NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Assignment for SC4002 / CE4045 / CZ4045

AY2023-2024

Group ID: G39

Group members:

Name	Matric No.
Seow Joo Hou, Javier	U2022843E
Heng Wei Jie	U2023731J
Jarel Tan Zhi Wen	U2121582H
Tan Wei Chuan	U2121801B
Jeon Subeen	U2023021K

Table of Contents

Table of Contents 1
Part 1 2
Question 1.1 2
Question 1.2 2
Part a 2
Part b 2
Question 1.3 3
Part a 3
Part b 4
Part c 6
Part d 7
Question 2 7
Part a 7
Part b 7
Part c 8
Part d 9
Part e 9

Part 1

Question 1.1

The instantiated word2vec model, implemented within our environment was trained to capture semantic similarities for an extensive lexicon of words. This simple question simply utilises the embedding to compute the cosine similarity between specified words and other words in the vocabulary. The utilisation involves invoking the 'most_similar' method with the designated word as the focal point and the parameters 'topn=1' to retrieve the most closely related word. Since the return type is a tuple, we simply use [0] and [1] index to obtain the word and similarity score respectively.

a) The most similar word to 'student' is 'students' with cosine similarity: 0.7294867038726807
b) The most similar word to 'Apple' is 'Apple_AAPL' with cosine similarity: 0.7456986308097839
c) The most similar word to 'apple' is 'apples' with cosine similarity: 0.720359742641449

Question 1.2

Part a

To begin, we read the respective train, dev and test data. After which, we simply get the description of the data using method get_info on the respective dataset. We then output again using the indexes of [0] and [1] to obtain the number of sentences and unique labels for each dataset respectively.

1. Training set: 14989 sentences
   Labels: {'I-PER', 'I-MISC', 'B-LOC', 'I-LOC', 'O', 'B-ORG', 'I-ORG', 'B-MISC'}
2. Development set: 3468 sentences
   Labels: {'I-PER', 'I-MISC', 'I-LOC', 'O', 'I-ORG', 'B-MISC'}
3. Test set: 3683 sentences
   Labels: {'I-PER', 'I-MISC', 'B-LOC', 'I-LOC', 'O', 'B-ORG', 'I-ORG', 'B-MISC'}

Part b

The B-I-O tagging scheme is used to identify named entities in text. After selecting a sentence that has at least two multi-word entities as per required by the question.

Sentences with at least two multi-word entities:

```
              word  label
47         Germany  I-LOC
48              's      O
49  representative      O
50              to      O
51             the      O
52        European  I-ORG
53           Union  I-ORG
54              's      O
55      veterinary      O
56       committee      O
57          Werner  I-PER
58       Zwingmann  I-PER
59            said      O
60              on      O
61       Wednesday      O
62       consumers      O
63          should      O
64             buy      O
65       sheepmeat      O
66            from      O
67       countries      O
68           other      O
69            than      O
70         Britain  I-LOC
71           until      O
72             the      O
73      scientific      O
74          advice      O
75             was      O
76         clearer      O
77               .      O
```

Afterwhich, we iterate through each row (*that includes both word and label*). It is trivial to find the first word entity, it is either 1) a prefix with ‘B’ or 2) a prefix with ‘I’ and a different label from the previous word indicates a new entity. In both scenarios, the row (*both word and label*) is stored, since it has the potential to be a multi-word entity. In the event where the prefix is ‘I’ and the label matches the preceding word, it indicates a continuation of the current entity and will be appended to the existing entity.

Output after following logic as described above:

```
Germany (LOC)
European Union (ORG)
Werner Zwingmann (PER)
Britain (LOC)
```

Question 1.3

Part a

We implement the same strategy for both Test and Training Set for Out-Of-Vocabulary (OOV) words. When a term within the training set is not present, the algorithm derives the average embedding by considering the surrounding words that are present in the pre-trained embeddings. To enhance the representation of the OOV word, this method involves the inclusion of the two preceding and two subsequent words surrounding the target word. In the event that none of its surrounder words are found in the pretrained embeddings, a zero vector of the same dimensionality as the pretrained embedding is assigned to the OOV word.

