ELECTRONICS ENGINEERING SOCIETY

MOSAIC '24

A Deep Learning and NLP Based Event

TEAM BABY DRIVERS

ARYANSH KUMAR
MITUL AGARWAL
ADITYA RAJ

PROBLEM STATEMENT 1

CLOSING PRICE PREDICTION

PS 1 WHAT THE PROBLEM STATEMENT ASKED US?

We were provided with historical stock market data and had to analyse this data, identify meaningful patterns, and create a predictive model that can anticipate future closing prices.

COLUMNS:-:
Open
High
Low
Volume
Close
Adjusted Close

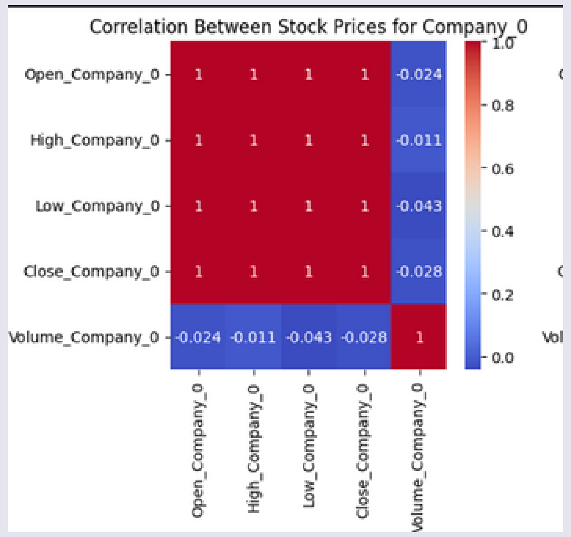
Observation

As we have seen that in Notebook that there is very strong correlation between opening, Low, High and close prices hence dropping Open, High and Low prices columns will be beneficial to avoid muticollinearity.

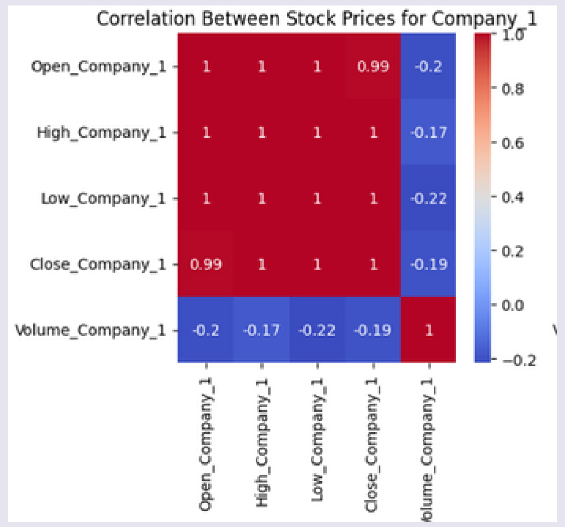
There are no null values as checked in notebook using .info() function

Company_4 has the widest range of close prices, demanding precise model training to minimize RMSE. The table shows no outliers, but a Boxplot will confirm this later.

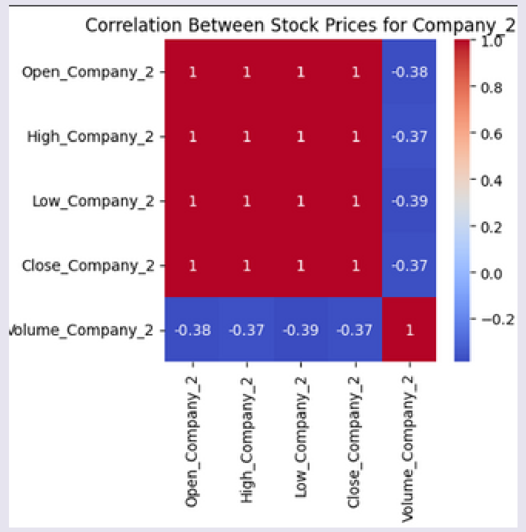
Collinearity Plots



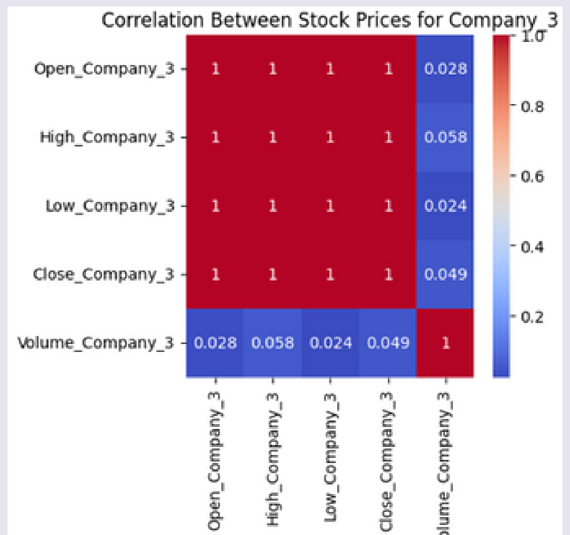
	Open_Company_0	High_Company_0	Low_Company_0	Close_Company_0	Volume_Company_0
Open_Company_0	1	1	1	1	-0.024
High_Company_0	1	1	1	1	-0.011
Low_Company_0	1	1	1	1	-0.043
Close_Company_0	1	1	1	1	-0.028
Volume_Company_0	-0.024	-0.011	-0.043	-0.028	1



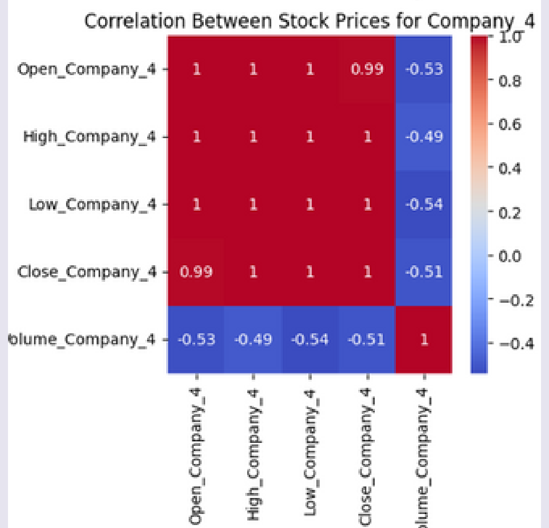
	Open_Company_1	High_Company_1	Low_Company_1	Close_Company_1	Volume_Company_1
Open_Company_1	1	1	1	0.99	-0.2
High_Company_1	1	1	1	1	-0.17
Low_Company_1	1	1	1	1	-0.22
Close_Company_1	0.99	1	1	1	-0.19
Volume_Company_1	-0.2	-0.17	-0.22	-0.19	1



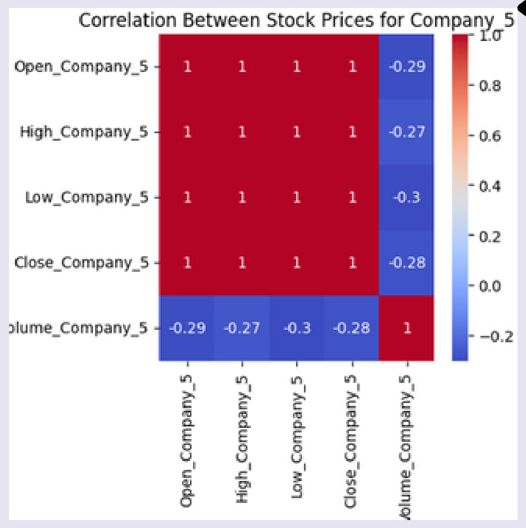
	Open_Company_2	High_Company_2	Low_Company_2	Close_Company_2	Volume_Company_2
Open_Company_2	1	1	1	1	-0.38
High_Company_2	1	1	1	1	-0.37
Low_Company_2	1	1	1	1	-0.39
Close_Company_2	1	1	1	1	-0.37
Volume_Company_2	-0.38	-0.37	-0.39	-0.37	1



