HW1_Report

1. Explain how to run My code in Step Ⅱ and Ⅲ
    a. When running Step2.py or Step3.py, users must enter the following command line into cmd. Among them, users can make changes to inputFile, minSupport, and outputPath.
    command line : python3 Step(2 or 3).py -f dataset(A, B or C) -s minSupport -p outputPath
2. Step Ⅱ
    a. The modifications you made for Task 1 and Task 2
        i. Modify command line option "inputFile" : 修改 help 參數中所說明的文字，以及將 default 值改為 'datasetA.data'，代表命令行中若未提供 -f 或 --inputFile，則該選項將默認為 'datasetA.data'。
        ii. Modify command line option "minSupport" : 將 default 值改為 '0.01'，代表命令行中若未提供 -s 或 --minSupport，則該選項將默認為 '0.01'。
        iii. Load data from file : 發現每筆 transaction 前三筆皆不屬於 item，因此將其移除，且因為 input file 格式從原先 csv 檔改為 data 檔，所以也不需要 remove trailing comma。
        iv. Add a new command line option "outputPath" : users可以透過 -p 來指定想要的 output file path。
        v. Run Apriori : 新增一個名為 num_iter 的 dict output。
        vi. Find closed frequent itemsets : 根據 result1 所產生的 frequent itemsets，找出 frequent closed itemsets。
        vii. Write output to file task1 / 2 : 根據 spec 需求運用對應的變數，將結果輸出到名為 task1 result1 / 2 & task2 result1 的 .txt 文件中。
        viii. Calculate computation time : 透過 time.time() function 來記錄各個 task 所花費的時間。
    b. The restrictions (e.g., the scalability, etc.)
        i. Scalability : 由於 Apriori algorithm 中的 candidate itemset 數量會隨著 items 數增長而指數級增加，亦即當有 n 筆 items 時，candidate itemset 數量為 $2^n - 1$。所以在 large dataset 中，會因為 itemset 數量龐大，導致 candidate itemset 爆炸，同時導致記憶體和計算資源的耗費。
        ii. Scan dataset multiple times : 由於 Apriori algorithm 需要多次掃描 dataset 以生成不同大小的 candidate itemset。就如，當尋找大小為 3 的 frequent itemset 時，必須先生成並計算大小為 2 的 candidate itemset。這種多次掃描對於大型數據集來說，I/O 負擔非常重，會導致運行速度顯著下降。
    c. Problems encountered in mining
        i. IBMGenerator for generating dataset : 當初只認為從 dataset 的第 3 column 開始就是 itemsets，後來 run toy dataset 發現到結果與 TA 提供的 answer 不同，跟同學討論後發現是沒弄清楚 dataset format，最後修正了讀檔的相關 code，才可以得到正確的結果。
        ii. Add a redundant variable : 原先為了計算 total number of frequent itemsets，而新增 total_num 此變數，後來為了精簡 Step2.py 的 code，重新檢查發現可以直接計算 item 的 length，便可以得到正確結果。
    d. Any observations/discoveries
        i. Time complexity of finding frequent itemset : 使用 Apriori 演算法來針對 data 數量為 1000 的 datasetA 找尋 frequent itemset，當 min support 值設定為 0.003 時，會需要 21 秒左右的時間，而相同設定下 dataset 換成數量為 100,000 的datasetB，則需要花費 2102 秒左右的時間，資料量差了 100 倍而花費時間剛好也是差了 100 倍，因此為了驗證花費時間是否都會是 100 倍的關係，就嘗試設定其他的 min support 值，發現