The strategy of averaging the embeddings of surrounding words utilises the idea that words can and should be inferred from its context to obtain a close approximation of a word's position in the semantic space. To ensure that the model receives input of consistent dimensionality across all instances in the event of extreme cases where neither target word nor its surrounding words are not present in the pre-trained embeddings, zero vectors are used. With word2vec being a huge dataset, the number of extreme cases occurs only within a small handful of cases (*~8%, 4339 [zero vector] out of 52176 [average embeddings]*) . Though simplistic, we are able to obtain a computationally efficient method, further changes could have been considered if the f1_score is not ideal or too low.

```python
def handle_oov_words(word, surrounding_words, embeddings):
    global zero_vector_count  # for counter of zero vector
    global average_embedding_count # for embeeding use count

    if word in embeddings:
        return embeddings[word]

    elif surrounding_words:
        surrounding_embeddings = [embeddings[w] for w in surrounding_words if w in embeddings]
        if surrounding_embeddings:
            # computing average
            avg_embedding = np.mean(surrounding_embeddings, axis=0)
            average_embedding_count += 1
            return avg_embedding
    # only return zero vector if otherwise
    zero_vector_count += 1
    return np.zeros(embeddings.vector_size)

def get_embedding_for_sentence(sentence, embeddings):
    sentence_embeddings = []
    for i, word in enumerate(sentence):
        surrounding_words = sentence[max(0, i-2):i] + sentence[i+1:min(len(sentence), i+3)]
        word_embedding = handle_oov_words(word, surrounding_words, embeddings)
        sentence_embeddings.append(word_embedding)
    return sentence_embeddings
```

Part b

We used Bi-LSTM followed by a linear layer to produce the final vector representation for each word, which is then used to predict the label of each word. Since 1.3b) consists of several parts, we will break down the question and answer them separately.

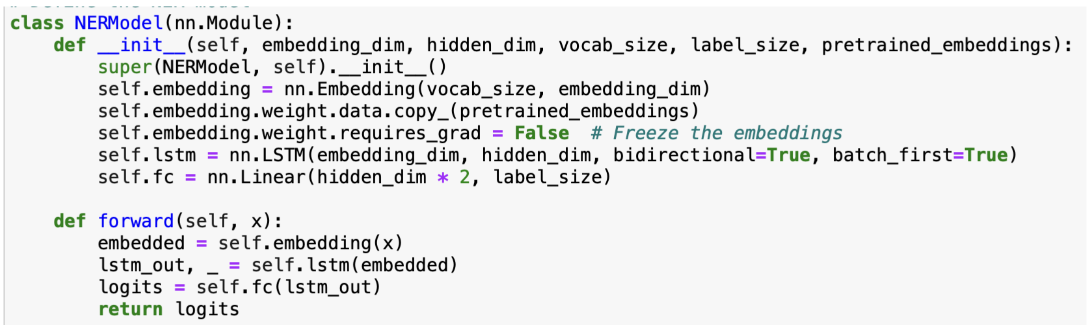
1. Neural Network Architecture:
    a. Embedding Layer: Transform each word index into a dense vector representation. As tasked in the assignment, the model uses the pre-trained Word2Vec embeddings, acting as a lookup table and retrieves corresponding embedded for each word index in the input sequence

    ```python
    word_to_index = {word: i for i, word in enumerate(word_vocab)}
    label_to_index = {label: i for i, label in enumerate(label_set)}
    index_to_label = {i: label for label, i in label_to_index.items()}
    ```

    b. Bi-LSTM: Processes the sequence of word embedding from Word2Vec in both forward and backward directions. Captures contextual information from both past and future words in the sequence with the ability to capture sequential/temporal dependencies in the data

c. Linear Layer (Fully Connected Layer): Concatenated output from Bi-LSTM later is passed through this linear layer. It projects high-dimensional LSTM outputs to the size of label space.

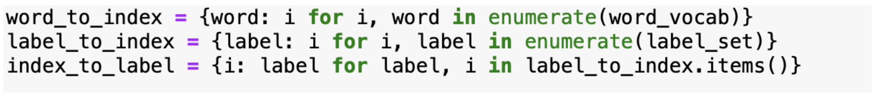
d. Softmax Function: Output from linear layer goes through softmax function during training that convert the scores into probability.



```python
class NERModel(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, label_size, pretrained_embeddings):
        super(NERModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding.weight.data.copy_(pretrained_embeddings)
        self.embedding.weight.requires_grad = False  # Freeze the embeddings
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, bidirectional=True, batch_first=True)
        self.fc = nn.Linear(hidden_dim * 2, label_size)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        logits = self.fc(lstm_out)
        return logits
```

e. Also uses vocabulary and maps words and labels to indices ensuring that the format is fed into the neural network



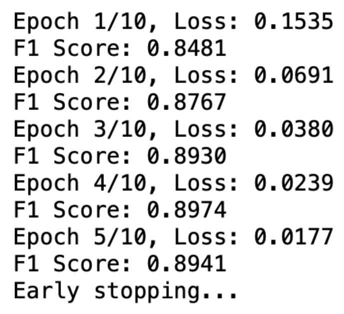
```python
word_to_index = {word: i for i, word in enumerate(word_vocab)}
label_to_index = {label: i for i, label in enumerate(label_set)}
index_to_label = {i: label for label, i in label_to_index.items()}
```