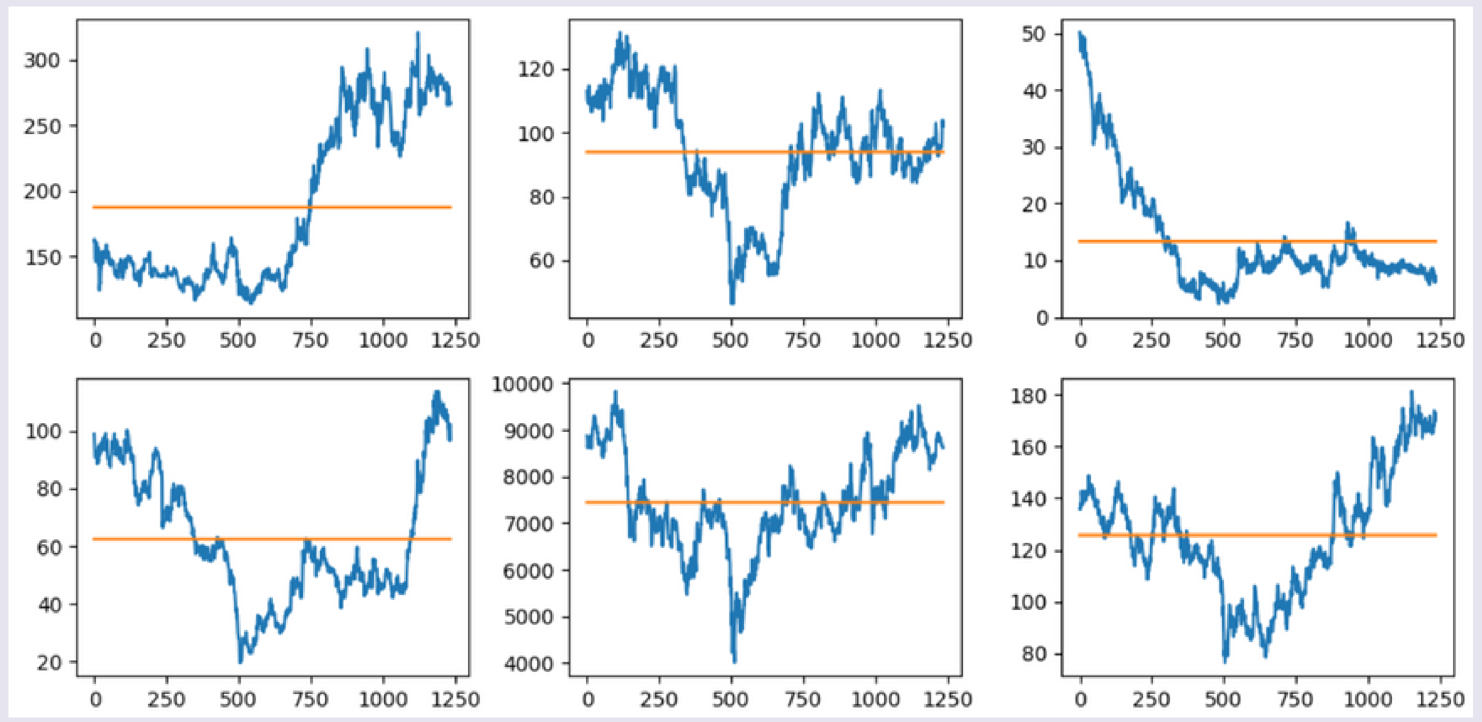
	Open_Company_3	High_Company_3	Low_Company_3	Close_Company_3	Volume_Company_3
Open_Company_3	1	1	1	1	0.028
High_Company_3	1	1	1	1	0.058
Low_Company_3	1	1	1	1	0.024
Close_Company_3	1	1	1	1	0.049
Volume_Company_3	0.028	0.058	0.024	0.049	1



	Open_Company_4	High_Company_4	Low_Company_4	Close_Company_4	Volume_Company_4
Open_Company_4	1	1	1	0.99	-0.53
High_Company_4	1	1	1	1	-0.49
Low_Company_4	1	1	1	1	-0.54
Close_Company_4	0.99	1	1	1	-0.51
Volume_Company_4	-0.53	-0.49	-0.54	-0.51	1



	Open_Company_5	High_Company_5	Low_Company_5	Close_Company_5	Volume_Company_5
Open_Company_5	1	1	1	1	-0.29
High_Company_5	1	1	1	1	-0.27
Low_Company_5	1	1	1	1	-0.3
Close_Company_5	1	1	1	1	-0.28
Volume_Company_5	-0.29	-0.27	-0.3	-0.28	1

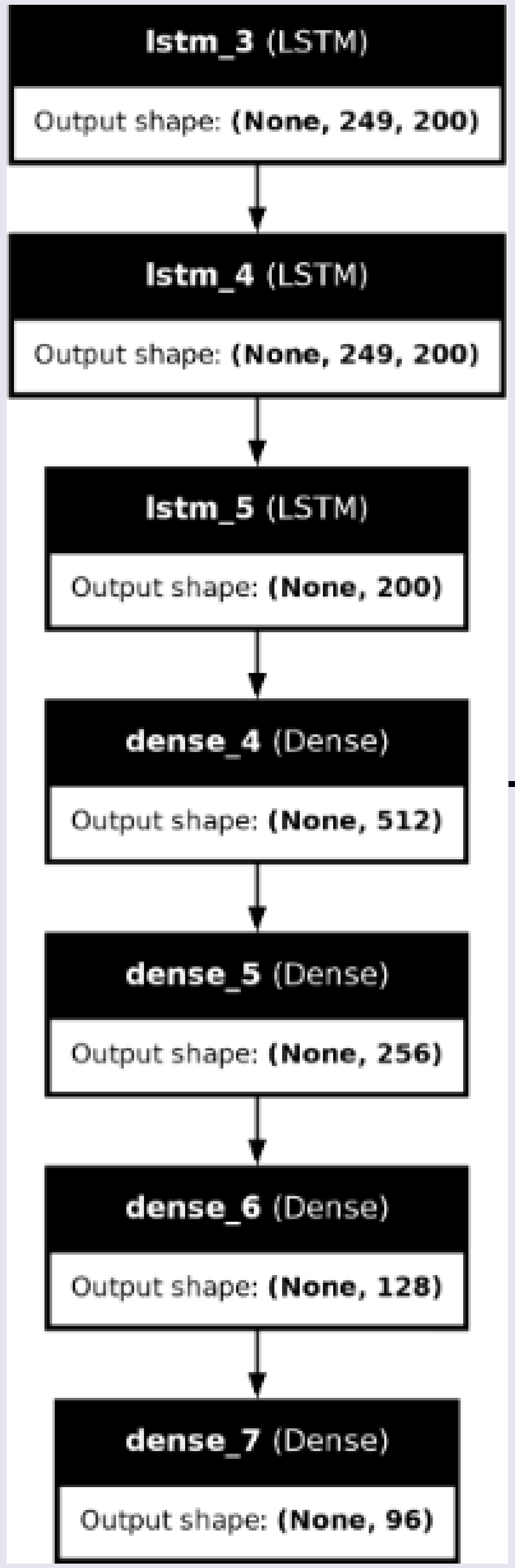


The plot indicates a consistent variation in closing stock prices across various companies and years, with no abrupt increases or decreases. This suggests a stable and gradual trend in the stock prices without major fluctuations.
The box plot confirms the absence of outliers in the data.

- To prepare X and Y for stock price prediction, first, we have defined parameters like n_lookback (249 days) for historical data and n_forecast (96 days) for future predictions.
- Then we have extracted the close column data and calculated the total data points (n).
- Then we have iterated through the data to create X by slicing n_lookback elements for each prediction point and Y by slicing n_forecast elements ensuring X has dimensions [n_lookback, n - n_lookback - n_forecast + 1] and Y has dimensions [n_forecast, n - n_lookback - n_forecast + 1].

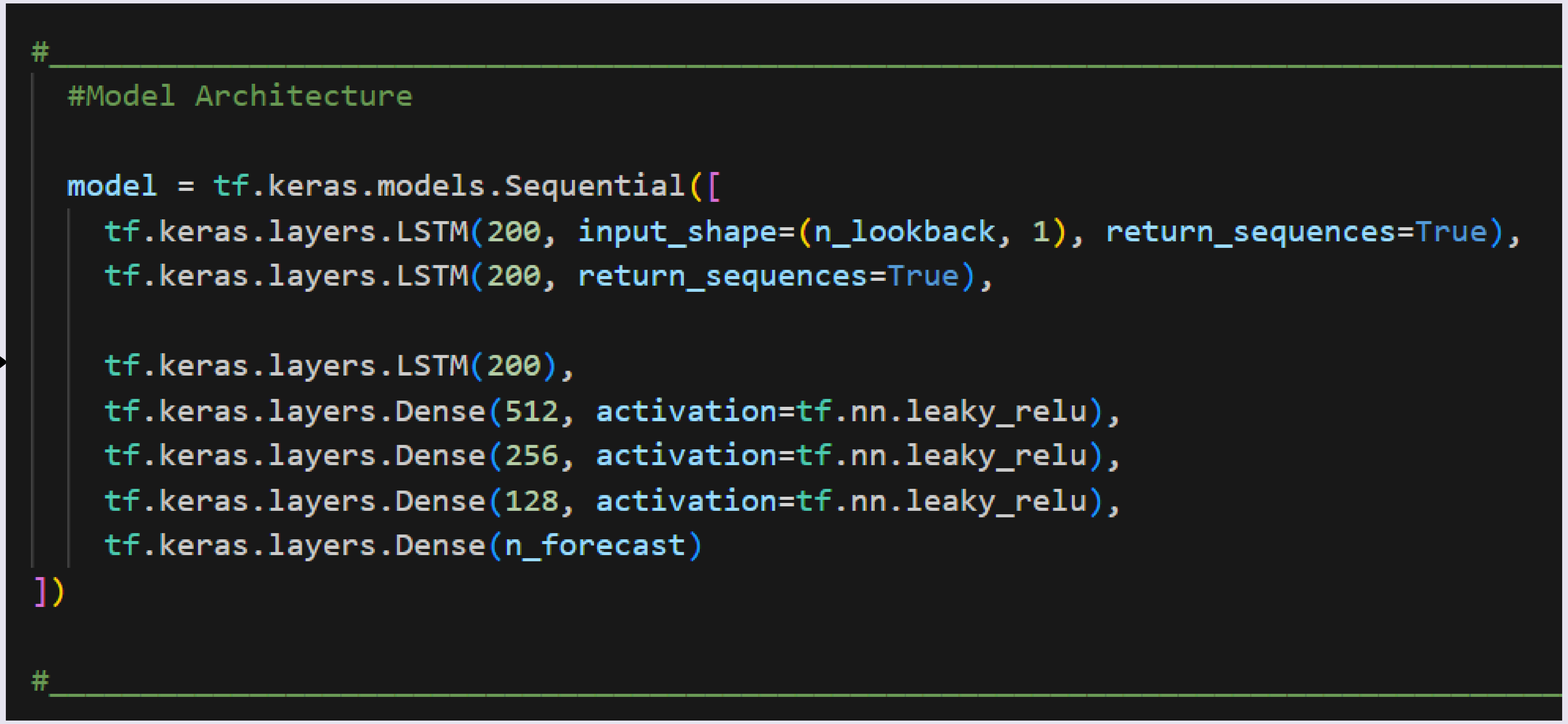
Model Selection

- **Time Series Dependency:**
  Models like GRU and LSTM are vital for time series predictions due to their ability to capture temporal dependencies.
- **Exclusion of GRU:**
  GRU was omitted due to its tendency to predict only incremental or decremental values, resulting in limited variation and lower accuracy.
- **Optimizing Forecasts:**
  After excluding GRU, we experimented with a model comprising three layers of LSTM followed by three layers of Dense layers..


lstm_3 (LSTM)
Output shape: (None, 249, 200)
lstm_4 (LSTM)
Output shape: (None, 249, 200)
lstm_5 (LSTM)
Output shape: (None, 200)
dense_4 (Dense)
Output shape: (None, 512)
dense_5 (Dense)
Output shape: (None, 256)
dense_6 (Dense)
Output shape: (None, 128)
dense_7 (Dense)
Output shape: (None, 96)

Why LSTM ??

