Rancel Hernandez

D213 Advanced Data Analytics

02/20/2025

TASK 2: SENTIMENT ANALYSIS USING NEURAL

A1: RESEARCH QUESTION

Which neural network architecture, sequential or functional, more accurately and confidently predicts sentiment in review data to help organizations better understand user sentiment?

A2: OBJECTIVES OR GOALS

The main goal of the data analysis is to compare two types of neural network architectures of similar complexity to assess which is better at predicting sentiment in reviews from various sources. This will be accomplished by optimizing a dual-branch functional model for the task, and then creating a basic sequential model of a comparable complexity. Both models will be trained on the same dataset and evaluated based on their performance metrics.

A functional model is flexible in its design, allowing for distinct branches and complex connections between layers and nodes [1]. In contrast, a sequential model follows a simpler approach where each layer is connected in order from the input to the output. What will set these two models apart is that the functional model will have two distinct branches, each designed to capture opposing patterns that may be associated with different sentiments.

The UCI Sentiment Labeled Sentences dataset contains reviews from three sources: Amazon, IMDb, and Yelp. To ensure there is sufficient training data, all sources will be combined into one, allowing the models to generalize better across different domains.These goals align with the scope of the dataset, and achieving them could provide insights into how model architecture influences classification performance in sentiment analysis, potentially enhancing organizations' understanding of user sentiment in reviews and leading to a more effective and efficient neural network design.

A3: PRESCRIBED NETWORK

The functional and sequential models used in this analysis are hybrid recurrent and feedforward neural networks designed for text classification. In this context, a recurrent neural network maintains memory by passing hidden states between the word embeddings, while a feedforward neural network processes the data sequentially through the layers without maintaining temporal dependencies [2 - 3]. Specifically, these models are hybrid because they include a recurrent component in the form of a Bidirectional Long Short-Term Memory (BiLSTM) layer, while passing its output through fully connected dense layers that follow a feedforward architecture.

These models require the input data to be in a numeric format, meaning the text reviews must be transformed using Natural Language Processing (NLP) techniques. This process includes using the pretrained word vectors from the spaCy en_core_web_md model to convert the reviews into sequences of word embeddings [4 - 5]. In simpler terms, these word vectors contain numerical values that capture semantic relationships between words in the model's vocabulary, allowing the network to associate words based on their meaning. Additionally, the en_core_web_md model includes approximately 685,000 word vectors, ensuring that the majority of words in the reviews can be tokenized.

Regarding the architecture of the models, the BiLSTM layer allows them to retain and utilize past and future context in a sequence for their predictions [6]. This layer can be beneficial in a sentiment analysis because context can significantly alter the meaning of words. For example, in the sentence “The movie was not bad,” analyzing the words “not” and “bad” individually could suggest negative sentiment, but together they convey positive sentiment. The BiLSTM layer ensures the model captures this contextual dependency and improves the accuracy of sentiment classification. Then, the output of the BiLSTM is passed through a Flatten layer, which converts its 3D sequential output into a 1D array for the following dense layers. This transformation removes the sequential structure of the data, leaving behind only the extracted features that capture the contextual relationships between past and future values.

Additionally, the functional model includes two distinct branches after the BiLSTM layer to potentially capture distinct patterns associated with each sentiment. There are two sentiments that will be predicted in this analysis: positive and negative. What sets these branches apart are their activation functions because one will use the standard Rectified Linear Unit (ReLU) function and the other will use the inverse ReLU. The ReLU function forces the model to focus on positive values, and it is defined as x if $x \geq 0$, and 0 if $x < 0$ [7]. In contrast, the inverse ReLU does the opposite and emphasizes negative values, with the definition x if $x \leq 0$, and 0 if $x > 0$.

The goal of this design is to specialize each branch in processing complementary patterns. The standard ReLU passes positive values while zeroing negative inputs, forcing one branch to learn from a specific subset of activation patterns [7]. Conversely, the inverse ReLU passes negative values while zeroing positive inputs, causing the other branch to learn from an

entirely different set of patterns. Input ignored in one function is exclusively processed in the other, so the branches are forced to learn from completely independent information.
This flexible architecture could improve the accuracy of the sentiment classifications by specializing each branch as opposed to a fully connected sequential model that only focuses on positive patterns. However, it is unclear whether the learned information from the inverse ReLU is informative for classifying sentiment. Therefore, this analysis aims to determine whether this dual-branch specialization improves performance compared to a traditional fully connected approach.

B1: DATA EXPLORATION

Using the combined domains, the reviews were analyzed to identify unusual characters, determine the vocabulary size, and establish an appropriate maximum sequence length for the sequences. Unusual characters can include emojis or non-English characters, and they were detected using pattern recognition from Python's re package. The pattern, [^a-z0-9\s], was initialized to find non-English characters, where the '^' tophat signifies negation to lowercase 'a-z,' digits '0-9,' and whitespaces '\s.' This pattern alone does not take punctuation into account, so a separate set with all the common punctuation symbols was set to exclude those as unusual since they are common in English sentences. Each review was passed through the pattern in lowercase, and 349 unusual characters were identified, including ['ê', 'é', '\x96', 'å', '\x97']. There were no emojis found, and the detected characters did not appear to be significant to the sentiment analysis. Therefore, since the unusual characters accounted for less than 1% of the dataset, they were removed for data consistency.

The dataset contained a total of 36,988 words, excluding unusual characters. Of that total, 35,557 words were found in the spaCy en_core_web_md model's vocabulary, while 1,431 words were not recognized. In total, there were 5,472 unique words across all the reviews. After tokenizing the sentences, the sequence lengths were analyzed to understand the distribution of the review lengths. There were five extreme outliers identified, ranging from 262 to 1,842 words, while the average sequence length was 15 words. Given their significant deviation, these outliers were removed to prevent them from skewing statistical properties. The proposed word embedding length was 300 values because that is the default length for the spaCy en_core_web_md model's pretrained embeddings. The word embedding sizes can vary, with lower dimensions, around 100, reducing computational cost but capturing less semantic information [8]. Therefore, the default size of 300 is large enough to provide sufficient semantic information for the analysis.

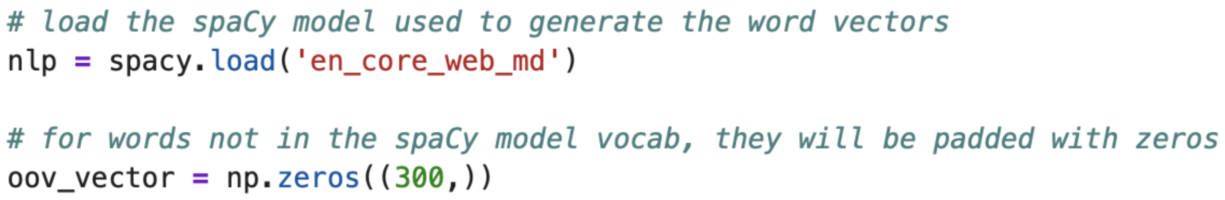
To determine the maximum sequence length, the 95th percentile of sequence lengths was calculated, resulting in a threshold of 31 words. This selection was statistically justified because it ensured that 95% of the data was preserved, with only 5% of the longest reviews being truncated. Choosing an appropriate maximum sequence length is critical for analysis because setting a too low threshold would lead to a significant information loss, while setting a too high threshold would result in excessive padding. Padded sequences are word vectors filled with zeros, serving as placeholders for sequences shorter than the maximum embedding threshold. They ensure that all sequences have a uniform length for the model training.

B2: TOKENIZATION

The goals of the tokenization process were to transform the text reviews into sequences of word embeddings while handling unusual characters and out-of-vocabulary (OOV) words. The full implementation of the tokenization process is in Cell 13 of the Jupyter Notebook that will be submitted separately, with relevant code snippets shown in *Figures 1 - 4*. The spaCy en_core_web_md model was chosen for the tokenization because of its large vocabulary of approximately 685,000 words, ensuring most words would be mapped to meaningful embeddings [*Figure 1*].

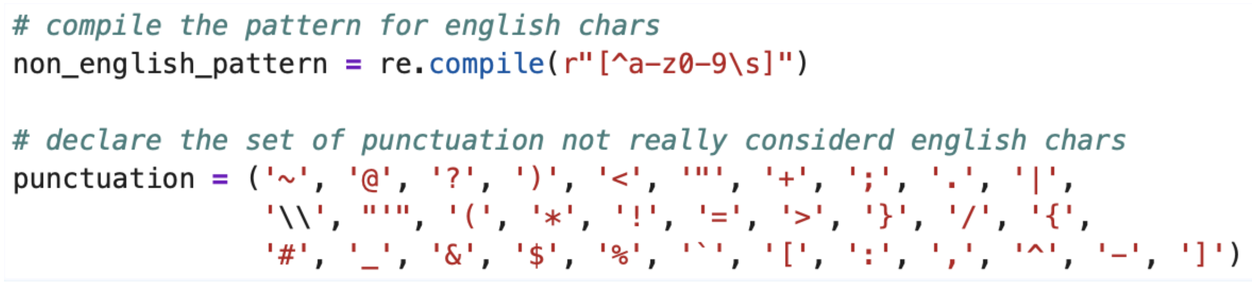
Python's re package was used to implement pattern recognition by passing the sentences through the [^a-z0-9\s] pattern to identify non-English characters and the following were identified: ['ê', 'é', '\x96', 'å', '\x97'] [*Figure 2 - 3*]. Since these accounted for less than 1% of the dataset, they were removed to maintain consistency.

Once the sentences were normalized for tokenization, the spaCy model was used to create a document object containing tokens for each word in a review [*Figure 4*]. Among the 36,988 words in the dataset, only 1,431 words were not found in the model's vocabulary. Since these words could not be mapped to meaningful word embeddings, these OOV tokens were replaced with zero vectors to exclude them from contributing to the analysis. After manually padding the OOV words, the resulting sequences of word embeddings were appended to a list for further processing.



