HOMERWORK 2

FINANTIAL TECHNOLOGY

MIGUEL BENALCAZAR - D11949001

(i) Please use APPLE.csv to plot a.) Candlestick chart with two moving average lines (10 days and 30 days). b.) KD line chart. c.) Volume bar chart.

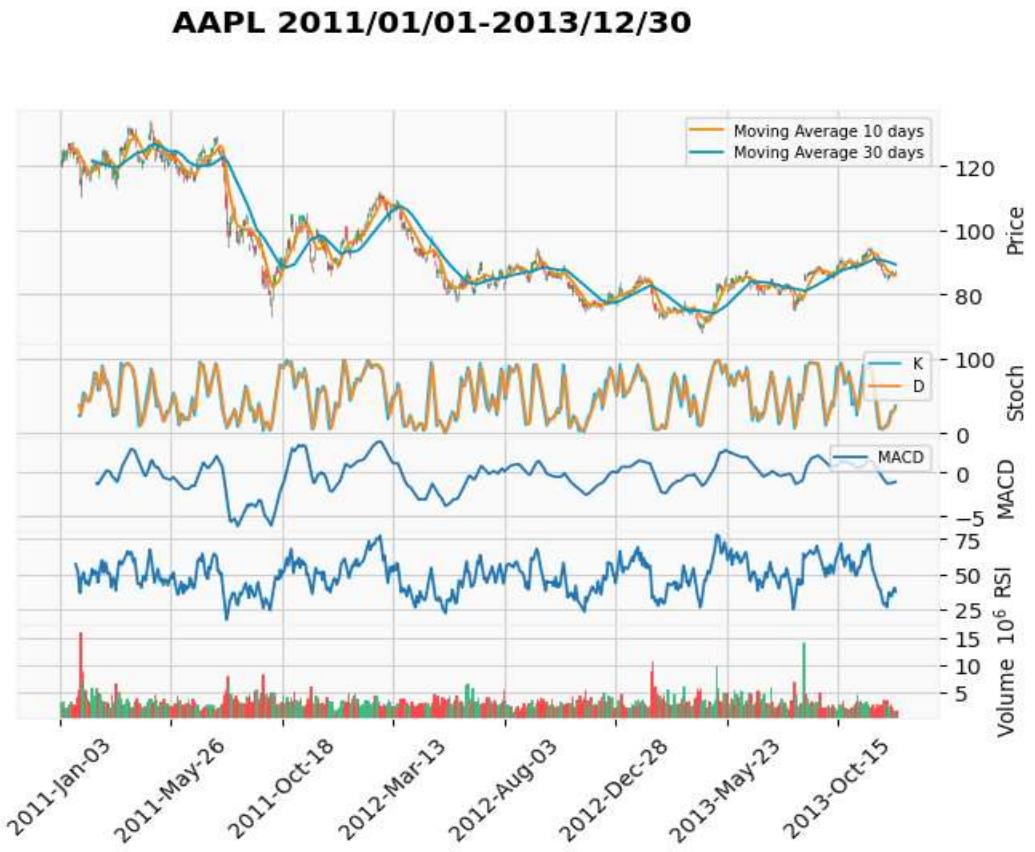

Figure 1. AAPL stock 2011/01/01 - 2013/12/30

(ii) Please at least add four features from question (i) into your input which are 'Moving Average 10 days', 'Moving Average 30 days', and 'K, D from KD line chart'. And we want all features except Date to be normalized on a scale of 0 to 1 by the below equation. You can also add other features to help your model get better performance. (e.g., If you think weekdays are important to stock price, you can add an one-hot attribute of weekdays.) Please discuss what you did for data preprocessing

The information from the APPLE.csv returns the stock information from 2011-01-03 to 2013-12-31, as features of APPL stock we have [Date, Open, High, Low, Close, Volume, Adj Close] features from APPL stock.

Adding more technical indicators

With the purpose of adding more features to our input, some information from the *technical analysis indicators* was added. These indicators are often used to understand the current market trend. Moving averages and momentum indicators are the most common indicators.
As mentioned in the hw2-guidlines, we can use the following indicators: 'Moving Average 10 days', 'Moving Average 30 days', and 'K, D from KD line chart'.

In addition to the previously mentioned indicators, I have decided to add two extras as features. These are the Moving Average Convergence/Divergence (MACD) and the Relative Strength Index (RSI). A brief introduction to MACD and RSI is shown below.

Moving Average Convergence/Divergence (MACD): It is a trend-following momentum indicator that shows the relationship between two exponential moving averages (EMAs) one with a period of 26 EMA and the other with a period of 12 EMA. The result from the previous calculation gives the MACD line.

From the MACD line, a nine-day EMA line is obtained which is well-known as a signal line. The signal line is useful to trigger signals for selling or buying.

Relative Strength Index (RSI): It is a momentum indicator that measures the speed and magnitude of a stock's recent price. It evaluates the conditions of the price of the security and it shows when security is overbought and oversold. If this index is above 70 indicates an overbought situation, while below 30 indicates an oversold. RSI is calculated under the following equation:

$$RSI = 100 - \left[\frac{100}{1 + \frac{Average\ Gain}{Average\ Loss}}\right]$$

Once we include the values of MACD and RSI, the inputs will be set as: [Open, High, Low, Volume, Adj Close, SMA-10, SMA-30, %K, %D, MACD, MACD_Signal, MACD_Hist, RSI], and set as output we have close.

Creating Datasets and Filling Missing Values

The dataset was split into 3 following the instructions, for the training dataset from 2011/01/01 to 2012/12/31, From 2013/01/01 to 2013/06/30 for the validation dataset, and 2013/07/01-2013/12/30 for the testing dataset.

Due to the fixed data window for the technical analysis indicators, some of the data in the table became Null or Nan during the division of the dataset. The mean function of each column was used to fill Nan or missing values. It is possible, however, to replace the missing values with zeros, medians, or most frequent values.

Date	Open	High	Low	Close	Volume	Adj Cl...	%K	%D	SMA-10	SMA-30	MACD	MACD...	MACD...	RSI
2011-01-...	120.59	121.63	120.01	121.03	1893100	100.75	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	121.78	122.25	119.46	121.86	2816000	101.44	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	121.5	125	120.57	124.91	3059700	103.98	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	125.16	125.59	121.91	122.38	2178000	101.87	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	122.8	124.03	122.17	123.17	2035300	102.53	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	123.28	125.24	122.17	124.74	2031100	103.83	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	125	125.42	124.3	125.13	1857700	104.16	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	126.25	127.14	125.64	126.2	1837500	105.05	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	126.93	127.17	124.41	125.18	2750400	104.2	nan	nan	nan	nan	nan	nan	nan	nan
2011-01-...	124.86	125.73	123.36	125.63	3199500	104.57	nan	nan	124.023	nan	nan	nan	nan	nan
2011-01-...	126.23	127.64	125.3	127.56	2570900	106.18	nan	nan	124.676	nan	nan	nan	nan	nan
2011-01-...	127.09	127.73	125.77	124.18	2649100	103.49	nan	nan	124.908	nan	nan	nan	nan	nan
2011-01-...	123.99	124.79	122.66	124.16	2075900	103.47	nan	nan	124.833	nan	nan	nan	nan	nan
2011-01-...	125.25	125.66	124	125.6	2222000	104.67	nan	nan	125.155	nan	nan	nan	nan	nan
2011-01-...	124.83	125.69	123.25	123.92	2693900	103.27	nan	nan	125.23	nan	nan	nan	nan	57.17721...
2011-01-...	123.08	123.08	120.32	122.64	4035200	102.21	nan	nan	125.02	nan	nan	nan	nan	53.51055...
2011-01-...	123.26	123.35	121.22	122.1	5431900	101.76	nan	nan	124.717	nan	nan	nan	nan	51.99791...
2011-01-...	121.66	121.9	114.03	116.33	12838700	96.95	24.03965...	37.87362...	123.73	nan	nan	nan	nan	39.23528...

