hw2_111511198

📌 作者: 111511198 鄭恆安

Part 0:

In this part, we implemented a method for text preprocessing.

Method

1. remove stopwards

   > remove words such as ‘a’, ‘I’, ‘not’, etc.

2. lemmatize sentence

   > turn same verb such as [make, making, makes, made]→make

3. Removing html tags

   > remove <h5/></h5>

4. Remove punctuations and numbers
5. Removing multiple spaces

Code

```
def preprocessing_function(text: str) -> str:
    import re
    from nltk.stem import WordNetLemmatizer
    #lemmatize sentence
    lm = WordNetLemmatizer()
    sentence = lm.lemmatize(text)

    # Removing html tags
    TAG_RE = re.compile(r'<[^>]+>')
    sentence = TAG_RE.sub('', sentence)

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Removing multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence)
    print(sentence)
    return sentence
'''
This function takes a text as input and removes the stopwords from it using the NLTK l
ibrary, which is a popular Python library for natural language processing.
'''
```

Result

👉 original = “Robert DeNiro plays the most unbelievably intelligent illiterate of all time.”
→preprocessed = “Robert DeNiro plays unbelievably intelligent illiterate time”

Part 1:

Perplexity

Perplexity is a measure of how well a language model is able to predict the next word in a sequence of words. It is often used to evaluate the performance of language models. A lower perplexity value indicates that the model is better at predicting the next word.

Code - compute_perplexity

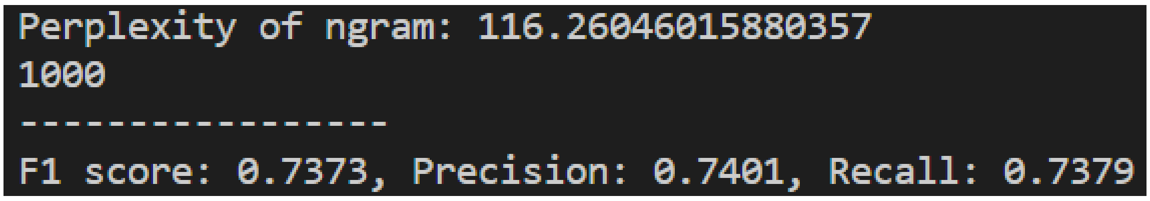
```
def compute_perplexity(self, df_test) -> float:
        '''
        Compute the perplexity of n-gram model.
        Perplexity = 2^(-entropy)
        '''
        if self.model is None:
            raise NotImplementedError("Train your model first")

        corpus = [['[CLS]'] + self.tokenize(document) for document in df_test['review']]
        perplexity = 0
        for document_tokenize in corpus:
            twograms = nltk.ngrams(document_tokenize, self.n)
            N = len(list(nltk.ngrams(document_tokenize, self.n)))
            probabilities = []
            for w1, w2 in twograms:
                numerator = 1 + self.model[w1][w2]
                denominator = sum(self.model[w1].values())
                # give a value to avoid divide-by-zero
                if denominator == 0:
                    probabilities.append(1e-3)
                else:
                    probabilities.append(numerator / denominator)

            cross_entropy = -1 / N * sum([math.log(p, 2) for p in probabilities])
            perplexity += math.pow(2, cross_entropy)

        perplexity /= len(corpus)
        return perplexity
```

Perplexity Comparison

We compared the perplexity values of three different preprocessing methods: without any preprocessing, with stopwords removed, and with our method of preprocessing.

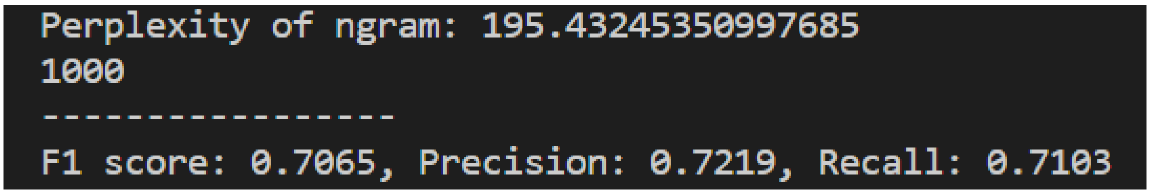
Test F1-Score for bi-gram model

- **without any preprocessing**



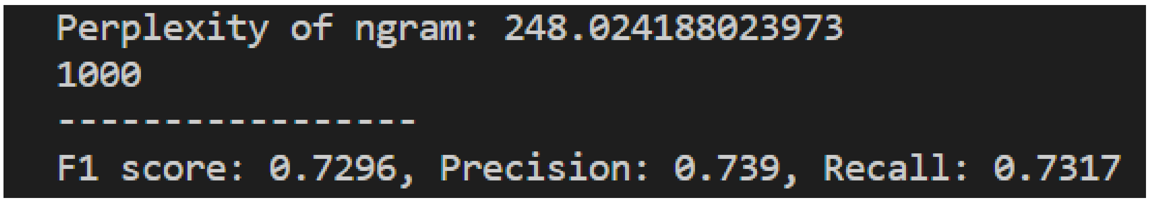
```
Perplexity of ngram: 116.26046015880357
1000
-----------------
F1 score: 0.7373, Precision: 0.7401, Recall: 0.7379
```

- **with stopwords removed**



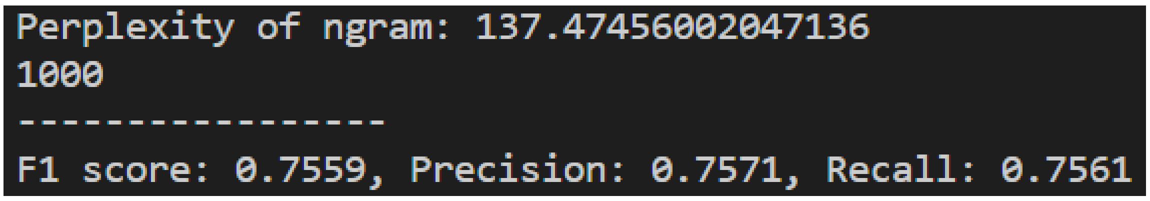
```
Perplexity of ngram: 195.43245350997685
1000
-----------------
F1 score: 0.7065, Precision: 0.7219, Recall: 0.7103
```

- **with our method of preprocessing**



```
Perplexity of ngram: 248.024188023973
1000
-----------------
F1 score: 0.7296, Precision: 0.739, Recall: 0.7317
```

- **with our method of preprocessing (without stopwords removed)**



```
Perplexity of ngram: 137.47456002047136
1000
-----------------
F1 score: 0.7559, Precision: 0.7571, Recall: 0.7561
```

Observations:

- The perplexity is **lowest** for the text **without any preprocessing**, indicating that maybe in these reviews, **stopwards such an “not” or other words play an important role in classifying sentiment.**
- The perplexity is higher for the text **with our method of preprocessing** , but also the higher F1 score than **with only stopwords removed,** suggesting that removing stopwords can help in improving the language model's ability to predict the next word.
- Therefore, I train the last model **with our method of preprocessing but excluding remove stopwards.** As you can see, it preforms **better than other two prerocessed method** and it’s **F1 score is the highest.**

Code - train_sentiment

```
def train_sentiment(self, df_train, df_test):
        # step 1. select the most feature_num patterns as features, you can adjust featur
        feature_num = 1000
        print(feature_num)
        features = self.features.most_common(feature_num)
        id, tmp = {}, 0
        for i, j in features:
            id[i] = tmp
            tmp += 1
        # step 2. convert each sentence in both training data and testing data to embeddi
        train_corpus_embedding = [[0 for i in range(feature_num)] for i in range(len(df_t
        test_corpus_embedding = [[0 for i in range(feature_num)] for i in range(len(df_te
        for i in range(len(df_train['review'])):
            text = ['[CLS]']+self.tokenize(df_train['review'][i])
            pre = ""
            for word in text:
                if pre != "" and id.get((pre, word)) is not None:
                    train_corpus_embedding[i][id[(pre, word)]] += 1
                pre = word
        for i in range(len(df_test['review'])):
            text = ['[CLS]']+self.tokenize(df_test['review'][i])
            pre = ""
            for word in text:
                if pre != "" and id.get((pre, word)) is not None:
                    test_corpus_embedding[i][id[(pre, word)]] += 1
                pre = word
        # feed converted embeddings to Naive Bayes
        nb_model = GaussianNB()
        nb_model.fit(train_corpus_embedding, df_train['sentiment'])
        y_predicted = nb_model.predict(test_corpus_embedding)
        precision, recall, f1, support = precision_recall_fscore_support(df_test['sentime
        precision = round(precision, 4)
        recall = round(recall, 4)
        f1 = round(f1, 4)
        print(f"F1 score: {f1}, Precision: {precision}, Recall: {recall}")
```

Part 2:

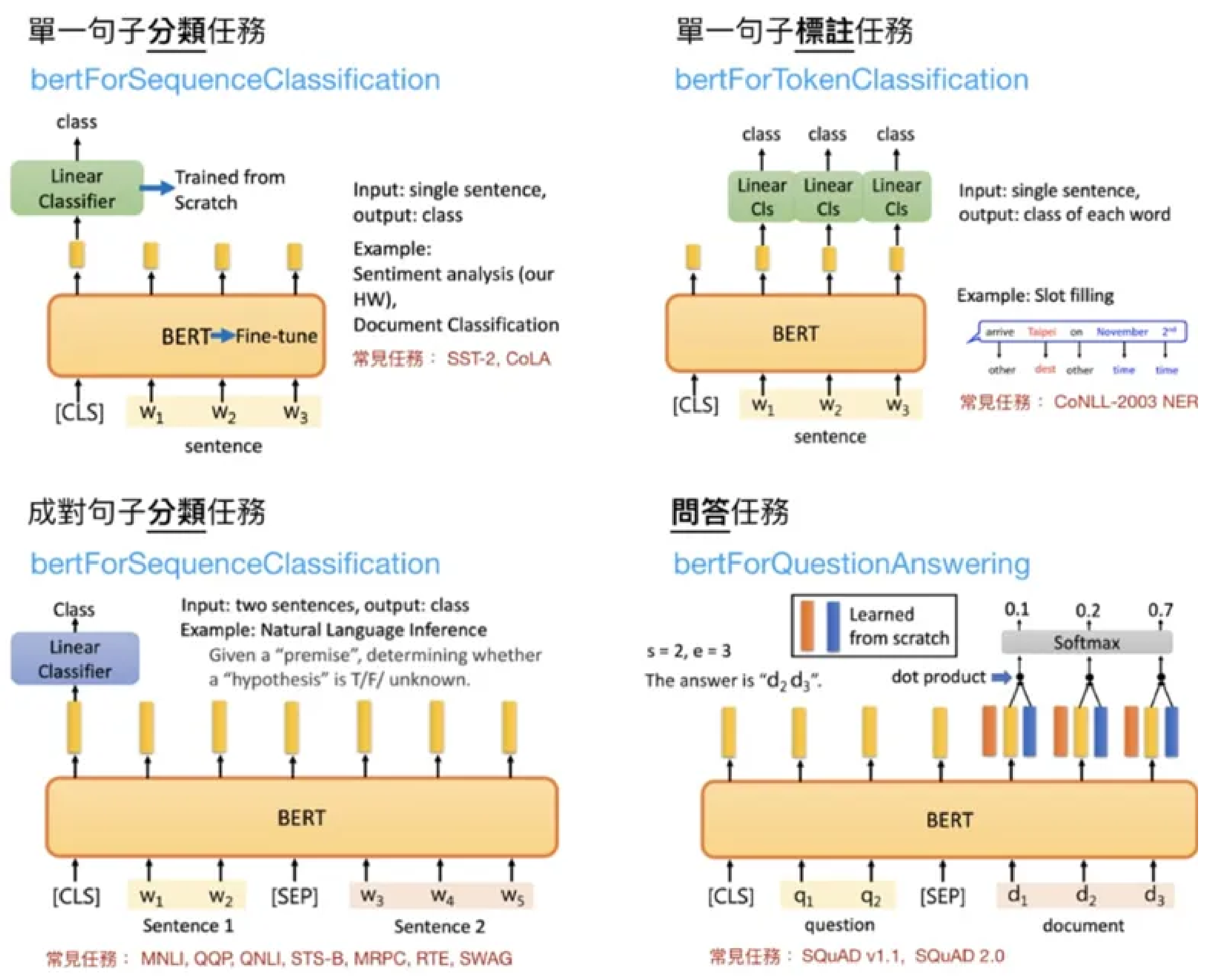
BERT and DistilBERT

In this part, we discussed the concept of pre-training in BERT, which involves training a language model on a large corpus of text data. The two main pre-training steps in BERT are:

1. Masked Language Model (MLM): In this step, random words are masked in the input text, and the model is trained to predict the masked words based on the context.

2. Next Sentence Prediction (NSP): In this step, pairs of sentences are created, and the model is trained to predict if the second sentence is the actual next sentence in the sequence.

Four Different BERT Application Scenarios



單一句子分類任務
bertForSequenceClassification
class
Linear Classifier
Trained from Scratch
BERT→Fine-tune
[CLS] w1 w2 w3
sentence
Input: single sentence, output: class
Example: Sentiment analysis (our HW), Document Classification
常見任務： SST-2, CoLA
單一句子標註任務
bertForTokenClassification
class class class
Linear Cls Linear Cls Linear Cls
BERT
[CLS] w1 w2 w3
sentence
Input: single sentence, output: class of each word
Example: Slot filling
arrive Taipei on November 2nd
other dest other time time
常見任務： CoNLL-2003 NER
成對句子分類任務
bertForSequenceClassification
Class
Linear Classifier
Input: two sentences, output: class
Example: Natural Language Inference
Given a "premise", determining whether a "hypothesis" is T/F/ unknown.
BERT
[CLS] w1 w2 [SEP] w3 w4 w5
Sentence 1
Sentence 2
常見任務： MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG
問答任務
bertForQuestionAnswering
Learned from scratch
s = 2, e = 3
The answer is "d2 d3".
dot product
0.1 0.2 0.7
Softmax
BERT
[CLS] q1 q2 [SEP] d1 d2 d3
question
document
常見任務： SQuAD v1.1, SQuAD 2.0

Four different BERT application scenarios are:

1. **Fine-tuning for specific tasks**: BERT can be fine-tuned on a smaller dataset for specific tasks such as text classification, named entity recognition, etc. **And it's the model we use in these project.**
2. **Feature extraction**: BERT can be used as a feature extraction tool to obtain contextualized word embeddings that can be used as input features for downstream tasks.
3. **Sentence similarity**: BERT can be used to measure the similarity between two sentences, which can be useful in applications such as question answering and information retrieval.
4. **Text generation**: BERT can be used to generate text by conditioning the model on a given input and sampling from the predicted distributions of the next word.

Difference Between BERT and DistilBERT

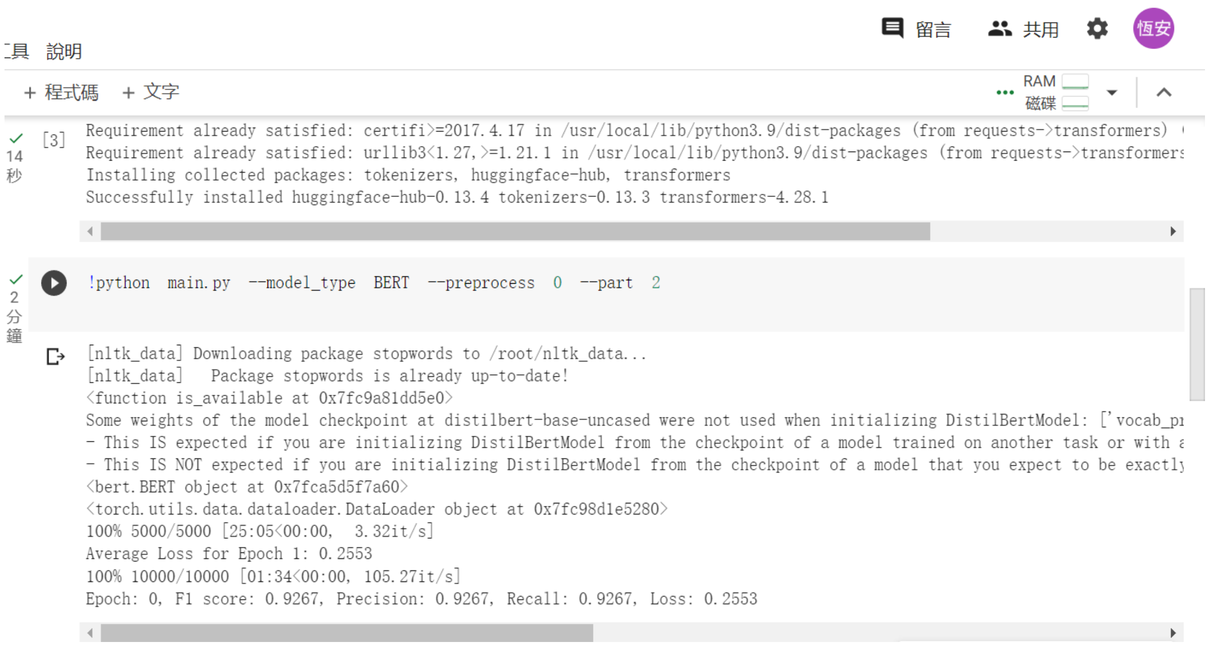
The difference between BERT and DistilBERT is that DistilBERT is a smaller and faster version of BERT, achieved by distilling the knowledge of the original BERT model into a smaller model during training. DistilBERT is designed to be more computationally efficient and suitable for deployment on resource-constrained environments.

Code - BERT Classifier

```
class BERT_IMDB(nn.Module):
    '''
    Fine-tuning DistillBert with two MLPs.
    '''

    def __init__(self, pretrained_type):
        super().__init__()

        num_labels = 2
        self.pretrained_model = AutoModel.from_pretrained(
            pretrained_type, num_labels=num_labels)

        # TO-DO 2-1: Construct a classifier
        # BEGIN YOUR CODE
        self.classifier= nn.Linear(self.pretrained_model.config.hidden_size, num_labels)
        #END YOUR CODE
    def forward(self, **pretrained_text):
        outputs = self.pretrained_model(**pretrained_text).last_hidden_state
        pretrained_output= outputs[:, 0, :] # Extract the first token's hidden state
        logits = self.classifier(pretrained_output)
        return logits
```

Test F1-Score for BERT Model

Here is a screenshot of the test F1-score for our BERT model:



```
[3] Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->transformers)
    Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->transformers
    Installing collected packages: tokenizers, huggingface-hub, transformers
    Successfully installed huggingface-hub-0.13.4 tokenizers-0.13.3 transformers-4.28.1
```

```
!python main.py --model_type BERT --preprocess 0 --part 2
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
<function is_available at 0x7fc9a81dd5e0>
Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing DistilBertModel: ['vocab_pr
- This IS expected if you are initializing DistilBertModel from the checkpoint of a model trained on another task or with a
- This IS NOT expected if you are initializing DistilBertModel from the checkpoint of a model that you expect to be exactly
<bert.BERT object at 0x7fca5d5f7a60>
<torch.utils.data.dataloader.DataLoader object at 0x7fc98d1e5280>
100% 5000/5000 [25:05<00:00,  3.32it/s]
Average Loss for Epoch 1: 0.2553
100% 10000/10000 [01:34<00:00, 105.27it/s]
Epoch: 0, F1 score: 0.9267, Precision: 0.9267, Recall: 0.9267, Loss: 0.2553
```

Observation:

- **The test F1-score for our BERT model is 0.9267**, indicating good performance on the test dataset.

Transformer and BERT

The Transformer is a type of neural network architecture that was introduced in the "Attention is All You Need" paper by Vaswani et al. It is a self-attention-based model that can process sequences of variable length without the need for recurrence or convolution.

BERT, on the other hand, is a specific implementation of the Transformer architecture for language modeling tasks. It includes the Masked Language Model (MLM) and Next Sentence Prediction (NSP) pre-training tasks, which allow the model to learn contextualized word embeddings.

The core of the Transformer architecture is the self-attention mechanism, which allows the model to weigh the importance of different words in the input sequence when making predictions. This allows the model to capture long-range dependencies and relationships between words in the text.

Part 3:

Difference Between Vanilla RNN and LSTM

Vanilla RNN (Recurrent Neural Network) is a type of neural network that is used for processing sequential data, but it suffers from the "vanishing gradient" problem, which makes it difficult for the model to capture long-term dependencies in the data. LSTM (Long Short-Term Memory), on the other hand, is a type of recurrent neural network that addresses the vanishing gradient problem by using a more complex architecture with memory cells and gates that can store and propagate information over longer sequences.

