NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC4002/CE4045/CZ4045 Natural Language Processing

Assignment

Group ID: G10

Name	Matriculation Number	Contribution
Chung Zhi Xuan	U2220300H	16.6% (1,2,3)
Kway Yi Shen	U2121308J	16.6% (1,2,3)
Yeoh Yu Shyan	U2122670F	16.6% (1,2,3)
Eddy Cheng Kuan Quan	U2121953D	16.6% (1,2,3)
Toh Leong Seng	U2121181G	16.6% (1,2,3)
Prakritipong Phuvajakrt	U2222277F	16.6% (1,2,3)

Everyone contributes by holding weekly meetings to discuss and compare the best solutions. We also support each other whenever there's uncertainty.

Question 1: Word Embedding

(a) What is the size of the vocabulary formed from your training data?

18029

*(b) We use **OOV** (out-of-vocabulary) to refer to those words that appear in the training data but not in the Word2vec (or Glove) dictionary. How many OOV words exist in your training data?*

3612

(c) The existence of the OOV words is one of the well-known limitations of Word2vec (or Glove). Without using any transformer-based language models (e.g., BERT, GPT, T5), what do you think is the best strategy to mitigate such limitations? Implement your solution in your source code. Show the corresponding code snippet.

To handle out-of-vocabulary (OOV) words, we considered three main options.

1. **Cosine similarity**

The steps include:

1. Converting the OOV word into a vector by either summing the embeddings of its subwords components or by the embedding of nearby words.
2. Selecting potential replacement words based on their similarity in the vector space.
3. Calculating cosine similarity between the OOV word and replacement candidates.
4. Choosing a candidate with a high cosine similarity score, ideally close to 1, as the replacement.

The advantages of using this method to handle OOV words is that it is scale-invariant, focusing on the direction of word vectors instead of their magnitude, making it ideal for comparing word embeddings. It is also an efficient method as it replaces the OOV words by finding a close semantic replacement within the existing vocabulary.

The function used to find the replacement word is defined as `*find_replacement_word*`, as seen in **Diagram 1**. For each OOV word, our function `*find_replacement_word*` attempts to find semantically similar words by first checking if subwords or morphemes (like prefixes or suffixes) have embeddings within the pre-trained model. If these subwords exist, their embeddings are averaged to create a composite vector representing the OOV word.

Once the composite OOV vector is generated, we use cosine similarity to find the most similar words within the vocabulary. By comparing directional alignment rather than vector magnitude, cosine similarity efficiently identifies a word that has the closest semantic relation to the OOV term.

If a replacement word is found, we assign the vector of this replacement to the OOV word's slot in the embedding matrix. If no suitable replacement exists, a vector of random values (drawn from a normal distribution) is assigned. This ensures that the embedding matrix is fully populated, preventing model errors.

2. <u>Byte-Pair Encoding (BPE) and Stemming</u>

The steps are as follows:

1. Stemming: We utilized PorterStemmer to reduce the unknown word to its room form
2. Subword Encoding: If the stemmed form does not match any word, we use a tokenization method "*SentencePiece*" to break down the word into smaller subwords.
3. Choosing the longest matching subword.

To perform the Subword Encoding, we trained our SentencePiece model using text from train_dataset (**Diagram 2**).

By breaking it down into subwords, BPE can handle OOV words, while stemming decreases the variety of word forms to be learned, enhancing computation efficiency.

3. <u>Back-Off Method.</u>

This method combines pre-processing, spelling correction and stopword handling with method 2, allowing for a more comprehensive and detailed method of handling OOV words (**Diagram 3**). The steps are as follows:

1. Stopword handling: As stopwords ('a', 'and', 'in', 'the', etc.) contribute little semantic value, we opted to exclude them. This optimizes performance of the algorithm and reduces computational runtime.
2. Pre-Processing: Regex is used to remove separators and punctuation.
3. Spelling Correction: A Symmetric Delete Spelling Correction algorithm, Symspell, is used to detect and resolve possible spelling errors.
4. Word2Vec Lookup: The resulting word is then re-compared in Word2Vec.