- **Handling Long-Term Dependencies:**
  LSTMs have a more elaborate memory cell, allowing them to capture and remember long-term dependencies in sequential data more effectively than GRU's.
- **Explicit Memory Control:**
  LSTMs have separate mechanisms for input, forget, and output, providing more explicit control over the flow of information and memory storage compared to GRUs, which have a more streamlined architecture.
- **Fine-Tuning:**
  LSTMs offer more parameters to fine-tune, which can be advantageous in scenarios where model customization and optimization are essential.



```
#___________________________________________________________________________________
 #Model Architecture

 model = tf.keras.models.Sequential([
   tf.keras.layers.LSTM(200, input_shape=(n_lookback, 1), return_sequences=True),
   tf.keras.layers.LSTM(200, return_sequences=True),

   tf.keras.layers.LSTM(200),
   tf.keras.layers.Dense(512, activation=tf.nn.leaky_relu),
   tf.keras.layers.Dense(256, activation=tf.nn.leaky_relu),
   tf.keras.layers.Dense(128, activation=tf.nn.leaky_relu),
   tf.keras.layers.Dense(n_forecast)
])

#___________________________________________________________________________________
```

Leaky ReLU ensures a smooth gradient throughout the entire input range, including the negative part, which can aid in more stable and effective training of deep neural networks. ReLU, on the other hand, has a gradient of zero for negative inputs, causing abrupt changes in the gradient during training.

JUST LAST FEW STEPS:

1.Model Architecture:

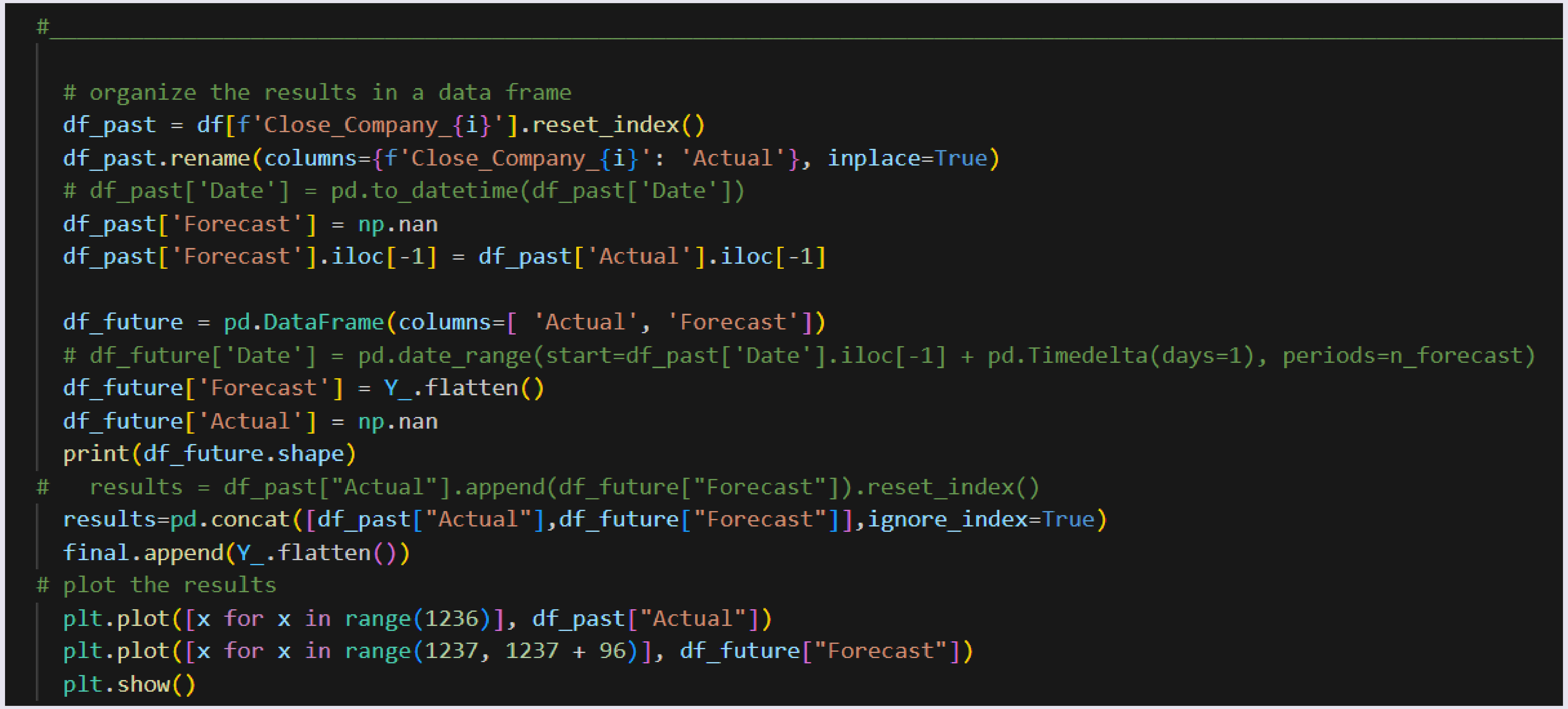
The final model utilized leaky ReLU activation units for improved gradient flow and employed the Adam optimizer for efficient descent during training, enhancing both performance and convergence speed

2.Iterative Training Approach:

Through several iterations of training models, each analyzed with plots to understand their behavior, we fine-tuned hyperparameters and architecture choices, leading to the selection of the final mode

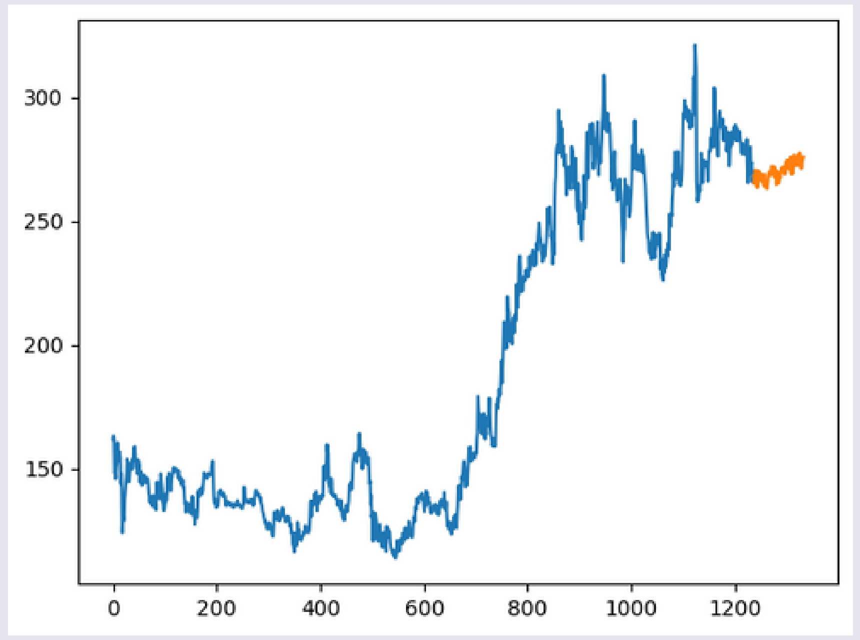
3.Data Handling and Validation:

Data was split into an 80:20 ratio for training and validation, aiming for minimal validation loss across fewer epochs, indicating a well-generalized model that balances accuracy and efficiency.