Figure 2. Missing Values Dataset

The process of filling in missing values was performed right after splitting the dataset since the mean will be calculated over the data in the training dataset but not over the entire dataset. In this way, we avoid leaking information from the validation or testing dataset in the training dataset. The following figure shows how the training dataset ended after filling the missing values.

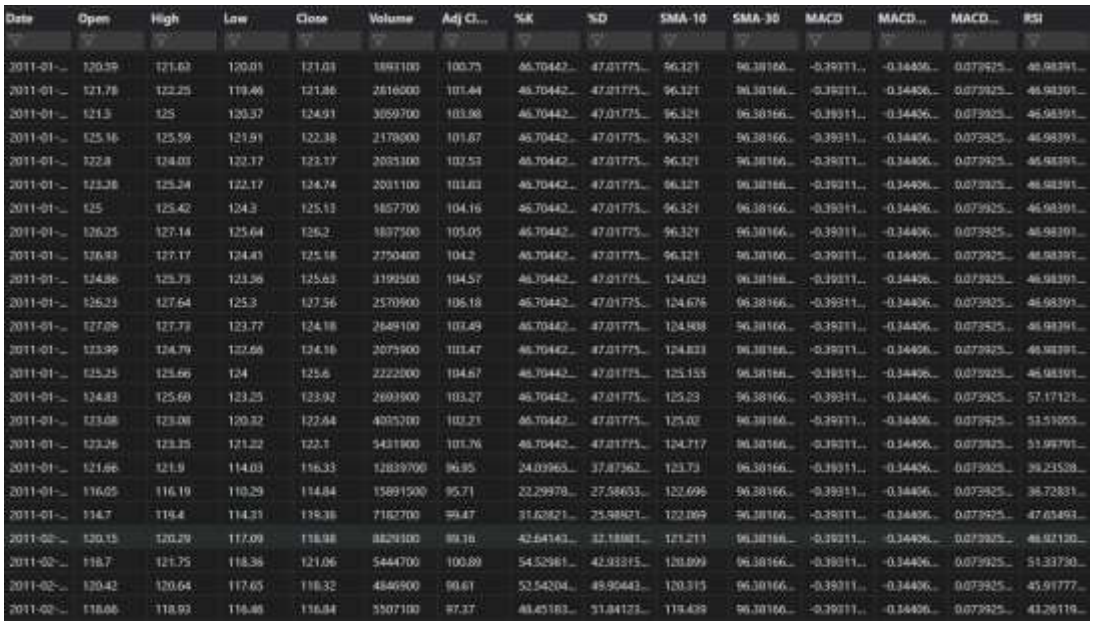

Figure 3. Missing Values Filled - Training Dataset

Normalize on a scale of 0 to 1

To normalize the dataset, I used the MinMaxScaler from the sklearn.preprocessing library. In the google colab file, the function is defined inside the class RNN as follows.

```
def NormalizeMinMax(self, data):
    scaler = MinMaxScaler(feature_range=(0, 1))
    return scaler, scaler.fit_transform(data)
```

Figure 4. Normalize Min Max function

A MinMaxScaler object was defined with a feature_range as a variable to take from 0 to 1. The function was created to return “scaler" and the Normalize data. The scaling object ‘scaler’ will be used afterward to reverse compute and convert the 0-1 data to its Min-Max values in “Close” label.

(iv) Please construct an RNN model with a Vanilla RNN cell for predicting the ’Close’ value of the next day according to the mean square error.

$$MSE = \frac{1}{n}\sum(y_i - \tilde{y}_i)^2$$

Please explain how you design your model.

RECURRENT NEURAL NETWORK (RNN)

Before designing the RNN model, let's verify the number of features or inputs and the output. For the inputs, the array is formed by 13 features, while the output is just 1, the "Close" value. Therefore, 13 will be used as the input size.

Then, RNNModel is defined. To use the model RNN from Pytorch it is required to import the library torch.nn. The following inputs are required:

Input size: The number of expected features
Hidden size: The number of features in the hidden state
Number of Layers: The number of RNN's stacks
Nonlinearity: It is the activation function that in our case is *RELU* since we need to perform a regression
Batch first: It was set to be *TRUE* since the input has the form *(batch, seq, feature)*
Dropout: 0

After processing the RNN it will pass to a Linear layer which applies a linear transformation to the input data base on the following equation:

$$y = xA^T + b$$

So, the model to be used in this stage will have a vanilla RNN and a linear layer stacked.
The linear layer will receive as input the hidden size and the output size from the vanilla RNN outputs, giving as a result the prediction of "CLOSE".
The mean square error (MSE) was used as a loss function, as mentioned in the hw2-guidelines.
As for the criterion or optimizer, Adam was the one to be used in this experiment. The learning rate was defined to be 1×10^{-4}.
The RNN model was defined as follows:

```
class RNNModel(nn.Module):
    def __init__(self,input_size, hidden_size, layer_size, output_size, device, dropout = 0 ):
        super(RNNModel, self).__init__()
        self.device = device
        # Hidden dimensions
        self.hidden_size = hidden_size
        # input dimensions
        self.layer_size = layer_size

        # Building RNN
        # (batch_size, time_step, input_size)
        # batch_size = number of samples per batch
        self.rnn = nn.RNN(
            input_size = input_size, # number of features
            hidden_size = hidden_size,
            num_layers = layer_size,
            batch_first = True,
            nonlinearity = 'relu',
            dropout = dropout)

        # hidden_size is the in_features
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):

        # x(batch, time_step, input_size)
        # h_state(n_layers, batch, hidden_size)
        # r_out(batch, time_step, output_size)
        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_size, x.size(0), self.hidden_size))
        if self.device != 'cpu':
          h0 = h0.to(self.device)
        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

In the training process, an early stop part was defined in order to stop the training process base on the loss obtained after verifying the validation dataset. Thus, the best model is saved and loaded once the training process stops, avoiding falling into overfitting. The early stop part was designed as follows:

```
if current_loss != 0:
  if best_Loss > current_loss:
    # Update best loss
    best_Loss = current_loss
    best_Model = copy.deepcopy(self.model.state_dict())

    self.loss_train = current_loss_training
    self.loss_validate = current_loss
    self.epoch = epoch
    patience_acum = 0
  else:
    patience_acum +=1
else:
  patience_acum += 1

if patiente_acum >= patiente:
  self.model.load_state_dict(best_Model)
  self.loss_train = current_loss_training
  self.loss_validate = current_loss
 self.epoch = epoch
 break
```

The early stop will compare the previous best validation loss with the current validation loss, and if it is not being reduced and the accumulator will increase by 1. This accumulator is called patience. If the number of the accumulator patience is greater than the value patience set, the training will immediately stop.

The hyperparameters for the RNN are shown below, it is worth mentioning that the same hyperparameters were used for all models RNN, LSTM, and GRU to be able to compare them. The only parameter that will change is the epoch value because it will depend on the early stop and the training process.

