Sentiment Analysis Using Neural Networks and NLP

Part I Research Question

A1. Research Question:

Can we analyze customers' sentiment from the dataset using NN and NLP?

A2. Objectives:

1. Develop a model to predict positive or negative sentiment from reviews.

2. Summarize the results of the model and recommend a course of action

A3. Neural Network Type:

I chose to use a bidirectional long short-term memory network. This network specializes in processing text data in both directions. This allows for greater context which is important when trying to determine if a review is positive or negative.

In [5]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
import h5py
```

In [6]:
```
# Path to the text file
txt_file = 'C:\\Users\\jhall\\Desktop\\D213\\d213Part2.txt'
csv_file = 'C:\\Users\\jhall\\Desktop\\D213\\D213Part2.csv'

# Load the .txt file
try:
    df = pd.read_csv(txt_file, delimiter="\t", header=None, on_bad_lines="skip", engir
except Exception as e:
    print(f"Error reading the file: {e}")
    exit()

# Add column names
df.columns = ['Review', 'Sentiment']
```

```
# Save as CSV
df.to_csv(csv_file, index=False)

print(f"File successfully converted and saved as {csv_file}")
```

```
File successfully converted and saved as C:\Users\jhall\Desktop\D213\D213Part2.csv
```

In [7]:
```
# Upload CSV
data = pd.read_csv('C:\\Users\\jhall\\Desktop\\D213\\D213Part2.csv')

# Display the first few rows
print(data.head())
```

```
                                              Review  Sentiment
0  So there is no way for me to plug it in here i...          0
1                        Good case, Excellent value.          1
2                             Great for the jawbone.          1
3  Tied to charger for conversations lasting more...          0
4                                   The mic is great.          1
```

In [8]:
```
# Check for null values
print(f"Null values:\n{data.isnull().sum()}")

# Drop null values
data = data.dropna()

# Remove duplicates
data = data.drop_duplicates()

print(f"Dataset shape after cleaning: {data.shape}")
```

```
Null values:
Review       0
Sentiment    0
dtype: int64
Dataset shape after cleaning: (1730, 2)
```

Part II Data Preperation

B1. Exploatory Analysis

The code below standardizes the text by removing all non-alphanumeric character, white space, numbers, and converts capital letters to lowercase. Cleaning the data in this way is critical as it ensure the input is valuable and consistent. The model will know "Awesome" and "awesome" are the same word. This will also increase the efficiency of the model as the extra inputs and noise can unnecessarily increase complexity.

In [10]:
```
import re

# Clean the text: lowercase, remove special characters
def clean_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    text = re.sub(r'\d+', '', text)  # Remove numbers
    return text
```

```
data['Review'] = data['Review'].apply(clean_text)

# Display cleaned reviews
print(data.head())
```

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

B1. Exploatory Analysis (cont.) & B2. Tokenization

The vocabulary size of the data set is 3805. The tokenizer calculates the top 10,000 words in the data set and assigns them tokens. The goal of tokenization is to convert the text data into numerical data that the model can process. We limit the number of words to 10,000 so that if the model were deployed on a larger data set it would focus on the most relevant data and it also is one lever to prevent overfitting. Anything outside the top 10,000 will get assigned an <oov>. I then sequenced the data which converts the words to their assigned tokens.

Later I will choose to a value to embed these tokens. Embedding assigns vector values to the tokens which allows the model to "learn" what words have similar or opposite meanings such as "happy" and "sad" by adding vector values that represent their relationship to the overall data. Since our vocabulary is relatively small in terms of NN and NLP I chose an embedding dimension of 50.

In [12]:
```
# Limit vocabulary size to 10,000 and handle out-of-vocabulary tokens
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(data['Review'])  # Fit tokenizer on the reviews

# Convert reviews to sequences
sequences = tokenizer.texts_to_sequences(data['Review'])

# Save the word index for reference
word_index = tokenizer.word_index
print(f"Vocabulary size: {len(word_index)}")
```

```
Vocabulary size: 3805
```

In [13]:
```
print(f"Example review: {data['Review'][0]}")
print(f"Example sequence: {sequences[0]}")
```

