Task 2:

Sentiment Analysis Using Neural Networks

Keniyah Chestnut

Advanced Analytics - D604

SID:012601305

A1: Research Question

Can a neural network be trained to accurately classify IMDB movie reviews as either positive or negative using sentiment analysis techniques? If so, what level of accuracy can we expect from the model?

A2: Objectives or Goals

The objective of this analysis is to build and train a neural network capable of performing sentiment classification on IMDB movie reviews. The aim is to reliably distinguish between positive and negative reviews with a strong level of predictive accuracy.

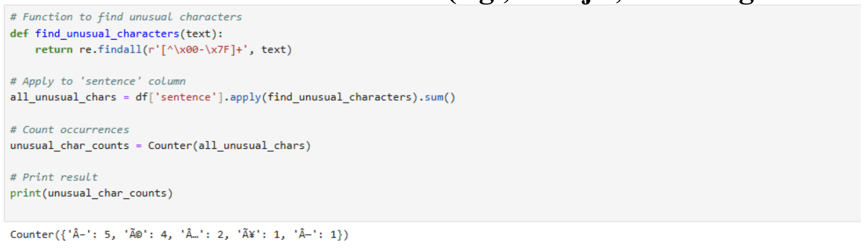
A3: Prescribed Network

This project will utilize a Recurrent Neural Network (RNN), which is well-suited for text classification tasks like sentiment analysis due to its ability to process sequential data effectively.

B1: Data Exploration

Below is a summary of how I explored and prepared the IMDB dataset in accordance with the four required components:

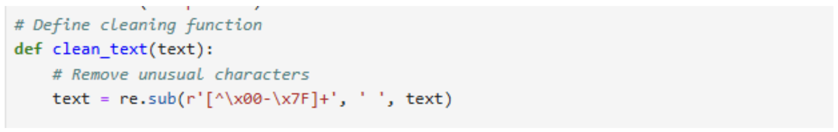
1. Presence of Unusual Characters (e.g., Emojis, Non-English Characters)



```
# Function to find unusual characters
def find_unusual_characters(text):
    return re.findall(r'[^\x00-\x7F]+', text)

# Apply to 'sentence' column
all_unusual_chars = df['sentence'].apply(find_unusual_characters).sum()

# Count occurrences
unusual_char_counts = Counter(all_unusual_chars)

# Print result
print(unusual_char_counts)
```

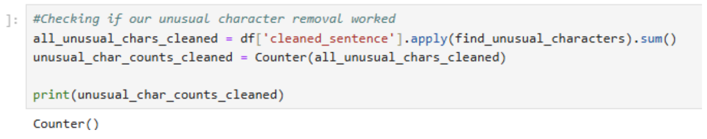
```
Counter({'Â–': 5, 'Ã©': 4, 'Â…': 2, 'Ã¥': 1, 'Â—': 1})
```

To ensure clean input for the neural network, I created a function that detects non-ASCII characters, including emojis and other symbols not typically found in standard English text. Using a counter, I tallied each unusual character. After confirming their presence, I replaced them with spaces, as I could not confidently map them to valid tokens.



```
# Define cleaning function
def clean_text(text):
    # Remove unusual characters
    text = re.sub(r'[^\x00-\x7F]+', ' ', text)
```

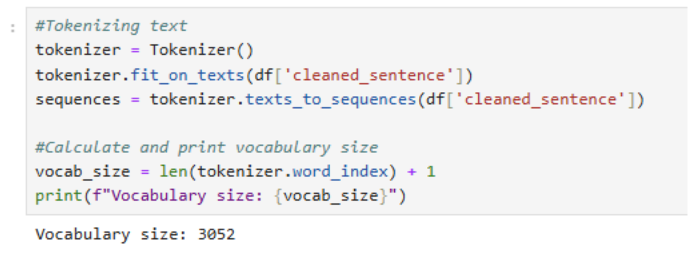
I then re-ran the counter and verified that all non-ASCII characters had been successfully removed from the dataset.



```
#Checking if our unusual character removal worked
all_unusual_chars_cleaned = df['cleaned_sentence'].apply(find_unusual_characters).sum()
unusual_char_counts_cleaned = Counter(all_unusual_chars_cleaned)

print(unusual_char_counts_cleaned)
```

```
Counter()
```

