國泰人壽
Cathay Life Insurance

2020 CIP 暑期實習計畫

實習成果報告
SPEC 文件推薦與 SPEC 片段推薦

壽險資訊部

壽險程式設計科

實習生：莊于霆

戴余修

中 華 民 國 一 ○ 九 年 八 月 二 十 五 日

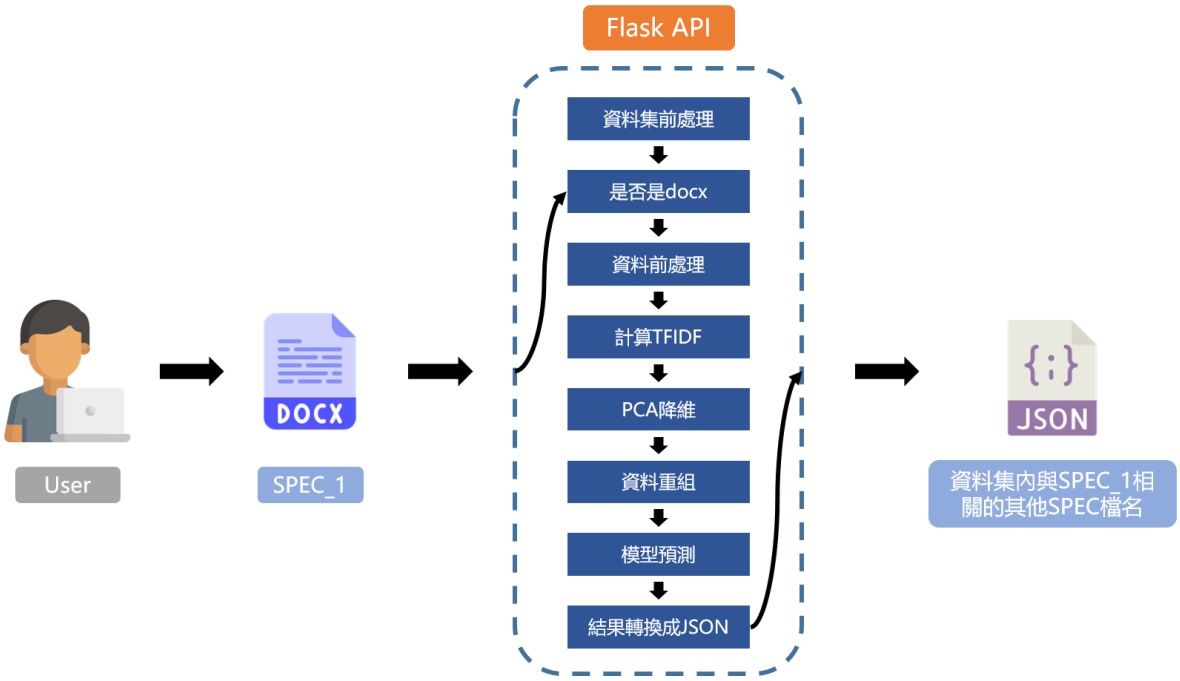
目錄

壹、 SPEC 文件推薦

一、 使用流程與目標

期望幫助使用者減少撰寫 SPEC 所需要的時間，使其能夠透過 SPEC 文件推薦，提供靈感或了解過去的 SPEC 資料集中有哪些可以參考。

本專案能提供使用者將其撰寫到一半或完成的 SPEC 上傳至本專案的 Flask API 上，而 API 將經過一連串的資料處理之後(判斷是否是 docx 檔、資料前處理、計算 TFIDF、PCA 降維、資料重組、模型預測、回傳預測結果)，以 JSON 的格式推薦過去可參考的 SPEC 檔名與相關機率提供給使用者參考。



二、 訓練與實作流程

1. 資料前處理

在取得原始資料時，一般有許多機器不需要的雜訊、缺漏值、機器難以閱讀等狀況。因此取得 SPEC 資料集後，在進行後續步驟前須進行資料前處理，去除髒亂無意義雜訊外，也將非結構化的文本資料轉換成結構化資料。本專案資料前處理的步驟與說明如下：

(1) 載入需要使用到之套件：

- docx、docx2txt：讀取 docx 檔案之套件。
- re：正規表示式常用套件。
- jieba：中文斷詞套件。
- numpy、pandas：資料處理常用套件。
- tqdm：進度條套件。
- CountVectorizer：計算詞頻(tf)套件。
- cosine_similarity：計算兩個向量餘弦相似度之套件。
- TfidfTransformer：計算 TFIDF 之套件。
- PCA：將高維資料已 PCA 降維法降維之套件。
- AgglomerativeClustering：層級聚類算法套件。
- plt、seaborn：資料視覺化之套件。

```
import os
import docx
import docx2txt
import re
import jieba
import numpy as np
import pandas as pd
from tqdm import tqdm
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.decomposition import PCA
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt
import seaborn as sns;sns.set()
```

(2) 讀取 docx 檔：

本專案在訓練階段使用 python-docx 與 docx2txt 兩個套件來讀取 SPEC 文檔；而當架設好 Flask API 服務時，只有使用到 docx2txt 來處理使用者上傳的 SPEC 文件檔。python-docx 可依照文件中的表格或段落分出獨立的物件，並可以分別讀取，但無法一次性讀取包含段落與表格的整份文件；docx2txt 可一次性讀取整份文件，但無法區分表格與段落。

(3) Jieba 斷詞：

人類了解語意的最小單位是字詞，而機器也不意外，因此需幫機器將 SPEC 文件進行斷詞，以清楚區分出一個一個的單詞。本專案所使用的斷詞工具為 Jieba。在斷詞之前，將公司資料庫內

的模組名稱、資料表名稱、資料庫欄位中文備註等資料匯入 Jieba 之中，使其學習 SPEC 中可能出現的專有名詞，令其斷詞效果能更符合預期。

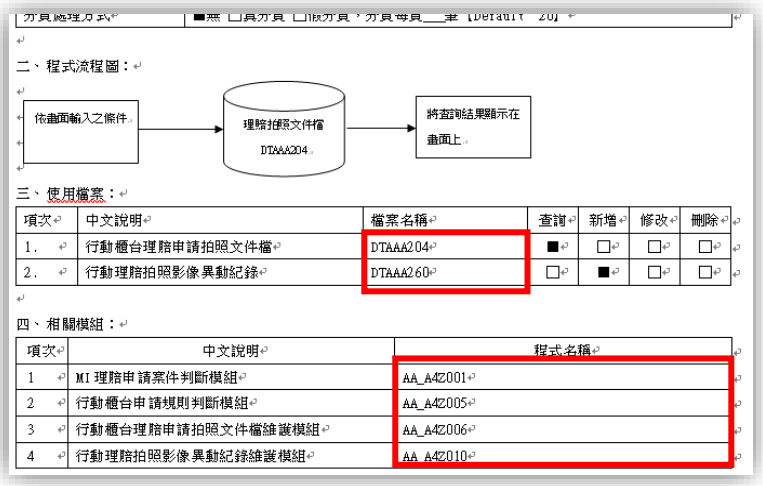
```
jieba.case_sensitive = True
jieba.set_dictionary('dict.txt.big.txt')
jieba.load_userdict('tableName.txt')
jieba.load_userdict('programNameList.txt')
jieba.load_userdict('allvocb.txt')
```

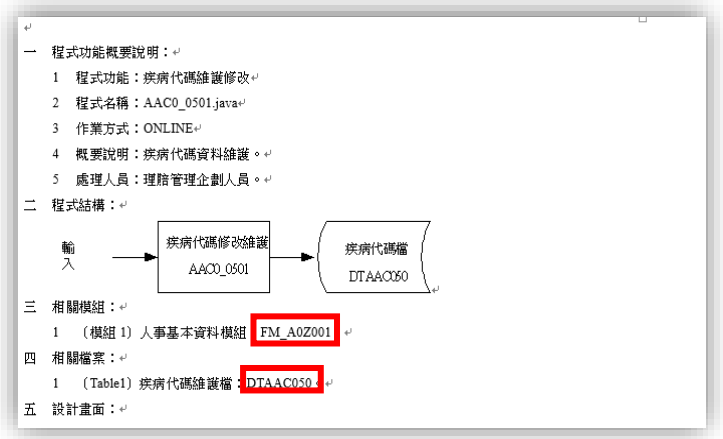
```
Building prefix dict from /content/dict.txt.big.txt ...
Dumping model to file cache /tmp/jieba.u43f464b336de155982df71dc6a2e3f9e.cache
Loading model cost 1.737 seconds.
Prefix dict has been built successfully.
```

(4) 取出 SPEC 中的模組與資料表名稱：

SPEC 內所使用到的模組與資料表為重要資訊，涵蓋了業務邏輯、可能操作的資料、程式功能等資訊，所以本專案取出每篇 SPEC 內所使用的模組與資料表，並組成字典的格式，可清楚了解哪篇 SPEC 使用了那些模組與資料表。

不過在 SPEC 資料集中，有些模組資料表資訊是寫在文件的表格中，有些則是以段落方式列點呈現。因此分別寫了兩支程式將其萃取出來。





從 SPEC 中取出資料表與模組資訊前，先讀取所有資料表名稱與所有模組名稱的檔案，將其存成陣列，以利後續讓程式便認該字詞是否是模組或資料表名稱。

```
tableNameFile = open('tableName.txt', 'r', encoding='UTF-8')
tableNameList = tableNameFile.read().split('\n')
programNameFile = open('programNameList.txt', 'r', encoding='UTF-8')
programNameList = programNameFile.read().split('\n')
```