```
Example review: so there is no way for me to plug it in here in the us unless i go by a converter
Example sequence: [32, 51, 6, 53, 118, 14, 68, 9, 236, 7, 11, 164, 11, 2, 355, 693, 5, 191, 61, 4, 1514]
```

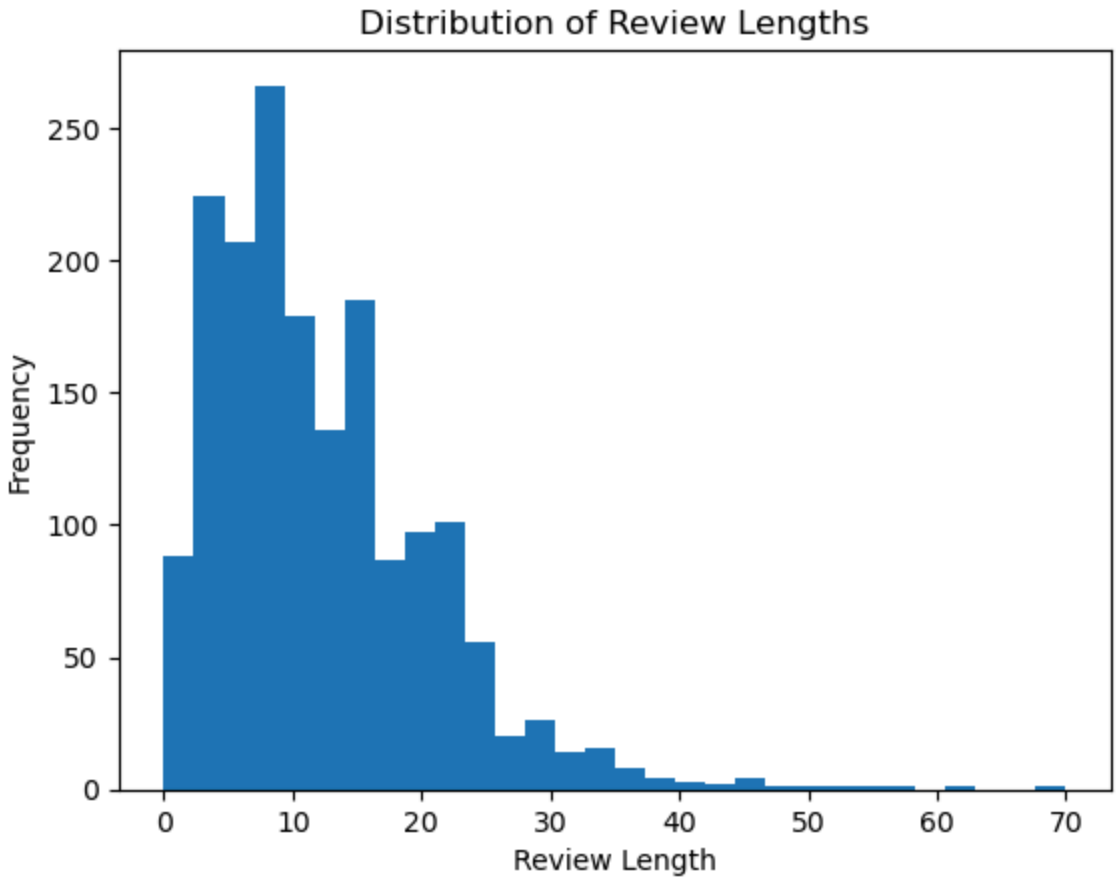
B1. Exploatory Analysis (cont.) & 3

To choose the maximum sequence length I charted all the sequences on a histogram. Visually we see most of the sequences are 30 or below. This will capture a significant portion of the data and standardizes the input while allowing for the model to run efficiently. To ensure the sequences are all 30 values in length we will add 0s to the front of sequences that are shorter and truncate the longer sequences at 30.

In [15]:
```
# Check the distribution of review lengths
review_lengths = [len(seq) for seq in sequences]

plt.hist(review_lengths, bins=30)
plt.xlabel("Review Length")
plt.ylabel("Frequency")
plt.title("Distribution of Review Lengths")
plt.show()

# Calculate a reasonable maximum sequence length
max_length = 30  # based on histogram
print(f"Chosen maximum sequence length: {max_length}")
```




```
Chosen maximum sequence length: 30
```

In [16]:
```
# Pad sequences to the same length
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='pre', truncati

# Display an example padded sequence
print(f"Example padded sequence: {padded_sequences[0]}")
```

```
Example padded sequence: [   0    0    0    0    0    0    0    0    0   32   51    6
53  118
   14   68    9  236    7   11  164   11    2  355  693    5  191   61
    4 1514]
```

B4 & B5 Categories, Activation Function, Data Preperation Summary

There are two categories (1) which is positive sentiment and (0) which is negative sentiment. Given the binary nature a sigmoid activation function will be used in the final dense layer. The sigmoid activation function will produce a probability between 0 and 1 which allows for simple classification of >=0.5 = positive and <0.5 is negative.

To recap the data preparation to this point. I've cleaned the data by removing null values, duplicates, special characters, numbers and punctuation. I standardized the data by making it all lower case, tokenizing the sentences into sequences and then padding them into uniform 30 token length sequences. Next, I will split the data into 80% training (1384 samples), 10% test (173 samples), and 10% validation (173 samples). These are based on industry standards.

In [18]:
```
labels = data['Sentiment'].values

