人工智慧與機器學習

HW2

牛逼!情緒分析

學號：109401553

系級：資管三 B

學生：楊雲杰

日期：2024/04/13

一、Colab 連結

bert-base-uncased

https://colab.research.google.com/drive/1EkncxhSo35OBG3PlZOOctvg-h_h3B4lC?usp=sharing

distilbert-base-uncased(加分項)

https://colab.research.google.com/drive/1nPzTdEJdjAO3jpC1Gz5AWMcrt-BufYy9?usp=sharing

二、Test accuracy

1. 使用 bert-base-uncased 最終 test accuracy 為 92.08%

```
[ ]  # 計算測試集的準確率，並印出結果
     correct = 0
     for idx, pred in enumerate(res['pred']):
       if pred == res['label'][idx]:
         correct += 1
     print('test accuracy = %.4f'%(correct/len(test_df)))

     test accuracy = 0.9208
```

圖一、bert-base-uncased 的 test accuracy

2. 使用 distilbert-base-uncased 的輸出結果(加分項)

```
# 計算測試集的準確率，並印出結果
correct = 0
for idx, pred in enumerate(res['pred']):
  if pred == res['label'][idx]:
    correct += 1
print('test accuracy = %.4f'%(correct/len(test_df)))

test accuracy = 0.8964
```

圖二、distilbert-base-uncased 的 test accuracy

三、程式撰寫的過程

本次作業我用了兩個 pretrained model，分別是 bert-base-uncased 和 distilbert-base-uncased。我會講述我如何用 bert-base-uncased 訓練模型，再針對兩個 pretrained model 的差異進行比較。

以下是我的撰寫過程：

首先，我從 logits 的 dimension=1 去取得結果中數值最高者作為預測結果，透過將 tensor 轉為 numpy，算出 accuracy, f1_score, recall 及 precision。(詳見圖三)

```
[ ]  # 把輸入的矩陣變成一為向量
     def get_pred(logits):
         return torch.argmax(logits, dim=1)

     # calculate confusion metrics
     def cal_metrics(pred, ans):
         # Convert tensors to numpy arrays
         pred = pred.cpu().detach().numpy()
         ans = ans.cpu().detach().numpy()
         # Calculate metrics
         accuracy = accuracy_score(ans, pred)
         f1 = f1_score(ans, pred, average='weighted')  # 'weighted' accounts for label imbalance
         recall = recall_score(ans, pred, average='weighted')
         precision = precision_score(ans, pred, average='weighted')

         return accuracy, f1, recall, precision
```

圖三、get_pred()以及 cal_metrics()

再來，我從 dataset['train'] 和 dataset['test'] 中提取數據，並將它們合併到一個 Pandas DataFrame all_df 中(圖四)，並使用 train_test_split 方法將合併後的數據集分成了訓練集（80%）和臨時數據集（20%），接著又將這個臨時數據集進一步平均分成了開發集和測試集。最後，我將這些新切割的數據集分別保存為 TSV 格式的文件，分別是訓練集 (train.tsv), 開發集 (val.tsv), 和測試集 (test.tsv)。(圖六)

```
import pandas as pd
all_data = [] # a list to save all data
for data in dataset['train']:
  all_data.append({'text':data['text'], 'label':data['label']})
for data in dataset['test']:
  all_data.append({'text':data['text'], 'label':data['label']})

all_df = pd.DataFrame(all_data, columns=['text', 'label'])
all_df.head(5)
```

	text	label
0	I rented I AM CURIOUS-YELLOW from my video sto...	0
1	"I Am Curious: Yellow" is a risible and preten...	0
2	If only to avoid making this type of film in t...	0
3	This film was probably inspired by Godard's Ma...	0
4	Oh, brother...after hearing about this ridicul...	0

圖四、合併資料

```
[ ]  all_df.label.value_counts() / len(all_df)

     label
     0    0.5
     1    0.5
     Name: count, dtype: float64
```

