hw2_report_109550017

Part0：Implement a different preprocessing method

在資料的預處理部分，共有兩種，第一種為助教提供的remove_stopwords，第二種為自定義的preprocessing_function，詳細介紹如下：

- 助教提供的remove_stopwords函式：利用nltk提供的ToktokTokenizer將句子拆成一個個的單字並除去句子首尾空格後，再檢查單字使否屬於stopwords，若「是」，則該單字為可用的資料。
- 自訂義preprocessing_function函式：我首先將其大寫的部分通通轉為小寫，並使用PorterStemmer做Stemming，將不同的單字表示型態一致化，降低文本複雜度。由於我在資料中發現藏有'<br / >'、'&amp'及'…….'等不重要的資訊，因此我將其從資料中移除。最後，移除句子中的單一字元。
    - 例子：若提供的資料為"<br / >This is a dog and that is a cat"則他會經過個階段
        - 將大寫部分轉為小寫

          '<br / >This is a dog and that is a cat'

          ⇒ '<br / >this is a dog and that is a cat'
        - 使用PorterStemmer做Stemming

          '<br / >this is a dog and that is a cat'

          ⇒ '<br / >thi is a dog and that is a cat'
        - 移除'…….'、'<br / >'及'&amp'

          '<br / >thi is a dog and that is a cat'

          ⇒ ' thi is a dog and that is a cat'
        - 移除單一字元

          ' thi is a dog and that is a cat'

          ⇒ 'thi is dog and that is cat'

Part1：Implement the bi-gram language model

Perplexity

Perplexity，又稱為「困惑度」，是一個可以用於衡量語言模型的指標，計算Perplexity的數學式子如下：

$$Perplexity = 2^{-l} \quad \text{where} \quad l = \frac{1}{M}\sum_{i=1}^{m} \log p(w_i)$$

Perplexity的計算公式

當我們使用不同模型時，Perplexity中P(wi)的計算方式會不同，比如unigram為P(wi)，bigram為P(wi|wi-1)等等，當句子愈通順時，單字的P(wi)會愈高，使得Perplexity愈低。

當Perplexity愈小時，則「困惑度」愈低，代表模型對原資料愈不困惑，該模型在資料下訓練得更好。

Output

1. No Preprocessing

在不做資料預處理的狀況下：

- Preplexity = 116.2605
- F1 score = 0.7057
- Precision = 0.7088
- Recall = 0.7065

```
# no preprocessing
!python main.py --model_type ngram --preprocess 0 --part 2
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
start second part...
100% 40000/40000 [00:17<00:00, 2348.54it/s]
100% 10000/10000 [03:46<00:00, 44.10it/s]
Perplexity of ngram: 116.26046015880357
100% 40000/40000 [06:50<00:00, 97.42it/s]
100% 10000/10000 [01:45<00:00, 94.99it/s]
F1 score: 0.7057, Precision: 0.7088, Recall: 0.7065
```

2. Only With remove_stopwords

若是使用remove_stopwords的話：

- Preplexity = 195.4325
- F1 score = 0.6769
- Precision = 0.6926
- Recall = 0.6815

```
# only with stopwords
!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!
start preprocess...
start second part...
100% 40000/40000 [00:11<00:00, 3496.70it/s]
100% 10000/10000 [01:48<00:00, 92.42it/s]
Perplexity of ngram: 195.43245350997685
100% 40000/40000 [04:12<00:00, 158.17it/s]
100% 10000/10000 [01:02<00:00, 161.28it/s]
F1 score: 0.6769, Precision: 0.6926, Recall: 0.6815
```

3. With My Method

若是使用自定義preprocessing_function的話：

- Preplexity = 164.3391

- F1 score = 0.7229
- Precision = 0.7242
- Recall = 0.7232

```
# with my method
!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!
start preprocess...
start second part...
100% 40000/40000 [00:13<00:00, 2861.02it/s]
100% 10000/10000 [02:22<00:00, 70.11it/s]
Perplexity of ngram: 164.33912759557577
100% 40000/40000 [05:54<00:00, 112.91it/s]
100% 10000/10000 [01:28<00:00, 113.60it/s]
F1 score: 0.7229, Precision: 0.7242, Recall: 0.7232
```