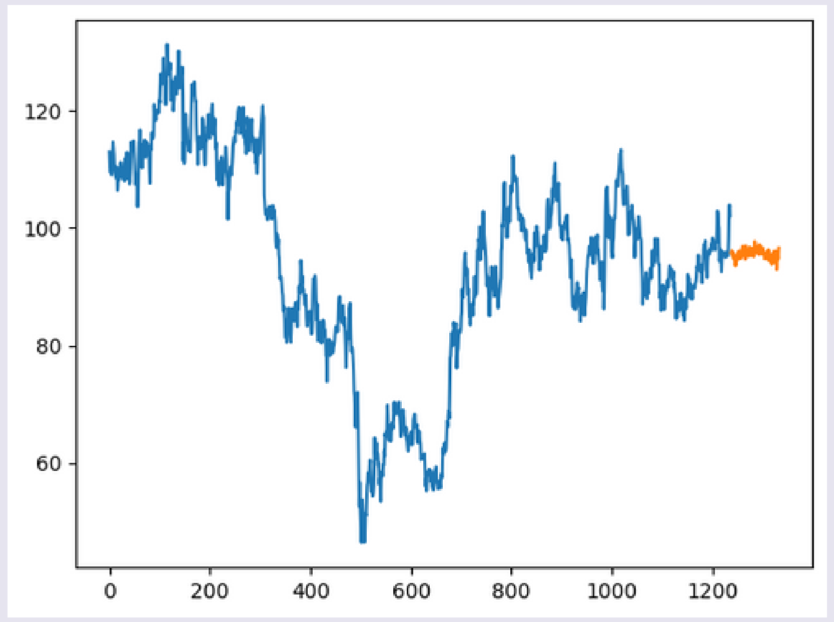


```
#

 # organize the results in a data frame
 df_past = df[f'Close_Company_{i}'].reset_index()
 df_past.rename(columns={f'Close_Company_{i}': 'Actual'}, inplace=True)
 # df_past['Date'] = pd.to_datetime(df_past['Date'])
 df_past['Forecast'] = np.nan
 df_past['Forecast'].iloc[-1] = df_past['Actual'].iloc[-1]

 df_future = pd.DataFrame(columns=[ 'Actual', 'Forecast'])
 # df_future['Date'] = pd.date_range(start=df_past['Date'].iloc[-1] + pd.Timedelta(days=1), periods=n_forecast)
 df_future['Forecast'] = Y_.flatten()
 df_future['Actual'] = np.nan
 print(df_future.shape)
#  results = df_past["Actual"].append(df_future["Forecast"]).reset_index()
 results=pd.concat([df_past["Actual"],df_future["Forecast"]],ignore_index=True)
 final.append(Y_.flatten())
# plot the results
 plt.plot([x for x in range(1236)], df_past["Actual"])
 plt.plot([x for x in range(1237, 1237 + 96)], df_future["Forecast"])
 plt.show()
```

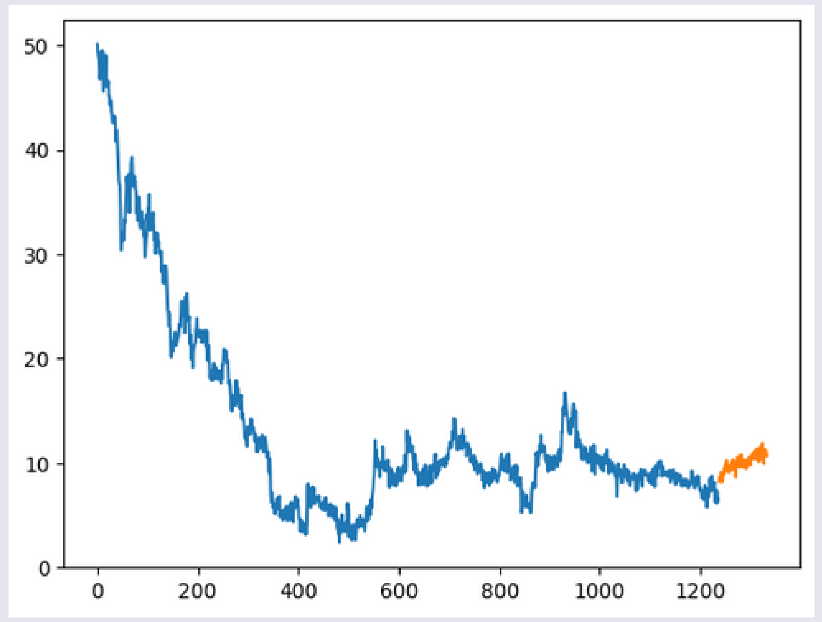
FINAL PREDICTIONS:



