資料探勘作業報告

程式運行說明

一、步驟Ⅱ:程式運行說明

- Task1 主要由 Task1.py 檔案處理，執行步驟如下:
  1. python Task1.py --dataset A --min_support 0.002 --result 1
     生成 min_support 0.02, dataset A 的頻繁資料集(名稱符合要求).txt 到 output_in_step2 這個資料夾中，command line 會輸出執行時間(秒)
  2. python Task1.py --dataset A --min_support 0.002 --result 2
     生成 min_support 0.02, dataset A 的統計資料(名稱符合要求).txt 到 output_in_step2 這個資料夾中，command line 會輸出執行時間(秒)
- Task2 主要由 Task2.py 檔案處理，執行步驟如下:
  1. python Task2.py --dataset A --min_support 0.002
     生成 min_support 0.02, dataset A 的頻繁資料集(名稱符合要求).txt 到 output_in_step2 這個資料夾中，command line 會輸出執行時間(秒)

二、步驟Ⅲ：程式運行說明

- 步驟Ⅲ主要由 step3.py 檔案處理，執行步驟如下:
  1. python step3.py --dataset A --min_support 0.002
     生成 min_support 0.02, dataset A 的頻繁資料集(名稱符合要求).txt 到 output_in_step3 這個資料夾中，command line 會輸出執行時間(秒)

算法/程式報告

一、步驟Ⅱ:

- 針對 Task1-(a)的程式修改(函數在:資料處理.py, apriori.py):
  1. 這個任務基本上就是使用原本的算法，所以只有針對資料集輸入形式做修改，讓原本的.data 檔轉成.csv 檔，並且剔除前一二行(因為是 index)，其餘皆沒有修改，以下是修改的程式碼

```
def process_dataset(dataset_name):
    # 資料檔案路徑
    file_path = fr'C:\Users\User\data_mining\datasets_hw1\{dataset_name}\{dataset_name}.data'
    formatted_contents = []
    with open(file_path, 'r') as f:
        for line in f:
            fields = line.strip().split(",")  # Split by comma and remove newline characters
            formatted_contents.append("  ".join(fields))

    # 轉換成 CSV 格式
    csv_file_path = fr'C:\Users\User\data_mining\datasets_hw1\{dataset_name}\{dataset_name}.csv'
    with open(csv_file_path, 'w', newline='') as csvfile:
        for line in formatted_contents:
            csvfile.write(line.replace(' ', ',') + '\r')
```

```
def dataFromFile_for_task1(fname):
    """Function which reads from the file and yields a generator"""
    with open(fname, "rU") as file_iter:
        for line in file_iter:
            line = line.strip().rstrip(",")  # Remove trailing comma
            line = ",".join(line.split(",")[2:])
            record = frozenset(line.split(","))
            yield record
```

