

1. 執行環境：Eclipse 2019-06

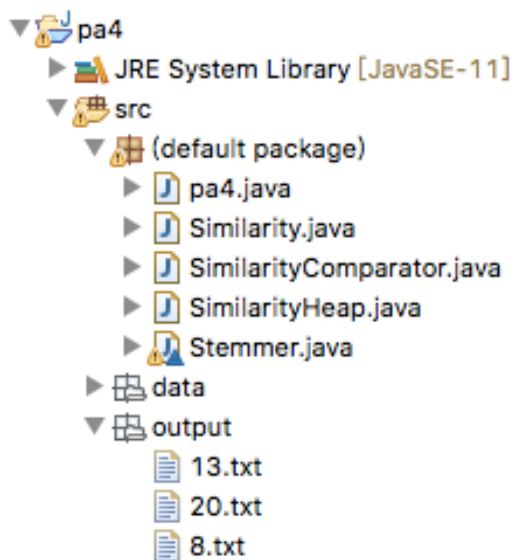
2. 程式語言：Java (version: 1.8.0\_101)

3. 執行方式：（\* 有提供助教整個 pa4 java project）

- (1) 新增名為 pa4 的 java project，在 src 下建立新的 package 及 pa4 class (pa4.java)
- (2) 在 src 之下新增名為“data”的資料夾，將下載的 1095 個文件複製進去，供後續讀取檔案用
- (3) 在 src 之下新增名為“output”的資料夾，供輸出最終的分群結果
- (4) 在 default package 新增一個名為“Stemmer”的 class，貼入 Porter's Stemmer 的 code（來源：<https://tartarus.org/martin/PorterStemmer/java.txt>）（同 PA2）
- (5) 有自建 heap 來取 maximal similarity！在 default package 新增名為“Similarity”、  
“SimilarityComparator”及“SimilarityHeap”的 class，後面「作業處理邏輯說明」會再做解釋！
- (6) 為了讀取及寫入 txt 檔案，有 import [java.io](#) 的部分套件（同 PA1）
- (7) 執行 pa4.java 後會在 output 的資料夾中建立 3 個 txt 檔，並將分群結果寫入，全部執行完共需  
花 10 到 11 分鐘！

\* 因為 pa4.java 要在 project 中執行，所以 data 和 output 的路徑設定為 src/data 及 src/output

\* 架構如下：



## 4. 作業處理邏輯說明：

## (1) 建立 dictionary (同 PA2)

a. 取出單篇文章的 tokens 並做前處理，寫在 docTokens(int docID) 的 method 中，不再贅述。

Code
<pre>public static void docTokens(int docID) throws Exception {     ... }</pre>

b. 用 for 迴圈將第1到第1095個文件帶入 docTokens(int docID) 的 method，將所有文章的 terms 及對應的 docID 都存到 tokensArrayList 中，並 import java.util.Collections 及 java.util.Comparator 套件來覆寫 sorting 方式，將 tokensArrayList 依 terms 的字母做排序。

Code
<pre>/* 1. Dictionary */ // (1) terms &amp; docID tokensArrayList = new ArrayList&lt;ArrayList&lt;String&gt;&gt;(); for (int docID = 1; docID &lt;= docSize; docID++) {     docTokens(docID); } Collections.sort(tokensArrayList, new Comparator&lt;ArrayList&lt;String&gt;&gt;() {     @Override     public int compare(ArrayList&lt;String&gt; o1, ArrayList&lt;String&gt; o2) {         return o1.get(0).compareTo(o2.get(0));     } });</pre>

c. 同篇文章重複的 term 做合併

建立一個名為 mergedTokensList 的 ArrayList<ArrayList<String>>，用 for 迴圈將 tokensArrayList 中每一個 element 取出來和前一個做比較，若兩者 term 和 docID 都不相同，就將這個 element 加進 mergedTokensList（移除同篇文章重複 term 的概念）。