圖五、兩個類別的分佈比例

```python
from sklearn.model_selection import train_test_split
# Split the combined dataset into training and a temp dataset (80% for training)
train_df, temp_data = train_test_split(all_df, random_state=1111, train_size=0.8)
# Further split the temp dataset into development and testing datasets evenly (50% each)
dev_df, test_df = train_test_split(temp_data, random_state=1111, train_size=0.5)
print('# of train_df:', len(train_df))
print('# of dev_df:', len(dev_df))
print('# of test_df data:', len(test_df))

# save data
train_df.to_csv('./train.tsv', sep='\t', index=False)
dev_df.to_csv('./val.tsv', sep='\t', index=False)
test_df.to_csv('./test.tsv', sep='\t', index=False)
```

```
# of train_df: 40000
# of dev_df: 5000
# of test_df data: 5000
```

圖六、重新切割並儲存

緊接著，我自訂一個 dataset(圖七)，並完成 tokenize 步驟，同時取得 input_ids、attention_mask 及 token_type_ids(圖八、圖九)。

```python
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import AutoTokenizer
import torch
import torch.nn.functional as Fun

# 使用PyTorch的Dataset來構建一個字定義資料加載器
class CustomDataset(Dataset):
  def __init__(self, mode, df, specify, args):
    assert mode in ["train", "val", "test"]  # 一般會切三份
    self.mode = mode
    self.df = df
    self.specify = specify # 指定用於預測的數據列名
    if self.mode != 'test':
      self.label = df['label'] #非測試模式需要標籤
    self.tokenizer = AutoTokenizer.from_pretrained(args["config"])
    self.max_len = args["max_len"]
    self.num_class = args["num_class"]

  def __len__(self):
    return len(self.df) #返回數據集的大小
```

圖七、自訂 CustomDataset

```python
# 獲取單個數據樣本
def __getitem__(self, index):

  sentence = str(self.df[self.specify][index])
  ids, mask, token_type_ids = self.tokenize(sentence)


  if self.mode == "test":
      return torch.tensor(ids, dtype=torch.long), torch.tensor(mask, dtype=torch.long), \
          torch.tensor(token_type_ids, dtype=torch.long)
  else:
      if self.num_class > 2:  #根據類別決定返回格式
        return torch.tensor(ids, dtype=torch.long), torch.tensor(mask, dtype=torch.long), \
          torch.tensor(token_type_ids, dtype=torch.long), self.one_hot_label(self.label[index])
      else:
        return torch.tensor(ids, dtype=torch.long), torch.tensor(mask, dtype=torch.long), \
          torch.tensor(token_type_ids, dtype=torch.long), torch.tensor(self.label[index], dtype=torch.long)

# 若類別數超過2，將標籤轉換為one_hot編碼
def one_hot_label(self, label):
  return Fun.one_hot(torch.tensor(label), num_classes = self.num_class)
```

圖八、get_item 方法以及 one_hot_label 方法

```python
def tokenize(self,input_text):
#使用 Bert tokenizer 對文本進行分詞和編碼
    encoded = self.tokenizer.encode_plus(
        input_text,                              # 輸入文本
        add_special_tokens=True,                 #添加特殊符號如CLS
        max_length=self.max_len,                 # 設定最大長度，超過會被截斷
        pad_to_max_length=True,                  # 填充到最大長度
        return_attention_mask=True,
        return_tensors='pt',                     # 返回 PyTorch tensors
        truncation=True,                         # 截斷文本以符合最大長度
        return_token_type_ids=True               # 返回 token id
    )

    # 取得 tokens, mask 和 token_type_ids
    input_ids = encoded['input_ids'][0]
    attention_mask = encoded['attention_mask'][0]
    token_type_ids = encoded['token_type_ids'][0]

    return input_ids, attention_mask, token_type_ids
```

