Project Ⅰ

Association Rule

Name : 黃柏喻

Student ID: N16064674

Department: Mechanical Engineering

Chapter 1. 使用方法與概念解析

1-1 暴力法

暴力法採用的是將所有可能的 candidate patterns 全部盡數列出，接著將所有列出的 patterns 全部丟進 dataset 進行比對並統計數量，當數量大於 minimum support (通常是佔該資料庫大小的一定比例) 時，該 pattern 便是我們要找的高頻特徵組合。

1-2 Apriori Method

Apriori 的方法是先找出所有符合 minimum support 的 one-item frequent set (L1)後，利用這些 item 組合出 two-item candidate set，經檢查數量後得出 L2，之後的步驟便以此類推，直到超出 candidate set 超過 item 的數量後才停止。

1-3 Fp growth

此方法利用的是建構”樹”(Fp tree)來快速搜索的方法，第一次從所有案例中抓出 one-item frequent set (L1)的時候，除了要滿足 minimum support 之外，還要照著數量由大到小排列，接著將此順序實行在資料庫的所有案例上，除了刪除不滿足 support 的 items 外，順序化的資料庫對後續建構 Fp tree(由上往下)也相當重要。像是找到 conditional pattern bases 後要建構 conditional FP tree 就是要依照此原則進而找出 frequent patterns。

演算法實做上，是從最後 mining 的演算法開始思考，用 for 迴圈包著 if 的遞迴函數，採用的想法是由下往上找父節點，因為子結點可能有很多個不過父節點只會有一個。因此在 node 的類別中，需要有子結點的集合還要有父節點的地址，接著就是利用整個 tree 中，包含相同 item 的 node 中可以不斷往右邊鏈結，也就是 singular link list 的概念。

Chapter 2. 結果比較與討論

測試的 dataset 包括

- IBM trans = 100 & item = 10…(a)
- IBM trans = 1000 & item = 10…(b)
- IBM trans = 10000 & item = 100…(c)
- UCI Absenteeism at work…(d)

所有的運行結果都在 Github 的 Report 資料夾內(HTML 形式)。

關於 UCI dataset 的處理是採用以下的方式:

(Github 上的 Readme.md 也有解說)

由於這份資料的項目有點煩雜且參差不齊，所以有些特徵並不考慮，然後主要抓出了以下幾點特徵並轉成數值化的項目模式。

1-- Absenteeism time in hours > 平均值
2-- Monday
3-- Tuesday
4-- Wednesday
5-- Thursday
6-- Friday
7-- Distance from residence to working > 中位數
8-- Disciplinary failure == 1 (Yes)
9-- Education == 1 (high school)
10-- Social drinker == 1 (Yes)

其中 2,3,4,5,6 一個案例中只會有一個數字出現，也就是該員工的請假日期。最後的資料形態變為 740 個工作缺席按例，然後有 10 項 items。

Table 2.1 各資料庫與方法所花費時間(sec)

	a	b	c	d
暴力法	0.13	0.295	X	1.141
Apriori	0.009	0.029	0.198	0.003
Fp growth	0.007	0.01	0.07	~=0.0

註: X 表示暴力法造成記憶體無法負荷

在使用暴力法、Aprioir、Fp growth 三種方法後，發現了以下幾點特徵。

- 暴力法在運行中最佔記憶體，且所耗的時間也最久。
- 在同樣的 dataset、minimum support 條件下，Fp growth 可以說是最快的方法。