Code
<pre>// (2) merge same term from the same document mergedTokensList = new ArrayList&lt;ArrayList&lt;String&gt;&gt;(); mergedTokensList.add(tokensArrayList.get(0)); for(int i = 1; i &lt; tokensArrayList.size(); i++) {     int pre = i-1;     if(!tokensArrayList.get(i).equals(tokensArrayList.get(pre))) {         mergedTokensList.add(tokensArrayList.get(i));     } }</pre>

d. 建立 dictionary，記錄 t\_index、term 和 df

建立一個名為 irtm\_dict 的 ArrayList<ArrayList<String>> 來儲存 dictionary 的內容。用 for 迴圈將 mergedTokensList 每一個 element 取出來和前一個做比較，如果該 element 的 term 和前一個不相同，就將該 term 加入 irtm\_dict，並給予 df=1；如果 element 的 term 和前一個相同，表示同個 term 出現在不同文章，則 df 要加 1（用自建的變數 count 來計算），更新 irtm\_dict 中該 term 的 df（因為 mergedTokensList 是有按照字母順序 sort 過的，所以同個字會排在一起，且 irtm\_dict 中的 element 是一個一個加進去，該 term 在 irtm\_dict 中的 index 會是當下的最後一個）。t\_index 有另外建一個變數來記錄。

## Code

```
// (3) t_index & term & df
irtm_dict = new ArrayList<ArrayList<String>>();
int t_index = 1;
ArrayList<String> firstEle = new ArrayList<String>();
firstEle.add(Integer.toString(t_index));
firstEle.add(mergedTokensList.get(0).get(0));
firstEle.add("1");
irtm_dict.add(firstEle);
for(int i = 1; i < mergedTokensList.size(); i++) {
    int pre = i-1;
    ArrayList<String> element = new ArrayList<String>();
    if(!mergedTokensList.get(i).get(0).equals(mergedTokensList.get(pre).get(0))) {
        t_index++;
        element.add(Integer.toString(t_index));
        element.add(mergedTokensList.get(i).get(0));
        element.add("1");
        irtm_dict.add(element);
    } else {
        int count = Integer.parseInt(irtm_dict.get(irtm_dict.size()-1).get(2));
        count++;
        irtm_dict.get(irtm_dict.size()-1).set(2, Integer.toString(count));
    }
}
}
```

**(2) 將文章轉為 tf-idf unit vector (PA2 的延伸)**

## a. main method

首先計算各個 term 的 idf。建立一個名為 index\_idf 的 ArrayList<ArrayList<String>> 來儲存各個 term 的 index 及 idf。用 for 迴圈將 irtm\_dict (有 t\_index, term, df) 的每一個 element 取出，將 element 的 t\_index 直接加入暫存的變數 ArrayList<String> element 中，再由 irtm\_dict 中的 df 去計算出 idf 值後加入 element，最後再把 element 加入 index\_idf，重複執行 irtm\_dict.size() 次。

接著建立名為 tfidfList 的三維 ArrayList，用來儲存所有文章的 tf-idf，避免後續重複使用還要重新計算。將單篇文章 tf、tf-idf、tf-idf unit vector 的計算寫成 method doc\_tfidf(docID)，此 method 可以回傳整理好的 ArrayList<ArrayList<String>>，再將此二維 ArrayList 存進 tfidfList 中，重複執行 1095 次。

## Code

```
/* 2. Transfer each document into a tf-idf unit vector */
// (1) idf
index_idf = new ArrayList<ArrayList<String>>();
for(int i = 0; i < irtm_dict.size(); i++) {
    ArrayList<String> element = new ArrayList<String>();
    element.add(irtm_dict.get(i).get(0));
    element.add(Double.toString(Math.log10(docSize/Integer.parseInt(irtm_dict.get(i).get(2)))));
    index_idf.add(element);
}

// (2) Build tf-idf Lists (size = 1095)
tfidfList = new ArrayList<ArrayList<ArrayList<String>>>();
for (int i = 1; i <= docSize; i++) {
    tfidfList.add(doc_tfidf(i));
}
System.out.println("Tf-idf lists done!");
```

