Task 2: Sentiment Analysis Using Neural Networks

D213

Abhishek Aern

Western Governor University

Contents

Part I - Research Question

A1: Question for analysis

Can a neural network-based sentiment analysis model predict whether a customer would (or would not) recommend Amazon products based on the previous review in an Amazon dataset? and how the model can be optimized to improve sentiment classification accuracy?

A2: Objective and Goals of Analysis

The goal of this research question is to create an accurate classifier that can assist in identifying customer sentiment toward product recommendations. The primary objectives of this research question are to develop a model that leverages sentiment analysis to determine the underlying sentiment that indicates recommendation or non-recommendation and optimize the sentiment classification accuracy of the model.

The Performance improvement will include exploring methods to optimize and improve its performance in accurately classifying sentiments and predicting customer recommendations for Amazon products based on previous reviews.

The research aims to provide insights into the feasibility and effectiveness of using neural network-based sentiment analysis for predicting product recommendations based on customer reviews. The findings can contribute to enhancing customer understanding, marketing strategies, and decision-making processes for Amazon.

A3: Prescribed Network

One type of neural network that is commonly used for text classification tasks, such as sentiment analysis, and can be trained on text sequences is a recurrent neural network (RNN). RNNs are designed to handle sequential data by maintaining internal memory that allows them to process and analyze information sequentially. They are well-suited for tasks where the order and context of words or tokens in a text sequence are important.

One of the popular variants of RNNs is the Long Short-Term Memory (LSTM) network. LSTM networks have an added advantage over traditional RNNs as they can effectively capture long-term dependencies and handle the vanishing gradient problem.

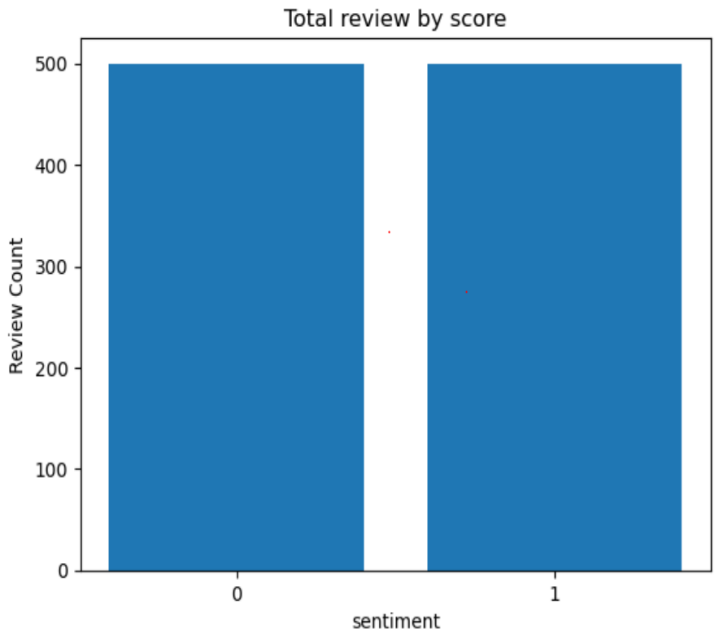
LSTM networks are capable of learning patterns and dependencies within the text data, making them suitable for sentiment analysis or any other text classification task on the selected Amazon dataset. They can be trained to produce useful predictions by learning the sentiment patterns and relationships within the text sequences, enabling accurate classification of positive, negative, or neutral sentiments.

By training an LSTM-based neural network on the Amazon dataset, we can leverage its ability to capture sequential information and make predictions on text sequences, providing valuable insights into sentiment or other classification tasks related to customer reviews.

Part II - Data Preparation

B1: Data Exploration

Here is the bar chart showing the total review by score present in the Amazon dataset. This dataset has the same number of sentiments positive and negative.



Presence of Unusual Characters:

I am using the regular expressions (regex) module for pattern matching and filtering. This function takes the review text as input and applies the regex pattern [^a-zA-Z\s] to match any character that is not a letter, digit, or whitespace. The re.sub() function replaces all occurrences of the pattern with an empty string, effectively removing the special characters.

I defined a function clean_text() that is used to preprocess text data by removing special characters, converting the text to lowercase, and returning the cleaned version of the text. The updated function is then applied to the 'review' column using the apply() function and a lambda function.

The below code will remove special characters and convert all the text in the 'review' column to lowercase for all 1000 records in the DataFrame.

```
# Function to remove special characters and convert to lowercase
def clean_text(text):
    # Remove special characters using regex
    #cleaned_text = re.sub(r'[^a-zA-Z0-9\s]', '', text)
    cleaned_text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Convert to lowercase
    cleaned_text = cleaned_text.lower()
    return cleaned_text
```

```
# Apply the clean_text function to the 'review' column
X['review'] = X['review'].apply(lambda x: clean_text(x))
```

```
X.head()
```

	review
0	so there is no way for me to plug it in here i...
1	good case excellent value
2	great for the jawbone
3	tied to charger for conversations lasting more...
4	the mic is great

Vocabulary size :

To calculate the vocabulary size, I had to tokenize the text review data into individual words to count the number of unique words in the data. (the tokenized process is explained in the next section B2).

Here is my code to calculate the vocabulary size –

```
# copy
reviews = X['review']

# Combine all reviews into a single list
all_tokens = [token for review in reviews for token in review]

# Calculate the vocabulary size
vocabulary_size = len(set(all_tokens))

print("Vocabulary Size:", vocabulary_size)
```

```
Vocabulary Size: 1750
```

In the above code, the normalized_tokens list represents the normalized tokens after performing tokenization and text preprocessing. By converting the list to a set using set(normalized_tokens) which provides a collection of unique words. Taking the length of this set using len() provides the vocabulary size(number of unique words in the data).