```
__LEARNING_RATE__ = 1e-4
__EPOCH__ = 1000
__HIDDEN_SIZE__ = 10
__LAYER_SIZE__ = 1
__BATCH_SIZE__ = 16
__PATIENTE__ = 2
__DROPOUT__ = 0
```

The hidden size was chosen by trial and error. Recall that 1 hidden layer will be enough for simple problems, if we increase this, the complexity of our model will increase, and with it the capacity to learn how to handle complex problems. On the internet, there are several pieces of advice regarding the number of hidden layers.

As a reference and based on my experience, I used the following rule "*The number of hidden neurons should be between the size of the input layer and the size of the output layer*" as the starting point. Thus, the value of the best-hidden layer size I tested was 10. 16 was defined as the batch size of this experiment.

RNN Results

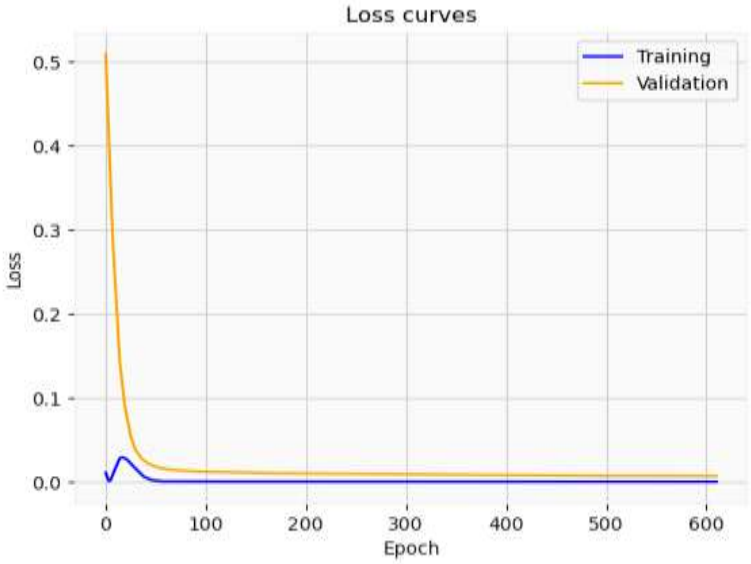

Figure 5. Training and Validation Loss Curves for RNN vanilla

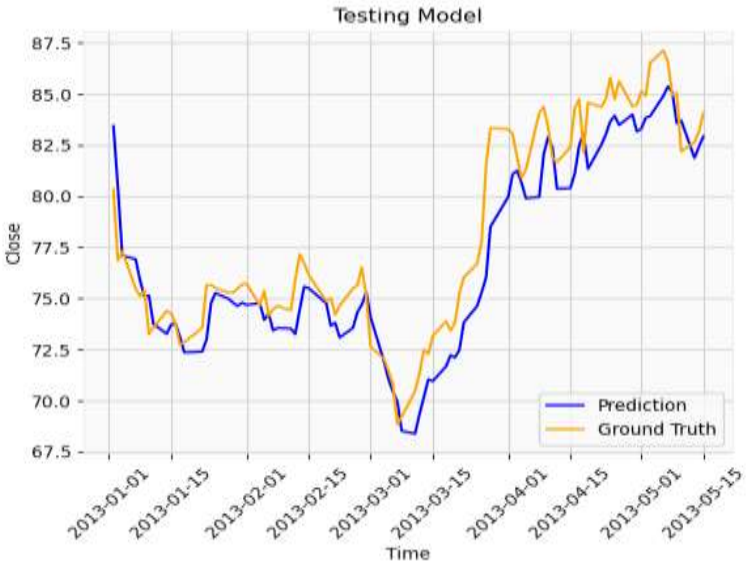

Figure 6. RNN Testing results

The early stop detains the training process at the following parameters

Epoch = 609
Training Loss = 0.00007
Validation Loss = 0.00691
Testing Loss = 0.00973

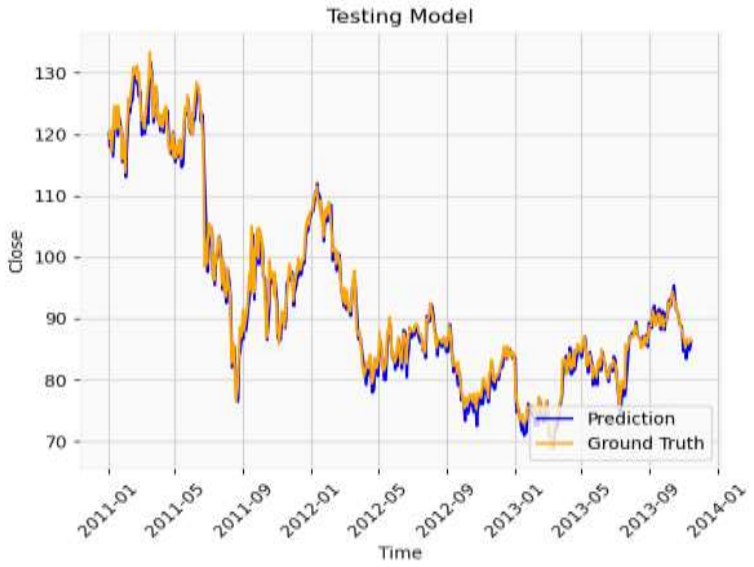

Figure 7. Testing RNN model with the entire data from APPL stock provided

Total Loss: = 0.000957

(vi) Substitute LSTM cell for Vanilla RNN and repeat (iv), (v).

LONG SHORT-TERM MEMORY (LSTM)

In the designing process of the LSTM process, I used the same library imported for the previous model torch.nn but in this case I used the LSTM. The following are the parameters required to use this cell:

Input size = The number of expected features
Hidden size = The number of features in the hidden state h
Number of Layers = The number of stacked LSTM cells
Batch First = True for (batch, seq, feature)
Dropout = 0

Similar to the RNN model previously exposed, the LSTM was stacked with a linear layer to get the final result.
The model is as follows:

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size, device, dropout = 0):
        super(LSTMModel, self).__init__()

        self.device = device
        self.hidden_size = hidden_size
```

```
        self.num_layers = num_layers

        self.lstm = nn.LSTM(
            input_size = input_size,
            hidden_size= hidden_size,
            num_layers = num_layers,
            batch_first = True,
            dropout = dropout)

        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
      # x(batch, time_step, input_size)
      # h_state(n_layers, batch, hidden_size)
      # c_state(n_layers, batch, hidden_size)
        h0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size)) # hidden state
        c0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size)) # internal state
        if self.device != 'cpu':
          h0 = h0.to(self.device)
          c0 = c0.to(self.device)

        out, (hn, cn) = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out
```

In this case, similar to the RNN, we need to initialize as zero tensors the hidden state tensor and the c state. Each requires 3 tuples of (n_layers, batch size, hidden size).