設定為 0.002 時，時間會差 33.5 倍，而設定 0.004 的話，時間會差 130 倍左右。因此從這些實驗結果看起來，dataset 的數量與執行時間並不是單純倍數的關係。附圖為 datasetB 的執行時間實驗結果。

```
michelle@DESKTOP-RE3242E:~/IBMGenerator/Apriori_python$ python3 Step2.py -f datasetB.data -s 0.003 -p 123
Results of task1 of Step2 :
--- Elapsed time of task1 : 2102.2287917137146 seconds ---
```

ii. Time complexity of find frequent closed itemsets function : 由於此 function 的 implement 是使用兩層的 for loop 來做，因此函數的時間複雜度為 $O(n^2)$。為了真實瞭解 find frequent closed itemsets 的執行時間，我比較 datasetA 與 datasetB 在 min support=0.003 下的執行時間，我發現 datasetA frequent itemsets 共有 7776 筆，在執行此函數時便花了約 18.556 秒，而 datasetB frequent itemsets 共有 2648 筆，則是花了不到 2 秒的時間，兩個的 frequent itemsets 數量差了 3 倍，而時間差了 9 倍左右，由此可知當 frequent itemsets 數量越大，執行時間也會以接近平方的比例成長。附圖為 datasetB 的執行時間實驗結果。

```
michelle@DESKTOP-RE3242E:~/IBMGenerator/Apriori_python$ python3 Step2.py -f datasetB.data -s 0.003 -p 123
Results of task1 of Step2 :
--- Elapsed time of task1 : 2102.2287917137146 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 2104.154888868332 seconds ---
```

e. Paste the screenshot of the computation time (try different settings of minimum support)

i. Step2.py -f datasetA.data -s 0.003 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 21.1325466632843 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 39.688565492630005 seconds ---
```

ii. Step2.py -f datasetA.data -s 0.006 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 4.023818254470825 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 4.265498161315918 seconds ---
```

iii. Step2.py -f datasetA.data -s 0.009 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 2.585451364517212 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 2.632643461227417 seconds ---
```

iv. Step2.py -f datasetB.data -s 0.002 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 4685.35379076004 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 4696.186039447784 seconds ---
```

v. Step2.py -f datasetB.data -s 0.004 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 1292.0596506595612 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 1292.6628878116608 seconds ---
```

vi. Step2.py -f datasetB.data -s 0.006 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 787.6707048416138 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 787.7986953258514 seconds ---
```

vii. Step2.py -f datasetC.data -s 0.005 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 4844.37987446785 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 4844.611583709717 seconds ---
```

viii. Step2.py -f datasetC.data -s 0.01 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 2664.661855697632 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 2664.6967072486877 seconds ---
```

ix. Step2.py -f datasetC.data -s 0.015 -p output :

```
Results of task1 of Step2 :
--- Elapsed time of task1 : 1606.2691893577576 seconds ---
Results of task2 of Step2 :
--- Elapsed time of task2 : 1606.285308599472 seconds ---
```

f. Show the ratio of computation time compared to that of Task 1 ((Task2/Task1)*100%)
   i. Step2.py -f datasetA.data -s 0.003 -p output : 187.808%
   ii. Step2.py -f datasetA.data -s 0.006 -p output : 106.006%
   iii. Step2.py -f datasetA.data -s 0.009 -p output : 101.825%
   iv. Step2.py -f datasetB.data -s 0.002 -p output : 100.231%
   v. Step2.py -f datasetB.data -s 0.004 -p output : 100.047%
   vi. Step2.py -f datasetB.data -s 0.006 -p output : 100.016%
   vii. Step2.py -f datasetC.data -s 0.005 -p output : 100.005%
   viii. Step2.py -f datasetC.data -s 0.01 -p output : 100.001%
   ix. Step2.py -f datasetC.data -s 0.015 -p output : 100.001%
g. Paste the screenshot of your code modification for Task 1 and Task 2 with comments and explain it (need to explain it in clear and well-structured ways)
   i. Modify command line option "inputFile" : 修改 help 參數中所說明的文字，以及將 default 值改為 'datasetA.data'，代表命令行中若未提供 -f 或 --inputFile，則該選項將默認為 'datasetA.data'。

```python
# adding inputFile option for command line
optparser.add_option(
    "-f",
    "--inputFile",
    dest="input",
    help="filename containing .data",
    default='datasetA.data'
)
```