The vocabulary size of my dataset is 1750. This means that there are 1750 unique words present in my normalized tokens.

Proposed word embedding length

The embedding length involves determining the optimal dimensionality for representing words in the embedding space. While there is no definitive rule for choosing the exact word embedding length. Determining the proposed word embedding length involves considering factors such as vocabulary size, dataset size, computational resources, and task performance (Terry-Jack, 2019).

Here is the code I am using to calculate the embedding length -

```
# Calculate dataset size
dataset_size = len(X['review'])  # Number of samples in your dataset

# Calculate the proposed word embedding length
embedding_length = min(50, int(vocabulary_size ** 0.25 * dataset_size ** 0.25))

print("Embedding length:", embedding_length)
```

```
Embedding length: 36
```

My formula to calculate the proposed word embedding length is min(50, int(vocabulary_size ** 0.25 * dataset_size ** 0.25)). This formula combines the fourth root of the vocabulary size and the fourth root of the dataset size, ensuring a balance between these two factors. The min of 50 and the calculated value ensures that the calculated embedding length does not exceed 50, providing an upper limit to prevent overfitting or excessive computational complexity. Based on this my embedding length is 36.

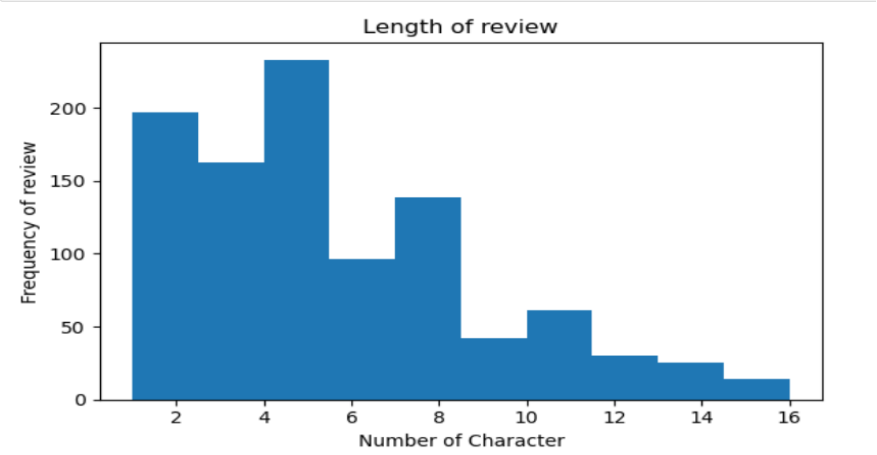
Statistical Justification for Chosen Maximum Sequence Length:

Determining the maximum sequence length using the below code to find out the length of the longest review in the Amazon dataset.

```
: # Calculate the maximum sequence length
  line_num_words = [len(t_line) for t_line in X['review']]

  max_sequence_length = max(line_num_words)

  print("Maximum Sequence Length:", max_sequence_length)
```

```
Maximum Sequence Length: 16
```

Also created the below histogram to visualize the distribution and noticed that the max length is 16 in my dataset.



B2: Tokenization

Tokenization is the process of breaking text into individual tokens or units, such as words, subwords, or characters. The main goal of tokenization is to divide the text into meaningful units that can be further processed or analyzed. Tokenization can be as simple as splitting text on whitespace or more complex, considering punctuation, special characters, or linguistic rules. Tokens will be used as input to NLP tasks, such as sentiment analysis.

To tokenize the 'review' column, which contains text review data, I am using the nltk library which provides various tokenization methods that can help split the text into individual tokens. Here is my code -

```
# Tokenize the 'review' column
X['review'] = X['review'].apply(word_tokenize)
```

```
X
```

	review
0	[so, there, is, no, way, for, me, to, plug, it...
1	[good, case, excellent, value]
2	[great, for, the, jawbone]
3	[tied, to, charger, for, conversations, lastin...
4	[the, mic, is, great]
...	...
995	[the, screen, does, get, smudged, easily, beca...
996	[what, a, piece, of, junk, i, lose, more, call...
997	[item, does, not, match, picture]
998	[the, only, thing, that, disappoint, me, is, t...
999	[you, can, not, answer, calls, with, the, unit...

1000 rows × 1 columns

In the above code, the word_tokenize() function from nltk.tokenize is used to tokenize each review in the 'review' column. The apply() function is used to apply the tokenization function to each review in the DataFrame. After running this code, I am replacing the original 'review' column with the tokenized version of the 'review' column.

Removal of stopwords :

To remove stop words from the tokenized 'review' column I am again using the nltk library which provides a list of common stop words in various languages which can be used to filter out those stop words from the tokenized text.

In my case, the stopwords module from nltk.corpus is used to access a set of English stop words. The modified set of stop words is created by excluding the negating words that I want to keep. The stop words are removed from the tokenized text using a lambda function and list comprehension. The 'or' condition ensures that the negating words are not removed.

```
# Remove most stop words
stop_words = set(stopwords.words('english'))  # Set the language to 'english'
```

```
# Define the negating words to keep
negating_words = ['not', 'no', 'never', 'none', 'neither', 'nor', 'cannot', "can't", "won't", "don't", 'but'
                  "doesn't", "isn't", "wasn't", "weren't", "hasn't", "haven't", "hadn't", "wouldn't", "shouldn't", "couldn't"
]
```

```
# Remove stop words excluding negating words
filtered_X = X['review'].apply(lambda tokens: [word for word in tokens if word not in stop_words or word in negating_words])

# Update the origional X review column with the filtered tokens
X['review'] = filtered_X
```

```
X['review']
```

```
0            [no, way, plug, us, unless, go, converter]
1                       [good, case, excellent, value]
2                                      [great, jawbone]
3     [tied, charger, conversations, lasting, minute...
4                                          [mic, great]
                            ...
995   [screen, get, smudged, easily, touches, ear, f...
996                  [piece, junk, lose, calls, phone]
997                        [item, not, match, picture]
998         [thing, disappoint, infra, red, port, irda]
999           [not, answer, calls, unit, never, worked]
Name: review, Length: 1000, dtype: object
```

After running the code snippet, the stop words (excluding the negating words) are removed from the tokenized 'review' column.

