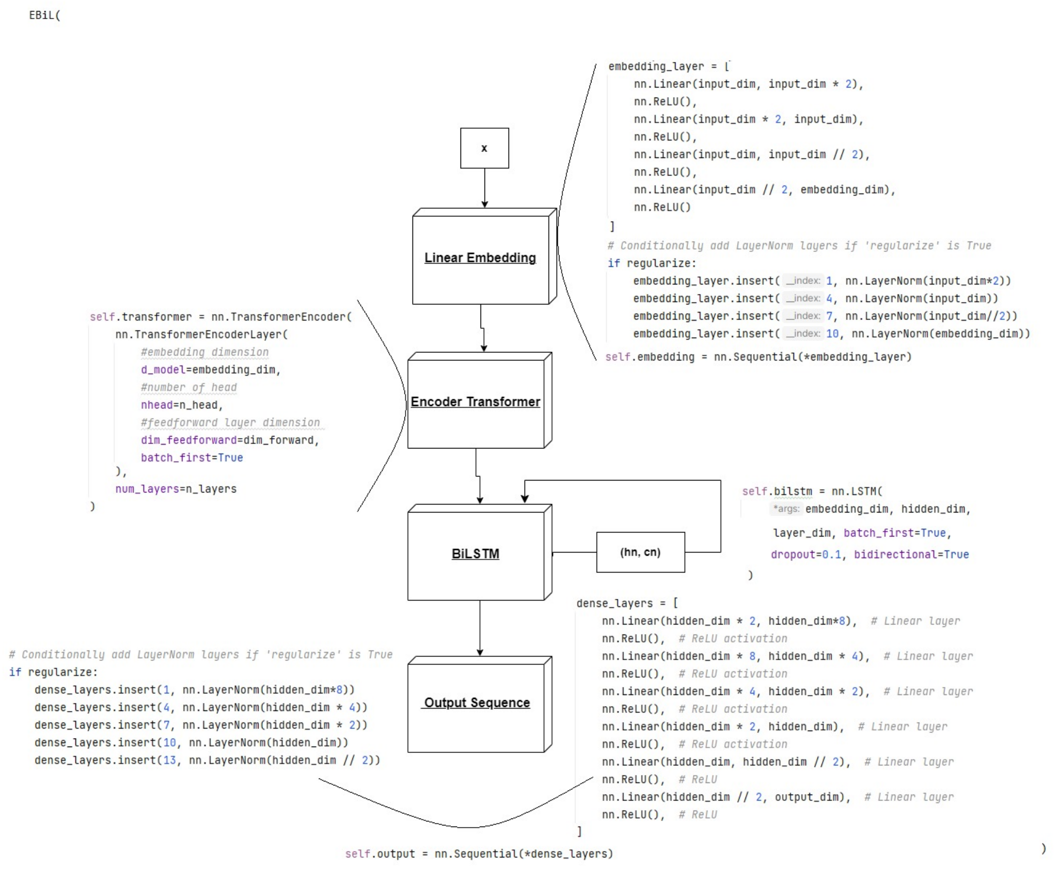
```
self.output = nn.Sequential(
    nn.Linear(input_dim*output_dim, output_dim),
    nn.ReLU(),  # ReLU activation
)
```

```
)
```

Figure 9: ToM (Transformer only Model) Architecture

5.3 Hybrid Model

The main structure of the Hybrid model is the same as the BiLSTM one. There is just one main difference: The transformer. Here the no of head, layers and the embedding dimension is also gives as input. Important to notice that the dense layer are deeper than BiL. Forward and the initialization weights function work as before.
EBiL takes the X input, embeds it in a Linear Embedding. The *embedding_layers* has input_dim and embedding_dim as hyperparameters setted by hand. Then the output goes straight into the transformer section. Here all the dimensions of the hyperparameters are also setted by hand (embedding_dim, n_head, dim_forward, n_layers). The output of the transformer is then used by the BiLSTM (as well as past information). Finally the overall output is normalized. The EBiL model combines the idea of predicting sequences with BiLSTM encoded by an Encoder Transformer. The transformer is used to create embeddings that will later be used in BiLSTM. So the transfomer is just Encoder Stack and is used by exploiting for its "stack-encoder architecture": Multi-Head Self-Attention + Skip Connection + Normalization Layer + FeedForward Layers.



```
EBiL(
```

```
embedding_layer = [
    nn.Linear(input_dim, input_dim * 2),
    nn.ReLU(),
    nn.Linear(input_dim * 2, input_dim),
    nn.ReLU(),
    nn.Linear(input_dim, input_dim // 2),
    nn.ReLU(),
    nn.Linear(input_dim // 2, embedding_dim),
    nn.ReLU()
]
# Conditionally add LayerNorm layers if 'regularize' is True
if regularize:
    embedding_layer.insert( _index: 1, nn.LayerNorm(input_dim*2))
    embedding_layer.insert( _index: 4, nn.LayerNorm(input_dim))
    embedding_layer.insert( _index: 7, nn.LayerNorm(input_dim//2))
    embedding_layer.insert( _index: 10, nn.LayerNorm(embedding_dim))
self.embedding = nn.Sequential(*embedding_layer)
```

```
self.transformer = nn.TransformerEncoder(
    nn.TransformerEncoderLayer(
        #embedding dimension
        d_model=embedding_dim,
        #number of head
        nhead=n_head,
        #feedforward layer dimension
        dim_feedforward=dim_forward,
        batch_first=True
    ),
    num_layers=n_layers
)
```

```
self.bilstm = nn.LSTM(
    *args: embedding_dim, hidden_dim,
    layer_dim, batch_first=True,
    dropout=0.1, bidirectional=True
)
```

```
dense_layers = [
    nn.Linear(hidden_dim * 2, hidden_dim*8),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim * 8, hidden_dim * 4),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim * 4, hidden_dim * 2),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim * 2, hidden_dim),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim, hidden_dim // 2),  # Linear layer
    nn.ReLU(),  # ReLU
    nn.Linear(hidden_dim // 2, output_dim),  # Linear layer
    nn.ReLU(),  # ReLU
]
```

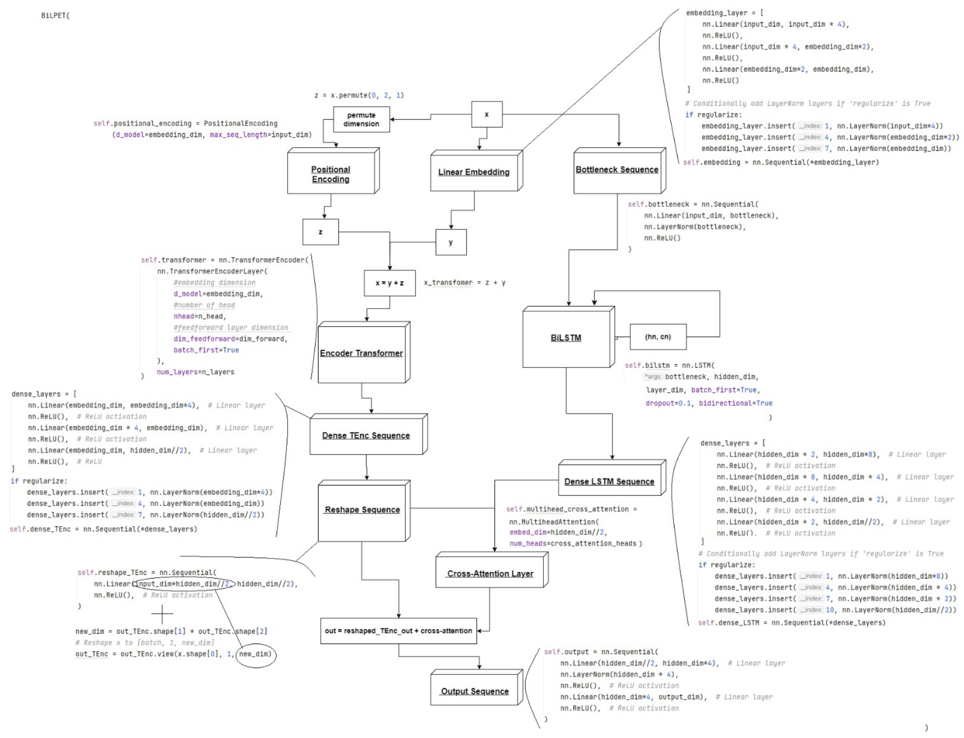
```
# Conditionally add LayerNorm layers if 'regularize' is True
if regularize:
    dense_layers.insert(1, nn.LayerNorm(hidden_dim*8))
    dense_layers.insert(4, nn.LayerNorm(hidden_dim * 4))
    dense_layers.insert(7, nn.LayerNorm(hidden_dim * 2))
    dense_layers.insert(10, nn.LayerNorm(hidden_dim))
    dense_layers.insert(13, nn.LayerNorm(hidden_dim // 2))
```

```
self.output = nn.Sequential(*dense_layers)
)
```

Figure 10: EBiL Architecture

BiLPET is another version. It takes input X, made an embedding, positional encoding. It sum them up and gives all to a Transformer that generates a *Z1*. Then the input X model takes another route (BiLSTM) and trough linear layers to catch invariant feature, then reduce dimensionality. They now go trough the real BiLSTM and generates a *Z2*. This is done as explained in the BiL model before. Then *Z2* is reshaped to match *Z1* dimension. By using the cross-attention mechanism in which the query is *Z2* and key and value are *Z1* an output *Z3* is obtained. *Z1* and *Z3* are summed and the output is send to other Linear Layers for a final reshape of [batch_size, output_size].