ii. Modify command line option "minSupport" : 將 default 值改為 '0.01'，代表命令行中若未提供 -s 或 --minSupport，則該選項將默認為 '0.01'。

```
# adding minSupport option for command line
optparser.add_option(
    "-s",
    "--minSupport",
    dest="minS",
    help="minimum support value",
    default=0.01,
    type="float",
)
```

iii. Load data from file : 發現每筆 transaction 前三筆皆不屬於 item，因此將其移除，且因為 input file 格式從原先 csv 檔改為 data 檔，所以也不需要 remove trailing comma。

```
def dataFromFile(fname):
    """Function which reads from the file and yields a generator"""
    with open(fname, "r") as file_iter:
        for line in file_iter:
            # line = line.strip().rstrip(",")  # Remove trailing comma
            record = frozenset(line.split()[3:])
            yield record
```

iv. Add a new command line option "outputPath" : users 可以透過 -p 來指定想要的 output file path。原本以為如果該資料夾不存在的話，程式可以自動新增以此命名的資料夾，但是卻發現這樣會 compile error。因此上網搜尋後學習到了 try 的指令，並使用此指令來建立以 users 給的 outputPath 命名的資料夾，如該資料夾已存在，則不會執行任何的動作，倘若資料夾不存在，則會執行 try 中的 os.makedirs(options.output)，以此名字建立新資料夾。

```
# adding outputPath option for command line
optparser.add_option(
    "-p",
    "--outputPath",
    dest="output",
    help="output file path",
    default='./'
)

(options, args) = optparser.parse_args()

try:
    """ 根據使用者的命名來 create a folder"""
    os.makedirs(options.output)
except:
    pass
```

v. Run Apriori : 新增一個名為 num_iter 的 dict output，其 key 為 iteration 次數，而對應的value 為一長度為 2 的 list，第一個 element 為 pruning 前的候選人數量，第二個 element 為 pruning 後的候選人數量。我在 runApriori function 的最一開始即使用 defaultdict(list) 宣告名為 num_iter 的 dict，並且在每次執行 returnItemWithMinSupport 的前面去根據第幾次的 iteration 來對對應 key 的 value append 要 input 進去此 function 的 itemSet 候選人的 length，而在此 function 執行結束後對相同 key 的 value append 此 function 的輸出變數的 length，亦即 pruning 過後的候選人數量。

```
items, num_iter = runApriori(inFile, minSupport)
```

```
def runApriori(data_iter, minSupport):
    """
    Run the apriori algorithm. data_iter is a record iterator
    Return both:
     - items (tuple, support)
     - num_iter {iter, [num of candidates before pruning, num of candidates after pruning]}
    """
    itemSet, transactionList = getItemSetTransactionList(data_iter)

    freqSet = defaultdict(int)
    num_iter = defaultdict(list)
    largeSet = dict()
    # Global dictionary which stores (key=n-itemSets,value=support)
    # which satisfy minSupport
    num_iter[1].append(len(itemSet))
    oneCSet = returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet)
    num_iter[1].append(len(oneCSet))
```

```
    currentLSet = oneCSet
    k = 2
    while currentLSet != set([]):
        largeSet[k - 1] = currentLSet
        currentLSet = joinSet(currentLSet, k)
        num_iter[k].append(len(currentLSet))
        currentCSet = returnItemsWithMinSupport(
            currentLSet, transactionList, minSupport, freqSet
        )
        num_iter[k].append(len(currentCSet))
        currentLSet = currentCSet
        k = k + 1


    def getSupport(item):
        """local function which Returns the support of an item"""
        return float(freqSet[item]) / len(transactionList)

    toRetItems = []
    for key, value in largeSet.items():
        toRetItems.extend([(tuple(item), getSupport(item)) for item in value])

    return toRetItems, num_iter
```

vi. Find frequent closed itemsets : 根據 result1 所產生的 frequent itemsets，找出 frequent closed itemsets。首先我創建一個空的 list 名為 closed_itemset，接著依序去