After this, I need to convert the text data to sequences using the code below.

```
# Create a tokenizer and fit it on the text data
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X['review'])

# Convert the text data to sequences
sequences = tokenizer.texts_to_sequences(X['review'])
word_index = tokenizer.word_index
```

```
# display the sequence of some random index
sequences[0], sequences[189]
```

```
([24, 136, 86, 436, 437, 155, 685],
 [868, 128, 869, 870, 497, 871, 457, 331, 46, 872, 27, 371])
```

B3: Padding Process

The padding process is used to standardize the length of sequences to create input data that is suitable for training an LSTM model. It involves adding extra tokens (or padding tokens) to sequences that are shorter than the desired length and truncating sequences that are longer than the desired length.

Regarding the placement of padding tokens, it can occur either before or after the text sequence. In the case of the Keras sequence.pad_sequences function, the default behavior is to pad sequences at the beginning (pre-padding). This means that the padding tokens are added to the beginning of the sequence, shifting the original text towards the end.

By standardizing the length of sequences through the padding, we ensure that the input data has consistent dimensions and can be efficiently processed by the LSTM model.

In the below code, padding is applied after converting the text data into sequences using the pad_sequences function from tensorflow.keras.preprocessing.sequence.

The pad_sequences function is applied to the sequences. The maxlen parameter is set to a specific value (in my case the max length is 16) to define the desired length of the sequences. If a sequence is shorter than the specified length, it is padded with zeros at the beginning to match the desired length.

```
# Pad sequences to ensure consistent length
max_sequence_length = max([len(seq) for seq in sequences])
padded_sequences = pad_sequences(sequences, maxlen=max_sequence_length)
```

```
padded_sequences
```

```
array([[   0,    0,    0, ...,  437,  155,  685],
       [   0,    0,    0, ...,   19,   21,  182],
       [   0,    0,    0, ...,    0,    3,  314],
       ...,
       [   0,    0,    0, ...,    2,  346,  160],
       [   0,    0,    0, ...,  641, 1748, 1749],
       [   0,    0,    0, ...,  134,  119,   35]])
```

```
padded_sequences[0]
```

```
array([  0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 136,  86, 436,
       437, 155, 685])
```

```
padded_sequences[635]
```

```
array([   0,    0,    0,    0,    0,    0,   27, 1329,    4,    6,   42,
         36,  379,    2,   83,  642])
```

```
padded_sequences.shape
```

```
(1000, 16)
```

By applying padding, all sequences will have the same length of 16, regardless of the original length of the text. This ensures that the input data is consistent and compatible with the model architecture. I have also provided a couple of examples of padded sequences.

B4: Categories of Sentiment

For binary sentiment classification, I have two categories of sentiment in the Amazon dataset. The final dense layer of the neural network will have a single output neuron with a sigmoid activation function.

The output of the sigmoid activation function will be a single value between 0 and 1, representing the probability of the positive sentiment class. Values closer to 1 indicate a higher probability of positive sentiment, while values closer to 0 indicate a higher probability of negative sentiment.

In binary sentiment classification, the goal is to classify text into either a positive or negative sentiment category. The final dense layer with a sigmoid activation function

allows the model to output a probability score that can be interpreted as the likelihood of the input belonging to the positive sentiment category.

B5: Steps to Prepare the Data

To prepare the data for analysis, the following steps are involved (Virahonda, 2020) :

Load the dataset: The first step is to load the Amazon dataset that contains the text review and the corresponding sentiment.

Basic data cleaning - Check for null and missing values using the .isnull() and .isna() functions. No null or missing value was found in the Amazon dataset.

Split the data into input and target variables: The text input reviews are stored in X as Independent Features and sentiment is stored in y as a dependent feature or target variable.

Preprocess the text data: Text data often requires preprocessing to convert it into a suitable format for analysis. My preprocessing steps include tokenization, removing punctuation, converting text to lowercase, and removing stopwords(excluding the negative words).

Tokenize and encode the text: The text review data needs to be converted into numerical representations that can be processed by the machine learning model. This involves tokenizing the text into individual words or subwords and encoding them as integers or using techniques like one-hot encoding or word embeddings.

Padding sequences: Since the LSTM model expects input sequences of fixed length, the sequences need to be padded or truncated to a uniform length. Padding involves adding extra tokens (e.g., 0s) to shorter sequences, while truncation involves cutting off excess tokens from longer sequences.

Split the dataset: The next step is to split the dataset into training, validation, and test sets. The purpose of splitting the data is to have separate subsets for training, evaluating, and testing the performance of the sentiment analysis model.

Training set: This is the largest subset of the data and is used to train the model. It is used to update the model's parameters during the training process. Out of 1000 records, this will have 600 records.

Validation set: This subset is used for tuning the hyperparameters of the model and evaluating its performance during training. It helps in selecting the best model configuration and preventing overfitting. The validation set is used for early stopping and monitoring the model's performance. This consists of 200 records out of 1000.