由上面三個資料預處理的結果來看，可發現Perplexity在未做資料預處理時最低，原因為直接閱讀訓練資料，因此是最不容易產生困惑的，當資料做處理後，可能會移除多餘字句或是一致化表現形態，這些都將會使得字句閱讀起來「困惑度」增加，因此做資料預處理後Preplexity較高。不過，若拿remove_stopwords與preprocessing_function比較，可發現remove_stopwords又比preprocessing_function高許多，但F1的表現低於preprocessing_function，推測為過度簡化字句、喪失訊息，造成模型表現不佳。

Part2：Implement BERT model

Two pre-training steps in BERT

BERT的pre-training分為兩步驟：

- 遮蓋語言模型（Masked Language Model，MLM）：隨意將句子中的某些單字遮住，然後訓練BERT模型基於周遭語境猜出被遮住的詞語。MLM助於BERT理解句子中單詞之間的雙向關係。
- 下一句預測（Next Sentence Prediction，NSP）：將兩個句子餵入模型，並隨機遮蓋其中一個句子，並以固定頻率替換未被遮蓋的句子，訓練模型判斷未被遮蓋的句子是否為另一句的下一句。NSP助於BERT理解句子之間的關係，並提高其進行文本分類和問答等任務的能力。

Four different BERT application scenarios

以下為四種不同的BERT應用場景：

1. 自然語言理解：BERT可將自然語言文本轉換為電腦可理解的向量表示形式，使其更好地理解自然語言含義。
2. 語言生成：如機器翻譯、文本摘要等，透過在BERT模型的基礎上添加生成模型，可以產生更符合上下文的結果。
3. 文本分類：BERT也可以用於文本分類任務，例如情感分析、主題分類等。BERT可以將文本轉換為向量表示形式，然後使用這些向量作為輸入並進行分類任務。
4. 問答系統：BERT可用於問答系統，透過將問題和答案轉換為向量表示形式，將其嵌入到BERT模型中進行計算，從而獲得更好的問答結果。BERT在自然語言理解方面的表現優異，可以幫助問答系統更好地理解問題和文本內容。

Difference between BERT and distilBERT

BERT（Bidirectional Encoder Representations from Transformers）和distilBERT之間的主要差別在於模型的大小和計算成本。

BERT是一個大而複雜的模型，擁有3.4億個參數，因此在訓練和使用上計算成本很高，而distilBERT是BERT的較小且更快速的版本，只有6600萬個參數，參數較少，但速度卻更快，並保有一定的精準度。BERT是一個強大但計算成本高的語言模型，而distilBERT是一個更小、更高效的版本，仍然能夠實現類似的性能。

Output

F1 score : 0.9233

```
!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!
start preprocess...
start second part...
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 7.78kB/s]
Downloading (…)lve/main/config.json: 100% 483/483 [00:00<00:00, 159kB/s]
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 3.07MB/s]
Downloading (…)/main/tokenizer.json: 100% 466k/466k [00:00<00:00, 5.62MB/s]
Downloading pytorch_model.bin: 100% 268M/268M [00:01<00:00, 235MB/s]
Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing DistilBe
- This IS expected if you are initializing DistilBertModel from the checkpoint of a model trained on anc
- This IS NOT expected if you are initializing DistilBertModel from the checkpoint of a model that you e
  0% 0/1 [00:00<?, ?it/s]Epoch: 0, F1 score: 0.9233, Precision: 0.9235, Recall: 0.9233, Loss: 0.2646
100% 1/1 [27:01<00:00, 1621.12s/it]
```

The relation of the Transformer and BERT and the core of the Transformer

BERT是基於Transformer架構建立而成的，主要由Transformer的Ecoder部分組成。

Transformer的核心為自注意力機制，能夠解決模型遺忘早期輸入序列，編碼無法反映完整序列內容的問題，以及輸出與輸入階段必須依次進行而使速度變緩慢的困擾。自注意力機制將原先Encoder與Decoder之間的運算用於輸入序列自身，使得序列中的每一個Embedding皆可以互相注意、採納權重。