The same hyperparameters as RNN were used in the LSTM.

```
__LEARNING_RATE__ = 1e-4
__EPOCH__ = 1000
__HIDDEN_SIZE__ = 10
__LAYER_SIZE__ = 1
__BATCH_SIZE__ = 16
__PATIENTE__ = 2
__DROPOUT__ = 0
```

LSTM Results

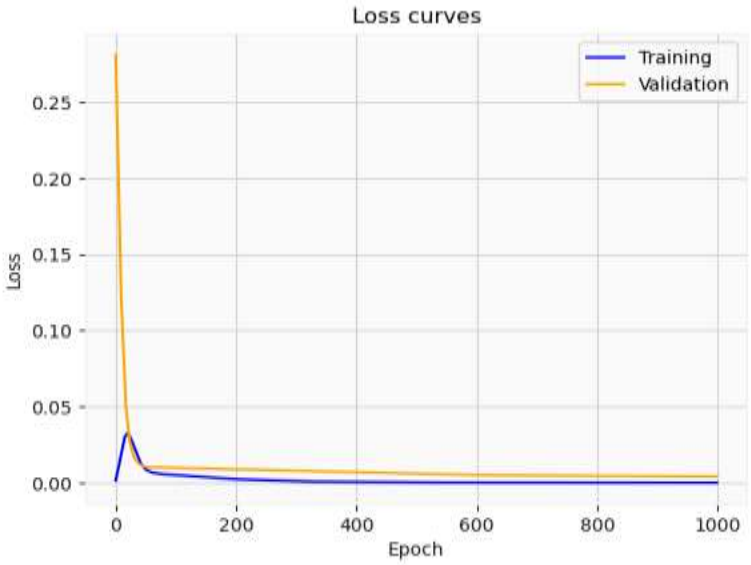

Figure 8. Training and Validation Loss Curves for LSTM cell

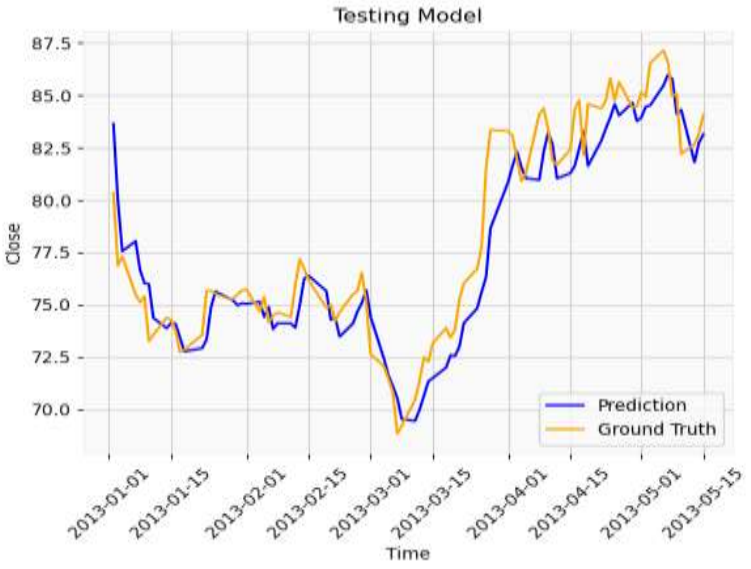

Figure 9 LSTM Testing results

Epoch = 999
Training Loss = 0.00009
Validation Loss = 0.00421
Testing Loss = 0.00735

The early stop was never triggered during the training process of the LSTM model. It seems the validation loss was slightly downward the whole time.

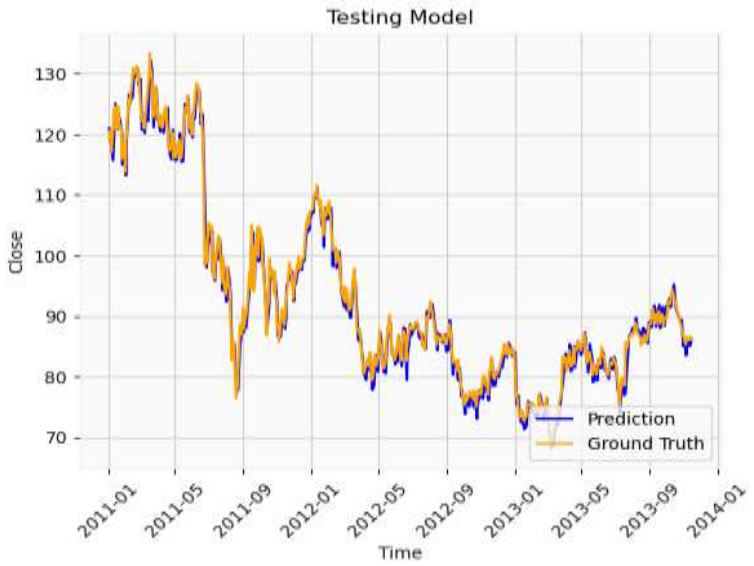

Figure 10. Testing LSTM model with the entire data from APPL stock provided

Total Loss: 0.00087400

(vii) Substitute GRU cell for Vanilla RNN and repeat (iv), (v).

GATED RECURRENT UNIT (GRU)

Similar to RNN and LSTM, GRU can be imported using torch.nn. It requires the same arguments as RNN requires. The hyperparameters used are the same as those ones used in the previous models.

GRU Results

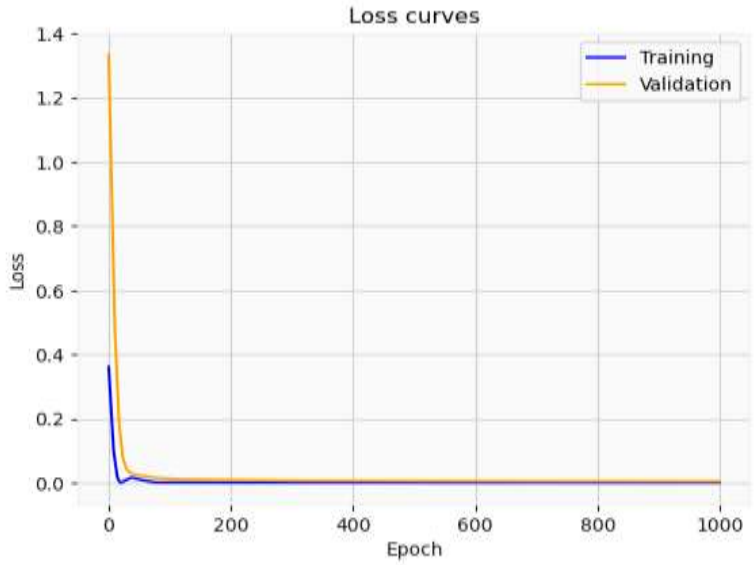

Figure 11. Training and Validation Loss Curves for GRU cell

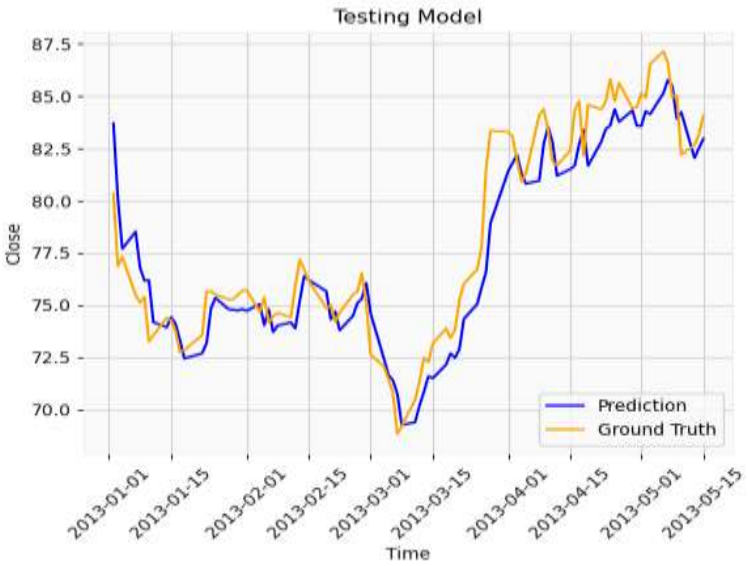

Figure 12. GRU Testing results

Epoch = 1000
Training Loss = 0.00019
Validation Loss = 0.00432
Testing Loss = 0.00732

