NANYANG TECHNOLOGICAL UNIVERSITY

COLLEGE OF COMPUTING AND DATA SCIENCE

SC4002 Natural Language Processing

G39 Group Assignment

November 10, 2024

No.	Name	Matriculation Number	Contribution
1	**Isaiah Loh Kai En**	U2140496L	Part 1
2	**Li Zihan**	U2121598G	Part 2
3	**Lye En Lih**	U2121387B	Part 2
4	**Ong Yi Xin Kelly**	U2040271D	Part 1
5	**Wang Shang An Davis**	U2121998F	Part 3
6	**Zhang Jing Wen**	U2121853G	Part 3

1 Preparing Word Embeddings

(a) Size of the vocabulary

The simple_preprocess function from the Gensim library was used to convert words to lowercase, and de-accent and tokenize the text. We then iterated through the tokens and obtained the unique words to build our vocabulary. This gave us a **vocabulary size of 16,256**.

(b) Number of OOV words in the training data

There are many pre-trained models available in the Gensim library for Python. We will analyze the OOVs for `word2vec-google-news-300`, `glove-twitter` and `glove-wiki-gigaword`, as the Rotten Tomatoes dataset likely contains more informal and social media language, making these models' vocabularies more relevant.

Lower-dimensional embeddings (like 25 or 50) are faster to compute and require less memory, but they might capture less nuance in word meanings. Higher-dimensional embeddings (like 100 or 200) capture more detailed relationships between words, which can improve model performance for NLP tasks but require more memory and computational power.

Our benchmark for dimensionality will start at **100**, with comparisons to higher-dimensional models as well.

Model	Vocabulary Size	Number of OOV Words
`word2vec-google-news-300`	3,000,000	1,454
`glove-twitter-100`	1,193,514	1,477
`glove-twitter-200`	1,193,514	1,477
`glove-wiki-gigaword-100`	400,000	546
`glove-wiki-gigaword-200`	400,000	546
`glove-wiki-gigaword-300`	400,000	546

Table 1: Vocabulary size and number of OOV Words for various GloVe models.

We can deduce that the number of OOV words is independent of the model's dimensionality. This is because GloVe embeddings are pre-trained word vectors with fixed vocabularies based on the corpus they were trained on.

Therefore, we will proceed with `glove-wiki-gigaword-300`, as it has the fewest OOV words among the 3 models, and the high dimensionality is expected to provide richer semantic representations. While higher dimensional embeddings require more computational resources, the trade-off is justified by the potential improvement in accuracy.

(c) The best strategy to mitigate OOV words

We tested 2 strategies to deal with our OOV words.

First Strategy: Lemmatization with Fallback Embedding

Lemmatization reduces words to their base or dictionary form (e.g., running, ran, runs → run). For words that are still not found or are rare, we create an embedding matrix that assigns the average vector of the vocabulary to these words.

```python
# Download WordNet lemmatizer data
nltk.download('wordnet')
lemmatizer = WordNetLemmatizer()

# Preprocess a word with lemmatization
def preprocess_word(word):
    lemmatized_word = lemmatizer.lemmatize(word.lower())  # Convert to lowercase and lemmatize
    return lemmatized_word

# Load glove-wiki-gigaword-200 model
glove_model = KeyedVectors.load(os.path.join(model_dir_path, "glove-wiki-gigaword-200.model"))
embedding_dim = glove_model.vector_size

# Calculate the average vector for fallback
avg_vector = np.mean([glove_model[word] for word in glove_model.key_to_index], axis=0)

# Function to get GloVe embeddings with fallback handling
def get_glove_embedding(word):
    word = preprocess_word(word)
    if word in glove_model.key_to_index:
        return glove_model[word]
    else:
        return avg_vector  # Use average vector as a fallback for OOV words

# OOV Count without Lemmatization and Fallback Embedding
oov_count_before = sum(1 for word in train_vocab if word not in glove_model.key_to_index)
print("Number of OOV words (without Lemmatization and Fallback Embedding):", oov_count_before)

# OOV Count with Lemmatization and Fallback Embedding
oov_count_after = sum(1 for word in train_vocab if preprocess_word(word) not in glove_model.key_to_index)
print("Number of OOV words (with Lemmatization and Fallback Embedding):", oov_count_after)
```