# Split data (80% training, 10% validation, 10% test)
X_train, X_temp, y_train, y_temp = train_test_split(padded_sequences, labels, test_siz
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_

print(f"Training set size: {len(X_train)}")
print(f"Validation set size: {len(X_val)}")
print(f"Test set size: {len(X_test)}")
```

```
Training set size: 1384
Validation set size: 173
Test set size: 173
```

B6 Export Clean Data Set

Clean csv will be attatched.

In [39]:
```
np.savez_compressed('prepared_reviews_data.csv',
                    X_train=X_train, y_train=y_train,
                    X_val=X_val, y_val=y_val,
                    X_test=X_test, y_test=y_test)

#Converting the array to DF
padded_df = pd.DataFrame(padded_sequences, columns=[f"Token_{i+1}" for i in range(padd

padded_df['Sentiment'] = labels

file_location = "C:\\Users\\jhall\\Desktop\\D213\\prepared_reviews_data.csv"

padded_df.to_csv(file_location, index=False)
```

```
print("Prepared data saved as 'prepared_reviews_data.csv'")
```

```
Prepared data saved as 'prepared_reviews_data.csv'
```

In [ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```

In [ ]:
```
# Load the prepared data
data = np.load('prepared_reviews_data.npz')
X_train, y_train = data['X_train'], data['y_train']
X_val, y_val = data['X_val'], data['y_val']
X_test, y_test = data['X_test'], data['y_test']

print(f"Training data shape: {X_train.shape}, Labels shape: {y_train.shape}")
print(f"Validation data shape: {X_val.shape}, Labels shape: {y_val.shape}")
print(f"Test data shape: {X_test.shape}, Labels shape: {y_test.shape}")
```

C1. Model Summary

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM, Dense, Dropout
from tensorflow.keras.regularizers import l2

# Model parameters
vocab_size = 3806  # Including <OOV> token
embedding_dim = 50
max_length = 30  # Same as padding length

# Build the BiLSTM model
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length)
    Bidirectional(LSTM(64, return_sequences=False)),  # BiLSTM with 64 units
    Dropout(0.6),  # Regularization to avoid overfitting
    Dense(32, activation='relu', kernel_regularizer=l2(0.005)),  # Dense layer with L2
    Dropout(0.5),  # Additional regularization
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])

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


# Display model summary
model.summary()
```

C2. Layers

The first layer is the embedding layer which converts tokens into dense vectors with 50 values. We determined this value earlier based on the size of the data set and vocabulary. The output shape is determined by the sequence length (30) and the embedding size (50) which gives us the total parameter number of 190300.

The second layer it the bidirectional LSTM layer. This is the part of the model that processes the sequences forwards and backwards. The output shape is determined by the number of units assigned to the model. In this case it is 64 and since it is BiLSTM is 64 forward and 64 backwards. features the model will learn per time step. This layer contains 58880 total parameters.

The third and fifth layers are a dropout layer. This is to help avoid overfitting. In this model 60% of the neurons were randomly dropped in the third layer and 50% were dropped in the fifth layer. There are no parameters because these layers don't add or learn. The fourth layer is a dense layer with 32 fully connected neurons. This layer processes and combine the features learned in the previous layers to produce a high-level perception of the data. The final layer uses 1 neuron that combines the 32 features in the previous hidden dense layer and produces one probability output using the sigmoid activation.

C3. Justification

1. Activation Functions: The first activation function used in the hidden dense layer is a ReLU function. The ReLu function only retains positive values to only focus on important signals and ignore irrelevant ones. This helps the network leanr more efficently. The sigmoid function in the last layer produces a probablitity between 0 and 1 which is perfect for the binary classification.

2. Nodes per Layer:

Embedding Layer:
I chose 50 dimensions for the embedding layer because it is good for smaller datasets. This number is a good balance between capturing word relationships and keeping the model efficient without adding complexity.

BiLSTM Layer:
I used 64 nodes in the BiLSTM layer because this layer processes the data in both forward and backward directions, effectively using 128 nodes (64 forward + 64 backward). This allows the model to understand context from both sides of a sentence, like in "not good" vs. "good not." This number balances pattern identification and computational efficiency.

Dense Layer:
The dense layer has 32 nodes, which is a good choice for summarizing patterns extracted by the BiLSTM layer. It provides a decent trade-off between accuracy and efficiency, ensuring the model doesn't overfit while retaining enough capacity to make accurate predictions.