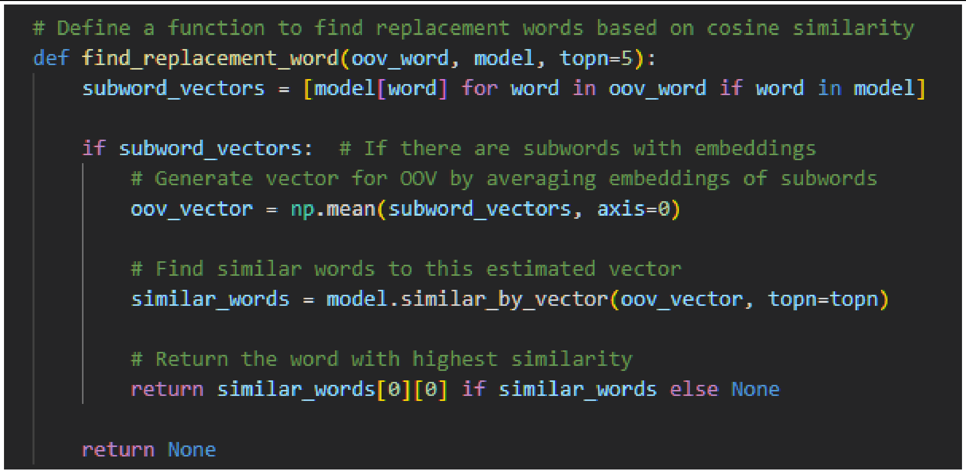
5. Byte-Pair encoding & Subword Lookup: Similar to method 2, Byte-Pair Encoding was done to derive subwords.

Based on our observations, we found that the **third option** slightly outperforms the rest, while maintaining a reasonable overhead. Due to the multi-layered approach, most unknown words were able to be effectively mapped. As such, we opted to use the **Back-Off Method** for future experiments.

4. Additional Options

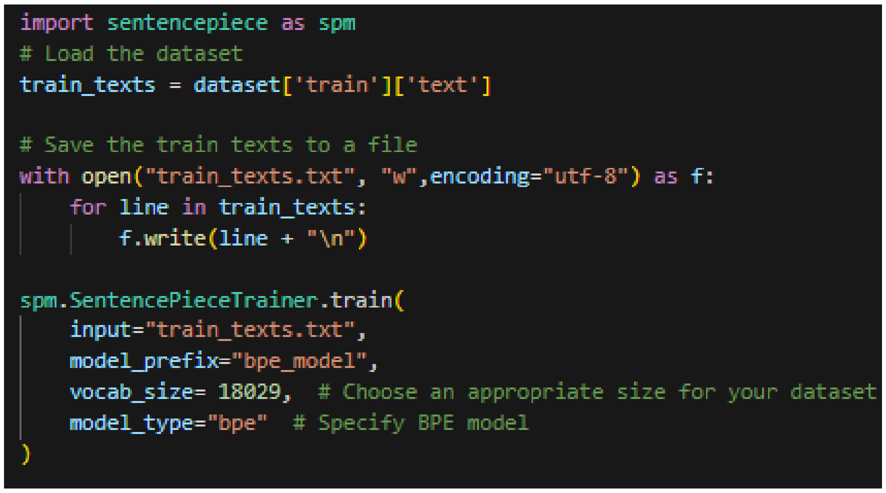
We integrated the Back-Off Method with Soundex specifically to handle OOV tokens like names, as names often exhibit diverse variations in spelling. Soundex helps by capturing phonetic similarities, allowing for better matching across different spellings of names. However, Soundex has limitations: it lacks adaptability to broader vocabulary variations, operates solely on phonetic patterns rather than context, and is resource-intensive. These drawbacks make it less effective for general OOV handling beyond name tokens.

We also experimented with combining the backoff method by replacing spelling correction with edit distance. However, this approach proved ineffective due to adjustments in our pipeline. For instance, a word like "leopar" that isn't found in word2vec will be split by BPE into "leo" and "par." The pipeline will then select the longest subword, and edit distance will try to identify the correct word. Besides being computationally expensive, edit distance did not yield the desired accuracy in this setup.