圖九、tokenize 方法

下一步是重新完成 BertClassifier，我先初始化 BertModel，並加入 Dropout 防止過擬合，同時我也放入一個線性層，輸出維度為類別數(圖十)。另外，我也按照題目需求在 forward function 中把輸入值放進對應層數(圖十)。

```python
import torch.nn as nn
# BERT Model
class BertClassifier(BertPreTrainedModel):
  def __init__(self, config, args):
    super(BertClassifier, self).__init__(config)
    self.bert = BertModel(config)                                      #初始化 BERT Model
    self.num_labels = args["num_class"]                                #類別數量
    self.dropout = nn.Dropout(args.get("dropout",0.5))                 #dropout層，減少過擬合
    self.classifier = nn.Linear(config.hidden_size, self.num_labels)  #初始化一個線性層，用於從 BERT 的輸出到最終的類別預測
    self.init_weights()                                               #初始化模型權重


  def forward(self, input_ids=None, attention_mask=None, token_type_ids=None, position_ids=None,
              head_mask=None, inputs_embeds=None, labels=None, output_attentions=None,
              output_hidden_states=None, return_dict=None):
    return_dict = return_dict if return_dict is not None else self.config.use_return_dict

      # bert output
    outputs = self.bert(
          input_ids = input_ids,
          attention_mask=attention_mask,
          token_type_ids=token_type_ids,
          return_dict=True
      )

    # 從輸出中獲取 [CLS] 標記的輸出，用於分類的任務
    pooled_output = outputs.pooler_output
    pooled_output = self.dropout(pooled_output) #在經過線性層之前應用 Dropout
    logits = self.classifier(pooled_output) #通過線性層得到最終的 logits

    return logits
```

圖十、BertClassifier

接下來，我定義 Hyperparameters，learning rate=0.000082，epochs=3，max_len=512，batch_size=16，dropout=0.7，並利用 pretrained model: bert-base-uncased。(詳見圖十一)

```
[ ]  from datetime import datetime
     parameters = {
         "num_class": 2,
         "time": str(datetime.now()).replace(" ", "_"),
         # Hyperparameters
         "model_name": 'BERT',
         "config": 'bert-base-uncased',
         "learning_rate": 0.000082,
         "epochs": 3,
         "max_len": 512,
         "batch_size": 16,
         "dropout": 0.7,
     }
```

圖十一、相關參數

讀入資料。(圖十二)

```
[ ]  import transformers
     import pandas as pd

     # load training data
     train_df = pd.read_csv('./train.tsv', sep = '\t').sample(4000).reset_index(drop=True)
     train_dataset = CustomDataset('train', train_df, 'text', parameters)
     train_loader = DataLoader(train_dataset, batch_size=parameters['batch_size'], shuffle=True)

     # load validation data
     val_df = pd.read_csv('./val.tsv', sep = '\t').sample(500).reset_index(drop=True)
     val_dataset = CustomDataset('val', val_df, 'text', parameters)
     val_loader = DataLoader(val_dataset, batch_size=parameters['batch_size'], shuffle=True)
```

圖十二、讀取資料

採用 Adam 作為 optimizer，並使用 scheduler(get_linear_schedule_with_warm_up)

```
[ ]  transformers.logging.set_verbosity_error() # 關閉警告消息

     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     # 從預訓練的 BERT 模型加載自定義的分類器，並將其移動到指定的設備（GPU或CPU）
     model = BertClassifier.from_pretrained(parameters['config'], parameters).to(device)
     loss_fct = torch.nn.CrossEntropyLoss() # 使用 cross entrophy loss
     # 定義優化器，這裡使用 Adam 優化演算法
     # 設置學習率，以及 beta 和 epsilon 參數（這些參數控制優化器的行為）
     optimizer = torch.optim.Adam(model.parameters(), lr=parameters['learning_rate'], betas=(0.9, 0.98), eps=1e-9)
     # 計算總的訓練步數，為訓練輪數乘以訓練數據加載器中的批次數量
     total_steps = len(train_loader) * parameters["epochs"]
     # 設定帶預熱的線性學習率調度器
     # 預熱步驟通常是訓練開始的一部分，在這期間學習率逐漸增加到初始設置的學習率
     scheduler = get_linear_schedule_with_warmup(optimizer,
                                                 num_warmup_steps=int(total_steps * 0.1),
                                                 num_training_steps=total_steps)
```