2. Vocabulary Size



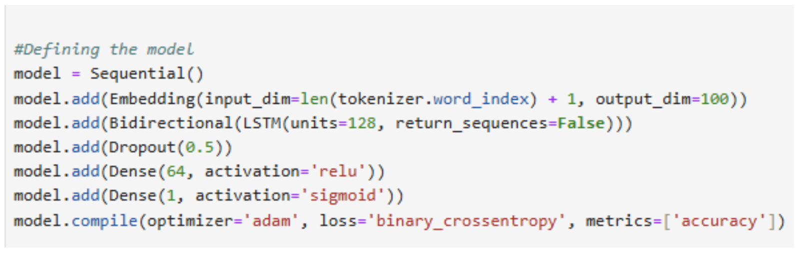
```
#Tokenizing text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['cleaned_sentence'])
sequences = tokenizer.texts_to_sequences(df['cleaned_sentence'])

#Calculate and print vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print(f"Vocabulary size: {vocab_size}")
```

```
Vocabulary size: 3052
```

Tokenization requires understanding the vocabulary size, which reflects the number of unique tokens in the dataset. After fitting the tokenizer on the cleaned text, I calculated the vocabulary size by taking the length of the tokenizer's word index and adding one to account for the padding token. This number is later used to define the input dimension for the embedding layer of the model.

3. Word Embedding Length

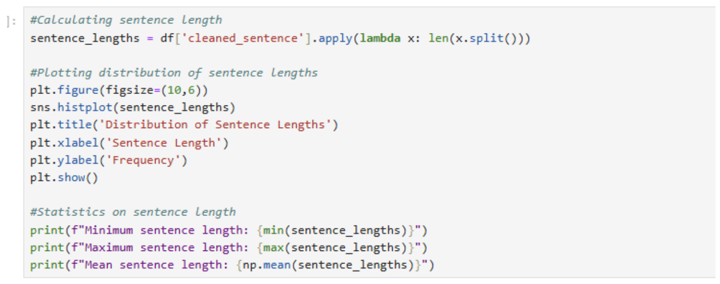
For the embedding layer in the neural network, I selected an embedding length of 100. This dimension was chosen as a balanced default that provides enough room to capture semantic relationships between words, while not adding excessive complexity that could lead to overfitting or extended training time.



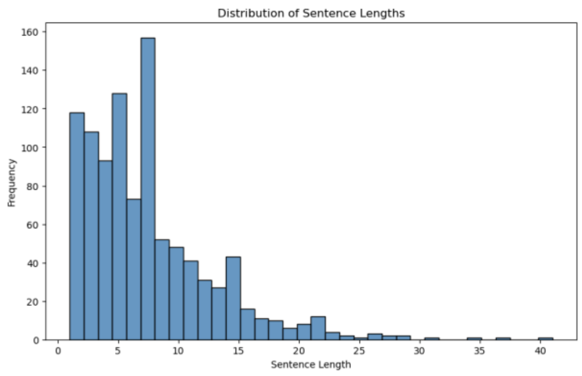
```
#Defining the model
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100))
model.add(Bidirectional(LSTM(units=128, return_sequences=False)))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

4. Statistical Justification for the Chosen Maximum Sequence Length

To determine an appropriate maximum sequence length, I calculated and visualized the distribution of sentence lengths in the dataset.



```
#Calculating sentence length
sentence_lengths = df['cleaned_sentence'].apply(lambda x: len(x.split()))

#Plotting distribution of sentence lengths
plt.figure(figsize=(10,6))
sns.histplot(sentence_lengths)
plt.title('Distribution of Sentence Lengths')
plt.xlabel('Sentence Length')
plt.ylabel('Frequency')
plt.show()

#Statistics on sentence length
print(f"Minimum sentence length: {min(sentence_lengths)}")
print(f"Maximum sentence length: {max(sentence_lengths)}")
print(f"Mean sentence length: {np.mean(sentence_lengths)}")
```



