HW1\_Report

系級:智能系統 姓名:張宸瑋 學號:312581006

1. Explain how to run your code in Step Ⅱ and Ⅲ.

|  |
| --- |
| **如何使用apriori.py** |
|  |
| 命令行選項 |
|  |
| 使用範例 |
| **如何使用fpgrowth.py(請使用這個做為step IV所需要使用的演算法)** |
|  |
| 命令行選項 |
|  |
| 使用範例(要先進到fpgrowth資料夾) |
| **如何使用negFIN.py** |
|  |

|  |
| --- |
| 命令行選項 |
|  |
| 使用範例 |

1. Step Ⅱ

* Report on the mining algorithms/codes:

|  |
| --- |
| 說明:  透過下面的計算時間報告我們可以看到，對於support的調整所帶來的影響，如果我們將support調高，那麼運算速度將會相對於低support來說，有明顯的提升，因為較低的support會造成更多的frequent itemset，因此增加更多的計算與儲存開銷。  而我們也可以看到當資料集越來越大，因為Apriori算法需要掃描更多次的資料集，並且也會產生更多的候選項集時，因此會造成更多的時間開銷。 |

* Paste the screenshot of the computation time and the ratio of Task2 computation time compared to that of Task 1 in your report.

|  |
| --- |
|  |
|  |
|  |
| Configuration for dataset A: {0.2, 0.5, 1.0} |
|  |
|  |
|  |
| Configuration for datasets B: {0.15, 0.2, 0.5} |
|  |
|  |
|  |
| Configuration for datasets C: {1.0, 2.0, 3.0} |

* Paste the screenshot of your code modification for Task 1 and Task 2 with comments and explain it.

|  |
| --- |
| Modification for Task 1 |
| 1. dataFromFile |
|  |
| 說明:  這裡我更改了助教所提供的Apriori原始碼資料前處理的部分，因為透過IBMGenerator所生成的資料格式，每一列的前三個資料分別是TID、TID、NITEMS，因為我們只需要ITEMSET的部分，因此只要取每一列第四個元素開始即可。 |

|  |
| --- |
| 1. runApriori |
|  |
| 圖(一) |
|  |
| 圖(二) |
| 說明:  圖(一)針對Task1的部分，透過start\_time以及task1\_end\_time計算所需的執行時間，而toRetItem則是用來記錄Result file 1所需要的資料內容，並且透過resultFileTwoList來記錄Result file 2所需要的資料內容。  圖(二)針對Task2的部分，透過start\_time以及task2\_end\_time計算所需的執行時間，而迴圈的部分，主要是在尋找Frequent Closed Itemset，一個Itemset如果是Frequent Closed Itemset，那它的超集就不會跟它有相同的support，而因為largeSet每層儲存不同的K-Itemset，而對於任何一層的每一個Itemset來說，如果它是Frequent Closed Itemset，那它的下一層的Itemset絕對不會出現跟它相同的support，因此根據這個原理，計算出所有的Frequent Closed Itemset，而largeSet的最後一層，因為絕對不會有比它大的Itemset，所以都會是Frequent Closed Itemset。 |

|  |
| --- |
| 1. Write File |
|  |
| 圖(一) |
|  |
| 圖(二) |
| 說明:  圖(一)針對Task1的部分，進行寫入檔案的步驟，而圖(二)針對Task2的部分，進行寫入檔案的步驟。 |