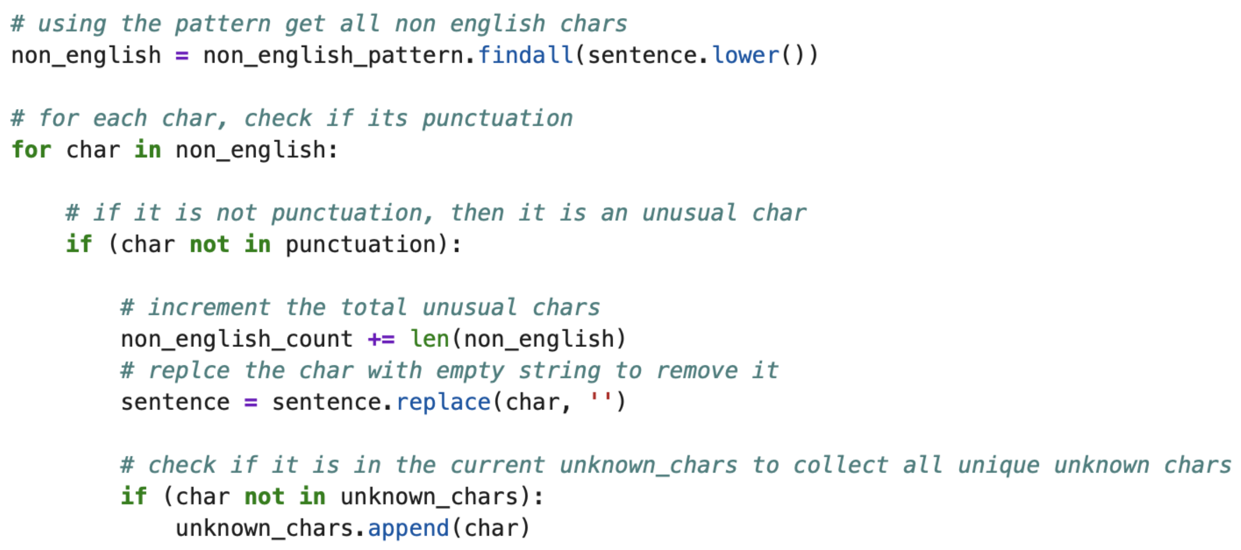
```python
# load the spaCy model used to generate the word vectors
nlp = spacy.load('en_core_web_md')

# for words not in the spaCy model vocab, they will be padded with zeros
oov_vector = np.zeros((300,))
```

Figure 1: The code snippet for loading the spaCy en_core_web_md model used to tokenize the sentences, and the initialization of the OOV vector to manually pad OOV words.



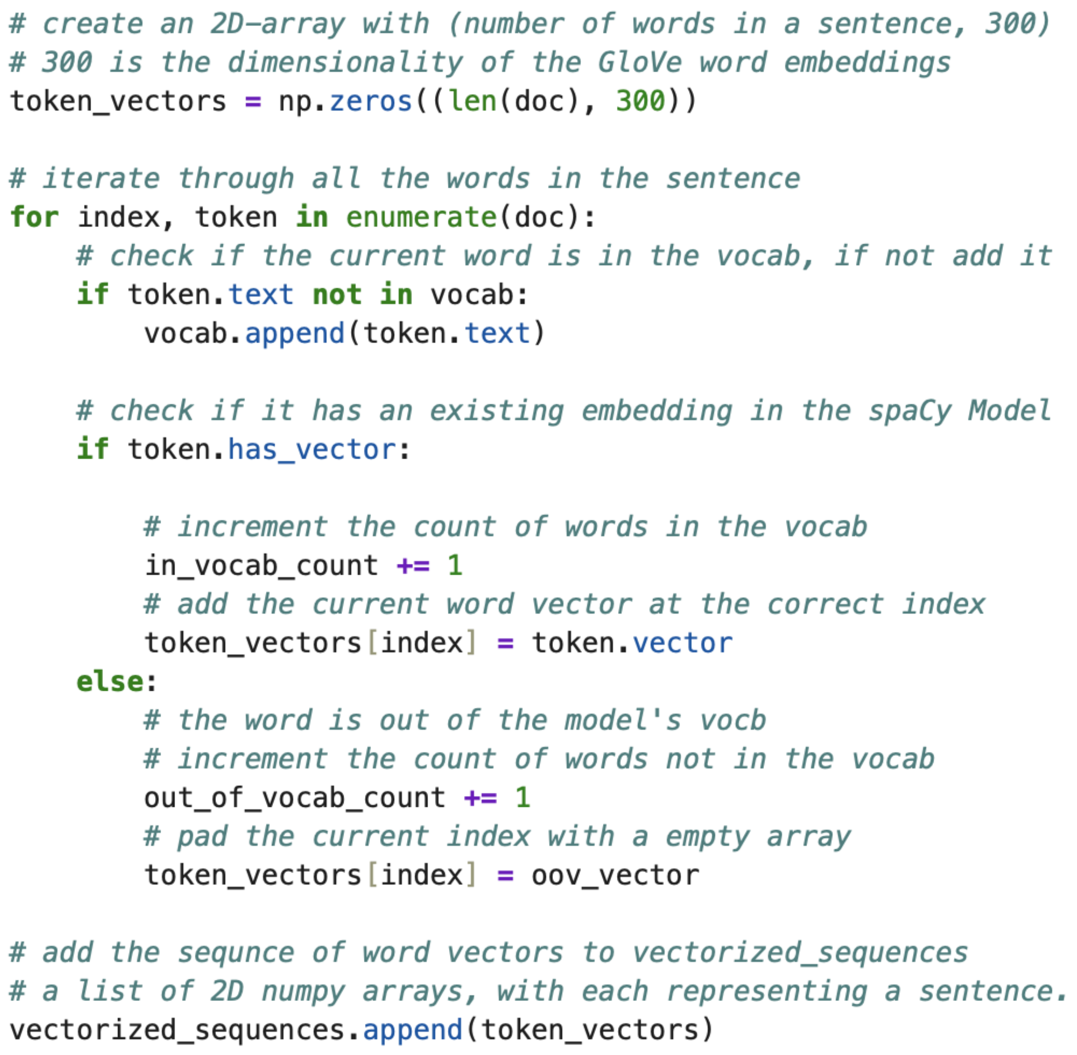
```python
# compile the pattern for english chars
non_english_pattern = re.compile(r"[^a-z0-9\s]")

# declare the set of punctuation not really considerd english chars
punctuation = ('~', '@', '?', ')', '<', '"', '+', ';', '.', '|',
               '\\', "'", '(', '*', '!', '=', '>', '}', '/', '{',
               '#', '_', '&', '$', '%', '`', '[', ':', ',', '^', '-', ']')
```

Figure 2: The code snippet to compile the re pattern used to identify non-English characters, and the punctuation set to disregard common English punctuation.



```python
# using the pattern get all non english chars
non_english = non_english_pattern.findall(sentence.lower())

# for each char, check if its punctuation
for char in non_english:

    # if it is not punctuation, then it is an unusual char
    if (char not in punctuation):

        # increment the total unusual chars
        non_english_count += len(non_english)
        # replce the char with empty string to remove it
        sentence = sentence.replace(char, '')

        # check if it is in the current unknown_chars to collect all unique unknown chars
        if (char not in unknown_chars):
            unknown_chars.append(char)
```

Figure 3: The code snippet demonstrating how the sentences are processed using the re pattern to identify non-English characters while excluding common English punctuation. Unusual characters are removed by replacing them with an empty string and are then appended to a list containing the unique unusual characters.



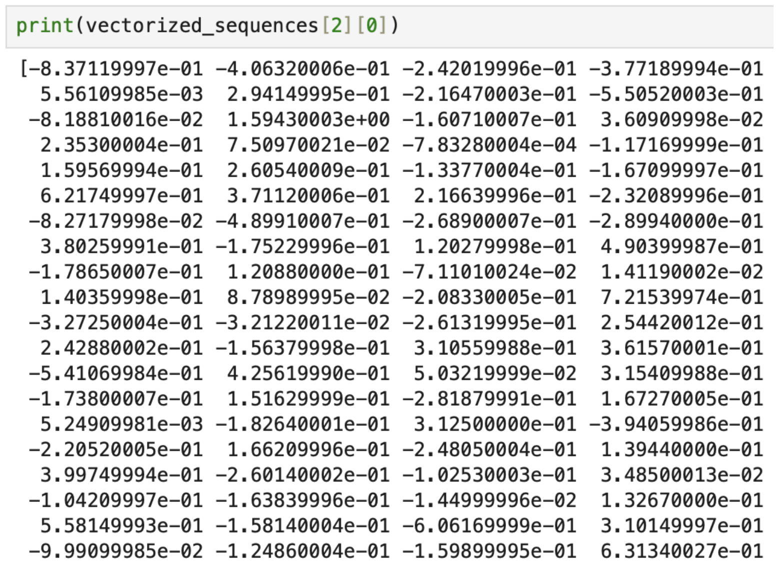
```python
# create an 2D-array with (number of words in a sentence, 300)
# 300 is the dimensionality of the GloVe word embeddings
token_vectors = np.zeros((len(doc), 300))

# iterate through all the words in the sentence
for index, token in enumerate(doc):
    # check if the current word is in the vocab, if not add it
    if token.text not in vocab:
        vocab.append(token.text)

    # check if it has an existing embedding in the spaCy Model
    if token.has_vector:

        # increment the count of words in the vocab
        in_vocab_count += 1
        # add the current word vector at the correct index
        token_vectors[index] = token.vector
    else:
        # the word is out of the model's vocb
        # increment the count of words not in the vocab
        out_of_vocab_count += 1
        # pad the current index with a empty array
        token_vectors[index] = oov_vector

# add the sequnce of word vectors to vectorized_sequences
# a list of 2D numpy arrays, with each representing a sentence.
vectorized_sequences.append(token_vectors)
```

Figure 4: The code snippet iterates through all tokens to check whether they are found in the spaCy model's vocabulary. If a token is not found, then it is manually padded. The token_vectors variable is initialized with the appropriate length and dimensions to correctly store each token's vector at its corresponding index. Additionally, each token is checked against the vocabulary to compile a list of unique words in the dataset, and the count variables are incremented to track the number of words in and out of the vocabulary.