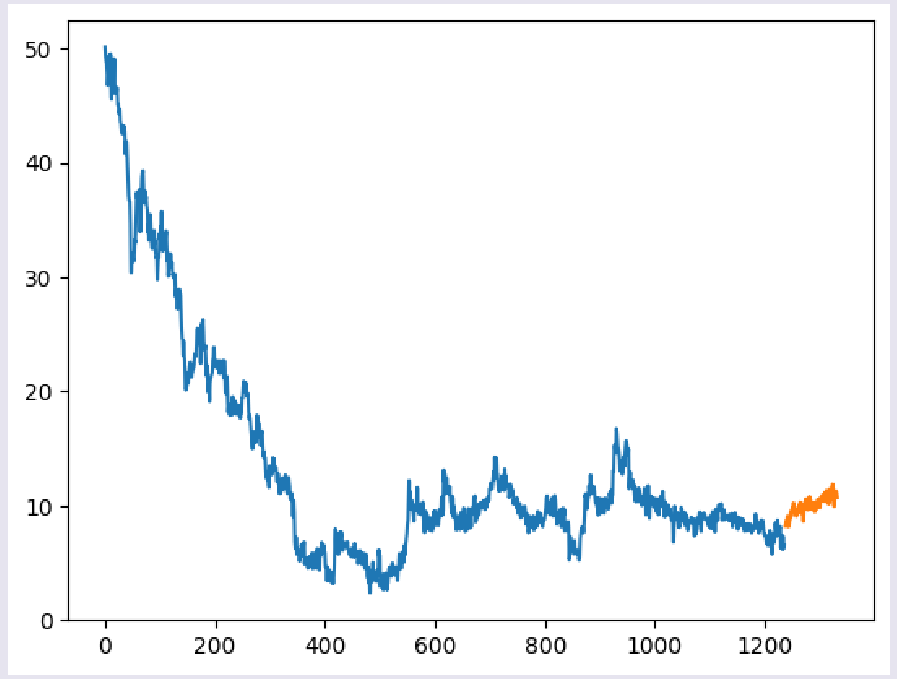
Company_0



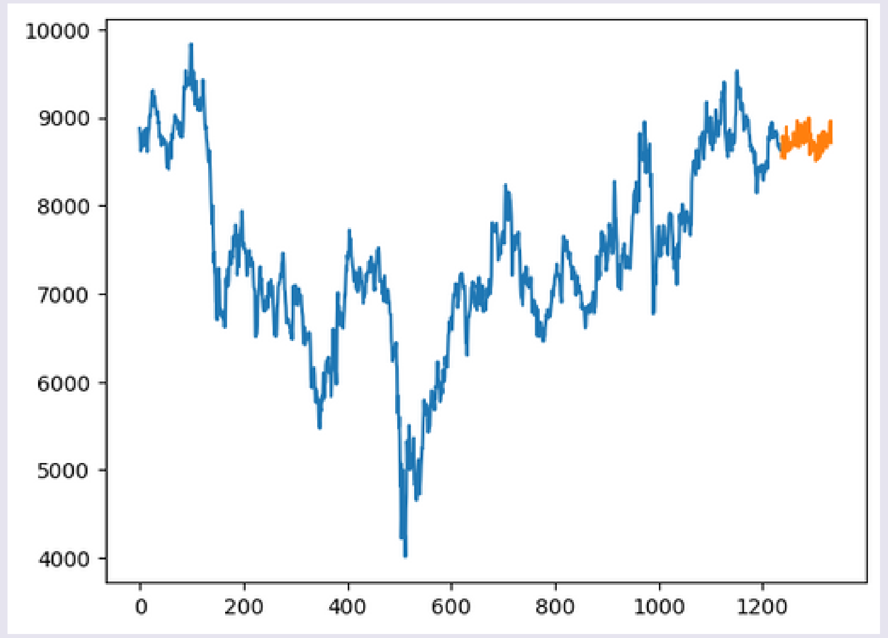
Company_1



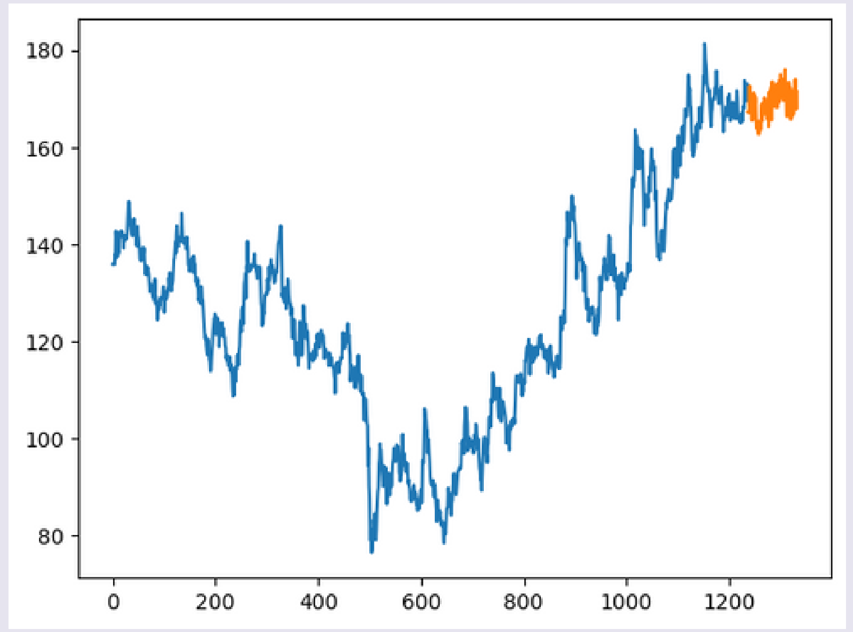
Company_2



Company_3



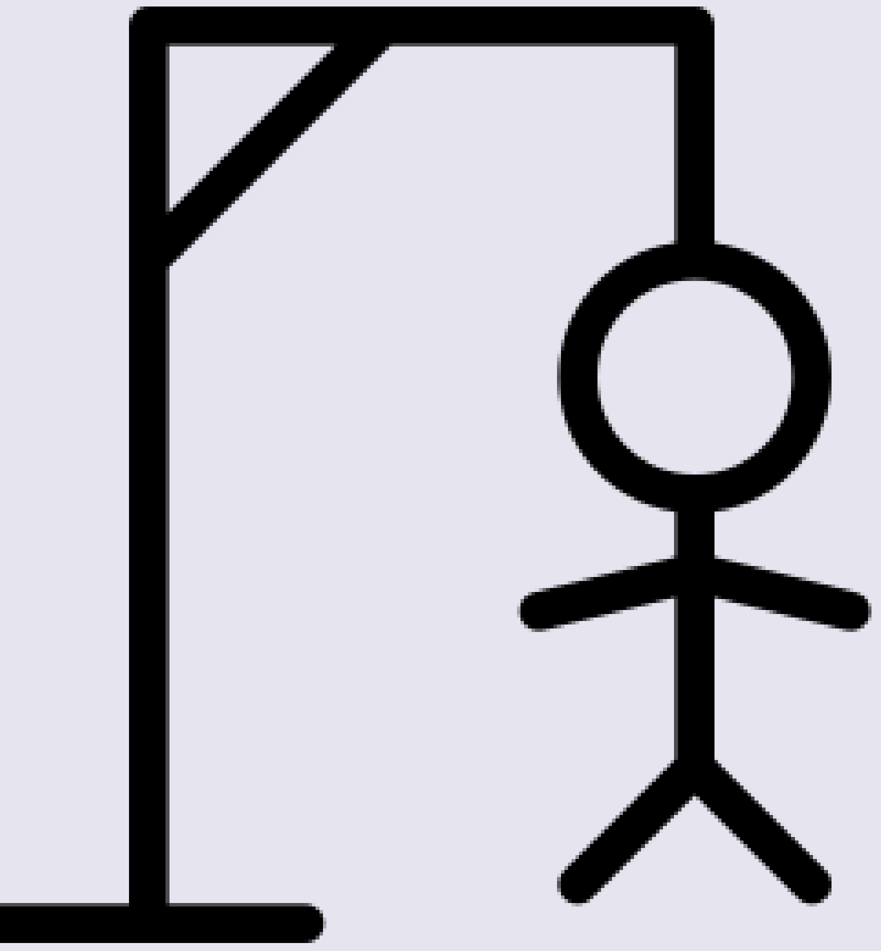
Company_4



Company_5

Public RMSE score of 90.6748

Private RMSE score of 255.2538



PROBLEM STATEMENT 2

THE HANGMAN GAME

PS 2 WHAT THE PROBLEM STATEMENT ASKED US?

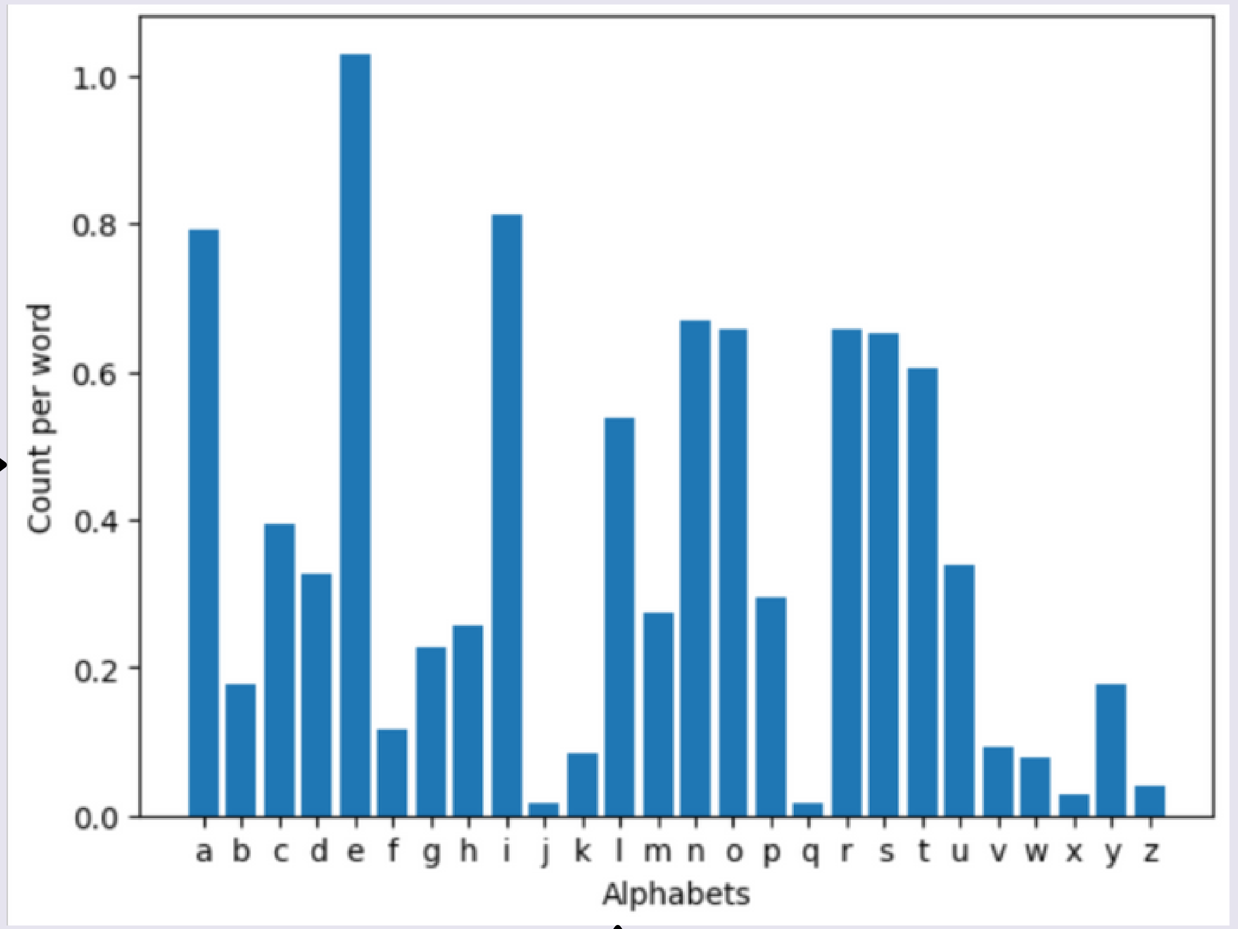
We have to develop an Algorithm to play Hangman game which predict word letter by letter ,predicting characters and we have 6 chances to guess the word

OUR APPROACH

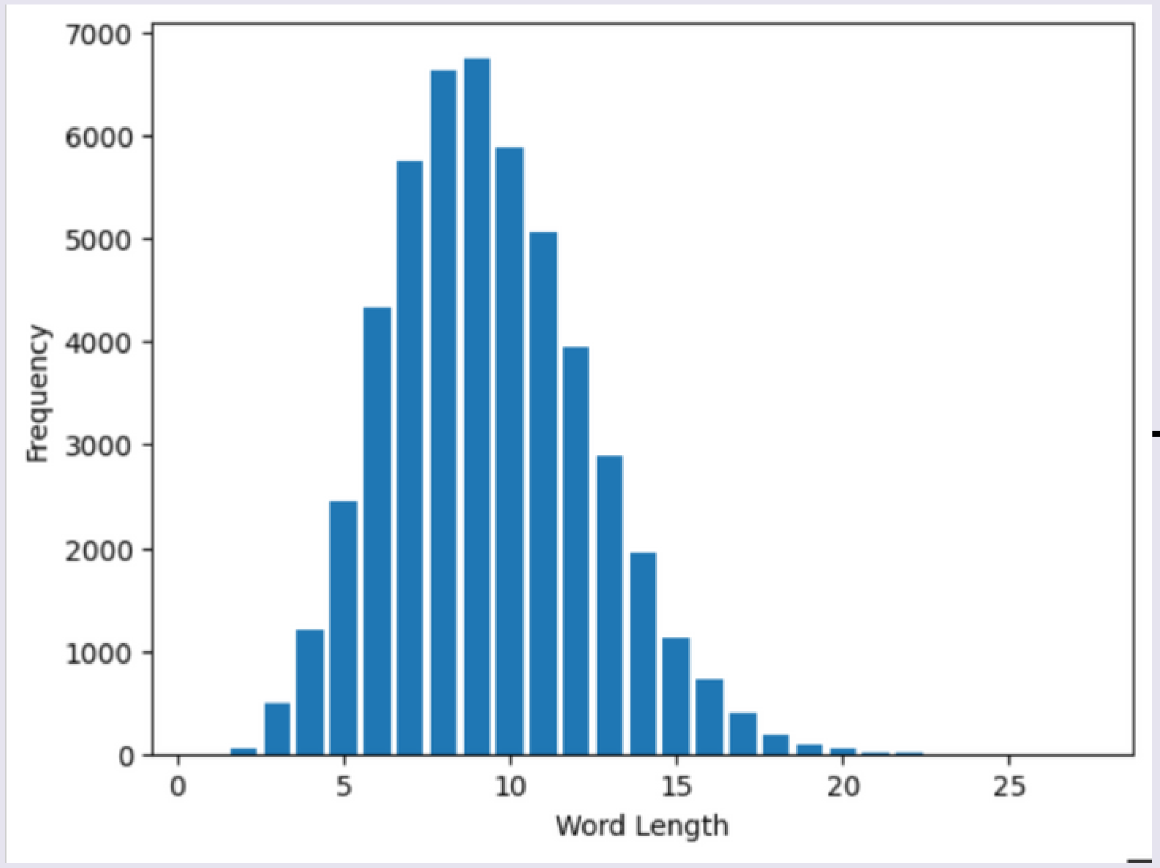
STEP 1:

Frequency model –

- Since in the beginning of the game with very little idea in which direction to go just based on the basis of word length , we choose to fill the **initial 15%** of the word based of the frequency of the occurrence from the given data and also prioritize vowels and made a list with descending frequency of occurrence and prioritizing vowel



Plot of Frequency of each letter

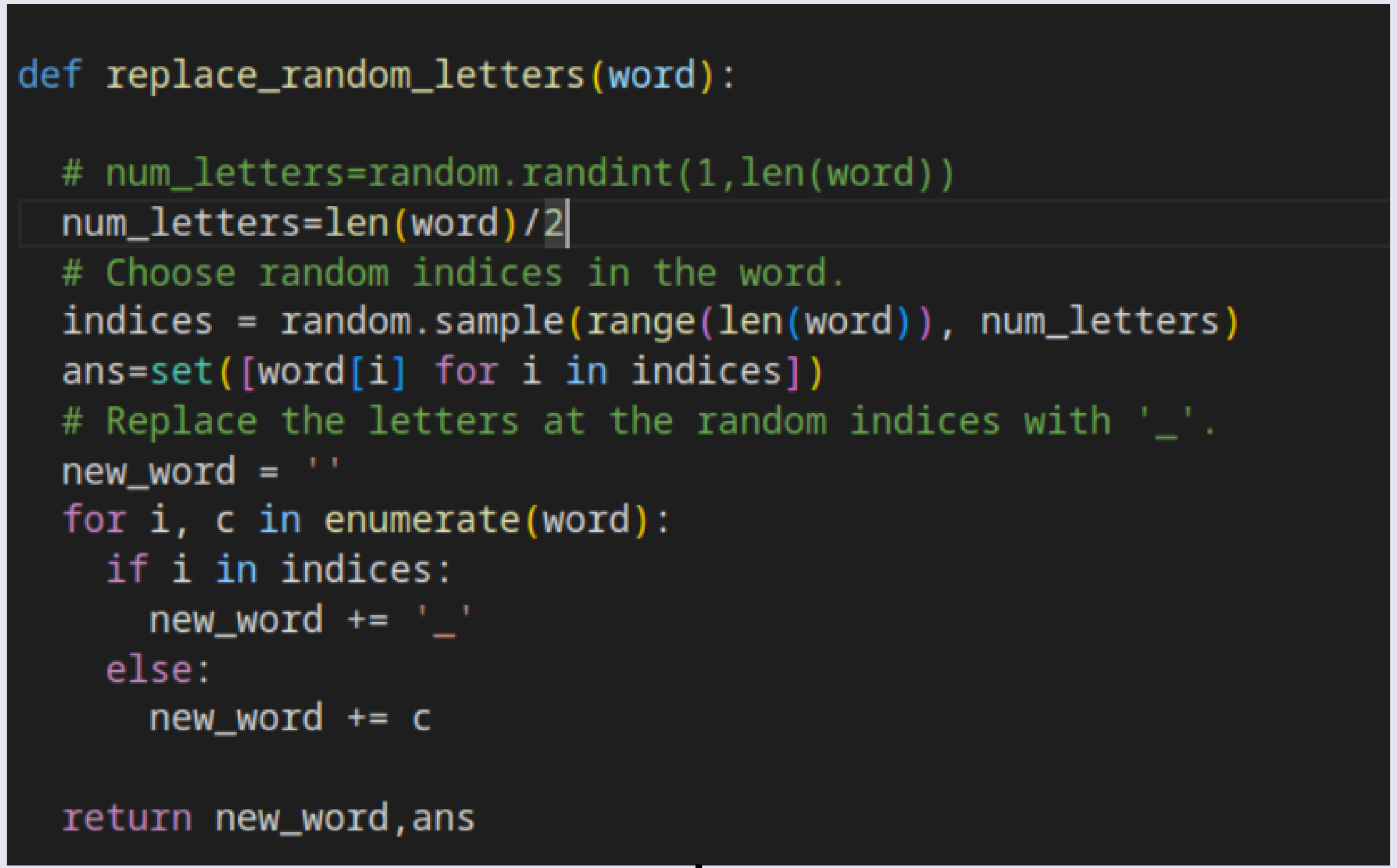


STEP 2:

Statistical Window model

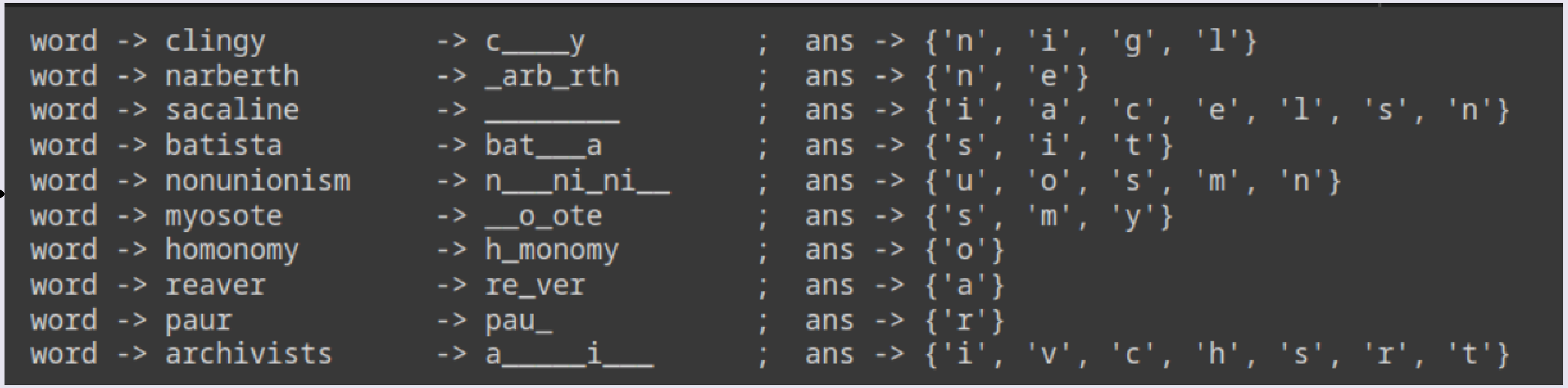
- But our initially based model isn't good for generalization as it doesn't relate to the position and relation of letters with other letters .
- Therefore we have constructed model which takes into account near letters wrt center letter with a window size of 2 like in case of _ _ a _ _ b we would get list of scores of 'a'-'z' for the indices 0,1,3 and 4 wrt 'a' and 3 and 4 wrt 'b' and after that we add up this scores of output the letter with highest scores till **25%** of the word

Masking Given words



```
def replace_random_letters(word):

  # num_letters=random.randint(1,len(word))
  num_letters=len(word)/2
  # Choose random indices in the word.
  indices = random.sample(range(len(word)), num_letters)
  ans=set([word[i] for i in indices])
  # Replace the letters at the random indices with '_'.
  new_word = ''
  for i, c in enumerate(word):
    if i in indices:
      new_word += '_'
    else:
      new_word += c

  return new_word,ans
```