1. Step Ⅲ

* Descriptions of your mining algorithm

|  |
| --- |
| 說明:  這次我使用了[mlxtend](https://github.com/rasbt/mlxtend/blob/master/mlxtend/frequent_patterns/fpgrowth.py)所提供的FP-growth來作為我這次step Ⅲ 使用的演算法，而在原始碼中，整個FP-growth的Program flow 為  1.建立頻繁模式樹（FP樹）：   * 掃描資料集，統計每個項目的頻度。 * 根據頻度對項目排序，將資料集轉換為FP樹結構。FP樹是一種緊湊的樹狀結構，用於表示頻繁項集之間的層次關係。     2.建立條件模式基：   * 對於每個頻繁項集，建立一個條件模式基。條件模式基是指包含特定頻繁項集的所有路徑。   3.遞歸挖掘頻繁項集：   * 對於每個頻繁項集，從FP樹中提取條件模式基。 * 根據提取的條件模式基，建立一個新的FP樹，然後遞歸挖掘頻繁項集。 * 這個遞歸過程持續進行，直到無法生成更多頻繁項集。      1. 合併頻繁項集：  * 將生成的所有頻繁項集合併為最終的頻繁項集列表。 |
| Anything you want to share:  而在這次的實驗中，我也參考了收錄在sciencedirect期刊的[negFIN: An efficient algorithm for fast mining frequent itemsets](http://www.philippe-fournier-viger.com/spmf/NEGFINarticle.pdf)，而這篇文章主要是提出一個高效率的數據結構，來幫助我們找到frequent itemset，它參考了之前一些透過前綴樹資料結構的方式，(1)Node-list、(2)N-list、(3)Nodeset、(4)DiffNodeset，儘管這些方式在大部分的表現表現優異，但是都存在這一些問題，像是占用過多的記憶體，抑或是對於特定的dataset會有執行時間過長的問題，因此作者提出了NegNodeset，透過位圖編碼的方式，以及bitwise的操作，來提供更高效的運算。  而程式碼主要的Program flow可以分為下面幾個部分  1.初始化F為空集。  2.呼叫構建BMC-tree（DB，min-support）函數，以構建BMC-tree並找到L1，L1包含頻繁1項集。  3.添加L1中的項集到F中。  4.對於BMC-tree中的每個節點N，按照任意順序遍歷BMC-tree。  5.將N的信息添加到與項N.item-name相關的節點集Nodeset中。    6.創建根節點root。  7.將根節點的層級root.level設置為0（根節點位於層級0）。 8.初始化根節點的子節點列表root.children-list為空。  9.初始化根節點的項名稱root.item-name為空。  10.初始化根節點的項集root.itemset為空。    11.對於L1中的每個項i：   * 創建一個名為child\_i的節點。 * 將child\_i的層級child\_i.level設置為root.level + 1。 * 將child\_i的項名稱child\_i.item-name設置為i。 * 將child\_i的項集child\_i.itemset設置為{i}。 * 將child\_i添加到根節點的children-list中。 * 呼叫構建頻繁項集樹的函數constructing\_frequent\_itemset\_tree（child\_i，∅）     12.返回根節點root。 |

|  |
| --- |
| Reference:   * [negFIN: An efficient algorithm for fast mining frequent itemsets (philippe-fournier-viger.com)](http://www.philippe-fournier-viger.com/spmf/NEGFINarticle.pdf) * [mlxtend/mlxtend/frequent\_patterns/fpgrowth.py at master · rasbt/mlxtend (github.com)](https://github.com/rasbt/mlxtend/blob/master/mlxtend/frequent_patterns/fpgrowth.py) * [【精选】Chapter 12 使用FP-growth算法来高效发现频繁项集\_fpgrowth算法算频繁项集\_DB架构的博客-CSDN博客](https://blog.csdn.net/u011868279/article/details/124549999) |

* Differences/Improvements in your algorithm

|  |
| --- |
|  |
|  |
| 這裡我更改了原始代碼中，資料前處理的部分，讓它可以符合原始碼中所需要的dataframe資料格式。 |
| 說明:  以上我使用的兩個改進算法，跟Apriori相比最大的差異就是不需要生成候選項集，因此並不需要頻繁的掃描資料集以及生成候選項集，它們透過樹狀結構來獲得frequent itemsets，大幅的提升了運算效率。 |

* Computation time

|  |
| --- |
| FP-growth(非**Candidate-based)** |
|  |
|  |
| Configuration for dataset A: {0.2, 0.5, 1.0} |
|  |
|  |
| Configuration for datasets B: {0.15, 0.2, 0.5} |
|  |
|  |
| Configuration for datasets C: {1.0, 2.0, 3.0} |

|  |
| --- |
| negFIN(非**Candidate-based)** |
|  |
|  |
|  |
|  |
| Configuration for dataset A: {0.2, 0.5, 1.0} |
|  |
|  |
|  |
|  |
| Configuration for datasets B: {0.15, 0.2, 0.5} |
|  |
|  |
|  |
|  |
| Configuration for datasets C: {1.0, 2.0, 3.0} |

* Discuss the **scalability** of your algorithm in terms of the size of dataset (i.e., the rate of change on computing time under different data size; the largest dataset size the algorithm can handle, etc).

|  |
| --- |
| 討論:  從上面的實驗結果來看，我們可以看到，這兩種改進算法，相對於Apriori，其計算速度都有明顯的提升，而這裡我多實驗了negFIN算法，是因為在查詢資料時，查詢到這篇paper，而我在這篇paper中看到在它的實驗結果中(圖一所示)，它的表現相對於FP-growth，有明顯的提升，因此希望能夠透過這個算法，來嘗試出比FP-growth更快的結果，但是從實驗結果中，我們可以看到，在資料集越來越大的時候，FP-growth跟negFIN的差異越來越大，這裡我探討了可能的結果，我認為其原因可能是因為在[mlxtend](https://github.com/rasbt/mlxtend/blob/master/mlxtend/frequent_patterns/fpgrowth.py)所提供的FP-growth原始碼 ，它是基於pandas這樣的資料結構來去實現演算法，這種資料結構在處理大數據的時候性能十分優異，相較於dict、list(negFIN所使用)這些資料結構來說，因此希望之後有機會，能夠去探討，是否能夠透過pandas來去實現negFIN演算法，但整體而言，我們可以看到，這兩種演算法在對於大數據集的處理速度來說，都是明顯優於Apriori的。 |
|  |
| 圖(一) |