首先會以 getTableModel()的函式來啟動讀取模組與資料表的過程，從設定的路徑下依序取得資料集中的 docx 檔，再經過 python-docx 套件的讀取後，再依我們欲取得的模組與資料表分別以 getRelatedFile()函式取得。而本專案將每一篇 SPEC 的模組與資料表的資訊以字典格式儲存，再用陣列來儲存每一個字典，將整個資料集存成一個字典陣列。

```
def getTableModel():

    path = 'subsystem/spec/'
    docTableModel = []

    for file in tqdm(os.listdir(path)):
        if file.endswith(".docx"):

            filename = path + file
            Doc = docx.Document(filename)
            TableModel = {}
            TableModel["文件檔名"] = file
            TableModel = getRelatedFile(Doc, TableModel, "相關檔案", ["檔案名稱","中文說明","檔案內容"], tableNameList)
            TableModel = getRelatedFile(Doc, TableModel, "相關模組", ["程式名稱","CLASS"], programNameList)

            docTableModel.append(TableModel)

    return docTableModel

docDictList = getTableModel()
```

getRelatedFile()的參數分別為：

- **doc**：docx 檔經過 python-docx 讀取過後的 python-docx 物件。
- **dic**：我們欲儲存這篇 SPEC 模組資料表資訊的字典。
- **dicKeyName**：設定字典內的 Key 值。若是要儲存 SPEC 內模組，則本參數為”相關模組”；若為儲存 SPEC 內資料表，則設定為”相關檔案”。
- **tableCol**：大部分 SPEC 內的模組與資料表資訊會寫在表格之中，但所屬的欄位不一，因此本參數為設定關鍵字陣列，若使程式能依照關鍵字去表格內搜索，順利取出關鍵字欄位下的模組或資料表的名稱資料。例如取模組時該參數設定為["程式名稱","CLASS"]原因是經過本專案觀察，當模組寫在 SPEC 中的表格時，其該欄第一列有可能是"程式名稱"或"CLASS"。
- **TargetList**：目標單詞陣列，程式會依照本陣列內所擁有的字詞去篩選單詞。例如若要取得 SPEC 內模組，則本參數為模組名稱陣列。

getRelatedFile()會依照 python-docx 物件去遍歷每一個表格，並在每個表格的第一列查詢是否有 tableCol 設定的欄未，若有，則爬取該行下的所有資料，並以正規表示式去除非英文的部分(因模組名稱與資料表名稱皆為英文)，再確定該單詞的確存在 TargetList 之中，最後存在陣列當中。若整行的所有資料皆爬完後，將陣列轉為字串存入字典之中。

若跑完上述程式後，仍未取得模組或資料表資料，代表可能該資料是寫在段落之中，因此函式 getRelatedFilePar()負責讀取段落中的模組、資料表資訊。

```
def getRelatedFile(doc, dic, dicKeyName, tableCol, TargetList):

    fileList = []
    findChinese = re.compile("[^A-Za-z0-9^.^_]")

    for table in doc.tables:    # 遍歷所有表格
        fileNameIndex = 0
        for cell in table.rows[0].cells:
            if cell.text in tableCol:
                for file in table.rows[1:]:
                    try:
                        content = file.cells[fileNameIndex].text
                        if len(findChinese.findall(content)) == 0:
                            content = "".join(findChinese.sub('',file.cells[fileNameIndex].text))
                            if content != "" and content in TargetList:
                                fileList.append(content)
                    except:
                        fileList = []
                        print("表格格式讀取錯誤")

                dic[dicKeyName] = ",".join(fileList)

            fileNameIndex += 1

    if len(fileList) > 0:
        return dic
    else:
        if dicKeyName == "相關檔案":
            return getRelatedFilePar(doc, dic, "相關檔案", "相關檔案", TargetList)
        elif dicKeyName == "相關模組":
            return getRelatedFilePar(doc, dic, "相關模組", "相關模組", TargetList)
        else:
            print("錯誤!")
```

getRelatedFilePar()的參數分別為 doc、dic、dicKeyName 與 TargetList，功能與意義皆與 getRelatedFile()相同。getRelatedFilePar()將遍歷讀取文件中所有的段落，而讀取段落時，將用 jieba 斷詞並查看該段落內的詞語是否存在於 TargetList，若存在 TargetList 且該詞語第一次出現，則以陣列儲存起來。最後將陣列轉為字串存入字典中並回傳。

```
def getRelatedFilePar(doc, dic, dicKeyName, TargetList):

    findChinese = re.compile("[^A-Za-z0-9^.]")

    findRelatedFile = False
    content = []
    for para in doc.paragraphs:
        contentList = " ".join(jieba.cut(para.text)).split(" ")
        for i in range(len(contentList)):
            if contentList[i] in TargetList and contentList[i] not in content:
                content.append(contentList[i])

    dic[dicKeyName] = ",".join(content)

    return dic
```

最後 getTableModel()得出的結果是一個字典陣列，每個陣列元素儲存一篇 SPEC 的模組、資料表資訊。如下圖所示。

(5) 去除 SPEC 中的停用詞、標點符號、程式碼：

除了模組與資料表的資訊外，其他中文字詞也涵蓋許多業務資訊、SPEC 主題等，可能也涵蓋撰寫風格、撰寫形式等等資訊，因此本專案另一個前處理方式便是將整篇 SPEC 斷詞後，去除每篇 SPEC 中的停用詞(語助詞或對意義不大的字詞)、標點符號與程式碼，保留大部分的中文字詞資訊。將整個 SPEC 資料集整理成陣列格式暫存。

在處理整篇 SPEC 前，先宣告一些變數與正規表示式規則，也讀取停用詞檔案，以陣列方式儲存。

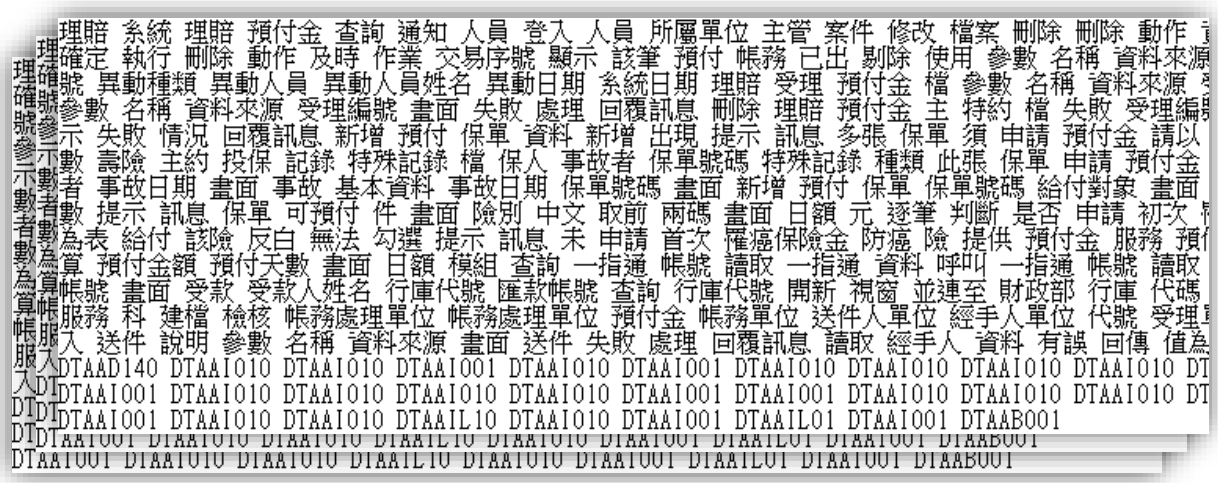
```
source_directory = 'subsystem/spec/'
findEnNum = re.compile("[^A-Za-z0-9]")
rule = re.compile("[^\u4e00-\u9fa5]")
docList = []
docIndex = []

stopWordFile = open('stop_word.txt', 'r', encoding='UTF-8')
stopWordList = stopWordFile.read().split('\n')
stopwordcount = 0
getstopword = []
```

接下來根據路徑依序以 docx2txt 讀取整篇 docx 檔，轉成字串格式。之後再依照正規表示式分別用兩個字串保留英文與中文的資訊，再將這兩個字串放入 jieba 斷詞後，中文的字串轉成陣列後再將經過停用詞的篩選；英文的字串也轉成陣列後再經過資料表、模組名稱陣列的篩選，最後合併並以陣列儲存回傳該 SPEC 前處理後的結果。

```
for  file  in  tqdm(os.listdir(source_directory)):
        noStopWordTokenList  =  []
        if  file.endswith(".docx"):
                filename  =  source_directory  +  file
                docIndex.append(file)
                docxToText  =  docx2txt.process(filename)

                docxToTextEnNum  =  "".join(findEnNum.sub('',docxToText))
                docxToText  =  "".join(rule.sub('',docxToText))

                tokenizeStr  =  " ".join(jieba.cut(docxToText))
                tokenizeStrEnNum  =  " ".join(jieba.cut(docxToTextEnNum))

                newTokenStr  =  tokenizeStr.split(" ")
                newTokenStrEnNum  =  tokenizeStrEnNum.split(" ")


                for  token  in  newTokenStr:
                    if  token  in  stopWordList:
                        stopwordcount  +=  1
                        getstopword.append(token)
                    else:
                        noStopWordTokenList.append(token)

                for  token  in  newTokenStrEnNum:
                    if  token  in  tableNameList  or  token  in  programNameList:
                        noStopWordTokenList.append(token)

                newTokenizeStr  =  " ".join(noStopWordTokenList)
                docList.append(newTokenizeStr)
```

```
100%|██████████| 786/786 [01:34<00:00,  8.34it/s]
```

最後整篇 SPEC 經過去除 SPEC 中的停用詞、標點符號、程式碼後，前處理後的結果會如下圖所示。



2. 特徵工程與分群

特徵工程的目的為將原始資料轉化成更好的表達問題本質的特徵的過程，這些特徵運用到預測模型中能提高對未知資料的模型預測精度。而本專案也根據前處理的兩種方式，進行不同的特徵工程。

本專案以 One-Hot Encode 的編碼方式處理取出的資料表與模組資料，了解那些模組或資料表有被這篇 SPEC 使用，而哪些沒有。將其組成陣列暫存起來。例如：若該 SPEC 只有使用到模組 1 時，One-Hot Encode 編碼為[1, 0, 0, 0, … 0, 0]，若該 SPEC 使用到模組 1 與模組 3 模組 4 時，則編碼為[1, 0, 1, 1, … 0, 0]。

實作方面，先從前段完成的 SPEC 資料表、模組字典陣列取出每篇 SPEC 的資料表與模組資料，放入陣列之中。因為陣列內的元素不會重複，因此可以以 TF 的套件計算出每篇 SPEC 模組與資料表的 TF，該 TF 同樣也為該 SPEC 的模組或資料表 One-Hot Encode 編碼，處理結果如下圖。

```
allDocTable = []
allDocModel = []
fileIndex = []
for i in range(len(docDictList)):
    tableStr = docDictList[i]["相關檔案"]
    modelStr = docDictList[i]["相關模組"]
    fileIndexStr = docDictList[i]["文件檔名"]
    fileIndex.append(fileIndexStr)
    allDocTable.append(tableStr)
    allDocModel.append(modelStr)

print(allDocTable)
print(allDocModel)

vectorizer = CountVectorizer()

T = vectorizer.fit_transform(allDocTable).toarray()
T_word = vectorizer.get_feature_names()

M = vectorizer.fit_transform(allDocModel).toarray()
M_word = vectorizer.get_feature_names()

print(T)
print(len(T_word))
print(T_word)

print(M)
print(len(M_word))
print(M_word)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
431
['dtaa0001', 'dtaa_d010', 'dtaa_d080', 'd
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
593
['aa_a0z001', 'aa_a0z002', 'aa_a0z003', '
```