OUTPUT



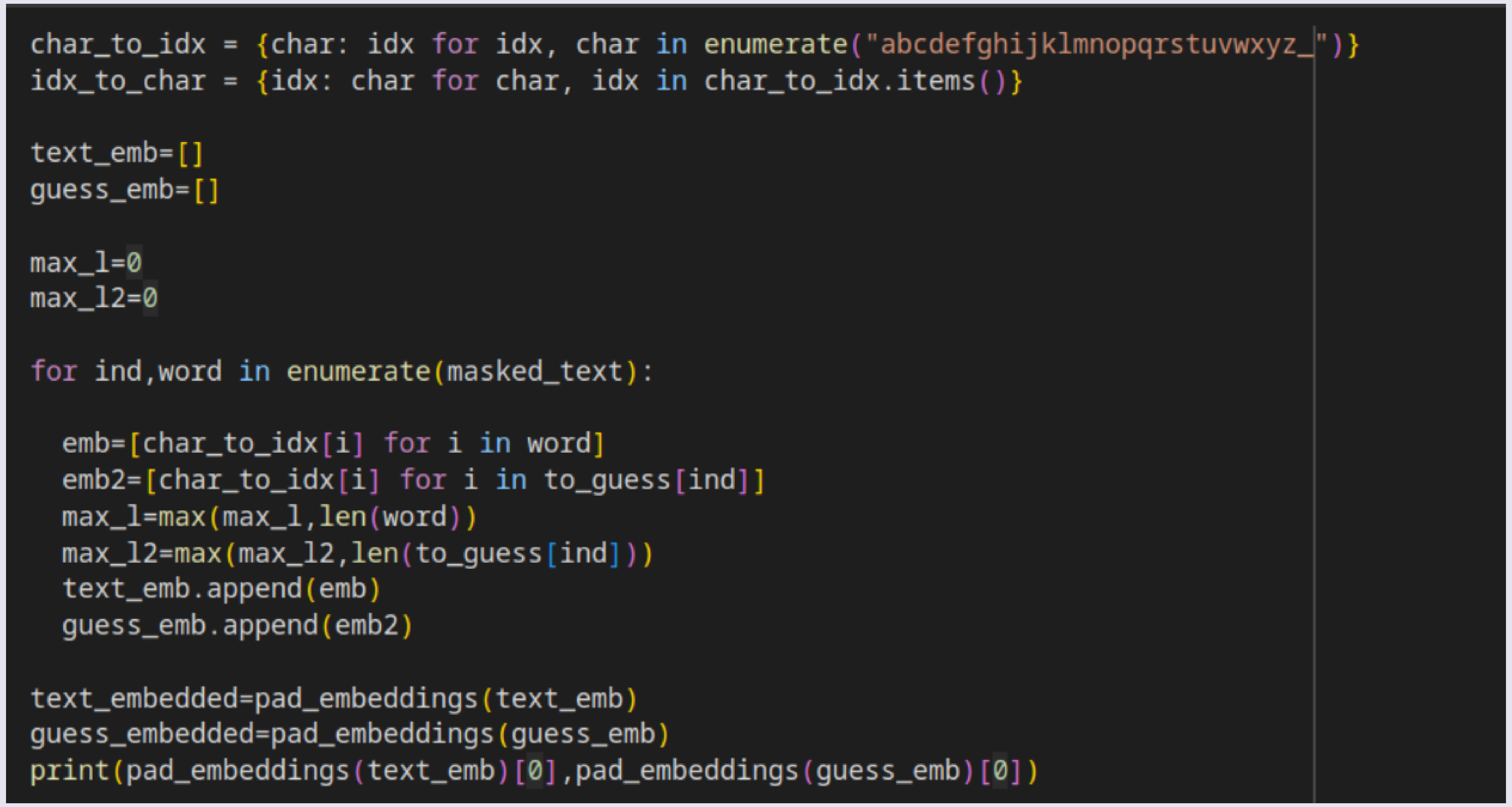
```
word -> clingy            -> c____y         ;  ans -> {'n', 'i', 'g', 'l'}
word -> narberth          -> _arb_rth       ;  ans -> {'n', 'e'}
word -> sacaline          -> ________       ;  ans -> {'i', 'a', 'c', 'e', 'l', 's', 'n'}
word -> batista           -> bat___a        ;  ans -> {'s', 'i', 't'}
word -> nonunionism       -> n___ni_ni__    ;  ans -> {'u', 'o', 's', 'm', 'n'}
word -> myosote           -> __o_ote        ;  ans -> {'s', 'm', 'y'}
word -> homonomy          -> h_monomy       ;  ans -> {'o'}
word -> reaver            -> re_ver         ;  ans -> {'a'}
word -> paur              -> pau_           ;  ans -> {'r'}
word -> archivists        -> a_____i___     ;  ans -> {'i', 'v', 'c', 'h', 's', 'r', 't'}
```

Code Explanation

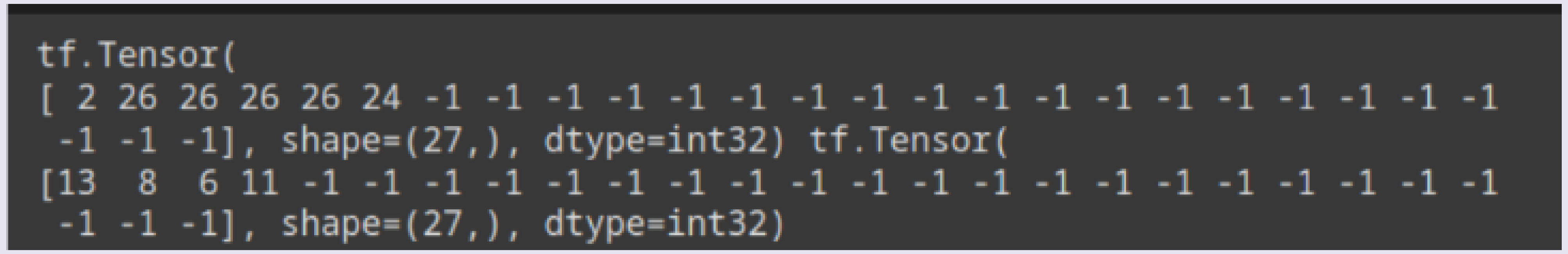
- Input Preparation: The training dataset comprised words with missing letters, denoted by ''. These incomplete words were converted into input data for the model, where each word with missing letters was represented as a sequence with ''.

- Target Output Creation: Corresponding to each incomplete word input, a list of letters was generated to fill in the missing spaces. This list served as the target output for the model, aiming to predict the final output based on the current state of words with missing and found letters.

Vector Embeddings



```
char_to_idx = {char: idx for idx, char in enumerate("abcdefghijklmnopqrstuvwxyz_")}
idx_to_char = {idx: char for char, idx in char_to_idx.items()}

text_emb=[]
guess_emb=[]

max_l=0
max_l2=0

for ind,word in enumerate(masked_text):

  emb=[char_to_idx[i] for i in word]
  emb2=[char_to_idx[i] for i in to_guess[ind]]
  max_l=max(max_l,len(word))
  max_l2=max(max_l2,len(to_guess[ind]))
  text_emb.append(emb)
  guess_emb.append(emb2)

text_embedded=pad_embeddings(text_emb)
guess_embedded=pad_embeddings(guess_emb)
print(pad_embeddings(text_emb)[0],pad_embeddings(guess_emb)[0])
```



```
tf.Tensor(
[ 2 26 26 26 26 24 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1], shape=(27,), dtype=int32) tf.Tensor(
[13  8  6 11 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1], shape=(27,), dtype=int32)
```

Code Explanation

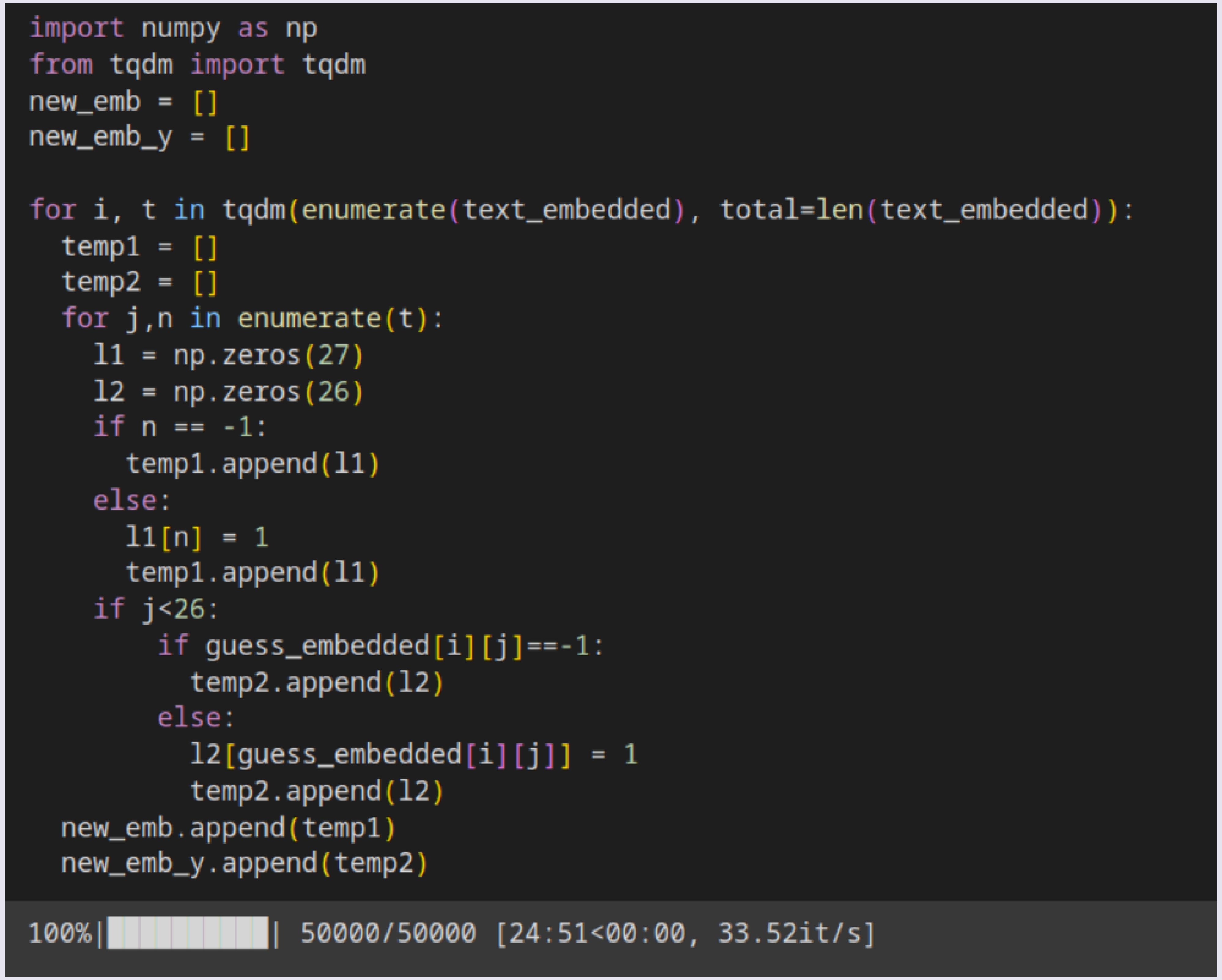
- Word to Vector Embeddings: Each letter in the input words was converted to vector embeddings, with 'a' represented as 1, 'b' as 2, up to 'z' as 26. Additionally, '_' was converted to 27 to ensure consistent input size.

