HW1 Report

A. How to Run the Code

Step II 執行說明

執行指令：

- 在 terminal 中輸入以下指令：（電腦為 mac 且 Python 版本為 3 以上需將指令中 `python` 改為 `python3`）：

```
python step2.py -f <dataset path> \
                -s <minsupport> \
                -p <output path>
```

  - **-f**：資料集，預設為 `datasetA`
  - **-s**：Minimum support，預設為 `0.003`
  - **-p**：txt 檔的輸出路徑，預設為當前目錄

Step III 執行說明

執行指令：

- 同 Step II 輸入以下指令：

```
python step3.py -f <資料集路徑> \
                -s <最低支持度> \
                -p <輸出文件路徑>
```

參數與 Step II 相同

B. Step II Report: Apriori Algorithm

1. The Modifications Made for Task1 and Task2

a. 更改 `dataFromFile()` 函數：

原本的 Apriori 程式無法正確讀入 Generator 所產生的 dataset。在網站中（IBMGenerator）查詢後得知資料格式為：

```
TID TID NITEMS ITEMSET
```

需要移除前三個欄位後才開始是真正的 `ITEMSET`，並且因為是用空格分隔，所以以空格去做 `split`。更改後的函數如下：

```python
def dataFromFile(fname):
    """讀取 dataset 並生成每一行 itemset 的 frozenset"""
    with open(fname, "r") as file_iter:
        for line in file_iter:
            line = line.strip()
            items = line.split(" ")
            items = items[3:]  # 移除前三個欄位
            yield frozenset(items)  # frozenset 為不可變動之集合
```

b. Result 1：

在 Task1 的 result 中，我們需要得出以下結果：

```
11.0            {186}
11.0            {408}
support(%)      itemset
...
```

執行程式後，資料會透過 `dataFromFile` 函數讀取，再來會進入 `runApriori` 函數：

```
def runApriori(data_iter, minSupport):
    """
    Run the Apriori algorithm. data_iter is a record iterator.
    Return both:
     - items (tuple, support)
     - stats
     - transactionList
    """
    itemSet, transactionList = getItemSetTransactionList(data_iter)
    freqSet = defaultdict(int)
    largeSet = dict()
    # Global dictionary which stores (key=n-itemSets, value=support)
    # which satisfy minSupport

    stats = []  # 新增 stats 列表以記錄每次迭代的候選數據

    # 處理第一輪 (1-項集) 並記錄候選數據
    num_candidates_before = len(itemSet)
    # 原始的 1-itemset 候選集數量
    oneCSet = returnItemsWithMinSupport(itemSet, transactionList, minSupport,
    freqSet)
    num_candidates_after = len(oneCSet)
    # Pruning 後的 1-itemset 候選集數量
    stats.append((1, num_candidates_before, num_candidates_after))
    # 紀錄第一輪的迭代狀況 (result2) 中需要用到

    currentLSet = oneCSet
    k = 2

    while currentLSet:
        largeSet[k - 1] = currentLSet
        currentCSet = joinSet(currentLSet, k)
        num_candidates_before = len(currentCSet)
        currentLSet = returnItemsWithMinSupport(currentCSet, transactionList,
        minSupport, freqSet)
        num_candidates_after = len(currentLSet)
        stats.append((k, num_candidates_before, num_candidates_after))
        k += 1
```

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

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

    return items, stats, transactionList
```

`runApriori` 函數會以每行迭代的形式呼叫 `getItemSetTransactionList` 函數，該函數得出了 `itemSet (1-項集)` 和 `transactionList`（由每筆交易集合構成）。接下來，程式會將資料轉入 `returnItemsWithMinSupport` 函數，此處我沒有做更改，和原版程式相同。

```
def getItemSetTransactionList(data_iterator):
    """從 dataset 中取出 item 作為 1-項集存入 itemSet
    並生成由每一筆交易之 frozenset 組成的 transactionList"""
    transactionList = []
    itemSet = set()
    for record in data_iterator:
        transaction = frozenset(record)
        # 取出 data 中每行 ITEMSET 並轉為 frozenset
        transactionList.append(transaction)
        # 將 frozenset 存入 transactionList
        for item in transaction:
            itemSet.add(frozenset([item]))
            # 將每行之每個 item 作為 1-項集存入 itemSet
    return itemSet, transactionList
```

接下來會進入 `returnItemsWithMinSupport` 函數，值得特別注意的是，我在計算這部分時為了提升效能，選擇了使用多重處理器來幫助計算。這部分的加速也讓我在後續與 Step III 的 speedup 上和本來預估的結果（比 FP-Growth 演算法慢很多）有所不同。也就是其實 Step II 在經過硬體加速後，效能並不比 Step III 程式差多少（在此次作業的資料規模上）。函數詳細運作如程式註解。

```
def returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet):
    """計算 itemSet 中各項支持度，傳回符合 minSupport 的項集"""
    _itemSet = set()  # 用來存儲符合 minSupport 的頻繁項集
    num_transactions = len(transactionList)  # 總交易數，用於計算支持度