```
Minimum sentence length: 1
Maximum sentence length: 41
Mean sentence length: 7.542
```

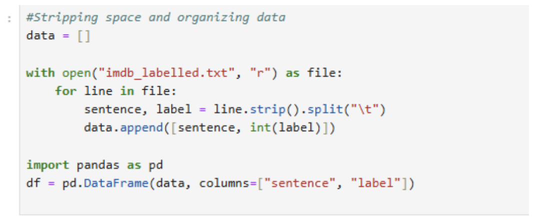
The majority of the reviews were between 2 and 15 words long, with the longest review containing 41 words. To cover the entire range and maintain consistency during training, I selected a maximum sequence length of 50 for padding. This ensures that all sequences, including the longest ones, are included without truncation.

B2: Tokenization

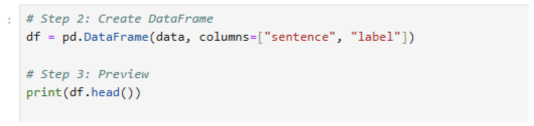
According to the Sentiment Analysis in Python course by Datacamp (Misheva, n.d.), the primary goal of tokenization is to split sentences into smaller parts called tokens so the text can be processed more efficiently by the neural network. Before tokenization, I normalized the text by converting everything to lowercase, removing punctuation, stripping out non-standard characters, and reducing extra whitespace. This ensures that words like "Good" and "good" are treated the same and avoids unnecessary complexity.

I then used the Keras Tokenizer class to build a vocabulary and convert each cleaned sentence into a sequence of integers. This step translates raw text into a numerical format that the model can understand. It also helps streamline the data to improve training performance and accuracy.

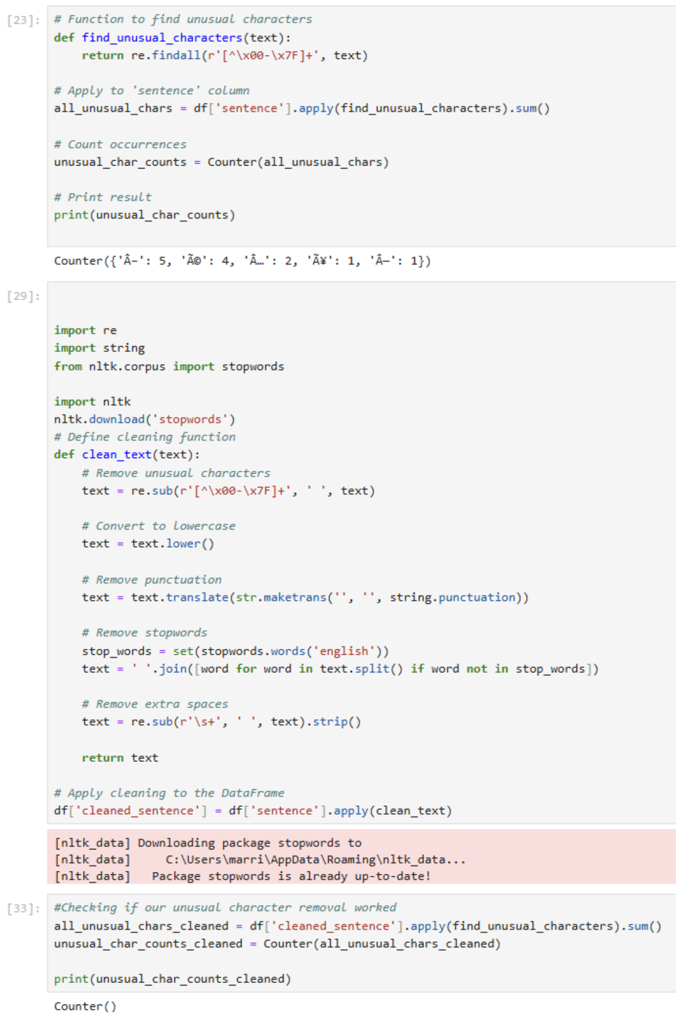
Here is the code I used for tokenization:



```python
#Stripping space and organizing data
data = []

with open("imdb_labelled.txt", "r") as file:
    for line in file:
        sentence, label = line.strip().split("\t")
        data.append([sentence, int(label)])

import pandas as pd
df = pd.DataFrame(data, columns=["sentence", "label"])
```



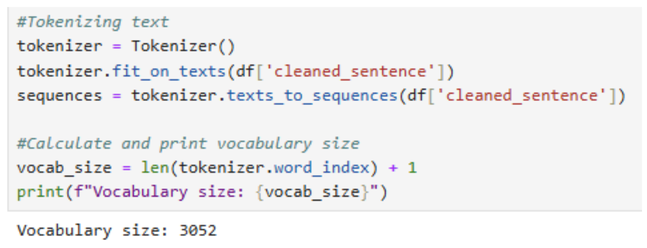
```python
# Step 2: Create DataFrame
df = pd.DataFrame(data, columns=["sentence", "label"])

# Step 3: Preview
print(df.head())
```



```
[23]: # Function to find unusual characters
      def find_unusual_characters(text):
          return re.findall(r'[^\x00-\x7F]+', text)

      # Apply to 'sentence' column
      all_unusual_chars = df['sentence'].apply(find_unusual_characters).sum()

      # Count occurrences
      unusual_char_counts = Counter(all_unusual_chars)

      # Print result
      print(unusual_char_counts)

      Counter({'Â–': 5, 'Ã©': 4, 'Â…': 2, 'Ã¥': 1, 'Â—': 1})
```

```
[29]:


      import re
      import string
      from nltk.corpus import stopwords

      import nltk
      nltk.download('stopwords')
      # Define cleaning function
      def clean_text(text):
          # Remove unusual characters
          text = re.sub(r'[^\x00-\x7F]+', ' ', text)