Test set: This subset is used for the final evaluation of the trained model. It provides an unbiased estimate of the model's performance on unseen data. The test set should be representative of real-world data that the model will encounter. This also consists of the remaining 200 records out of 1000.

The specific size of the training, validation, and test sets can vary depending on the available data and the requirements of the analysis. We have only 1000 records for the Amazon dataset so dividing it into 600, 200, and 200 for training, validation, and testing.

```
# Split the data into train, test, and validation sets
X_train, X_test, y_train, y_test = train_test_split(X_final,y_final, test_size=0.2, random_state=42)

# Split the train data to train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42)
```

```
X_train.shape, X_val.shape, X_test.shape, y_train.shape, y_val.shape, y_test.shape
```

```
((600, 16), (200, 16), (200, 16), (600,), (200,), (200,))
```

By following these steps, the data is prepared for analysis using an LSTM-based sentiment analysis model.

B6: Prepared Dataset

I am providing prepared and clean training, validation, and testing data in CSV files as below -

Training Data	X_train_Task2.csv
	y_train_Task2.csv
Validation Data	X_val_Task2.csv
	y_val_Task2.csv
Testing Data	X_test_Task2.csv
	y_test_Task2.csv

This is the code used to extract the data (Resources)–

Cleaned Dataset

```
: # Extract the training data in CSV format
  pd.DataFrame(X_train).to_csv('X_train_Task2.csv', index=False)
  pd.DataFrame(y_train).to_csv('y_train_Task2.csv', index=False)

  # Extract the validation data in CSV format
  pd.DataFrame(X_val).to_csv('X_val_Task2.csv', index=False)
  pd.DataFrame(y_val).to_csv('y_val_Task2.csv', index=False)

  # Extract the testing data in CSV format
  pd.DataFrame(X_test).to_csv('X_test_Task2.csv', index=False)
  pd.DataFrame(y_test).to_csv('y_test_Task2.csv', index=False)
```

Part III - Network Architecture

C1: Model Summary

The output of the model summary -

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 16, 36)            63036

 lstm_2 (LSTM)               (None, 16, 128)           84480

 lstm_3 (LSTM)               (None, 64)                49408

 dense_3 (Dense)             (None, 64)                4160

 dense_4 (Dense)             (None, 32)                2080

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 203,197
Trainable params: 203,197
Non-trainable params: 0
_________________________________________________________________
```

A detailed description of this summary is presented in the next section C2.

C2: Network Architecture

Here is what my model looks like –

```
# Instantiate a sequential model

embedding_dim = embedding_length

model = Sequential()
model.add(Embedding(len(word_index) + 1, embedding_dim, input_length=max_sequence_length))
model.add(LSTM(128, return_sequences=True))
model.add(LSTM(64))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

This LSTM model consists of the following layers:

Embedding layer: The Embedding layer is responsible for learning and mapping word embeddings for each word in the input sequence. It takes the input of shape (batch_size, input_length) and outputs a sequence of word embeddings of shape (batch_size, input_length, embedding_dim). The embedding_dim parameter specifies the dimensionality of the word embeddings. I am using the embedding_dim as 36. (Additional details for this calculation are provided in previous section B1)

This embedding layer has a total of (vocab_size + 1) * embedding_dim parameters.

LSTM layer (with return_sequences=True): This LSTM layer is configured to return the sequence of hidden states for each time step in the input sequence. It takes the sequence of word embeddings as input and outputs a sequence of hidden states. The number of units in this LSTM layer is set to 128, which represents the dimensionality of the output space. It has a total of 4 * (embedding_dim + 128) * 128 parameters. The multiplication by 4 comes from the fact that an LSTM cell consists of four gates (input, forget, output, and cell update gates).

LSTM layer: This second LSTM layer is configured to return the last hidden state of the sequence. It takes the sequence of hidden states from the previous LSTM layer

and outputs a single vector representing the final hidden state. The number of units in this LSTM layer is set to 64. It has a total of 4 * (128 + 64) * 64 parameters.

Dense layer with 64 units and 'relu' activation: This dense layer applies a linear transformation to the input vector from the previous LSTM layer, followed by the rectified linear unit (ReLU) activation function. ReLU introduces non-linearity to the model and helps in capturing complex patterns. This dense layer has (64 + 64) * 64 parameters.

Dense layer with 32 units and 'relu' activation: This additional dense layer further applies a linear transformation followed by ReLU activation. It can provide more expressive power to the model by adding more non-linear transformations. This dense layer has (64 + 32) * 32 parameters.

Dense layer with 1 unit and 'sigmoid' activation: This final dense layer is responsible for producing the output prediction. It applies a linear transformation followed by the sigmoid activation function, which squashes the output between 0 and 1, representing the predicted sentiment probability (0 for negative sentiment, 1 for positive sentiment). This dense layer has (32 + 1) parameters.

This architecture allows the LSTM model to learn the representations of words through the embedding layer, capture sequential patterns and dependencies through the LSTM layers, and make a final prediction through the dense layers.

To calculate the total number of parameters in the model, sum up the parameters from all the layers:

Total parameters = Embedding layer parameters + LSTM layer 1 parameters + LSTM layer 2 parameters + Dense layer 1 parameters + Dense layer 2 parameters + Dense layer 3 parameters

Note that the bias parameters are also included in the calculation. The total parameter count is mentioned in the screen print of section C1.

C3: Hyperparameters

The choice of hyperparameters in my model is justified as follows (Parikh):

Activation functions:

The 'relu' activation function is used in the Dense layers. ReLU is a commonly used activation function in deep learning models as it introduces non-linearity and helps the model learn complex patterns in the data.