B3: PADDING PROCESS

The sequence of word embeddings was padded using the pad_sequences function from TensorFlow. This function takes in the word embeddings, the maximum sequence threshold, and the padding order [9]. For this analysis, the maximum sequence length was set to 31, based on the 95th percentile of sequence lengths, and the padding was added at the end of the sequences. To validate this process, an example of a short sequence significantly below the maximum threshold was analyzed before and after padding. Since the order of sequences in vectorized_sequences and padded_vectorized_sequences remained unchanged, the second sequence was chosen for analysis because it originally had a length of 7 before padding and was well below the threshold.

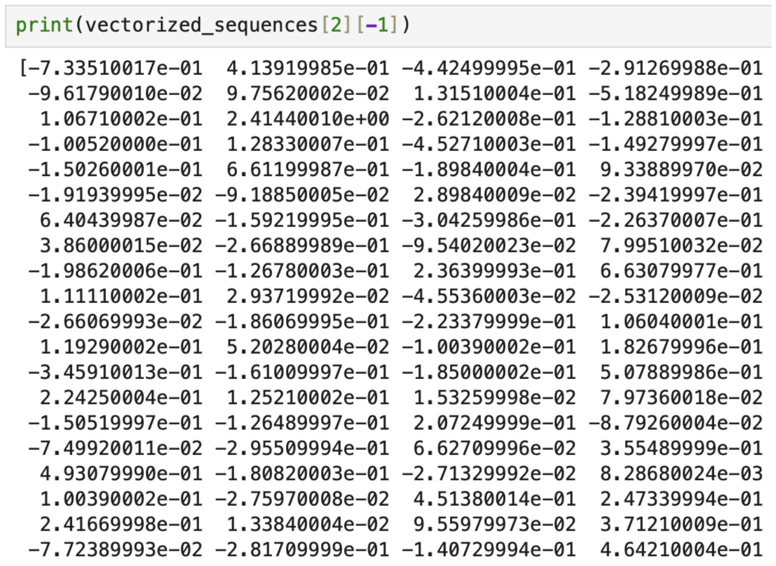
Each word embedding includes 300 values, and each sequence contains 31 word vectors post-padding. Since printing an entire sequence is impractical, selected portions of the sequence were screenshotted to provide sufficient evidence of a successful padding implementation. In *Figures 5 - 6*, the first few values of the first and last word embeddings of the second sequence are shown, confirming that they are not all zeros. After padding, the first few values of the last word embedding for the same sequence contain only zeros, verifying that padded vectors were correctly added to the end of the sequence, as shown in *Figure 7.* Additionally, *Figure 8* confirms the change in sequence length before and after padding. The second sequence initially contained 7 word embeddings and increased to 31 word embeddings post-padding, further validating the implementation of the padding process.



```
print(vectorized_sequences[2][0])
```

```
[-8.37119997e-01 -4.06320006e-01 -2.42019996e-01 -3.77189994e-01
  5.56109985e-03  2.94149995e-01 -2.16470003e-01 -5.50520003e-01
 -8.18810016e-02  1.59430003e+00 -1.60710007e-01  3.60909998e-02
  2.35300004e-01  7.50970021e-02 -7.83280004e-04 -1.17169999e-01
  1.59569994e-01  2.60540009e-01 -1.33770004e-01 -1.67099997e-01
  6.21749997e-01  3.71120006e-01  2.16639996e-01 -2.32089996e-01
 -8.27179998e-02 -4.89910007e-01 -2.68900007e-01 -2.89940000e-01
  3.80259991e-01 -1.75229996e-01  1.20279998e-01  4.90399987e-01
 -1.78650007e-01  1.20880000e-01 -7.11010024e-02  1.41190002e-02
  1.40359998e-01  8.78989995e-02 -2.08330005e-01  7.21539974e-01
 -3.27250004e-01 -3.21220011e-02 -2.61319995e-01  2.54420012e-01
  2.42880002e-01 -1.56379998e-01  3.10559988e-01  3.61570001e-01
 -5.41069984e-01  4.25619990e-01  5.03219999e-02  3.15409988e-01
 -1.73800007e-01  1.51629999e-01 -2.81879991e-01  1.67270005e-01
  5.24909981e-03 -1.82640001e-01  3.12500000e-01 -3.94059986e-01
 -2.20520005e-01  1.66209996e-01 -2.48050004e-01  1.39440000e-01
  3.99749994e-01 -2.60140002e-01 -1.02530003e-01  3.48500013e-02
 -1.04209997e-01 -1.63839996e-01 -1.44999996e-02  1.32670000e-01
  5.58149993e-01 -1.58140004e-01 -6.06169999e-01  3.10149997e-01
 -9.99099985e-02 -1.24860004e-01 -1.59899995e-01  6.31340027e-01
```

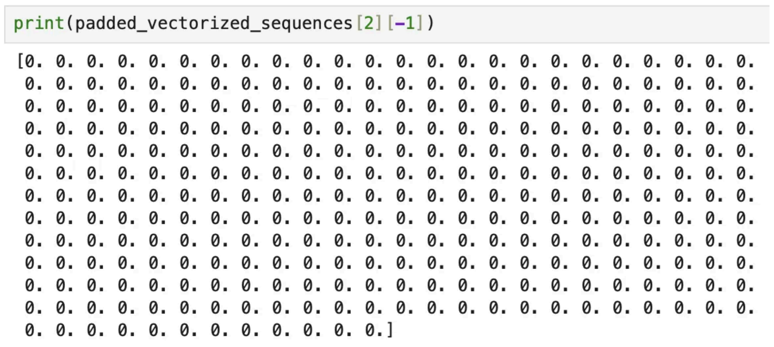
Figure 5: The first few values of the first word embedding from the second sequence in vectorized_sequences.



```
print(vectorized_sequences[2][-1])
```

```
[-7.33510017e-01  4.13919985e-01 -4.42499995e-01 -2.91269988e-01
 -9.61790010e-02  9.75620002e-02  1.31510004e-01 -5.18249989e-01
  1.06710002e-01  2.41440010e+00 -2.62120008e-01 -1.28810003e-01
 -1.00520000e-01  1.28330007e-01 -4.52710003e-01 -1.49279997e-01
 -1.50260001e-01  6.61199987e-01 -1.89840004e-01  9.33889970e-02
 -1.91939995e-02 -9.18850005e-02  2.89840009e-02 -2.39419997e-01
  6.40439987e-02 -1.59219995e-01 -3.04259986e-01 -2.26370007e-01
  3.86000015e-02 -2.66889989e-01 -9.54020023e-02  7.99510032e-02
 -1.98620006e-01 -1.26780003e-01  2.36399993e-01  6.63079977e-01
  1.11110002e-01  2.93719992e-02 -4.55360003e-02 -2.53120009e-02
 -2.66069993e-02 -1.86069995e-01 -2.23379999e-01  1.06040001e-01
  1.19290002e-01  5.20280004e-02 -1.00390002e-01  1.82679996e-01
 -3.45910013e-01 -1.61009997e-01 -1.85000002e-01  5.07889986e-01
  2.24250004e-01  1.25210002e-01  1.53259998e-02  7.97360018e-02
 -1.50519997e-01 -1.26489997e-01  2.07249999e-01 -8.79260004e-02
 -7.49920011e-02 -2.95509994e-01  6.62709996e-02  3.55489999e-01
  4.93079990e-01 -1.80820003e-01 -2.71329992e-02  8.28680024e-03
  1.00390002e-01 -2.75970008e-02  4.51380014e-01  2.47339994e-01
  2.41669998e-01  1.33840004e-02  9.55979973e-02  3.71210009e-01
 -7.72389993e-02 -2.81709999e-01 -1.40729994e-01  4.64210004e-01
```

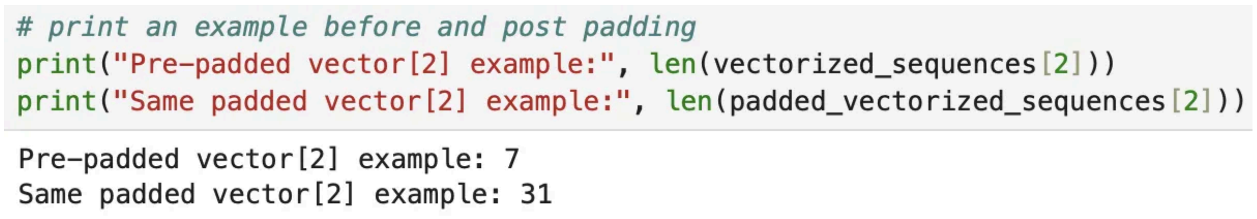
Figure 6: The first few values of the last word embedding from the second sequence in vectorized_sequences.



```
print(padded_vectorized_sequences[2][-1])
```

```
[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. 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. 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. 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. 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. 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. 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.]
```

Figure 7: All vector values of the last word embedding in the second sequence of padded_vectorized_sequences, which corresponds to the second sequence in vectorized_sequences, confirming the successful implementation of the padding process.



```
# print an example before and post padding
print("Pre-padded vector[2] example:", len(vectorized_sequences[2]))
print("Same padded vector[2] example:", len(padded_vectorized_sequences[2]))
```

```
Pre-padded vector[2] example: 7
Same padded vector[2] example: 31
```

Figure 8: The length of the example sequence before and after padding, further validating the padding process as the sequence length increased from 7 to 31 post-padding.

B4: CATEGORIES OF SENTIMENT

There are two sentiment categories in this analysis: 1 for positive and 0 for negative sentiment. Since there are only two possible outcomes, the sigmoid activation function will be used in the output layer of the networks. This function assigns a probability between 0 and 1 to each entry, making it ideal for binary classification and suitable for this task [10].

B5: STEPS TO PREPARE THE DATA

The steps to prepare the data for analysis included identifying and removing unusual characters, tokenizing sentences, manually padding out-of-vocabulary words, padding sequences below the maximum embedding threshold, and truncating sequences above the threshold. Once the data was transformed, it was split into a 70% training, 15% test, and 15% validation sets, reflecting industry standards [11]. With these proportions, there was sufficient data to train the models and evaluate their performance.