Figure 1: Code snippet of Lemmatization with Fallback Embedding

State	Number of OOV Words
Without Lemmatization and Fallback Embedding	546
With Lemmatization and Fallback Embedding	538

Table 2: Change in OOV words after implementation

Second Strategy: FastText with Subword Embeddings

FastText's subword embeddings reduce OOV words by:

- Generating embeddings for any word through character n-grams, even if the full word does not appear in the training data
- Producing meaningful representations for morphologically rich words, misspellings, and unknown terms by using common subword patterns
- Eliminating the need for fallback strategies, unlike GloVe, which requires preprocessing or fallback vectors to handle OOV words

FastText has the following two models:

- `wiki.en.vec`
  - This model is trained exclusively on English Wikipedia data. Its vocabulary focuses on formal, encyclopedic language, covering terms from diverse domains such as history, science, arts, and popular culture.
  - Typically smaller than Common Crawl models since Wikipedia has a limited (though diverse) lexicon, centered around factual, standardized language.
- `cc.en.300.vec`
  - This model is trained on the Common Crawl dataset, a massive, multilingual dataset collected from a wide range of online sources. Its vocabulary is larger and more diverse, capturing a broader array of language, including slang, informal speech, niche terminology, and multilingual content.
  - It is significantly larger due to the vast range of sources, making it suitable for general NLP applications.

```python
# Load the Wiki FastText vectors from the .vec file
wiki_fasttext_model = KeyedVectors.load_word2vec_format(os.path.join(model_dir_path, "wiki.en.vec"), binary=False)

# Load the Common Crawl FastText vectors from the .vec file
crawl_fasttext_model = KeyedVectors.load_word2vec_format(os.path.join(model_dir_path, "cc.en.300.vec"), binary=False)

# Function to get FastText embeddings, with subword handling
def get_fasttext_embedding(model, word):
    return model[word]

# Initialize OOV counters
wiki_oov_count = 0
crawl_oov_count = 0

# Loop through each word in the vocabulary
for word in train_vocab:
    preprocessed_word = preprocess_word(word)

    # Check if the word is in the vocabulary of the Wikipedia model
    if preprocessed_word not in wiki_fasttext_model:
        wiki_oov_count += 1
    else:
        # Retrieve embedding if the word is in the vocabulary
        wiki_embedding = get_fasttext_embedding(wiki_fasttext_model, preprocessed_word)

    # Check if the word is in the vocabulary of the Common Crawl model
    if preprocessed_word not in crawl_fasttext_model:
        crawl_oov_count += 1
    else:
        # Retrieve embedding if the word is in the vocabulary
        crawl_embedding = get_fasttext_embedding(crawl_fasttext_model, preprocessed_word)

# Print the number of OOV words for each model
print("Number of OOV words (Wikipedia Model):", wiki_oov_count)
print("Number of OOV words (Common Crawl Model):", crawl_oov_count)
```

Figure 2: Code snippet of FastText with Subword Embeddings

Model	Number of OOV Words
Wikipedia Model	240
Common Crawl Model	734

Table 3: Number of OOV words for the different models

From the OOV word counts from the two models, we observe that the `wiki.en.vec` model performed better than expected. This is somewhat surprising, as the Rotten Tomatoes reviews are more conversational and might be thought to contain slang or informal language.

Some possible reasons for this outcome could be:

- **Relevant Vocabulary**: Wikipedia includes extensive formal descriptions and movie-related vocabulary that aligns with the language used in reviews.
- **Reduced Informal Noise**: `cc.en.300.vec` covers broader, more informal internet language, leading to mismatches with the structured, descriptive tone of movie reviews.
- **Coverage of Proper Nouns and Critic Terms**: Wikipedia's curated content better captures names, technical terms, and critic jargon, reducing OOV rates for datasets centered on entertainment media.