Part3：Implement LSTM model

我選擇GRU模型進行實作，使用pytorch實作GRU模型有三個要素：nn.Embedding、nn.GRU、nn.Linear。

Difference between vanilla RNN and LSTM

由於傳統的RNN在長時間記憶上的表現並不好，在處理長序列時會遇到梯度消失或梯度爆炸的問題，造成長時間的記憶被短時間的記憶所隱藏，因此LSTM設計了較佳的激勵函數(Activation Function)來改善RNN，此外，LSTM在神經單元中加入了遺忘、更新以及輸出三個步驟，進而提高RNN在長期記憶的表現，同時也是vanilla RNN與LSTM的主要差異。

Meaning of each dimension of the input and output for each layer in the model

RNN層的輸入是(batch_size, sequence_length, embedding_dim)，而輸出是(batch_size, sequence_length, hidden_dim)，其中hidden_dim是指隱藏層的特徵維度，即RNN層的輸出特徵維度。最後，將每個時間步長的隱藏狀態經過線性層，得到了(batch_size, output_dim)形狀的輸出，output_dim是最終輸出的維度。

舉例：100個句子，每個句子20個單字，每個單字有80維度，則形狀為(100, 20, 80)

Output

GRU的結果為：F1 score = 0.8744

```
!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!
start preprocess...
start second part...
  0% 0/10 [00:00<?, ?it/s]Epoch: 0, F1 score: 0.6744, Precision: 0.6834, Recall: 0.6772, Loss: 0.6903
 10% 1/10 [00:50<07:36, 50.70s/it]Epoch: 1, F1 score: 0.8157, Precision: 0.8383, Recall: 0.8184, Loss: 0.4421
 20% 2/10 [01:40<06:40, 50.12s/it]Epoch: 2, F1 score: 0.8744, Precision: 0.8752, Recall: 0.8745, Loss: 0.2643
 30% 3/10 [02:30<05:52, 50.29s/it]Epoch: 3, F1 score: 0.8795, Precision: 0.8817, Recall: 0.8797, Loss: 0.1753
 40% 4/10 [03:20<05:00, 50.08s/it]Epoch: 4, F1 score: 0.8795, Precision: 0.8813, Recall: 0.8796, Loss: 0.1132
 50% 5/10 [04:11<04:11, 50.31s/it]Epoch: 5, F1 score: 0.8798, Precision: 0.8804, Recall: 0.8798, Loss: 0.0692
 60% 6/10 [05:01<03:21, 50.37s/it]Epoch: 6, F1 score: 0.8765, Precision: 0.8768, Recall: 0.8765, Loss: 0.0456
 70% 7/10 [05:52<02:31, 50.44s/it]Epoch: 7, F1 score: 0.8775, Precision: 0.8775, Recall: 0.8775, Loss: 0.0314
 80% 8/10 [06:42<01:40, 50.41s/it]Epoch: 8, F1 score: 0.8736, Precision: 0.876, Recall: 0.8738, Loss: 0.0231
 90% 9/10 [07:32<00:50, 50.31s/it]Epoch: 9, F1 score: 0.8744, Precision: 0.8746, Recall: 0.8744, Loss: 0.0177
100% 10/10 [08:23<00:00, 50.32s/it]
```

Discussion

Why the technique is evolving from ngram -> LSTM -> BERT

最初為統計模型ngram，ngram僅在意單字出現的頻率和序列，也僅能用於簡單的語言處理。而接著有了深度學習模型LSTM與BERT，LSTM是基於RNN的神經網路技術，比起ngram能更好地處理自然語言的任務，然而RNN本身的設計結構在處理序列資料時必須一個一個處理資料，無法進行平行運算，也大幅拉長運算時間，而Transformer的出現就是為了解決RNN難以平行運算的問題，他使用了自注意力的機制改善此問題。由前述的演變進程來看，由於自然語言的任務愈趨困難，而深度學習的技術也愈趨進步，因此我們能看見不斷增進技術層面的模型出現，從單一統計模型(ngram)進化為具神經網路的深度學習模型(LSTM)，再演變為適用於平行運算能力的深度學習模型(Transformer)。

Problem and Solution

由於先前對ngram、BERT、RNN模型皆非常陌生，尤其在資料傳遞的部分常常需要做許多處理，要去對應模組的輸入輸出也很困難，網路上的資料也不知從何找起，很難找到類似的實作例子，所以在建構模型的時候花了很多心力，最後是靠著我非常厲害的同學們，向他們請教怎麼做會比較好，用甚麼方式處理資料會比較好，最後才勉強將作業完成。