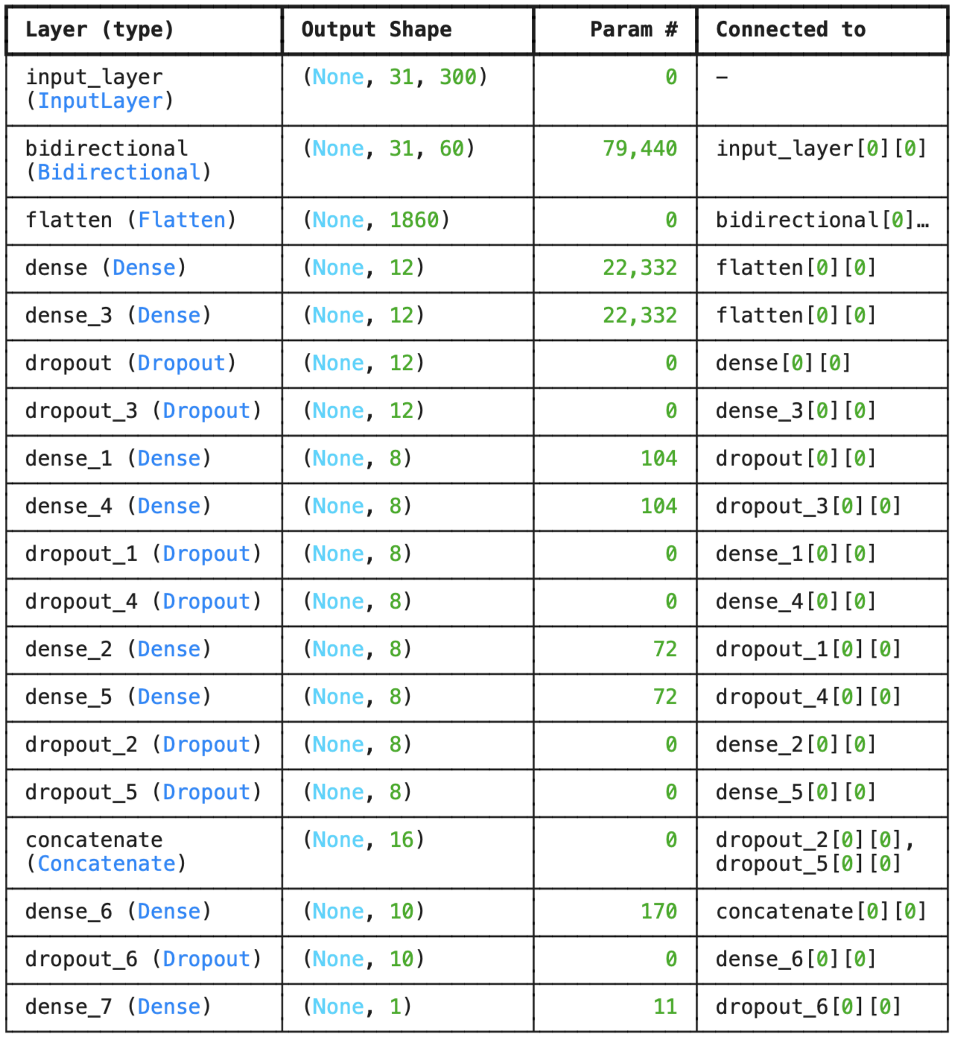
Since all three sources within the UCI Sentiment Labeled Sentences dataset were used for the analysis, the sets were stratified using both sentiment labels and domains to ensure an equal distribution of outcomes across all sets. However, the domain sizes were not equal because Amazon and Yelp each contained 1,000 reviews, while IMDb had only 748. To maintain uniformity, the Amazon and Yelp reviews were randomly sampled down to match the size of the IMDb domain. This approach prevented model bias due to any possible class imbalance and ensured that if sentiment expression varied across domains, the model would still generalize well.

B6: PREPARED DATA SET

A prepared copy of the dataset will be submitted separately as an NPZ file.

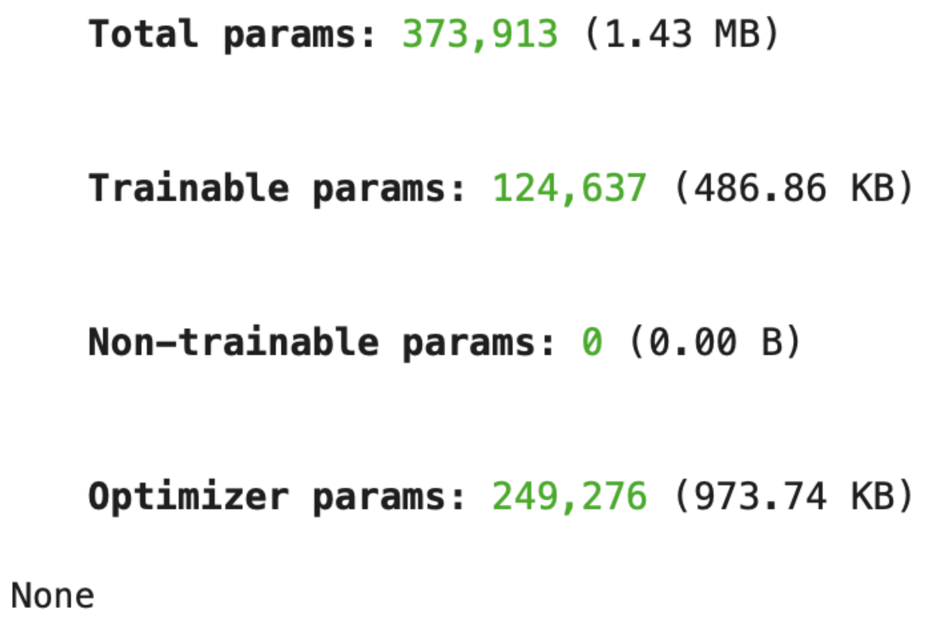
C1: MODEL SUMMARY

The functional and sequential models' summary outputs are shown in *Figures 9 - 10*.



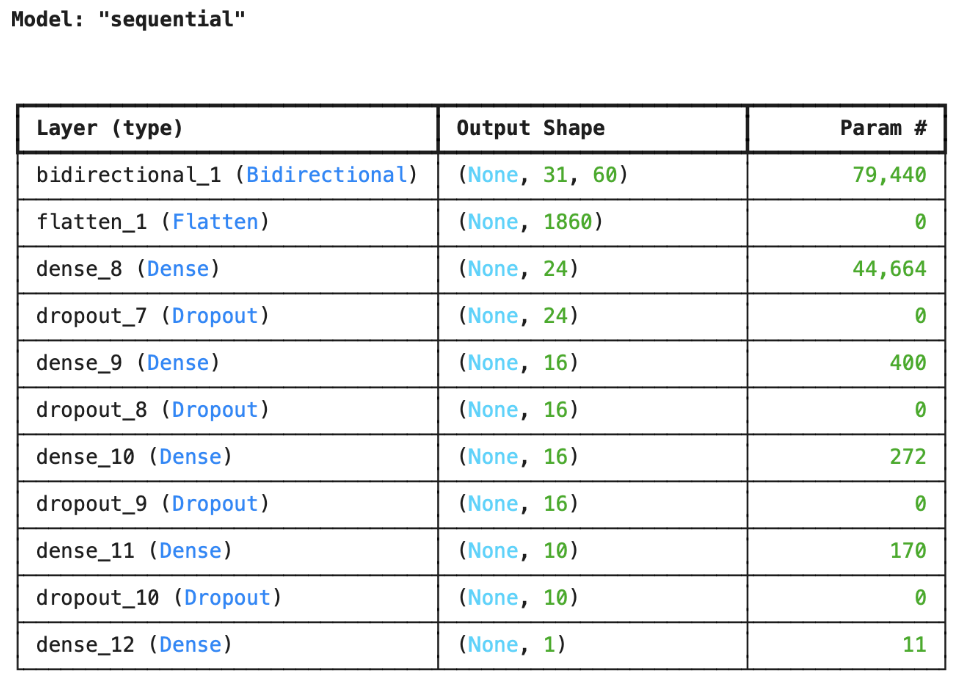
Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 31, 300)	0	-
bidirectional (Bidirectional)	(None, 31, 60)	79,440	input_layer[0][0]
flatten (Flatten)	(None, 1860)	0	bidirectional[0]…
dense (Dense)	(None, 12)	22,332	flatten[0][0]
dense_3 (Dense)	(None, 12)	22,332	flatten[0][0]
dropout (Dropout)	(None, 12)	0	dense[0][0]
dropout_3 (Dropout)	(None, 12)	0	dense_3[0][0]
dense_1 (Dense)	(None, 8)	104	dropout[0][0]
dense_4 (Dense)	(None, 8)	104	dropout_3[0][0]
dropout_1 (Dropout)	(None, 8)	0	dense_1[0][0]
dropout_4 (Dropout)	(None, 8)	0	dense_4[0][0]
dense_2 (Dense)	(None, 8)	72	dropout_1[0][0]
dense_5 (Dense)	(None, 8)	72	dropout_4[0][0]
dropout_2 (Dropout)	(None, 8)	0	dense_2[0][0]
dropout_5 (Dropout)	(None, 8)	0	dense_5[0][0]
concatenate (Concatenate)	(None, 16)	0	dropout_2[0][0], dropout_5[0][0]
dense_6 (Dense)	(None, 10)	170	concatenate[0][0]
dropout_6 (Dropout)	(None, 10)	0	dense_6[0][0]
dense_7 (Dense)	(None, 1)	11	dropout_6[0][0]

Figure 9: The functional model's summary output, illustrating the model's architecture.