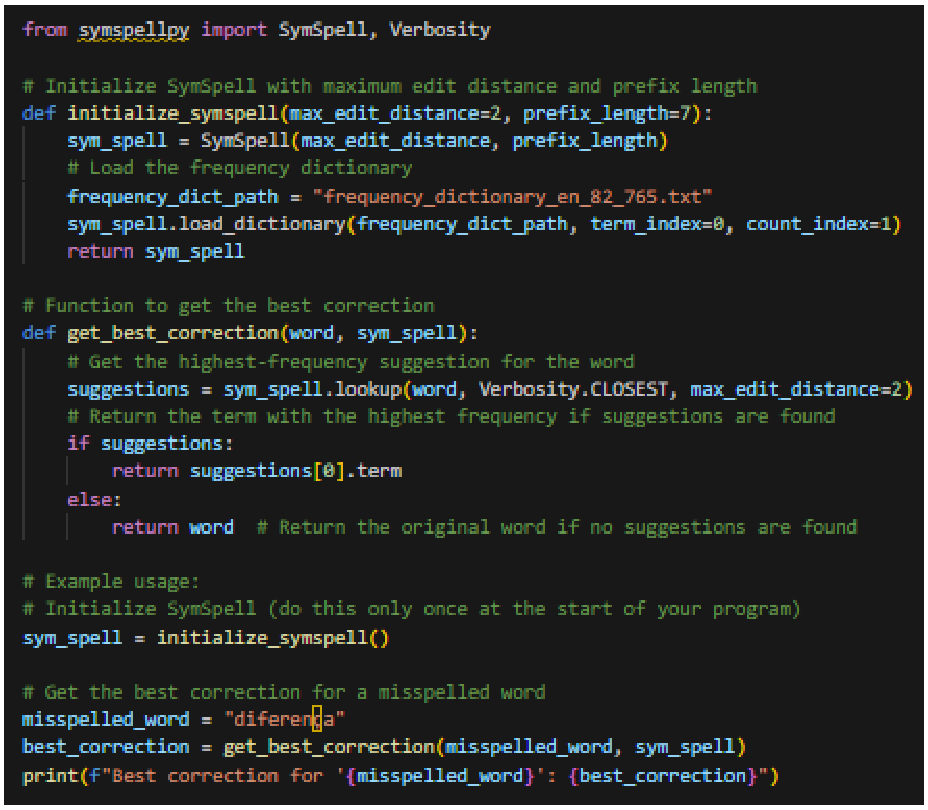
```python
# Define a function to find replacement words based on cosine similarity
def find_replacement_word(oov_word, model, topn=5):
    subword_vectors = [model[word] for word in oov_word if word in model]

    if subword_vectors:  # If there are subwords with embeddings
        # Generate vector for OOV by averaging embeddings of subwords
        oov_vector = np.mean(subword_vectors, axis=0)

        # Find similar words to this estimated vector
        similar_words = model.similar_by_vector(oov_vector, topn=topn)

        # Return the word with highest similarity
        return similar_words[0][0] if similar_words else None

    return None
```