掃整個 frequent itemsets，如果發現有其他的 itemsets 包含了此 itemsets 且 support 值相等的話，則判斷此 itemsets 不為 frequent closed itemsets，最後將符合 frequent closed itemset 的 itemsets 加進 closed_itemset 中。

```
closed_itemset = find_closed_frequent_itemsets(items)
```

```
def find_closed_frequent_itemsets(frequent_itemsets):
    """
    Run the finding closed frequent itemsets algorithm
    Retuen:
    -closed_itemsets(itemset, support)
    """
    closed_itemsets = []

    for itemset, support in frequent_itemsets:
        is_closed = True
        for other_itemset, other_support in frequent_itemsets:
            if set(itemset) < set(other_itemset) and support == other_support:
                is_closed = False
                break

        if is_closed:
            closed_itemsets.extend([(itemset, support)])

    return closed_itemsets
```

vii. Write output to file task1 / 2 :

1. writeOutputToFile_task1 : 用於將名為 items 的 frequent itemsets 結果以及名為 num_iter 的 dict output 分別輸出到 result1.txt 和 result2.txt 文件中。其中，input items 格式為 [((itemset), support)]，itemset 為一組 frequent itemset，support 為該 frequent itemset 的支持度，我將這兩個值各別擷取出來，並根據 spec 需求，以每個 frequent itemset 的 support 值由大到小排序，support 值將以百分比且保留至小數點後第一位表示，最後將排序後的結果輸出至 result1 文件中 。而 result2 文件則是記錄 total number of frequent itemsets、index of iterations，以及 before pruning and after pruning 的候選人數量。首先透過 len(items) 得到 total number of frequent itemsets，再進行 num_iter.items()，得到 list[tuple]格式，便可擷取 key=index of iterations、value[0]=before pruning 的候選人數量和 value[1]=after pruning 的候選人數量。另外，輸出的檔案名稱為了符合 spec 需求，我將 options.input 和 options.output 以及 minSupport 當作 input，以便能夠擷取對應的資訊至輸出檔案名稱中。

```
def writeOutputToFile_task1(inFile, outFile, minSupport, items, num_iter):
    """Writes and stores task1 output in .txt"""
    dataset_name = os.path.basename(inFile).split('.')[0]
    output_filename = os.path.join(outFile, f"step2_task1_{dataset_name}_{minSupport}_result1.txt")
    with open(output_filename, 'w') as f:
        for item, support in sorted(items, key=lambda x: x[1], reverse=True):
            item_str = "{%s}" % ",".join(map(str, item))
            f.write("%.1f\t%s\n" %(support*100, item_str))

    output_filename = os.path.join(outFile, f"step2_task1_{dataset_name}_{minSupport}_result2.txt")
    with open(output_filename, 'w') as f:
        f.write("%s\n" %(len(items)))
        for key, value in num_iter.items():
            f.write("%s\t%s\t%s\n" %(key, value[0], value[1]))
```

2. writeOutputToFile_task2 : 用於將名為 closed itemset 的結果輸出到 result1.txt 文件中。首先透過 len(closed_itemset) 得到 total number of frequent closed itemsets，接著，為了各別擷取 closed_itemset 與對應的 support 值以及針對 support 值進行排序，我利用 "for item, support in sorted(...)" 的方式達到上述的效果，最後將 frequent closed itemsets 的數量與排序後的結果輸出至符合 spec 檔名需求的 result1 文件中。

```
def writeOutputToFile_task2(inFile, outFile, minSupport, closed_itemset):
    """Writes and stores task2 output in .txt"""
    dataset_name = os.path.basename(inFile).split('.')[0]
    output_filename = os.path.join(outFile, f"step2_task2_{dataset_name}_{minSupport}_result1.txt")
    with open(output_filename, 'w') as f:
        f.write("%s\n" %(len(closed_itemset)))
        for item, support in sorted(closed_itemset, key=lambda x: x[1], reverse=True):
            item_str = "{%s}" % ",".join(map(str, item))
            f.write("%.1f\t%s\n" %(support*100, item_str))
```