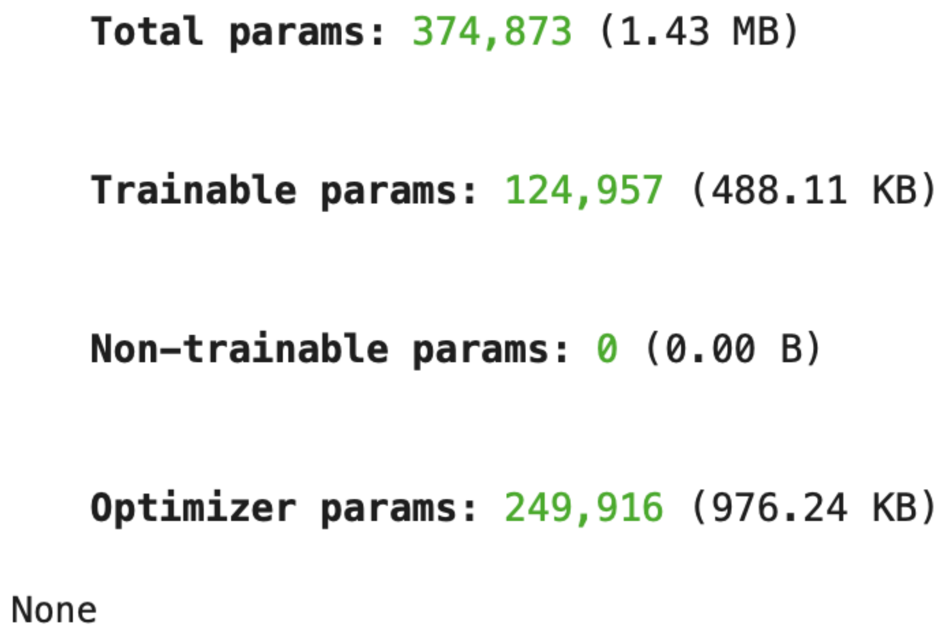
```
Total params: 373,913 (1.43 MB)

Trainable params: 124,637 (486.86 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 249,276 (973.74 KB)

None
```

Figure 10: The functional model's output summary, showing the total number of the model's parameters.



```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ bidirectional_1 (Bidirectional)  │ (None, 31, 60)         │        79,440 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_1 (Flatten)              │ (None, 1860)           │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_8 (Dense)                  │ (None, 24)             │        44,664 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_7 (Dropout)              │ (None, 24)             │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_9 (Dense)                  │ (None, 16)             │           400 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_8 (Dropout)              │ (None, 16)             │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_10 (Dense)                 │ (None, 16)             │           272 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_9 (Dropout)              │ (None, 16)             │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_11 (Dense)                 │ (None, 10)             │           170 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_10 (Dropout)             │ (None, 10)             │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_12 (Dense)                 │ (None, 1)              │            11 │
└──────────────────────────────────┴────────────────────────┴───────────────┘
```

Figure11: The sequential model's summary output, illustrating the model's architecture.



```
 Total params: 374,873 (1.43 MB)

 Trainable params: 124,957 (488.11 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 249,916 (976.24 KB)

None
```

Figure 12: The sequential model's output summary, showing the total number of the model's parameters.

C2: NETWORK ARCHITECTURE

The functional model contains 19 total layers. It begins with an input layer, which feeds into a BiLSTM layer that captures patterns in both forward and backward directions from the current word. The output then passes through a Flatten layer to prepare the data for the dense layers. At this point, the model branches into two distinct paths consisting of three Dense layers, each followed by its own Dropout layer. To merge the branches, a Concatenation layer combines their outputs before passing the data into a shared Dense layer that is also followed by its own Dropout layer. Finally, the model ends with an output layer with a single node for binary classification. There are 373,913 total parameters, including 124,657 trainable parameters, 0 non-trainable parameters, and 249,276 optimizer parameters. This architecture can be confirmed in the functional model's summary output, shown in *Figures 9 - 10*.

The comparable sequential model contains a total of 12 layers, as shown in *Figure 11*. While this model may appear simpler than the functional model due to having fewer layers, this difference is because the distinct branches in the functional model count as additional layers. To validate their comparable complexity, both models have approximately the same number of total parameters. The functional model has 373,913 parameters, while the sequential model has 374,873. In fact, the sequential model has slightly more parameters, likely due to its fully connected layers adding additional parameters. Of the total parameters, 124,957 are trainable, 0 are non-trainable, and 249,916 are optimizer parameters. The total number of the sequential model's parameters are shown in *Figure 12*.

The sequential model consists of an input layer, followed by a BiLSTM, then a Flatten layer. Afterward, it has four Dense layers, each followed by its own Dropout layer, before ending with an output layer with a single node for binary classification. This model follows the same overall structure as the functional model, except it lacks the distinct branches, making it suitable for comparing traditional sequential architectures with more flexible functional designs.

C3: HYPERPARAMETERS

The hyperparameters tuned for the model include activation functions, number of nodes per layer, loss function, optimizer, stopping criteria, and evaluation metrics. The model utilizes four activation functions: ReLU, inverse ReLU, Tanh, and Sigmoid. The ReLU function is a piecewise linear activation that helps the model capture nonlinear patterns by retaining positive activations while setting negative values to zero [7]. Conversely, the inverse ReLU function retains only negative activations by setting positive values to zero. These functions were applied to separate branches of the model, allowing it to specialize in distinct activation patterns and balance how it learns from the word embeddings.

While it is uncertain whether the positive or negative values in the embeddings directly correlate with sentiment, these activation functions help the model specialize in distinct semantic patterns, ensuring that potentially meaningful information is not ignored. Both ReLU functions were applied twice in their respective branches. While the first application zeros out opposing values, the randomly initialized weights after this step may still cause activations that can transform the remaining values and potentially create opposing values again. Therefore, applying the ReLU functions a second time helps to reinforce the intended activation patterns by removing these newly created opposing values, ensuring that the branches remain strictly specialized and distinct.

The Tanh activation function outputs values in the range -1 to 1, allowing it to handle both positive and negative activations equally [12]. Thus, it was used in both branches to capture additional nonlinear relationships that ReLU alone may have overlooked. Additionally, it was applied in the shared layer after merging the branches to ensure the activations from both branches were processed equally. Finally, the Sigmoid activation function was used in the output layer because it generates values between 0 and 1, making it ideal for binary classification, where the outputs can be interpreted as probabilities [12].

The number of nodes per layer was determined by following a partial pyramid architecture design, where the network starts off with a wide layer and gradually decreases in deeper layers [13]. This design allows the model to learn many low-level patterns in the early phases while reducing overfitting in later layers by compacting the learning process. However, not all the layers contained trainable nodes because some served functional purposes. These layers include the Flatten, Dropout, and Concatenate layers, while the BiLSTM and Dense layers contained trainable nodes.

To determine the optimal range of nodes per layer, the model was initially tested with a small number of nodes per layer around 1 to 3 and gradually increased until the performance plateaued. The validation accuracy peaked at around 70% despite the increase in model complexity, suggesting more training data is needed to improve the performance beyond this point. Thus, the smallest node range that achieved peak performance was preferred because it reduced complexity and overfitting. When using 50 or more nodes per layer, the model suffered severe overfitting, with the training accuracy nearing 95% while the validation accuracy remained around 65%. This increase in complexity required strong regularization to achieve performance that is comparable to a simpler model. Conversely, using fewer nodes prevented the model from converging, resulting in a training and validation accuracy around 50%, no better than a random guessing model.

The model begins with a wide BiLTSM layer with 60 total nodes because it consists of two LSTM with 30 nodes each. Starting with a wide BiLSTM layer helps the model capture contextual information from surrounding word embeddings and could improve its ability to distinguish between sentiments [6]. The number of nodes reduces to 24 in the first hidden layers of both branches. This reduction in nodes continues in the branches' following hidden layers, with the final two layers containing 16 total nodes.

At this point, the node reduction stops, which is why this becomes only a partial pyramid design rather than a strict pyramid structure. Initially, the number of nodes continued decreasing throughout the model, but the performance improved when additional nodes were included in the later layers. After merging the branches, the shared layer contains 10 nodes, followed by an output layer with a single node that is sufficient for binary classification.

The comparable sequential model had the same number of total nodes per layer as the functional model because the nodes in the distinct branches were summed for the corresponding conjoined layer in the sequential model. This approach ensured both models had the same shape, width and length, and number of total nodes, allowing for a fair comparison between the two architectures.

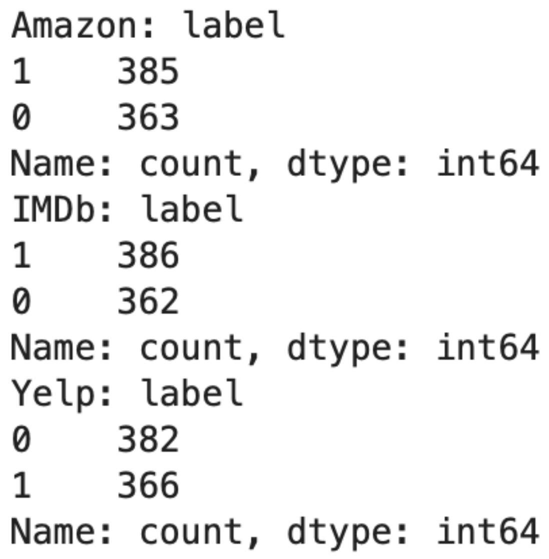
The models were compiled using the binary cross-entropy loss function, which is the standard for binary classification tasks. In this analysis, there were two possible outcomes between positive and negative sentiment, making it a binary classification problem. The binary cross-entropy function was the most appropriate choice because it quantifies how well the model's predictions align with the true labels while considering only two possible states, making it specifically designed for binary tasks [14].

The Adam optimizer, using default properties, achieved the best performance because it incorporates momentum and adaptive learning rates [15]. The momentum keeps track of a weighted average of past gradients, where the recent gradients have more influence and past information is still considered. This allows the model to not get stuck in a local minima because the accumulated past gradients allow it to push through and maintain a consistent movement in the same direction.

The adaptive learning rates assign individual learning rates to weights based on the magnitude and frequency of their updates [15]. Weights with larger or more frequent updates have large gradients, so they receive smaller learning rates to stabilize their updates and prevent overshooting the optimal solution. Conversely, weights with smaller or less frequent updates have small gradients, so they receive larger learning rates to prevent them from stagnating and ensure they continue learning. Various initial learning rates were tested, but the default value of 0.001 performed the best. Lower values were too conservative, causing the model to get stuck in local minima and fail to converge, while higher values were too unstable, also preventing the model from converging.