  2. 發現資料規模越大，因為每次都要生成大量候選集，apriori 算法越來越慢，因為每次都要生成大量候選集。

- 針對 Task1-(b)的程式修改(函數在:apriori.py):
  1. 因為要輸出每個 iteration 的頻繁集增減跟輸出最終頻繁集的數量，所以在算法裡面加入了計算 iteration，並且將 before pruning 跟 after pruning 的數據寫到.txt 裡，以下是針對上述的程式修改:

```python
def runApriori_for_task1_2(data_iter, minSupport, minConfidence, output_filename):
    """
    run the apriori algorithm. data_iter is a record iterator
    Return both:
     - items (tuple, support)
     - rules ((pretuple, posttuple), confidence)
    """
    itemSet, transactionList = getItemSetTransactionList(data_iter)

    freqSet = defaultdict(int)
    largeSet = dict()
    # Global dictionary which stores (key=n-itemSets,value=support)
    # which satisfy minSupport

    assocRules = dict()
    # Dictionary which stores Association Rules

    oneCSet = returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet)

    currentLSet = oneCSet
    k = 2
    iteration = 0 # for task1_2
    with open(output_filename, 'w') as output_file:        You, 5 小時前 • Uncommitted ch
        while currentLSet != set([]):
            largeSet[k - 1] = currentLSet
            currentLSet = joinSet(currentLSet, k)
            # 剪枝前的候選集數量
```

```python
            num_candidates_before_pruning = len(currentLSet)


            currentCSet = returnItemsWithMinSupport(
                currentLSet, transactionList, minSupport, freqSet
            )

            # 剪枝後的候選集數量
            num_candidates_after_pruning = len(currentCSet)

            currentLSet = currentCSet
            k = k + 1
            iteration += 1

            # 印出每一次的候選集數量
            #print(f'Iteration {iteration}:')
            #print(f'Before pruning: {num_candidates_before_pruning} candidates')
            #print(f'After pruning: {num_candidates_after_pruning} candidates')



            # 寫入檔案
            output_file.write(f'{iteration}\t{num_candidates_before_pruning}\t{num_candidates_after_pruning}\n')
        output_file.close()
```

```
    # 將frequent itemsets數量寫入檔案
    with open(output_filename, 'a') as output_file:
        output_file.write(f'[{total_frequent_itemsets}]\n')
        output_file.close()
    # 將最後一行的候選集數量換到第一行
    with open(output_filename, 'r') as file:
        data = file.readlines()
    last_line = data[-1]
    data.pop(-1)
    data.insert(0, last_line)
    with open(output_filename, 'w') as file:
        file.writelines(data)
    #print(f'Total frequent itemsets: {total_frequent_itemsets}')

    toRetRules = []
    for key, value in list(largeSet.items())[1:]:
        for item in value:
            _subsets = map(frozenset, [x for x in subsets(item)])
            for element in _subsets:
                remain = item.difference(element)
                if len(remain) > 0:
                    confidence = getSupport(item) / getSupport(element)
                    if confidence >= minConfidence:
                        toRetRules.append(((tuple(element), tuple(remain)), confidence))
    return toRetItems, toRetRules
```

- 計算時間結果:

Dataset/minsup	0.01	0.005	0.002
Dataset A	2.43sec	7.53sec	149.28sec
Dataset/minsup	0.005	0.002	0.0015
Dataset B	708.73sec	3427.9sec	5621.77sec
Dataset/minsup	0.03	0.03	0.01
Dataset C	619.26sec	1514.87sec	3783.69sec

- 針對 Task2 的程式修改(函數在:apriori.py):
  1. 因為要尋找出 Frequent Closed Itemset，我的作法是利用 apriori 生成的 frequent itemsets 並且遍歷 frequent itemsets，找出是否在與自己項集相同的情況下，且超集的 support 少於等於原本項集的 support，加到 Frequent Closed Itemset list，算法實現如下:

```python
# 是否為closed itemsets
def is_closed(itemset, freqSet):
    item_support = freqSet[itemset]
    for frequent_itemset in freqSet:
        if itemset != frequent_itemset and itemset.issubset(frequent_itemset):
            if item_support <= freqSet[frequent_itemset]:
                return False
    return True

# 挖掘frequent closed itemsets
def runApriori_for_frequent_closed_itemsets(data_iter, minSupport, minConfidence):
    items, rules, freqSet = runApriori(data_iter, minSupport, minConfidence)
    freqSet = list_to_dict(items)
    frequent_closed_itemsets = []
    result_list = []
    start_time = time.time()
    for itemset in items:
        itemset = frozenset(itemset[0])
        if is_closed(itemset, freqSet):
            frequent_closed_itemsets.append((tuple(itemset), freqSet[itemset]))
    elapsed_time = time.time() - start_time
    return frequent_closed_itemsets, elapsed_time      You, 1 秒前 • Uncommitted ch
def list_to_dict(items):
    freqSet = defaultdict(float)
    for itemset, support in items:
        freqSet[frozenset(itemset)] = float(support)
    return freqSet
```

- Task1 與 Task2 時間比較 ratio 如下：

Dataset/minsup	0.01	0.005	0.002
Dataset A	100.5818%	104.7215%	145.0464%
Dataset/minsup	0.005	0.002	0.0015
Dataset B	100.0078%	100.0657%	100.1223%
Dataset/minsup	0.03	0.03	0.01
Dataset C	100.0012%	100.000132%	100.0000265%