viii. Calculate computation time : 透過 time.time() function 來記錄 main function 第一行 code 的執行時間，接著在 task1 output file 後，再次使用 time.time() function 與一開始紀錄的 start time 相減，便能夠得到 task1 所花費的時間，之後在 code 最後面 task2也 output file 之後，進行相同的動作計算 task2 所花費的時間。

```
if __name__ == "__main__":

    start_time = time.time()
```

```
items, num_iter = runApriori(inFile, minSupport)

writeOutputToFile_task1(options.input, options.output, minSupport, items, num_iter)
print("Results of task1 of Step2 :")
print("--- Elapsed time of task1 : %s seconds ---" %(time.time() - start_time))

closed_itemset = find_closed_frequent_itemsets(items)

writeOutputToFile_task2(options.input, options.output, minSupport, closed_itemset)

print("Results of task2 of Step2 :")
print("--- Elapsed time of task2 : %s seconds ---" %(time.time() - start_time))
```

3. Step Ⅲ
   a. Descriptions of your mining algorithm (Relevant references, Program flow)
   程式透過 mlxtend library 中的 fpgrowth 函數來使用 FP-growth 演算法從 dataset 中找尋 frequent itemsets，並將結果輸出到指定格式的 .txt 文件中。FP-growth 演算法能夠有效率地識別 frequent itemsets，對於大型數據集較有利。以下針對程式中各個流程進行細說明：
      i. Command line解析：從 command line 中讀取初 inputFile、minSupport、outputPath 此三個變數。
      ii. 輸入檔案讀取：對於 inputFile 的 dataset 進行逐行讀取，並且將每行 transaction 的前三筆不屬於 item 的 data 捨棄掉，其餘 data append 到一 list 中。
      iii. Transaction list encode：使用 mlxtend.preprocessing.TransactionEncoder funcion 以及 pandas DataFrame function 將交易資料轉換為可以丟進去 FP-growth function 的 DataFrame 格式，其中每列對應一 transaction，每行對應一 item。假如該 transaction 沒有包含某 item，則該列的那行便會是 False，反之則會是 True。

iv. FP-growth 演算法：將上述 pandas DataFrame 格式的 transaction list 以及 command line 的 min support 值使用 fpgrowth function 尋找 frequent itemsets，此 function 將輸出一個同為 DataFrame 格式的結果，其中包含了 "itemsets" & "support" 兩欄位。

v. 結果輸出：由於 fpgrowth function 輸出格式與 Step2 的輸出格式已不相同，因此對此調整了 writeOutputToFile function 的輸入格式，讓此 function 可以接受格式為 DataFrame 的輸入，並同樣能產出與 Step2 相同格式的 result1 / 2 .txt 檔，由於此為 **non candidate based**，因此沒有 **before / after pruning** 的過程。

vi. 運算時間：於整個 main function 的最前面紀錄開始時間，並於程式最後計算整個程式執行所花費的時間。

b. Explain the main differences/improvements of your algorithm compared to Apriori (need to explain it in clear and well-structured ways)