The stopping criteria used 5 epochs while monitoring the loss. This approach reduces wasteful computations while still providing some flexibility to escape local minima or training plateaus. Although the accuracy was considered for monitoring, the loss was deemed more reliable. A model with high accuracy can still have a high loss if its prediction probabilities are borderline correct, meaning they are close to 0.5 but slightly above or below. This suggests the model lacks confidence in its predictions and barely distinguishes between the outcomes.

This issue is avoided when prioritizing the loss because high-confidence predictions typically correlate with a good accuracy. Instead, a high loss and lower accuracy can occur in an imbalanced dataset, where the model confidently predicts the majority class while misclassifying the minority class. However, the UCI Sentiment Labeled dataset has an equal distribution of labels, as shown in *Figure 13*. Therefore, the loss was prioritized for the stopping criteria to balance confidence and accuracy.



```
Amazon: label
1    385
0    363
Name: count, dtype: int64
IMDb: label
1    386
0    362
Name: count, dtype: int64
Yelp: label
0    382
1    366
Name: count, dtype: int64
```

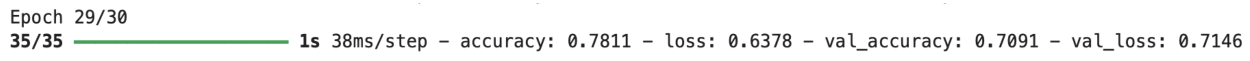
Figure 13: The distribution of the labels for each source in the UCI Sentiment Labeled dataset.

The loss and the accuracy will be used to evaluate the model's performance to balance confidence and correctness. The accuracy will be the primary performance metrics since the dataset is balanced with an equal distribution of the outcome labels, providing a measure for how correctly the model classifies the sentiments. Then the binary cross-entropy loss can be used to interpret how confident the model is in its predictions, since a lower loss indicates more confidence and is inherently correlated with correct predictions. Together, these performance metrics provide a balanced evaluation of the models.

The thoroughly selected hyperparameters resulted in a model that was able to consistently converge to the best performance given the limited data and model complexity. The activation functions in the functional model allowed the branches to specialize non-overlapping information. The partial pyramid architecture for determining the number of nodes increased the model's capacity for learning simpler patterns early on and reduced the risk of overfitting by compacting the learning process in deeper layers. The binary cross-entropy loss function was specifically designed for binary classification tasks, appropriate for this sentiment analysis. The Adam optimizer with the default properties, momentum, and adaptive learning rates helped the model consistently converge to the best performance. The defined epochs in the stopping criteria were flexible while monitoring the loss to balance confidence and correctness, with both the accuracy and loss evaluation metrics further validating the model's performance.

D1: STOPPING CRITERIA

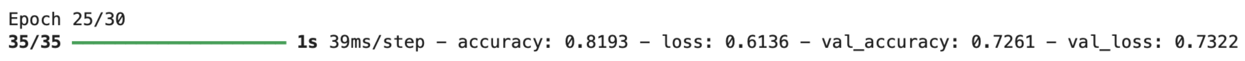
For the Functional model, the training almost completed the full 30-epoch threshold, ending at 29 epochs. The stopping criteria set 5 epochs for the model to improve its performance, and this was evident in the accuracy and loss fluctuating around the final values for the last 5 epochs. Stopping early only saved 1 epoch worth of computational waste and is a sign the model has reached its optimal performance with the current hyperparameters or training data. The final epoch is shown in *Figure 14*.



```
Epoch 29/30
35/35 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.7811 - loss: 0.6378 - val_accuracy: 0.7091 - val_loss: 0.7146
```

Figure 14: The final training epoch of the functional model.

For the sequential model, the training was completed faster, ending at only 23 epochs. The computational waste was more significant for this model. Notably, both models could have reduced computational waste by using a lower patience in the stopping criteria, but 5 epochs is not large for the scope of this analysis and allows for extra flexibility.



```
Epoch 25/30
35/35 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - accuracy: 0.8193 - loss: 0.6136 - val_accuracy: 0.7261 - val_loss: 0.7322
```

Figure 15: The final training epoch of the sequential model.

D2: FITNESS

The dual-branch functional model was a better fit than the traditional sequential model because it consistently converged with high accuracy and lower loss. However, both models showed signs of overfitting due to the significantly higher training accuracy and lower loss compared to the test and validation sets. The performance gap between these sets was smaller for the functional model but much wider for the sequential model. The overfitting prevention methods were identical for both models, utilizing LSTM dropout, Dropout layers, and L2 regularization.

The BiLSTM layer did not include a Dropout layer or L2 regularization because it already utilized built-in input dropout and recurrent dropout, both set to 40% [16]. The input dropout temporarily ignores input information for the LSTM layer for the current iteration. This technique is typically not recommended when the LSTM is the first layer because the ignored input would be absent from the entire model for that iteration. However, applying it significantly improved the performance by reducing the training accuracy from 95% to 87% while slightly increasing validation accuracy by 2% in both models. The recurrent dropout randomly drops internal nodes for sequential word embeddings to prevent the model from memorizing specific patterns. This further shortened the gap between the training and validation sets while slightly improving the validation accuracy.

Dropout layers were added to the remaining layers, with a moderate dropout rate of 30% to 40%. These layers reduce overfitting by temporarily dropping nodes during each training iteration to ensure the model does not overly rely on specific information for its predictions [17]. Additionally, L2 regularization was applied to the remaining layers because it mitigates overfitting by penalizing larger weights [18]. This discourages the model from overfitting to dominant patterns and improves its ability to generalize to unseen data. A moderate L2 value of 0.05 was chosen because lower values did not significantly reduce overfitting, while higher values hindered the models' ability to learn the structure of the data, resulting in poor performance. Despite moderate overfitting prevention efforts, the models still showed clear signs of overfitting, indicating that additional methods may be required.

D3: TRAINING PROCESS

The training visualizations of the loss and accuracy for both models are shown in *Figures 16 - 17*.

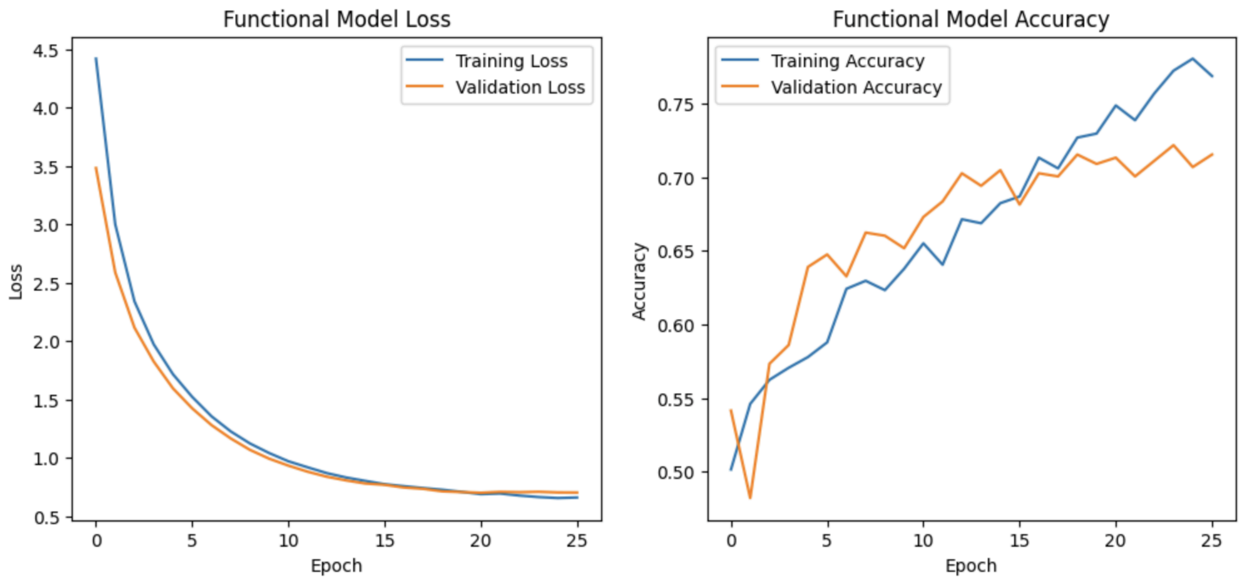


Figure 16: The visualizations of the functional model's training process, showing loss and accuracy.

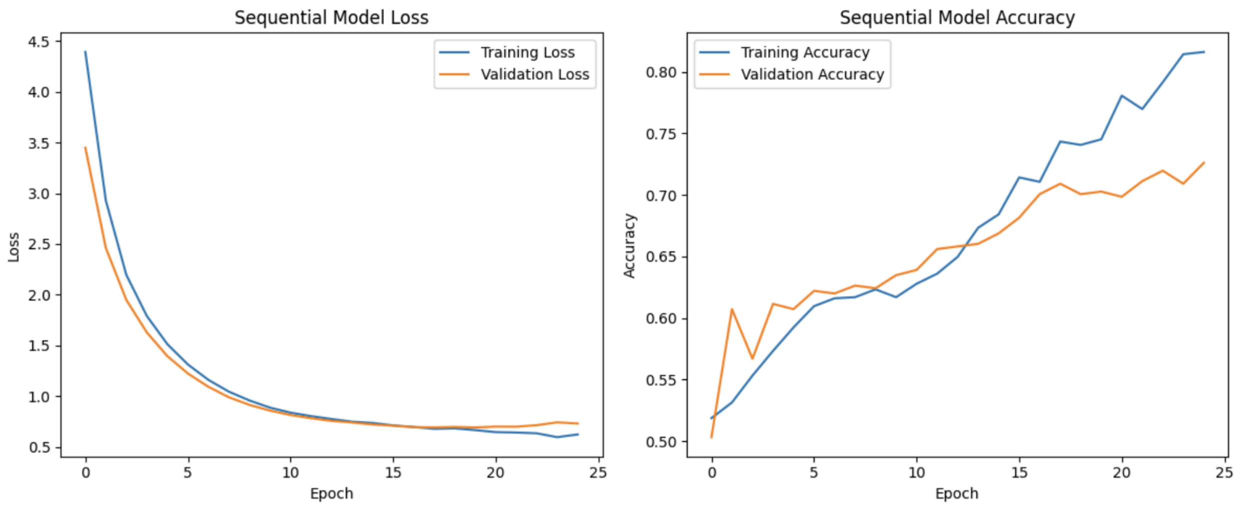


Figure 17: The visualizations of the sequential model's training process, showing loss and accuracy.