Diagram 1: find_replacement_word function



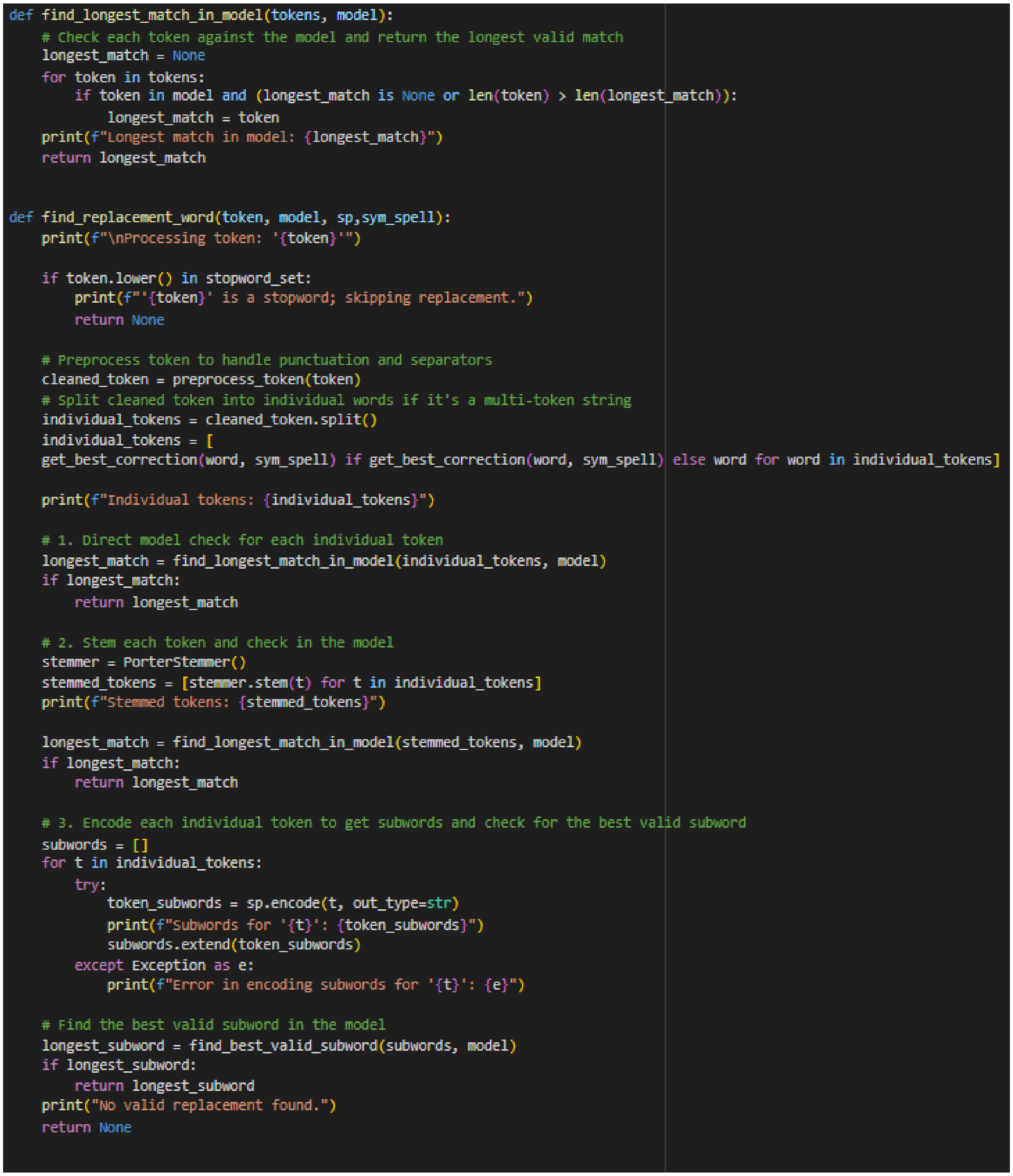
```python
import sentencepiece as spm
# Load the dataset
train_texts = dataset['train']['text']

# Save the train texts to a file
with open("train_texts.txt", "w",encoding="utf-8") as f:
    for line in train_texts:
        f.write(line + "\n")

spm.SentencePieceTrainer.train(
    input="train_texts.txt",
    model_prefix="bpe_model",
    vocab_size= 18029,  # Choose an appropriate size for your dataset
    model_type="bpe"  # Specify BPE model
)
```

Diagram 2: Training of our SentencePiece Model with the training Dataset



```python
from symspellpy import SymSpell, Verbosity

# Initialize SymSpell with maximum edit distance and prefix length
def initialize_symspell(max_edit_distance=2, prefix_length=7):
    sym_spell = SymSpell(max_edit_distance, prefix_length)
    # Load the frequency dictionary
    frequency_dict_path = "frequency_dictionary_en_82_765.txt"
    sym_spell.load_dictionary(frequency_dict_path, term_index=0, count_index=1)
    return sym_spell

# Function to get the best correction
def get_best_correction(word, sym_spell):
    # Get the highest-frequency suggestion for the word
    suggestions = sym_spell.lookup(word, Verbosity.CLOSEST, max_edit_distance=2)
    # Return the term with the highest frequency if suggestions are found
    if suggestions:
        return suggestions[0].term
    else:
        return word  # Return the original word if no suggestions are found

# Example usage:
# Initialize SymSpell (do this only once at the start of your program)
sym_spell = initialize_symspell()

# Get the best correction for a misspelled word
misspelled_word = "diferenca"
best_correction = get_best_correction(misspelled_word, sym_spell)
print(f"Best correction for '{misspelled_word}': {best_correction}")
```

