HW4 – Sentiment Analysis

一、 程式碼截圖與講解：

1. 資料前處理：

```
import pandas as pd
from keras.preprocessing import sequence

# 1 read files split by \n (data has no header)
training = pd.read_csv('training_label.txt', sep = '\n', header = None)
testing = pd.read_csv('testing_label.txt', sep = '\n', header = None)

# 1-a split data into train and test by +++$+++ & #####
train = pd.DataFrame(training[0].str.split(' \+\+\+\$\+\+\+ ').tolist(), columns=['y', 'x'])
test =  pd.DataFrame(testing[0].str.split('#####').tolist(), columns=['y', 'x'])

# 1-b set test and train
train_x = train['x']
train_y = train['y']
test_x = test['x']
test_y = test['y']

# 2-1 Tokenize the data
from keras.preprocessing.text import Tokenizer
token = Tokenizer(7000)
token.fit_on_texts(train_x)

x_train_seq = token.texts_to_sequences(train_x)
x_test_seq = token.texts_to_sequences(test_x)

# 2-2 set max length of data
x_train = sequence.pad_sequences(x_train_seq, maxlen=100)
x_test = sequence.pad_sequences(x_test_seq, maxlen=100)
```

a、 1 讀檔案

b、 1-a 將檔案依照+++$+++和#####切出 train 和 test

c、 1-b 設定好 train 和 test 的 x 和 y

2. 將 train 和 test 經 Tokenize 處理：

a、 2-1 把文字處理成 Tokenize 的數字轉換

b、 2-2 設定每段輸入的截長補短，設為 100

```
modelRNN = Sequential()  #建立模型
#Embedding層將「數字list」轉換成「向量list」
modelRNN.add(Embedding(output_dim=4,   #輸出的維度是32，希望將數字list轉換為32維度的向量
     input_dim=7000,  #輸入的維度是3800，也就是我們之前建立的字典是3800字
     input_length=100)) #數字list截長補短後都是380個數字

# 3-1 set RNN model
modelRNN.add(Dropout(0.7))                      #隨機在神經網路中放棄70%的神經元，避免overfitting
modelRNN.add(SimpleRNN(units=16))               #建立16個神經元的RNN層
modelRNN.add(Dense(units=256,activation='relu')) #建立256個神經元的隱藏層，ReLU激活函數
modelRNN.add(Dropout(0.7))
modelRNN.add(Dense(units=1,activation='sigmoid'))#建立一個神經元的輸出層，Sigmoid激活函數
modelRNN.summary()
```