2. Mathematical Functions for Forward Computation:
    a. Linear Layer: Each concatenated vector is passed through a linear layer which then computes a vector score for each possible label. Applies linear transformation to LSTM outputs to map them to space of label set
    b. Log Softmax: Lastly, a log softmax function is applied along the label dimension to the scores to obtain the log probability of each label for each word in the sentence
3. Parameters Being Updated:
    a. Bi-LSTM Layer
        i. Weights for Input Forget, Cell and Output Gates: Matrices multiplied with input embedding and previous hidden state to determine the gates' activations and cell state update
        ii. Bias term Input Forget, Cell and Output Gates: Vectors added to gates' activations to offset results before applying non-linear activation function
        iii. Two sets of parameters for the forward and backward direction for Bi-LSTM
    b. Linear Layer
        i. Weight: Matrix multiplied with output of LSTM to produce logits for each label
        ii. Biases: Vector added to the result of weight multiplication to produce logits for each label

4. Size of Parameters:
    a. Bi-LSTM layer
        i. Each LSTM has 2 set of weights and biases, one each for forward and backward pass
        ii. Each set of weight matrix size: ([hidden_dim], [embedding_dim] + [hidden_dim])
        iii. Total size for weights direction: (4*2*[hidden_dim] *([embedding_dim] + [hidden_dim]) = 4*2*256*556 = **1,132,544**
        iv. Each set of biases size: [hidden_dim]
        v. Total size for bias direction = (4*2*[hidden_dim]) = 4*2*256 = **2,048**
    b. Linear Layer
        i. Weight matrix size: [label_labels, 2*hidden_dim]
        ii. Bias Vector Size: [label_labels]
5. Length of Final Vector Representation:
    a. Length of the final vector is equal to the number of unique labels since the output of the neural network for each word is a vector of scores
6. Vector fed to Softmax Classifier: Vector fed to softmax classifier has a length corresponding to the number of possible labels for each word. Softmax classifier operates on the logits produced by the final linear layer. It uses them to predict the probability distribution over all possible labels for each word in a sequence

Part c

While we planned for a maximum of 10 epochs for the model training, we have also incorporated an early stopping mechanism to preempt overfitting. Overfitting occurs when a model learns the training data too well (noise and outliers), which negatively impacts on its performance on new and unseen data. As seen from the figure below, training was halted after 5 epochs.


```
Epoch 1/10, Loss: 0.1535
F1 Score: 0.8481
Epoch 2/10, Loss: 0.0691
F1 Score: 0.8767
Epoch 3/10, Loss: 0.0380
F1 Score: 0.8930
Epoch 4/10, Loss: 0.0239
F1 Score: 0.8974
Epoch 5/10, Loss: 0.0177
F1 Score: 0.8941
Early stopping...
```

The total time taken for all epochs was 839.22 seconds and averages out to 167.84 seconds per epoch.

Part d

```
predicted_labels = [predict_labels(ner_model, sentence) for sentence in dev_sentences]
true_labels = [[label for _, label in sentence] for sentence in dev_sentences]

f1 = f1_score(true_labels, predicted_labels)
print(f'F1 Score: {f1:.2f}')
```

```
F1 Score: 0.89
```

Question 2

Part a

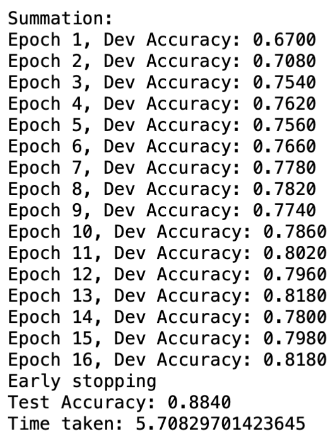
We used the numpy's random choice function to pseudo-randomly obtain '0', '1', '5', '2' and 'OTHERS' as the 5 classes. The 'OTHERS' category serves as an aggregates of all labels not included in the selected subset. We fixed the seed for the random number function to ensure reproducibility of results for consistent validation and documentation. Our design allows for a focused classification task, and hence improves the performance of the classifier.