Meaning of Dimensions in LSTM Model

The LSTM model has several input and output dimensions, each of which has a specific meaning:

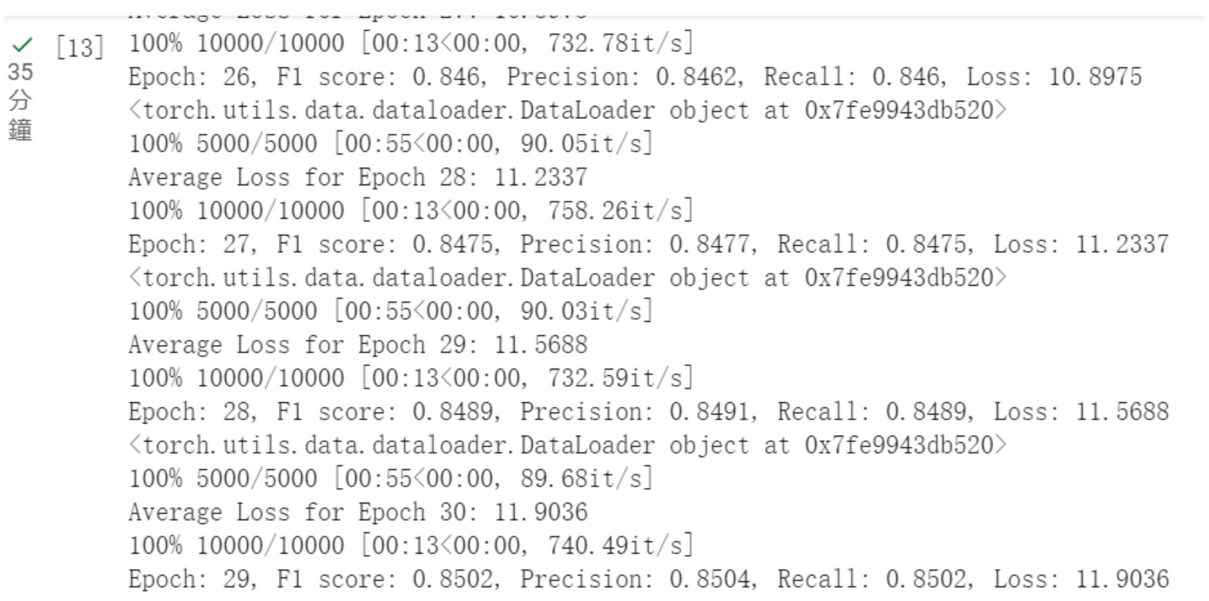
- First Dimension (Batch Size): It represents the number of input sequences processed in a single batch during training or inference.
- Second Dimension (Sequence Length): It represents the length of the input sequences, which can vary depending on the input data.
- Third Dimension (Input Features): It represents the number of input features or dimensions in each input sequence.
- Output Dimension: It represents the number of output units or neurons in the LSTM layer, which determines the size of the output tensor.

Code - LSTM Model

```
class YourModel(nn.Module):
    def __init__(
            self,
            # TO-DO 3-1: Pass parameters to initialize model
            # BEGIN YOUR CODE
            vocab_size,
            embedding_dim,
            hidden_dim,
            num_layers,
            dropout,
            output_dim
            # END YOUR CODE
        ):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.dropout1 = nn.Dropout(dropout)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers)
        self.dense1 = nn.Linear(hidden_dim, 128)
        self.dropout2 = nn.Dropout(dropout)
        self.dense3 = nn.Linear(128, output_dim)
        # END YOUR CODE

    def forward(self, text):
        '''
        In the forward() function, we decide which operations the input will undergo to g
        For example, in a sentiment classification model, the input usually goes through 
        Embedding() -> RNN() -> Linear() in sequence to obtain the final output.
        '''
        # TO-DO 3-3: Determine how the model generates output based on input
        # BEGIN YOUR CODE
        x = self.embedding(text)
        x = self.dropout1(x)
        x, _ = self.lstm(x)
        x = x[-1,:, :]  # Get the last output of LSTM
        x = nn.functional.relu(self.dense1(x))
        x = self.dropout2(x)
        output = nn.functional.sigmoid(self.dense3(x))
        return output
        # END YOUR CODE

class RNN(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.model = YourModel(
            # BEGIN YOUR CODE
            vocab_size=config['vocab_size'],
            embedding_dim=config['embedding_dim'],
            hidden_dim=config['hidden_dim'],
            num_layers=config['num_layers'],
            dropout=config['dropout'],
            output_dim=config['output_dim']
            # END YOUR CODE
        ).to(config['device'])

    def forward(self, text):
        return self.model(text)
```