    # 使用多重處理器加速的部分
    pool = Pool()  # 建立多處理池
    args = [(item, transactionList) for item in itemSet]  # 將 1-項集傳入 args

    # 使用多重處理器來計算每個候選項集的支持度
    results = pool.map(count_support, args)  # `count_support` 計算支持度
    pool.close()  # 關閉處理池，防止新任務提交
    pool.join()   # 等待所有進程完成

    # 處理支持度計算結果
    for item, count in results:
        freqSet[item] += count  # 更新全局計數字典中的支持度計數
        support = float(count) / num_transactions  # 計算支持度
        if support >= minSupport:  # 判斷是否符合最小支持度
            _itemSet.add(item)     # 若符合則加入頻繁項集集合 `_itemSet`

    return _itemSet  # 返回符合支持度要求的頻繁項集
```

與原版程式不同，為了程式碼的加速運行，我也另外將計算支持度的函數獨立分出如下：

```
def count_support(args):
    """獨立出來的支持度計算函數"""
    item, transactionList = args
    count = sum(1 for transaction in transactionList
    if item.issubset(transaction))
    return (item, count)
```

再來 `runApriori` 程式將會繼續進行。在從 `returnItemsWithMinSupport` 函數中取得 `itemset` 之後，將會進入 while 迴圈。在每輪 while 迴圈會把頻繁集存入 `largeSet` 中，並利用 `joinSet` 函數將項集逐漸增加，迴圈結束即可得到所有項集的頻繁集。

```
oneCSet = returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet)
num_candidates_after = len(oneCSet)
# Pruning 後的 1-itemset 候選集數量
stats.append((1, num_candidates_before, num_candidates_after))
# 紀錄第一輪的迭代狀況（result2）中需要用到
largeSet[1] = oneCSet

currentLSet = oneCSet
k = 2

while currentLSet:
    largeSet[k - 1] = currentLSet
    # 將當前頻繁項目集儲存到 largeSet 中的第 k-1 層
    currentCSet = joinSet(currentLSet, k)
    # 透過 joinSet 函數產生候選項目集，項目數量為 k
    num_candidates_before = len(currentCSet)
    # 計算當前候選項目集的項目數量
    currentLSet = returnItemsWithMinSupport(
        currentCSet, transactionList, minSupport, freqSet
    )  # 過濾候選項目集，僅保留符合最小支持度的項目
    num_candidates_after = len(currentLSet)
    # 計算過濾後的候選項目數量
    stats.append((k, num_candidates_before, num_candidates_after))
    # 記錄當前 k 層的候選數量變化
    k += 1  # 增加項目集的大小，進入下一層的計算
```

c. Result 2:

Result2 任務中每輪候選集 pruning 前後的資料在 result1 執行時已經透過 `stats` 來記錄，如以下程式碼所示：

```
stats.append((1, num_candidates_before, num_candidates_after))
stats.append((k, num_candidates_before, num_candidates_after))
```

再來 result2 的資料會在主程式執行 `printStats` 函數時將資料輸入進檔案，並同時透過 `sum` 函數將 `stats` 中的第三項頻繁集數量做疊加，並且輸出進檔案的第一行。最後再將本來的 `stats` 資料輸出，即可完成 result2 的任務。

```
def printStats(stats, outputFile):
    total_itemsets = sum([after for _, _, after in stats])
    # 第三項之 after 即每輪之頻繁集數量，加總後即為頻繁集總數
    with open(outputFile, 'w') as f:
        f.write(f"{total_itemsets}\n")
        for stat in stats:
            iteration, before, after = stat  # 將每輪資訊分項輸出
            f.write(f"{iteration}\t{before}\t{after}\n")
```

d. Task 2:

以下是 Task2 在主程式中的程式碼，除了第二行中的 `findClosedItemsets` 函數，其他部分大致上和 Result1 的檔案輸出沒有差異。接下來我在這個段落中將會講解 `findClosedItemsets` 函數。

```
# -------------------- Task 2 --------------------
print("\nRunning Task2:")
start_time_task2 = time.time()
# 使用 Task1 中的 frequent itemset 進一步計算 closed frequent itemset
closed_items = findClosedItemsets(items_task1)
# 输出 Task 2 结果
with open(output_file3, 'w') as f:
    total_closed_itemsets = len(closed_items)
    f.write(f"{total_closed_itemsets}\n")
    closed_items = sorted(closed_items, key=lambda x: x[1], reverse=True)
    print(f"Total number of frequent closed itemset: {total_closed_itemsets}")
    for itemset, support in closed_items:
        support_percent = round(support * 100, 1)
        itemset_str = ','.join(sorted(itemset))
        f.write(f"{support_percent}\t{{{itemset_str}}}\n")

elapsed_time_task2 = time.time() - start_time_task2
print(f"Task2 time: {elapsed_time_task2:.4f} seconds")
```

以下是 `findClosedItemsets` 函數，為避免篇幅過於複雜，我把大部分的解釋放在程式的註解當中。首先會宣告一個字典 `supportData`，其 key 為已經經過 Task1 的 `frequent itemset`，value 則為相對應的 `support`。再宣告一個存放閉鎖頻繁集的 `closedItemsets`。接著從 `supportData` 中以 list 的形式讀取每行的 `frequent itemset`，並

確認是否為閉鎖頻繁集。若為閉鎖頻繁集，則將 `isClosed = True` 並將此 `itemset` 加入 `closedItemsets` 中；若不是閉鎖頻繁集，則將 `isClosed = False` 並 `break` 迴圈。