D4: PREDICTIVE ACCURACY

The accuracy of the optimal functional model showed fluctuations throughout the epochs while maintaining an overall upward trend. However, the validation accuracy started to plateau after 15 epochs, while the training accuracy continued increasing. This difference in growth led to a moderate gap between the training and validation accuracy in the final epoch. Therefore, early stopping should be reduced from 5 to 2 epochs to minimize overfitting because each additional epoch without improvement causes the model to overfit to the training data.

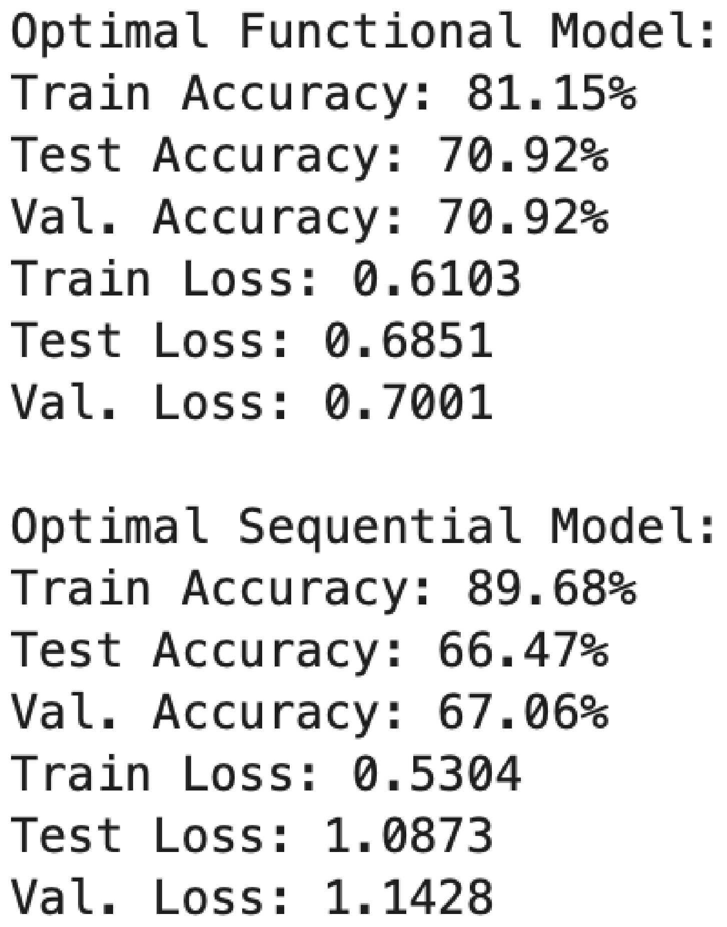
The final accuracy for the sets was as follows: 81.15% train, 70.92% test, and 70.92% validation. The accuracy was the exact same for the test and validation sets, meaning the model is able to perform consistently well to unseen data. The analysis resulted in a moderate accuracy for this binary classification that demonstrates the model was able to learn some of the structure of the data and is significantly better than a model that randomly guesses the output. However, a higher accuracy could be achieved with a more complex model with additional training data. But for the scope of this analysis, 70% accuracy is reasonable given the limited dataset size and model complexity.

The validation accuracy of the sequential model showed more stable fluctuations throughout the training and also gradually plateaued after 15 epochs. However, the training accuracy continued to increase, creating a large gap in the final epoch. The consistent accuracy trends of both models suggest that reducing the training from 30 to 15 epochs could significantly reduce overfitting without sacrificing validation performance because both models reached similar accuracy at that point, when validation accuracy plateaued and showed minimal gains beyond this epoch.

Nevertheless, the optimal sequential model performed substantially worse than the functional model, and the final accuracy for the sets were as follows: 89.68% train, 66.47% test, and 67.06% validation. The gap between the sets was much larger than in the functional model, with the training accuracy being higher and the validation accuracy being lower. These traits show that the sequential model was overfitting the training data despite using the same regularization strategies, and more preventative measures are required.

The training and validation loss for both models gradually decreased at the same rate and exhibited gaps in the final epoch, with the sequential model showing a larger gap. The final loss for the functional model is as follows: 0.6103 train, 0.6851 test, and 0.7001 validation. There's a small gap between the loss values, showing the model was more confident in its prediction for the training set. However, this gap is not concerningly high. The loss for the sequential model is as follows: 0.5304 train, 1.0873 test, and 1.1428 validation. The gap between the loss for these sets is quite large with the training loss being almost half of the test and validation loss, meaning the model was not confident in its predictions for the test and validation sets. This further confirms that the sequential model was severely overfitting the training data, despite all the preventative measures.

Overall, the functional model outperformed the sequential model by better generalizing to unseen data and balancing accuracy and loss. However, further improvements could be made by reducing early stopping patience and lowering the total number of training epochs to mitigate overfitting. For a clearer comparison, the final evaluation metrics of the optimal models are shown in *Figure 18*.

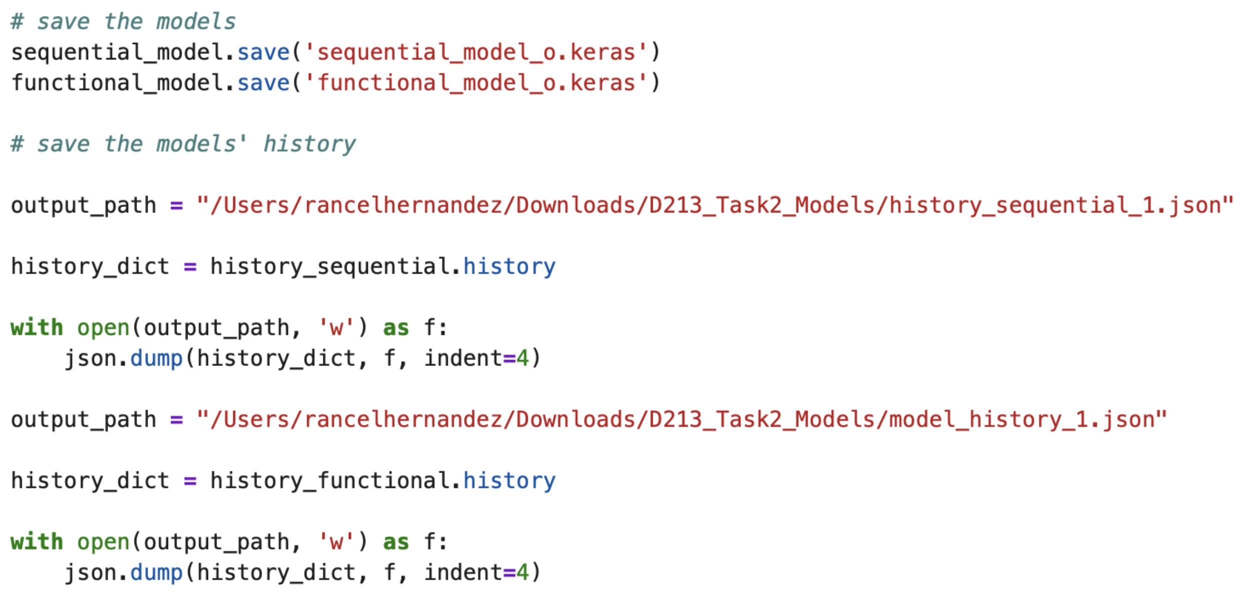


```
Optimal Functional Model:
Train Accuracy: 81.15%
Test Accuracy: 70.92%
Val. Accuracy: 70.92%
Train Loss: 0.6103
Test Loss: 0.6851
Val. Loss: 0.7001

Optimal Sequential Model:
Train Accuracy: 89.68%
Test Accuracy: 66.47%
Val. Accuracy: 67.06%
Train Loss: 0.5304
Test Loss: 1.0873
Val. Loss: 1.1428
```

Figure 18: The final evaluation metrics of the optimal models.

E: CODE

The code to save the trained models is shown in *Figure 19*. The models were saved in the .keras file format to align with TensorFlow 2's saving methods, and their training history was stored in JSON files for efficiency.



```
# save the models
sequential_model.save('sequential_model_o.keras')
functional_model.save('functional_model_o.keras')

# save the models' history

output_path = "/Users/rancelhernandez/Downloads/D213_Task2_Models/history_sequential_1.json"

history_dict = history_sequential.history

with open(output_path, 'w') as f:
    json.dump(history_dict, f, indent=4)

output_path = "/Users/rancelhernandez/Downloads/D213_Task2_Models/model_history_1.json"

history_dict = history_functional.history

with open(output_path, 'w') as f:
    json.dump(history_dict, f, indent=4)
```

Figure 19: The code to save the trained models.

F: FUNCTIONALITY

The dual-branch functional model followed a partial pyramid architecture and utilized various activation functions to capture nonlinear relationships in the data. Additionally, both models included a BiLSTM layer to provide a temporal component that captures contextual information from surrounding embeddings and were compiled with the most efficient optimizer.

The distinct branches in the functional model may have helped to improve performance because each branch specialized in independent information within the embeddings by utilizing opposing activation functions. One branch utilized only the standard ReLU activation function that ignores negative inputs, while the other utilized the inverse ReLU to focus specifically on negative inputs. Without these distinct activation functions, both branches would likely capture similar information. Therefore, using opposing activation functions helped the model to learn a wider range of patterns and separating the branches isolated the learning process, mimicking an ensemble model that learns diverse information and integrates it into a shared layer for the predictions.

The sequential model was designed to be of a comparable complexity to the functional model, but without the distinct branches. This approach allowed for a comparison of a traditional fully connected sequential model against a flexible and complex functional model to gain insights into how model architecture can influence performance. The sequential model only utilized the standard ReLU function, so it was learning from a more restricted information space. However, both models utilized the Tanh functions along with their corresponding ReLU to capture additional nonlinear patterns.

