IRTM HW4 Report

R06725035 資管碩二 陳廷易

執行環境

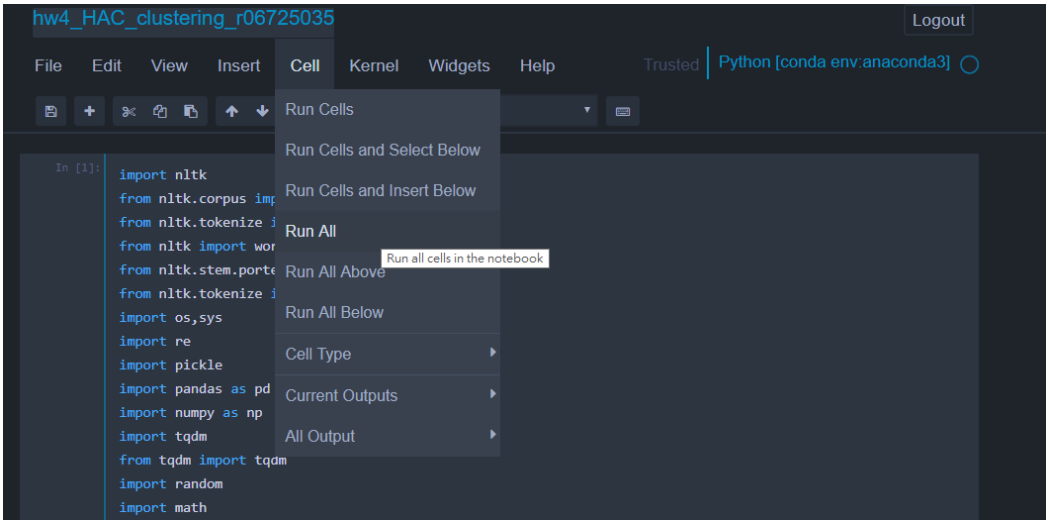
- Ubuntu 16.04.5 LTS
- Jupyter Notebook

程式語言

- Linux Anaconda 5 Python 3.6.6

執行方式

- 提供 jupyter notebook: *hw4_HAC_clustering_r06725035.ipynb*
  - 可利用 jupyter 開啟 ipynb 以後，選擇上方工具列 Cell=>Run All 即可，同時也提供.py 檔案供助教測試



- 需 pip install:
  - Python 內建既有套件: random、math、os、sys、heapq(實做 bonus)、time(計算時間差異)
  - pickle、pandas、numpy
  - tqdm(顯示當前進度)

- 確保 hw2 所計算的各 document normalized tf-idf vector .txt 放於 data/tf-idf/ 目錄下: X.txt
- data/目錄下需要有 stop_words.txt 及 stop_list.pkl 來製作 stop words list
- data/目錄下存放 hw3 所給定的 training.txt 及當時分類結果 voting_rev.csv 來輔助修正 HAC 分群結果
- data/目錄下需要有 C.pkl 與 voting_lis.pkl 中間站存檔以加速程式運行
- 三種分群結果放於 output/目錄下: K.txt (windows 記事本開啟若遇到編碼問題，可能需改用其他文字編輯器如:notepad++或是 vim 開啟)

作業邏輯說明

1. 定義 Cosine Similarity 函數：計算 pair-wise document 的相似度

```
def cosine(DOCx, DOCy):
    '''
    input: doc1 path name(str) , doc2 path name(str)
    ouput: two doc's cosine similarity
    '''
    dfx = pd.read_csv(DOCx,sep=' ',names=['tindex','tfidf'],header=None)
    dfy = pd.read_csv(DOCy,sep=' ',names=['tindex','tfidf'],header=None)
    dfx = dfx.drop(0)
    dfy = dfy.drop(0)
    dfxy = pd.merge(dfx,dfy,on='tindex',how='outer')
    dfxy.fillna(0,inplace=True)
    up = sum(dfxy.tfidf_x * dfxy.tfidf_y)
    down = np.sqrt(sum(np.square(dfxy.tfidf_x)))* np.sqrt(sum(np.square(dfxy.tfidf_y)))
    result = up / down
    return result
```