```
def findClosedItemsets(items):
    """計算 Closed frequent itemset"""
    # 宣告 supportData 字典並且 value 為每 frozenset(key) 的 support
    supportData = dict()
    for itemset, support in items:
        supportData[frozenset(itemset)] = support
    closedItemsets = []  # 存儲閉鎖頻繁集
    allFrequentItemsets = list(supportData.keys())
    # 將頻繁集轉換為列表方便檢查是否為閉鎖頻繁集

    for itemset in allFrequentItemsets:
        isClosed = True
        itemsetSupport = supportData[itemset]
        # 取得該行 itemset 的 support(value)
        for otherItemset in allFrequentItemsets:
            if len(otherItemset) > len(itemset):
                # otherItemset 是比較大的那個
                if itemset.issubset(otherItemset):
                    # 當前 itemset 是否為 otherItemset 之子集
                    if supportData[otherItemset] == itemsetSupport:
                        # support 相同而且比較大，代表他不是閉鎖頻繁集，改 False 並退出
                        isClosed = False
                        break
        if isClosed:
            # 若沒有比他大的 itemset，他就是閉鎖頻繁集，加入 closedItemsets
            closedItemsets.append((tuple(itemset), itemsetSupport))

    return closedItemsets
```

得到 `closedItemsets` 之後，即可將結果傳回主程式中的 `closed_items` 變數並輸入進檔案，完成 Task2 的作業要求。

---

2. The Restrictions

1. **資料規模**：雖然我使用了多處理器加速，但隨著資料規模增大，計算時間仍顯著增加，這是處理數據一定會面臨的問題。

```
pool = Pool()  # 使用多處理加速支持度計算
results = pool.map(count_support, args)
pool.close()
pool.join()
```

2. **內存消耗**：程式碼當中的 `largeSet` 和 `freqSet` 用來儲存大量的頻繁項集，

```
freqSet = defaultdict(int)
largeSet = dict()
```

   當資料量大時，內存需求會顯著增加。

---

3. Problems Encountered in Mining

1. **Memory Overflow 問題**
   處理大規模數據時，內存需求過高。

```
transactionList = list(data_iter)  # 直接將資料載入內存
```

   將整個資料集載入 `transactionList` 中，對於大型數據可能會導致內存溢出。
2. **支持度計算緩慢**
   尤其在高維資料中，支持度計算時間較長，例如：

```
count = sum(1 for transaction in transactionList
if item.issubset(transaction))
```

   這段程式碼必須對每個交易執行一個檢查操作，並且無法優化為部分數據處理。當數據量或維度增加時，支持度計算的耗時也會急劇上升。
3. **閉合項集識別準確性**
   為確保準確性，進行了多次測試和除錯。

```
if supportData[otherItemset] == itemsetSupport:
    isClosed = False
    break
```

在閉合頻繁項集的檢查中，需逐一比較支持度以確保準確性，這在大規模資料集下會非常耗時。

---

4. Observation and Discoveries

在多次執行 Task1 以後，我發現 Task1 大多數的計算可以經過多處理器優化，且效果非常顯著。以我的 MacBook Air（8 核 CPU）為例，本來在 `datasetB` 0.2% 支持度下約需 35 分鐘完成整個任務（Task1 + Task2），但在加速後只需 13 分鐘即可完成。以下以舉例且較為簡單的方式解釋我在程式碼中如何利用多處理器進行優化，主要是利用 `multiprocessing` 中的 `Pool` 進行加速：

由於我的程式碼中利用 `pool` 計算支持度的程式碼主要在 `results = pool.map(count_support, args)` 這行。接下來演示 `itemset` 在不同核心上的並行計算過程，每個核心同時計算一個 `itemset` 的支持度。假設電腦有 4 核心，可以並行處理最多 4 個計算任務。

候選項集（`itemSet`）：

```
itemSet = {frozenset({1}), frozenset({2}), frozenset({3})}
```

交易列表（`transactionList`）：

```
transactionList = [
    frozenset({1, 2, 3}),
    frozenset({1, 3}),
    frozenset({2, 3}),
    frozenset({1}),
    frozenset({3})
]
```

步驟 1： 準備多重處理的任務參數

將每個候選項集與交易列表組合成一個元組 `(item, transactionList)`，並存入 `args` 列表中：