BiLPET is an architecture designed to make processing slower, but more precise, even if the results obtained are not worth its complexity. The BiLPET architecture aims to exploit the strengths of BiL and ToM by processing the 2 sequences independently, subsequently merging them with a cross-attention mechanism. It was decided to use the transformer output as a query to capture the BiLSTM information in the cross attention layer output and add it again with the transformer output to integrate its information. It was thought that with this mechanism it would be possible to use the values predictive capacity of BiL (key and value) with that of ToM trends (query + merging). It didn't turn out to be as good as expected, but it remains a good model that can be used in the bagging/ensemble phase.



```
BiLPET(
```

```
z = x.permute(0, 2, 1)
```

```
self.positional_encoding = PositionalEncoding
    (d_model=embedding_dim, max_seq_length=input_dim)
```

```
embedding_layer = [
    nn.Linear(input_dim, input_dim * 4),
    nn.ReLU(),
    nn.Linear(input_dim * 4, embedding_dim*2),
    nn.ReLU(),
    nn.Linear(embedding_dim*2, embedding_dim),
    nn.ReLU()
]
# Conditionally add LayerNorm layers if 'regularize' is True
if regularize:
    embedding_layer.insert( _index: 1, nn.LayerNorm(input_dim*4))
    embedding_layer.insert( _index: 4, nn.LayerNorm(embedding_dim*2))
    embedding_layer.insert( _index: 7, nn.LayerNorm(embedding_dim))
self.embedding = nn.Sequential(*embedding_layer)
```

```
self.bottleneck = nn.Sequential(
    nn.Linear(input_dim, bottleneck),
    nn.LayerNorm(bottleneck),
    nn.ReLU()
)
```

```
self.transformer = nn.TransformerEncoder(
    nn.TransformerEncoderLayer(
        #embedding dimension
        d_model=embedding_dim,
        #number of head
        nhead=n_head,
        #feedforward layer dimension
        dim_feedforward=dim_forward,
        batch_first=True
    ),
    num_layers=n_layers
)
```

```
x_transfomer = z + y
```

```
self.bilstm = nn.LSTM(
    *args: bottleneck, hidden_dim,
    layer_dim, batch_first=True,
    dropout=0.1, bidirectional=True
                                    )
```

```
dense_layers = [
    nn.Linear(embedding_dim, embedding_dim*4),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(embedding_dim * 4, embedding_dim),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(embedding_dim, hidden_dim//2),  # Linear layer
    nn.ReLU(),  # ReLU
]
if regularize:
    dense_layers.insert( _index: 1, nn.LayerNorm(embedding_dim*4))
    dense_layers.insert( _index: 4, nn.LayerNorm(embedding_dim))
    dense_layers.insert( _index: 7, nn.LayerNorm(hidden_dim//2))
self.dense_TEnc = nn.Sequential(*dense_layers)
```

```
dense_layers = [
    nn.Linear(hidden_dim * 2, hidden_dim*8),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim * 8, hidden_dim * 4),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim * 4, hidden_dim * 2),  # Linear layer
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim * 2, hidden_dim//2),  # Linear layer
    nn.ReLU().  # ReLU activation
]
# Conditionally add LayerNorm layers if 'regularize' is True
if regularize:
    dense_layers.insert( _index: 1, nn.LayerNorm(hidden_dim*8))
    dense_layers.insert( _index: 4, nn.LayerNorm(hidden_dim * 4))
    dense_layers.insert( _index: 7, nn.LayerNorm(hidden_dim * 2))
    dense_layers.insert( _index: 10, nn.LayerNorm(hidden_dim//2))
self.dense_LSTM = nn.Sequential(*dense_layers)
```

```
self.multihead_cross_attention =
 nn.MultiheadAttention(
 embed_dim=hidden_dim//2,
 num_heads=cross_attention_heads )
```

```
self.reshape_TEnc = nn.Sequential(
    nn.Linear(input_dim*hidden_dim//2, hidden_dim//2),
    nn.ReLU(),  # ReLU activation
)
```

```
new_dim = out_TEnc.shape[1] * out_TEnc.shape[2]
# Reshape x to [batch, 1, new_dim]
out_TEnc = out_TEnc.view(x.shape[0], 1, new_dim)
```

```
self.output = nn.Sequential(
    nn.Linear(hidden_dim//2, hidden_dim*4),  # Linear layer
    nn.LayerNorm(hidden_dim * 4),
    nn.ReLU(),  # ReLU activation
    nn.Linear(hidden_dim*4, output_dim),  # Linear layer
    nn.ReLU(),  # ReLU activation
)
                                                            )
```

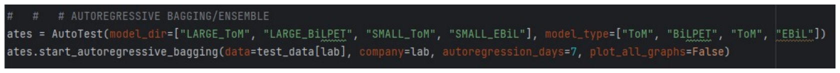
Figure 11: BiLPET Architecture

6 Testing

As already explained, two testing has been done. The first to evaluate the metrics(MAE, MSE) for next day prediction, these metrics are saved in the dir of each model and used as indicator for the quality of the models. The second to predict up to seventh day in an autoregressive way. In the following subsection the Autoregressive testing is explained.

6.1 AutoRegressive Testing

The testing fase is the last of the overall fases in the construction of a Deep Learning Architecture. It helps to understand if the main goal is achieved. In this specific case an Autoregressive Testing was done for a simply reason: it predicts the values up to day n(th) even if it might loose accuracy based on the errors that tent to sum up in the predictor. So that since the autoregressive testing uses past information to predict this stack of errors might highly influence the outcome. This is done in **AutoregressiveTest.py**. Summarizing very quickly how it works: it just load the directory from the chosen model and the chosen company, creates a pipeline x that is given as a new input to the testing. Since it is autoregressive this is mandatory: it erase the older data from the input and append the latest discovered information. At the end graphics, metrics and stock values are printed. This class takes models as input and use them for an ensembled/bagging prediction. To use this class in ensembled/bagging mode:
- models must be in the same dir
- must use an ordered list of 'model_dir' and 'model_type'
- all models must be trained on the company you are trying to predict
- must use 'start_autoregressive_bagging' instead of 'start_autoregression



```
#   #   # AUTOREGRESSIVE BAGGING/ENSEMBLE
ates = AutoTest(model_dir=["LARGE_ToM", "LARGE_BiLPET", "SMALL_ToM", "SMALL_EBiL"], model_type=["ToM", "BiLPET", "ToM", "EBiL"])
ates.start_autoregressive_bagging(data=test_data[lab], company=lab, autoregression_days=7, plot_all_graphs=False)
```

Figure 12: Starting autoregressive in ensembled/bagging

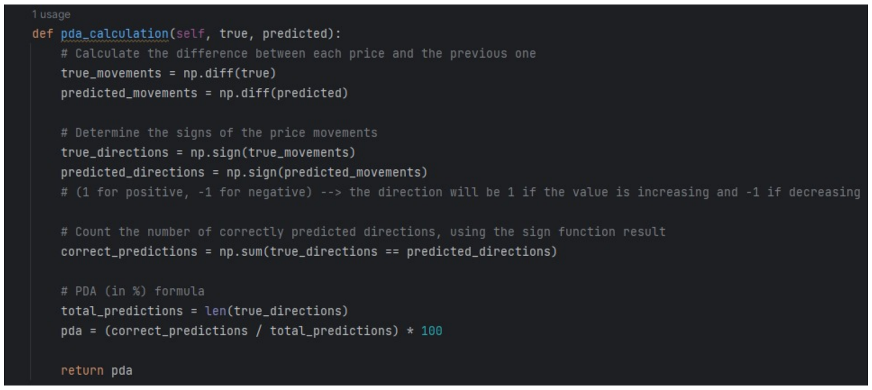
Bagging uses different subset of data. Meanwhile here this is not done, because different architecture are used with the same data. So it is better to say that within the trained models predictions are in an ensembled mode.