Part b

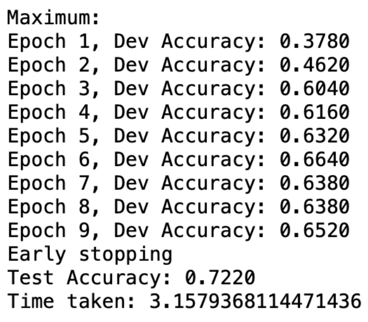
We performed multiple aggregation methods for our model to better understand the nature of the downstream task and empirical performance on the respective development and test set. Stated below are their details:

- Summation: Sums word embedding in sentence
  - + Preserves word frequency information
  - + Suitability for certain tasks
  - - Could lead to large magnitude vector for long sentences
- Max: Obtain maximum value across each dimension of word embedding in sentence
  - + Captures important features
  - + Task-Specific Performance
- Mean: Obtain average across each dimension of word embedding within sentence
  - + Captures overall representation of sentence
  - + Robust to variance in sentence length and structure
  - + Smooths out outlier influence and provides central tendency

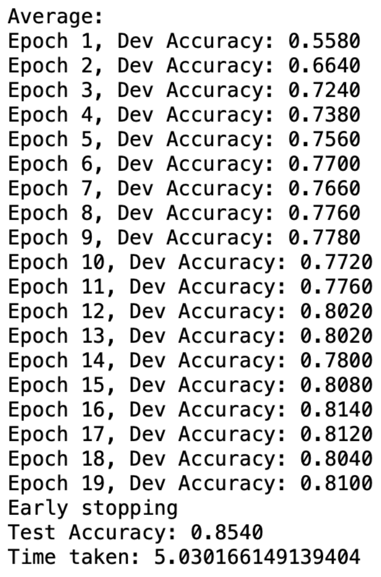
Their respective screenshots of accuracy results and other relevant information:



```
Summation:
Epoch 1, Dev Accuracy: 0.6700
Epoch 2, Dev Accuracy: 0.7080
Epoch 3, Dev Accuracy: 0.7540
Epoch 4, Dev Accuracy: 0.7620
Epoch 5, Dev Accuracy: 0.7560
Epoch 6, Dev Accuracy: 0.7660
Epoch 7, Dev Accuracy: 0.7780
Epoch 8, Dev Accuracy: 0.7820
Epoch 9, Dev Accuracy: 0.7740
Epoch 10, Dev Accuracy: 0.7860
Epoch 11, Dev Accuracy: 0.8020
Epoch 12, Dev Accuracy: 0.7960
Epoch 13, Dev Accuracy: 0.8180
Epoch 14, Dev Accuracy: 0.7800
Epoch 15, Dev Accuracy: 0.7980
Epoch 16, Dev Accuracy: 0.8180
Early stopping
Test Accuracy: 0.8840
Time taken: 5.70829701423645
```



```
Maximum:
Epoch 1, Dev Accuracy: 0.3780
Epoch 2, Dev Accuracy: 0.4620
Epoch 3, Dev Accuracy: 0.6040
Epoch 4, Dev Accuracy: 0.6160
Epoch 5, Dev Accuracy: 0.6320
Epoch 6, Dev Accuracy: 0.6640
Epoch 7, Dev Accuracy: 0.6380
Epoch 8, Dev Accuracy: 0.6380
Epoch 9, Dev Accuracy: 0.6520
Early stopping
Test Accuracy: 0.7220
Time taken: 3.1579368114471436
```



```
Average:
Epoch 1, Dev Accuracy: 0.5580
Epoch 2, Dev Accuracy: 0.6640
Epoch 3, Dev Accuracy: 0.7240
Epoch 4, Dev Accuracy: 0.7380
Epoch 5, Dev Accuracy: 0.7560
Epoch 6, Dev Accuracy: 0.7700
Epoch 7, Dev Accuracy: 0.7660
Epoch 8, Dev Accuracy: 0.7760
Epoch 9, Dev Accuracy: 0.7780
Epoch 10, Dev Accuracy: 0.7720
Epoch 11, Dev Accuracy: 0.7760
Epoch 12, Dev Accuracy: 0.8020
Epoch 13, Dev Accuracy: 0.8020
Epoch 14, Dev Accuracy: 0.7800
Epoch 15, Dev Accuracy: 0.8080
Epoch 16, Dev Accuracy: 0.8140
Epoch 17, Dev Accuracy: 0.8120
Epoch 18, Dev Accuracy: 0.8040
Epoch 19, Dev Accuracy: 0.8100
Early stopping
Test Accuracy: 0.8540
Time taken: 5.030166149139404
```

After reviewing the accuracy results and additional relevant metrics, we have selected for the average method as our final aggregation method for the following factor:

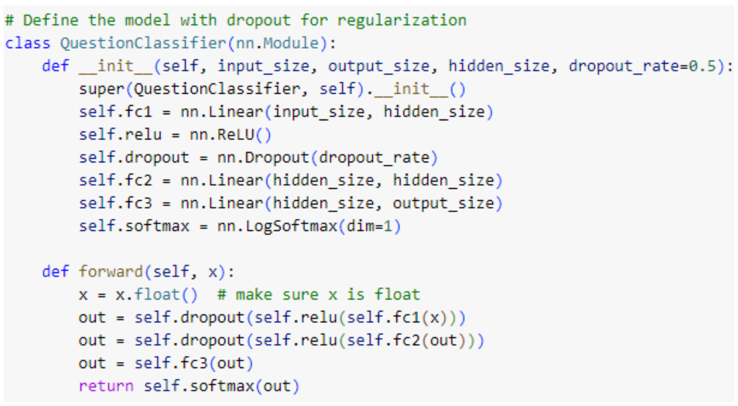
1. Fixed-length Output: The mean, similar to the sum, generates a fixed-length vector independent of sentence length
2. Robustness to sentence Length: Exhibits reduced sensitivity to sentence length variations, ensuring that longer sentences do not disproportionately influence the scale of the resulting vector
3. Test Accuracy: Although marginally lower than the summation approach, the mean aggregation method offers a more stable and consistent performance
4. General Application: The mean is generally more reliable and consistent across various sentence structures and lengths, making it versatile for diverse testing scenarios

Part c

Our chosen architecture is a simple feedforward neural network with one hidden layer and dropout for regularisation. The network's operational sequence first applies a linear transformation of the input followed by a ReLU activation function and dropout. The process is repeated twice before a final linear transformation and LogSoftmax function. During training, the model's parameters are continually updated to optimise the loss function's value. The specifics details of the neural network's architecture are as follows:

- Input Layer Size: Equal to the size of word embeddings
- Hidden Layer Size:128 neurons, offering sufficient complexity for feature extraction
- Output Layer Size: 5 neurons, corresponding to the class labels
- Activation Function: ReLU used after first and second linear transformation, introducing non-linearity to capture complex patterns

- Dropout Rate: 0.5, used after activation function to prevent overfitting by randomly zeroing some layer's outputs
- Output Activation Function: LogSoftmax, used to convert final layer's output to log-probabilities suitable for multi-class classification
- Loss Function: NLLLoss (Negative Log-Likelihood Loss), used in combination with LogSoftmax to form a cross-entropy loss framework ideal
- Optimization Algorithm: Adam optimizer with learning rate of 0.001 and L2 regularisation (weight decay) of e^{-4} to prevent overfitting and model generalisation



```
# Define the model with dropout for regularization
class QuestionClassifier(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, dropout_rate=0.5):
        super(QuestionClassifier, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = x.float()  # make sure x is float
        out = self.dropout(self.relu(self.fc1(x)))
        out = self.dropout(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return self.softmax(out)
```

Part d

The number of epochs set for the training loop is 20. Similar to the early stopping logic used in Question 1.3, the training will halt if there is no improvement for 3 consecutive epochs. The time taken to train the model for 20 epochs was ~16.04 seconds, showing that the model is generally computationally effective.

Part e

The model's accuracy on the development set started at 0.5580 and generally improved over the 20 epochs, the highest being 0.8140 at epoch 16. After epoch 12, the accuracy was fluctuating and could be an indicator of the model's max capacity for learning with the current architecture and hyperparameters. Conversely, the final test which was performed on a separate set of data not seen during training achieved an accuracy of 0.0.8540. The accuracy achieved was relatively high and serves as an indicator that the model has been generalised well.

```
Average:
Epoch 1, Dev Accuracy: 0.5580
Epoch 2, Dev Accuracy: 0.6640
Epoch 3, Dev Accuracy: 0.7240
Epoch 4, Dev Accuracy: 0.7380
Epoch 5, Dev Accuracy: 0.7560
Epoch 6, Dev Accuracy: 0.7700
Epoch 7, Dev Accuracy: 0.7660
Epoch 8, Dev Accuracy: 0.7760
Epoch 9, Dev Accuracy: 0.7780
Epoch 10, Dev Accuracy: 0.7720
Epoch 11, Dev Accuracy: 0.7760
Epoch 12, Dev Accuracy: 0.8020
Epoch 13, Dev Accuracy: 0.8020
Epoch 14, Dev Accuracy: 0.7800
Epoch 15, Dev Accuracy: 0.8080
Epoch 16, Dev Accuracy: 0.8140
Epoch 17, Dev Accuracy: 0.8120
Epoch 18, Dev Accuracy: 0.8040
Epoch 19, Dev Accuracy: 0.8100
Early stopping
Test Accuracy: 0.8540
Time taken: 5.030166149139404
```