```
args = [
    (frozenset({1}), transactionList),
    (frozenset({2}), transactionList),
    (frozenset({3}), transactionList)
]
```

步驟 2： `pool.map` 分配計算任務

`pool.map(count_support, args)` 會將 `args` 中的每個元組傳遞給 `count_support` 函數，並行計算每個候選項集的支持度。假設電腦有 4 個 CPU 核心，分配如下：

核心	分配的任務
核心 1	`count_support(frozenset({1}), transactionList)`
核心 2	`count_support(frozenset({2}), transactionList)`
核心 3	`count_support(frozenset({3}), transactionList)`
核心 4	無任務分配

各核心的計算過程：

- **核心 1**：
  - 執行 `count_support(frozenset({1}), transactionList)`
  - 計算 `{1}` 在 `transactionList` 中的出現次數，結果為 `3`。
- **核心 2**：
  - 執行 `count_support(frozenset({2}), transactionList)`
  - 計算 `{2}` 在 `transactionList` 中的出現次數，結果為 `2`。
- **核心 3**：
  - 執行 `count_support(frozenset({3}), transactionList)`
  - 計算 `{3}` 在 `transactionList` 中的出現次數，結果為 `4`。

步驟 3： 彙總結果

所有進程完成後，`pool.map` 會將每個計算結果收集到 `results` 列表中：

```
results = [
    (frozenset({1}), 3),  # 核心 1 的計算結果
    (frozenset({2}), 2),  # 核心 2 的計算結果
    (frozenset({3}), 4)   # 核心 3 的計算結果
]
```

- **並行處理**：每個核心同時處理一個候選項集的支持度計算，將總計算時間縮短。
- **加速效果**：假設每個支持度計算獨立完成需要 1 秒，使用 4 核心的並行處理可以將總計算時間縮短至僅需 1 秒（而非逐個順序計算的 3 秒）。

這樣的並行計算通過 `pool.map` 高效利用多核處理器，使得支持度計算大幅加速。

以上是針對 Step II 中最有用的觀察，再花了許多時間在此作業上以後，我發現效能提升是我覺得最重要的事情了。

---

5. Computation Time

以下為 Task2/Task1 之 time ratio 截圖：

Dataset A:

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetA.data -s 0.003 -p testing
Running Task1:
Task1 time: 6.5232 seconds
total number of frequent itemset: 7776

Running Task2:
Task2 time: 1.1278 seconds
total number of frequent closed itemset: 5116

 time ratio of (Task2 / Task1 * 100%): 17.29%
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetA.data -s 0.006 -p testing
Running Task1:
Task1 time: 0.7708 seconds
total number of frequent itemset: 909

Running Task2:
Task2 time: 0.0156 seconds
total number of frequent closed itemset: 909

 time ratio of (Task2 / Task1 * 100%): 2.03%
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetA.data -s 0.009 -p testing
Running Task1:
Task1 time: 0.5493 seconds
total number of frequent itemset: 414

Running Task2:
Task2 time: 0.0030 seconds
total number of frequent closed itemset: 414
```

（由於截圖上的失誤，此處重新補充上圖 `datasetA` 在 `s` 為 `0.003` 以下的 Task2/Task1 約莫為 `0.5%`）

可以觀察到，當數據量越來越小（支持度門檻提高），Task2 和 Task1 的差距就越來越小。我理解為，當頻繁集較少時，閉鎖頻繁集的計算也相對較為容易。但在 Task1 挖掘頻繁集時，仍需看整體資料，時間無法有效下降，導致分子越來越小，造成比例變小的狀況。

Dataset B:

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetB.data -s 0.002 -p testing
Running Task1:
Task1 time: 829.9797 seconds
total number of frequent itemset: 5820

Running Task2:
Task2 time: 0.6485 seconds
total number of frequent closed itemset: 5817

 time ratio of (Task2 / Task1 * 100%): 0.08%
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetB.data -s 0.004 -p testing
Running Task1:
Task1 time: 229.5058 seconds
total number of frequent itemset: 1445

Running Task2:
Task2 time: 0.0470 seconds
total number of frequent closed itemset: 1445

 time ratio of (Task2 / Task1 * 100%): 0.02%
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetB.data -s 0.006 -p testing
Running Task1:
Task1 time: 136.5609 seconds
total number of frequent itemset: 667

Running Task2:
Task2 time: 0.0105 seconds
total number of frequent closed itemset: 667

 time ratio of (Task2 / Task1 * 100%): 0.01%
```

再來是 `datasetB`，由於 `datasetB` 的閉鎖頻繁集數量與頻繁集數量大致相同，只需要一輪的比較即可確認為閉鎖頻繁集並加入 `closed_itemsets` list 中。由於 `datasetB` 的規模較大，Task1 花很多時間，以至於所有比率幾乎都是 `0.0X%`。

Dataset C:

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetC.data -s 0.005 -p testing
Running Task1:
Task1 time: 904.6214 seconds
total number of frequent itemset: 911

Running Task2:
Task2 time: 0.0170 seconds
total number of frequent closed itemset: 911

 time ratio of (Task2 / Task1 * 100%): 0.00%
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetC.data -s 0.015 -p testing
Running Task1:
Task1 time: 298.4092 seconds
total number of frequent itemset: 247

Running Task2:
Task2 time: 0.0010 seconds
total number of frequent closed itemset: 247

 time ratio of (Task2 / Task1 * 100%): 0.00%
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetC.data -s 0.01 -p testing
Running Task1:
Task1 time: 557.8277 seconds
total number of frequent itemset: 356

Running Task2:
Task2 time: 0.0022 seconds
total number of frequent closed itemset: 356

 time ratio of (Task2 / Task1 * 100%): 0.00%
```

最後是 `datasetC`，由於在資料規模和支持度的綜合條件下，`datasetC` 和 `datasetB` 的規模大致相同（可從 Task1 的時間觀察得出），但 `datasetC` 在各個支持度下 Task1 所得出的頻繁集非常少，所以在 Task2 時時間會非常低。在小數點四捨五入到後兩位的計算底下，全部都是 `0.00%`，代表 Task2 幾乎不需花費任何時間即可馬上得出。

不同支持度設置

以 `datasetA` support = `0.004` 為例：

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step2.py -f datasetA.data -s 0.004

Running Task1:
total number of frequent itemset: 3380
Task1 time: 2.4425 seconds

Running Task2:
total number of frequent closed itemset: 2529
Task2 time: 0.2233 seconds

 time ratio of (Task2 / Task1 * 100%): 9.14%
```

Computation Time 觀察結論：

以上為在不同 dataset 中 Task2/Task1 的觀察與解釋。可得出結論，Task1 的時間比率很難下降，但 Task2 在閉鎖頻繁集數量與頻繁集數量相等，或頻繁集數量本身很少的狀況下，時間幾

乎可以忽略（比起 Task1）。

以上內容包括了 Task1 和 Task2 的修改、遇到的限制、問題、觀察與計算時間截圖。請將 `screenshot_path.png` 替換為實際的計算時間截圖路徑。

C. Step III Report: FP-Growth

1. Descriptions of Your Mining Algorithm

Resources: mlxtend GitHub

我採用 `FP-Growth`（頻繁模式增長）作為 Step III 的演算法。`FP-Growth` 通過構建 FP-Tree 來避免冗長的候選集生成，以下為 FP-Growth 的步驟描述。

首先，`FP-Growth` 會利用 `dataFromFile` 函數將 `itemset` 從文件中讀取，並轉換為不可變集合 (`frozenset`) 的格式，以便有效管理和處理每筆交易。

```
def dataFromFile(fname):
    with open(fname, "r") as file_iter:
        for line in file_iter:
            line = line.strip()
            items = line.split()
            items = items[3:]
            yield items
```

此部分和 Step II 沒有差異。在主程式中，會將交易列表轉換為二進制矩陣，每行代表一筆交易，每列代表一個商品 ID。商品出現用 `1` 表示，未出現用 `0` 表示。這種表示方式便於後續的支持度計算，因為每個商品的出現頻率可以透過矩陣操作快速計算。此部分由主函數中的以下程式碼達成:

```
transaction_list = list(data)
df = pd.DataFrame([{item: 1 for item in t} for t in transaction_list]).fillna(0)
```

`item` 如果是在 `transaction` 的某行內，就會以字典且 `item:1` 的格式加入 `list` 生成式中並轉換為矩陣；其餘該行沒出現的 `item` 則由 `fillna(0)` 填入 `0`。

範例:

	0	14	25	30	67	78	86	88	131	139	210	221	308	366
itemset1	0	0	0	0	1	1	0	0	0	1	0	0	0	1
itemset2	0	1	1	0	0	0	0	0	0	0	1	1	0	0
itemset3	1	0	0	1	0	0	1	1	1	0	0	0	1	0

再來執行:

```
frequent_items_df = fpgrowth(df.astype('bool'), min_support=min_support)
```

正式開始執行 `FP-Growth` 函數，並將矩陣中的 `1` 轉換為 `bool`（`bool` 型態佔用記憶體比 `int` 小）。執行後，以下列程式碼取得執行結果並輸入進檔案內，和 Step II 無太大差異。

```
frequent_items = [
    (frozenset(row['itemsets']), row['support'])
    for _, row in frequent_items_df.iterrows()
]
```

接下來講解 `FP-Growth` 的主要部分，以下是 `FP-Growth` 函數，我將從這裡作為出發點逐步講解整個的執行流程。

```
def fpgrowth(df, min_support=0.5):
    fptree, rank_map, index_to_name = setup_fptree(df, min_support)
    min_support_count = math.ceil(min_support * len(df.index))
    generator = fpg_step(fptree, min_support_count)
    return generate_itemsets(generator, min_support, index_to_name)
```

在程式的第一行會進入 `setup_fptree` 函數，並將結果指派給 `fptree`，`rank_map`，`index_to_name` 等變數。該函數會計算支持度並以 NumPy 陣列的方式操作資料。`item_support` 是存項目支持度的 NumPy 陣列，`frequent_items` 則是篩選出支持度大於或等於最小支持度的項目。`sorted_indices` 透過 `argsort` 函數得出 `frequent_items` 在 `item_support` 中排序後的索引，而 `rank_map` 則利用 `sorted_indices` 的索引去 `frequent_items` 抓取其在 `item_support` 中的索引，並給予從 1 開始的排名編號。由於變數

轉換過程稍微複雜，因此在這裡做上述說明。由於在 class 中定義的 node 具有 `count` attribute，`FP-Growth` 會在每輪建 tree 時將出現過的 item 的 `count` 疊加，並且在最後輸出一個鏈狀的 tree。

```
def setup_fptree(df, min_support):
    num_transactions = len(df.index)  # 交易數據的筆數（交易的總數）
    transaction_data = df.values  # 將 DataFrame 的值提取成 NumPy 陣列，方便計算
    # 計算每個項目的支持度，表示每個項目出現的比例：
    item_support = np.sum(df.values == 1, axis=0) / float(num_transactions)
    # np.sum(df.values == 1, axis=0) 沿著 column 疊加計算每個項目出現的次數並除以總長
    # item_support 輸出即各項支持度之 NumPy 陣列
    frequent_items = np.nonzero(item_support >= min_support)[0]
    # 篩選出支持度大於或等於最小支持度 (min_support) 的項目 index

    sorted_indices = item_support[frequent_items].argsort()
    # 抓出 NumPy 陣列中支持度排序後的索引
    # 將排序（排名）後索引加上 rank 編號
    rank_map = {item: i for i, item in enumerate(frequent_items[sorted_indices])}

    # 建立索引到 item 的映射
    index_to_name = {idx: item for
    idx, item in enumerate(df.columns)}

    fptree = FPTree(rank_map)  # 建構 FP-Tree
    for i in range(num_transactions):
        itemset = [
            item for item in np.where(transaction_data[i, :])[0]
            if item in rank_map
        ]
        itemset.sort(key=rank_map.get, reverse=True)
        fptree.insert_itemset(itemset)

    return fptree, rank_map, index_to_name
```

以下是建 tree 的簡略流程：

範例 **FP-Tree** 結構：

- **Itemset 1:** `C -> A`

- **Itemset 2:** `A -> B`
- **Itemset 3:** `C -> B`

假設支持度的排序為：`C` 的支持度最高，其次是 `A`，然後是 `B`。

插入交易 1 (`C -> A`)

FP-Tree 初始狀態為空，插入項目 `C` 和 `A`，生成路徑 `C -> A`。

```
(root)
   |
   C (1)
   |
   A (1)
```

插入交易 2 (`A -> B`)

`A` 不在現有的根路徑中，因此從根節點開始，插入 `A` 和 `B`，生成新的路徑 `A -> B`。

```
(root)
   |
   C (1)
   |
   A (1)

   A (1)
   |
   B (1)
```

插入交易 3 (`C -> B`)

`C` 已存在於根節點的分支，因此更新 `C` 的計數為 2。插入 `B` 為 `C` 的子節點，形成路徑 `C -> B`。

```
(root)
   |
   C (2)
   |
   A (1)
   |
   B (1)
```

- `C` 的支持度最高，因此在根節點。
- `C` 和 `A` 的路徑共享，同時 `C` 的分支也包含 `B`。

上述內容大略解釋了建 tree 的過程。接下來在 `fpgrowth` 函數中得出 `fptree`，`rank_map`，`index_to_name` 等變數後，`min_support_count` 會將支持度乘上長度變成出現次數，並將次數帶入 `fpg_step` 函數，並將結果放入變數 `generator` 中，讓後續的 `generate_itemsets` 函數去做輸出前的轉換。以下是 `fpg_step` 函數：

```python
def fpg_step(tree, min_support_count):
    # 把 FP-Tree 的項目抓出來放進一個 items 中
    items = list(tree.nodes.keys())

    # 檢查 FP-Tree 是否是鏈狀
    if tree.is_path():
        # 設定 max_size 為 items 的數量加 1，生成所有可能大小的組合
        max_size = len(items) + 1

        # 從 1 生成組合大小，直到大小達到 max_size
        for size in range(1, max_size):
            # 生成 k 項集
            for itemset in itertools.combinations(items, size):
                # 計算項目支持度，取所有 item 的出現最少的次數作為支持度
                support = min([tree.nodes[i][0].count for i in itemset])
                # 傳回 itemset 的 support 和 itemset
                yield support, tree.conditional_items + list(itemset)
    else:
        # 如果 FP-Tree 非鏈狀
        for item in items:
            # 計算 item 的總次數，作為 support
            support = sum([node.count for node in tree.nodes[item]])
            yield support, tree.conditional_items + [item]