2. 定義群間相似度計算函式：因採用很多種不同計算方法來投票各 HAC 的最終結果，因此這邊有四種計算方式，於訓練時需依情況進行調整。1. a 為 single link 計算方法；2. b 為 complete link 計算方法；3. avg1 為 a 與 b 的幾何平均數；4. avg2 為 a 與 b 的算術平均數。

similarity measure between clusters

```
def merge_sim(C, j, i, m):
    a = np.max([C[j][i], C[j][m]]) #single link
    b = np.min([C[j][i], C[j][m]]) # complete link
    return (a+b)/2 # max: single ，min:complete
```

3. 讀入作業二所輸出的 normalized tf-idf vector，計算兩篇文章相似度可直接使用 cosine similarity 來獲得 pair-wise 的 document similarity 並存放於 C

Cosine similarity for pair-wise document similarity

```
N=1095
C=np.zeros([N,N])
I=np.ones((N,), dtype=int)
eps=1e-10
for i in tqdm(range(N)):
    for j in range(N-i-1):
        cos = cosine('data/tf-idf/'+str(i+1)+'.txt','data/tf-idf/'+str(j+i+2)+'.txt')
        C[i][j+i+1]=C[j+i+1][i]=cos+eps
print(C.shape)
pickle.dump(obj=C,file=open('data/C.pkl','wb'))
```

4. 實作 HEAP EfficiencyHAC

甲、依照講義 priority-queue 進行初始化：C 用以記錄各群集的相似度以及所對應 index；I 用以代表該群集是否尚未被合併(若被合併者值為 0)；A 儲存 merges，一開始為空；P 為 priority queue，最大相似度者會放於根節點，會儲存經過 heapify 的群集相似度(C[n])且不儲存 self-similarities。

```python
for n in range(N):
    c = []
    for i in range(N):
        the_sim = -C_[n][i]
        c.append([the_sim, i])
    C.append(c)
    I[n] = 1
    C_list = sorted(C[n])
    C_list.remove(C[n][n])
    P.append(C_list)
    heapq.heapify(P[n])
    cluster_representations.append([n])
heap_A = [] # list of merges
```

```
EfficientHAC (d1, ..., dN)
  for n ← 1 to N
  do for i ← 1 to N
     do C[n][i].sim ← dn·di
        C[n][i].index ← i
     I[n] ← 1
     P[n] ← priority queue for C[n] sorted on sim
     P[n].Delete(C[n][n]) // don't want self-similarities
  A ← []
```

C[i][j]: the similarity between clusters *i* and *j*.

I: indicate which clusters are still available to be merged.

A: a list of merges

***P*: an array of priority queue**

乙、進行 N(文章數量)-1 次 iteration：先從 P 各 priority queue 中找到 similarity 最大值，從這些根節點中再比對出最大的。接下來再將 k1、k2 放入 A 中，並使 k2 之 I 設為 0(已併至 k1)。接著要更新各群集之於 k1 的相似度，因此先將各群原本對 k1 及 k2 的相似度刪除，重新計算群間相似度(上述 2.有四種)，將各群對於 k1 及 k1 對於各群的相似度重新置入 priority queue 中。

```python
for k in tqdm(range(N - 1)):
    min_neg_sim = 0
    min_i = 0
    min_m = 0
    for i in range(N):
        if I[i] == 1 and P[i][0][0] <= min_neg_sim: #similarity
            min_i = i
            min_neg_sim = P[i][0][0]
    min_m = P[min_i][0][1]
    k2, k1 = sorted([min_i, min_m])
    heap_A.append((k1, k2))
    I[k2] = 0 #k2併去k1
    P[k1] = []
    for i in range(N):
        if I[i] == 1 and i != k1: #for each i with
            P[i].remove(C[i][k1])
            P[i].remove(C[i][k2])
            heapq.heapify(P[i])
            the_sim = heap_merge_sim(C, i, k1, k2) #sim
            C[i][k1][0] = the_sim
            heapq.heappush(P[i], C[i][k1])
            C[k1][i][0] = the_sim
            heapq.heappush(P[k1], C[k1][i])
    cluster_representations[k1] += cluster_representations[k2]
    cluster_representations[k2] = None
```

```
for k ← 1 to N-1                                          O(N)
  do k1 ← argmax_{k:I[k]=1} P[k].Max().sim
     k2 ← P[k1].Max().index
     A.Append(<k1, k2>)
     I[k2] ← 0
     P[k1] ← {}
     for each i with I[i]=1 and i≠k1                      O(N)
     do P[i].Delete(C[i][k1])
        P[i].Delete(C[i][k2])                             O(logN)
        C[i][k1].sim ← Sim(i, k1, k2)
        P[i].Insert(C[i][k1])                             O(N²logN)
        C[k1][i].sim ← Sim(i, k1, k2)
        P[k1].Insert(C[i][k1])
return A
```