另外，本專案也以 TFIDF 演算法處理整篇 SPEC 的字詞，幫助了解這篇 SPEC 有使用到哪些字詞，進而了解到哪些字詞在這篇 SPEC 是重要的，哪些字詞可以代表這一篇 SPEC。也以陣列形式暫存處理後的資料。TFIDF 簡介與處理結果如下圖。

```
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docList)
word = vectorizer.get_feature_names()
print(len(word))
transformer = TfidfTransformer()
tfidf = transformer.fit_transform(X)
weight = tfidf.toarray()
print(weight)
```

```
9074
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

得出每篇 SPEC 的 TFIDF 向量，即為整篇 SPEC 內的單詞特徵，其中含有許多除了模組與資料表的資訊，而本專案期望藉以分群的方式，囊括更多的特徵來代表一篇 SPEC，而透過分群的方式，能使這些特徵相似的資料產生關聯。因此本專案依據向量進行分群。

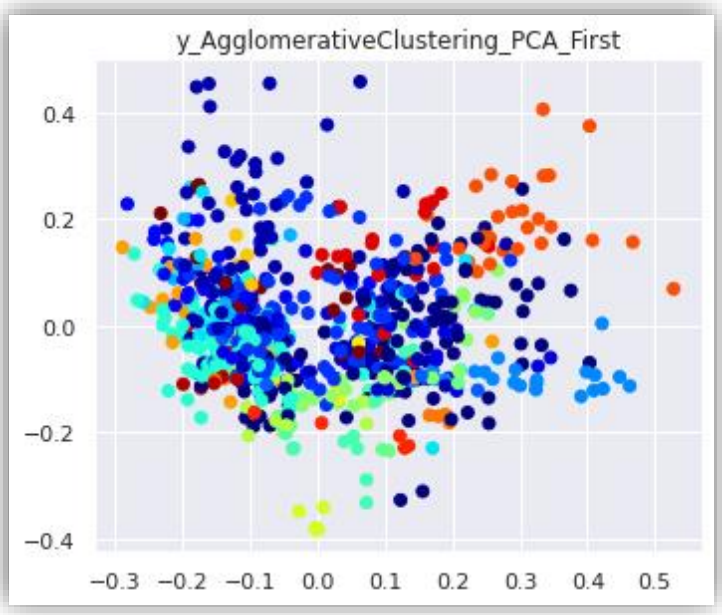
分群前，本專案會先將資料降維，因原始 TFIDF 矩陣維度過大，直接分群有記憶體不足的可能，而且將資料適度的降維也有助於特徵萃取。本專案以 PCA 主成分分析法將 9074 維度降至 256 維。

分群的演算法本專案選擇 Agglomerative 演算法分群，它解決了傳統常用的 KMeans 分群演算法設定分群數但不知道群內是否真的相似的問題。Agglomerative 分群法可設定 threshold 決定每群內文本向量之間的最大距離，意即群內資料具有數學上一定的相似度。

```
pca = PCA(n_components = 256)
tfPCA = pca.fit_transform(weight)

clustering = AgglomerativeClustering(n_clusters=None,distance_threshold=2.0).fit(tfPCA)

pca = PCA(n_components = 2)
tfPCA_2D = pca.fit_transform(weight)

fig = plt.figure(figsize=(12,10))
ax1 = fig.add_subplot(2, 2, 1)

ax1.set_title("y_AgglomerativeClustering_PCA_First")
ax1.scatter(tfPCA_2D[:, 0], tfPCA_2D[:, 1], c=clustering.labels_, cmap='jet')

plt.show()
```

事後本專案計算群內餘弦相似度，檢查分群結果的群內相似度是否控制在期望之中。本專案餘弦相似度的閥值為 0.35。下圖為分群後

結果示意圖，同一顏色及代表分為同一群，而資料點在此圖上無法區分開來的原因是因為原始資料為 9074 維，壓縮成人類可以觀察的 2 維時已難以區分出原本在高維空間上的差異。



驗證分群結果的實作以下說明。先以 getClusterDocVec()函式根據每一群所含有的資料取出相對應的 TFIDF 向量並組成字典後，再以 getClusterCosScore()函式計算每一群內的兩兩 TFIDF 向量餘弦相似度，最後加總平均得出該群的平均相似度，最後再將所有群的平均分數加總平均，得出本次分群結果的群內平均相似度。

getClusterDocVec()的參數為：

- num_cluster：本次分群的分群數。
- output：本次分群結果。

```
def getClusterDocVec(num_cluster, output):

    resultDic = {}
    for i in range(num_cluster):
        resultDic[str(i)] = []

    for index, class_ in enumerate(output):
        resultDic[str(class_)].append(weight[index].tolist())

    return resultDic
```

getClusterCosScore ()的參數為：

- ClusterDocVecDic：由 getClusterDocVec()回傳的每群 TFIDF 向量字典。

```
def getClusterCosScore(ClusterDocVecDic):
    totalCosScore = 0
    # 群迴圈
    for i in range(len(ClusterDocVecDic)):
        # 群內迴圈
        cosScore = 0
        flag = 0
        loopCount = 0
        for j in tqdm(range(len(ClusterDocVecDic[str(i)]))):
            for q in range(flag, len(ClusterDocVecDic[str(i)])):
                if j == q:
                    continue
                else:
                    reshapeVect_j = np.array(ClusterDocVecDic[str(i)][j]).reshape(1, -1)
                    reshapeVect_q = np.array(ClusterDocVecDic[str(i)][q]).reshape(1, -1)
                    cosScoreMatrix = cosine_similarity(reshapeVect_j, reshapeVect_q)
                    cosScoreMatrix = cosScoreMatrix.reshape(cosScoreMatrix.shape[0])
                    cosScore += cosScoreMatrix[0]
                    loopCount += 1
            flag += 1
        try:
            cosScore /= loopCount
        except:
            cosScore = cosScore

        totalCosScore += cosScore

    return totalCosScore/len(ClusterDocVecDic)
```

```
getClusterCosScore(getClusterDocVec(aggNumCluster, aggClusterResult))
```

3. 上標籤

因取得的 SPEC 資料並未告訴我們哪些 SPEC 跟哪些 SPEC 是彼此相關，因此本專案必須自行想出一套合理的定義來定義兩篇 SPEC 彼此間是否相關，是可以讓機器進行推薦的。而本專案定義兩篇 SPEC 是否相關的規則有三項：

(1) 兩篇 SPEC 在分群時皆被分為同一群

```
def getRelateMatrixFromCluster(clusterResult,csvName):

    # clusterResult為分群結果
    # csvName為設定輸出關聯矩陣的csv檔名

    # 因自己與自己有關係，要扣掉
    relateCount = -1 * len(docIndex)
    clusterRelateMatrix = []
    for i in clusterResult:
        relateList = []
        for j in clusterResult:

            # 如果i跟j都屬於同一群的話則視為兩者有相關
            if i == j:
                relateList.append(1)
                relateCount += 1
            else:
                relateList.append(0)

        # 將與i的相關矩陣存入clusterRelateMatrix中
        clusterRelateMatrix.append(relateList)

    # 以Dataframe儲存成關聯矩陣並轉成csv檔
    clusterRelate_df = pd.DataFrame(clusterRelateMatrix,index=docIndex,columns=docIndex)
    clusterRelate_df.to_csv(csvName)
    print(clusterRelate_df)
    print("有多少個文件彼此關聯(不含自己)：",relateCount/2)

    # 回傳clusterRelateMatrix
    return clusterRelateMatrix

# clusterRelateMatrixKmeans = getRelateMatrixFromCluster(y_kmeans_PCA_First, "clusterRelateKmeans6.
clusterRelateMatrixAgg = getRelateMatrixFromCluster(aggClusterResult, "clusterRelateAgg2.3_df.csv")
```

```
                UCAAM2_B100.docx  ...  UCAAA0_B200.docx
UCAAM2_B100.docx                1  ...                1
UCAAZ6_B003.docx                0  ...                0
UCAAC0_0200.docx                0  ...                0
UCAAN1_B004.docx                0  ...                0
UCAAI1_B102.docx                0  ...                0
```

(2) 兩篇 SPEC 的模組向量餘弦相似度 ＞0.5

```
# 剛剛計算的模組TFIDF矩陣
M_1 = M

# 因自己與自己有關係，要扣掉
relateCount = -1 * len(fileIndex)
relateMatrix_model = []

# 將每一個文件的TFIDF向量取出，並與整個TFIDF矩陣做cos相似度計算
for vect_1 in M_1:
    reshapeVect = np.array(vect_1).reshape(1, -1)
    cosScoreMatrix = cosine_similarity(M, reshapeVect)
    cosScoreMatrix = cosScoreMatrix.reshape(cosScoreMatrix.shape[0])

    # 計算後的cos相似度矩陣為vect_1與其他vect的cos相似度。若相似度大於0.5則視為相關
    for i in range(len(cosScoreMatrix)):
        if cosScoreMatrix[i] >= cosThreshold:
            cosScoreMatrix[i] = 1
            relateCount += 1
        else:
            cosScoreMatrix[i] = 0
    relateMatrix_model.append(cosScoreMatrix)

# 以Dataframe儲存成關聯矩陣並轉成csv檔
relateMatrix_df_model = pd.DataFrame(relateMatrix_model,columns=fileIndex,index=fileIndex)
relateMatrix_df_model.to_csv('relateMatrix_df_model.csv')
print(relateMatrix_df_model.head())
print("有多少個文件彼此關聯(不含自己)：",relateCount/2)
```

```
                UCAAM2_B100.docx  ...  UCAAA0_B200.docx
UCAAM2_B100.docx               1.0  ...               0.0
UCAAZ6_B003.docx               0.0  ...               0.0
UCAAC0_0200.docx               0.0  ...               0.0
UCAAN1_B004.docx               0.0  ...               0.0
UCAAI1_B102.docx               0.0  ...               0.0