Run_tesitng.py is just a Python file to starts the testing of a model given the company name.

6.2 Predictions of each model

All the "SMALL" model uses data from just ten years of time. Meanwhile the "LARGE" model have forty years (or thirty it depends of the lifespan of the company) time of training data. They also differ of some Hyperparameters of the architecture, to see all the difference check each model directory in the **models** directory. One metric has been calculated as well: the **PDA** (Percentage of

Direction Accuracy). The metrics calculates the difference between a value and the previous one within a Numpy array, both for predicted and real values. Following the resulting arrays, the sign function is applied so as to obtain 1 (increase in the trend) if positive and -1 (decrease) if negative. At this point they are compared, taking the sign array of the real values as a reference, and the probability is then calculated.



```python
def pda_calculation(self, true, predicted):
    # Calculate the difference between each price and the previous one
    true_movements = np.diff(true)
    predicted_movements = np.diff(predicted)

    # Determine the signs of the price movements
    true_directions = np.sign(true_movements)
    predicted_directions = np.sign(predicted_movements)
    # (1 for positive, -1 for negative) --> the direction will be 1 if the value is increasing and -1 if decreasing

    # Count the number of correctly predicted directions, using the sign function result
    correct_predictions = np.sum(true_directions == predicted_directions)

    # PDA (in %) formula
    total_predictions = len(true_directions)
    pda = (correct_predictions / total_predictions) * 100

    return pda
```

Figure 13: Pda Python Code

MSE and MAE were used to understand whether the trained model was able to predict the values. These metrics were used both during single value testing and during autoregressive testing using the *run_testing.py* file which takes advantage of the *AutoregressiveTest.py* class. Here MAE (Mean Absolute Error):

$$\frac{\sum_{i=1}^{n} |x_i - y_i|}{n} \tag{3}$$

where n is the batch size, and i is the n(th) element, and the MSE (Mean Square Error):

$$\frac{\sum_{i=1}^{n} (x_i - y_i)^2}{n} \tag{4}$$

where n is the batch size, and i is the n(th) element. The MAE is used to understand the difference between predicted X and the target Y. Meanwhile the MSE does the same w.r.t their quadratic distance. In the autoregressive test it was decided to add a metric that would allow to understand the trend. The function for calculating the PDA (Percentage of Direction Accuracy) was then implemented, which explains, in percentage, the correct directions (increase or decrease) predicted by the model. In all the cases tasted these two metrics are low, near *0.01*. So the prediction are accurate. The PDA is as well accettable and ranges between 60% and 50% for each model, in the 25 lags model. Despite in ToMSmall, ToMLarge, EBiL Large and EBiL Small just one company is shown they work just fine for the other three. This was done to give the document a better readability. Some models have just the tend graphs, this was done in the purpose to give the document a better readability.

6.2.1 BiL Small

BiL Small model tend to have good prediction however it present some difficult into predict GOOGL and MSFT trend. Nonetheless is the lightest model of all.

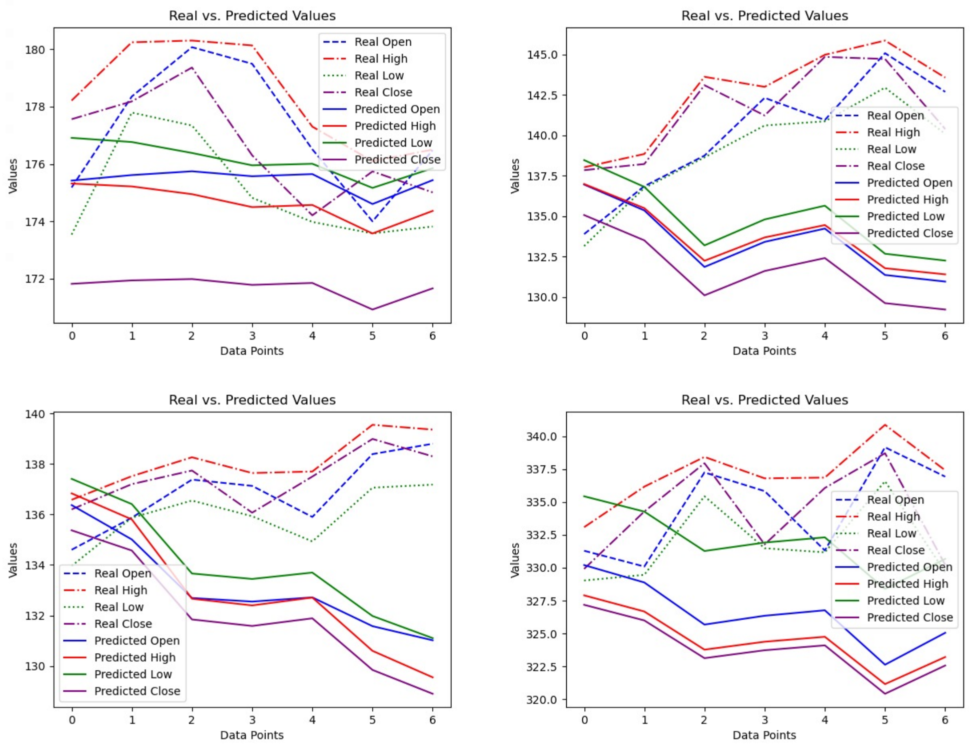


Figure 14: BiLSMALL prediction of, in order: APPLE, AMAZON, GOOGLE MICROSOFT

6.2.2 ToM Large

APPLE prediction for ToM Large.

```
DAY: 1 NORMALIZED PRED: [[0.97336984 0.9746916 0.9693675 0.96662694]]
PRED: [[177.76828 177.40526 177.3381 173.14436]] REAL: 175.179993
178.210007 173.539993 177.559998
DAY: 2 NORMALIZED PRED: [[0.9811715 0.9821949 0.97636163 0.97482675]]
PRED: [[179.19258 178.77045 178.61716 174.6126 ]] REAL: 178.350006
180.240005 177.789993 178.179993
DAY: 3 NORMALIZED PRED: [[0.97487354 0.97770405 0.97268873 0.9680376
]] PRED: [[178.0428 177.95335 177.94548 173.39696]] REAL: 180.070007
180.300003 177.339996 179.360001
DAY: 4 NORMALIZED PRED: [[0.9697877 0.9725373 0.9676919 0.963438 ]]
PRED: [[177.1143 177.01329 177.0317 172.57336]] REAL: 179.490005
180.130005 174.820007 176.300003
DAY: 5 NORMALIZED PRED: [[0.96416044 0.96750927 0.9629551
0.95780903]] PRED: [[176.08696 176.09846 176.16545 171.56548]] REAL:
176.509995 177.300003 173.979996 174.210007
DAY: 6 NORMALIZED PRED: [[0.96210575 0.96577746 0.96200293 0.9554877
]] PRED: [[175.71185 175.78337 175.99132 171.14984]] REAL: 174.0
176.100006 173.580002 175.740005
DAY: 7 NORMALIZED PRED: [[0.95723873 0.96069014 0.95713836
0.95100087]] PRED: [[174.8233 174.85776 175.10172 170.34644]] REAL:
176.479996 176.5 173.820007 175.009995
——————— METRICS ———————
’Open_mae’: ’0.009096849025773965’, ’Open_mse’: ’9.865311312437114e-05’,
’Open_pda[%]’: ’66.66666666666666’, ’High_mae’: ’0.008556817114518231’,
’High_mse’: ’9.609882324728557e-05’, ’High_pda[%]’: ’66.66666666666666’,
’Low_mae’: ’0.010406001548681956’, ’Low_mse’: ’0.00013926243017826386’,
’Low_pda[%]’: ’83.33333333333334’, ’Close_mae’: ’0.0235928301996045’,
’Close_mse’: ’0.0005858584934405586’, ’Close_pda[%]’: ’66.66666666666666’
```

