Apriori + Hash Tree 程式碼簡介

```
file = 'data\\BreadBasket_list.csv'
file_freq = 'data\\Bread_freq_HashTree.csv'
file_rule = 'data\\Bread_rule_HashTree.csv'

with open(file,newline='') as file:
        contents = csv.reader(file)
        transactions  = [row for row in contents if row]

...
```

↑ 載入BreadBasket的csv檔（已將原BreadBasket_DMS檔轉換成每筆transaction的型態）

```
def get_L1(_transactions, min_support):
        candidate_items = {} #stores support for each item
        for transaction in _transactions:
                for item in transaction:
                        candidate_items[item] = candidate_items.get(item,0) + 1
        L1_support = []
        L1_list = []
        for item in candidate_items:
                if candidate_items[item] >= min_support:
                        L1_support.append(([item],candidate_items[item]))
                        L1_list.append([item])
        return L1_support, L1_list
```

↑ 給入 transactions dataset 及 minimum support，函式中建立candidate_items字典，並儲存交易中每筆item對應的support，並篩選 support 值大於 minium support 的 item。

```
def get_k_subsets(datasets, length):
        subsets = []
        for itemset in datasets:
                subsets.extend([sorted(itemset) for itemset in list(itertools.combi
        subsets = [val for val in subsets]
        return subsets
```

↑ 給定transaction dataset及itemset長度，以便後續將transactions dataset中取出長度為k的所有組合並插入建立好的hash tree中計算各pattern的support值

```
def ck_generator(Lk,k):
        ck_set = []
        #join Ck=(Lk-1 self join Lk-1)
        lenlk = len(Lk)
        for i in range(lenlk):
                for j in range(i+1,lenlk):
                        L1 = list(Lk[i])[:k - 2] #該Lk倒數第一前的元素
                        L2 = list(Lk[j])[:k - 2]
                        if L1 == L2:
                                ck_set.append(sorted(list(set(Lk[i]).union(set(Lk[j]
        return ck_set
```

↑ ck_generator函數用來將Lk產生所有Ck+1候選集，**其中針對所有Lk，利用self join的方式，將兩個長度為k的itemset做聯集以產生一組Ck+1候選itemset**。函數output為所有Ck+1的候選集，以利後續建立Hash Tree。

```
class HNode:

    def __init__(self):
        self.children = {}
        self.isLeaf = True
        self.bucket = {}
```

↑ hash tree的建立方式參考github其他作者的公開程式碼。建立hash tree節點的class，每個節點皆能儲存子節點、葉節點狀態、及input之itemset

```
class HTree:

    def __init__(self, max_leaf_cnt, max_child_cnt):
        self.root = HNode()
        self.max_leaf_cnt = max_leaf_cnt
        self.max_child_cnt = max_child_cnt
        self.frequent_itemsets = []
    def recur_insert(self, node, itemset, index, cnt):
    def insert(self, itemset):
    def add_support(self, itemset):
    def dfs(self, node, support_cnt):
    def get_frequent_itemsets(self, support_cnt):
    def hash(self, val):
```

（因程式碼內容太多，function內容省略）

↑ 建立hash tree的class，這個tree的attribute紀錄著其根節點，最多葉節點及子節點數量、和由這個tree產生的frequent itemsets。

hash tree分為三大部分：

1. 以候選集建立樹狀結構
    - **利用insert function以某候選itemset建立樹枝，並呼叫recur_insert透過不斷疊帶的方式依照itemset中為每個item建立相對應的node，疊代的過程將會判斷某node是否已超過其max_leaf的數量而往下增加子節點**
    - **其中也會依照層級將itemset中第k個item利用hash function做hash後將其視為其子節點的key。**
2. 利用transaction dataset中長度為k的所有組合將itemset插入已建立完成的hash tree中
    - **利用 add_support的function將長度為k的itemset以item的順序先後插入hash tree當中，並依照其條件在節點的部分增加該itemset的support**
3. 利用get_frequent_itemsets函數取得該hash tree中有滿足minimum support的itemset集合
    - **判斷該node是否為葉節點，並將葉節點中itemset以字典的方式儲存其itemset的support**

```
def apriori(_transactions,min_support,max_leaf_cnt,max_child_cnt, freq_patterns, fr
        k=2
        L1_dict, L1_list = get_L1(_transactions, min_support)
      ... generate_hash_tree(C2_candidates,max_leaf_cnt,max_child_cnt)
        k_subsets = get_k_subsets(_transactions,k)
        for subset in k_subsets:
                h_tree.add_support(subset)
        L2 = [items[0] for items in h_tree.get_frequent_itemsets(min_support)]
        ...
        while(len(freq_patterns[k-2])>0):
                Ck_candidates = ck_generator(freq_patterns[k-2], k)
                ...
        return
```

（因程式碼內容太多 部分已省略）
↑ apriori這個function主要是統整利用hash tree建立frequent patterns的順序，首先會從input中的transactions dataset建立L1，再由L1利用ck_generator函數建立C2，之後再利用C2建立hash tree後，從transactions找出所有長度為2的itemset後，利用tree中的functions找出L2，然後以此邏輯繼續疊代後即可找出所有frequent itemsets

```python
def rule_generator(freq_itemset, _candidate_sets, init_num, min_confidence, rules_l
        ...
    if init_num == 1 : #第一層
        _candidate_sets = ...
        init_num += 1
        rule_generator(freq_itemset...)
    else :
        pruned_subsets = ...
        if pruned_subsets != []:
            ...
            else:
                init_num += 1
                rule_generator(freq_itemset,...)

```

（部分程式碼已省略）

↑ 針對每一個frequent itemset，先找出其k-1的所有組合，再利用self join的方式及課堂上講解的技巧，取左邊交集右邊聯集的方式，找出關聯組合，並計算每個組合的confidence，若低於minimum值則prune掉，再利用pruned完後的itemset繼續以相同邏輯疊代找出該組itemset所有符合條件的關聯集合。

```python

min_sup = 10
max_leaf_cnt = 20
max_child_cnt = 20
apriori(_transactions...)

with open(file_freq,'w') as file:
        ...
        for freq_pattern, support in freq_dict_sorted.items():
                file_writer.writerow([set(freq_pattern),support])
'''
Rule Generation
'''
min_conf =0.5
rules_list = []
for _freq_pattern in flattened_patterns:
        rule_generator(freq_itemset...)
...
final_rules = sorted(flattened_rules, key = lambda x: x[2], reverse = True)

with open(file_rule,'w') as csv_file:
        ...
print('writing done!')
```

(部分程式碼已省略)

↑ 最後僅分別依照設定條件呼叫apriori函數及rule_generator函數得到frequent itemsets及rules，並將結果分別寫入以file_freq與file_rule為名的檔案