[5 rows x 785 columns]
```

(3) 兩篇 SPEC 的資料表向量餘弦相似度 ＞0.5

```
# 剛剛計算的資料表TFIDF矩陣
T_1 = T

# 因自己與自己有關係，要扣掉
relateCount = -1 * len(fileIndex)
relateMatrix_table = []

# 將每一個文件的TFIDF向量取出，並與整個TFIDF矩陣做cos相似度計算
for vect_1 in T_1:
    reshapeVect = np.array(vect_1).reshape(1, -1)
    cosScoreMatrix = cosine_similarity(T, reshapeVect)
    cosScoreMatrix = cosScoreMatrix.reshape(cosScoreMatrix.shape[0])

    # 計算後的cos相似度矩陣為vect_1與其他vect的cos相似度。若相似度大於0.5則視為相關
    for i in range(len(cosScoreMatrix)):
        if cosScoreMatrix[i] >= cosThreshold:
            cosScoreMatrix[i] = 1
            relateCount += 1
        else:
            cosScoreMatrix[i] = 0
    relateMatrix_table.append(cosScoreMatrix)

# 以Dataframe儲存成關聯矩陣並轉成csv檔
relateMatrix_df_table = pd.DataFrame(relateMatrix_table,columns=fileIndex,index=fileIndex)
relateMatrix_df_table.to_csv('relateMatrix_df_table.csv')
print(relateMatrix_df_table.head())
print("有多少個文件彼此關聯(不含自己)：",relateCount/2)
```

```
                  UCAAM2_B100.docx  ...  UCAAA0_B200.docx
UCAAM2_B100.docx               1.0  ...               0.0
UCAAZ6_B003.docx               0.0  ...               0.0
UCAAC0_0200.docx               0.0  ...               0.0
UCAAN1_B004.docx               0.0  ...               0.0
UCAAI1_B102.docx               0.0  ...               0.0

[5 rows x 785 columns]
```

三項規則彼此之間為聯集關係，意為其中一項成立則視為兩篇 SPEC 是彼此相關。由 SPEC 內所有字詞的分群結果代表群內重要的詞語有一定的相似度，主題上、撰寫風格與形式上有相似，可以提供推薦供使用者參考。而模組與資料表具有高相似度時，可以解釋兩份 SPEC 要完成的功能與使用到的資料具有高度相似，業務邏輯也可能高度相似，因此也可推薦供使用者參考。

```
def outputFinalRelateMatrix(clusterRelateMatrix, relateMatrix_model, relateMatrix_table):

    # clusterRelateMatrix, relateMatrix_model, relateMatrix_table為分群關聯矩陣、模組關聯矩陣、資料表關聯矩陣

    # 因自己與自己有關係，要扣掉
    relateCount = -1 * len(docIndex)

    # 宣告新關聯矩陣
    relateMatrix_result = []

    for i in range(len(docIndex)):

        # 宣告某SPEC與其他SPEC的關聯陣列：relateMatrix_row
        relateMatrix_row = []
        for j in range(len(docIndex)):
```

```
            # 若分群關聯矩陣、模組關聯矩陣、資料表關聯矩陣內的其中一個元素為1則視為相關
            if clusterRelateMatrix[i][j] == 1 or relateMatrix_model[i][j] == 1 or relateMatrix_table[i][j] == 1:
                relateMatrix_row.append(1)
                relateCount += 1
            else:
                relateMatrix_row.append(0)

        # 將結果存入新關聯矩陣中
        relateMatrix_result.append(relateMatrix_row)

    # 以Dataframe儲存成關聯矩陣並轉成csv檔
    resultRelate_df = pd.DataFrame(relateMatrix_result,index=docIndex,columns=docIndex)
    resultRelate_df.to_csv('resultRelate_df.csv')
    print(resultRelate_df)
    print("有多少個文件彼此關聯(不含自己)：",relateCount/2)
    return relateMatrix_result

# outputRelateMatrixKmeans = outputFinalRelateMatrix(clusterRelateMatrixKmeans, relateMatrix_model, relateMatrix_table)
outputRelateMatrixAgg = outputFinalRelateMatrix(clusterRelateMatrixAgg, relateMatrix_model, relateMatrix_table)
```

```
                 UCAAM2_B100.docx  ...  UCAAA0_B200.docx
UCAAM2_B100.docx                 1  ...                 1
UCAAZ6_B003.docx                 0  ...                 0
UCAAC0_0200.docx                 0  ...                 0
UCAAN1_B004.docx                 0  ...                 0
UCAAI1_B102.docx                 0  ...                 0
...                            ...  ...               ...
UCAAA0_0200.docx                 0  ...                 0
UCAAB1_1400.docx                 0  ...                 0
UCAAH2_B301.docx                 0  ...                 0
UCAAB7_0200.docx                 1  ...                 1
UCAAA0_B200.docx                 1  ...                 1
```

本專案的資料範圍為 AA 理賠內主程式的 SPEC，一共 785 篇 SPEC。SPEC 與 SPEC 兩兩成對後一共產生 307,720 筆資料，其中 37,128 筆的資料標籤為"相關"，只佔資料集資料量的 1/9。而其中標籤為"相關"的資料內，有 2,650 筆資料是因為模組相似而產生的相關；有 6,276 筆是因資料表相似而產生的相關；而有 31,175 筆資料是因為同屬一群而產生的相關。下圖為標籤完後的資料示意圖。

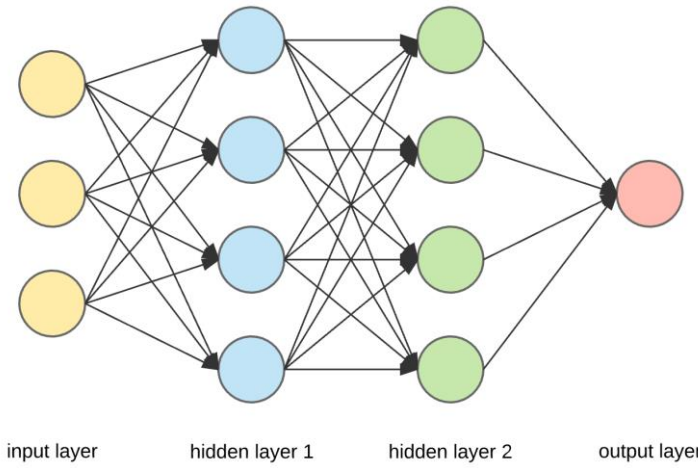
```
def createDataset(relateMatrix_result, csvName):

    # relateMatrix_result最後的關聯矩陣
    # csvName 輸出的資料集名稱設定

    # 宣告資料集陣列
    dataset = []

    # flag 若過去已有紀錄的關聯便不再重複紀錄EX: A與B相關，則遇到B與A相關時便不再紀錄
    flag = 0
    for i in range(len(relateMatrix_result)):
        for j in range(flag, len(relateMatrix_result[0])):

            # 自己與自己的關聯不記錄
            if i == j:
                continue
            else:
                dataset.append([docIndex[i],docIndex[j],relateMatrix_result[i][j]])
        flag += 1

    # 以Dataframe儲存成資料集並轉成csv檔
    dataset_df = pd.DataFrame(dataset,columns=["SPEC1","SPEC2","relation"])
    print(dataset_df.head())
    dataset_df.to_csv(csvName)

# createDataset(outputRelateMatrixKmeans, "SPEC_relation_dataset_Kmeans_df.csv")
createDataset(outputRelateMatrixAgg, "SPEC_relation_dataset_Agg_df_0.4.csv")
```

```
              SPEC1             SPEC2  relation
0  UCAAM2_B100.docx  UCAAZ6_B003.docx         0
1  UCAAM2_B100.docx  UCAAC0_0200.docx         0
2  UCAAM2_B100.docx  UCAAN1_B004.docx         0
3  UCAAM2_B100.docx  UCAAI1_B102.docx         0
4  UCAAM2_B100.docx  UCAAA6_0100.docx         0
```

SPEC1	SPEC2	relation
UCAAV4_0100.docx	UCAAV3_B005.docx	0
UCAAV4_0100.docx	UCAAH3_B303.docx	0
UCAAV4_0100.docx	UCAAH3_B302.docx	0
UCAAV4_0100.docx	UCAAC0_1202.docx	0
UCAAV4_0100.docx	UCAAD0_1500.docx	0
UCAAV4_0100.docx	UCAAD0_0101.docx	0
UCAAV4_0100.docx	UCAAB2_B010.docx	0
UCAAV4_0100.docx	UCAAV0_B012.docx	1
UCAAV4_0100.docx	UCAAH5_B203.docx	0
UCAAV4_0100.docx	UCAAA6_B102.docx	0
UCAAV4_0100.docx	UCAAA7_0200.docx	0
UCAAV4_0100.docx	UCAAD3_0200.docx	0
UCAAV4_0100.docx	UCAAMI_0100.docx	0
UCAAV4_0100.docx	UCAAA6_1200.docx	0
UCAAV4_0100.docx	UCAAD0_0210.docx	0
UCAAV4_0100.docx	UCAAV2_0103.docx	1

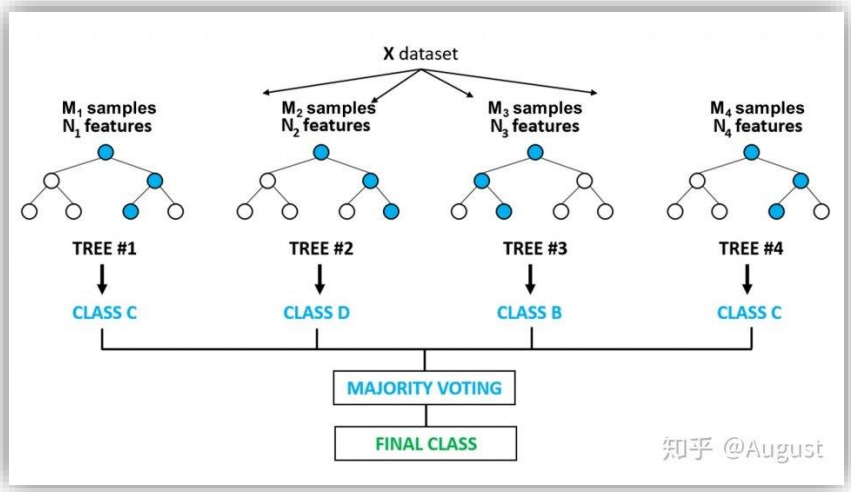
4. 模型訓練與評估

本專案使用兩個不同的模型實作 SPEC 推薦，第一個是使用以 Pytorch 撰寫的簡單三層 NN 類神經網路，因其架構簡單有助於未來在使用其他較進階的模型時，有個比較的標準。而模型的架構圖與程式碼如下。其餘實作部分因已架設於 Flask API 上，因此由下一部分說明。