Diagram 3: Symspell for Spelling Correction



```python
def find_longest_match_in_model(tokens, model):
    # Check each token against the model and return the longest valid match
    longest_match = None
    for token in tokens:
        if token in model and (longest_match is None or len(token) > len(longest_match)):
            longest_match = token
    print(f"Longest match in model: {longest_match}")
    return longest_match


def find_replacement_word(token, model, sp,sym_spell):
    print(f"\nProcessing token: '{token}'")

    if token.lower() in stopword_set:
        print(f"'{token}' is a stopword; skipping replacement.")
        return None

    # Preprocess token to handle punctuation and separators
    cleaned_token = preprocess_token(token)
    # Split cleaned token into individual words if it's a multi-token string
    individual_tokens = cleaned_token.split()
    individual_tokens = [
    get_best_correction(word, sym_spell) if get_best_correction(word, sym_spell) else word for word in individual_tokens]

    print(f"Individual tokens: {individual_tokens}")

    # 1. Direct model check for each individual token
    longest_match = find_longest_match_in_model(individual_tokens, model)
    if longest_match:
        return longest_match

    # 2. Stem each token and check in the model
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(t) for t in individual_tokens]
    print(f"Stemmed tokens: {stemmed_tokens}")

    longest_match = find_longest_match_in_model(stemmed_tokens, model)
    if longest_match:
        return longest_match

    # 3. Encode each individual token to get subwords and check for the best valid subword
    subwords = []
    for t in individual_tokens:
        try:
            token_subwords = sp.encode(t, out_type=str)
            print(f"Subwords for '{t}': {token_subwords}")
            subwords.extend(token_subwords)
        except Exception as e:
            print(f"Error in encoding subwords for '{t}': {e}")

    # Find the best valid subword in the model
    longest_subword = find_best_valid_subword(subwords, model)
    if longest_subword:
        return longest_subword
    print("No valid replacement found.")
    return None
```

Diagram 4: find_longest_match_in_model() and find_replacement_word() functions

<u>Question 2: RNN</u>

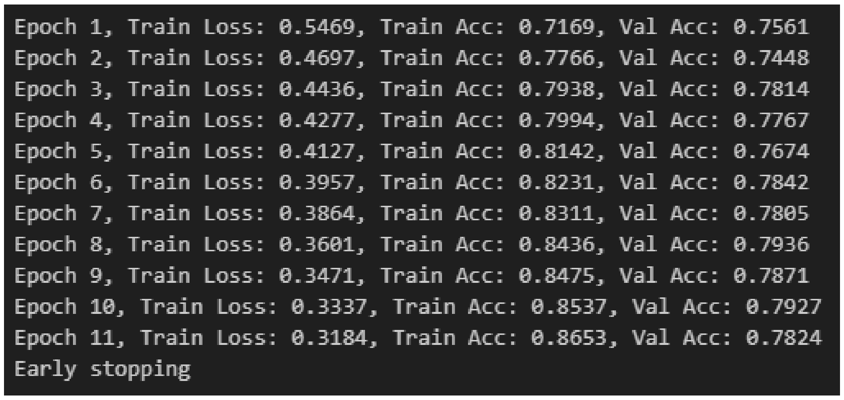
(a) Report the final configuration of your best model, namely the number of training epochs, learning rate, optimizer, batch size.

The best model selected is the RNN Max pooling model with the following configurations:

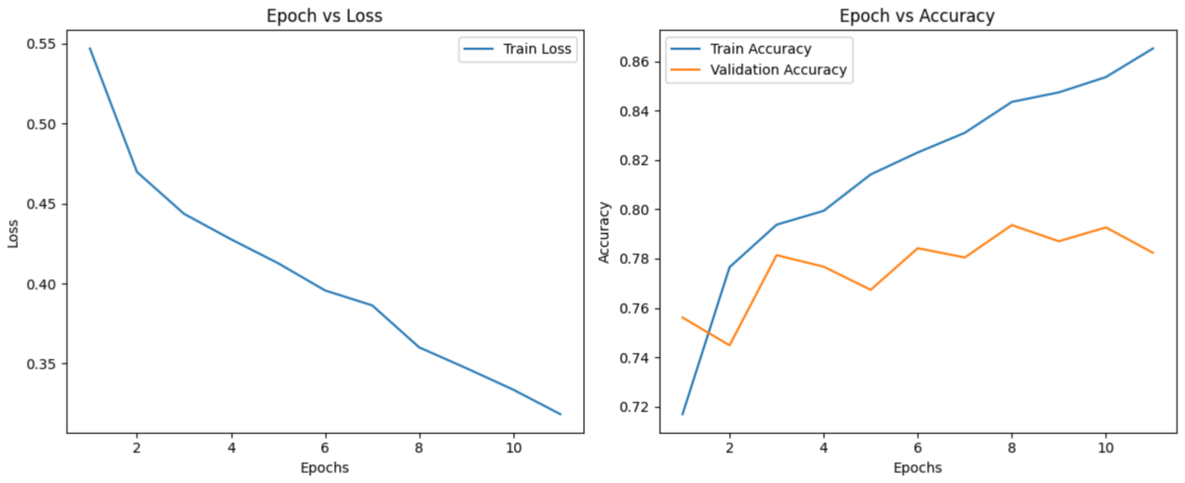
- **Training epochs**: 11 epochs (early stopped)
- **Learning rate**: 0.001
- **Optimizer**: Adam
- **Batch size**: 32