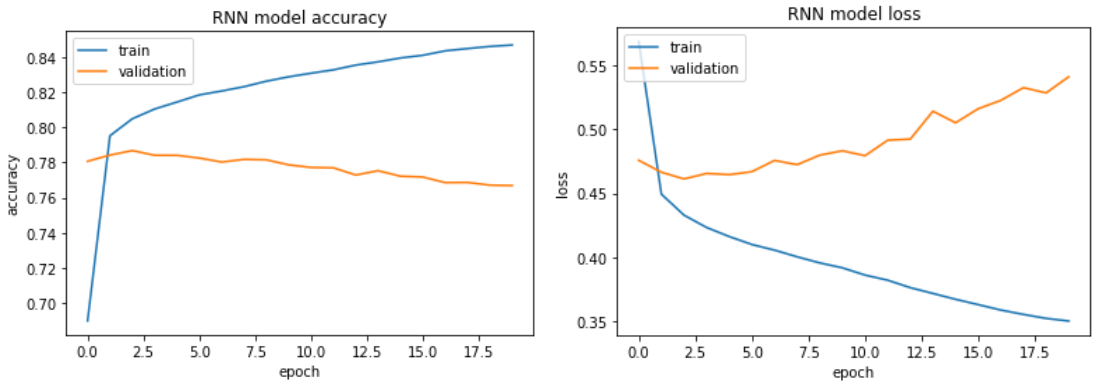
3. 撰寫 RNN Model：

a、 建立 model，測試有 dropout 和沒有 dropout 之間的差異

b、 撰寫成 LSTM 的版本（詳情請看 Code）

二、 程式實作結果：

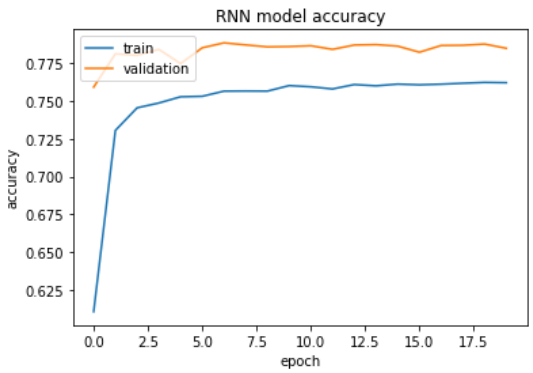
1. RNN 的 Overfitting：左 Accuracy，右 Loss

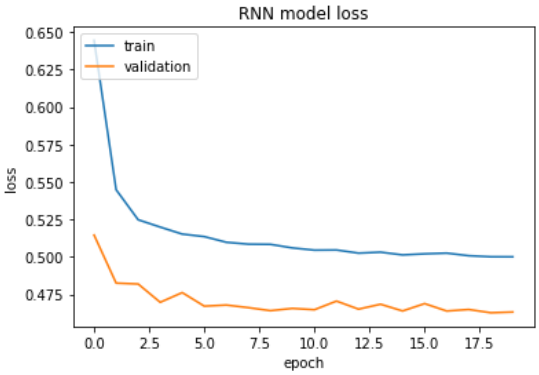


評估模型的 Accuracy 值：

1	float64	1	0.8222222169240315

2. RNN 的 Dropout(0.7)：左 Accuracy，右 Loss

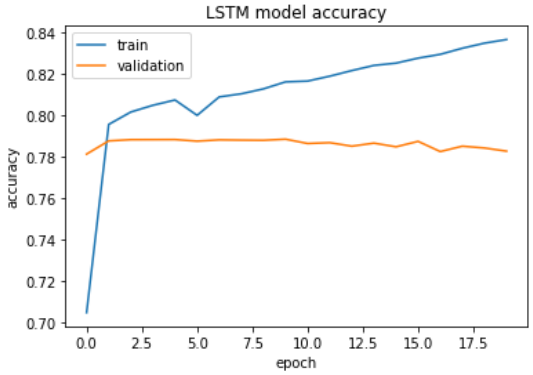


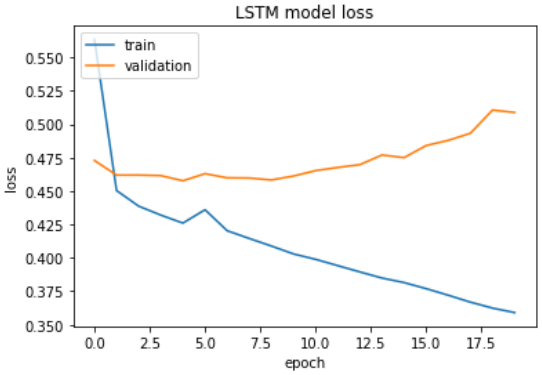


評估模型的 Accuracy 值：

```
1    float64  1    0.8111111137602064
```

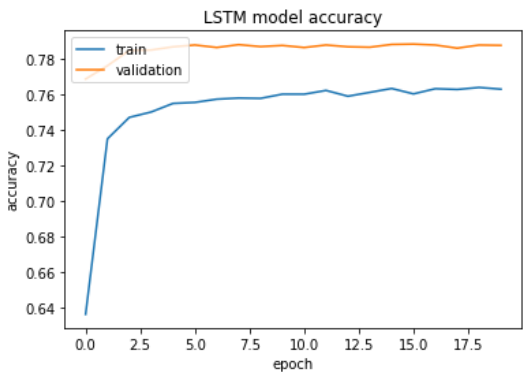
3. LSTM 的 overfitting：左 Accuracy，右 Loss

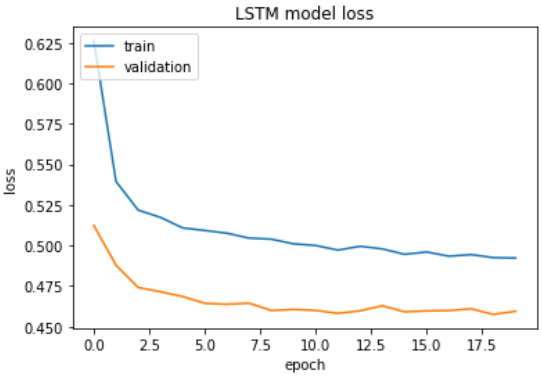





```
1    float64  1    0.8222222169240315
```

4. LSTM 的 Dropout(0.7)：左 Accuracy，右 Loss





評估模型的 Accuracy 值：

```
1    float64  1    0.8444444391462538
```

三、 發現與討論：

1. **RNN 與 LSTM 的準確度比較：**兩個的精準度上面差異並不大。

2. **加入 Dropout 後的 Loss 圖變化：**

   加入以後 Loss 的差異很大，無論是 RNN 或是 LSTM，在加入 dropout 以後，train 的 loss 雖然下降快速，但是 validation 的 loss 卻逐漸上升。

3. **是否有 Overfitting/Underfitting 的情況發生：**

   加入 Dropout 以後，可以大幅度減低 Overfitting 的狀況（當 training loss 下降的時候，Validation loss 卻持續上升）。

4. **Testing Accuracy 大於 Training Accuracy：**

   實務上發生這樣的狀況並不太正常，認為原因有兩個，第一個是因為這是時間序列資料，如果參考課堂 PPT 上的程式碼撰寫 validation_split=0.2 的話，可能會因此隨機拿到比較後面的資料，有先看到答案的問題。

   因此做了個小實驗，將 validation data 設成 test data ，不過無論是 RNN 還是 LSTM 的模型，還是出現 Testing Accuracy 高於 training 的狀況，想了想，因為時序資料的部分只是每一筆資料當中的時序，每筆資料之間是沒有時序關係的，所以推翻先前的推測。

   於是認為是**因為 testing data 相較於 training data 比數過少**，所以

容易被網路預測正確，做了一個只取更少比資料的測試，網路在前幾個 epoch 就都準確猜中了資料的答案。