```python
class Classifier(torch.nn.Module):     # 繼承 torch 的 Module
    def __init__(self):
        super(Classifier, self).__init__()     # 繼承 __init__ 功能
        self.hidden1 = torch.nn.Linear(256, 128)   # 隱藏層線性輸出
        self.hidden2 = torch.nn.Linear(128, 64)
        self.out = torch.nn.Linear(64, 2)       # 輸出層線性輸出

    def forward(self, x):
        # 正向傳播輸出值,神經網路輸出值
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))# 激勵函數(隱藏層的線性值)
        x = self.out(x)
        return x
```

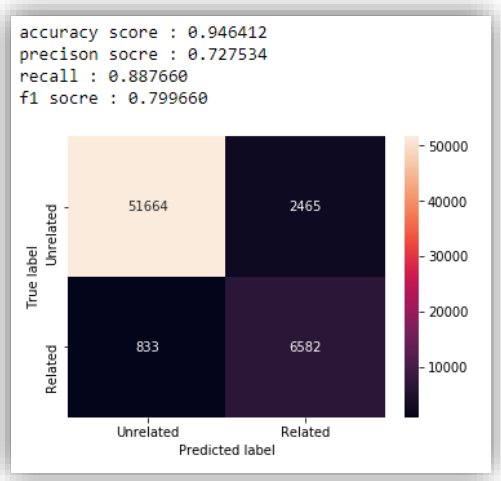
本專案另一個使用的模型為 Random Forest 隨機森林模型，由決策樹進階發展的集成機器學習模型，一個模型內由許多個決策樹分類器所組成，每棵決策樹產生預測結果後進行投票，投票結果集為該模型的預測結果。因此隨機森林模型預測與抗噪能力強、不容易過擬合，適合處理大量數值型數據。本專案當中，模型內決策樹的數量 n_estimators 設為 75 棵，加上因本專案資料集為不平衡資料集，因此 class_weight 的參數設為 balance，模型架構是意圖與程式碼如下圖。

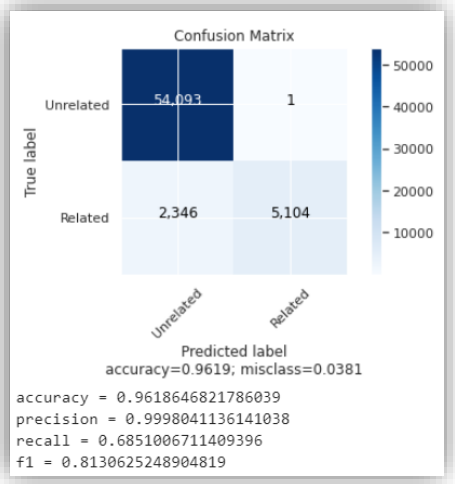



```python
from sklearn import ensemble
from sklearn.metrics import f1_score,precision_score,recall_score,accuracy_score