(b) Report the accuracy score on the test set, as well as the accuracy score on the validation set for each epoch during training.

Test accuracy (Max pooling): 0.7871
The accuracy score on the validation set for each epoch during training can be seen in the figure below:



```
Epoch 1, Train Loss: 0.5469, Train Acc: 0.7169, Val Acc: 0.7561
Epoch 2, Train Loss: 0.4697, Train Acc: 0.7766, Val Acc: 0.7448
Epoch 3, Train Loss: 0.4436, Train Acc: 0.7938, Val Acc: 0.7814
Epoch 4, Train Loss: 0.4277, Train Acc: 0.7994, Val Acc: 0.7767
Epoch 5, Train Loss: 0.4127, Train Acc: 0.8142, Val Acc: 0.7674
Epoch 6, Train Loss: 0.3957, Train Acc: 0.8231, Val Acc: 0.7842
Epoch 7, Train Loss: 0.3864, Train Acc: 0.8311, Val Acc: 0.7805
Epoch 8, Train Loss: 0.3601, Train Acc: 0.8436, Val Acc: 0.7936
Epoch 9, Train Loss: 0.3471, Train Acc: 0.8475, Val Acc: 0.7871
Epoch 10, Train Loss: 0.3337, Train Acc: 0.8537, Val Acc: 0.7927
Epoch 11, Train Loss: 0.3184, Train Acc: 0.8653, Val Acc: 0.7824
Early stopping
```



(c) RNNs produce a hidden vector for each word, instead of the entire sentence. Which methods have you tried in deriving the final sentence representation to perform sentiment classification? Describe all the strategies you have implemented, together with their accuracy scores on the test set.

For deriving sentence representations, we implemented different strategies:

- **Last Hidden State**: Use the last hidden state of the RNN. Uses the cumulated information from the entire input sequence but may lose information from earlier parts of the sequence when the sequence is long.
- **Average Pooling**: Calculates the average of feature dimensions of all hidden states which provides a balanced evaluation across the whole input sequence.
- **Max Pooling**: Uses the maximum value of each feature dimension of all hidden states allowing prominent features of the input sequence to be captured more accurately.

For deriving the best hyperparameters for the model, Grid search strategy was implemented to find the best configuration for Learning rate, Optimizer and Batch size on the validation dataset.

- **Training epochs**: 20 epochs
- **Early stop loss patience**: 3
- **Learning rate**: [0.001, 0.005, 0.01, 0.1]
- **Optimizer**: [SGD, Adam, RMSprop, Adagrad]
- **Batch size**: [16, 32, 64]

Max pooling performed the best with an accuracy of 0.7871 vs 0.7682 using average pooling vs 0.4981 using the last hidden state method.

Question 3: Enhancement

(a) Report the accuracy score on the test set when the word embeddings are updated (Part 3.1).

Test Accuracy (RNN (Update Word Embeddings)): 0.8096

(b) Report the accuracy score on the test set when applying your method to deal with OOV words in Part 3.2.

Test Accuracy (RNN (Handle OOV with Word Embeddings Update)): 0.7992

Test Accuracy (RNN (Handle OOV while Freezing Word Embeddings)): 0.6538

Based on the results above, it can be seen that the method used to handle OOV has helped to improve the accuracy when word embeddings are updated.