Additionally, both models were compiled with the Adam optimizer because it stabilizes the training, allowing the models to converge more consistently [15]. This optimizer included momentum to help the model avoid getting stuck in local minima and adaptive learning rates to assign each weight its own learning rate so larger weights do not overshoot the optimal solution and smaller weights do not plateau during the learning process. The best learning rate was a value of 0.001 because higher values were too unstable and lower values trapped the model in local minima, both leading to poor performance.

The accuracy and loss of the functional model consistently outperformed the sequential model, with the optimal functional model achieving an accuracy of 70.92% and loss of 0.7001 and the sequential model achieving an accuracy of 67.06% and loss of 1.1428 on the validation sets. The functional model's predictions were more correct and statistically more confident, so it is likely the increased complexity was valuable for the sentiment classification. Moreover, the functional model converged at a higher rate of around 85% of the time while the sequential model converged closer to 30%. Despite having a similar complexity, the functional model was considerably more reliable and showed optimization stability.

Both models followed a partial pyramid design starting with a wide BiLSTM layer that fed into decreasingly complex dense layers. The idea is that many patterns are captured in the early phases and overfitting is reduced in the later compact layers. The final number of nodes was tested to maintain a moderate range that balanced complexity. Additional nodes increased overfitting while fewer nodes led to poor performance. Nevertheless, overfitting was still prominent with the training accuracy far ahead of the validation accuracy, despite the standard level of regularization. The training visualization revealed that the overfitting was likely due to overtraining the model rather than architectural issues because the gap in performance occurred after 15 epochs when the validation accuracy stopped improving, yet the model continued modeling the training data. Therefore, the early stopping patience was set too high since the additional epochs increased the gap in performance, so the overall training duration should be reduced from 30 to 15 epochs and the early stopping patience should be lowered from 5 to 2 epochs to mitigate overfitting and reduce the gap in performance.

G: RECOMMENDATIONS

Based on the results of the sentiment analysis, there are various recommendations related to overfitting, model architecture, and additional testing. Overfitting the training data was a consistent issue, but the visualization of both models' accuracy revealed that reducing the training duration from 30 to 15 epochs and decreasing early stopping criteria patience from 5 to 2 epochs would narrow the gap in performance between the training and validation sets, effectively reducing overfitting.

The functional model's complex and flexible architecture resulted in more reliable and confident predictions, so this architecture should be explored more. As of now, it is unknown if both branches learned useful or redundant patterns related to sentiment prediction, so additional testing should be done to explore these trends further. One suggestion could be to add output nodes at the end of the branches to see if the standard ReLU had a higher true positive rate and the inverse ReLU had a higher true negative rate, providing evidence that the negated activation functions captured non-overlapping information correlated with sentiment.

The functional model was also able to converge more frequently than the traditional model, indicating a more stable training process. Therefore, the complexity of the model could further be increased by adding more nodes, hidden layers, and additional activation functions. Potentially, even a third branch could be added that considers both inverse functions or other alternative patterns.

All the sources in the UCI Sentiment Labeled dataset were combined to train a model that can generalize well. However, users could express sentiment differently across these domains, so further testing should be done to compare how accurate the model performs between domains. For example, if the model achieves a higher accuracy on reviews from Amazon and lower accuracy on reviews from IMDb, the difference in performance would be obscured in the overall accuracy. This test could evaluate if the model is generalizing well and provide insights into how users express sentiment across domains.

Despite combining all the datasets for additional training data, the models seemed to reach a peak performance around 70% regardless of the increase in complexity, so it is likely more training data is required to make further improvements. With these recommendations and insights from the sentiment analysis, more complex and reliable models could be developed to achieve accurate and confident predictions in review classification, enhancing how organizations understand user sentiment.

I: SOURCES FOR THIRD-PARTY CODE

Python Software Foundation, “re - Regular Expression Operations,” *Python Documentation*, 2025. [Online]. Available: https://docs.python.org/3/library/re.html. [Accessed: Feb. 20, 2025].

spaCy, “Doc Class,” *spaCy*, 2025. [Online]. Available: https://spacy.io/api/doc. [Accessed: Feb. 20, 2025].

NumPy, “numpy.reshape,” *NumPy Documentation*, 2025. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.reshape.html. [Accessed: Feb. 21, 2025].

TensorFlow, “tf.keras.layers.ReLU,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU. [Accessed: Feb. 21, 2025].

TensorFlow, “tf.keras.Input,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/Input. [Accessed: Feb. 21, 2025].

TensorFlow, “tf.keras.layers.Bidirectional,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional. [Accessed: Feb. 22, 2025].

TensorFlow, “tf.keras.layers.Flatten,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten. [Accessed: Feb. 22, 2025].

Keras, “Concatenate Layer,” *Keras*, 2025. [Online]. Available: https://keras.io/api/layers/merging_layers/concatenate/. [Accessed: Feb. 22, 2025].

TensorFlow, “tf.keras.regularizers.L2,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L2. [Accessed: Feb. 22, 2025].

TensorFlow, “tf.keras.callbacks.EarlyStopping,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping. [Accessed: Feb. 22, 2025].

Keras, “Model Training APIs,” *Keras*, 2025. [Online]. Available: https://keras.io/api/models/model_training_apis/. [Accessed: Feb. 23, 2025].

Keras, “Save Method,” *Keras*, 2025. [Online]. Available: https://keras.io/api/models/model_saving_apis/model_saving_and_loading/#save-method. [Accessed: Feb. 23, 2025].

Python Software Foundation, “json - JSON Encoder and Decoder,” *Python Documentation*, 2025. [Online]. Available: https://docs.python.org/3/library/json.html. [Accessed: Feb. 24, 2025].

GeeksforGeeks, “Read JSON File Using Python,” *GeeksforGeeks*, 2025. [Online]. Available: https://www.geeksforgeeks.org/read-json-file-using-python/. [Accessed: Feb. 24, 2025].

J: SOURCES

Reference List

[1] TensorFlow, “Keras Functional API,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/guide/keras/functional_api. [Accessed: Feb. 25, 2025].

[2] IBM, "Recurrent Neural Networks," IBM Think Blog, 2025. [Online]. Available: https://www.ibm.com/think/topics/recurrent-neural-networks. [Accessed: Mar. 7, 2025].

[3] University of Amsterdam, "Feedforward Neural Networks 1: What is a Feedforward Neural Network?," UvA Faculty of Humanities, 2025. [Online]. Available: https://www.fon.hum.uva.nl/praat/manual/Feedforward_neural_networks_1__What_is_a_feedforward_ne.html. [Accessed: Mar. 7, 2025].

[4] spaCy, “spaCy Models: English,” *spaCy*, 2025. [Online]. Available: https://spacy.io/models/en. [Accessed: Feb. 25, 2025].

[5] GeeksforGeeks, “Word Embeddings in NLP,” *GeeksforGeeks*, 2025. [Online]. Available: https://www.geeksforgeeks.org/word-embeddings-in-nlp/. [Accessed: Feb. 25, 2025].

[6] Anishnama, “Understanding Bidirectional LSTM for Sequential Data Processing,” *Medium*, 2025. [Online]. Available: https://medium.com/@anishnama20/understanding-bidirectional-lstm-for-sequential-data-processing-b83d6283befc. [Accessed: Feb. 26, 2025].

[7] J. C. Luna, “Rectified Linear Unit (ReLU),” *DataCamp*, 2025. [Online]. Available: https://www.datacamp.com/blog/rectified-linear-unit-relu. [Accessed: Feb. 26, 2025].

[8] Z. Yin and Y. Shen, "On the Dimensionality of Word Embedding," Advances in Neural Information Processing Systems (NeurIPS), 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/b534ba68236ba543ae44b22bd110a1d6-Paper.pdf. [Accessed: Mar. 7, 2025].

[9] TensorFlow, "tf.keras.utils.pad_sequences," *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/utils/pad_sequences. [Accessed: Feb. 26, 2025].

[10] Y. Chen, L. Li, W. Li, Q. Guo, Z. Du, and Z. Xu, , "Fundamentals of neural networks," in *AI Computing Systems: An Application Driven Perspective*. Morgan Kaufmann, 2024. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/sigmoid-function [Accessed: Feb. 27, 2025].

[11] M. Sadeghi, "How to Split Your Data for Machine Learning," *Medium*, 2025. [Online]. Available: https://medium.com/@masadeghi6/how-to-split-your-data-for-machine-learning-eae893a8799c. [Accessed: Feb. 28, 2025].

[12] Lakshmi, "Activation Functions and Their Derivatives: A Quick Complete Guide," *Analytics Vidhya*, 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/#h-tanh-activation-function. [Accessed: Feb. 28, 2025].

[13] Machine Learning Theory, "Designing the Architecture," *Machine Learning Theory*, 2025. [Online]. Available: https://machinelearningtheory.org/docs/Deep-Learning/designing-the-architecture/. [Accessed: Feb. 28, 2025].

[14] GeeksforGeeks, “Binary Cross-Entropy / Log Loss for Binary Classification,” *GeeksforGeeks*, 2025. [Online]. Available: https://www.geeksforgeeks.org/binary-cross-entropy-log-loss-for-binary-classification/. [Accessed: Mar. 1, 2025].

[15] R. Agarwal, “Adam Optimization,” *Built In*, Sep. 13, 2023. [Online]. Available: https://builtin.com/machine-learning/adam-optimization. [Accessed: Mar. 2, 2025].

[16] TensorFlow, “tf.keras.layers.LSTM,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM. [Accessed: Mar. 2, 2025].

[17] TensorFlow, “tf.keras.layers.Dropout,” *TensorFlow*, 2025. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout. [Accessed: Mar. 3, 2025].

[18] TensorFlow for R, “Overfitting and Underfitting,” *TensorFlow for R*, 2025. [Online]. Available: https://tensorflow.rstudio.com/tutorials/keras/overfit_and_underfit. [Accessed: Mar. 4, 2025].