Intro. to Artificial Intelligence

- HW2

109350008 張詠哲

Part 0

```python
def preprocessing_function(text: str) -> str:
    preprocessed_text = remove_stopwords(text)

    # TO-DO 0: Other preprocessing function attemption
    # Begin your code
    preprocessed_text = remove_HTML(preprocessed_text)
    preprocessed_text = remove_symbol(preprocessed_text)
    preprocessed_text = lemmatization(preprocessed_text)
    # End your code

    return preprocessed_text
```

在這裡我有採取的 preprocessing 方式有 remove stopword、remove HTML、remove symbol、lemmatization。

- **remove stopword**:

```python
def remove_stopwords(text: str) -> str:
    '''
    E.g.,
        text: 'Here is a dog.'
        preprocessed_text: 'Here dog.'
    '''
    stop_word_list = stopwords.words('english')
    tokenizer = ToktokTokenizer()
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    filtered_tokens = [token for token in tokens if token.lower() not in stop_word_list]
    preprocessed_text = ' '.join(filtered_tokens)

    return preprocessed_text
```

先把 input 做 tokenize(變成詞)之後再把一些介係詞、代名詞、定冠詞(例如: the、who、it、he)這些資訊量不大的詞去掉

Ex. “let me do it for you”(input) -> “let”

- **remove HTML** :

```python
import re
```

```python
def remove_HTML(text: str) -> str:
    rmHTML = re.sub("<[^>]+>", "", text).strip()

    return rmHTML
```

把 HTML 的格式符號刪掉，像是這次作業 dataset 中的
</br>。這裡有另外 import re，可以用來

- **remove symbol:**

```
from nltk.tokenize import RegexpTokenizer
```

```
def remove_symbol(text: str) -> str:
    filter = RegexpTokenizer(r'\w+', gaps = False)
    rmsym = filter.tokenize(text)
    preprocessed_text = ' '.join(rmsym)

    return preprocessed_text
```

把標點符號刪掉，這裡有另外 import RegexpTokenizer。Ex. Hello!! -> Hello

- **lemmatization:**

```
from nltk.stem import WordNetLemmatizer
```

```
def lemmatization(text: str) -> str:
    filter = nltk.stem.wordnet.WordNetLemmatizer()
    text = text.split(" ")
    lemed = []
    for i in text:
        lem = filter.lemmatize(i, "n")
        if(lem == i): lem = filter.lemmatize(i, "v")
        lemed.append(lem)

    preprocessed_text = ' '.join(lemed)

    return preprocessed_text
```

把詞性還原，這裡有另外 Import WordNetLemmatizer，且因為還要輸入詞性，因此用了 for 迴圈去判斷每個句子中的每個字，一開始先用名詞判定，若是變動完和原本的一樣則改用動詞。

Ex. happened -> happen

Part 1

1-1

```
for s in corpus_tokenize:
    for i in range(len(s)-1):
        pair = (s[i], s[i+1])
        model[pair[0]][pair[1]] += 1 #每一個字後面所接的各個字的次數
        features[pair] += 1 #每一種features的出現次數

return model, features
```

計算讀進來的 token 後面所接的詞的次數(model)，以及每個詞一起出現的次數(feature)

1-2

```
# step 1. select the most feature_num patterns as features, you can adjust fea
feature_num = 1900

# step 2. convert each sentence in both training data and testing data to embe
# Note that you should name "train_corpus_embedding" and "test_corpus_embeddi
m, f = self.model, self.features
selected = sorted(f.items(), key=lambda x: x[1], reverse=True)[:feature_num]
f_idx = {feature: i for i, (feature, occur_time) in enumerate(selected)} # inc

train_corpus_embedding = []
for sentence in df_train['review']:
    tokens = self.tokenize(sentence)
    embedding = [0] * feature_num

    for i in range(len(tokens) - self.n + 1):
        ngram = tuple(tokens[i:i+self.n])
        if ngram in f_idx:
            embedding[f_idx[ngram]] += 1

    train_corpus_embedding.append(embedding)

test_corpus_embedding = []
for sentence in df_test['review']:
    tokens = self.tokenize(sentence)
    embedding = [0] * feature_num

    for i in range(len(tokens) - self.n + 1):
        ngram = tuple(tokens[i:i+self.n])
        if ngram in f_idx:
            embedding[f_idx[ngram]] += 1

    test_corpus_embedding.append(embedding)
```