    # 如果 tree 不是鏈狀，則對每個 item 生成一個條件子樹進行更深入的處理
    if not tree.is_path():
        for item in items:
            # 生成以 item 為條件的子樹，並傳入 min_support_count
            cond_tree = tree.conditional_tree(item, min_support_count)
            # 遞迴呼叫 fpg_step，進行下一步的 itemset 生成
            for support, itemset in fpg_step(cond_tree, min_support_count):
                yield support, itemset
```

在 `fpg_step` 函數中，會先檢查 `tree` 是否為單一路徑。如果 `tree.is_path()` 為 `True`，代表 FP-Tree 中所有節點連成一條鏈狀結構，`fpg_step` 會直接生成所有 `itemset`，並使用 `itertools.combinations` 來列出每個 `itemset` 的所有可能子集。最後對 `itemset` 中的每個項目取 `count` 的最小值來計算支持度，確保他們有在 `min_support` 的標準內，並以 `yield` 返回支持度和項目集。

如果 `tree` 不是鏈狀，會逐一計算 `item` 的 `support`（利用累加 `node` 的 `count`）。接著 `fpg_step` 函數會執行 `tree.conditional_tree(item, min_support_count)`，將每個 `item` 生成條件子樹，並在子樹上遞迴呼叫 `fpg_step` 完成 `frequent itemset` 的計算，並將結果指定到 `generator` 中。以下稍微解釋並註解 `generate_itemsets` 函數。

```
def generate_itemsets(generator, min_support, index_to_name):
    """從 generator 中生成頻繁項集"""
    itemsets = []
    support_values = []
    # 從 generator 中迭代出支持度和頻繁項集
    for support, itemset in generator:
        # 將項集中的每個索引轉換成實際 item
        named_itemset = set(index_to_name[item] for item in itemset)

        # 把轉換後的項集名稱加到 itemsets 列表
        itemsets.append(named_itemset)

        # 把支持度值加到 support_values 列表
        support_values.append(support)

    # 建立一個 DataFrame 將支持度值和頻繁項集存成表格格式，方便後續處理
    result_df = pd.DataFrame({"support": support_values, "itemsets": itemsets})

    # 過濾出支持度大於或等於 min_support 的 itemset 並返回結果
    return result_df[result_df["support"] >= min_support]
```

`generate_itemsets` 函數的目的是從 `generator` 中取出 `frequent itemset`，並將 `index` 轉換成 `item` 的數字（在 IBM 的 dataset 中 item 為數字），最後只保留符合 `min_support` 的 `itemset`。

執行完 `generate_itemsets` 後，`FP-Growth` 的部分就結束了。可以回到主程式，利用以下程式碼將 `frequent itemset` 的 `DataFrame` 轉換為許多 `frozenset` 組成的 `list`，並將結果輸入檔案中，完成 Task1（Result1）的任務。

```
frequent_items = [
    (frozenset(row['itemsets']), row['support'])
    for _, row in frequent_items_df.iterrows()
]
```

2. Differences/Improvements in Your Algorithm

以下是已調整為 Markdown 文件格式的內容：

---

1. 排序映射 (`rank_map`) 的使用

在 `FPTree` 類別中引入了 `rank_map` 來排序項目：

```
class FPTree:
    def __init__(self, rank_map=None):
        ...
        self.rank_map = rank_map
```

- **優化效果**：`rank_map` 提前將項目根據支持度排序，使得每次插入新的項目集時可以直接根據這個順序排列。這避免了在插入過程中每次重新排序，顯著提高了樹的構建速度。

---

2. 條件樹的過濾和排序 (`conditional_tree` 方法)

`conditional_tree` 方法用來生成條件樹，同時進行了項目的過濾和排序：

```
def conditional_tree(self, cond_item, min_support_count):
    ...
    filtered_items = [item for item in item_count
    if item_count[item] >= min_support_count]
    filtered_items.sort(key=item_count.get)
    updated_rank = {item: i for i, item in enumerate(filtered_items)}
    ...
```

- **優化效果**：這段程式碼過濾掉低於 `min_support_count` 的項目，從而減少了條件樹的大小，降低了記憶體需求。並且，過濾後的項目根據支持度重新排序，生成 `updated_rank` 映射，確保條件樹的生成和遍歷更高效。

---

3. FP-Tree 建立過程中的支持度預計算 (`setup_fptree`)

在 `setup_fptree` 函數中，支持度被提前計算並篩選出頻繁項目：

```
def setup_fptree(df, min_support):
    ...
    item_support = np.sum(df.values == 1, axis=0) / float(num_transactions)
    frequent_items = np.nonzero(item_support >= min_support)[0]
    ...
```

- **優化效果**：將支持度提前換算成具體的計數，僅保留滿足 `min_support` 條件的頻繁項目，減少不必要的資料處理，從而縮短了 FP-Tree 建立的時間。

---

4. 頻繁項集生成時的路徑優化 (`fpg_step` 方法)

在 `fpg_step` 方法中，檢查樹是否為單一路徑，並在該情況下快速生成所有組合：