以下是我使用 FP-growth algorithm 與 Apriori 的主要差異：

i. 掃描 dataset 的效率：Apriori 需要多次掃描整個 dataset。每次掃描都用於生成逐步增長的 candidate itemset，而這些 candidate itemset 會根據 min support 進行篩選。這種方法會導致大量的 I/O 操作，對於大型 dataset 來說，計算負擔很重。反而 FP-growth 僅需掃描 dataset 兩次。在第一次掃描中，收集 the support of each item 以生成 header table；在第二次掃描中，構建一個稱為 FP 樹的數據結構。藉由減少掃描次數降低 I/O 成本，因此特別適合處理大型 dataset。

ii. 候選人生成：Apriori 在每次 iteration 中生成 candidate itemset，每個 candidate itemset 都需要與整個 dataset 進行比對檢查，對於大型數據集效率較差。反之 FP-growth 不產生 candidate itemset，而是透過建立 FP tree 直接壓縮 data，方便高效率地直接識別 frequent patterns。

c. Compare the computation time of task1 with Step2 and Paste the screenshot of the computation time (try different settings of minimum support as Step2.py)

i. Step3.py -f datasetA.data -s 0.003 -p output : **the percentage of speedup : 97.208%**

ii. Step3.py -f datasetA.data -s 0.006 -p output : **the percentage of speedup : 96.446%**

iii. Step3.py -f datasetA.data -s 0.009 -p output : **the percentage of speedup : 95.783%**

iv. Step3.py -f datasetB.data -s 0.002 -p output : **the percentage of speedup : 99.803%**

v. Step3.py -f datasetB.data -s 0.004 -p output : **the percentage of speedup : 99.526%**

vi. Step3.py -f datasetB.data -s 0.006 -p output : **the percentage of speedup : 99.277%**

vii. Step3.py -f datasetC.data -s 0.005 -p output : **the percentage of speedup : 99.306%**

viii. Step3.py -f datasetC.data -s 0.01 -p output : **the percentage of speedup : 98.86%**

```
Results of task1 of Step3 :
--- Elapsed time of task1 : 30.386484146118164 seconds ---
```

ix. Step3.py -f datasetC.data -s 0.015 -p output : **the percentage of speedup : 98.295%**

```
Results of task1 of Step3 :
--- Elapsed time of task1 : 27.38575053215027 seconds ---
```

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

i. Difference from Step3.py without mlxtend fpgrowth function : 由於一開始有完成一版沒使用 mlxtend.fpgrowth function 的 Step3.py，但是發現對於 toy dataset 的運算時間異常的久，因此改為使用 mlxtend.fpgrowth function 的版本，就有發現對於 toy dataset 與 datasetB 與 datasetC 的運算時間跟沒使用 mlxtend.fpgrowth function 的版本相比快了許多，但奇怪的是對於 datasetA 的實驗結果卻是原先沒使用 mlxtend.fpgrowth function 的版本比較快，所以我分析各個 function 在處理 datasetA 的執行時間，在 minSupport=0.003 時，沒有使用 mlxtend fpgrowth 的版本總共花費了 0.15 秒左右，而使用 mlxtend fpgrowth 的版本則花費了 0.57 秒，比較兩者在執行每個 function 所花費的時間發現到，讀檔的部分兩者皆花費了 0.0015 秒左右，frequent itemsets mining 花費的時間也皆是 0.15 秒左右，但是在最後要輸出檔案的部分就有明顯的差距，沒有使用 mlxtend fpgrowth 的版本在輸出檔案的階段僅花費了 0.01 秒左右，而使用 mlxtend fpgrowth 的版本則花費了 0.4 秒，因此推測是由於 mlxtend fpgrowth function output format 在進行 sorting 時花費的時間比 dict 還要久所導致這個現象。附圖為各 function 的執行時間實驗結果，其中，Step3_bk.py 為未使用 mlxtend fpgrowth function 檔案，Step3.py 為有使用 mlxtend fpgrowth function 的版本也是上傳 E3 的版本。

```
michelle@DESKTOP-RE3242E:~/IBMGenerator/Apriori_python$ python3 Step3_bk.py -f datasetA.data -s 0.003 -p 123
--- Elapsed time of load data : 0.0015265941619873047 seconds ---
--- Elapsed time of fptree : 0.019515037536621094 seconds ---
--- Elapsed time of mine patterns : 0.14071154594421387 seconds ---
Results of task1 of Step3 :
--- Elapsed time of task1 : 0.1517777442932129 seconds ---
michelle@DESKTOP-RE3242E:~/IBMGenerator/Apriori_python$ python3 Step3.py -f datasetA.data -s 0.003 -p 123
--- Elapsed time of load data : 0.0014524459838867188 seconds ---
--- Elapsed time of data transform : 0.0047724246978759766 seconds ---
--- Elapsed time of fpgrowth : 0.1721363067626953 seconds ---
Results of task1 of Step3 :
--- Elapsed time of task1 : 0.5703959465026855 seconds ---
```