          # Convert to lowercase
          text = text.lower()

          # Remove punctuation
          text = text.translate(str.maketrans('', '', string.punctuation))

          # Remove stopwords
          stop_words = set(stopwords.words('english'))
          text = ' '.join([word for word in text.split() if word not in stop_words])

          # Remove extra spaces
          text = re.sub(r'\s+', ' ', text).strip()

          return text

      # Apply cleaning to the DataFrame
      df['cleaned_sentence'] = df['sentence'].apply(clean_text)

      [nltk_data] Downloading package stopwords to
      [nltk_data]     C:\Users\marri\AppData\Roaming\nltk_data...
      [nltk_data]   Package stopwords is already up-to-date!
```

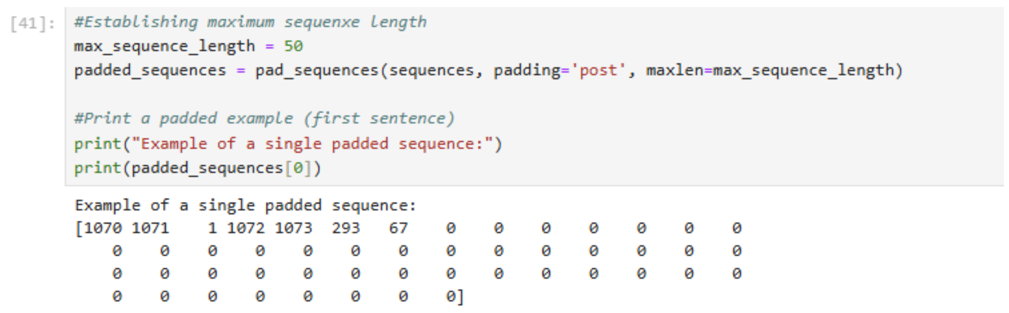
```
[33]: #Checking if our unusual character removal worked
      all_unusual_chars_cleaned = df['cleaned_sentence'].apply(find_unusual_characters).sum()
      unusual_char_counts_cleaned = Counter(all_unusual_chars_cleaned)

      print(unusual_char_counts_cleaned)

      Counter()
```



```
#Tokenizing text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['cleaned_sentence'])
sequences = tokenizer.texts_to_sequences(df['cleaned_sentence'])

#Calculate and print vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print(f"Vocabulary size: {vocab_size}")

Vocabulary size: 3052
```

B3: Padding Process

Padding is necessary to make sure that all input sequences are the same length. Neural networks require inputs to be uniform in size so they can be processed in batches. If a review is shorter than the maximum sequence length, padding adds zeros to the end of the sequence. This standardization allows the model to train consistently across all samples. Here is the code I used to perform padding, along with a sample output:

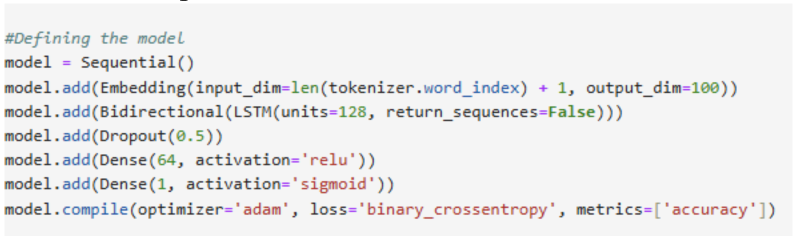

```
[41]: #Establishing maximum sequenxe length
      max_sequence_length = 50
      padded_sequences = pad_sequences(sequences, padding='post', maxlen=max_sequence_length)

      #Print a padded example (first sentence)
      print("Example of a single padded sequence:")
      print(padded_sequences[0])
```

```
Example of a single padded sequence:
[1070 1071    1 1072 1073  293   67    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0]
```

As you can see in the example, shorter sequences are padded at the end with zeros until they reach the defined maximum length.

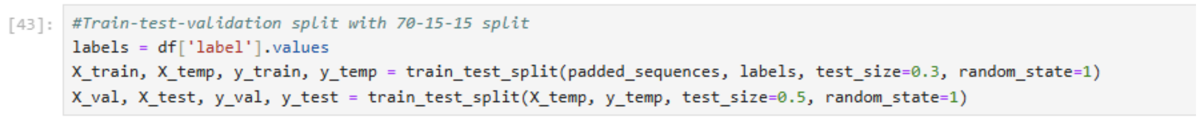
B4: Categories of Sentiment

In this analysis, we are classifying IMDB reviews into two categories: positive and negative. Since there are only two classes, this is a binary classification task. I used the Sigmoid activation function in the final dense layer of the network. According to IBM's website (Stryker, 2024), the Sigmoid function is commonly used for binary classification because it outputs values between 0 and 1, which makes it well-suited for tasks like this.