```
def fpg_step(tree, min_support_count):
    if tree.is_path():
        max_size = len(items) + 1
        for size in range(1, max_size):
            for itemset in itertools.combinations(items, size):
                support = min([tree.nodes[i][0].count for i in itemset])
                yield support, tree.conditional_items + list(itemset)
    ...
```

- **優化效果**：當 FP-Tree 是單一路徑時，所有頻繁項集可以直接由組合生成，而無需建立更多的條件樹，這樣可以避免不必要的計算，顯著加速頻繁項集的生成過程。

---

3. Computation Time

Dataset A:

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetA.data -s 0.003 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 7776
Task 1 computation time: 0.1440 s
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetA.data -s 0.006 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 909
Task 1 computation time: 0.0478 s
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetA.data -s 0.009 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 414
Task 1 computation time: 0.0394 s
```

Step2: Dataset A

Support	Execution Time (Step2)	Execution Time (Step3)	Speed Up
0.003	6.5232	0.1440	97.79%
0.006	0.7708	0.0478	93.80%
0.009	0.5493	0.0394	92.83%

Dataset B:

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetB.data -s 0.002 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 5820
Task 1 computation time: 5.4293 s
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetB.data -s 0.004 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 1445
Task 1 computation time: 4.1550 s
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetB.data -s 0.006 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 667
Task 1 computation time: 3.9852 s
```

Step2: Dataset B

Support	Execution Time (Step2)	Execution Time (Step3)	Speed Up
0.002	829.9797	5.4293	99.35%
0.004	229.5058	4.1550	98.19%
0.006	136.5609	3.9852	97.08%

Dataset C:

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetC.data -s 0.005 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 911
Task 1 computation time: 31.7786 s
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetC.data -s 0.01 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 356
Task 1 computation time: 29.6435 s
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetC.data -s 0.015 -p testing
Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 247
Task 1 computation time: 28.2249 s
```

Step2: Dataset C

Support	Execution Time (Step2)	Execution Time (Step3)	Speed Up
0.005	904.6214	31.778	96.49%
0.010	557.8277	29.643	94.69%
0.015	298.4092	28.2249	90.54%

以上的表格為 speed up 的百分比（對比到 Step2 的 Task1），所有的 speed up 都達到 90% 以上，效能提升顯著。

不同支持度設置

以 `datasetA` support = `0.004` 為例：

```
richardlin@Richarddebijixingdiannao nyLAB1 % python3 step3.py -f datasetA.data -s 0.004

Start Task 1:  find all frequent itemsets
number of the frequent itemsets: 3380
Task 1 computation time: 0.0843 s
```

以上為 `datasetA` 在 support 為 `0.004` 的執行（Step3），與 Step2 的執行時間 `2.4425` 秒來說，speed up 為 `(2.4425 - 0.0843) / 2.4425` 約為 `96%`。在不同的 support 設定中仍然有很好的效能提升。

---

4. Scalability in Terms of Different Dataset Sizes

1. FP-Tree 的構建

- **描述**：構建 FP-Tree 隨著資料集的增長，會在內存中儲存所有的項目節點與其關聯的頻繁項目集，這意味著資料集大小的增加會導致 FP-Tree 樹的深度與寬度增加。
- **影響**：由於此資料結構的設計特性，若資料集包含大量不同項目，FP-Tree 的構建速度將大幅度降低，並且會佔用大量記憶體。

2. 資料集的頻率計算與排序

- **描述**：`setup_fptree` 函數中通過計算每個項目在交易中的出現頻率，並篩選出頻繁項目以構建排名映射。
- **影響**：隨著資料集的增加，這個頻率計算操作的時間複雜度也會相應增加。排序操作也可能對大規模資料集造成負擔，因此資料增長可能導致排序時間顯著增長。

3. 遞迴構建條件 FP-Tree

- **描述**：`fpg_step` 遞迴地構建條件 FP-Tree 以找到更小的頻繁項目集。
- **影響**：當資料集變大，頻繁項目組合數量隨之增加，遞迴的次數也隨之增長，尤其是在資料集包含較多重複或高度關聯的項目時，會影響遞迴的深度與頻率。因此，這種遞迴結構可能隨著資料量增大而成為瓶頸。

4. 記憶體佔用

- **描述**：FP-Tree 是一種緊湊的資料結構，但隨著資料集的擴展，FP-Tree 的尺寸也將增大，進一步增加記憶體需求。
- **影響**：若資料集過大，系統內存可能會無法支援，導致記憶體溢出問題。

以上四點是在面臨不同大小的 dataset（通常為變大）時可能遇到的問題。

以上是我的HW1 report感謝助教撥冗查閱。