D604 Advanced Analytics Task 2
By Eric Williams

A1:RESEARCH QUESTION

Is it possible to create a neural network that can differentiate between positive and negative movie reviews from IMDB using sentiment analysis? If so, with what accuracy can the reviews be predicted?

A2:OBJECTIVES OR GOALS

The goal of this analysis is to train a neural network to perform sentiment analysis on movie reviews from IMDB. The goal will be to differentiate between positive reviews and negative reviews with a high enough degree of accuracy that the model can reliably categorize the reviews.

A3:PRESCRIBED NETWORK

For this analysis, I will use a Recurrent Neural Network (RNN), which is efficient at sentiment analysis.

B1:DATA EXPLORATION

Here is how I performed data exploration on the four required items:

1. **Presence of unusual characters** (e.g., emojis, non-English characters)

```
#Checking for unusual characters (emojis, non-English, etc.)
def find_unusual_characters(text):
    return re.findall(r'[^\x00-\x7F]+', text)

#Count and sum unusual characters
all_unusual_chars = data['sentence'].apply(find_unusual_characters).sum()
unusual_char_counts = Counter(all_unusual_chars)
print(unusual_char_counts)
```

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

This function looks for any input that is not in the ASCII range. That means that it would return any strange symbols, emojis, or other non-standard American English characters. The counter then summed the count of each character so I could affirm that they were unusual characters. I then filled those characters with space since I didn't know precisely what letters they should be, if any:

```
#Cleaning the sentences
def clean_text(text):
    #Removing unusual characters
    text = re.sub(r'[^\x00-\x7F]+', ' ', text)
```

Then I ran the counter again and confirmed the odd symbols had been removed:

```
#Checking if our unusual character removal worked
all_unusual_chars_cleaned = data['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**

For the tokenization process, it is helpful to know the vocabulary size. I calculated that here:

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

```
Vocabulary size: 3052
```

Adding one helps ensure there's space for the index.

3. **Word embedding length**

When defining the model, I chose a word embedding length of 100. The choice is somewhat arbitrary, but generally, it is necessary to pick a number with enough entries that the similarity of words can be adequately captured. Choosing 100 is a good balance of allowing enough information while not including an unnecessary amount of information that won't help our model.

```
#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'])
```

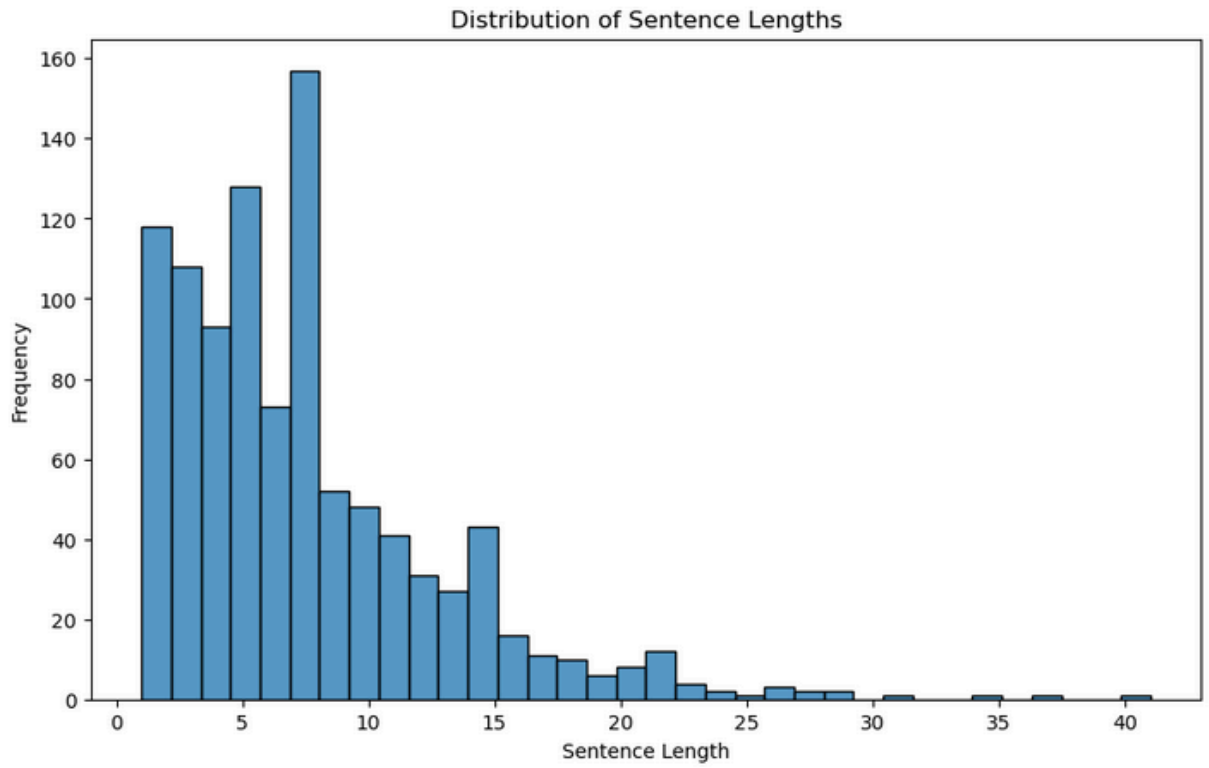
4. **Statistical justification for the chosen maximum sequence length**

When considering the maximum sequence length, I looked at the length of most sentences and created a plot:

```
#Calculating sentence length
sentence_lengths = data['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
```

As you can see, most sentences are between 2 and 15 words. However, the longest sentence is 41 words. I chose a padding length of 50 to account for the longest sequence length in my dataset.

B2:TOKENIZATION

According to the Datacamp course *Sentiment Analysis in Python* (Misheva, n.d.), the goal of tokenization is to break the strings of text into smaller chunks for easier and more accurate processing. To help normalize the data, I converted all the text to lowercase because the uppercase letters are not significant from a data standpoint. With the data preprocessing of eliminating odd characters, reducing unnecessary spaces, removing unnecessary punctuation, and converting all letters to lowercase, the tokenization is able to chunk the words together into tokens for more efficient processing. Hopefully, this will result in higher accuracy in our final model. Here is the code I used to process the data and the code I used to tokenize the data:

```
#Stripping space and organizing data
with open(file_path, 'r') as file:
    for line in file:
        sentence, label = line.strip().split('\t')
        sentences.append(sentence)
        labels.append(int(label))

data = pd.DataFrame({'sentence': sentences, 'label': labels})

print("Original data:")
print(data.head())
```

```
#Checking for unusual characters (emojis, non-English, etc.)
def find_unusual_characters(text):
    return re.findall(r'[^\x00-\x7F]+', text)

#Count and sum unusual characters
all_unusual_chars = data['sentence'].apply(find_unusual_characters).sum()
unusual_char_counts = Counter(all_unusual_chars)
print(unusual_char_counts)
```

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

```
#Cleaning the sentences
def clean_text(text):
    #Removing unusual characters
    text = re.sub(r'[^\x00-\x7F]+', ' ', text)

    #All text lowercase
    text = text.lower()

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

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

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

    return text

#Cleaning the sentences in the dataframe
data['cleaned_sentence'] = data['sentence'].apply(clean_text)
```

```
#Checking if our unusual character removal worked
all_unusual_chars_cleaned = data['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(data['cleaned_sentence'])
sequences = tokenizer.texts_to_sequences(data['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 standardize the length of the sequences so they are all the same size. For the neural network to process them properly, they need to be the same length. If a sentence is shorter than the maximum length, then padding provides a string of zeros to the end of the sequence. Here is an example of my code that performs the padding, including a result of the padding process on the first sentence in my data:

```
#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]
```

In this case, padding occurs after the text sequence as the numbers in the array are followed by zeroes.

B4:CATEGORIES OF SENTIMENT

In this network, I am separating the IMDB reviews into **positive** and **negative reviews**, meaning there are two categories of sentiment. I will be using the Sigmoid function as an activation function in the final dense layer of my network. According to IBM's website (Stryker, 2024), the Sigmoid function handles binary classification because it outputs 0 or 1 as a result.

```
#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

Above I explain the process of preparing the data. To summarize, I looked for unusual characters and deleted them, found the size of vocabulary in the data and accommodated the size, tokenized the data to chunk the data in a way the neural network can better analyze the sentences, and padded the sentences so they are all the same length. Next, I used test/train/split to perform a split for 70% training, 15% testing, and 15% validation. When deciding how long to let our model run later, it will be essential to have a lot of training data to create an accurate model, but still have both sufficient test and validation sets for the early stopping algorithm. This is a typical split in the industry.

```
#Train-test-validation split with 70-15-15 split
labels = data['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 copy of the prepared dataset will be attached in this submission.