計算前 feature_num 的出現頻率的 feature 並編號，接著去計算 input 中有那些地方有出現，並記錄出現次數且記錄進 embedding

Q&A

Q1 Briefly explain the method you implemented and give an example

perplexity 是一種用於評估 NLP 的指標。它表示模型對一個句子的預測能力有多強，越低的 perplexity 表示模型預測得越好。其會受多種因素影響，例如 Corpus 的大小以及品質、NLP 的複雜度、訓練數據的量和質量等。更大更高質量的 Corpus 和更複雜的模型或是更多的訓練數據能夠降低 perplexity。

Q2 Screenshot the outputs and tell your observations about the differences in the perplexity caused by the preprocessing methods(1. Without preprocess 2. with remove stopwords 3. with your method)

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

[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
Perplexity of ngram: 116.26046015880357
F1 score: 0.7934, Precision: 0.7956, Recall: 0.7937
```

```
!python  main.py  --model_type  ngram  --preprocess  1  --part  2

[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
Perplexity of ngram: 267.7001901990911
100% 40000/40000 [00:15<00:00, 2633.63it/s]
100% 10000/10000 [00:03<00:00, 3184.70it/s]
F1 score: 0.777, Precision: 0.7843, Recall: 0.7782
```

利用 train 第一筆數據去計算 perplexity 並比較，以下是結果

```python
import pandas as pd
import preprocess
model = Ngram(2)
df_train = pd.read_csv('./data/IMDB_train.csv')
test = {'review': [df_train.iloc[0,0]]}
st = {'review': [preprocess.remove_stopwords(df_train.iloc[0,0])]}
my_pro = {'review': [preprocess.preprocessing_function(df_train.iloc[0,0])]}

model.train(df_train)
print("without pro: ", model.compute_perplexity(test))
print("only stopword: ", model.compute_perplexity(st))
print("my pro: ", model.compute_perplexity(my_pro))
```

```
without pro:  113.55684700706816
only stopword:  246.897552620742
my pro:  295.84728466039167
```

由結果可看出 perplexity 越來越大，我猜測應該是有點過度去除標點符號、stopword，以至於破壞了原始文本中的結構，導致 perplexity 上升。

Part 2

2-1

```
# TO-DO 2-1: Construct a classifier
# BEGIN YOUR CODE
hidden_size = self.pretrained_model.config.hidden_size
self.classifier = nn.Sequential(
    nn.Linear(hidden_size, hidden_size),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(hidden_size, hidden_size),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(hidden_size, hidden_size),
    nn.ReLU(),
    nn.Dropout(0.1),
    nn.Linear(hidden_size, num_labels),
)
# END YOUR CODE
```

2-2

```
if model_type == "BERT":
    for epoch in range(config['epochs']):
        total_loss = 0
        labels, pred = [], []

        # training stage
        model.train()
        for batch in tqdm(train_dataloader):
            inputs, targets = batch
            inputs = inputs.to(config['device'])
            targets = targets.to(config['device'])

            optimizer.zero_grad()
            outputs = model.forward(inputs)
            outputs = outputs.to(config['device'])
            loss = loss_fn(outputs, targets)

            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        # testing stage
        with torch.no_grad():
            model.eval()
            for batch in tqdm(test_dataloader):
                inputs, targets = batch
                inputs = inputs.to(config['device'])
                targets = targets.to(config['device'])
                outputs = model.forward(inputs)
                outputs = outputs.to(config['device'])
                predicted = torch.argmax(outputs, dim=1)

                labels += targets.cpu().numpy().tolist()
                pred += predicted.cpu().numpy().tolist()
```

依照 pytorch 官網的教學，若輸入的是 BERT 的話就近 BERT 的訓練以及評估。訓練部分是先處理好 input 以及 target 並把他們放進 GPU 裡面。接著清空 optimizer 後把 input 丟進 model 裡面訓練。接著計算 output 和 target 的 loss 再做 backward 以及更新參數(optimizer.step())。最後把 test 丟進訓練好的 model 評估。

Q&A

Q1 Briefly explain the two pre-training steps in BERT.

BERT 的兩個 pretrain 步驟是：Masked Language Model（MLM）和 Next Sentence Prediction（NSP）。

- **MLM**：

  對輸入的文本進行隨機 masking，將部分單詞替換為 [MASK] 標記。然後模型需要通過上下文來預測被遮住的單詞是什麼。

- **NSP:**

  判斷第 2 個句子在原始文本中是否跟第 1 個句子相接。

Q2 Briefly explain four different BERT application scenarios

以下是四種不同的 BERT 使用情境：

- 文本分類：

  可將文章分類為正面或負面評論，或將新聞分類為體育、財經、政治等類別。

- 問答系統：

  可用於回答問題和解決常見問題。模型將問題和文本段落作為輸入，並生成答案或指示。也可以用於自動摘要，從長文本中提取重要信息生成摘要。

- 自然語言生成：

  可用於生成文本，例如電子郵件、新聞報導、故事等。

- 文本匹配：

  可用於文本匹配任務，例如將兩個句子進行比較，判斷它們是否相似。常見於搜索引擎和推薦系統中。

Q3 Discuss the difference between BERT and distilBERT

BERT 是一個較大的模型，有很多層(包括 340 兆個參數)。而 DistilBERT 是在 BERT 基礎上進行了簡化(只有 66 兆個參數)，但是所需要的計算資源更少。而在訓練方式也不同，例如 BERT 的訓練過程中包含 MLM 和 NSP。而 DistilBERT 只訓練了 MLM。因此，DistilBERT 的預訓練時間比 BERT 短，同時也較容易實現，但也保持了不錯的表現也提升了模型速度。。

Q4 Screenshot the required test F1-score.