Therefore, we will use the `wiki.en.vec` embedding matrix to reduce the number of OOV words present.

2 Model Training and Evaluation - RNN

(a) Final configuration of our best model

To obtain the best model configuration for performance, we used **GridSearch** to try out and tune various combinations of hyperparameter values. For example, RMSProp, SGD, Adam and Adagrab were explored as possible optimisers, with Adam returning the best accuracy.

Model Parameters	Value
Hidden Dimension	128
Output Dimension	2
Learning Rate	0.001
Weight Decay	1e-4
Batch Size	32
Epochs	100
Pooling Strategy	Last Hidden State
Optimizer	Adam
Loss Function	Cross Entropy Loss

Table 4: Hyperparameters of best performing model

(b) Accuracy score of the test set and the validation set for each epoch during training

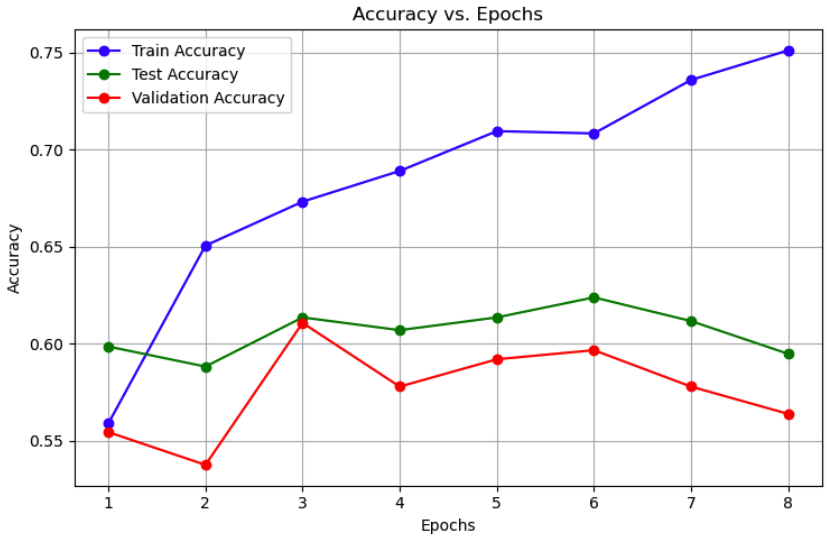

Figure 3: Accuracy scores during training

(c) Methods tried in deriving the final sentence representation to perform sentiment classification

To derive the final sentence representation for sentiment classification, we implemented and tested 4 different aggregation strategies for the hidden states produced by our RNN:

1. Last Hidden State

We used the hidden state from the last time step of the RNN as the representation of the entire sentence. The last hidden state is assumed to capture information from the entire sequence, summarizing the context up to the final word. After processing the input sequence through the RNN, we took the final hidden state and passed it directly to the classifier layer for sentiment prediction. This method achieved the highest accuracy among the strategies we tested. It suggests that the last hidden state effectively encapsulates the essential information needed for sentiment classification in our dataset.

2. Max Pooling

We applied a max pooling operation across all time steps for each feature dimension of the hidden states. Max pooling captures the most salient features present in any time step, highlighting the strongest signals in the sequence. For each feature in the hidden states, we selected the maximum value across all time steps to form a fixed-size representation. This method resulted in a lower accuracy compared to using the last hidden state. It indicates that relying solely on the most activated features might omit important contextual information necessary for accurate sentiment prediction.

3. Mean Pooling

We computed the average of the hidden states across all time steps to obtain the sentence representation. Mean pooling provides a smooth, overall representation by averaging

information from all words in the sequence. We calculated the mean value for each feature dimension across the time steps of the hidden states. This strategy yielded the lowest accuracy among the methods tested. Averaging may have diluted significant signals, making it less effective for capturing the nuances required for sentiment analysis.