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

As seen above, there are 5 layers in my model. Below I will discuss each layer, the type of layer, and the parameters:

1. **Embedding Layer**: This layer converts the input words into readable vectors. This layer has 305,200 parameters as a result of a calculation by multiplying the embedding number above (100) by the total vocabulary.
2. **Bidirectional Layer**: A bidirectional layer is a layer that can process data forward and backward. It processes the data and outputs it into a specific sized vector. This layer has 234,496 parameters as a calculation based on the needed size of vector and the input from the previous layer.
3. **Dropout Layer**: This layer helps prevent overfitting by randomly selecting some of the inputs and dropping them. As a result, this layer has no parameters.
4. **Dense Layer**: The first dense layer transforms the data and connects each datapoint to one of the 64 neurons. A simple multiplication describes why there are 16,448 parameters, based on the number of neurons and the number of datapoints.
5. **Output Dense Layer**: This layer transforms the data again, reducing it to a single node to calculate the probability of the sentiment to either be positive or negative.

C3:HYPERPARAMETERS

Below are the hyperparameters I chose and a justification of each one.

1. **Activation functions**
    a. According to the IBM website (Stryker, 2024), **ReLU** is used in the dense layer because it helps improve efficiency in both computation and training.
    b. As discussed above, the final output layer uses a **Sigmoid** function because it outputs values between 0 and 1 that will be used to calculate the probability of positive or negative sentiment.

2. **Number of nodes per layer**
    a. The **Embedding layer** has 100 dimensions. As mentioned above, the choice to include 100 entries as the embed length somewhat arbitrary, but generally, it is necessary to pick a number with enough entries that the similarity of words can be adequately captured. Choosing 100 is a good balance of allowing enough information while not including an unnecessary amount of information that won't help our model.
    b. The **Bidirectional layer** has 128 units. Because it can process data forward and backward, it can be especially efficient. 128 units is standard for this kind of layer.
    c. The **dropout layer** has no nodes as it is just a layer that drops neurons to prevent overfitting.
    d. The **first dense layer** has 64 units, which is a reasonable balance for a fully connected layer to capture the patterns in the sentences while simultaneously not being too large that it might cause overfitting or be computationally heavy.
    e. The **final dense layer** reduces the information into one node because at this point, all we need is a single number to calculate the probability of positive or negative sentiment.
3. **Loss function**: I chose **Binary Crossentropy** as the loss function because our model is reducing the neural network output into one single binary decision based on probability. Because we are predicting positive or negative sentiment, binary crossentropy is necessary.
4. **Optimizer:** I used Adam as my optimizer because it is a versatile optimizer that can be especially powerful at optimizing processes when there is high variance among the data. It is a standard optimizer works great in many different neural networks and helped contribute to a fairly accurate sentiment analysis.
5. **Stopping criteria**: I chose EarlyStopping with patience=3. With too high of a patience, the model will overfit and it won't perform well because the model will overtrain on the train data. The stopping criteria of patience=3 will stop the training if the validation loss stops improving for 3 epochs in a row, which would show that even though the model is performing with higher accuracy, it is performing worse on *new* data. Since this model's purpose is to correctly categorize new data, a loss in the validation set indicates our model is getting worse. Using early stopping, we can optimize the computational power of the training while also preventing overfitting.

D1:STOPPING CRITERIA

As mentioned above, I chose EarlyStopping with patience=3. Here is a screenshot of the final epochs, including the model stopping after 3 consecutive increases in loss on the validation set:

```
Non-trainable params: 0 (0.00 B)
Epoch 1/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 6s 126ms/step - accuracy: 0.4919 - loss: 0.6938 - val_accuracy: 0.6267 - val_loss: 0.6910
Epoch 2/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 70ms/step - accuracy: 0.5828 - loss: 0.6893 - val_accuracy: 0.6733 - val_loss: 0.6842
Epoch 3/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - accuracy: 0.7267 - loss: 0.6689 - val_accuracy: 0.5067 - val_loss: 0.6780
Epoch 4/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 78ms/step - accuracy: 0.7328 - loss: 0.5866 - val_accuracy: 0.8000 - val_loss: 0.5214
Epoch 5/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 62ms/step - accuracy: 0.8585 - loss: 0.4631 - val_accuracy: 0.5800 - val_loss: 0.6991
Epoch 6/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 57ms/step - accuracy: 0.7710 - loss: 0.4620 - val_accuracy: 0.5933 - val_loss: 0.6571
Epoch 7/50
11/11 ━━━━━━━━━━━━━━━━━━━━ 1s 63ms/step - accuracy: 0.8311 - loss: 0.4357 - val_accuracy: 0.6600 - val_loss: 0.6366
```

This should prevent overfitting and optimize the accuracy of the model while not overtraining on the training set. With early stopping, our model should perform better on new data than if we did not stop the training.