The 'sigmoid' activation function is used in the last Dense layer to produce a binary output representing the sentiment probability. Sigmoid squashes the output between 0 and 1, making it suitable for binary classification tasks to predict customer sentiment probability (0 for negative sentiment, 1 for positive sentiment).

Number of nodes per layer:

The number of nodes per layer is determined based on the complexity of the task and the size of the dataset. It is a hyperparameter that can be tuned to find the optimal balance between model capacity and overfitting. In my case, I ran the model with multiple parameter values and finally found that this model parameter is giving me consistent results with higher accuracy. All the parameters are discussed in section C2 in detail.

Binary cross-entropy ('binary_crossentropy') is a suitable loss function for binary classification tasks where the output is a probability between 0 and 1.
It measures the dissimilarity between the true labels and the predicted probabilities and is commonly used for sentiment analysis tasks.

Optimizer:

The 'adam' optimizer is widely used for training deep learning models. It combines the advantages of adaptive learning rates and momentum methods, making it suitable for a variety of tasks.

Adam adapts the learning rate during training, which helps in faster convergence and handling different learning rate requirements for different parameters.

Stopping criteria:

I am using EarlyStopping as the stopping criteria using the EarlyStopping callback. Early stopping is a technique to prevent overfitting and save computational resources by stopping the training process when the validation loss does not improve for a certain number of epochs (patience).

Here is my code for the EarlyStopping callback -

```
# Define EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_accuracy', patience=3)
```

In this case, the choice of stopping criteria is based on the validation accuracy, with a patience of 3 epochs. If the validation accuracy does not improve for 3 consecutive epochs, the training will be stopped early.

Evaluation metric:

The model is evaluated using the 'accuracy' metric, which measures the proportion of correct predictions over the total number of predictions.

The below code is using the evaluate() method which computes the loss and accuracy for the given test(unseen) data. It returns the test loss and the calculated accuracy.

```
## Evaluate the model on the test data
loss, accuracy = model.evaluate(X_test, y_test)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
7/7 [==============================] - 0s 9ms/step - loss: 0.8533 - accuracy: 0.8100
Test Loss: 0.8533172011375427
Test Accuracy: 0.8100000023841858
```

Based on these results the test loss is 0.853 which means, on average, the model's predictions have a relatively higher error compared to the ground truth labels in the test dataset. Lower values of the test loss indicate better performance.

The test accuracy is 0.810 which means the model correctly predicts the labels for approximately 81% of the instances in the test dataset. Higher values of test accuracy indicate better performance. This model achieved a training accuracy of 98% with a training loss of 0.0849 along with a validation accuracy of 77% with a loss of 0.6810 for the 8th epoch.

Based on the training of the model it looks like the training loss decreases and accuracy increases steadily with each epoch, indicating that the model is learning and improving its predictions on the training data. The validation loss and validation accuracy initially decrease, but then starts to fluctuate and increase slightly in later epochs. Also, the model achieved a test accuracy of 81%, which indicates that it can correctly predict the sentiment of the given text with relatively good accuracy.

However, the test loss of 0.853 suggests that there is still room for improvement in terms of reducing the prediction errors.

Part IV - Model Evaluation

D1: Stopping Criteria

The stopping criteria allow the model to stop training early if no further improvement is observed in the validation metrics. This helps save computational resources and time by avoiding unnecessary training iterations when the model has already converged. This also prevents overfitting. Setting a fixed number of epochs may lead to overfitting if the model continues to train on the training data even when it has already achieved optimal performance. Stopping criteria helps prevent overfitting by monitoring the validation metrics and stopping the training process when overfitting is detected (Parikh).

Stopping criteria ensure that the model is trained sufficiently to achieve a certain level of performance. If the validation metrics do not show improvement even after a few epochs, it indicates that the model may be underfitting the data. Stopping the training process helps avoid underfitting by allowing for model adjustments or hyperparameter tuning. Overall, using stopping criteria, such as the EarlyStopping callback, provides a more dynamic and adaptive approach to training the sentiment analysis model. It helps optimize training time, prevents overfitting, and allows for early detection of convergence, ultimately improving the model's performance.

My EarlyStopping criteria are mentioned below –

```
# Define EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_accuracy', patience=3)
```

The EarlyStopping callback is defined with the parameter monitor='val_accuracy' and patience=3. This means that if the validation accuracy does not improve for 3 consecutive epochs, the training process will be stopped early.

Here is a screenshot showing the final training epoch :

```
num_epochs = 15
batch_size = 64
results = model.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, validation_data=(X_val, y_val),
                    callbacks=[early_stopping])
```