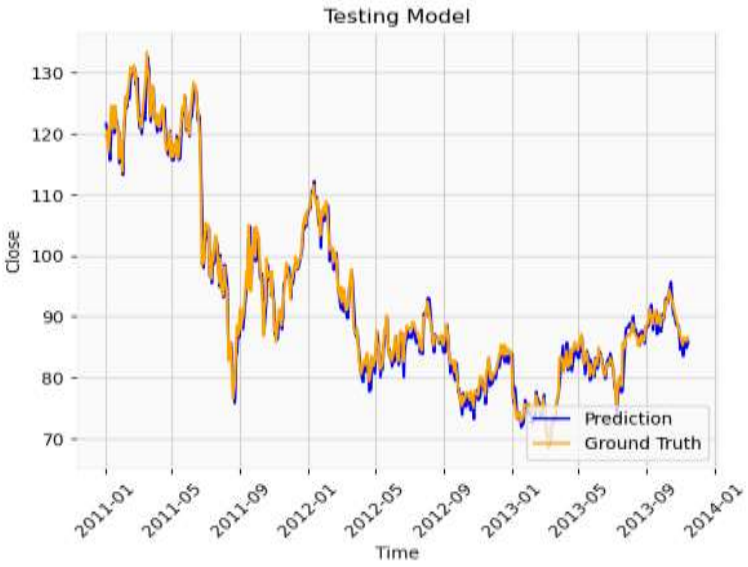

Figure 13. Testing GRU model with the entire data from APPL stock provided

Total Loss = 0.000956280

(viii) Discuss your findings from (iv) to (vii)?

The following table contains all of the results using early stop and in the second table the results of the training, validation, and testing process by epoch 300.

Table 1. Comparison models by using early stop

__EPOCH__ = 1000			
	RNN	*LSTM*	*GRU*
Epoch	609	999	999
Training Loss	0.00007	0.00009	0.00019
Validation Loss	0.00691	0.00421	0.00432
Testing Loss	0.00973	0.00735	0.00732
Time	1 m 23.5 s	4m 41.3s	4m 53.1s

Table 2. Comparison models result by 300 epochs

__EPOCH__ = 300			
	RNN	*LSTM*	*GRU*
Epoch	300	300	300
Training Loss	0.00012	0.00049	0.00011
Validation Loss	0.00633	0.01497	0.01641
Testing Loss	0.00861	0.02446	0.01593
Time	38.4s	1m 22.5s	1m 28.3s

Table 2, it is easily perceived that the RNN training process takes less time to train 300 epochs than the training process of the LSTM cell and the GRU cell. It takes 38.4 seconds to train the model while 1 minute and 22 seconds to train the LSTM model, and lastly, 1 minute and 28 seconds for the GRU training process. If we compare the testing loss in the table 2, we verify that the testing loss from the RNN is much better that the testing loss obtained by the LSTM and the GRU. It seems that LSTM and GRU require more time and more epochs to reach their best performance as Table 1 shows.

If we check the testing loss from table 1, it is easy to see that the testing loss from the RNN is not so good as the one from the LSTM and GRU. The testing loss from the GRU cell is even slightly better than the LSTM cell testing function in table 2. However, in table 1 the testing loss from the GRU is much better that LSTM. It is because GRU is derived from the LSTM and GRU uses less parameters. Therefore, the training process is faster than the LSTM. We need to keep in mind that LSTM is slower since it has more parameters due to its memory gate. Hence, LSTM will perform better in large sequences. Besides, both LSTM and GRU were designed to tackle the vanishing gradient problem.

2. (Bonus) Daily News for Stock Price Prediction

The news could be one of the most important factors in stock market prediction. It is challenging to monitor news from many sources in real-time. We provide a dataset 'APPLE_news.pkl', historical news of Apple Inc. from Bloomberg. The features are introduced as follows: - Tickers: Stock symbol *Apple Inc. (AAPL) - Date: Published date - Title: Headline of the news - Content: Content of the news.

(i)(Bidirectional Encoder Representations from Transformers) Please use the feature 'title' or 'content' in the 'APPLE_news.pkl ' dataset. You can concatenate daily title and content, then generate 768 dimensions embeddings using pre-train BERT first. Furthermore, you can also add the feature you used in 1-(ii) and news embedding to improve your model (You can choose one from Vanilla RNN, LSTM, or GRU). Please describe your idea, how it was implemented, and compare the results with previous models. (Hint: You can use the Huggingface Transformers library with PyTorch.) Reference document: https://huggingface.co/transformers/v3.0.2/

For opening the file 'APPLE_news.pkl', it is required to install the library pickle. All of the functions were included into the class LoadFile in google collaboration.

As mentioned, the file has some columns which are Date, Title, and Content. A function will extract the information base on the requirements that can be only 'title', only 'Content', or the combination of 'title + Content'.

Similar to the first stage, the data frame is divided into 3 different parts, training, validation, and testing dataset. Figure 14 shows the training dataset. A blank space was used to fill in the missing values in the last column.

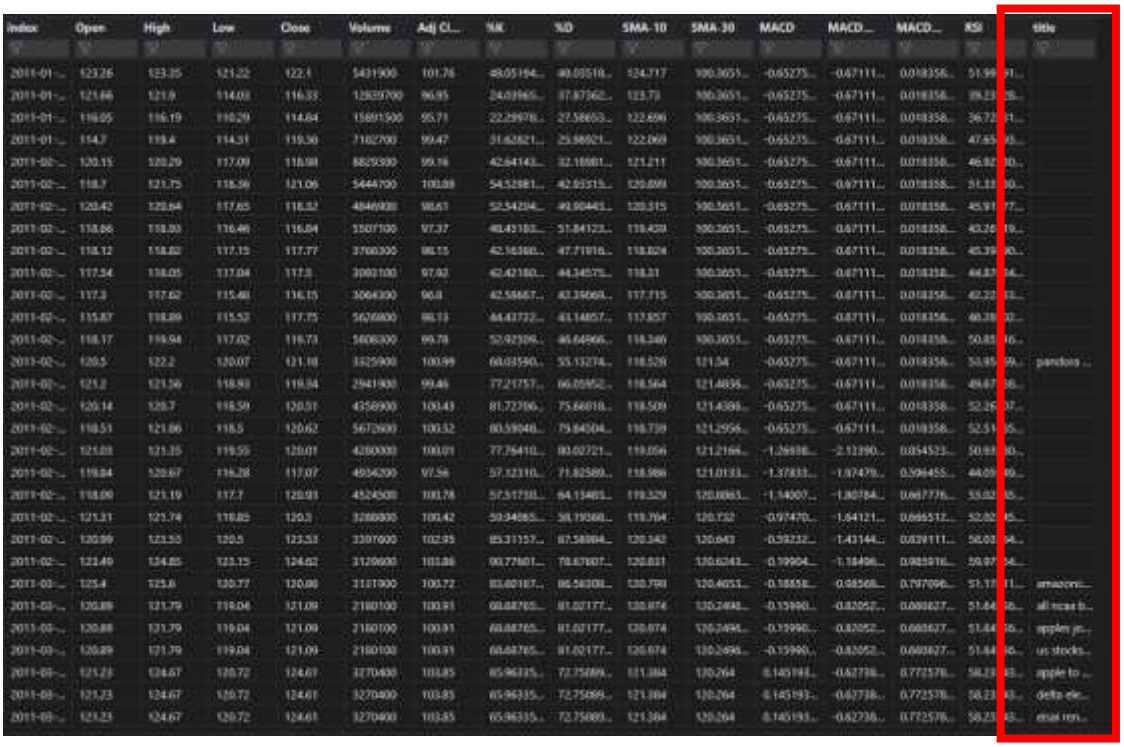

Figure 14. Training dataset including title of the news

The main goal of this stage is to use the text as input of BERT to extract it embedding, taking into account that BERT is formed by the encoding part of the transformers.