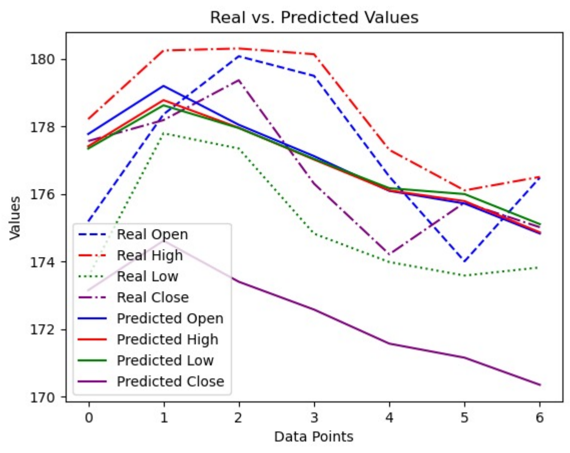


Figure 15: ToM large prediction of APPLE

6.2.3 ToM Small

ToM seems to better catch trends than the other architecture. MAE and MSE are low. AMAZON prediction for ToM Small

```
DAY: 1 NORMALIZED PRED: [[0.73980117 0.74583185 0.7425322 0.7398391
]] PRED: [[142.1908 142.79689 143.82037 140.44577]] REAL: 133.899994
138.029999 133.160004 137.850006
DAY: 2 NORMALIZED PRED: [[0.7529791 0.7569049 0.7573169 0.7589823]]
PRED: [[144.47032 144.70393 146.39487 143.71236]] REAL: 136.860001
138.850006 136.75 138.229996
DAY: 3 NORMALIZED PRED: [[0.7497868 0.75408775 0.7522552 0.7489307 ]]
PRED: [[143.9181 144.21875 145.51346 141.99715]] REAL: 138.75 143.619995
138.639999 143.100006
DAY: 4 NORMALIZED PRED: [[0.7406241 0.7454625 0.744094 0.7434573]]
PRED: [[142.33316 142.73328 144.09233 141.06317]] REAL: 142.320007 143.0
140.610001 141.229996
DAY: 5 NORMALIZED PRED: [[0.74394363 0.7496186 0.74799854
0.74591756]] PRED: [[142.90736 143.44905 144.77223 141.48299]] REAL:
140.949997 144.979996 140.869995 144.850006
DAY: 6 NORMALIZED PRED: [[0.7383426 0.74235576 0.7411328 0.7412587 ]]
PRED: [[141.93849 142.19823 143.57669 140.688 ]] REAL: 145.080002
145.860001 142.949997 144.720001
DAY: 7 NORMALIZED PRED: [[0.7407616 0.74670583 0.7436781 0.7419178 ]]
PRED: [[142.35693 142.94742 144.01991 140.80048]] REAL: 142.690002
143.570007 140.089996 140.389999
——————— METRICS ———————
```

'Open_mae': '0.021897108180070988', 'Open_mse': '0.0007981459467235416', 'Open_pda[%]': '16.666666666666664', 'High_mae': '0.01435151046983174', 'High_mse': '0.00035429937522751847', 'High_pda[%]': '50.0', 'Low_mae': '0.03209359318910299', 'Low_mse': '0.0013997357138840017', 'Low_pda[%]': '33.333333333333333', 'Close_mae': '0.014363875594350848', 'Close_mse': '0.00032282976967862206', 'Close_pda[%]': '66.66666666666666'

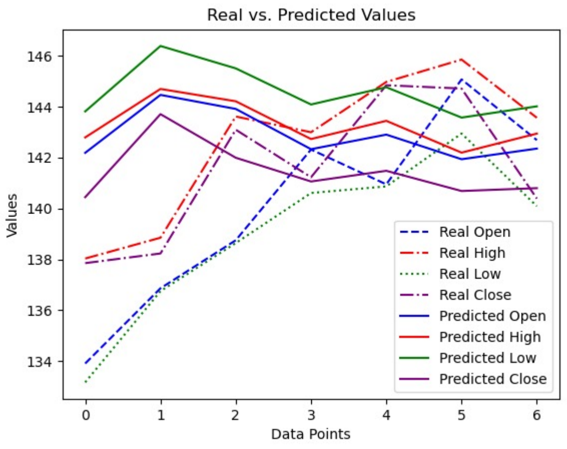


Figure 16: ToM small prediction of AMAZON

6.2.4 BilPET Large

The BiLPET Large model is trained with a bigger amount of data for each companies: in some cases even forty years of stocks data. It is not an amazing tool for autonomous prediction. Despite so it can be a good model for auto ensemble/bagging.

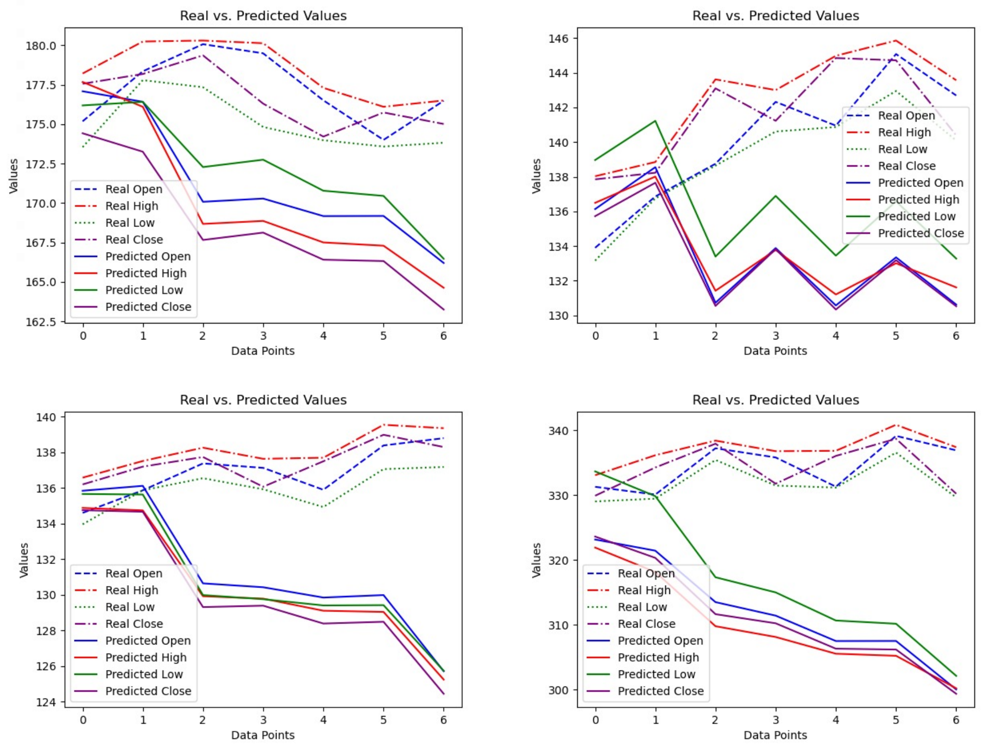


Figure 17: BiLPET large prediction of, in order: APPLE; AMAZON; GOOGLE; MICROSOFT

6.2.5 BiLPET Small

Works the same as BilPET Large, the main difference is the the architecture is slided modified, like some parameters and data used for training: ten years of stocks instead of forty.

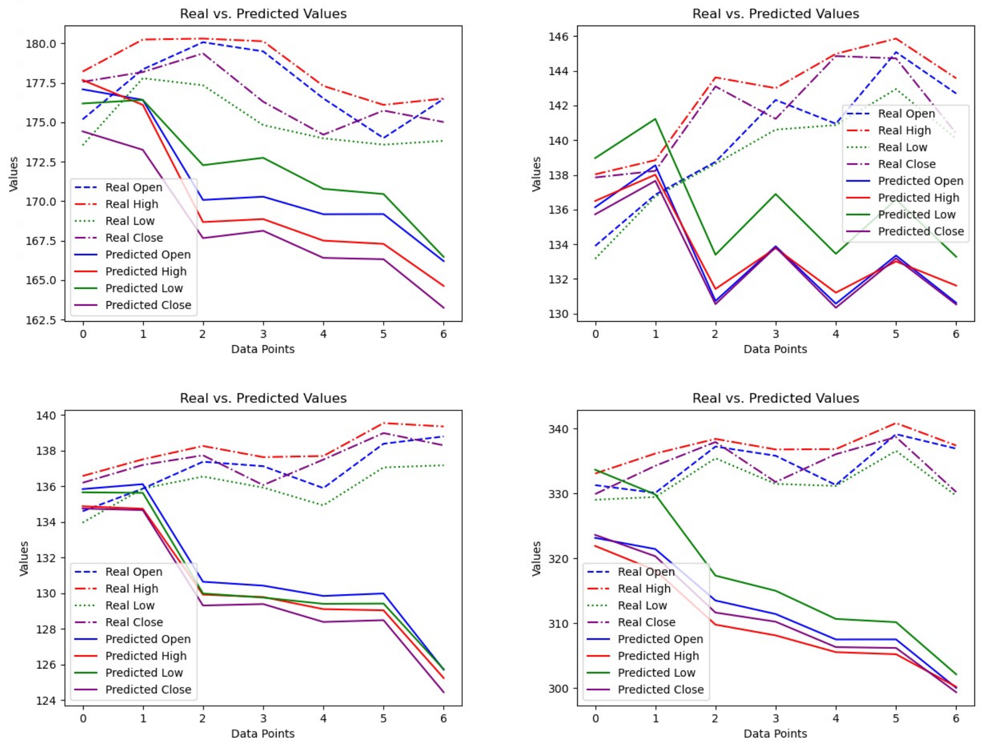


Figure 18: BiLPET small prediction of, in order: APPLE; AMAZON; GOOGLE; MICROSOFT