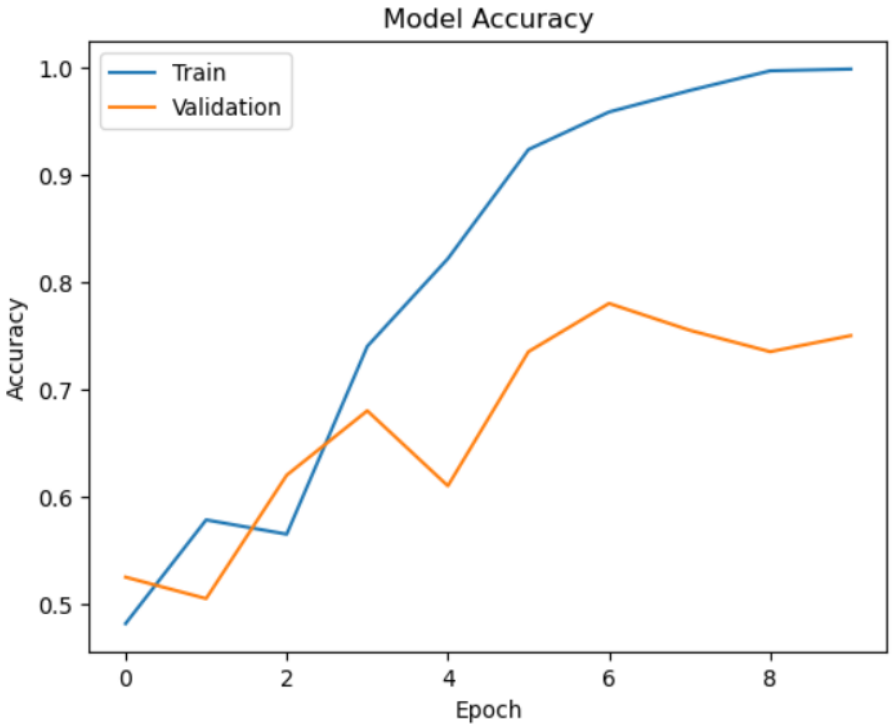
```
Epoch 1/15
10/10 [==============================] - 6s 157ms/step - loss: 0.6932 - accuracy: 0.5167 - val_loss: 0.6921 - val_accuracy: 0.5
200
Epoch 2/15
10/10 [==============================] - 0s 41ms/step - loss: 0.6906 - accuracy: 0.5700 - val_loss: 0.6893 - val_accuracy: 0.61
50
Epoch 3/15
10/10 [==============================] - 0s 35ms/step - loss: 0.6796 - accuracy: 0.6533 - val_loss: 0.6761 - val_accuracy: 0.62
50
Epoch 4/15
10/10 [==============================] - 0s 39ms/step - loss: 0.6128 - accuracy: 0.7917 - val_loss: 0.6142 - val_accuracy: 0.72
00
Epoch 5/15
10/10 [==============================] - 0s 39ms/step - loss: 0.4178 - accuracy: 0.8500 - val_loss: 0.5635 - val_accuracy: 0.67
50
Epoch 6/15
10/10 [==============================] - 0s 36ms/step - loss: 0.2974 - accuracy: 0.8767 - val_loss: 0.5768 - val_accuracy: 0.66
00
Epoch 7/15
10/10 [==============================] - 0s 40ms/step - loss: 0.1901 - accuracy: 0.9333 - val_loss: 0.5344 - val_accuracy: 0.76
50
Epoch 8/15
10/10 [==============================] - 0s 38ms/step - loss: 0.0849 - accuracy: 0.9800 - val_loss: 0.6810 - val_accuracy: 0.77
50
Epoch 9/15
10/10 [==============================] - 0s 35ms/step - loss: 0.0485 - accuracy: 0.9850 - val_loss: 0.7883 - val_accuracy: 0.71
50
Epoch 10/15
10/10 [==============================] - 0s 37ms/step - loss: 0.0253 - accuracy: 0.9933 - val_loss: 0.8309 - val_accuracy: 0.77
00
Epoch 11/15
10/10 [==============================] - 0s 35ms/step - loss: 0.0124 - accuracy: 0.9967 - val_loss: 0.9628 - val_accuracy: 0.74
50
```

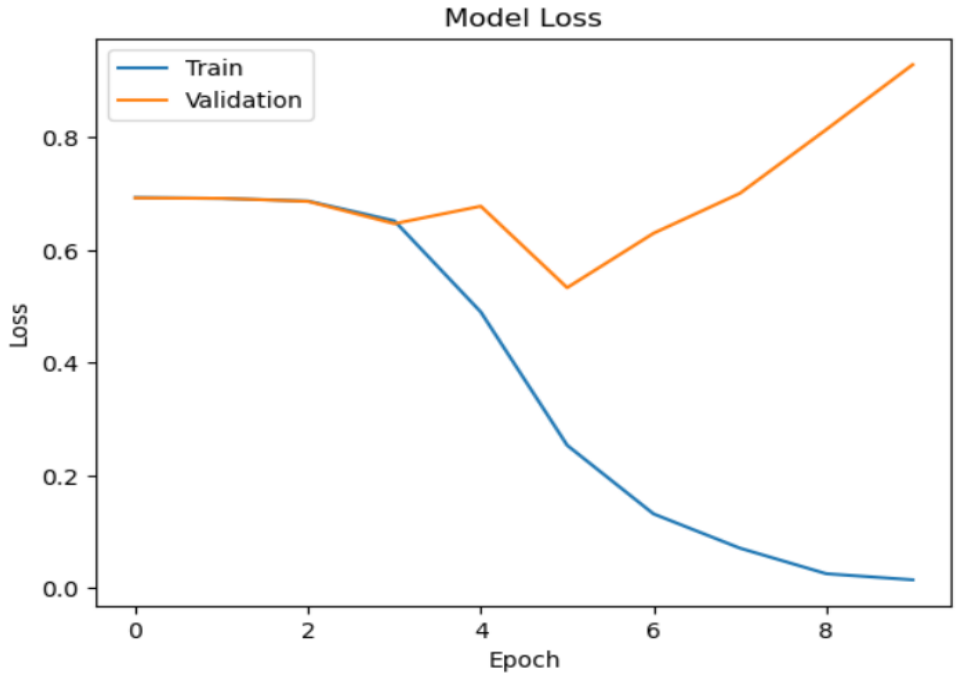
This output indicates the loss, accuracy, validation loss, and validation accuracy values at the end of the training process. Looking at these results, we can see that the validation accuracy did not improve after the 8th epoch so the training process is stopped early.