```
!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!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
Some weights of the model checkpoint at distilbert-base-uncased were not used w
- This IS expected if you are initializing DistilBertModel from the checkpoint
- This IS NOT expected if you are initializing DistilBertModel from the checkpo
100% 5000/5000 [30:51<00:00,  2.70it/s]
100% 10000/10000 [01:51<00:00, 89.90it/s]
Epoch: 0, F1 score: 0.9331, Precision: 0.9332, Recall: 0.9331, Loss: 0.233
```

```
#whole  preprocess
!python  main.py  --model_type  BERT  --preprocess  1  --part  2
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
Some weights of the model checkpoint at distilbert-base-uncased were not used w
- This IS expected if you are initializing DistilBertModel from the checkpoint
- This IS NOT expected if you are initializing DistilBertModel from the checkpo
100% 5000/5000 [20:07<00:00,  4.14it/s]
100% 10000/10000 [01:16<00:00, 130.66it/s]
Epoch: 0, F1 score: 0.9129, Precision: 0.9134, Recall: 0.9129, Loss: 0.2856
```

```
#stopword  only
!python  main.py  --model_type  BERT  --preprocess  1  --part  2
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 7.26kB/s]
Downloading (…)lve/main/config.json: 100% 483/483 [00:00<00:00, 123kB/s]
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 548kB/s]
Downloading (…)/main/tokenizer.json: 100% 466k/466k [00:00<00:00, 43.5MB/s]
Downloading pytorch_model.bin: 100% 268M/268M [00:01<00:00, 190MB/s]
Some weights of the model checkpoint at distilbert-base-uncased were not used
- This IS expected if you are initializing DistilBertModel from the checkpoint
- This IS NOT expected if you are initializing DistilBertModel from the checkp
100% 5000/5000 [26:15<00:00,  3.17it/s]
100% 10000/10000 [01:37<00:00, 102.90it/s]
Epoch: 0, F1 score: 0.9233, Precision: 0.9234, Recall: 0.9233, Loss: 0.272
```

(Stopword only 是把 preprocessing.py 中 Part 0 實作的方法先註解掉，只留下 remove_stopword 再去跑一次得到)

BONUS Explain the relation of the Transformer and BERT and the core of the Transformer

而 BERT 則是一種基於 Transformer 模型的語言模型，其核心是 Transformer 的 Encoder 部分，它能夠將 input 中的每個詞彙進行向量化，並且同時捕捉到它

們之間的語義關係。這樣，BERT 就能夠進行下游任務（如文本分類、情感分析等）。

Transformer 的核心概念是自注意力機制（Self-Attention Mechanism）。Self-Attention Mechanism 可以讓模型專注於 input 中不同位置之間的相互作用關係。其每個詞彙都會計算一組注意力分數，以指示與其他詞彙之間的關係。這樣可以在不使用 RNN 或 CNN 的情況下，對 input 建模。

Part 3

3-1

```
# TO-DO 3-1: Pass parameters to initialize model
self.config = config
self.model = YourModel(
    # BEGIN YOUR CODE
    vocab_size = config['vocab_size'],
    embedding_dim = 400,
    hidden_dim = 256,
    n_layers = 2,
    output_dim = 2,
    dropout= 0.5,
    bidirectional = True
    # END YOUR CODE
).to(config['device'])
```

3-2

```
# TO-DO 3-2: Determine which modules your model should consist of
# BEGIN YOUR CODE
self.embedding = nn.Embedding(vocab_size, embedding_dim)
self.encoder = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=n_layers,
            dropout=dropout,
            bidirectional = bidirectional
            )

self.decoder = nn.Linear(4 * hidden_dim, output_dim)
# END YOUR CODE
```

3-3

```python
# TO-DO 3-3: Determine how the model generates output based on input
# BEGIN YOUR CODE