6.2.6 EBiL Large

MICROSOFT prediction for EBiL Large:

```
DAY: 1 NORMALIZED PRED: [[0.95201385 0.95582515 0.9594333 0.957908 ]]
PRED: [[328.08755 327.95728 335.48895 327.79993]] REAL: 331.290009
333.079987 329.029999 329.910004
DAY: 2 NORMALIZED PRED: [[0.9462551 0.9506241 0.95249677 0.95096165]]
PRED: [[326.10352 326.17325 333.0641 325.42352]] REAL: 330.089996
336.160004 329.459991 334.269989
DAY: 3 NORMALIZED PRED: [[0.9528222 0.95652616 0.960323 0.9588732 ]]
PRED: [[328.36606 328.19775 335.79996 328.13013]] REAL: 337.23999
338.420013 335.429993 337.940002
DAY: 4 NORMALIZED PRED: [[0.94351614 0.94789314 0.94936395 0.9478561
]] PRED: [[325.15988 325.23648 331.96893 324.36108]] REAL: 335.820007
336.790009 331.480011 331.769989
DAY: 5 NORMALIZED PRED: [[0.9619391 0.96309924 0.96797514 0.9668949
]] PRED: [[331.50705 330.45242 338.47495 330.87445]] REAL: 331.309998
336.850006 331.170013 336.059998
DAY: 6 NORMALIZED PRED: [[0.9567492 0.95978737 0.96415055 0.9630684
]] PRED: [[329.719 329.3164 337.13797 329.56537]] REAL: 339.149994
340.859985 336.570007 338.700012
DAY: 7 NORMALIZED PRED: [[0.9615555 0.96283644 0.9676845 0.96658665]]
PRED: [[331.37488 330.36227 338.37335 330.76898]] REAL: 336.920013
337.399994 329.649994 330.220001
———————— METRICS ————————
'Open_mae': '0.017372240497245395', 'Open_mse': '0.00040711428130538006',
'Open_pda[%]': '50.0', 'High_mae': '0.025764754944762176', 'High_mse':
'0.0007135177753327555', 'High_pda[%]': '50.0', 'Low_mae':
'0.011245559900978737', 'Low_mse': '0.0002161107257273285', 'Low_pda[%]':
'33.33333333333333', 'Close_mae': '0.017974361280020994', 'Close_mse':
'0.00042045790230951135', 'Close_pda[%]':'50.0'
```

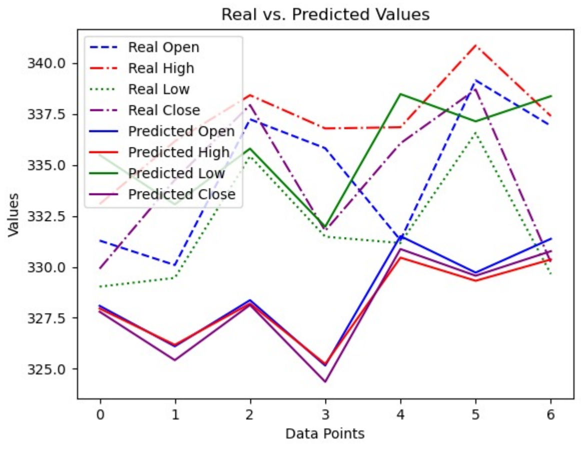


Figure 19: EBiL Large prediction of MICROSOFT

6.2.7 EbiLSmall

GOOGLE preditction for EBiLSmall:

DAY: 1 NORMALIZED PRED: [[0.9559453 0.9588756 0.95952463 0.96182156]] PRED: [[330.89166 330.3573 336.86523 330.38376]] REAL: 331.290009 333.079987 329.029999 329.910004
DAY: 2 NORMALIZED PRED: [[0.95187515 0.95510286 0.9555379 0.95784944]] PRED: [[329.62332 329.18738 335.604 329.15442]] REAL: 330.089996 336.160004 329.459991 334.269989
DAY: 3 NORMALIZED PRED: [[0.9559388 0.95886916 0.95951784 0.9618148]] PRED: [[330.88968 330.35532 336.8631 330.3817]] REAL: 337.23999 338.420013 335.429993 337.940002
DAY: 4 NORMALIZED PRED: [[0.9545529 0.95758396 0.9581624 0.96048725]] PRED: [[330.4578 329.9568 336.4343 329.9708]] REAL: 335.820007 336.790009 331.480011 331.769989
DAY: 5 NORMALIZED PRED: [[0.9574832 0.96029896 0.9610275 0.96328086]] PRED: [[331.3709 330.7987 337.3407 330.83545]] REAL: 331.309998 336.850006 331.170013 336.059998
DAY: 6 NORMALIZED PRED: [[0.9520742 0.95528775 0.9557339 0.95804614]] PRED: [[329.68536 329.24472 335.66602 329.2153]] REAL: 339.149994 340.859985 336.570007 338.700012
DAY: 7 NORMALIZED PRED: [[0.96537566 0.9673768 0.96871114 0.9706895]] PRED: [[333.83038 332.99353 339.77148 333.12842]] REAL: 336.920013 337.399994 329.649994 330.220001
———————— METRICS ————————
'Open_mae': '0.011549199710249005', 'Open_mse': '0.0002480108964821599', 'Open_pda[%]': '50.0', 'High_mae': '0.0214982694065786', 'High_mse':

'0.0005328995000239896', 'High_pda[%]': '50.0', 'Low_mae': '0.016962058717190982', 'Low_mse': '0.0003812208386131988', 'Low_pda[%]': '33.33333333333333', 'Close_mae': '0.015030932948803057', 'Close_mse': '0.0003168729340920084', 'Close_pda[%]':'50.0'

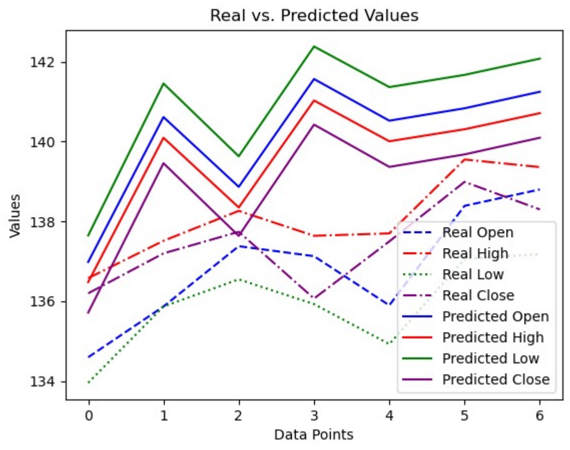


Figure 20: EBiL Small prediction of GOOGLE

6.3 Other testing

Moreover other testing has been done for all four model. In this new scenario the testing was done to get bagging metrics. Bagging metrics are the avg of predictions of all models used in the ensemble/bagging mode.