Test F1-Score for LSTM Model

Here is a screenshot of the test F1-score for our LSTM model:



```
✓  [13]  100% 10000/10000 [00:13<00:00, 732.78it/s]
35       Epoch: 26, F1 score: 0.846, Precision: 0.8462, Recall: 0.846, Loss: 10.8975
分       <torch.utils.data.dataloader.DataLoader object at 0x7fe9943db520>
鐘       100% 5000/5000 [00:55<00:00, 90.05it/s]
         Average Loss for Epoch 28: 11.2337
         100% 10000/10000 [00:13<00:00, 758.26it/s]
         Epoch: 27, F1 score: 0.8475, Precision: 0.8477, Recall: 0.8475, Loss: 11.2337
         <torch.utils.data.dataloader.DataLoader object at 0x7fe9943db520>
         100% 5000/5000 [00:55<00:00, 90.03it/s]
         Average Loss for Epoch 29: 11.5688
         100% 10000/10000 [00:13<00:00, 732.59it/s]
         Epoch: 28, F1 score: 0.8489, Precision: 0.8491, Recall: 0.8489, Loss: 11.5688
         <torch.utils.data.dataloader.DataLoader object at 0x7fe9943db520>
         100% 5000/5000 [00:55<00:00, 89.68it/s]
         Average Loss for Epoch 30: 11.9036
         100% 10000/10000 [00:13<00:00, 740.49it/s]
         Epoch: 29, F1 score: 0.8502, Precision: 0.8504, Recall: 0.8502, Loss: 11.9036
```

Observation:

- The test F1-score for our LSTM model is 0.85, indicating decent performance on the test dataset.

Discussion:

Innovation in NLP Field and Evolution of Techniques

The field of natural language processing (NLP) has witnessed significant innovation and evolution over the years. The techniques have evolved from simple n-gram models to more complex LSTM-based models, and now to advanced models like BERT.

The key reasons for this evolution are:

1. Better Representation Learning
    a. N-gram models only rely on simple statistical methods that do not capture the semantic meaning of words or the context in which they appear.
    b. LSTM-based models, on the other hand, can learn more meaningful representations of words and capture long-term dependencies in the data, since it has memory and forget gate.
    c. BERT, with its self-attention mechanism, can further capture contextualized word embeddings, leading to better representation learning.

   d. **In conclusion, as we can see these three model in this HW, the ranking of F1 score are truly Bert Model > LSTM Model > N-gram Model**

2. Advances in Deep Learning: The field of deep learning has witnessed rapid advancements in recent years, including the introduction of advanced architectures like the Transformer and self-attention mechanisms. These advancements have enabled the development of more powerful and efficient models like BERT.
3. Availability of Large Datasets and Computing Resources: With the availability of large datasets and increased computing resources, it has become possible to train complex models like BERT on massive amounts of data, leading to improved performance.
4. Application-Specific Requirements: Different NLP tasks require different levels of model complexity

Problems I have encountered

1. Part 1
   a. I discovered that no matter which feature_num I select the F1 score are always the same?
      i. it turns out that my code for train_corpus_embedding mulfunction, which leads to always 0 in train_corpus_embedding
      ii. So I change these code and finally make a progress on F1 score
   b. performance are still low when i directly use text for df['review']
      i. So i change text = ['[CLS]']+self.tokenize(df_train['review'][i]) to be tokenize list
2. Part 2
   a. I can only make my F1 score to 0.9267 no matter what Classifier I create
      i. **I still can't conquer it**
3. Part 3
   a. I can't find any ideas at the beginning, especially the Model Bert?

    i. Later, I finally discover that it is a class defined in bert.py

b. The performance is poor (F1 score=0.6)
    i. make my model more complex by adding more layers
    ii. increase epochs from 10 to 30
    iii. And F1 score finally reach 0.85