From the 8th epoch onward, the validation accuracy fluctuates but does not improve consistently for 3 epochs. Hence, after the 11th epoch, the EarlyStopping criteria are met, and the training process is stopped early. Therefore, the training process stops after the 11th epoch because the validation accuracy did not improve for 3 consecutive epochs, as defined by the patience parameter in the EarlyStopping callback.

D2: Training Process

To visualize the training process of the model, including the loss and accuracy I am plotting the line graphs using the Matplotlib library.


Model Accuracy
Train
Validation
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0
2
4
6
8


Model Loss
Train
Validation
Loss
Epoch
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8

This is using the output values from results.history which contains the recorded values of loss, accuracy, validation loss, and validation accuracy during the training process. Here is my interpretation based on this plot –

- The training loss decreases steadily with each epoch, indicating that the model is improving its performance on the training data.
- The validation loss also decreases, although with some fluctuations, suggesting that the model is generalizing well and not overfitting to the training data.
- The training accuracy increases with each epoch, indicating that the model is learning to classify the training data more accurately.
- The validation accuracy shows some fluctuations but generally increases, indicating that the model is performing well on unseen validation data.

I also provided more details about other evaluation matrices in sections C3 and D4.

D3: Fit

To assess the fitness of the model and address overfitting, several measures were taken in the provided code:

EarlyStopping: The EarlyStopping callback was used to monitor the validation accuracy and stop the training process if there was no improvement for a certain number of epochs (defined by the patience parameter). This helps prevent overfitting by stopping the training when the model starts to overfit the training data.

Validation Data: A separate validation dataset (X_val and y_val) was created to evaluate the model's performance on unseen data during the training process. The validation data helps assess the model's generalization ability and detect overfitting.

The dataset has only 1000 datapoint for customer review which is further divided into train/test and validation sets which make this dataset even smaller. Due to this reason, we can see some overfitting based on the results.

D4: Predictive Accuracy

Based on the above results, the predictive accuracy of the trained network can be

assessed by looking at the validation accuracy and loss values.

Here are the training and validation accuracy and loss values –

```
results.history
```

```
{'loss': [0.6932420134544373,
  0.6906256675720215,
  0.6795720458030701,
  0.6127599477767944,
  0.41776618361473083,
  0.2974136471748352,
  0.19006846845149994,
  0.08491211384534836,
  0.04854964092373848,
  0.02525251917541027,
  0.012444114312529564],
 'accuracy': [0.5166666507720947,
  0.5699999928474426,
  0.653333306312561,
  0.7916666865348816,
  0.8500000238418579,
  0.8766666650772095,
  0.9333333373069763,
  0.9800000190734863,
  0.9850000143051147,
  0.9933333396911621,
  0.996666669845581],
 'val_loss': [0.6921133995056152,
  0.6892898082733154,
  0.6761240363121033,
  0.6142022609710693,
  0.5635402202606201,
  0.5768404603004456,
  0.5344261527061462,
  0.6809774041175842,
  0.7883343696594238,
  0.8308792114257812,
  0.9628426432609558],
 'val_accuracy': [0.5199999809265137,
  0.6150000095367432,
  0.625,
  0.7200000286102295,
  0.675000011920929,
  0.6600000262260437,
  0.7649999856948853,
  0.7749999761581421,
  0.7149999737739563,
  0.7699999809265137,
  0.7450000047683716]}
```

Based on these results the model achieved a maximum validation accuracy of around 0.77 and the validation loss did not improve significantly after the 8th epoch. The training accuracy reached a high value of around 0.99, indicating a good fit for the training data.

Test data predictive accuracy –

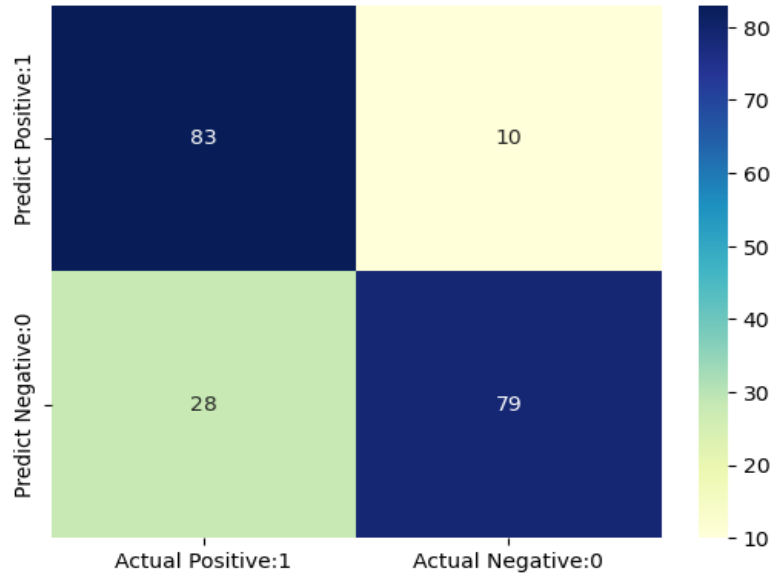
```
## Evaluate the model on the test data
loss, accuracy = model.evaluate(X_test, y_test)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
7/7 [==============================] - 0s 9ms/step - loss: 0.8533 - accuracy: 0.8100
Test Loss: 0.8533172011375427
Test Accuracy: 0.8100000023841858
```

The test accuracy of 0.810 indicates that the trained model achieved an accuracy of approximately 81% on the test dataset. This means that out of all the instances in the test set, the model correctly classified 81% of them.