後來在（c）資料庫中，我把 minimum support 往下調到 0.1 後，分別再使用兩個方法跑一次，圖 2.1 中可以發現 Apriori 方法花了 0.72 秒左右，但是圖 2.2 中，Fp growth 反而花了將近 2.6 秒，與前面的結論相反。可能的原因是 minimum support = 0.1 時，符合條件的 set 數量會激增，而也因此需要花更多時間進行 tree 的建構，導致 time complexity 比 Apriori 直接進行數量上的比較來的高。

```
start = time.time()
C1 = generateC1(dataset_list)
L1,support = check_freq(dataset_list,C1,min_support = 0.1)
All_freq_set = [L1]
k = 2

while (len(All_freq_set[k-2]) > 0):
    Ck = apriori_gen(All_freq_set[k-2],k)
    Lk,supportk = check_freq(dataset_list,Ck,min_support = 0.1)
    support.update(supportk)
    All_freq_set.append(Lk)

    k += 1

end = time.time()

print("Time Taken is:")
print(end-start)
print("All frequent itemsets:")
print(All_freq_set)
```

```
Time Taken is:
0.7220413684844971
All frequent itemsets:
[[frozenset({66}), frozenset({28}), frozenset({89}), frozenset({3}), frozenset({48}), frozenset({72}), frozenset({86}), frozens
et({29}), frozenset({23}), frozenset({14}), frozenset({80}), frozenset({43}), frozenset({40}), frozenset({21}), frozenset({9
3}), frozenset({11}), frozenset({9}), frozenset({87}), frozenset({85}), frozenset({81}), frozenset({73}), frozenset({39}), froz
enset({35}), frozenset({17}), frozenset({62}), frozenset({61}), frozenset({52}), frozenset({51}), frozenset({36}), frozenset({7
1}), frozenset({8}), frozenset({83}), frozenset({78}), frozenset({69}), frozenset({67}), frozenset({63}), frozenset({47}), froz
enset({38})], [frozenset({38, 87}), frozenset({69, 87}), frozenset({63, 87}), frozenset({36, 38}), frozenset({36, 87}), frozens
et({36, 63}), frozenset({38, 63}), frozenset({69, 38}), frozenset({69, 63})], []]
```

Figure 2.1 Apriori method with minimum support = 0.1

```
min_support = float(0.1*len(dataset))
start = time.time()

Fptree, HeaderTable = createTree(dataset, min_support)

frequent_set = []
mining(set([]), Fptree, HeaderTable, min_support, frequent_set)

end = time.time()

print("Time Taken is:")
print(end-start)
print("All frequent itemsets:")
print(frequent_set)
```

```
Time Taken is:
2.5731470584869385
All frequent itemsets:
[{26}, {52}, {66}, {21}, {23}, {71}, {73}, {93}, {51}, {72}, {89}, {35}, {78}, {67}, {62}, {29}, {86}, {9}, {40}, {61}, {14},
{83}, {47}, {43}, {80}, {39}, {81}, {3}, {28}, {17}, {85}, {11}, {8}, {48}, {36}, {36, 87}, {36, 63}, {36, 38}, {69}, {69, 87},
{69, 63}, {69, 38}, {87}, {63, 87}, {38, 87}, {63}, {38, 63}, {38}]
```

Figure 2.2 Fp growth method with minimum support = 0.1

Chapter 3. 心得

三種方法在 coding 難度上各有不同，暴力解好理解但是非常吃資源，Apriori 的優點是讓前面產出的結果去篩選出後面的 frequent itemset，整體的 space complexity 相對低很多，Fp growth 則是 tree 的特性由高到低、由上而下進行排列、建構，優點是有同樣特徵的分枝可以用計數的方式省掉很多空間，在找尋鏈結上也相當迅速。三個方法大致上呈現的趨勢是行數愈多、效率愈高。

再來就是資料的前處理，IBM data generator 產生的資料庫有點紊亂，python 是非常適合用來進行資料處理的語言，所以我採用的是 pandas 套件裡的 Dataframe 資料型態，另外就是 UCI 的 dataset，這個資料檔裡面有一些我認為不會與預測目標有直接關係的參數，我把這些刪除後並把剩餘許多中文資料數值化以便進行與”工作缺席”相關的參數分析，因此，我認為把資料整理成可用性高的形態也是資料科學家的一大工作。

由於本身是機械系，有一些資工的課程 miss 掉，像是 data structure、algorithm 等，看到 tree 時並沒有立即與 recursive 連結上，導致花了很多時間在使用 iterative 上，行數曾一度衝上 250 行但是仍失敗，後來看了一些線上課程和弄懂了網路上的解法

後才慢慢增加了一些概念，逐一的 trace 後才自己實做出來。

此次學到最多的東西就是寫一個好的演算法該有的步驟。首先，除了要了解該方法的核心概念並手動解過一個較小的資料庫，第二步，開始計畫需要用甚麼方式寫，包括資料型態、類別變數、函式功能，先用 pseudo code 寫一遍在紙上後，第三步才有可能順利的打完整個程式，不然中間會有很高的機會要砍掉重練，當然在嘗試過後失敗了，就有參考了很多網路上的程式碼，再進行逐一的 trace 還有修改。