Introduction-to-Information-Retrieval-and-Text-Mining, HW 3

B07302230, 財金三 王博奕

1. 執行環境：Visual Studio Code
2. 程式語言：Python 3.8
3. 執行方式：
   A. Pip install nltk 以便於接下來使用 Porter's algorithm 及 stopword。
   B. Pip install math 以進行 log 運算。
   C. Pip install re 以便切字。
4. 作業處理邏輯說明：

首先，tokenize 的函數沿用作業二的部分，因此不再贅述。接下來程式主要分成四大部分進行，以下將分點描述。最後再提出改進的地方。

A. 處理 Training data:

載入作業提供的訓練資料，存為 13*15 的二維 list(train)，第一層為 class、第二層為該 class 底下的文件。此處我另外建一個 training_doc_id 的 list，專門存訓練資料的號碼，以便後面排除。

```
# Load training data
train = []  # 13*15
training_doc_id = []
f = open("C:\\Users\\asdfg\\OneDrive - g.ntu.edu.tw\\NTU\\109-1\\1
words = f.read().splitlines()
for i in range(13):
    temp = words[i].strip().split(" ")
    temp.pop(0)
    train.append(temp)
for c in train:
    for docID in c:
        training_doc_id.append(int(docID))
```

接下來我建一個三維的 list(class_doc_token)，第一層為 class、第二層為 class 底下的文件、第三層為各文件的 token。這裡我先用 temp 來執行 tokenize，再轉為 token 是因為當初 tokenize 寫法的關係，原先的的 temp 是二維的，經降維成 token。我在這裡也先建 terms 這字典，是先存訓練資料的 token，屆時這個字典的 value 將儲存 LLR 算出的各個分數。

```
# Store info of training data
class_doc_token = []
terms = {}
for c in train:
    doc_token = []
    for docID in c:
        # Load the text of training doc
        f = open("C:\\Users\\asdfg\\OneDrive\\桌面\\IRTM\\IRTM\\"+str(docID)+".txt"
        text = f.read()
        # Tokenize the training doc
        temp = tokenize(text)  # 2D-list: [['a','b',...]]
        token = []
        for j in temp[0]:
            token.append(j)    # 1D-list: ['a','b',...]
        # create a dict for each term
        for k in token:
            if k not in terms.keys():
                terms[k] = []
        doc_token.append(token)
    class_doc_token.append(doc_token)
```

B. Log Likelihood Ratios:

這次作業我使用的 Feature Selection 的方法為 LLR，計算方法即為上課 ppt 所記載的。這裡的目標為選取 LLR 分數高的 term。而 terms 這字典的 value 為各個 class 其 LLR 的分數(一個有 13 個 elements 的一維 list)。

```
# LLR
for term in terms.keys():
    for classnum1 in range(len(class_doc_token)):
        n11 = 0  # on topic & present
        n10 = 0  # on topic & absent
        n01 = 0  # off topic & present
        n00 = 0  # off topic & absent
        for classnum2 in range(len(class_doc_token)):
            # on topic
            if classnum2 == classnum1:
                for token in class_doc_token[classnum1]:
                    if term in token:   # present
                        n11 += 1
                    else:   # absent
                        n10 += 1
            # off topic
            else:
                for token in class_doc_token[classnum2]:
                    if term in token:   # present
                        n01 += 1
                    else:   # absent
                        n00 += 1

        n = n11+n10+n01+n00
        # hypothesis 1(the prob. of the occurence of the term & the class is indep
        pt = (n11+n01)/n    # on/off topic & present
        # hypothesis 2(the prob. of the occurence of the term & the class is deper
        p1 = n11/(n11+n10)  # on topic & present
        p2 = n01/(n01+n00)  # off topic & absent
        LH1 = (pt**n11)*((1-pt)**n10)*(pt**n01)*((1-pt)**n00)
        LH2 = (p1**n11)*((1-p1)**n10)*(p2**n01)*((1-p2)**n00)
        LLR = (-2)*math.log10(LH1/LH2)
        terms[term].append(LLR)
```

C. 篩選 500 個最有力的 feature term:

原先的 terms 這個 dict 的 value 是個 list，包含了這個 term under different classes 時不同的分數，而我這裡直接把 list 裡的 element 直接取平均算出一個平均 LLR，最後挑出前 500 個高分的 feature term，作為篩選 term，這時的 feature_term 這個 dict 只剩 500 個 key。有了 feature_term，接著就把原先的 class_doc_token 這個 list 中的 token 部分只留下 feature term。

```
# build the new dict to store LLR of each term
# use average LLR to select feature term
terms_scored = {}
for term in terms.keys():
    terms_scored[term] = sum(terms[term])/len(terms[term])

count = 0
feature_term = {}
terms_scored = sorted(terms_scored, key=terms_scored.get, reverse=True)  # from big to small
for term in terms_scored:
    if count < 500:  # required
        count += 1
        feature_term[term] = []

# store only feature term
for i in range(len(class_doc_token)):
    for j in range(len(class_doc_token[i])):
        for k in range(len(class_doc_token[i][j])):
            if class_doc_token[i][j][k] not in feature_term.keys():
                class_doc_token[i][j][k] = ""
        while "" in class_doc_token[i][j]:
            class_doc_token[i][j].remove("")
```

D. Multi-Nominal Naïve Bayes Classification:

i. Training phase:

這裡就將利用老師上課 ppt 所提供的 Pseudocode 來完成，值得一提的事是因為 training data 提供的樣式關係，可以直接知道 prior 是多少。另外，我也將各個詞的條件機率加到 feature_term 這個字典的 value。

```
# Multi-Nominal NB Classification - Training Phase
voc = 500  # extract vocabulary
prior = 1/13
for term in feature_term.keys():
    for c in class_doc_token:
        alldoc_num = 0
        term_appear = 0
        for doc in c:
            alldoc_num += len(doc)   # concatenate text of all docs in class
            for token in doc:
                if term == token:
                    term_appear += 1
        condprob = (term_appear+1)/(alldoc_num+voc)
        feature_term[term].append(condprob)
```

ii. **Testing phase:**

最後將每一篇的文章先 tokenize 後，再來針對 feature term 來計分，最後選大的來當作該篇為哪個 class 的依據。

```
# Multi-Nominal NB Classification - Training Phase
ans = []
for docID in range(1, 1096):
    if docID not in training_doc_id:
        c_map = [math.log10(prior)]*13
        f = open("C:\\Users\\asdfg\\OneDrive\\桌面\\IRTM\\IRTM\\"+str(docID)+".txt"
        text = f.read()
        temp = tokenize(text)
        token = []
        for i in temp[0]:
            token.append(i)
        for term in token:
            if term in feature_term.keys():
                for x in range(13):
                    c_map[x] += math.log10(feature_term[term][x])
        ans.append(docID)
        ans.append(c_map.index(max(c_map))+1)
```

E. **Improvement:**

結束了以上程式碼，我在最後再試圖調整了模型，其中我發現收太多 feature term 會導致分數降低(可能原因是有雜訊)，因此我把 feature term 選取數量降低為一半，也就是 250 個，發現在 kaggle 上的成績比較好。