```
#Defining the model
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100))
model.add(Bidirectional(LSTM(units=128, return_sequences=False)))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

B5: Steps to Prepare the Data

The data preparation process involved several key steps. First, I searched for unusual characters and removed them. Then, I calculated the vocabulary size to define the embedding layer dimensions. Next, I tokenized the sentences so they could be converted to sequences of numbers. After tokenization, I padded all sequences to ensure a consistent input length. Finally, I split the dataset into training, testing, and validation sets using a 70-15-15 split. This approach provides enough data for the model to learn from while still preserving enough data for validation and testing. The validation set will also be useful when we implement early stopping during training.

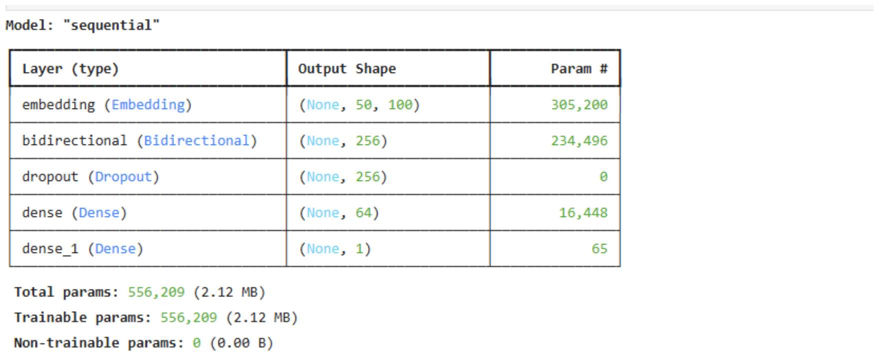

```
[43]: #Train-test-validation split with 70-15-15 split
      labels = df['label'].values
      X_train, X_temp, y_train, y_temp = train_test_split(padded_sequences, labels, test_size=0.3, random_state=1)
      X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=1)
```

B6: Prepared Dataset

A cleaned and tokenized version of the IMDB dataset was saved and will be included as part of the submission. It includes the cleaned sentence and corresponding sentiment label for each review.

C1: Model Summary

Here is a screenshot of the model summary:



```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 50, 100)	305,200
bidirectional (Bidirectional)	(None, 256)	234,496
dropout (Dropout)	(None, 256)	0
dense (Dense)	(None, 64)	16,448
dense_1 (Dense)	(None, 1)	65