4. Combined Mean and Max Pooling
We concatenated the results of both mean and max pooling operations to form the sentence representation. Combining mean and max pooling aims to leverage both the overall trends (mean) and the most significant features (max) in the sequence. We performed both mean and max pooling on the hidden states and concatenated the resulting vectors before passing them to the classifier. This method showed a slight improvement over individual pooling strategies but did not surpass the accuracy of the last hidden state method. It suggests that while combining pooling methods adds more information, it may also introduce redundancy or noise that affects the classifier's performance.

Method	Validation Accuracy	Test Accuracy
Last Hidden State	0.6107	**0.6135**
Maximum Pooling	0.5084	0.5047
Mean Pooling	0.5291	0.5300
Mean Max Pooling	0.5169	0.5169

Table 5: Validation and test accuracies for different RNN methods

Overall, using the **Last Hidden State** provided the best sentence representation for sentiment classification, achieving the highest accuracy on the test set. This suggests that for our specific task and dataset, preserving the sequential nature of the data through the last hidden state is more beneficial than aggregating hidden states using pooling methods. It is important to note that handling the OOVs in the pre-trained vectors in the gloVe dataset will improve the model significantly.

3 Enhancement

(a) Accuracy score on the test set when the word embeddings are updated

Method	Test Accuracy in Part 3(a)	Test Accuracy in Part 2
RNN Last State	**0.7167**	0.6135

Table 6: Test accuracies for the RNN Last State method

Comparing the RNN test accuracies above, we can see that unfreezing the word embeddings of the pre-trained GloVe vectors significantly increased the model's accuracy from **61.35%** to **71.67%**.

(b) Accuracy score of the test set when FastText is applied to deal with OOV words

We created the embedding matrix using the FastText pre-trained vectors. For the remaining 240 OOV words after applying FastText, we initialized their vectors with normalized values from the vocabulary. To keep the random embeddings close to zero and reduce variance in OOV word representation, we set the scale to 0.2.

Method	Test Accuracy in Part 3(b)	Test Accuracy in Part 3(a)	Test Accuracy in Part 2
RNN Last State	**0.7261**	0.7167	0.6135

Table 7: Test accuracies for the RNN Last State method

(c) Accuracy scores of biLSTM and biGRU on the test set

Method	2 Layers Test Accuracy	3 Layers Test Accuracy	4 Layers Test Accuracy
BiLSTM	0.7402	0.7411	0.7430
BiLSTM with Dropout	0.7523	0.7533	0.7570
BiGRU	0.7533	0.7345	0.7420

Table 8: Test Accuracy of Different Methods Across Layers

From the table above, the test accuracy of the BiLSTM (and with dropout) model improves as the number of layers increases. In contrast, the test accuracy of the BiGRU model does not show a consistent improvement with additional layers. In addition,

BiGRU and BiLSTM achieved these higher test accuracies compared to RNN because they capture bidirectional dependencies, which allow the model to process both past and future contexts in each sequence. Their bidirectional capability also allows the model to understand sequential patterns better, which is beneficial for complex language tasks.

(d) Accuracy scores of CNN on the test set

Model	Test accuracy
CNN	0.7833

Table 9: Test accuracy score for the CNN model

The CNN model achieved the highest test accuracy of **78.33%**, outperforming both the BiLSTM and BiGRU models. CNN also performed better than RNN as the convolutional layers can capture local dependencies and important features within fixed-length contexts, regardless of sequence order. This makes CNNs effective for language tasks that involve identifying key patterns across sequences.

(e) Final improvement strategy

Model	Test accuracy
Attention BiLSTM Model with Dropout	0.8105

Table 10: Test accuracy score for the final model

Explanation of Model Architecture:

1. **BERT Embeddings:** Provides contextualized word embeddings that adapt based on the word's meaning in the sentence. For example, the word "bank" in "river bank" vs. "money in the bank" will have different embeddings, capturing its context dynamically.
2. **BiLSTM Layer:** Captures bidirectional dependencies by processing the input sequence both forward and backwards, ensuring the model understands relationships between words in the entire sequence, not just in one direction.
3. **BERT + BiLSTM Combination:** The BERT embeddings provide rich contextual information, ensuring that word representations are sensitive to their context. The BiLSTM captures sequential patterns and bidirectional relationships in the text, complementing BERT's contextual embeddings for each sequence.
4. **Combined Attention Mechanism:** Applied to the merged output of forward and backward BiLSTM states. Attention addresses the challenges of:
   - **Long-Range Dependencies**: Traditional models often fail to connect distant words due to vanishing gradients or fixed receptive fields (e.g., in CNNs). Attention allows the model to selectively focus on relevant words, regardless of their distance in the sequence.

- **Context Understanding**: Attention assigns weights to tokens, enabling the model to emphasize critical words or phrases dynamically based on their relevance. *Example*: For sentiment analysis, the model might focus on words like "excellent" or "horrible" while ignoring filler words like "the" or "and." This improves interpretability by providing insight into which tokens contribute most to predictions.

5. **Dropout Regularization:** Dropout is applied after attention to reduce overfitting when learning from high-dimensional BERT embeddings, ensuring the model generalizes well to unseen data and reduces reliance on specific patterns in the data.
6. **Fully Connected Layer:** Processes the attention-weighted token representations and outputs the sentiment classification (e.g., positive or negative).

In summary, this architecture combines BERT embeddings, BiLSTM layers, and a combined attention mechanism to address the limitations of traditional models in capturing long-range dependencies, dynamically assigning focus to relevant tokens, and improving context understanding and interoperability. Additionally, the model is regularized with dropout, promoting robust generalization. Through these components, we achieved a test accuracy of **81.05%**, the highest among all the models we trained and analyzed.

(f) Comparison of results across different solutions

Feature	**RNN**	**CNN**	**BiLSTM/ BiGRU**	**Combined Attention BiLSTM + BERT**
Embedding Type	FastText (static)	FastText (static)	FastText (static)	BERT (contextual)
Contextual Awareness	Limited	Limited	Bidirectional	Bidirectional + Contextual
Focus on Relevant Tokens	Uniform	Local (n-grams)	Uniform	Attention on most relevant tokens
Handling Long Sequences	Poor	Fixed receptive field	Good	Excellent
Interpretability	Limited	Limited	Limited	High (via attention weights)

Table 11: Comparison of Features Across Different Model Architectures

Model	Test Accuracy
RNN	0.7261
CNN	0.7833
BiLSTM with Dropout	0.7570
BiGRU	0.7533
Combined Attention BiLSTM + BERT	0.8105

Table 12: Best test accuracy scores for the different model architectures

Key Advantages:

Firstly, in **contextual embeddings (BERT vs. FastText)**, FastText embeddings are static and fail to adapt to context while BERT embeddings are dynamic, improving the handling of ambiguity and complex sentences. Secondly, in **relevance via attention**, the attention mechanism focuses on the most critical tokens in the sequence, reducing noise. Lastly, for **bidirectionality**, BiLSTM captures both forward and backward dependencies, while attention emphasizes their most important aspects.

4 Conclusion

The **Combined Attention BiLSTM + BERT Embeddings model** outperforms the standalone RNNs, CNNs, BiLSTM and BiGRU with FastText embeddings, achieving the highest accuracy of **81.05%**.

Key Summary Points for how the Combined Attention BiLSTM + BERT Embeddings model surpassed the other models:

- Leveraged BERT embeddings for contextual understanding.
- Focused on relevant tokens via the attention mechanism.
- Efficiently utilized bidirectional dependencies.
- Provided better generalization with dropout regularization.