6.3.1 APPL

——————- BAGGING METRICS ——————-
'Open_mae': 0.011410155437480727, 'Open_mse': 0.00017099668127915557, 'Open_pda[%]': 58.33333333333332, 'High_mae': 0.011076597155227148, 'High_mse': 0.00018352389172299216, 'High_pda[%]': 54.166666666666664, 'Low_mae': 0.015517676680672863, 'Low_mse': 0.00034075155455498725, 'Low_pda[%]': 62.5, 'Close_mae': 0.01538349811515627, 'Close_mse': 0.00031936829764784826, 'Close_pda[%]': 45.83333333333333

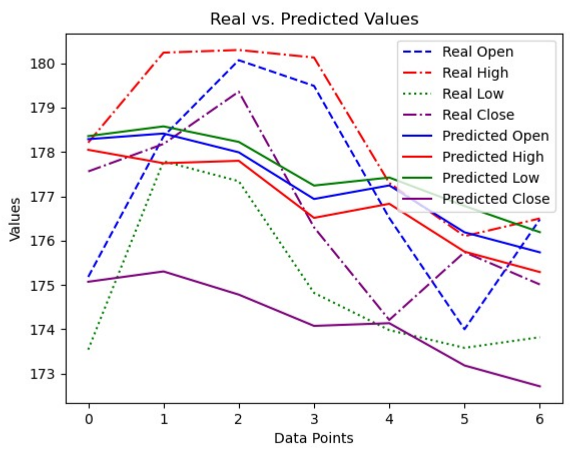


Figure 21: Bagging metrics for: APPL

6.3.2 AMZN

———————- BAGGING METRICS ———————-
'Open_mae': 0.021219748138402286, 'Open_mse': 0.0006581103076004558,
'Open_pda[%]': 41.666666666666664, 'High_mae': 0.01992241542156584,
'High_mse': 0.0005847418043104655, 'High_pda[%]': 49.99999999999999,
'Low_mae': 0.027480459269424187, 'Low_mse': 0.001074038189328428,
'Low_pda[%]': 58.333333333333336, 'Close_mae': 0.0235081747832747,
'Close_mse': 0.0007970702091233631, 'Close_pda[%]': 41.666666666666664

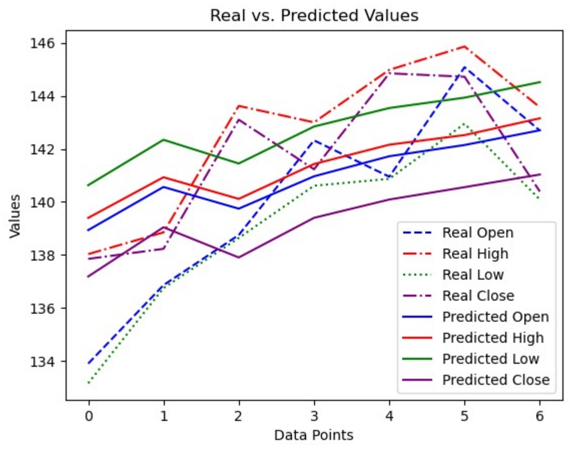


Figure 22: Bagging metrics for: AMZN

6.3.3 GOOGL

—————————- BAGGING METRICS —————————-
'Open_mae': 0.027984070773201246, 'Open_mse': 0.0011849825077570974, 'Open_pda[%]': 62.49999999999999, 'High_mae': 0.019544155288388806, 'High_mse': 0.000674443128955131, 'High_pda[%]': 49.999999999999986, 'Low_mae': 0.03502007956704118, 'Low_mse': 0.0016904009271768022, 'Low_pda[%]': 49.99999999999999, 'Close_mae': 0.01998350540839892, 'Close_mse': 0.0006521369648126105, 'Close_pda[%]': 54.16666666666666

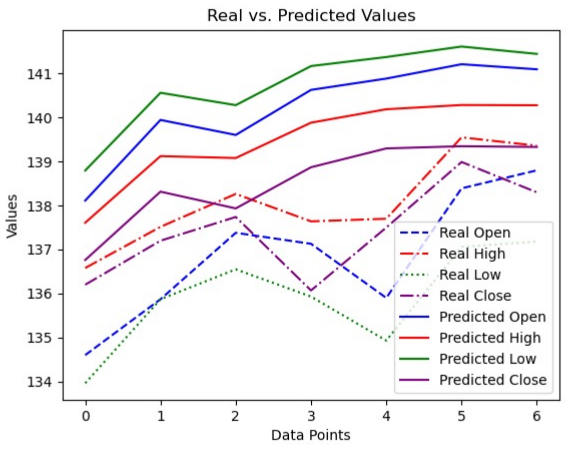


Figure 23: Bagging metrics for: GOOGL

6.3.4 MSFT

—————– BAGGING METRICS —————– 'Open_mae': 0.018578807353107098, 'Open_mse': 0.0005628310580641693, 'Open_pda[%]': 50.0, 'High_mae': 0.02655762552572118, 'High_mse': 0.0009198759319730573, 'High_pda[%]': 41.666666666666664, 'Low_mae': 0.013755260415948631, 'Low_mse': 0.0002640726201597218, 'Low_pda[%]': 49.999999999999986, 'Close_mae': 0.020623757875785655, 'Close_mse': 0.0006154715914384036, 'Close_pda[%]': 41.666666666666664

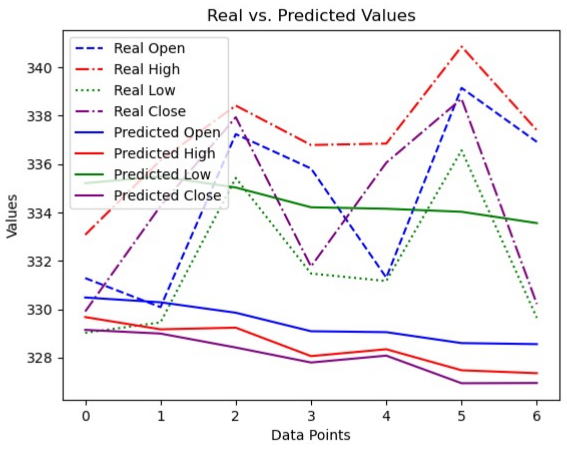


Figure 24: Bagging metrics for: MSFT

6.3.5 Forty five lags model

These models are ToM and BiL model for Small data and 45 lags. TOM: APPLE

'Open_mae': '0.06889278261180719', 'Open_mse': '0.005001381345799904', 'Open_pda[%]': '66.66666666666666', 'High_mae': '0.07755598219841864', 'High_mse': '0.006269646692483468', 'High_pda[%]': '50.0', 'Low_mae': '0.051448704364316776', 'Low_mse': '0.002873785328493597', 'Low_pda[%]': '50.0', 'Close_mae': '0.08180776535035715', 'Close_mse': '0.006979432221419621', 'Close_pda[%]':'50.0'

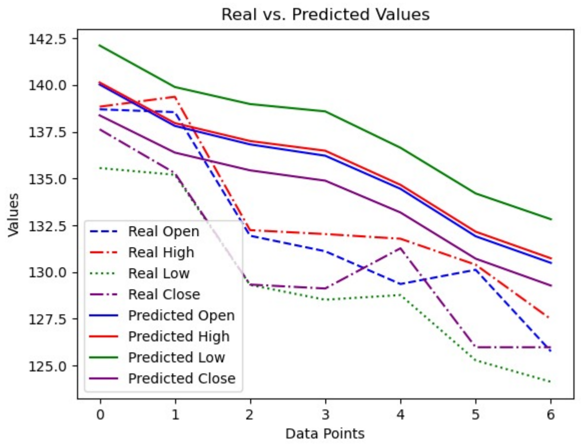


Figure 25: APPL ToM 45

AMZN

'Open_mae': '0.019543205576430 43', 'Open_mse': '0.000495183210997717', 'Open_pda[%]': '83.33333333333334', 'High_mae': '0.01643925066728231', 'High_mse': '0.0003284375851234289', 'High_pda[%]': '83.33333333333334', 'Low_mae': '0.046328429306219006', 'Low_mse': '0.002247226982278156', 'Low_pda[%]': '83.33333333333334', 'Close_mae': '0.019807144062940058', 'Close_mse': '0.0005358564532204659', 'Close_pda[%]': '66.66666666666666'

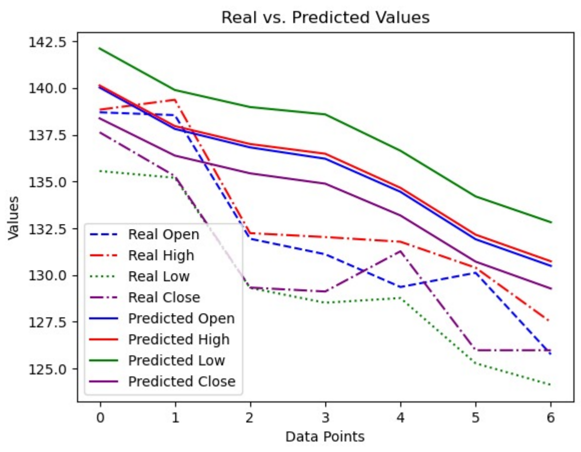


Figure 26: AMZN ToM 45

GOOGL

'Open_mae': '0.034629993523151166', 'Open_mse': '0.0013309986000032204', 'Open_pda[%]': '66.66666666666666', 'High_mae': '0.04730174275989033', 'High_mse': '0.0023703251576253533', 'High_pda[%]': '83.33333333333334', 'Low_mae': '0.013241779587836535', 'Low_mse': '0.0002880428918219353', 'Low_pda[%]': '83.33333333333334', 'Close_mae': '0.04200288359151753', 'Close_mse': '0.002080361960955128', 'Close_pda[%]': '66.66666666666666'