```
Total params: 556,209 (2.12 MB)
Trainable params: 556,209 (2.12 MB)
Non-trainable params: 0 (0.00 B)
```

C2: Network Architecture

The model includes five primary layers. Below is a breakdown of each layer type and the reasoning behind its inclusion:

1. Embedding Layer: This layer converts each word in the sequence into a dense vector of size 100. The total number of parameters in this layer is 305,200, based on the vocabulary size multiplied by the embedding dimension. This layer allows the model to learn the context and meaning of words in relation to each other.
2. Bidirectional LSTM Layer: This layer processes the input in both forward and backward directions, enabling the network to capture patterns that may appear at the start or end of the sentence. It has 234,496 parameters and helps improve performance on sequential data like text.
3. Dropout Layer: This layer randomly disables 50 percent of neurons during training to reduce overfitting. It has no trainable parameters and improves the model's generalization to unseen data.

4. Dense Layer: This fully connected layer consists of 64 neurons. With 16,448 parameters, it allows the model to combine and transform features learned in previous layers into more abstract representations useful for classification.
5. Output Dense Layer: This final layer consists of a single neuron with a Sigmoid activation function. It outputs a probability value between 0 and 1, which corresponds to negative or positive sentiment.

C3: Hyperparameters

Below are the selected hyperparameters along with the justification for each one:

1. **Activation Functions**

a. ReLU is used in the dense layer to introduce non-linearity and improve computational efficiency, as supported by IBM (Stryker, 2024).

b. Sigmoid is used in the output layer to produce a binary result between 0 and 1, making it suitable for binary classification problems like sentiment analysis.

2.Number of Nodes per Layer

a. The Embedding Layer uses 100 dimensions to ensure word meanings are captured without being overly complex.

b. The Bidirectional Layer contains 128 units, a common choice that balances performance and resource usage.

c. The Dropout Layer has no units because its function is to prevent overfitting rather than learn features.

d. The Dense Layer has 64 units, which is sufficient to model the underlying relationships in the text without overfitting.

e. The Output Layer has 1 unit to produce a single probability value.

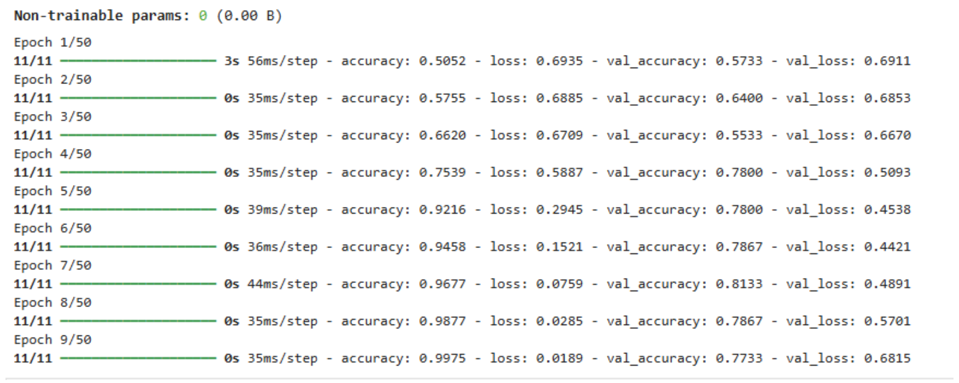
3.Loss Function : Binary Crossentropy is used because the task involves predicting one of two classes. It is ideal for binary classification and interprets the output as probability.

4. Optimizer: Adam is chosen as the optimizer due to its adaptive learning rate and strong performance across many types of deep learning problems. It helps achieve faster convergence with fewer resources.

5. Stopping Criteria: EarlyStopping is used with patience=3 to prevent overfitting. If the validation loss does not improve for three consecutive epochs, training stops automatically. This ensures the model does not continue to train past the point of optimal performance on unseen data.

D1: Stopping Criteria

As described earlier, I implemented EarlyStopping with patience=3 to prevent the model from continuing to train after the validation loss stopped improving. This technique monitors the validation set and stops training after three consecutive increases in loss.

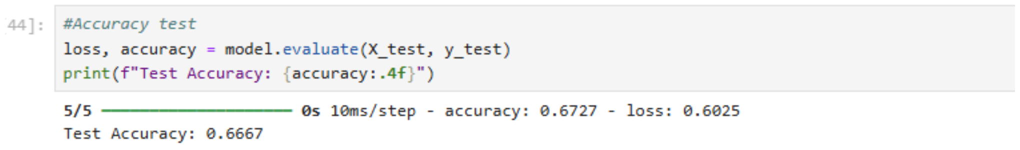