二、步驟Ⅲ：

- 算法描述
  1. 主要採用 FP-Growth 算法，為非 Candidate-based，算法參考來源如下：
     https://github.com/evandempsey/fp-growth
- Program flow
  1. 建立 FP-樹：
     - 遍歷每個交易，對 FP-樹進行構建。
     - 對於每個項目，根據其頻率將其添加到 FP-樹的適當分支或節點。
     - 對 FP-樹中的節點進行連接，以建立樹的結構
  2. 建立頻繁項目集：

- 遍歷 FP-樹，找到滿足最小支援度（min_support）閾值的項目集。
- 每次找到一個頻繁項目集，都將其添加到頻繁項目集列表中。

3. 挖掘條件基：
   - 對於每個頻繁項目集，創建一個條件基數據集，其中僅包含該項目集的交易。
   - 對條件基進行遞迴挖掘，以找到更多的頻繁項目集。
4. 迭代：
   - 重複執行步驟 2 到步驟 3，直到不再找到新的頻繁項目集。

- Differences/Improvements in this algorithm

1. 候選項目集生成：
   FP-Growth 算法不需要生成候選項目集。它通過建立 FP-樹（Frequent Pattern Tree）來表示事務數據集，這個樹的結構包含了頻繁項目集的信息，並且不需要額外的候選項目集生成過程。
2. 計算效率：
   FP-Growth 通過 FP-樹結構實現了高效的頻繁項目集挖掘，避免了生成候選項目集和多次數據集掃描，因此在計算效率上通常更快。
3. 空間消耗：
   FP-Growth 的空間消耗較低，因為它只需要構建 FP-樹，而不需要存儲大量候選項目集。這使得它在內存有限的情況下更實用。
4. 以下是我的修改部分(算法實現在 pyfpgrowth.py, 下圖只是調用:)，基本上遵循 FP-Growth 算法流程(上述 program flow)，並且讓資料符合算法的輸入和輸出:

```python
def convert_to_list_of_lists(itemsets):
    result = []
    for itemset in itemsets:
        result.append(list(itemset))
    return result

def runFPGrowth(data_iter, minSupport):
    """
    run the FP-Growth algorithm. data_iter is a record iterator
    Return:
     - items (tuple, support)
    """
    itemSet, transactionList = getItemSetTransactionList(data_iter)

    itemSet = convert_to_list_of_lists(itemSet)
    start_time = time.time()
    patterns = find_frequent_patterns(transactionList, minSupport*len(transactionList))
    elapsed_time = time.time() - start_time
    toRetItems = []
    for itemset, support in patterns.items():
        toRetItems.append((tuple(itemset), support/len(transactionList)))
    # support由大到小排序
    toRetItems.sort(key=lambda x: x[1], reverse=True)
    return toRetItems, elapsed_time
```

- Computation time
  1. the percentage of speedup:

Dataset/minsup	0.01	0.005	0.002
Dataset A	98.27%	99.06%	99.8%
Dataset/minsup	0.005	0.002	0.0015
Dataset B	77.35%	95.22%	96.88%
Dataset/minsup	0.03	0.03	0.01
Dataset C	-1044.82%	-747.3%	-286.73%%

  2. computation time

Dataset/minsup	0.01	0.005	0.002
Dataset A	0.042sec	0.071sec	0.305sec
Dataset/minsup	0.005	0.002	0.0015
Dataset B	160.56sec	163.88sec	175.32sec
Dataset/minsup	0.03	0.03	0.01
Dataset C	7089.37sec	12835.47sec	14632.59sec

- Discuss
  1. 我發現雖然 FP-Growth 演算法相比於 Apriori 演算法在資料集 A 跟 B 大幅提高了速度(有些接近 99%)，但在資料集更大的資料集 C 中反而表現不佳，原因分析如下:可能在大數據構建 FP-Tree 可能反而需要更多的內存跟時間，因為 FP-Tree 的大小跟深度可能會增加。且 FP-Growth 需要遞迴的訪問 FP-Tree 來生成頻繁集，有可能在大數據集上

變得較慢，未來可以往內存優化跟多進程處理研究。