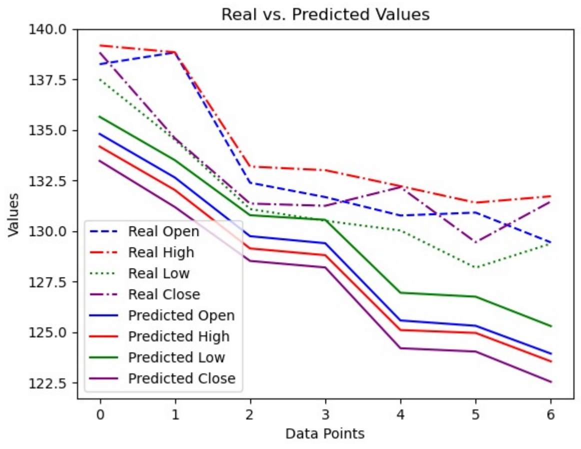


Figure 27: GOOGL ToM 45

MSFT

'Open_mae': '0.04894322020689552', 'Open_mse': '0.002564348652219678', 'Open_pda[%]': '50.0', 'High_mae': '0.05608686027157672', 'High_mse': '0.0032589941284066858', 'High_pda[%]': '66.66666666666666', 'Low_mae': '0.015061522230576345', 'Low_mse': '0.00040014976329067414', 'Low_pda[%]': '66.66666666666666', 'Close_mae': '0.04776453583670283', 'Close_mse': '0.002465966964339725', 'Close_pda[%]':'50.0'

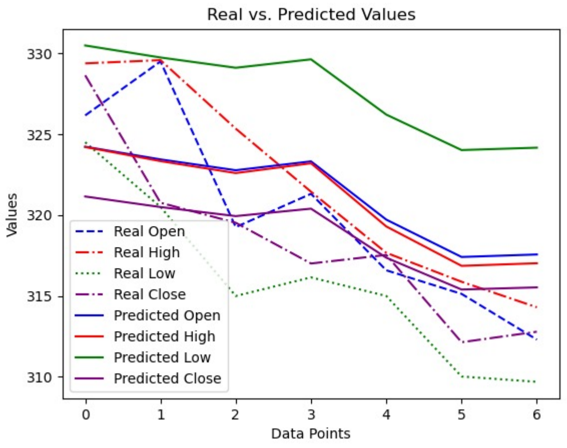


Figure 28: MSFT ToM 45

Bil APPL:

'Open_mae': '0.017909668709851445', 'Open_mse': '0.0003979265882084788', 'Open_pda[%]': '66.66666666666666', 'High_mae': '0.025308943289051485', 'High_mse': '0.0007030733740760431', 'High_pda[%]': '83.33333333333334', 'Low_mae': '0.007249409071767188', 'Low_mse': '9.720940483495835e-05', 'Low_pda[%]': '50.0', 'Close_mae': '0.03689348568509138', 'Close_mse': '0.0015224779988548894', 'Close_pda[%]':'50.0'

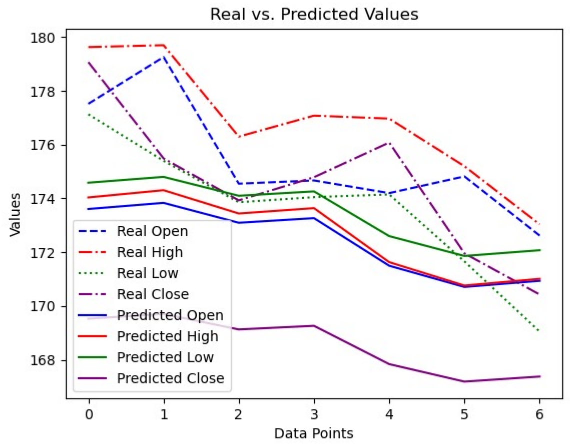


Figure 29: APPL BiL 45

AMZN

'Open_mae': '0.02978692593010663', 'Open_mse': '0.001166405931886773', 'Open_pda[%]': '50.0', 'High_mae': '0.033248624451713', 'High_mse': '0.0013594876209080367', 'High_pda[%]': '50.0', 'Low_mae': '0.061724819948657625', 'Low_mse': '0.004019612577141194', 'Low_pda[%]': '83.33333333333334', 'Close_mae': '0.022372450480546955', 'Close_mse': '0.0007113271342293227', 'Close_pda[%]': '66.66666666666666'

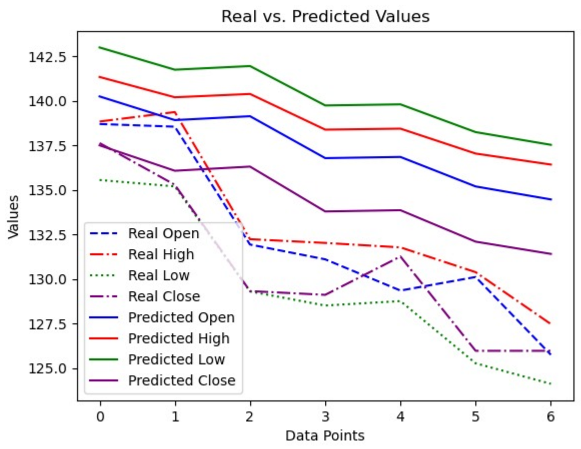


Figure 30: AMZN BiL 45

GOOGL

'Open_mae': '0.023003285473293493', 'Open_mse': '0.0005872467275851492', 'Open_pda[%]': '33.333333333333333', 'High_mae': '0.014314663282895117', 'High_mse': '0.0003287389582955435', 'High_pda[%]': '83.33333333333334', 'Low_mae': '0.033373437261232465', 'Low_mse': '0.0013658829960066787', 'Low_pda[%]': '83.33333333333334', 'Close_mae': '0.01626256071055635', 'Close_mse': '0.0003740497046302841', 'Close_pda[%]': '66.66666666666666'

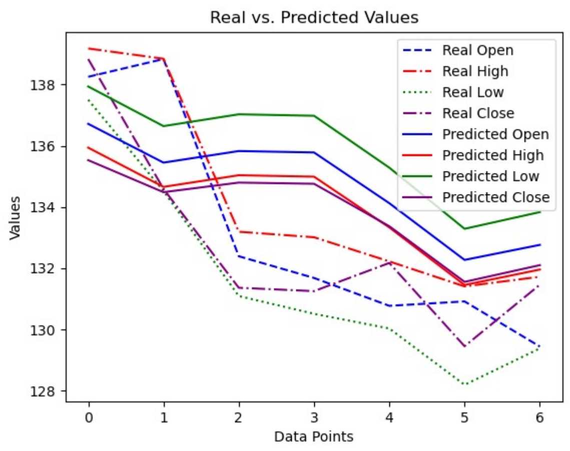


Figure 31: GOOGL BiL 45

MSFT

'Open_mae': '0.011160300354362948', 'Open_mse': '0.00014828093721926125', 'Open_pda[%]': '66.66666666666666', 'High_mae': '0.009860209187270057', 'High_mse': '0.00013163104552757113', 'High_pda[%]': '50.0', 'Low_mae': '0.03755893126335937', 'Low_mse': '0.001499025566195913', 'Low_pda[%]': '83.33333333333334', 'Close_mae': '0.008240018769672572', 'Close_mse': '0.00013000197751364498', 'Close_pda[%]': '66.66666666666666'

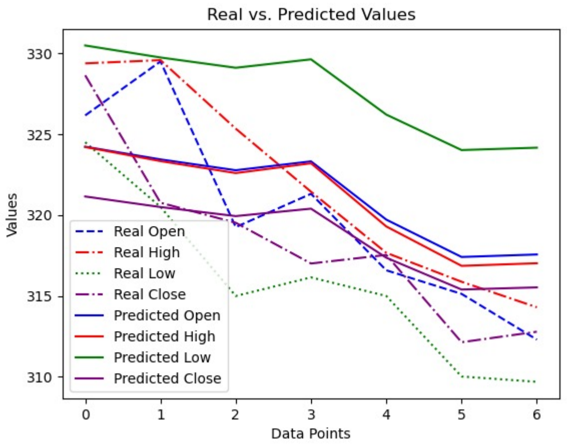


Figure 32: MSFT BiL 45

Ensamble/Bagging for the models:

ENSEMBLE "45LAG_SMALL_ToM" + "45LAG_SMALL_BiL" +"45LAG_SMALL_ToM"

APPLE:

'Open_mae': 0.051898411311155274, 'Open_mse': 0.003466896426602763, 'Open_pda[%]': 66.66666666666666, 'High_mae': 0.06014030256196292, 'High_mse': 0.004414122253014327, 'High_pda[%]': 61.111111111111114, 'Low_mae': 0.03671560593346691, 'Low_mse': 0.0019482600206073841, 'Low_pda[%]': 50.0, 'Close_mae': 0.06683633879526857, 'Close_mse': 0.005160447480564711, 'Close_pda[%]':50.0

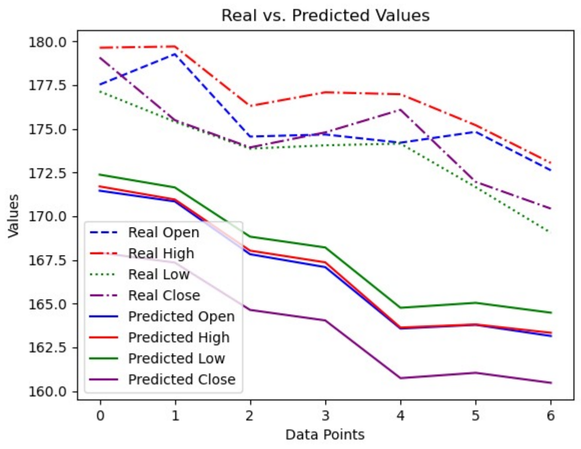


Figure 33: APPLE Ensamble/Bagging 45

AMZN

'Open_mae': 0.02295777902765583, 'Open_mse': 0.0007189241179607356, 'Open_pda[%]': 72.22222222222223, 'High_mae': 0.022042375262092535, 'High_mse': 0.0006721209303849648, 'High_pda[%]': 72.22222222222223, 'Low_mae': 0.05146055952036521, 'Low_mse': 0.0028380221805658357, 'Low_pda[%]': 83.33333333333334, 'Close_mae': 0.020662246202142357, 'Close_mse': 0.0005943466802234182, 'Close_pda[%]': 66.66666666666666

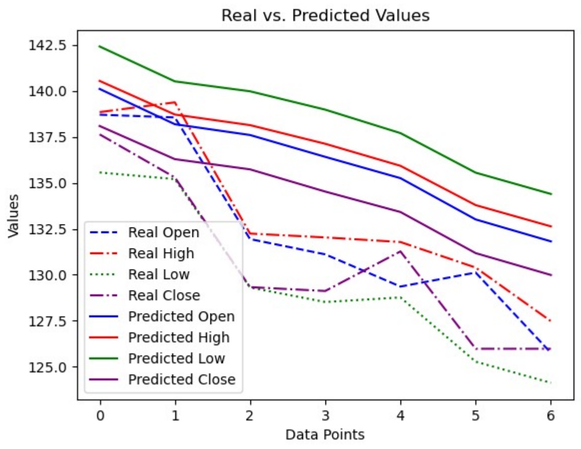


Figure 34: AMZN Ensamble/Bagging 45

GOOGL

'Open_mae': 0.030754424173198607, 'Open_mse': 0.0010830813091971968, 'Open_pda[%]': 55.55555555555554, 'High_mae': 0.036306049600891926, 'High_mse': 0.0016897964245154169, 'High_pda[%]': 83.33333333333334, 'Low_mae': 0.019952332145635177, 'Low_mse': 0.0006473229265501831, 'Low_pda[%]': 83.33333333333334, 'Close_mae': 0.03342277596453047, 'Close_mse': 0.0015115912088468468, 'Close_pda[%]': 66.66666666666666

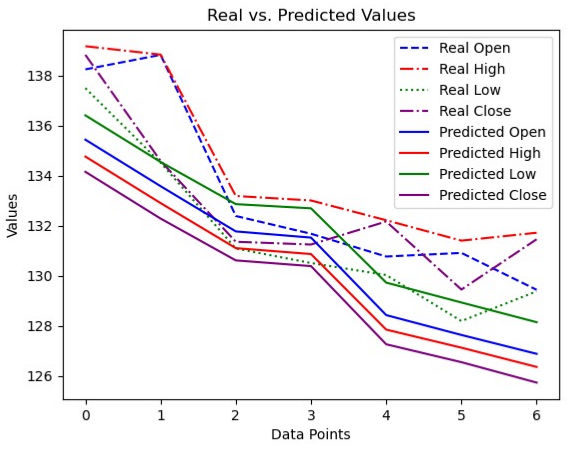


Figure 35: GOGGLE Ensamble/Bagging 45

MSFT

'Open_mae': 0.03634891358938466, 'Open_mse': 0.001758992747219539, 'Open_pda[%]': 55.55555555555555, 'High_mae': 0.040677976576807835, 'High_mse': 0.002216539767446981, 'High_pda[%]': 61.11111111111111, 'Low_mae': 0.022560658574837356, 'Low_mse': 0.0007664416975924204, 'Low_pda[%]': 72.22222222222221, 'Close_mae': 0.03458969681435941, 'Close_mse': 0.0016873119687310315, 'Close_pda[%]': 55.55555555555555

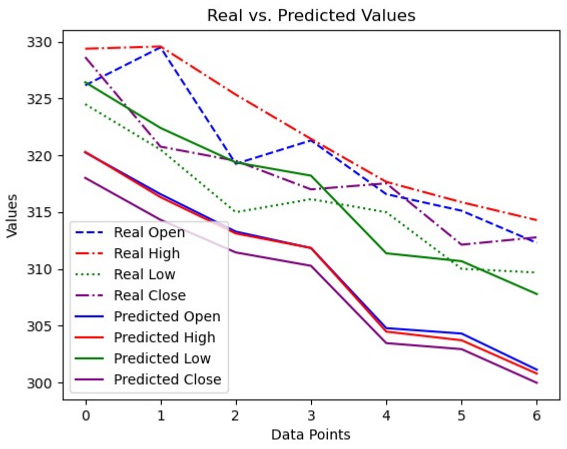


Figure 36: MST Ensamble/Bagging 45

7 Conclusion

Based on the information provided, here are some conclusions that can be drawn:

Data Quantity Matters: The performance and capabilities of the machine learning models discussed are influenced by the amount of historical data used for training. The BiLPET Large model, trained on up to forty years of data, has access to a more extensive dataset compared to the BiLPET Small model, which uses only ten years of data. This suggests that having a larger historical dataset can potentially lead to better predictions.Despite so Small data models seems to work better to catch trends.

Autonomous vs. Auto begging: While the BiLPET Large model is not described as an ideal tool for autonomous prediction, it seems to perform well for "auto begging/ensembled." The bagging testing is done by using a combination of all models to provide better and more robust metrics and overall results. The obtained results with 25 lags model without begging works good for metrics but not for trend prediction. 45 lag models individually give good results in both cases (prediction and trends). Ensembled model with both 45 and 25 lags can catch trends of all companies even tough with a small loss of value in the prediction.

Predictions: These predictions are presented alongside real values and metrics such as mean absolute error (mae) and mean squared error (mse). These metrics help assess the accuracy of the models in predicting stock prices for these specific companies. The PDA metrics has been used for a better comprehension of trends.

Bagging Metrics: The passage introduces "bagging metrics" for various companies, including Apple, Amazon, Google, and Microsoft. These metrics measure the performance of the models when used in an ensemble (bagging) approach. Bagging is a technique that combines multiple models to improve overall prediction accuracy. The provided metrics indicate how well the models perform in this context.

In summary, the passage highlights the importance of data quantity in training machine learning models for stock price prediction. It also suggests that different models may have specific strengths and use cases. Additionally, the introduction of bagging metrics suggests an interest in assessing the performance of these models in ensemble-based prediction strategies.

Generally speaking models perform goods in both MAE and MSE, so the main focus is posed upon trend during ensemble/bagging.

More addition and adjustment on these models is adopting a fine-tuning on data of the chosen company. This approach might halt to have better prediction of both trend and stocks values for the specific case. The architecture can be expanded by adjusting layer dimension, in these cases it is optimal to use more data from more company, even more than the four major here used.

Further context and information would be needed to make more specific conclusions and recommendations regarding the suitability of these models for particular applications in finance or investment.

References

[1] *Pandas*. https://pandas.pydata.org/, 2023.

[2] *Matplotlib*. https://matplotlib.org/, 2023.

[3] *Scikit-Learn*. https://scikit-learn.org/stable/, 2023.

[4] *PyTorch*. https://pytorch.org/, 2023.

[5] *Numpy*. https://numpy.org/, 2023.

[6] https://docs.python.org/3/library/os.html, 2023.