Confusion Matrix: Here are the results from the Confusion matrix –

```
Confusion matrix

 [[83 10]
 [28 79]]

True Positives(TP) =  83

True Negatives(TN) =  79

False Positives(FP) =  10

False Negatives(FN) =  28

<AxesSubplot:>
```



The confusion matrix provides a detailed breakdown of the model's performance. The model correctly predicted 83 instances as positive (True Positives) along with 79 instances as negative (True Negatives).
Also, the model incorrectly predicted 10 instances as positive when they were negative (False Positives) and 28 instances as negative when they were positive (False Negatives).

Additional evaluation metrics are also calculated below –

```
# calculate the evaluation metrics
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1score = f1_score(y_test, y_pred)

print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1score)
```

```
Precision: 0.8876404494382022
Recall: 0.7383177570093458
F1-score: 0.8061224489795918
```

These results suggest that the model performs reasonably well in classifying positive instances, with a relatively high precision. However, there is some room for improvement in identifying all positive instances, as reflected by the slightly lower recall. The F1 score combines both precision and recall and provides a single metric to evaluate the model's overall performance.

Along with all the above metrics, I am also performing the prediction for new user input review. We need to preprocess the new review before applying the model. Here are the 2 predictions for each sentiment positive and negative.

```
# Example prediction on new data
new_text = ['This product is amazing!']
new_text = [re.sub('[^a-zA-Z]', ' ', text) for text in new_text]
new_sequences = tokenizer.texts_to_sequences(new_text)
new_sequences = pad_sequences(new_sequences, maxlen=max_sequence_length)
prediction = model.predict(new_sequences)

# Convert the probability to a class label
predicted_sentiment = "Positive" if prediction > 0.5 else "Negative"

# Print the predicted sentiment
print("Predicted Sentiment:", predicted_sentiment)
```

```
1/1 [==============================] - 0s 32ms/step
Predicted Sentiment: Positive
```

```
# Example prediction on new data
# new_text = ['not recomended rediculas!']
new_text = ['i am not happy with this procuct!']
new_text = [re.sub('[^a-zA-Z]', ' ', text) for text in new_text]
new_sequences = tokenizer.texts_to_sequences(new_text)
new_sequences = pad_sequences(new_sequences, maxlen=max_sequence_length)
prediction = model.predict(new_sequences)

# Convert the probability to a class label
predicted_sentiment = "Positive" if prediction > 0.5 else "Negative"

# Print the predicted sentiment
print("Predicted Sentiment:", predicted_sentiment)
```

```
1/1 [==============================] - 0s 25ms/step
Predicted Sentiment: Negative
```

Based on these results we can say that the model is correctly predicting customer sentiment.

Part V- Data Summary and Implications

E: Code Execution

I am using the save() function from the model object to save the trained model.

Save the model

```
model.save("LSTM_sentiment_analysis_model.h5")
```

The model will be saved with a file named "LSTM_sentiment_analysis_model.h5" in the current directory. The ".h5" extension indicates that it's saved in the Hierarchical Data Format (HDF5) format, which is commonly used for saving Keras models.

All the other codes are provided in the .ipynb file with my submission.

F: Functionality

This network architecture is designed to effectively process and analyze customer reviews, providing sentiment predictions based on the learned patterns in the dataset. The embedding layer helped the model learn specific representations for the words in the Amazon dataset, enabling it to capture the unique semantics and patterns in customer reviews. With a smaller dataset like the one provided (1000 reviews), the embedding layer helps the model generalize better by capturing the underlying semantics and relationships between words.

The LSTM layers allowed the model to leverage the sequential dependencies and long-term context present in the customer reviews, improving its ability to identify sentiment-bearing words and phrases. With binary sentiment classification (positive or negative), the sigmoid activation function in the output layer helped the model assign a probability of positive sentiment to each review.

G: Recommendation

Based on all the above analysis and results -

The model can be utilized to predict whether a customer would recommend an Amazon product based on their previous review. This model shows promising results in predicting sentiment and classifying customer reviews. The precision, recall, and F1-score metrics indicate a good balance between correctly identifying positive and negative sentiments.

However, with this type of relatively small dataset size like 1000 reviews, the network architecture should be carefully designed to avoid overfitting. Regularization techniques such as dropout and early stopping help to prevent the model from memorizing the limited training data and ensure generalization to unseen reviews. Hyperparameter tuning and experimentation may also be necessary to optimize the network architecture's performance on the specific Amazon dataset.

It is important to continuously monitor and evaluate the performance of the sentiment analysis model as new customer reviews become available. Also, Regular retraining and updating of the model with new data can ensure that it remains accurate and aligned with the evolving customer sentiments and language patterns.

By leveraging this sentiment analysis model, the company can make data-driven decisions to enhance customer satisfaction, product quality, and overall business performance.

Part VI – Reporting

H. Report

The final report in an HTML document is provided with the submission.

I. Third-Party Code References

Parikh, A. B. (n.d.). Machine Learning & Deep Learning in Python & R. https://wgu.udemy.com/course/data_science_a_to_z/learn/lecture/20811912#overview

WGU, R. (n.d.). D213 Sentiment Analysis I. https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=a1e9802c-5808-4287-a7e5-aed601580d89

Resources, W. (n.d.). D213 T2 May 15 22. https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=cedbd86a-2543-4d9d-9b0e-aec4011a606d

Terry-Jack, M. (2019, Apr 21). NLP: Everything about Embeddings. https://medium.com/@b.terryjack/nlp-everything-about-word-embeddings-9ea21f51ccfe

J. References

Virahonda, S. (2020, Oct 8). An easy tutorial about Sentiment Analysis with Deep Learning and Keras. https://towardsdatascience.com/an-easy-tutorial-about-sentiment-analysis-with-deep-learning-and-keras-2bf52b9cba91

Brownlee, J. (2020, August 25). Use Early Stopping to Halt the Training of Neural Networks At the Right Time. https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/