圖十三、定義 optimizer 以及 scheduler

最後，完成我的模型訓練。(圖十四、圖十五)

```python
import time

# 初始化存儲訓練和驗證階段性能指標的字典
metrics = ['loss', 'acc', 'f1', 'rec', 'prec']
mode = ['train_', 'val_']
record = {s+m :[] for s in mode for m in metrics}

# 開始訓練過程
for epoch in range(parameters["epochs"]):

    st_time = time.time()
    train_loss, train_acc, train_f1, train_rec, train_prec = 0.0, 0.0, 0.0, 0.0, 0.0
    step_count = 0

    model.train()
    for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/{parameters['epochs']}"):
        ids, masks, token_type_ids, labels = [b.to(device) for b in batch]

        optimizer.zero_grad()  # 清除之前的梯度

        logits = model(input_ids=ids, attention_mask=masks, token_type_ids=token_type_ids)

        loss = loss_fct(logits.view(-1, parameters['num_class']), labels.view(-1))
        loss.backward()           # 反向傳播計算梯度
        optimizer.step()          # 更新模型參數
        scheduler.step()          # 更新學習率
        preds = get_pred(logits)  # 獲取預測結果

        # 累加損失和計算準確率、F1分數、召回率、精確率
        train_loss += loss.item()
        acc, f1, rec, prec = cal_metrics(preds, labels)
        train_acc += acc
        train_f1 += f1
        train_rec += rec
        train_prec += prec
        step_count += 1
```

```python
    # 每個epoch結束後在驗證集上評估模型性能
    val_loss, val_acc, val_f1, val_rec, val_prec = evaluate(model, val_loader, device)
    # 計算平均損失和性能指標
    train_loss = train_loss / step_count
    train_acc = train_acc / step_count
    train_f1 = train_f1 / step_count
    train_rec = train_rec / step_count
    train_prec = train_prec / step_count

    print('[epoch %d] cost time: %.4f s'%(epoch + 1, time.time() - st_time))
    print('         loss     acc     f1      rec     prec')
    print('train | %.4f, %.4f, %.4f, %.4f, %.4f'%(train_loss, train_acc, train_f1, train_rec, train_prec))
    print('val   | %.4f, %.4f, %.4f, %.4f, %.4f\n'%(val_loss, val_acc, val_f1, val_rec, val_prec))

     # 紀錄每個訓練epoch的性能指標
    record['train_loss'].append(train_loss)
    record['train_acc'].append(train_acc)
    record['train_f1'].append(train_f1)
    record['train_rec'].append(train_rec)
    record['train_prec'].append(train_prec)

    record['val_loss'].append(val_loss)
    record['val_acc'].append(val_acc)
    record['val_f1'].append(val_f1)
    record['val_rec'].append(val_rec)
    record['val_prec'].append(val_prec)
```

圖十四、進行訓練

```
Epoch 1/3: 100%|██████████| 250/250 [05:59<00:00,  1.44s/it]
100%|██████████| 32/32 [00:16<00:00,  1.93it/s]
[epoch 1] cost time: 376.0096 s
          loss     acc     f1      rec     prec
train | 0.4030, 0.8020, 0.7991, 0.8020, 0.8283
val   | 0.1642, 0.9492, 0.9490, 0.9492, 0.9544

Epoch 2/3: 100%|██████████| 250/250 [05:57<00:00,  1.43s/it]
100%|██████████| 32/32 [00:16<00:00,  1.93it/s]
[epoch 2] cost time: 374.1910 s
          loss     acc     f1      rec     prec
train | 0.1717, 0.9447, 0.9448, 0.9447, 0.9525
val   | 0.2290, 0.9102, 0.9094, 0.9102, 0.9286

Epoch 3/3: 100%|██████████| 250/250 [05:57<00:00,  1.43s/it]
100%|██████████| 32/32 [00:16<00:00,  1.93it/s][epoch 3] cost time: 374.1745 s
          loss     acc     f1      rec     prec
train | 0.0520, 0.9852, 0.9852, 0.9852, 0.9870
val   | 0.1611, 0.9434, 0.9430, 0.9434, 0.9509
```

圖十五、訓練結果

訓練完模型後，為了對結果進行預測我寫了 Softmax、label2class()(見圖十六)以及 predict_one()(見圖十七)並進行單筆的預測(見圖十八)。

```
[ ]  def Softmax(x):
         """
         計算給定輸入張量 x 的Softmax。
         Softmax 函數用於多類別分類的輸出層，將 logits（原始輸出）轉換為概率分佈。

         參數:
         - x: 一個張量，通常是未經歸一化的預測值或logits。

         返回:
         - 一個張量，表示 x 中每個元素的Softmax概率值。
         """
         return torch.exp(x) / torch.exp(x).sum()

     def label2class(label):
         """
         將數字標籤轉換為對應的類別名稱。
         這在將模型的預測輸出（通常是數字標籤）轉換為更易於理解的形式時非常有用。

         參數:
         - label: 數字標籤（整數），在這個函數中，0 被映射到 'negative'，1 被映射到 'positive'。

         返回:
         - 字符串，表示標籤對應的類別名稱。
         """
         l2c = {0:'negative', 1:'positive'}
         return l2c[label]
```