# text: (seq_len, batch_size)
embedding = self.embedding(text) #embedding: (seq, batch, emb_dim)
outputs, _ = self.encoder(embedding) #output: (seq, batch, 4*hid_dim)
encoding = torch.cat((outputs[0], outputs[-1]), dim = -1)

logits = self.decoder(encoding)
return logits
# END YOUR CODE
```

Q&A

Q1 Briefly explain the difference between vanilla RNN and LSTM.

Vanilla RNN 只有一個隱藏層，將當前的輸入與前一個時間步的隱藏層狀態作為輸入，並產生新的隱藏層狀態和輸出。但由於長期依賴問題，當序列長度變長時，Vanilla RNN 很難學習到長期的依賴關係，因此對於長序列的處理效果不佳。

而 LSTM 使用了三個單元：forget gate、input gate、output gate，更好地處理長期依賴問題。Forget gate 可以控制過去的記憶是否被遺忘，input gate 可以控制當前的輸入對隱藏層狀態的影響，而 output 可以控制生成的輸出。

Q2 Please explain the meaning of each dimension of the input and output for each layer in the model. For example, the first dimension of input for LSTM is batch size.

	Input	Output
Embedding	**(sequence_length, batch_size)**	**(sequence_length, batch_size, embedding_dim)**
Encoder(LSTM)	**(sequence_length, batch_size, embedding_dim)**	**(sequence_length, batch_size, 4*hidden_dim)**
Decoder	**(batch_size, 4*hidden_dim)**	**(batch_size, output_dim)**

Q3 Screenshot the required test F1-score

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

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
100% 5000/5000 [06:06<00:00, 13.63it/s]
100% 10000/10000 [01:40<00:00, 99.78it/s]
Epoch: 0, F1 score: 0.8812, Precision: 0.8818, Recall: 0.8812, Loss: 0.5375
```

```
#whole  preprocess
!python  main.py  --model_type  RNN  --preprocess  1  --part  2
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
100% 5000/5000 [05:16<00:00, 15.80it/s]
100% 10000/10000 [01:10<00:00, 141.49it/s]
Epoch: 0, F1 score: 0.8339, Precision: 0.8588, Recall: 0.8365, Loss: 0.4742
```

```
#stopword  only
!python  main.py  --model_type  RNN  --preprocess  1  --part  2
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
100% 5000/5000 [03:49<00:00, 21.82it/s]
100% 10000/10000 [00:56<00:00, 176.74it/s]
Epoch: 0, F1 score: 0.8703, Precision: 0.8705, Recall: 0.8703, Loss: 0.4704
```

(Stopword only 是把 preprocessing.py 中 Part 0 實作的方法先註解掉，只留下 remove_stopword 再去跑一次得到)

Discussion

Q1 Discuss the innovation of the NLP field and your thoughts of why the technique is evolving from ngram -> LSTM -> BERT.

ngram 利用了條件機率去計算下一個最高機率出現的詞。可以捕捉短語和詞的信息。但是，它無法處理長距離依賴關係。LSTM 引入了 RNN 的方式以及 forget gate 和 input gate、output gate，使其能夠處理長距離依賴關係。BERT 則是一種基於 Transformer 的模型，通過在大量文本數據上進行預訓練來學習通用語言表示，並在下游任務中進行微調來達成我們想要執行的事情。有更好的表徵能力和更好的泛化能力，可以處理更複雜的自然語言處理任務。

我覺得 NLP 技術的發展從 ngram 到 LSTM 再到 BERT，是由於多種因素的推動。像是 CPU 以及 GPU 性能的提高，可以處理更大和更複雜的數據集，不斷改進算法使得神經網絡的訓練和運行變得更加高效，Attention 的發明使得後面有 Transformer。最重要的數據集越來越大，使得可以更好地訓練和測試模型。

Q2 Describe problems you meet and how you solve them.

一開始要訓練 BERT 的時候 training 時間一直很長，但之後發現是 colab 的 GPU 沒開。還有在跑 RNN 的時候結果一直很差，大約都落在 30~50%，但之後改成雙向 LSTM 後訓練結果就上升到 87%，好耶。但是 BERT 不管用多少層的 linear 或是 drop，我的 test F1-score 都一直卡在 92~93%左右一直上不去。最後是不知道為甚麼沒有做 preprocessing 的結果一直都比有做的還要高，我認為有可能是因為是有點過度去除標點符號、stopword 或是一些有利於判斷的文字，以至於破壞了原始文本中的結構，導致準度下降。