## b. doc\_tfidf(int docID) (同 PA2，僅以文字簡述)

首先計算單篇文章各個 term 的出現次數，接著計算 term frequency (某個 term 的出現次數除以所有 term 出現次數的加總值)，再透過字典找出各個 term 所對應的 t\_index，以及前面建立的 index\_idf 找出對應的 idf，計算出單篇文章各個 term 的 tf-idf。由於要將 tf-idf 轉換成 unit vector，所以先計算單篇文章的 vector length，再將剛才所有的 tf-idf 值除以 vector length，最後將 t\_index 及 tf-idf unit vector 存成 ArrayList<ArrayList<String>> 並回傳給呼叫方。

## (3) 計算 cosine similarity 的 method (PA2 的延伸)

參數為兩篇文章各個 term 的 tf-idf unit vector，利用雙迴圈 for 找出兩篇文章所有相同的 term (t\_index 相同) 做內積，最後會回傳 similarity。

## Code

```
/* Method cosine(Docx, Docy) */
public static double cosine(ArrayList<ArrayList<String>> x, ArrayList<ArrayList<String>> y) {
    double similarity = 0;
    for(int i = 0; i < x.size(); i++) {
        ArrayList<String> xi = x.get(i);
        for(int j = 0; j < y.size(); j++) {
            ArrayList<String> yj = y.get(j);
            if(xi.get(0).equals(yj.get(0))) {
                double multi = Double.parseDouble(xi.get(1)) * Double.parseDouble(yj.get(1));
                similarity += multi;
            }
        }
    }
    return similarity;
}
```

## (4) 自建 heap 為各篇文章找出 maximal similarity

## a. main method

首先建立名為 hac 的 ArrayList<SimilarityHeap> 來儲存各篇文章的 heap、建立名為 similarityList 的 ArrayList<ArrayList<Similarity>> 來儲存 pair-wise document similarity 以及其對應的兩篇文章的 docID、建立名為 clusterLife 的 ArrayList<Integer> 來記錄此 cluster 是否還存在 (1 為存活，0 為已被併進其他 cluster)。此外，新增了 liveCount 來記錄現存的 cluster 有幾個，供後續輸出 K = 8、13、20 的分群結果使用。

接著以 for 迴圈的方式對每篇文章執行以下步驟：

- ① 為單篇文章建立自己的 heap 以及 similarity list
- ② 以 cosine(...) 的 method 算出此篇文章與其他篇文章的 similarity (排除 self similarity)，參數 (單篇文章所有 term 的 tf-idf) 可由先前建立的三維 tfidfList 中取出。建立一個 Similarity 物件來儲存兩篇文章的 docID 及 similarity，並將此物件加進 heap 以及 similarity list 中，重複執行 1095 次。
- ③ 將此篇文章的 heap 加進先前建立的 hac、similarity list 加進先前建立的 similarityList
- ④ 在 clusterLife 中加入此篇文章的存活值 (1)、liveCount 也加 1

## Code

```

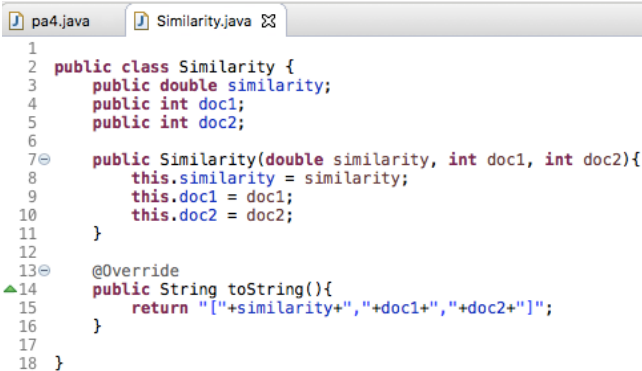
/* 3. Build priority queue for each document */
hac = new ArrayList<SimilarityHeap>();
similarityList = new ArrayList<ArrayList<Similarity>>();
clusterLife = new ArrayList<Integer>();
liveCount = 0; // 計算存活的 cluster 有幾個
for (int n = 1; n <= docSize; n++) {
    SimilarityHeap heap = new SimilarityHeap(); // heap for doc_n
    ArrayList<Similarity> docSimList = new ArrayList<Similarity>(); // similarity list for doc_n
    ArrayList<ArrayList<String>> doc_n = tfidfList.get(n-1);
    for (int i = 1; i <= docSize; i++) {
        if(n != i) {
            ArrayList<ArrayList<String>> doc_i = tfidfList.get(i-1);
            double similarity = cosine(doc_n, doc_i);
            Similarity s = new Similarity(similarity, n, i);
            heap.add(s);
            docSimList.add(s);
        }
    }
    hac.add(heap);
    similarityList.add(docSimList);
    clusterLife.add(1);
    liveCount++;
}
System.out.println("Priority queue lists done!");

```

## b. class Similarity

Similarity constructor 包含了兩篇文章的 docID 以及 similarity。由於要建立 similarityList，所以設定 doc2 是第一維的 docID，doc1 是第二維的 docID，也就是之後要取出此 similarity 會先找到 doc1 的 index，在找到 doc2 的 index。

## Code




```

1
2 public class Similarity {
3     public double similarity;
4     public int doc1;
5     public int doc2;
6
7     public Similarity(double similarity, int doc1, int doc2){
8         this.similarity = similarity;
9         this.doc1 = doc1;
10        this.doc2 = doc2;
11    }
12
13    @Override
14    public String toString(){
15        return "["+similarity+","+doc1+","+doc2+"]";
16    }
17
18 }

```

## c. class SimilarityComparator

用來比較兩個 Similarity (class) 中的 similarity，similarity 較大者往前排。

Code	
	
<pre>1 import java.util.Comparator; 2 3 public class SimilarityComparator implements Comparator&lt;Similarity&gt;{ 4     @Override 5     public int compare(Similarity s1, Similarity s2){ 6         if(s1==null    s2==null) throw new NullPointerException(); 7 8         if(s1.similarity &gt; s2.similarity){ 9             return -1; 10        }else if(s1.similarity &lt; s2.similarity){ 11            return 1; 12        } 13        return 0; 14    } 15 }</pre>	

## d. class SimilarityHeap

import java.util.PriorityQueue 來建立 heap，PriorityQueue 內的型態為先前建立的 Similarity (class)，並且以 SimilarityComparator 去做排序。add(Similarity s) 可以將 s 加入 heap。peek() 會回傳 root，也就是 similarity 最大的 s。removeMax() 可以從 heap 刪除 peek。removeAll() 可以清空整個 heap。remove(Similarity s) 可以從 heap 中刪除特定 element。

Code	
	
<pre>1 import java.util.PriorityQueue; 2 3 public class SimilarityHeap { 4     private PriorityQueue&lt;Similarity&gt; heap; 5 6     public SimilarityHeap(){ 7         this.heap = new PriorityQueue&lt;Similarity&gt;(1095, new SimilarityComparator()); 8     } 9 10    public void add(Similarity s){ 11        heap.offer(s); 12    } 13 14    public Similarity peek(){ 15        Similarity s = heap.peek(); 16        if(s == null){ 17            return null; 18        } else { 19            return s; 20        } 21    } 22 23    public void removeMax(){ 24        heap.poll(); 25    } 26 27    public void removeAll(){ 28        heap.clear(); 29    } 30 31    public void remove(Similarity s){ 32        heap.remove(s); 33    } 34 35 }</pre>	



**(5) 分群 (Complete-Link Clustering)**

a. 先建立用來儲存分群結果的 `ArrayList<ArrayList<Integer>>`

先讓各篇文章自成一群，存進 size 為 1095 的 `kList` 中，並建立空的 `k1List`、`k2List`、`k3List`，供後續儲存 `k1 = 20`、`k2 = 13`、`k3 = 8` 的分群結果。

Code
<pre> /* 4. Clustering result */ kList = new ArrayList&lt;ArrayList&lt;Integer&gt;&gt;(); for (int i = 1; i &lt;= docSize; i++) {     ArrayList&lt;Integer&gt; a = new ArrayList&lt;Integer&gt;();     a.add(i);     kList.add(a); } k1List = new ArrayList&lt;ArrayList&lt;Integer&gt;&gt;(); k2List = new ArrayList&lt;ArrayList&lt;Integer&gt;&gt;(); k3List = new ArrayList&lt;ArrayList&lt;Integer&gt;&gt;(); System.out.println("Clustering result lists built!"); </pre>

b. 做 N-1 次 merge，從各篇文章的 maximal similarity 中找出最大的 maximal similarity 做合併

N-1 次 merge 的 for 迴圈停止條件還包括分到 8 群就停止（在 main method 中一開始就有設定 `k3=8`）。新建一個 heap，將還存活的各群的 maximal similarity (peek) 從先前建立的 `hac` 中取出並加入 heap 中，找出最大的 similarity。

Code
<pre> /* 5. 做 N-1 次 merge */ for (int i = 1; i &lt; docSize-1 &amp;&amp; liveCount &gt; k3; i++) {     // 從 1095 個 maxSim 中找出 maxSim     SimilarityHeap heap = new SimilarityHeap();     for (int j = 0; j &lt; hac.size(); j++) {         if (clusterLife.get(j) == 1) {             heap.add(hac.get(j).peek());         }     }     Similarity maxSim = heap.peek(); </pre>

c. clusterID 較大者併進 clusterID 較小者 (N-1 次 merge 中)

從剛才取得的 `maxSim` 取出對應的 `docID`（也就是 cluster ID），`docID` 較小者設為 `docS`，較大者設為 `docL`，由於 `docL` 要併進 `docS`，所以 cluster L 的存活值改為 0，`liveCount` 也減 1。由於 `Similarity` 中記錄的只是 cluster ID，並不包含此 cluster 的所有 `docID`，因此須在從先前建立的 `kList` 將該 cluster 中所有 `docID` 取出並加進 cluster S 中，加入後再依照 cluster S 中所有 `docID` 的大小做排序，並把 cluster L 中的 `docID` 全部清空。

Code
<pre> // ID 大的 doc 併到 ID 小的 doc (cluster) int docS = Math.min(maxSim.doc1, maxSim.doc2); int docL = Math.max(maxSim.doc1, maxSim.doc2); clusterLife.set(docL-1, 0); liveCount--; ArrayList&lt;Integer&gt; docL_docs = kList.get(docL-1); for (int j = 0; j &lt; docL_docs.size(); j++) {     kList.get(docS-1).add(docL_docs.get(j)); } Collections.sort(kList.get(docS-1)); kList.get(docL-1).clear(); </pre>

- d. 清除 cluster L 的 heap，並刪除所有和 cluster S、cluster L 相關的 Similarity (N-1 次 merge 中)

首先從 hac 中取出 cluster L 的 heap，並整個清空。接著用 for 迴圈從 hac 中取出還存活的 cluster 的 heap，找出和 cluster S、cluster L 算出的 Similarity，移除這兩個 element。由於從 heap 中移除 element 要知道 Similarity 中的所有變數值，doc1 和 doc2 已知 (doc1 是某個還存活的 cluster，doc2 是 cluster S 或是 cluster L)，similarity 則必須從 similarityList 中去找。

Code
<pre>// 刪除 docL 的 priority queue hac.get(docL-1).removeAll();  // 刪除 docS 及 docL 的 similarity for (int j = 0; j &lt; hac.size(); j++) {     if (clusterLife.get(j) == 1) {         SimilarityHeap pq = new SimilarityHeap();         pq = hac.get(j);         ArrayList&lt;Similarity&gt; docSimList = new ArrayList&lt;Similarity&gt;();         docSimList = similarityList.get(j);         if (pq != null) {             for (int l = 0; l &lt; docSimList.size(); l++) {                 Similarity s = docSimList.get(l);                 if (s.doc2 == docS) {                     pq.remove(s);                 }                 if (s.doc2 == docL) {                     pq.remove(s);                 }             }         } else {             System.out.println("There is error for deleting similarity.");         }     } } }</pre>

- e. 以 complete-link 的方法重算和 cluster S 相關的所有 similarity (N-1 次 merge 中)

首先為 cluster S 建立一個新的 heap，並從 kList 中取出 cluster S 的 document list。用 for 迴圈取出各個還存活且非 cluster S 的所有 cluster 的 document list，命名為 clusterO，接著從 similarityList 中出 cluster S 及 cluster O 各取一篇文章的 similarity，若該 similarity 為第一個找出的 similarity 或是當下最小值，則存為 minSim。

找出 minSim 後便可以建立新的 Similarity 物件，將新的 Similarity 加入為 cluster S 建立的新 heap 中，並更新 similarityList。各個 cluster 和 cluster S 的 Similarity 也需要加入各 cluster 原有的 heap 中，並更新 similarityList。for 迴圈結束便可以將 hac 中 cluster S 原有的 heap 更新為 newHeap。



## Code

```

// 重算 docS 的 Complete Link Similarity
SimilarityHeap newHeap = new SimilarityHeap(); // new heap for docS
ArrayList<Integer> clusterS = new ArrayList<Integer>(); // clusterS 中所有 doc
clusterS = kList.get(docS-1);
for (int j = 0; j < kList.size(); j++) {
    if (clusterLife.get(j) == 1 && j != docS-1) {
        ArrayList<Integer> cluster0 = new ArrayList<Integer>(); // cluster0 中所有 doc
        cluster0 = kList.get(j);
        double minSim = 0;
        for (int o = 0; o < cluster0.size(); o++) {
            int docID_o = cluster0.get(o);
            for (int m = 0; m < clusterS.size(); m++) {
                int docID_m = clusterS.get(m);
                int index = docID_o-2;
                if (docID_m > docID_o) {
                    index = docID_o-1;
                }
                double s = similarityList.get(docID_m-1).get(index).similarity;
                if (minSim == 0 || s < minSim) {
                    minSim = s;
                }
            }
        }
        // update docS 的 heap 和 similarityList
        Similarity sim1 = new Similarity(minSim, docS, j+1);
        newHeap.add(sim1);
        int id1 = j-1;
        if (docS > j+1) {
            id1 = j;
        }
        similarityList.get(docS-1).set(id1, sim1);
        // update j 的 heap 和 similarityList
        Similarity sim2 = new Similarity(minSim, j+1, docS);
        hac.get(j).add(sim2);
        int id2 = docS-2;
        if (j+1 > docS) {
            id2 = docS-1;
        }
        similarityList.get(j).set(id2, sim2);
    }
}
hac.set(docS-1, newHeap);

```

★ 解釋 similarityList 的 index 怎麼找

因為並沒有將 self similarity 存入 similarityList 中，所以找出第一維（column）的 index 會比較麻煩，舉以下例子做說明：假設現在要找的是 doc3 的各個 similarity（以藍底表示），doc3 的 index 即為 2。第一維的 index 以紅字表示，由下表可以看到在相同 row 下，不同 column 有不同的 Similarity.doc2（以紅字表示），若 column 的 docID < 3，index = docID - 1；若 column 的 docID > 3，index = docID - 2。

index	0	1	2	3
0	(s, doc1, doc2)	(s, doc1, doc3)	(s, doc1, doc4)	(s, doc1, doc5)
1	(s, doc2, doc1)	(s, doc2, doc3)	(s, doc2, doc4)	(s, doc2, doc5)
2	(s, doc3, doc1)	(s, doc3, doc2)	(s, doc3, doc4)	(s, doc3, doc5)
3	(s, doc4, doc1)	(s, doc4, doc2)	(s, doc4, doc3)	(s, doc4, doc5)
4	(s, doc5, doc1)	(s, doc5, doc2)	(s, doc5, doc3)	(s, doc5, doc4)

## f. 建立並寫出分群結果 k1List、k2List、k3List (N-1 次 merge 中)

在 a. 中已有建立空的 k1List、k2List、k3List，供後續儲存 k1 = 20、k2 = 13、k3 = 8 的分群結果。若 liveCount = k1，表示目前已分成 20 群，便可以 for 迴圈的方式將 kList 中非空的 cluster 的 document list 加入 k1List 中，並以 writeClusterResult(k1List, k1) 的 method 將分群結果輸出為 20.txt 檔，k2 及 k3 亦然。

## Code

```
// Build clustering result
if (liveCount == k3) {
    System.out.println("There are " + k3 + " clusters!");
    for (int j = 0; j < kList.size(); j++) {
        ArrayList<Integer> list = kList.get(j);
        if (!list.isEmpty()) {
            k3List.add(list);
        }
    }
    writeClusterResult(k3List, k3);
    System.out.println("Write k3List done!");
    // check
    System.out.println(k3List.size() + "-" + k3List);
} else if (liveCount == k2) {
    System.out.println("There are " + k2 + " clusters!");
    for (int j = 0; j < kList.size(); j++) {
        ArrayList<Integer> list = kList.get(j);
        if (!list.isEmpty()) {
            k2List.add(list);
        }
    }
    writeClusterResult(k2List, k2);
    System.out.println("Write k2List done!");
    // check
    System.out.println(k2List.size() + "-" + k2List);
} else if (liveCount == k1) {
    System.out.println("There are " + k1 + " clusters!");
    for (int j = 0; j < kList.size(); j++) {
        ArrayList<Integer> list = kList.get(j);
        if (!list.isEmpty()) {
            k1List.add(list);
        }
    }
    writeClusterResult(k1List, k1);
    System.out.println("Write k1List done!");
    // check
    System.out.println(k1List.size() + "-" + k1List);
}
}
```

輸出完後也會在 console 中印出 k1List 的 size 及分群結果，k2 及 k3 亦然。

## Code

```
Problems @ Javadoc Declaration Console
<terminated> pa4 [Java Application] /Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home/bin/java (2022年1月23日 上午10:35:58)
Tf-idf lists done!
Priority queue lists done!
Clustering result lists built!
There are 20 clusters!
Write k1List done!
20-[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 32, 33,
There are 13 clusters!
Write k2List done!
13-[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 32, 33,
There are 8 clusters!
Write k3List done!
8-[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 32, 33,
```

以下說明 method writeClusterResult (k1List, k1) 的處理邏輯：寫出方式和之前的作業都差不多，這裡給予兩個參數，一個是要寫成 txt 檔的 list，一個是要用來為 txt 檔命名的 k。先設定要印出的 String 為空值，接著以 for 迴圈將 list 中各個 cluster 的 document list 取出，再以第二層迴圈從 document list 取出所有 docID 加入 String 中，並以換行來區隔，而不同 cluster 也是以換行來區隔，因此 cluster 間會形成空行。為了確定分群結果確實分了 k 群，另設有限制條件 list 的 size 需等於 k，否則會輸出空白的 txt 檔。

## Code

```
/* Write Clustering Result */
public static void writeClusterResult (ArrayList<ArrayList<Integer>> list, int k) throws Exception {
    File writepath = new File(String.format("src/output/%d.txt", k));
    writepath.createNewFile();
    BufferedWriter bw = new BufferedWriter(new FileWriter(writepath));
    String str = "";
    for(int i = 0; i < list.size() && list.size() == k; i++) {
        String s = "";
        for(int j = 0; j < list.get(i).size(); j++) {
            s = s + Integer.toString(list.get(i).get(j)) + "\n";
        }
        str = str + s + "\n";
    }
    bw.write(str);
    bw.flush();
    bw.close();
}
```

2022.01.23