(c) Report the accuracy scores of biLSTM and biGRU on the test set (Part 3.3).

Test Accuracy (bilSTM): 0.7833
Test Accuracy (biGRU): 0.8039

(d) Report the accuracy scores of CNN on the test set (Part 3.4).

Test Accuracy (CNN): 0.7917

(e) Describe your final improvement strategy in Part 3.5. Report the accuracy on the test set using your improved model.

Ensemble Methods: This setup aggregates the model predictions by averaging the class probabilities across each model, then assigns the final prediction based on the highest average probability. This approach should increase overall stability and potentially improve the test accuracy of your sentiment classification task by leveraging the diverse perspectives of different models.

The models we used for the ensemble were RNN Max Pooling, RNN (Handle OOV with Word Embeddings Update), biLSTM and biGRU as they are the highest performing models from our observations.

Test Accuracy (Ensemble): 0.8227

From the test accuracies, we see that when employing ensemble methods with **soft voting (probability averaging)** gives us a higher test accuracy compared to the test accuracies of each individual model.

(f) Compare the results across different solutions above and describe your observations with possible discussions.

'RNN (Handle OOV)': 0.6538

'RNN (Handle OOV with Word Embeddings Update)': 0.7992

'biLSTM': 0.7833

'biGRU': 0.8039

'CNN': 0.7917

'RNN (Update Word Embeddings)': 0.8096

'Ensemble': 0.8227

RNN (Handle OOV) is a basic RNN implementation, which includes handling OOV tokens, however it does not leverage on bidirectionality or GRUs or update work embeddings. Therefore, it performs the worst out of the models.

RNN (Handle OOV and word embedding update) includes updating word embeddings directly with OOV handling to enhance the model's effectiveness compared to the basic RNN model (0.6538 vs 0.7992). Since the word embeddings are now more fine-tuned to capture features in the training data, this results in the model having a higher accuracy.

Both **biLSTM** and **biGRU** displayed similar performances. This is due to them both possessing bidirectional nature which allows them to capture contextual information from both forward and backwards directions. Even though biLSTM has a more complex architecture compared to biGRU to capture longer range dependencies, for moderate sequence lengths, biGRU is still able to perform just as well as biLSTM due to similarities in their structures.

CNN performed relatively similarly to both biLSTM and biGRU. CNN is known for parallelism and feature extraction capabilities. By using filters over windows of text it can capture nearby context which can be comparable to bidirectional architectures for shorter dependencies. The inclusion of additional stopwords reduces noise, thereby enhancing the accuracy of the CNN model.

RNN (Update Word Embeddings) performance slightly better than all of the previous approaches. This suggests that fine-tuning by learning task-specific embedding aids in capturing nuance of the data leading to better generalization of the model against unseen data.

The **Ensemble method** with soft voting (probability averaging) achieves the highest performance by combining the strengths of diverse models, creating a more robust and generalized solution for speech summarization. By averaging the probability distributions of individual models, the ensemble balances out the specific weaknesses of each model—such as the biLSTM's ability to capture bidirectional context and the CNN's strength in local feature extraction—resulting in a more reliable final prediction.

Appendix

Word2Vec	**Cosine Similarity**	**BPE + Stemming**	**Back Off Method (sympell+ stopwords)**
RNN (Handle OOV)	**0.7777**	0.7589	0.6538
RNN (Handle OOV with Word Embeddings Update)	0.7711	0.7636	**0.7992**
biLSTM	0.7833	**0.7917**	0.7833
biGRU	0.7270	0.8030	**0.8039**
CNN	0.7795	0.7767	**0.7917**
RNN (Update Word Embeddings)	0.7870	0.7964	**0.8095**
Ensemble	0.8123	0.8114	**0.8227**

Glove	**Cosine Similarity**	**BPE + Stemming**	**Back Off Method (sympell+ stopwords)**
RNN (Handle OOV)	0.7804	0.7870	0.7654
RNN (Handle OOV with Word Embeddings Update)	0.8058	0.8170	0.8142
biLSTM	0.8039	0.8114	0.8245
biGRU	0.8142	0.8030	0.8198
CNN	0.7410	0.7842	0.7908
RNN (Update Word Embeddings)	0.7814	0.6726	0.8133
Ensemble	0.820	0.8264	0.8339