# 0.25:120
forest = ensemble.RandomForestClassifier(n_estimators = 75, n_jobs=-1, class_weight="balanced")
forest_fit = forest.fit(x_train, y_train)
output = forest_fit.predict(x_test)
```

兩個模型表現皆不錯，簡單三層 NN 類神經網路模型準確率約 94%，F1 值 0.79；而隨機森林模型準確率約 96%，F1 值 0.81。比較特別的是

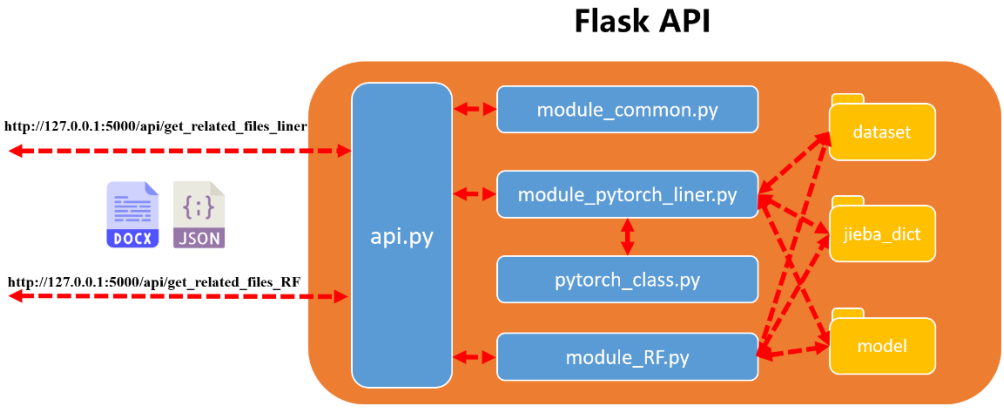
隨機森林模型的 Precision 很高，意即其預測為相關的資料，幾乎真的都是相關；不過它的 Recall 表現較差，意即預測為無關的資料，其實有部分資料是有關的。而 NN 三層神經網路模型表現得較為平均。兩個模型的混淆矩陣如下圖所示(左圖為 NN 三層類神經網路，右圖為隨機森林模型)。





5. 架設 Flask API

模型訓練完成後，即架設本專案之 Flask API，本專案的 SPEC 推薦可分別由兩個模型完成，因此有兩個不同的 URL 可以提供服務。而 API 內部的架構如下圖所示。



(1) api.py

在 API 開啟時，會先對 dataset 資料夾內的 SPEC 進行資料前處理、建立每篇 SPEC 的 TFIDF 向量、取出所有 SPEC 內有使用到的單詞，並 TFIDF 向量將進行 PCA 壓縮，取得 PCA 壓縮器與壓縮後的向量，為了後續使用者上傳 SPEC 時

能更快速的提供服務。

而因隨機森林模型的敏感性高，因此由不同機器或不同時間所訓練出的模型會因前處理的隨機數字不一致導致模型無法使用，因此隨機森林模型是在 API 執行時才開始進行訓練。

另外 api.py 作為 API 的 Controller，控制什麼樣的 request 要執行什麼樣的程式，並決定呼叫哪些模組來逐步完成 SPEC 文件推薦的工作。

(2) module_common.py

此模組所含有的方法是在使用兩個模型推薦 SPEC 時皆會使用到的方法，其中含有：

- **load_docxfile(request)**：取的使用者上傳的 docx 檔。
- **allowed_file(filename)**：判斷使用者上傳的檔案是否是 docx 檔。
- **get_label_file()**：載入資料及內的 SPEC 檔名。
- **get_top_related_doc(sort_predicted_list)**：處理預測結果陣列，回傳預測是相關文章，但若相關機率大於 0.9 的文章數超過 30 個的話，則只回傳相關機率大於 0.9 的全部資料。
- **result_list_to_json(related_doc_list)**：將最後的結果陣列轉成 JSON 格式並回傳。

(3) module_pytorch_liner.py

此模組所含之方法為在使用簡單三層 NN 神經網路模型時所會用到的方法，其中含有：

- **load_vocab(vocab_file)**：從 jieba_dict 資料夾內讀取 spec_vocab.txt，建立含有資料表、模組、AA 理賠中文字詞的關鍵字字典。
- **keywords_extraction(all_content)**：將 SPEC 的內容依照 load_vocab()所產生的關鍵字字典內含有的字詞將關鍵字取出。
- **tfidf(specs)**：將 dataset/spec/資料夾內的所有 SPEC 以取 keywords_extraction()出關鍵字後，計算得出每篇 SPEC

關鍵字的 TFIDF 向量，回傳每篇 SPEC 的關鍵字陣列與資料集內所有有用到的關鍵字。

- **tfidf_user_upload(specs, kw_string_list, dataset_key_word)**：將使用者上傳之文件取出關鍵字後，與先前前處理完的資料集內每篇 SPEC 的關鍵字陣列一同計算，取出回傳使用者上傳文件的 TFIDF 向量。
- **PCA_dimension_reduction(df_tfidf, specs)**：將資料集的 TFIDF 向量進行 PCA 降維至 128 維。回傳降維後的資料與降維的 PCA 壓縮器。
- **PCA_dimension_reduction_user_upload(user_tfidf_vect, pca)**：將使用者的上傳文件的 TFIDF 向量以先前壓縮資料集的 PCA 壓縮器進行降維至 128 維，並回傳降維後的向量。
- **concat_user_dataset_V(user_pca_vect, dataset_pca_vect)**：將使用者的降維後的 TFIDF 向量與資料集其他 SPEC 降維後的 TFIDF 向量合併，並回傳等待丟入模型預測。
- **predict(x)**：輸入 concat_user_dataset_V()合併後的向量，從 model 資料夾內載入模型後，預測使用者上傳的文件與資料集內的其他 SPEC 是否相關。
- **sort_predicted_list(predicted_list)**：將資料集結果依照模型預測相關的機率由大到小排序並回傳。

(4) pytorch_class.py

此模組內定義了簡單三層 NN 類神經網路模型的模型架構，其中含有 Classifier()類別。

(5) module_RF.py

此模組所含之方法為在使用隨機森林模型時所會用到的方法，其中含有：

- **preprocessing_allSPEC(source_directory)**：將 dataset/spec/資料夾內的所有 SPEC 做前處理，以 jieba 斷詞、去除停用詞、去除標點符號與程式碼，並回傳處理完的 SPEC 陣列與 SPEC 檔名陣列。
- **all_spec_tfidf_pca(docList)**：將以前處理完的 SPEC 陣列計算並取出每篇 SPEC 的 TFIDF 向量，並以 PCA 降維至 128 維，回傳 SPEC 資料集內有的字詞與 PCA 降維的

壓縮器。

- **train_RF_model(tfPCA, docIndex)**：傳入 SPEC 資料集降維後的 TFIDF 向量，並載入 dataset 資料夾內的 SPEC_relation_dataset_Agg_df_0.35.csv，該檔案內部含有 SPEC 與 SPEC 之間的關係標籤，並依照資料集內的兩兩 SPEC 的組合，將兩篇降維後的 TFIDF 向量合併。合併後以 sklearn 的隨機森林模型訓練。訓練完後印出訓練結果並儲存模型。
- **preprocessing(docx, datasetWords)**：與 preprocessing_allSPEC()的功能類似，將使用者上傳的文件進行前處理。以 jieba 斷詞、去除停用詞、去除標點符號與程式碼，並回傳處理完的文件字串。
- **load_RF_model(path)**：從 model 資料夾內載入訓練好的隨機森林模型。
- **get_tfidf_vec(docxList, TokenizeStr)**：將前處理後的使用者上傳文件與前處理後的資料集 SPEC 一同計算得出 TFIDF 向量，並取出回傳屬於使用者上傳文件的向量。
- **dimendion_reduction(PCAModel, user_vect_weight)**：將使用者上傳文件的 TFIDF 向量依照壓縮全部資料集的 PCA 壓縮器進行降維至 128 維。
- **concat_vec(dataset_vec, user_vec)**：將使用者上傳文件的已降維 TFIDF 向量，與資料集內的 SPEC 降維後的 TFIDF 向量合併連接。
- **RF_predict(pretrainModel, datalist)**：載入隨機森林模型與傳入合併後的向量，預測使用者上傳的文件與哪些既有的 SPEC 是相關的。
- **return_result(result, result_proba, docxNameList)**：取出預測結果為相關的 SPEC 檔名，並依照預測相關機率排序(由大到小)。

(6) dataset

該資料夾內含有本專案的資料集。其中 spec 資料夾內是未經過處理的 SPEC 文件集，而 SPEC_relation_dataset_Agg_df_0.4.csv、SPEC_relation_dataset_Agg_df_0.35.csv、SPEC_relation_dataset_Agg_df_0.3.csv、SPEC_relation_dataset_Agg_df_0.25.csv 為定義 SPEC 與 SPEC 之間的關係標籤檔案，檔名最後的數字代表該檔案的分群平

均群內相似度，數字越大代表群內的相似度越大，反之越小。

(7) jieba_dict

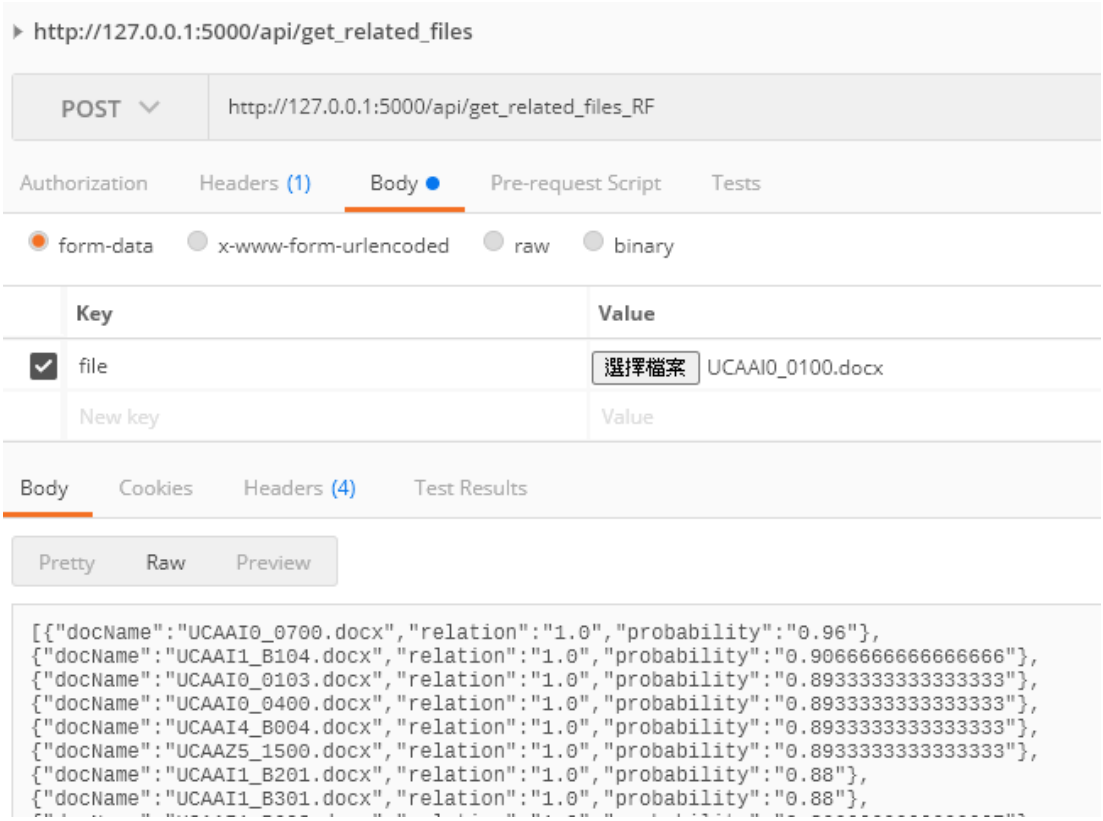
該資料夾內包含所有在斷詞時會需要讀取的檔案，包誇中文詞彙表、停用詞、模組名稱單詞、資料表名稱單詞、AA理賠資料庫內的欄位中文註釋單詞。

(8) model

存放簡單 NN 類神經網路模型參數與隨機森林模型檔。

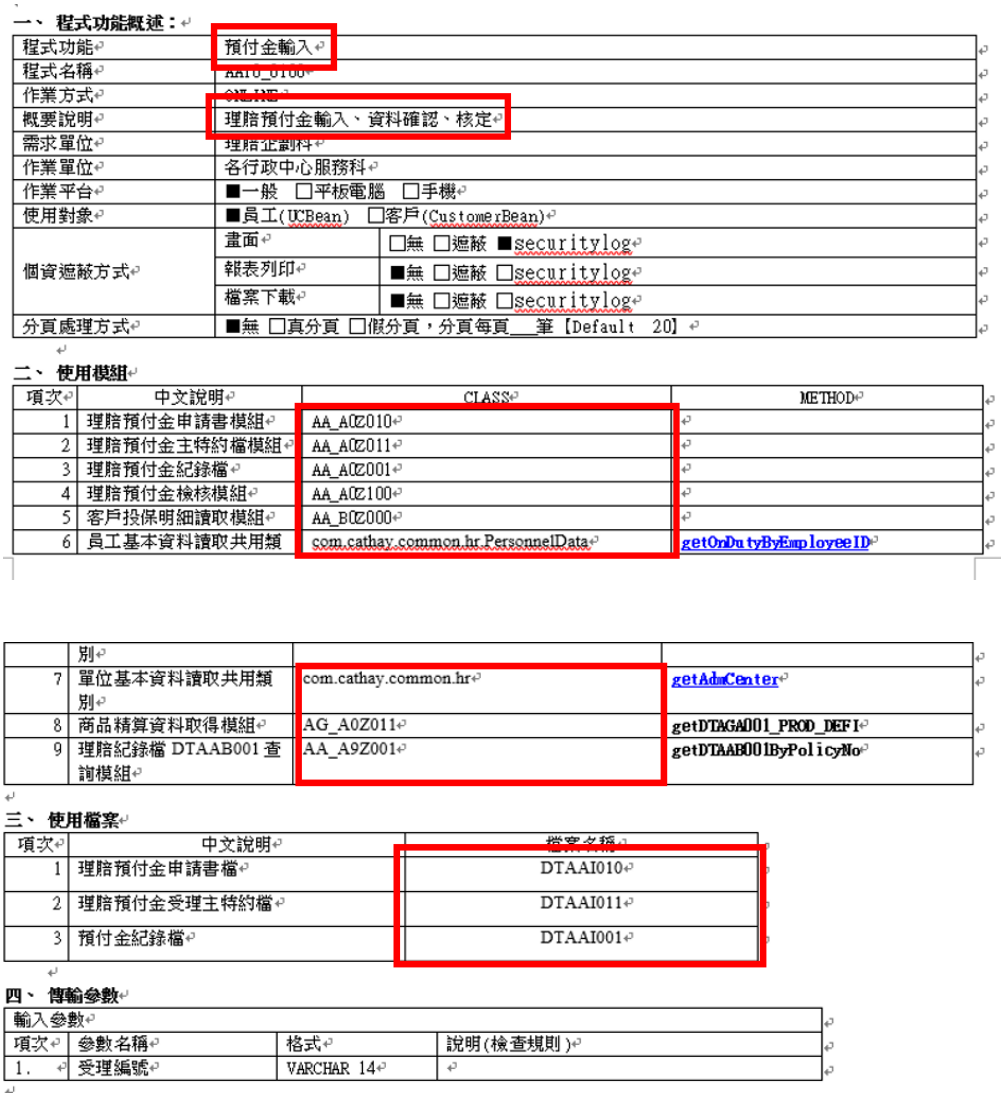
6. 測試 API

本專案架設完 Flask API 後，以 postman 進行測試。其中 headers 的 key 為 enctype，value 為 multipart/form-data；而 body 的 key 設定為 file，value 為使用者欲上傳的文件。設定為即可傳送測試，如下圖所示。



本測試上傳 UCAAI0_0100.docx，該 SPEC 的程式功能為預付金輸入，概要說明為理賠預付金輸入、資料確認、核定，該 SPEC 也使用到了許多

模組(AA_A0Z010、AA_A0Z011、AA_A0Z001、AA_A0Z100、AA_B0Z000、com.cathay.common.hr.PersonnelData、AG_A0Z011、com.cathay.common.hr、AA_A9Z001)，也使用到了 DTAAI010、DTAAI011、DTAAI001 三個資料表，內容如下圖所示。



一、 程式功能概述：

程式功能	預付金輸入	
程式名稱	AAI0_0100	
作業方式	線上	
概要說明	理賠預付金輸入、資料確認、核定	
需求單位	理賠企劃科	
作業單位	各行政中心服務科	
作業平台	■一般 □平板電腦 □手機	
使用對象	■員工(UCBean) □客戶(CustomerBean)	
個資遮蔽方式	畫面	□無 □遮蔽 ■securitylog
	報表列印	■無 □遮蔽 □securitylog
	檔案下載	■無 □遮蔽 □securitylog
分頁處理方式	■無 □真分頁 □假分頁，分頁每頁___筆【Default 20】	

二、 使用模組

項次	中文說明	CLASS	METHOD
1	理賠預付金申請書模組	AA_A0Z010	
2	理賠預付金主特約檔模組	AA_A0Z011	
3	理賠預付金紀錄檔	AA_A0Z001	
4	理賠預付金檢核模組	AA_A0Z100	
5	客戶投保明細讀取模組	AA_B0Z000	
6	員工基本資料讀取共用類別	com.cathay.common.hr.PersonnelData	getOnDutyByEmployeeID
7	單位基本資料讀取共用類別	com.cathay.common.hr	getAdmCenter
8	商品精算資料取得模組	AG_A0Z011	getDTAGA001_PROD_DEFI
9	理賠紀錄檔 DTAAB001 查詢模組	AA_A9Z001	getDTAAB001ByPolicyNo

三、 使用檔案

項次	中文說明	檔案名稱
1	理賠預付金申請書檔	DTAAI010
2	理賠預付金受理主特約檔	DTAAI011
3	預付金紀錄檔	DTAAI001

四、 傳輸參數

輸入參數			
項次	參數名稱	格式	說明(檢查規則)
1.	受理編號	VARCHAR 14	

而本專案接收到 UCAAI0_0100.docx 後，推薦了 UCAAI0_0700.docx 這篇 SPEC 給使用者。從檔名即可得知這兩篇 SPEC 都屬於同一個子系統，廣義上兩篇 SPEC 就有一定的相關性。而進行細看可以發現，推薦的 UCAAI0_0700.docx 的程式功能為預付金明細，概要說明為預付金明細，雖然未使用到任何模組，但推薦的 SPEC 也有使用上傳的 SPEC 的兩個資料表 DTAAI010 與 DTAAI001。

一、 程式功能概述：

程式功能	預付金明細	
程式名稱	AAI0_0700	
作業方式	ONLINE	
概要說明	預付金明細	
需求單位	理賠企劃科	
作業單位	各行政中心服務科	
作業平台	■一般 □平板電腦 □手機	
使用對象	■員工(UCBean) □客戶(CustomerBean)	
個資遮蔽方式	畫面	■無 □遮蔽 □securitylog
	報表列印	■無 □遮蔽 □securitylog
	檔案下載	■無 □遮蔽 □securitylog
分頁處理方式	■無 □真分頁 □假分頁，分頁每頁___筆【Default 20】	

二、 使用模組

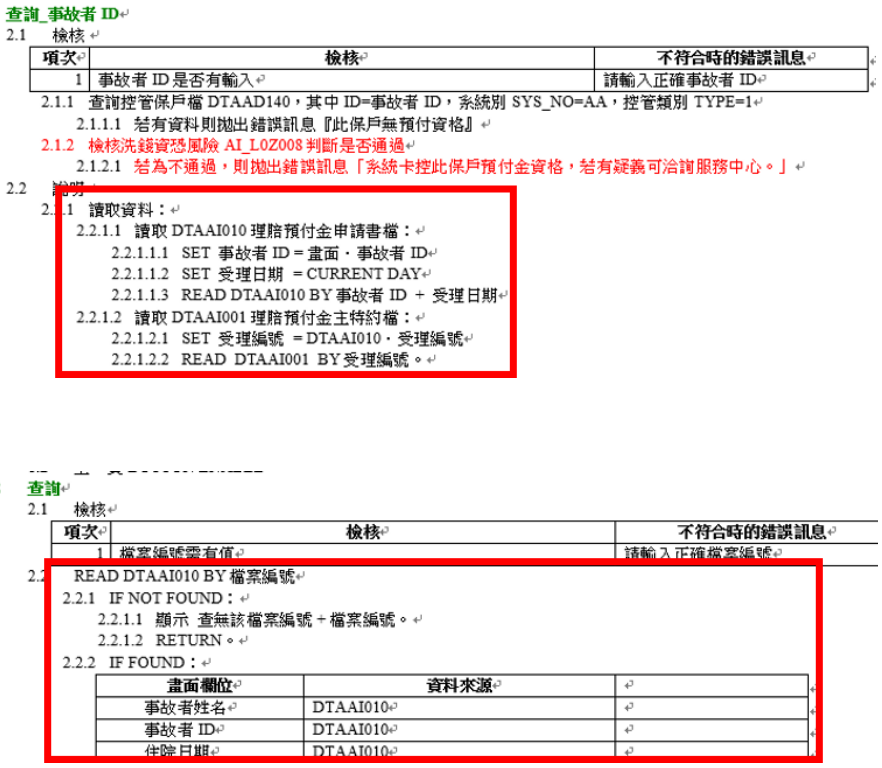
項次	中文說明	CLASS	METHOD
1.			

三、 使用檔案

項次	中文說明	檔案名稱
1	理賠預付金申請書檔	DTAAI010
2	理賠預付金給付紀錄檔	DTAAI001

四、 [illegible]

若再更進一步觀察，可以發現本專案測試時所上傳的 UCAAI0_0100.docx 內含有一個片段為讀取 DTAAI010 資料表的動作，而 UCAAI0_0700.docx 內也有一個片段是讀取 DTAAI010 資料表，因此可以推論本專案的 SPEC 文件推薦是有效的。如下圖所示(上圖為上傳檔案片段，下圖為推薦檔案片段)。



查詢_事故者 ID

2.1 檢核

項次	檢核	不符合時的錯誤訊息
1	事故者 ID 是否有輸入	請輸入正確事故者 ID

2.1.1 查詢控管保戶檔 DTAAD140，其中 ID=事故者 ID，系統別 SYS_NO=AA，控管類別 TYPE=1

2.1.1.1 若有資料則拋出錯誤訊息『此保戶無預付資格』

2.1.2 檢核洗錢資恐風險 AI_L0Z008 判斷是否通過

2.1.2.1 若為不通過，則拋出錯誤訊息「系統卡控此保戶預付金資格，若有疑義可洽詢服務中心。」

2.2 說明

2.2.1 讀取資料：

2.2.1.1 讀取 DTAAI010 理賠預付金申請書檔：

2.2.1.1.1 SET 事故者 ID = 畫面．事故者 ID

2.2.1.1.2 SET 受理日期 = CURRENT DAY

2.2.1.1.3 READ DTAAI010 BY 事故者 ID + 受理日期

2.2.1.2 讀取 DTAAI001 理賠預付金主特約檔：

2.2.1.2.1 SET 受理編號 = DTAAI010．受理編號

2.2.1.2.2 READ DTAAI001 BY 受理編號。

2 查詢

2.1 檢核

項次	檢核	不符合時的錯誤訊息
1	檔案編號需有值	請輸入正確檔案編號

2.2 READ DTAAI010 BY 檔案編號

2.2.1 IF NOT FOUND：

2.2.1.1 顯示 查無該檔案編號＋檔案編號。

2.2.1.2 RETURN。

2.2.2 IF FOUND：

畫面欄位	資料來源	
事故者姓名	DTAAI010	
事故者 ID	DTAAI010	
住院日期	DTAAI010	

7. 串接知識管理機器人

目前本專案已將 API 串接到科內的知識管理聊天機器人上，輸入”SPEC 推薦”後，機器人便會請使用者上傳文件，而使用者上傳文件之後，機器人便會呼叫本專案的 API 完成 SPEC 文件推薦的服務，回傳推薦的 SPEC 檔名。如下圖所示。

貳、 Spec 片段推薦

一、 介紹

在 SPEC 推薦的情境中，有時候 user 想要的回傳結果不一定是整篇 SPEC，舉例來說，如果 user 對 SPEC 某個模組的使用方法或是特定的業務邏輯不熟悉的話，相較於回傳整篇 SPEC 給 user，如果能回傳給 user 最相關的 SPEC 片段，更能幫助 user 抓住重點。

我們使用 BERT 模型作為 SPEC 片段推薦功能核心架構，然而 BERT 有最大輸入長度限制，輸入序列長度不得超過 512 個 token，在這樣的限制之下，為了利用 BERT 強大的自然語言理解能力，我們將 SPEC 切割成 BERT 可以接受的長度，進行 SPEC 的片段推薦。

二、 流程簡介

SPEC 片段推薦流程主要分為三個階段，分別是：

1. 資料預處理

將到手的 SPEC 文件從 word 檔解析、斷詞、SPEC 的切割，到轉換成 BERT 可以接受的格式的這一系列流程。

2. 模型訓練

載入模型後，設定模型參數、訓練參數並且進行模型的訓練。

3. 計算相似度

將所有 SPEC 片段丟入訓練好的 BERT 模型去產生每一個 SPEC 的向量，並且去計算這些向量之間的相似度，我們便可以去根據給定 SPEC 片段的向量去找出與之相似度最高的 SPEC 片段作為我們的推薦結果。

三、 實作細節

1. 資料預處理

(1) 解析 docx 文件

- 我們使用 python 的 docx 套件來解析 word 檔。
- 只擷取 SPEC 的文字部分，捨棄掉表格的部分。
- 解析過後的 SPEC 格式為許多 paragraph 所組成的 list，如下圖所示：