To use BERT from Huggingface Transformers library with Pytorch, it is necessary to install the library transformers as follows:

! pip install transformers

In this case I used DistilBERT which is smaller, faster, cheaper, and a lighter version of BERT. It is worthy to mention that I faced some problems with my memory RAM when using the 'bert-base-uncased' which is the original BERT.

Once DistilBERT was installed, I loaded the pre-trained BERT model by using the following code:

```
''' For DistilBERT: samller, faster, cheaper, lighter 40% less parameters than bert-base-uncased
   run 60% faster
   95% of BERT's performance as meassured on the GLUE language understanding benchmark.
'''
# For DistilBERT:
model_class, tokenizer_class, pretrained_weights = (ppb.DistilBertModel, ppb.DistilBertTokenizer, 'distilbert-base-uncased')

# Load pretrained model/tokenizer
tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
model = model_class.from_pretrained(pretrained_weights)
```

Tokenization

Before the title of the news is passed to BERT, it is required to tokenize the sentences. The tokenize process is the way the text is preprocessed to be used by the pre-trained BERT model. In this process two special tokens are added to the sentence. [SEP] and [CLS] are the special tokens. [SEP] means the separation between two sentences and [CLS] is added to the beginning of the text.

For example, if 'pandora sales more than double ahead of ipo' is the title of a determined news, the first step of the tokenization is adding the special tokens. Therefore, the sentence will become **'[CLS]**pandora sales more than double ahead of ipo **[CLS]**'. The second step of the tokenization is substitute tokens with their ids '**101** 1037 17453 14726 19379 12758 2006 22293 **102**'.

```
tokenized = data.apply((lambda x: self.tokenizer.encode(
        x, # Sentence to encode.
        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
.
```

Padding

After tokenization, the sentence has been represented by a list of tokens. Since the length of the token ids is different from each other. Thus, the longest representation is taken to be the base standard that each token ID or representation might have. If the representation does not have the same length as the base, in order to complete the length, the remainder will be filled with zeros.

```
'''Padding process base on max length of token'''
    max_len = 0
    for i in tokenized.values:
      if len(i) > max_len:
        max_len = len(i)
    padded = np.array([i + [0]*(max_len-len(i)) for i in tokenized.values])
```

Masking

Now, Bert needs another variable to tell it to ignore the padding added. This step consists in creating an array where it contains 1 if it is a representation and 0 if it is padding.

```
attention_mask = np.where(padded != 0, 1, 0)
```

Model

To use the DistillBERT model, the variable tokenized and the attention_mask need to be converted into torch and then they will be use as inputs in the model as follows:

```
'''convert into torch'''
    input_ids = torch.tensor(padded)
    attention_mask = torch.tensor(attention_mask)

    with torch.no_grad():
        last_hidden_states = self.bert(input_ids, attention_mask = attention_mask)
        return last_hidden_states[0][:,0,:].numpy()
```

The last_hidden_states[0] represents the BERT output Tensor/predictions and last_hidden_states[0][: (all sentences), 0 (only the first position: [CLS]), : (all hidden unit outputs)]. The result of the embedding of each sentence is an array of 767 positions.
The function in charge of performing this process is the following:

```
def embeddings_pretrain_BERT(self, data):
    tokenized = data.apply((lambda x: self.tokenizer.encode(
        x, # Sentence to encode.
        add_special_tokens = True # Add '[CLS]' and '[SEP]'
    )))

    # print(tokenizer.convert_ids_to_tokens(tokenized[190]))

    '''Padding process base on max length of token'''
    max_len = 0
    for i in tokenized.values:
        if len(i) > max_len:
            max_len = len(i)

    padded = np.array([i + [0]*(max_len-len(i)) for i in tokenized.values])

    attention_mask = np.where(padded != 0, 1, 0)

    '''convert into torch'''
    input_ids = torch.tensor(padded)
    attention_mask = torch.tensor(attention_mask)

    with torch.no_grad():
        last_hidden_states = self.bert(input_ids, attention_mask = attention_mask)
        return last_hidden_states[0][:,0,:].numpy()
```

So far, the titles were embedding by the use of DistillBert. These embeddings were merged with the previous information as it is shown in the figure 15.

The process to train the RNN, LSTM, and GRU models are the same. The size of the new features is the addition of the 13 previous features used plus the new 767 features, giving a total of 780 features as inputs.

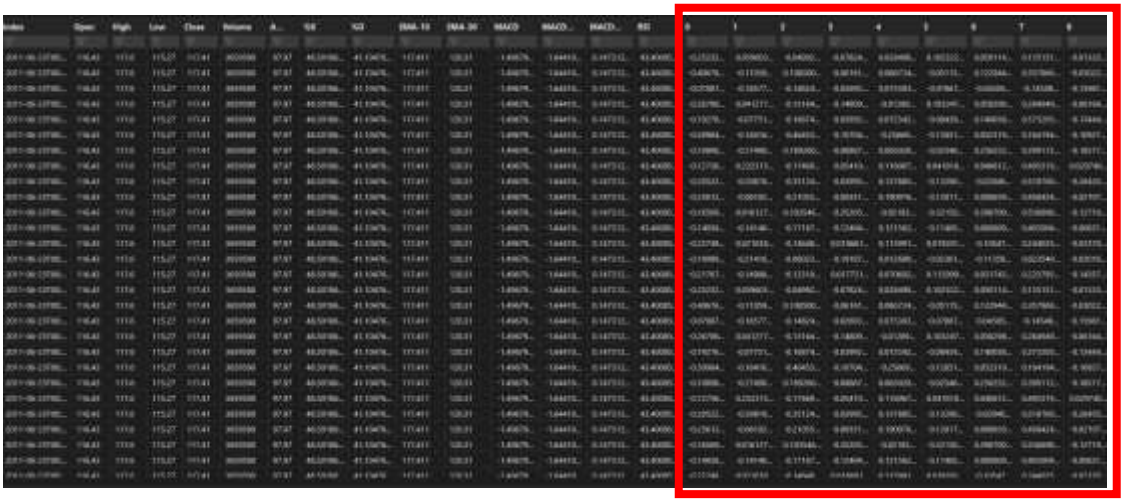

Figure 15. New dataset including the BERT embeddings

I could not use the same parameters as I did previously because when I tried to train the models with the same hyperparameters, the results showed a low performance. The validation loss got stuck in a value and the model stopped learning.

Therefore, new hyperparameters were used to train each model. The way to define the new hyperparameters was based on trial and error. Moreover, a dropout layer was included to help with the training process in addition to increasing the batch size.

RNN

Hyperparameters

__LEARNING_RATE_BERT__ = 1e-4
__EPOCH_BERT__ = 1000
__HIDDEN_SIZE_BERT__ = 20
__LAYER_SIZE_BERT__ = 1
__BATCH_SIZE_BERT__ = 200
__PATIENTE_BERT__ = 2
__DROPOUT__ = 0.2

Results

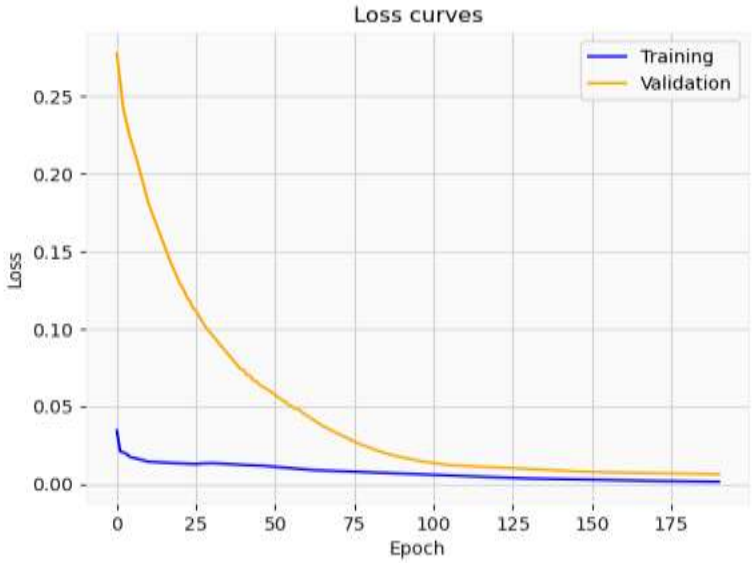

Figure 16. Training and Validation Loss Curves for RNN including the title of news