```
Non-trainable params: 0 (0.00 B)
Epoch 1/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 3s 56ms/step - accuracy: 0.5052 - loss: 0.6935 - val_accuracy: 0.5733 - val_loss: 0.6911
Epoch 2/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.5755 - loss: 0.6885 - val_accuracy: 0.6400 - val_loss: 0.6853
Epoch 3/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.6620 - loss: 0.6709 - val_accuracy: 0.5533 - val_loss: 0.6670
Epoch 4/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.7539 - loss: 0.5887 - val_accuracy: 0.7800 - val_loss: 0.5093
Epoch 5/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - accuracy: 0.9216 - loss: 0.2945 - val_accuracy: 0.7800 - val_loss: 0.4538
Epoch 6/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - accuracy: 0.9458 - loss: 0.1521 - val_accuracy: 0.7867 - val_loss: 0.4421
Epoch 7/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 44ms/step - accuracy: 0.9677 - loss: 0.0759 - val_accuracy: 0.8133 - val_loss: 0.4891
Epoch 8/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.9877 - loss: 0.0285 - val_accuracy: 0.7867 - val_loss: 0.5701
Epoch 9/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.9975 - loss: 0.0189 - val_accuracy: 0.7733 - val_loss: 0.6815
```

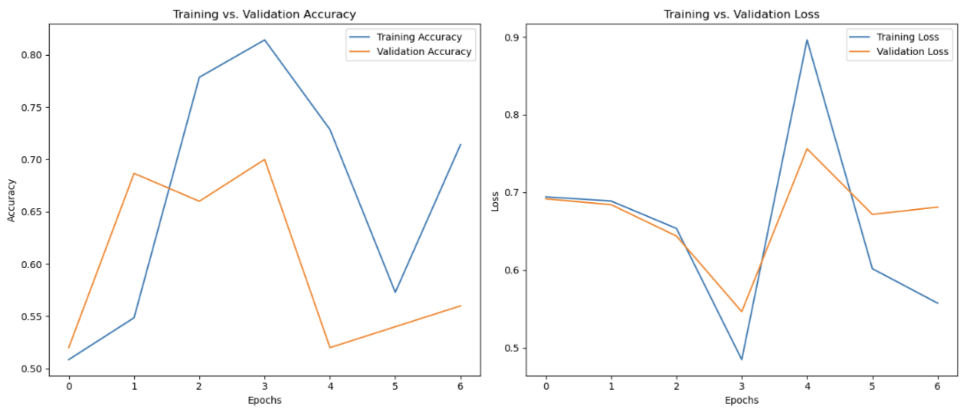
The screenshot included in the submission shows that training was halted automatically once this condition was met. This helped prevent overfitting and ensured that the model did not learn patterns that are too specific to the training data.

D2: Fitness

The final model achieved an accuracy of approximately 66.67% on the test set. This suggests that the model has learned meaningful patterns from the training data but still has room for improvement. To evaluate the model's fitness and detect overfitting or underfitting, I plotted both the training and validation accuracy and loss across each epoch. The plots show that training accuracy steadily increased while validation accuracy peaked and then slightly declined, which may indicate some overfitting. Likewise, validation loss began to rise while training loss continued to decrease. To help prevent overfitting, I applied early stopping with a patience value of 3 and included a dropout layer in the model architecture. These steps limited overtraining and allowed the model to better generalize to new data.

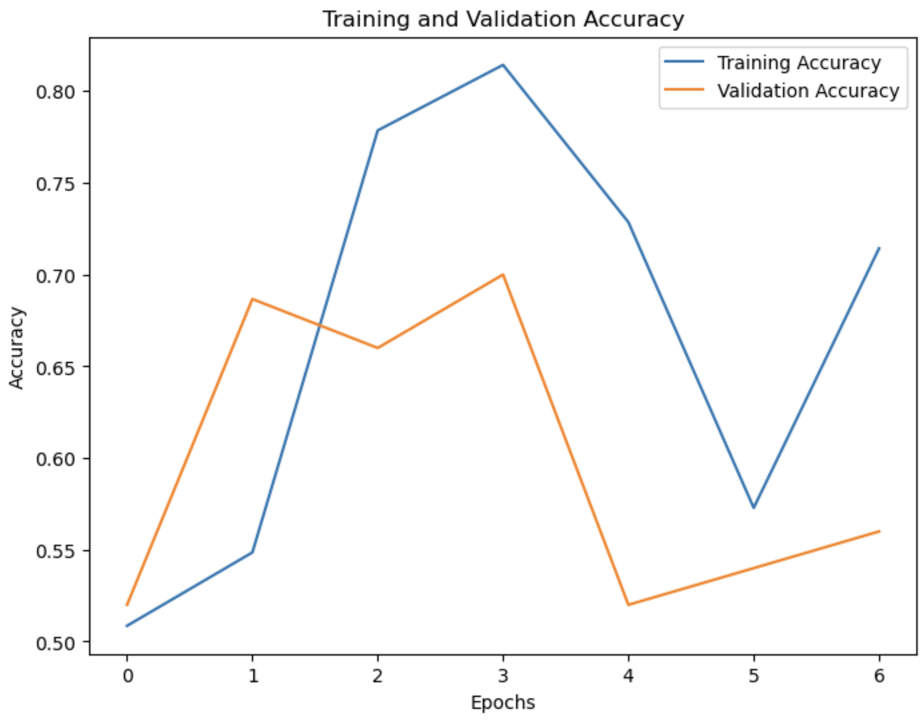


```
44]: #Accuracy test
     loss, accuracy = model.evaluate(X_test, y_test)
     print(f"Test Accuracy: {accuracy:.4f}")