```
['程式功能概要說明：',
 '使用檔案',
 '輸出入參數',
 '異常訊息紀錄模組ErrorLog.java',
 '程式內容：',
 '初始： ',
 'IF 輸入參數 資料年月 = NULL',
 '資料年月 = 程式執行當時的上個月 (如:程式執行日期為,資料年月=201002)',
 'ELSE',
 '資料年月 = 輸入參數']
```

(2) 斷詞

- 我們使用 python 的 Jieba 套件來進行 SPEC 的斷詞。
- 斷詞用字典建立：

由於 SPEC 大量用到一些特殊的專有名詞，例如 table 欄位名稱、模組代號，因此，如果我們用 Jieba 幫 SPEC 進行斷詞的時候，Jieba 可能會沒有辦法辨別那些專有名詞，導致斷出來的詞不合乎邏輯。透過讓 Jieba 載入自己建立的字典，可以幫助我們斷詞斷得更乾淨俐落。

我們經由 toad，從 db2 上面撈出 table 欄位名稱及 table 代號來建立屬於專有名詞的字典。

(3) 形成訓練樣本

BERT 的 input 為三種格式的序列，分別是 token id、token type id、attention mask。

將 SPEC 中的每個 paragraph 斷詞後，我們將 paragraph 聚集成一個 SPEC 片段，每個片段的長度不超過 128 個 token。

Token id 為 token 經過字典的對照後轉換而成的 token id。一般來說，通常都會用 bert 的 tokenizer，透過 bert 本身的字典，去將每個 token 映射成 token id。但是我們發現，蘊藏較豐富 domain knowledge 的 SPEC 文件，如果透過自建立的字典將每個 token mapping 成 token id，模型效果看起來會比原來的好。我們自建立的字典來自每個 SPEC 斷詞過後的詞彙及 table 中文欄位名稱、table 英文欄位名稱、及每個模組的代號。此外，為了讓訓練樣本長度一致，長度不足 128 的部分以 0 來補齊(padding)。

Token type id 用來區別 input sequence。在上下句的配對任務中，因為輸入的 token sequence 有上下兩句之分，因此我們會將上句以 0 來表示，將下句以 1 來表示，讓 bert 模型去區分兩個不同的句子。在 SPEC 片段推薦中，token type id 我們全部以 0 來表示，因為在我們的 task 中，BERT 的輸入沒有上下句之分。

Attention mask 代表模型在訓練的時候，要去關注哪些

token。在 token id 序列中，長度不足的部分我們以 0 來補齊，在 attention mask 我們同樣以 0 來表示那些 padding 的部分，其他部分以 1 來表示，代表我們的模型要關注的這些 token。

2. 模型訓練

(1) 使用套件

- Pytorch