ii. Computation time between different datasets : 為了比較 Step3.py 在不同 dataset 數量的運算效率，我將使用具有 1,000 筆 transaction 的 datasetA、具有 100,000 筆 transaction 的 datasetB 以及具有 500,000 筆 transaction 的 datasetC，同時設定 minSupport=0.006，並計算三者在尋找 frequent itemset 的執行時間，計算結果分別花了約 0.143 秒、5.695 秒以及 32.378 秒，可以看出隨著 transaction 數量的增加，執行時間也顯著增加。這是因為 FP-growth 演算法雖然減少了 candidate itemset 生成的需求，但仍需要在 FP-tree 中掃描數據集並壓縮數據結構，例如，當不同 transaction 中出現相同的項目集前綴，FP-tree 會將這些前綴結合成同一分支，並增加這些項目集的計數。當 itemset 的規模擴大時，這個過程會消耗更多時間。尤其當 transaction 的數量增加至 500,000 筆時，FP-growth 在構建 FP-tree 及後續的模式挖掘過程中所需的計算資源增加，導致了更高的時間成本。相較於 Apriori 來說，FP-growth 在運算時間上仍然表現出顯著優勢，但在大規模數據集上，運行時間的增長依舊不可避免。這表明即便是優化後的 FP-growth 演算法，在處理大量交易數據時也會受到運算效率的影響。

iii. Largest dataset size the algorithm can handle : 除了 datasetA、datasetB以及 datasetC 外，在嘗試執行 toy dataset 後，發現當 dataset 的 transaction 數量約達到

1,000,000 筆時，FP-growth 演算法的計算資源需求明顯增加，執行時間也顯著延長，就如實驗結果顯示，當 minSupport=0.01 時，其執行時間延長至 63 秒，使得記憶體占用變得很高。這是因為隨著 transaction 數量增大，FP-tree 的建構變得愈加複雜，尤其是條件 FP-tree 的遞迴生成需要更多的記憶體空間來儲存中間結果。在更高的規模下，例如超過 2,000,000 筆 transaction 時，儘管 FP-growth 能夠處理這些數據，但受限於硬體資源，執行速度大幅下降，且在部分情況下可能因記憶體不足而出現運行失敗。這表明 FP-growth 在大規模數據集處理上雖具備優勢，但在實際應用中，仍須考慮硬體資源的限制。

iv. Relationship between itemset size and execution time : 為了瞭解 itemset 大小與執行時間的關係，我選擇執行 datasetB 與 datasetC，且在minSupport 為 0.002 的情況下，其執行時間分別為 9.24 秒和 46.52 秒，兩個 dataset 數量差 5 倍，執行時間也相差將近 5 倍。除此之外，我還測試其他不同 dataset，發現到在同樣 minSupport 下，不同數量的 dataset 與執行時間似乎存在接近線性的關係。為了更深入探究，我上網搜尋並了解到隨著 transaction 數量的增加，FP-growth 演算法的執行時間呈現「超線性但低於指數增長」的趨勢。此意味著儘管較大的數據集會增加執行時間，但 FP-growth 的高效 FP 樹結構有助於避免指數級增長，因為它減少了冗餘計算。