Epoch = 190
Training Loss = 0.00132
Validation Loss = 0.00611
Testing Loss = 0.00406

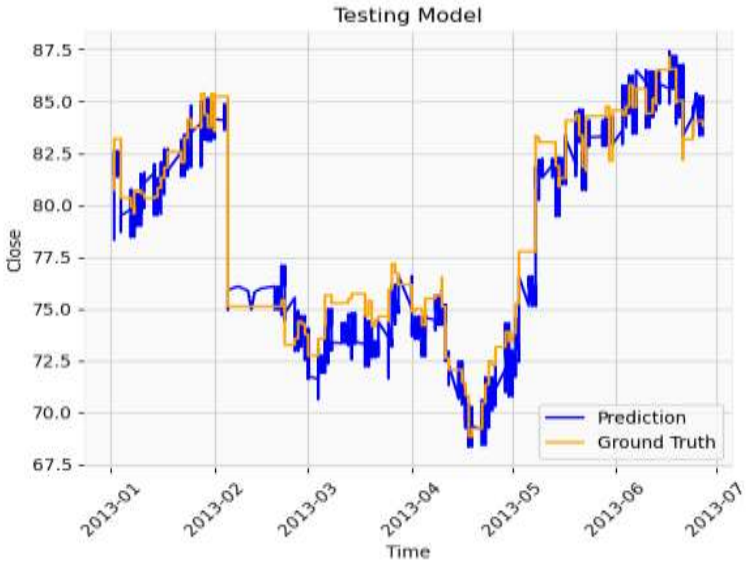

Figure 17. RNN Testing results

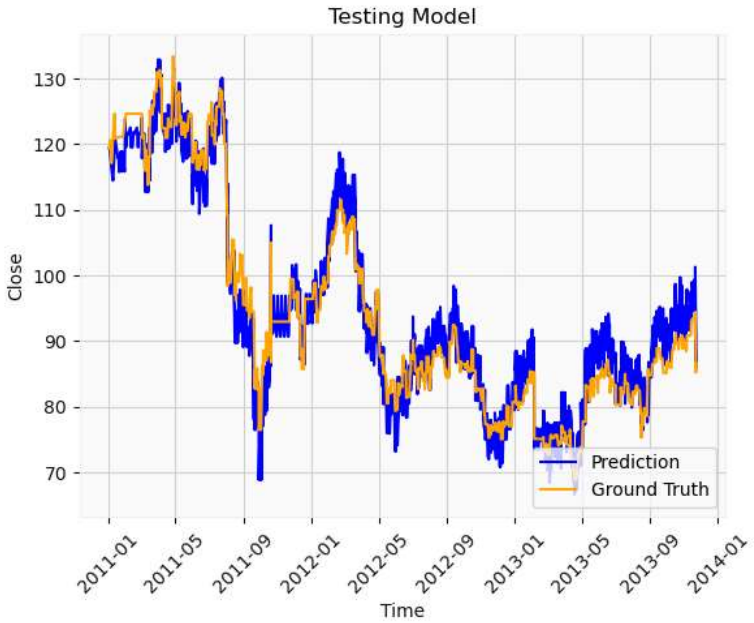

Figure 18. Testing RNN model with the entire data from APPL stock and news provided

Total Loss: 0.002740

LSTM

Hyperparameters

__LEARNING_RATE_BERT__ = 1e-4
__EPOCH_BERT__ = 1000
__HIDDEN_SIZE_BERT__ = 20
__LAYER_SIZE_BERT__ = 1
__BATCH_SIZE_BERT__ = 200
__PATIENTE_BERT__ = 20
__DROPOUT__ = 0.2

Results

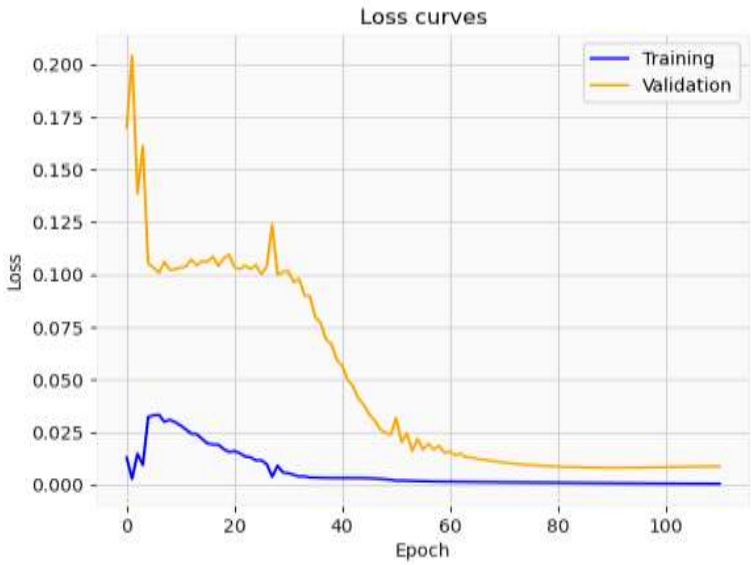

Figure 19. Training and Validation Loss Curves for LSTM including the title of news

Epoch = 110
Training Loss = 0.00051
Validation Loss = 0.00872
Testing Loss = 0.01194

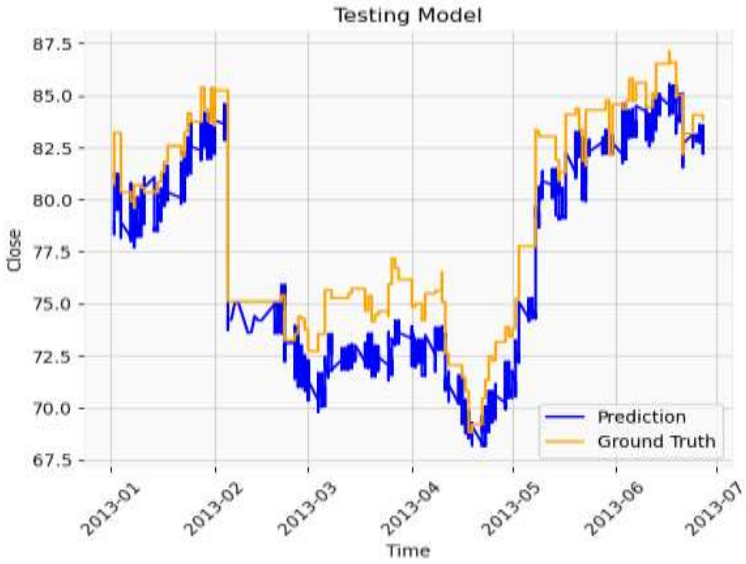

Figure 20. LSTM Testing results

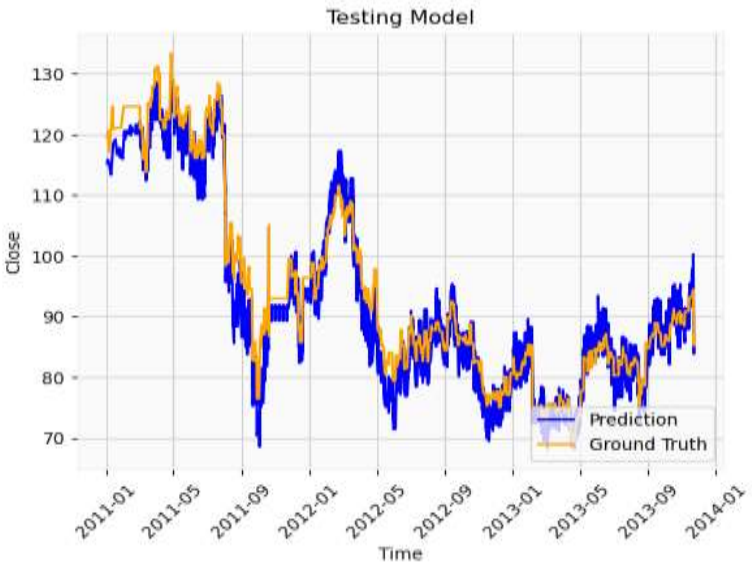

Figure 21. Testing LSTM model with the entire data from APPL stock and news provided

Total Loss: 0.00339

GRU

Hyperparameters

__LEARNING_RATE_BERT__ = 1e-4
__EPOCH_BERT__ = 1000
__HIDDEN_SIZE_BERT__ = 20
__LAYER_SIZE_BERT__ = 1
__BATCH_SIZE_BERT__ = 200
__PATIENTE_BERT__ = 10
__DROPOUT__ = 0.2
__WEIGHT_DEKAY__ = 1e-9

Results

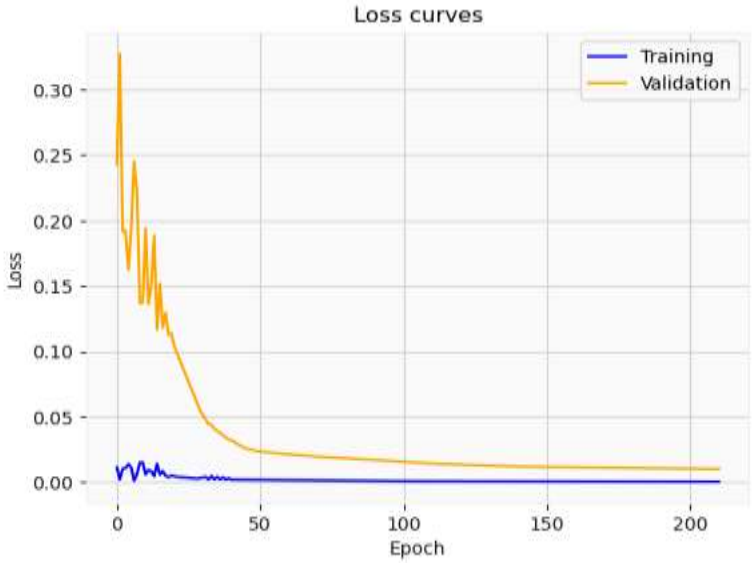

Figure 22.. Training and Validation Loss Curves for GRU including the title of news

Epoch = 210
Training Loss = 0.00044
Validation Loss = 0.01014
Testing Loss = 0.00451

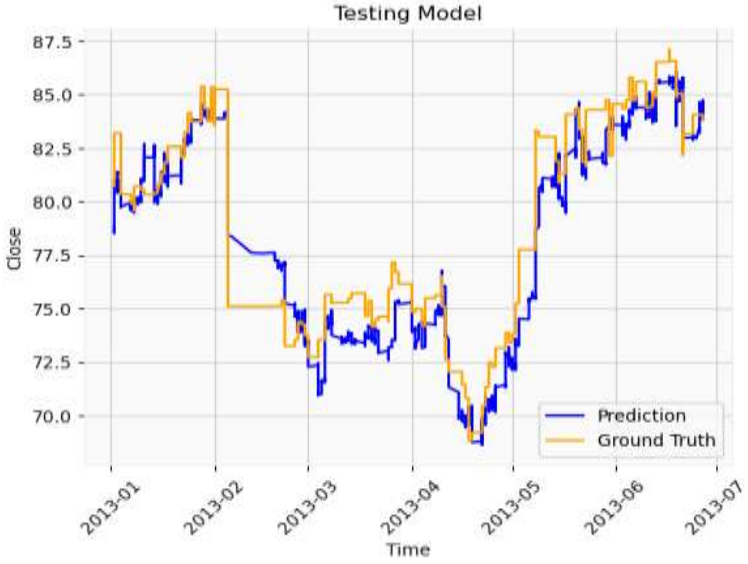

Figure 23. GRU Testing results

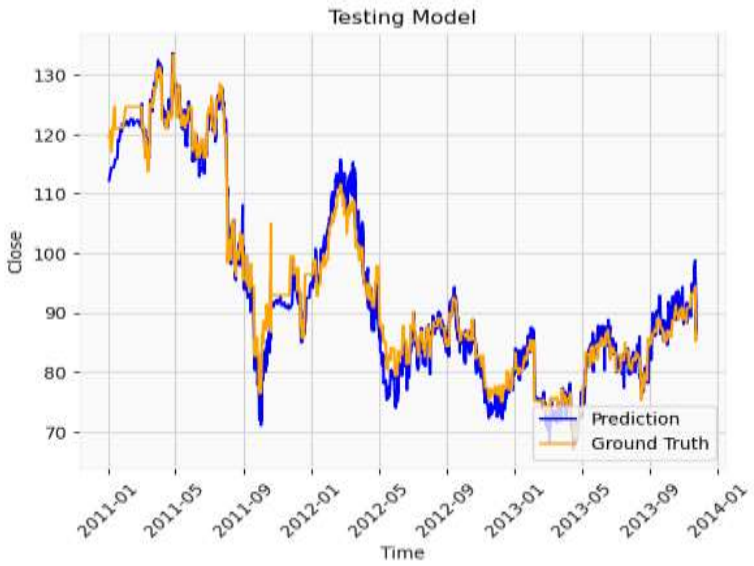

Figure 24. Testing GRU model with the entire data from APPL stock and news provided

Total Loss: 0.001596

Table 3. Comparison Models Results

	RNN	*RNN BERT*	*LSTM*	*LSTM BERT*	*GRU*	*GRU BERT*
Epoch	609	190	999	110	999	210
Training Loss	0.00007	0.00132	0.00009	0.00051	0.00019	0.00044
Validation Loss	0.00691	0.00611	0.00421	0.00872	0.00432	0.01014
Testing Loss	0.00973	0.00406	0.00735	0.01194	0.00732	0.00451
Total Loss	0.000957	0.002740	0.00087400	0.00339	0.000956280	0.001596
Time	1 m 23.5 s	15m 54.9s	4m 41.3s	16m 25.7s	4m 53.1s	29m 23.9s

The results of all the experiments performed are shown in Table 3. The ones with 'BERT' in their titles come from the use of titles of the news and their embeddings after passing through BERT. Based on the testing loss, it can be concluded that the best models are RNN BERT with 0.00406 in its testing loss, and the GRU BERT model with 0.00451. These models are the ones that are generalized better. On the other hand, if we check the value of the total loss in the table, it is easy to perceive that the RNN, LSTM, and GRU models attained better performance if we compare them with those models that were using BERT. Furthermore, the process of training the model with a big amount of data was challenging. In some of them, dropouts were introduced in addition to increasing the batch size, and in the last model GRU BERT, even an L2 penalization or weight decay was used from the optimizer.