- Fixed Input Length: To maintain a fixed input length for the model, the sizes of all words were matched, ensuring uniformity in the input dimensions.

- One-Hot Encoding for Input: The one-hot encoding technique was applied to each position in the words, representing the presence of different letters. This resulted in input data of size [27x27], indicating the presence or absence of each letter in the word.

- Output Variable Preparation: The output variable 'y' representing missing letters was generated in embedding forms without one-hot encoding, aiming to generate probabilities for each alphabet. The output size was [27x1], indicating the probability distribution for each letter in the missing positions.

Converting Vector Embedding Back Into Letters



```
import numpy as np
from tqdm import tqdm
new_emb = []
new_emb_y = []

for i, t in tqdm(enumerate(text_embedded), total=len(text_embedded)):
  temp1 = []
  temp2 = []
  for j,n in enumerate(t):
    l1 = np.zeros(27)
    l2 = np.zeros(26)
    if n == -1:
      temp1.append(l1)
    else:
      l1[n] = 1
      temp1.append(l1)
    if j<26:
        if guess_embedded[i][j]==-1:
          temp2.append(l2)
        else:
          l2[guess_embedded[i][j]] = 1
          temp2.append(l2)
  new_emb.append(temp1)
  new_emb_y.append(temp2)
```

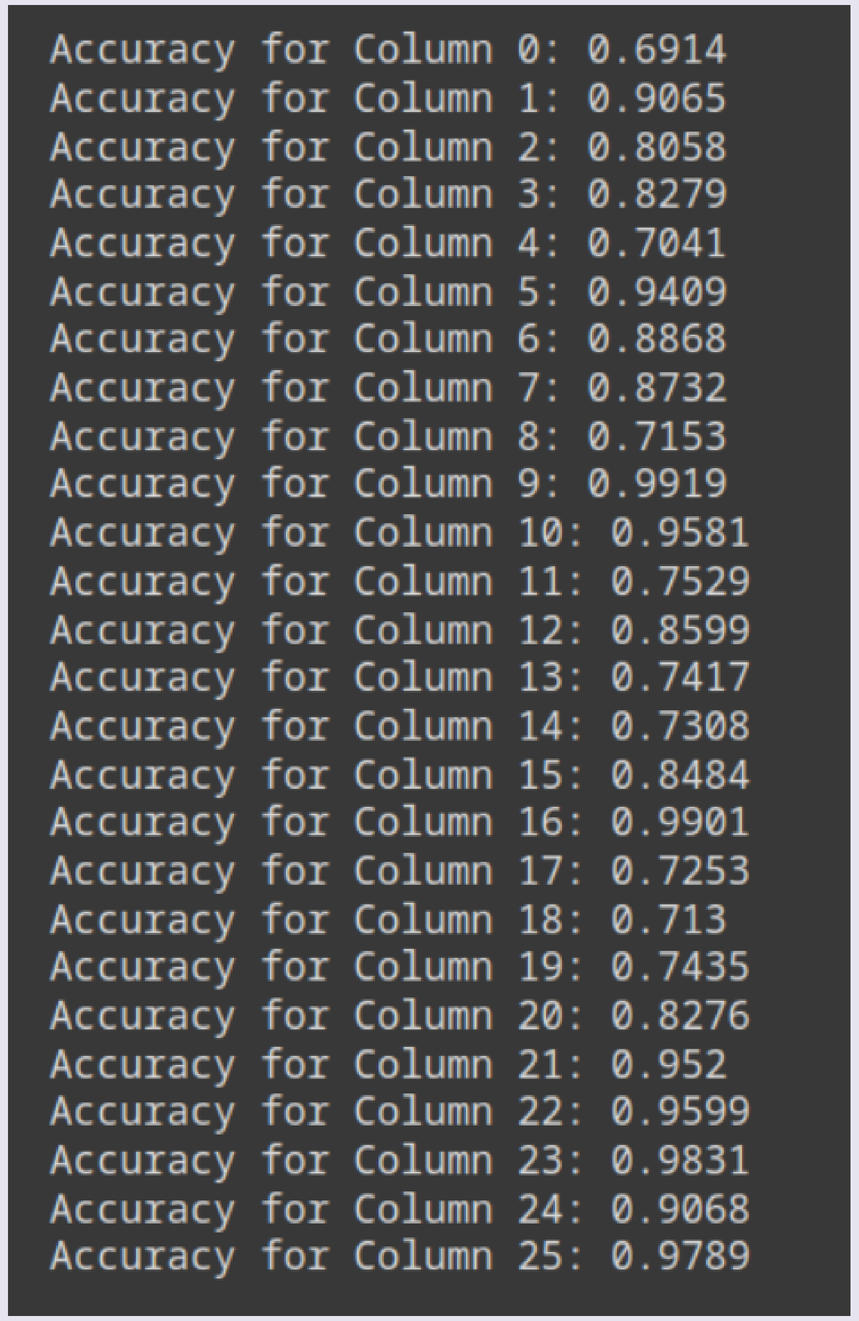
```
100%|██████████| 50000/50000 [24:51<00:00, 33.52it/s]
```

Code Explanation

- After the model predicted the letters to fill in the missing spaces, the vector embeddings were converted back to letters using the idx_to_char function. These guessed letters were determined based on the predicted probabilities obtained from an XGBoost model that was used in conjunction with the main model for more accurate letter predictions. This process ensured that the final output of the model reflected the guessed letters with probabilities calculated by the XGBoost model.

Our Final Approach

Training Outputs and Results:



```
Accuracy for Column 0: 0.6914
Accuracy for Column 1: 0.9065
Accuracy for Column 2: 0.8058
Accuracy for Column 3: 0.8279
Accuracy for Column 4: 0.7041
Accuracy for Column 5: 0.9409
Accuracy for Column 6: 0.8868
Accuracy for Column 7: 0.8732
Accuracy for Column 8: 0.7153
Accuracy for Column 9: 0.9919
Accuracy for Column 10: 0.9581
Accuracy for Column 11: 0.7529
Accuracy for Column 12: 0.8599
Accuracy for Column 13: 0.7417
Accuracy for Column 14: 0.7308
Accuracy for Column 15: 0.8484
Accuracy for Column 16: 0.9901
Accuracy for Column 17: 0.7253
Accuracy for Column 18: 0.713
Accuracy for Column 19: 0.7435
Accuracy for Column 20: 0.8276
Accuracy for Column 21: 0.952
Accuracy for Column 22: 0.9599
Accuracy for Column 23: 0.9831
Accuracy for Column 24: 0.9068
Accuracy for Column 25: 0.9789
```

Model Training Approach:

- Utilized a structure combining frequency analysis, statistical windowing, and fine-tuned XGBoost integration.

Accuracy Evaluation:

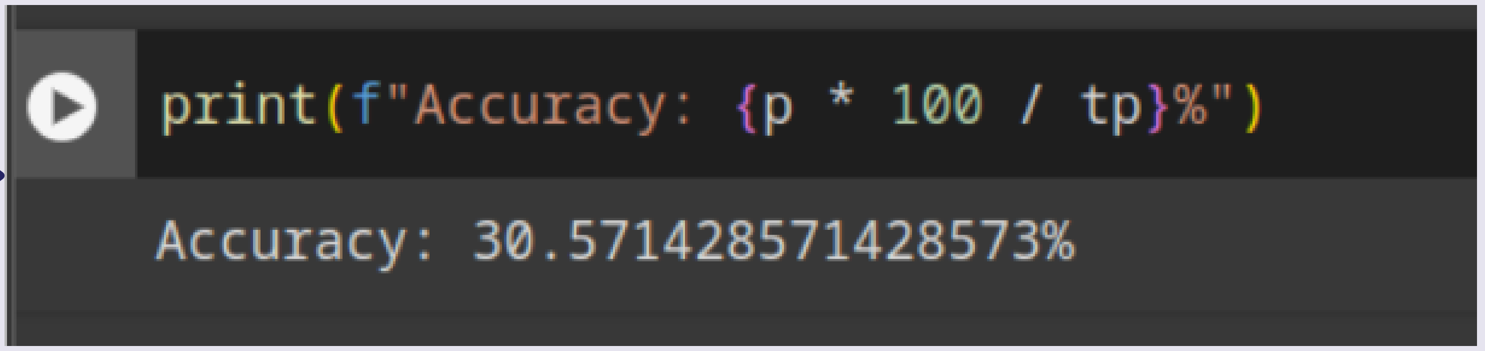
- Printed accuracy metrics to assess the model's proficiency in predicting missing letters for each alphabet.

File Storage for Prediction:

- Saved individual .pkl files, facilitating accurate predictions on words from the test dataset using the trained model.

We employed a phased approach, using the frequency model for the initial 15% and the statistical window model for the next 25% of letters. The remaining letters were accurately predicted using a fine-tuned XGBoost model.

RESULTS:



```
print(f"Accuracy: {p * 100 / tp}%")

Accuracy: 30.571428571428573%
```

THANK YOU