     5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.6727 - loss: 0.6025
     Test Accuracy: 0.6667
```

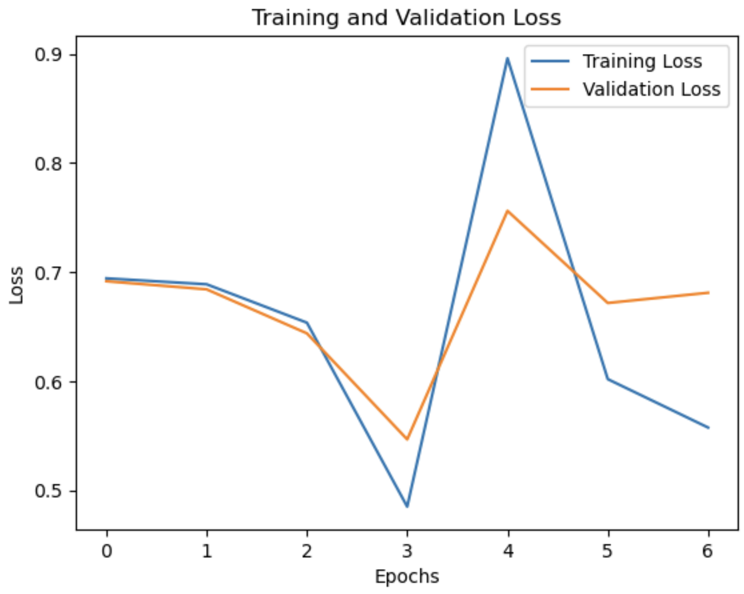


D3: Training Process

A screenshot of the training process is included to visualize the performance of the model during training.

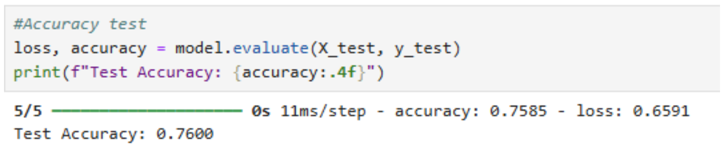


It shows the accuracy and loss for both the training and validation sets. Ideally, both training and validation accuracy should improve together, and the loss should decrease. If the model only improves on the training data and not on the validation data, it is likely overfitting.



In this case, the visual confirms that validation loss began to increase after a few epochs while training accuracy still improved. EarlyStopping intervened at that point, stopping the training to preserve the model's performance on new data.

D4: Predictive Accuracy



```
#Accuracy test
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7585 - loss: 0.6591
Test Accuracy: 0.7600
```

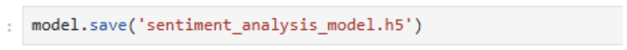
As mentioned above, the test accuracy was around 76 percent. This means that when the model is given a new movie review, it can correctly classify the sentiment with about 76 percent confidence. While not perfect, this is strong performance for a basic sentiment classifier using a relatively small and straightforward dataset. Because of the steps taken to prevent overfitting and bias, we can trust that this score is a fair reflection of how the model will perform on real-world reviews.

D5: Ethical Standards Compliance

The model complies with ethical standards by working exclusively with publicly available, non-sensitive text data. The goal of this model is to sort IMDB reviews into positive and negative categories, which does not introduce harm or privacy concerns. I processed the data to remove unusual characters and formatting issues and used all available data to reduce any unintentional exclusion or bias. While the dataset may have underlying sampling biases, the model itself does not introduce additional bias. By making the modeling process transparent and reproducible, I ensured the approach remains aligned with ethical data science principles.

E: Code

Below is the code I used to save the trained neural network model. This allows the model to be reused later for inference without retraining:



```
model.save('sentiment_analysis_model.h5')
```

This line saves the model in the recommended Keras format, ensuring compatibility with future versions of TensorFlow and ease of deployment.

F: Functionality

The trained network was successful in analyzing movie reviews and categorizing them into positive or negative sentiment. The final accuracy of approximately 76 percent demonstrates that the model has learned meaningful patterns from the training data. The architecture included an embedding layer to transform the input into dense vectors, followed by a bidirectional RNN layer to capture sequential relationships. A dropout layer helped reduce overfitting, while dense layers were used to process the transformed text data and make the final prediction. The final dense layer used a sigmoid activation function to output a probability between zero and one, making it ideal for binary classification. Overall, the model was computationally efficient and performed well on the test data.

G: Recommendations

I recommend using this model to assist with sorting IMDB movie reviews into positive or negative sentiment. It can be especially useful for organizations or researchers who want to analyze large volumes of user-generated content. Although the model performs well, additional improvements could be explored. One area of future work could be analyzing the reviews that were most difficult to classify. Some reviews may not be clearly positive or negative, which could confuse the model. Adding a third category for neutral sentiment or exploring sentiment intensity might help improve classification accuracy. For now, the model provides a solid starting point for binary sentiment classification and can help streamline the review analysis process..

I: Sources for Third-Party Code

Code for tokenization, padding, and building the neural network was used from the TensorFlow website. Code to remove unusual characters came from Stack Overflow

References

Misheva, V. (n.d.). Sentiment Analysis in Python. DataCamp. Retrieved June 20, 2025, from https://app.datacamp.com/learn/courses/sentiment-analysis-in-python

Stryker, C. (2024). What is a recurrent neural network?. IBM. Retrieved June 20, 2025, from https://www.ibm.com/think/topics/recurrent-neural-networks