```
from  transformers  import  (
        CONFIG_MAPPING,
        BertModel,
        AutoModelWithLMHead,
        AutoTokenizer,
        DataCollatorForLanguageModeling,
        Trainer,
        TrainingArguments,
        BertTokenizer
)
```

- Hugging Face Transformers
- Sklearn

(2) 將訓練樣本轉換為 torch tensor 格式

因為我們使用了 transformers 的 trainer 模組進行訓練，因此只需要 token id 作為輸入，模組會自動生成對應的 token type id 跟 attention mask。

```
tensor([  101, 12544,   104,  8318,   104,  4542, 14064,  8708, 13876,   112,
         5219, 13876,   112, 12542,  9370, 12343,  6949,   108,   110, 12542,
         9375, 13876,   112,   525,   109,  3904, 13876,   112, 14064,  8708,
         3971,  4542, 12887,   112, 13876, 14642,   112, 13876,  8645,   108,
        13876,   112,   525, 14642,   112,   512,   109, 13814, 14711, 12167,
        12979, 12167,   108,  3323,   109,   104,   108, 11459,  9514,   109,
          102,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0])
```

(3) 設定模型訓練參數

- 載入 TrainingArguments 物件來進行訓練參數的設定。
- 參數：(Epochs = 30 , batch size = 20 , warmup steps = 500)

我們設定 epoch 為 30，代表每一個樣本會被反覆丟進模型訓練 30 次。

P.S. warmup steps 是一個訓練神經網路的技巧。learning rate 一開始會在設定的 warmup steps 之內慢慢遞增，當超過了設定的 warmup steps 之後，learning rate 便會開始遞減，warmup steps 可以防止模型在訓練一開始時 learning rate 過大，出現 overfitting 的問題。

```
training_args = TrainingArguments('./drive/My Drive/bert_spec')

training_args.do_train = True
training_args.num_train_epochs = 30
training_args.logging_dir = './drive/My Drive/bert_spec/logs'
training_args.per_device_train_batch_size = 20
#設定每幾個step將模型下載下來
training_args.save_steps=3000
training_args.warmup_steps = 500
training_args.device
```

(4) 設定 BERT 模型參數

- **載入 CONFIG_MAPPING object 來進行模型架構的參數設定**

因為的訓練樣本較少，總共只有 5137 份 SPEC 片段，不需要使用太複雜的模型去訓練。因此將 hidden layer 跟 attention heads 的數目減少，相較於基本的 bert 有 12 層的 hidden layer 及 12 個 attention heads，我們都將數量減少為 8，減少模型的參數數量來加快模型訓練的速度。

```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 8,
  "num_hidden_layers": 8,
  "pad_token_id": 0,
  "type_vocab_size": 1,
  "vocab_size": 30522
}
```

(5) 載入 BERT 模型與 BERT TOKENIZER

- **使用的 BERT model 為 Transformers 的 AutoModelWithLMHead 模型**

  這是用來進行 BERT 預訓練所使用的模型，我們會將輸入的 token sequence 隨機進行遮蔽，模型的最上層會加一層 multi-layer classifier 來預測訓練集之中被隨機遮蔽的 token。透過這種訓練，我們能藉由無監督式的方式來學習句子的語義資訊。

- **載入 BERT-BASE-CHINESE TOKENIZER**

  Tokenizer 在這裡的用意只是做為 data collator 模組的參數傳入，其用途是將 pad token([PAD])跟 mask token([MASK]) mapping 為各自的 token id，因此用什麼 tokenizer 都可以。

(6) 設定 data collator

```
data_collator = DataCollatorForLanguageModeling(
                    tokenizer=tokenizer, mlm=True, mlm_probability=0.15
                )
```

Data collator 模組的功能是將樣本整理成一個 batch，並以 tensor 的 dictionary 呈現，如下圖：

```
data_collator(input_tensor_samples)
```

```
{'input_ids': tensor([[  101, 12544,   103,  ...,     0,     0,     0],
        [  101,  5886,   103,  ...,     0,     0,     0],
        [  101,   103,  2758,  ...,     0,     0,     0],
        ...,
        [  101, 11193,  7410,  ...,     0,     0,     0],
        [  101, 11193,  7410,  ...,     0,     0,     0],
        [  101, 12542,  8529,  ...,     0,     0,     0]]),
 'labels': tensor([[-100, -100,  104,  ..., -100, -100, -100],
        [-100, -100, 2758,  ..., -100, -100, -100],
        [-100, 5886, -100,  ..., -100, -100, -100],
        ...,
        [-100, -100, -100,  ..., -100, -100, -100],
        [-100, -100, -100,  ..., -100, -100, -100],
        [-100, -100, -100,  ..., -100, -100, -100]])}
```

我們所使用的 DataCollatorForLanguageModeling，可以隨機遮蔽訓練樣本的字詞(15%)，進行遮蔽式語言建模(masked language modeling)。

下圖是 DataCollatorForLanguageModeling 產生的一個樣本的 label，圖中不是-100 的 token 為模型在訓練時被遮蔽的 token，而 masked language modeling 的目標就是預測這些被遮蔽的 token。

```
tensor([ -100, 12544,  -100,  -100,   104,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100, 12542,
         9375,  -100,   112,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100, 13876,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
        12979, 12167,   108,  3323,   109,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100])
```

(7) 開始訓練

- **引入 transformers trainer 套件，並訓練模型。**

```
trainer = Trainer(
                model=bert_model,
                args = training_args,
                data_collator=data_collator,
                train_dataset=input_tensor_samples,
                eval_dataset=None,
                prediction_loss_only=True,
        )
```

- 訓練完畢後將模型儲存下來。

```
model_path = './drive/My Drive/bert_spec/'
trainer.train(model_path=model_path)
trainer.save_model()
```

3. 相似度計算&推薦結果

(1) 載入訓練好的模型

將模型及訓練資料丟進 GPU 運算，可以提升向量產生的速度。

```
embedding_model = BertModel.from_pretrained('./drive/My Drive/bert_spec/checkpoint-6000/')
embedding_model.to('cuda')
```

(2) 將樣本丟入訓練好的模型產生向量(hidden vector)

- **我們取的向量是 pooled hidden vector。**

  根據文件(https://modelzoo.co/model/pytorch-pretrained-bert)，bert，輸出兩種 output， 第一種是模型最上層 output 的所有 hidden state(vector)，第二種是 pooled hidden state(vector)。

This model *outputs* a tuple composed of:

- `encoded_layers`: controled by the value of the `output_encoded_layers` argument:
  - `output_all_encoded_layers=True`: outputs a list of the encoded-hidden-states at the end of each attention block (i.e. 12 full sequences for BERT-base, 24 for BERT-large), each encoded-hidden-state is a torch.FloatTensor of size [batch_size, sequence_length, hidden_size],
  - `output_all_encoded_layers=False`: outputs only the encoded-hidden-states corresponding to the last attention block, i.e. a single torch.FloatTensor of size [batch_size, sequence_length, hidden_size],
- `pooled_output`: a torch.FloatTensor of size [batch_size, hidden_size] which is the output of a classifier pretrained on top of the hidden state associated to the first character of the input (`CLF`) to train on the Next-Sentence task (see BERT's paper).

- **我們從模型的 output 中取三種向量：**

  i. Pooled hidden vector :

  模型最上層第一個位置([CLS]的輸出位置)輸出的

hidden vector。

ii. Mean pooled hidden vector :

將模型最上層所有位置輸出的 hidden vector 做 mean pooling 所得到的 hidden vector。

iii. Max pooled hidden vector :

將模型最上層所有位置輸出的 hidden vector 做 max pooling 所得到的 hidden vector。

(3) 計算向量之間的 cosine similarity

使用 sklearn 的 cosine similarity 模組來計算向量之間的相似度(cosine similarity)並建立相似度矩陣(pandas dataframe)，以方便直接存取片段間的相似度。

	0	1	2	3	4
0	1.000000	0.645733	0.629186	0.650327	0.731934
1	0.645733	1.000000	0.883826	0.896967	0.804413
2	0.629186	0.883826	1.000000	0.987467	0.776836
3	0.650327	0.896967	0.987467	1.000000	0.775908
4	0.731934	0.804413	0.776836	0.775908	1.000000
...	...	...	...	...	...

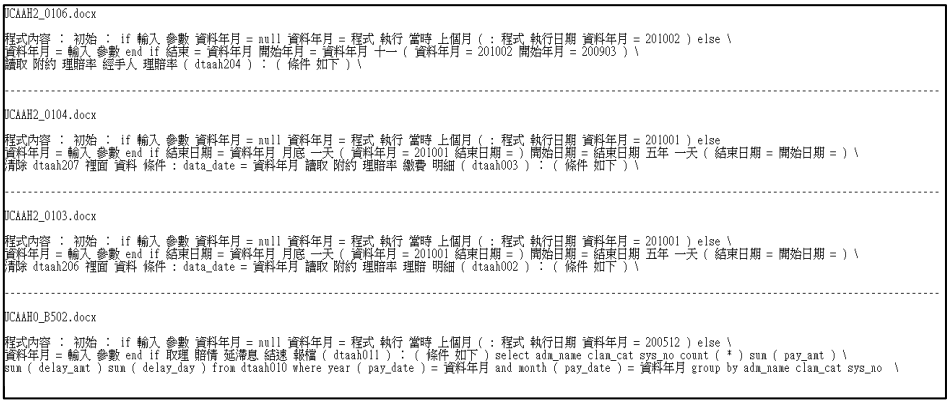
(4) 回傳 SPEC 片段推薦結果

給定一個 SPEC 片段，片段進來之後會經過以下處理：

- 用 jieba 搭配字典進行斷詞。
- 將斷詞後的 token sequence 轉換成 token id、token type id、attention mask。
- 丟進去 bert model 產生 output。
- 轉換成向量，三種可選：
  - i. Pooled hidden vector
  - ii. Max pooled hidden vector
  - iii. Mean pooled hidden vector

將給定 SPEC 片段向量去與其他 SPEC 片段向量計算相似度：

- 跑迴圈逐一與其他 SPEC 片段向量計算相似度。
- 回傳相似度最高的前 n 筆結果。



4. 相關檔案

- 斷詞用字典
- 字典(用來將 token 映射成 token id)
- SPEC 文件(共 786 份)