D2:FITNESS

The accuracy of the test is around 76%, which means we can surmise the model is not underfitting:

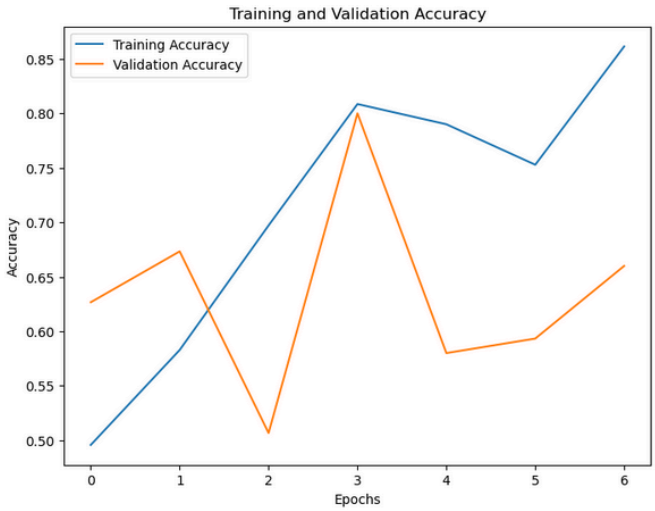
```
#Accuracy test
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - accuracy: 0.7806 - loss: 0.5351
Test Accuracy: 0.8067
```

As discussed above, I used Early Stopping to prevent overfitting. In addition, I used a dropout layer to randomly turn off half the neurons to prevent the model from overfitting. A good way to tell that the model is overfit is by comparing the training accuracy and loss against the validation training and loss. This comparison, which I will do below, will tell us if the model only performs well on the data it was trained on, or if it performs well on new data.

D3:TRAINING PROCESS

Here is a screenshot of a visualization of the training process:



Our goal is to have the accuracy increase on all sets of data. However, if our model is only increasing at identifying the training set and not new data, then we need to stop training the model. Here is a visualization of the training and validation loss:

This is why early stopping is important--when we see consecutive increases in validation loss, the model must stop training before it is overfit.

D4:PREDICTIVE ACCURACY

As mentioned above, the model accuracy is about 76%.

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

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.7377 - loss: 0.5652
Test Accuracy: 0.7600
```

This means the model can accurately predict whether a review is positive or negative with 81% accuracy when given a new movie review. While it could be higher, this is fairly accurate and we can be certain that the model is not overestimating its accuracy because of the measures taken to prevent overfitting.

D5:ETHICAL STANDARDS COMPLIANCE

My model conforms to ethical standards in several ways. First, sorting movie reviews into positive and negative categories does not provide any harm to anyone. I also worked to decrease bias in the data by including all the data available, processing it beforehand to be sure it is not confused by formatting or unknown characters, and using industry best practices. While the "behind the scenes" of the model can be confusing, the model itself is transparent and simple; the goal of the model is to categorize movie reviews using neural networks. Although I cannot account for bias in sample or demographic when the data was collected, by including all available data, I can be sure that any bias does not come from the neural network itself.

E:CODE

Here is the code I used to save the neural network:

```
model.save(r'C:\Users\18014\Desktop\sentiment_analysis_model.h5')
```

F:FUNCTIONALITY

The network can accurately predict movie reviews by analyzing text, separating them into positive or negative categories with 76% accuracy. Using a Recurrent Neural Network (RNN) was efficient when performing sentiment analysis, and the specific architecture mentioned above was computationally efficient while also creating a model that was fairly accurate. Using a dense model with a Sigmoid function was efficient in reducing the network into a single binary probability. Using a dropout layer helped prevent overfitting, while the rest of the layers were great at transforming the data and connecting the data to neurons.

G:RECOMMENDATIONS

I recommend this model be used to automate sorting IMDB reviews into positive and negative with a moderate degree of accuracy. Further analysis should be done on which reviews were the most difficult to categorize into positive or negative. In particular, it might be worth looking into balanced or neutral reviews because the reviewer might not see the movie just as "positive" or "negative," and that could account for where the model falls short. Although the model is fairly accurate and can be used to generally understand the sentiment of a review, further analysis should be done to identify problems and improve the model.

I:SOURCES FOR THIRD-PARTY CODE

Code was used from the Tensorflow website for tokenization, padding, and neural network architectural design.

Code for removing unusual text was used from Stack Overflow.

Sources

Misheva, V. (n.d.). Sentiment Analysis in Python. DataCamp. Retrieved December 2024 from https://app.datacamp.com/learn/courses/sentiment-analysis-in-python

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