圖十六、Softmax()方法以及 label2class()方法

```
# 對單一句子進行預測，返回每個類別的概率及預測的類別
def predict_one(query, model):

    # 載入tokenizer
    tokenizer = AutoTokenizer.from_pretrained(parameters['config'])

    # 將文本tokenize
    inputs = tokenizer(query, return_tensors="pt", max_length=parameters['max_len'], truncation=True, padding="max_length")
    input_ids = inputs['input_ids']
    attention_mask = inputs['attention_mask']

    # 將數據移動到模型所在設備
    input_ids = input_ids.to(device)
    attention_mask = attention_mask.to(device)

    # 使用模型進行預測
    with torch.no_grad():
        logits = model(input_ids=input_ids, attention_mask=attention_mask)

    # 將logits轉成softmax
    probs = Fun.softmax(logits, dim=1)

    # 最大值的索引即是類別
    _, pred = torch.max(probs, dim=1)
    return probs, pred
```

圖十七、predict_one()

```
[ ]  %%time
     probs, pred = predict_one("This movie doesn't attract me", model)
     #print(label2class(pred))
     print(f"Predicted class: {label2class(pred.item())}, Probabilities: {probs.cpu().numpy()}")

     Predicted class: negative, Probabilities: [[0.9886756  0.01132445]]
     CPU times: user 89.8 ms, sys: 982 µs, total: 90.8 ms
     Wall time: 196 ms
```

圖十八、預測結果

最後，我計算測試集的準確率並印出結果(圖十九)

```
[ ]  # 計算測試集的準確率，並印出結果
     correct = 0
     for idx, pred in enumerate(res['pred']):
       if pred == res['label'][idx]:
         correct += 1
     print('test accuracy = %.4f'%(correct/len(test_df)))

     test accuracy = 0.9208
```

圖十九、最終結果

根據訓練結果，對 Bert-base-uncased 以及 distilbert-base-uncased 進行比較：

表一、Bert-base-uncased 及 distilbert-base-uncased 之比較

	Bert-base-uncased	distilbert-base-uncased
模型大小	約有 1.1 億個參數	約有 6600 萬個參數
架構	12 層 transformer layer 12 個 self attention heads	6 層 transformer layer 12 個 self attention heads
準確率	準確率較佳(92.08%)	準確率較差(89.64%)
訓練速度	較慢	較快
使用時機	用在準確率要求高的任務中	用在資源有限的環境中
程式碼主要差異	使用 token_type_ids	不支援 token_type_ids

四、心得

有別於第一次作業名畫分類器的訓練是用 keras 這個相對高層的 api，這次作業的訓練是用 PyTorch 對 Bert 模型進行 fine-tune，做出一個情緒分析的模型。在本次作業中，我認為與上次作業比較不同的地方在於利用 keras 對模型進行 fine-tune 並不方便，而 Pytorch 可以較輕鬆達成 fine-tune。除此之外，keras 是相對高層的 api，程式碼寫起來較容易，反之，PyTorch 程式碼較難寫。

另外，在完成這次作業的過程中我遇到了許多問題，例如 PyTorch 相關程式碼不熟悉，fine-tune 找不到最佳的值等等。儘管如此，我並沒有因此而氣餒，反而上網尋找更多不同的方法。在測試過程中，我不只嘗試了不同的 optimizer，例如 SGD、AdamW，我也測試不同 scheduler 以及 learning rate 所訓練出來的模型的正確率。最後，我訓練出一個正確率為 92%的模型。

總而言之，這次作業不只是我學習到如何用 Pytorch 框架對模型進行 fine-tune，也讓我更深入了解原來人工智慧的領域如此寬闊。希望未來我可以繼續精進我的能力，在人工智慧這條道路上前行。

五、參考資料

1. distilbert
https://huggingface.co/docs/transformers/model_doc/distilbert
https://huggingface.co/transformers/v2.9.1/model_doc/distilbert.html
2. bert-base-uncased
https://huggingface.co/google-bert/bert-base-uncased