丙、判斷目前還活著群集個數是否為所要求者(K=8、13、20)，若是則會將當前結果儲存下來。

```python
if np.sum(I) in Ks:
    the_cluster_result = sorted([sorted(cluster) for cluster
                                 in cluster_representations if cluster is not None])
    heap_cluster_results.append(the_cluster_result)
```

丁、我也有實作原本的 simple HAC 部分，經時間比較後 HEAP HAC 所需時間僅需原本的三分之一。

5. 因我採用了多種不同的群間相似度計算方法，因此所得各不相同，而為了能進行投票得到最後結果，將依據各模型分群結果好壞(用人工方式判斷分布平均度，普遍來說 complete link 表現較 single link 好)來進行權重投票。而為了使各方法能互相對齊投票，將會先照各方法中各群的長度來排序(如:第一群的個數最少，最後一群個數最多)。

甲、在分八群時，主要是利用三種不同的 complete link 來投票(差別在於 term dictionary 取的個數不同)

乙、在分十三群時，因為等同於分類問題，因此也會將作業三的結果拿進來投票(占一半權重)，其他方法(complete link、single link、幾何平均、算術平均)則占另一半權重

丙、在分二十群時，則主要是以其中兩種 complete link 來投票

```
for i,voting_li in enumerate(voting_lis):
    pd8 = pd.DataFrame(0,index=range(1095),columns=range(8))
    pd13 = pd.DataFrame(0,index=range(1095),columns=range(13))
    pd20 = pd.DataFrame(0,index=range(1095),columns=range(20))
    for method in voting_li : #20,13,8
        sort20 = sorted(method[0],key=len)
        sort13 = sorted(method[1],key=len)
        sort8 = sorted(method[2],key=len)
        for c,cluster in enumerate(sort8): #8群
            for doc in cluster: #各群裡面成員
                pd8.loc[doc,c] +=1
        for c,cluster in enumerate(sort13):
            for doc in cluster:
                pd13.loc[doc,c] +=1
        for c,cluster in enumerate(sort20): #8群
            for doc in cluster: #各群裡面成員
                pd20.loc[doc,c] +=1
    sort13 = sorted(clf13,key=len)
    for c,cluster in enumerate(sort13):
        for doc in cluster:
            pd13.loc[doc,c] += int(len(voting_li)/1.3)
    if i ==0:
        df20 = pd20
        print(len(voting_li))
    elif i ==1:
        df13 = pd13
        print(len(voting_li))
    elif i == 2:
        df8 = pd8
        print(len(voting_li))
```

丁、因為投票常常會有平手情況，因此會特別檢查各篇文章是否最高票皆僅屬於其中一群，以避免分群偏誤

```
for df in [df8,df13,df20]:
    df['vote'] = 0
    for i in range(len(df)):
        df.loc[i,'vote'] = df.loc[i].idxmax()
```

```
counts = 0
counts_minimize = 0
for pd_ in [df8,df13,df20]:
    cols = pd_.columns.tolist()[:-1]
    count = 0
    count_minimize = 0
    for col in cols:
        count += len(pd_[pd_[col]>int(len(voting_li)/2 )])
    pd_2 = pd_.drop(['vote'],axis=1)
    for i in range(len(pd_2)):
        count_minimize += pd_2.loc[i].isin([pd_2.loc[i].max()]).sum()

    print(count,count_minimize)
    counts+=count
    counts_minimize += count_minimize
print("ALL:",counts,'Minimize:',counts_minimize)
```

6. 最後再將結果依照 K 寫入 txt 輸出至 output/資料夾下

```
for df in [df20,df13,df8]:
    clus = df.columns.tolist()[:-1]
    f = open('output/'+str(len(clus))+'.txt','w')
    for clu_num in clus:
        for doc_id in df[df['vote']==clu_num].index.tolist():
            f.write(str(doc_id+1))
            f.write('\n')
        f.write('\n')
    f.close()
```