Loss Function:
The loss function is binary cross-entropy, which is specifically designed for binary classification problems (e.g., positive vs.

negative). It calculates the error between the predicted probabilities and actual labels. Additionally, binary cross-entropy punishes the model more for confident but wrong predictions, encouraging the model to output probabilities that are more accurate.

Optimizer:
I used the Adam optimizer, which combines adaptive learning rates and momentum. Adaptive learning rates adjust how quickly the model learns, depending on the data, while momentum helps the optimizer make smoother adjustments by remembering past updates. Adam is highly efficient and works particularly well for text-based problems like sentiment analysis.

Stopping Criteria:
I applied early stopping, which monitors the validation loss during training. If the validation loss doesn't improve after a set number of epochs (patience), then the training stops. This prevents the model from overfitting to the training data and saves computational resources.

Evaluation Metric:
The model is evaluated using accuracy, which measures the percentage of correct predictions. Accuracy is a straightforward metric for binary classification tasks. This model achieved a test accuracy of 74.57%, meaning it correctly predicts the sentiment of a review about 75% of the time.

In [ ]:
```
early_stopping = EarlyStopping(
    monitor='val_loss', patience=1, restore_best_weights=True
)
```

D1 & D2 Stopping Criteria and Overfitting

I applied early stopping, which monitors the validation loss during training. If the validation loss doesn't improve after a set number of epochs (patience), then the training stops. This prevents the model from overfitting to the training data and saves computational resources. The final epoch is 7 and you can see in the output that the training stopped the next epoch after val_loss increased.

The model's fitness was assessed by evaluating accuracy and validation loss. The test accuracy is 74.57% which is acceptable performance. I took several measures to prevent overfitting. One is utilizing early stopping which prevents over fitting by stopping when validation loss starts to increase. I added dropout layers which randomly deactivate neurons during training. This helps generalize the model by ensuring it is not too reliant on any specific neurons. Third, I used L2 regularization on the dense layer. L2 regularization prevents overfitting by penalizing large weights. This ensures the model does not learn too specifically to the training data. Lastly, splitting the data into training, validation, and test data ensure the model is evaluated on "new" data.

In [ ]:
```
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=10,
    batch_size=64,
    callbacks=[early_stopping]
)
```

D3 Visualization of Training Process

In [ ]:
```
import matplotlib.pyplot as plt

# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Over Epochs')
plt.show()

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Over Epochs')
plt.show()
```

D4 Model Accuracy

The model is evaluated using accuracy, which measures the percentage of correct predictions. Accuracy is a straightforward metric for binary classification tasks. This model achieved a test accuracy of 74.57%, meaning it correctly predicts the sentiment of a review about 75% of the time.

In [ ]:
```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

E. Saving Trained Newtowrk

In [ ]:
```
model.save('bilstm_sentiment_model.keras')
print("Model saved as 'bilstm_sentiment_model.keras'")
```

In [ ]:
```
from tensorflow.keras.models import load_model

loaded_model = load_model('bilstm_sentiment_model.keras')
```

In [ ]:
```
# Example new reviews
new_reviews = ["This product is amazing!", "Terrible experience, not recommended."]
```

```
new_sequences = tokenizer.texts_to_sequences(new_reviews)
new_padded = pad_sequences(new_sequences, maxlen=max_length, padding='pre')

# Predict
predictions = model.predict(new_padded)
print(f"Predictions: {predictions}")
```

F. Neural Network Functionality

The model predicts if a review is positive or negative at a 75% rate. The main architecture that impacts model performance would be the BiLSTM layer which looks at the data in both directions. This helps the model learn context. Also, the dense layers and dropout layers add higher level features and combats overfitting in such a small dataset. Overall, the model performs well.

G. Recommendations

I recommend this model could be used for sentiment analysis. There are several things that could be done to improve the model. The first thing would be to expand the dataset and tune the relevant hyperparameters accordingly.

H. Project is completed in Jupyter Notebook

I & J Sources

Keras. (n.d.). Keras: Deep learning for humans. Retrieved from https://keras.io

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. arXiv. Retrieved from https://arxiv.org/abs/1412.6980

UCI Machine Learning Repository. (n.d.). Sentiment labelled sentences data set. Retrieved from https://archive.ics.uci.edu/dataset/331/sentiment+labelled+sentences

Chollet, F. (2017). Deep learning with Python. Manning Publications.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.

TensorFlow. (n.d.). An end-to-end open source machine learning platform. Retrieved from https://www.tensorflow.org

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. Retrieved from https://scikit-learn.org/stable/

Zhang, Y., Jin, R., & Zhou, Z.-H. (2010). Understanding bag-of-words model: A statistical framework. International Journal of Machine Learning and Cybernetics, 1(1–4), 43–52.

https://doi.org/10.1007/s13042-010-0001-0

In [ ]: