Data Mining Project 1 - Association Analysis

電機所 N26114277 卓冠廷

1. What do you observe in the below 4 scenarios?

High support, high confidence(1)

Support 和 confidence 兩個指標閥值設高，篩選出來的關聯法則在關聯性和可信度相對比較高，且 rule 通常較少，但可能會發生一種情況，假如有兩種關聯性很高但物品出現頻率很少，這類型的就會因為 High support 篩選掉，因此得到的 rule 關聯性雖然高，但不一定可以找到所有 rule。

High support, low confidence(2)

和(1)相似，在 High support 的情況下會有關聯性高但次數少的組合被篩掉，除此之外，在選擇 rule 的時候因為 confidence 閥值較低，所以會得到較多結果，就 rule 量而言，(1) ⊆ (2)。

Low support, high confidence(3)

Low support 的優點在於不會篩掉可能有用的項目，但也因此保留較多的資料量，花費較多運算時間在排序、篩選、掃描或建立 tree 上，而經 high confidence 篩選後得到的 rule 關聯性和可信度也就會相對高，比較不容易像(1)一樣篩選掉可能資訊，總結來說(1) ⊆ (3)。

Low support, low confidence(4)

相比(3)的結果，這種方式得到的結果較多，但因本來篩選掉的項目就較少，又 confidence 很低的情況下，很多關聯性不高的 rule 都會被納入，導致整個結果可信度很低，(3)和(4)之間的關係可表示為(3) ⊆ (4)。

以上結論將以 Kaggle 測試集做驗證去做分析與探討

2. Experiment with other dataset(s) selected from Kaggle.

Kaggle 測試集- Market Basket Optimisation(以下為連結)

Market_Basket_Optimisation

此測試集共 7501 筆資料，包含 120 種商品

```
a = set(flatten_itemset)
count = []
for i in a:
    count.append(flatten_itemset.count(i))
print("總共幾種商品 : %d" %len(a))
print("交易商品總量 : %d" %sum(count))
print("交易資料量 : %d" %len(transaction_list))
print("平均每筆交易購買商品總數 : %.2f" %float(sum(count)/len(transaction_list)))
```

```
總共幾種商品 : 120
交易商品總量 : 29363
交易資料量 : 7501
平均每筆交易購買商品總數 : 3.91
```

由於資料集較小，但不同商品的數量(120)遠大於平均每筆交易購買商品總數(3.91，單筆最多 20)，support 跟 confidence 皆不能設太高，經過測試將 High support、Low support、high confidence、low confidence 設定為 0.03、0.01、0.3、0 去比較

High support, high confidence(0.03,0.3)(1)

```
spaghetti -> mineral water : sup 0.060 conf 0.343 lift 1.439
chocolate -> mineral water : sup 0.053 conf 0.322 lift 1.352
milk -> mineral water : sup 0.048 conf 0.370 lift 1.554
ground beef -> mineral water : sup 0.041 conf 0.417 lift 1.748
ground beef -> spaghetti : sup 0.039 conf 0.399 lift 2.291
frozen vegetables -> mineral water : sup 0.036 conf 0.375 lift 1.572
pancakes -> mineral water : sup 0.034 conf 0.355 lift 1.489
length of Association rule 7
```

跟(3)的情況做比較，可以發現關聯性高但物品出現頻率較少的 rule 被刪掉。如下圖: confidence 為 0.448，但 support 值 0.01 被刪掉

```
spaghetti olive oil -> mineral water : sup 0.010 conf 0.448 lift 1.878
spaghetti -> mineral water : sup 0.060 conf 0.343 lift 1.439
chocolate -> mineral water : sup 0.053 conf 0.322 lift 1.352
```

High support, low confidence(0.03,0)(2)

跟(1)的情況做比較，rule 數從 7 上升到 36

```
ground beef -> spaghetti : sup 0.039 conf 0.399 lift 2.291
ozen vegetables -> mineral water : sup 0.036 conf 0.375 lift 1.572
ncakes -> mineral water : sup 0.034 conf 0.355 lift 1.489
length of Association rule 36
```

Low support, high confidence(0.01,0.3)(3)

比較(3)和(1)，從下圖可以看出(3)保留較多的資料量，花費較多運算時間在排序、篩選、掃描或建立 tree 上

(3)的情況:

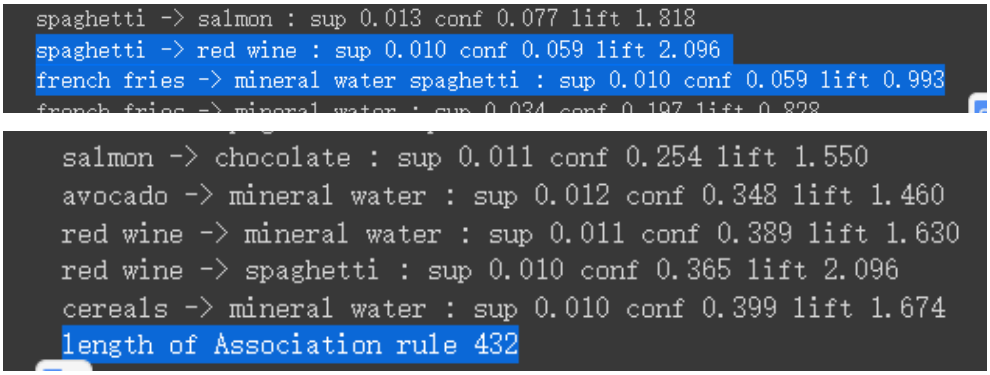
```
apriori Freq_Pattern :  257
length of Association rule :  63
fp_growth Freq_Pattern :  257
length of Association rule :  63
kaggle apriori time cost -74.1748149394989 s
kaggle fp_growth time cost -0.47536373138427734 s
```

(1)的情況:

```
apriori Freq_Pattern :  54
length of Association rule :  7
fp_growth Freq_Pattern :  54
length of Association rule :  7
kaggle apriori time cost -2.7585842609405518 s
kaggle fp_growth time cost -0.20563983917236328 s
```

Low support, low confidence(0.01,0)(4)

得到的結果很多，但整個結果可信度卻很低，(3)和(4)之間比較可看到 rule 從 63 上升到 432



3. 演算法撰寫架構:

Apriori:

[1] 演算法步驟:

Step1:

掃描資料集中各 item 出現的 frequency，除上總交易筆數，將低於 min_support 的 item 刪去，得到 L_1

Step2:

從剩餘的 item 去組合出新的一批候選人(C_2)，每位候選人(C)內含兩個 items，並去計算每位候選人在資料集中出現的次數，除上總交易筆數，得到 support 值

Step3:

若候選人 support 值小於 min_support，刪除此候選人，剩餘這批候選人為 L_2，而刪除的這批候選人必需紀錄(C_2_delete)

Step4:

從 L_2 做組合，內含 3 個 items，且組合不得包含 C_2_delete 內出現的 C，生成 C_3，並計算 C_3 中每位候選人的 support 值，刪除小於 min_support 的候

選人，得到 L_3，而刪除的這批候選人為(C_3_delete)

Step4':

從 L_i 做組合，內含 i+1 個 items，且組合不得包含 C_i_delete 內出現的 C，生成 C_{i+1}，並計算 C_{i+1} 中每位候選人的 support 值，刪除小於 min_support 的候選人，得到 L_{i+1}，而刪除的這批候選人為(C_{i+1}_delete)

Step5:

重複 **Step4'**，直到產生的 L_n 內沒有候選人

Step6:

將 L_1 到 L_n 的結果集合起來，此為 Frequency_Pattern，並以此去列出關聯法則

[2] 各函數功能介紹:

- ✓ Candidate( ): 將 L_{i-1} 到 C_i 的所有組合與 C_{i-1}_delete 做比對，得到 C_i
- ✓ Candidate_i( ): 將 C_i 對資料集做掃描得到每個 C 的 support 值
- ✓ List_i ( ): 將 C_i 內小於 support 值的 C 砍掉得到 L_i
- ✓ Li_item_combination ( ): 算 L_{i-1} 到 C_i 的所有組合用，這邊會用到下面的函式 Li_combination ( )
- ✓ **Li_combination ( ): 快速組合方法，加速計算用! (加速運算的方式在時間分析)**
- ✓ Frequency_pattern ( ): 將上述函式整合，當 L_i 內沒有東西時就回傳所有 pattern 供計算關聯法則用
- ✓ **apriori (file, min_support, min_conf): 整合以上所有函式，從讀檔到寫檔，可更改 min_support 或 min_confidence**

FP_growth:

[1] 演算法步驟:

Step1:

掃描資料集中各 item 出現的 frequency，除上總交易筆數，將低於 min_support 的 item 刪去，也將資料集中這些 item 去除得到新資料集

Step2:

將 **Step1** 剩餘的項目以 support 排序，由大至小，得出 header table，並將新資料集中每筆資料依 header table 的 item 排序方式做排序

Step3:

建立 FP-tree，先將第一筆資料從原點(root)做分支，下一筆資料一樣從原點開始排，規則是遇到同樣 item 的節點就合併，否則就另立新節點，且建立 header table 的 pointer(建節點時順便紀錄該項 item 的 Node 位置)

Step4:

對 header table 中每個 item 紀錄的 Node 位置去回朔到源頭(root)，每個 item 都有其對應的分支(condition_pattern_base)，分支數等於每個 item 紀錄的 Node 數

Step5:

再對 Header table 中每個 item 去以 **Step4** 得到的分支建立樹，再重複 Step4 一次(建立的 header table 後面統稱 item_header_table)，會得到每個 item 建樹(item_fp_tree)時的 item_header_table_pointer 和對應分支，將這些對應的分支內的 item 做組合和計算 support，大於 min_support 的組合才留下(conditional_FP_tree)**(這邊計算要考慮各分支內 item 的組合可能會重複，要累加後做篩選!!!)**

Step6:

把 item 配上 conditional_FP_tree 內此 item 對應的組合去產生 Frequency_Pattern，並以此去列出關聯法則

[2] 各函數功能介紹:

- ✓ header_table ( ): 建立 hearder table
- ✓ transaction_list_process ( ): 將原始資料刪除不到 min_support 的 item 並排序
- ✓ class Node: 節點 Class

  def __init__(self, item, count=None, parent=None, children=None):
- ✓ FP_tree ( ): 建樹以及回傳 Header table pointer
- ✓ cond_pattern_base ( ): 由每個 item 和其對應的分支組成
- ✓ item_FP_tree ( ): item 對應的樹和回傳 item_header_table_pointer
- ✓ combinations ( ): 組合的函式，可設定組合的 item 數
- ✓ **freq_pattern ( ): condition_pattern_base 到 conditional_FP_tree 到產生 freq_pattern(加速運算的方式在時間分析)**
- ✓ **fp_growth (file, min_support, min_conf): 將上述函式整合，當 Li 內沒有東西時就回傳所有 pattern 供計算關聯法則用**

4. 測資結果時間分析:

2022 新測資結果:

參數(min_support、min_conf) : 0.2、0.1

```
216
216
test_data apriori time cost -3.8367607593536377 s
test_data fp_growth time cost -0.5684981346130371 s
```

參數(min_support、min_conf) : 0.15、0.1

```
497
497
test_data apriori time cost -12.727112770080566 s
test_data fp_growth time cost -2.60945987701416 s
```

參數(min_support、min_conf) : 0.1、0.1

```
1490
1490
test_data apriori time cost -146.75170826911926 s
test_data fp_growth time cost -61.918609380722046 s
```

kaggle 測資結果:

參數(min_support、min_conf) : 0.03、0.1

```
54
54
kaggle apriori time cost -2.7838714122772217 s
kaggle fp_growth time cost -0.17989253997802734 s
```

參數(min_support、min_conf) : 0.01、0.1

```
257
257
kaggle apriori time cost -73.14704155921936 s
kaggle fp_growth time cost -0.466270685195922 85 s
```

以上測資結果可發現，當 min_support 越小時，程式跑越久，而上述時間成果是經加速後產生，以下來分析加速方式。

Apriori:

起初新測資 min_support、min_conf 為 0.1、0.1 時的時間需要花二十幾分鐘，主要卡在 L_{i-1} 組合產生 C_i，又要砍去包含跟 C_{i-1}_delete 的組合:

[1] 原始方法(二十幾分鐘):

L_{i-1} 兩兩組合產生 i 個 items 組成的 itemset，不能包含 C_{i-1}_delete，但兩兩組合的過程會產生很多重複的 itemset，這些重複 itemset 會浪費計算時間

[2] 新方法(八分鐘):

兩兩組合部分改成兩兩取聯集後再組合

刪去重複的 itemset 再去做不能包含 C_{i-1}_delete 的問題

但尚未縮減至三分鐘內，經 colab 測試發現主因出在兩兩組合上，何不在產生 i 個 items 組成的 itemset 時，就不讓重複的問題產生?

[3] 最終方法(三分內):

在 L_{i-1} 組合時，先算出所有 L_{i-1} 取聯集的結果，以此做為定序列(list_A)，找 L_{i-1} 第一個 itemset(B,內含 i-1 個 items)內，其 item 在定序列中 index 最高者，把定序列在最高 index 後的序列(list_C)取出來，將 B 與 list_C 中的各個元素組出新 itemsets(內含 i 個 items)，當 L_{i-1} 跑到最後一個 itemset 時也代表所有組合做完了(C_i_temp)

將此方法也用在 C_{i-1}_delete 上，可產生要刪去的 C_i_delete，C_i_temp 扣掉 C_i_delete 即 C_i

FP_growth:

[1] 原始方法(十分鐘):

此想法來自於李照棋同學的開導，方才突破 **Step5** 中各分支重複的問題，建 item_FP_tree(假設 itemA 的 tree)後，去算 itemA_header_table 中每個 item 分支上方 itemset 組合，並將這些組合紀錄累加，若組合的 support 大於 min_support，保留這組合並加入 itemA，此為一種 Frequency_Pattern，且不會有重複組合的情形。

但尚未縮減至三分鐘內，同樣以 colab 測試發現主因出在多算了很多 item_header_table 內多餘的 item，因為 item_header_table 內的 item 其本身的 count 可能就不超過 min_support，何需多計算上方的分支，更不需去算那些組合

[2] 新方法(一分鐘):

計算 item_header_table_pointer 中各 item 節點 count 的加總，若小於 min_support，砍掉那個 item

總結兩種演算法，前者花費大量時間在計算候選人和其對應的 support 值(需重複掃描資料集)，後者則只需掃一次，即可製作 Frquency_Pattern，這也是為何 Apriori 相較 FP_growth 慢很多的原因所在