資料探勘Project 1 Report

賴廷瑋 F44054045

# 目錄

內容

[一、 目錄 2](#_Toc55241593)

[二、 程式碼簡述（程式碼部分因為內容太多有省略，詳細可參考python程式檔) 3](#_Toc55241594)

[Apriori Hash Tree 實作 3](#_Toc55241595)

[Apriori FP Growth 實作 8](#_Toc55241596)

[資料集 12](#_Toc55241597)

[結果 12](#_Toc55241598)

[三、 心得 & 研究 12](#_Toc55241599)

[Hash Tree 與 FP growth 方法運算時間 12](#_Toc55241600)

[Minimum Support 與 Minimum Confidence 的決定 12](#_Toc55241601)

[Hash Tree中minimum leaf counts (最小葉節點數量) 參數設定 13](#_Toc55241602)

[Pruning Stage: 13](#_Toc55241603)

[Itemset順序 13](#_Toc55241604)

[最後心得 13](#_Toc55241605)

# 程式碼簡述（程式碼部分因為內容太多有省略，詳細可參考python程式檔)

## Apriori Hash Tree 實作

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.combinations(itemset,length))])

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, freq\_patterns\_support):

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。

def rule\_generator(freq\_itemset, \_candidate\_sets, init\_num, min\_confidence, rules\_list):

...

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。

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

## Apriori FP Growth 實作

def get\_transaction\_dict(transactions):

transactions\_dict = {}

for transaction in transactions:

transactions\_dict[frozenset(transaction)] = transactions\_dict.get(frozenset(transaction),0) + 1

return transactions\_dict

transactions\_dict = get\_transaction\_dict(transactions)

建立字典，將每筆itemset的frosenset作為key, value為該itemset在資料集出現的次數

class treenode:

def \_\_init\_\_(self, nodeName, support, parentNode):

self.name = nodeName

self.count = support

self.nodelink = None

self.parent = parentNode

self.children = {}

def add\_support(self, support):

self.count += support

定義fp tree中結點的treenode物件（class），每個node都有自己的名稱、support、鏈結點、父結點、子結點，其中add support function可對該物件support作更新。

def get\_fp\_tree(\_transactions\_dict, min\_support): #\_transactions\_dict : {'frozenset()':int}

\_headerTable = {}

for transaction in \_transactions\_dict.keys():

. . .

root\_node = treenode('Root',1,None)

for key in list(\_headerTable): # keys returns an iterator instead of a list.

. . .

freq\_items\_dict\_ = \_headerTable.copy()

freq\_items = set(\_headerTable.keys()) #frequent L1

這個步驟為建立FP Tree，適用於第一次資料庫掃描後的fp tree與後續條件基(Conditional pattern base)的建立。函數給入儲存itemsets的dictionary以及minimum support。而函數中會建立該itemsets的header table，並移除不符合Min support的item

if len(freq\_items) != 0:

for key in \_headerTable.keys():

\_headerTable[key] = [\_headerTable[key],None]

for transaction, count in \_transactions\_dict.items(): #每筆trans 及 其出現的次數(之後trans會是條件基 #

freq\_item\_dict = {}

for item in transaction:

if item in freq\_items:

freq\_item\_dict[item] = \_headerTable[item][0] #針對每筆trans做排序，建立為該trans的dict以儲存該item的suppor #

if len(freq\_item\_dict) > 0:

sorted\_transaction = [x[0] for x in sorted(freq\_item\_dict.items(),key = lambda x : (x[1],x[0]),reverse=True)]

# 將 sorted\_transaction 加入 fp\_tree 中

insert\_to\_tree(sorted\_transaction, root\_node, \_headerTable, count)

return root\_node, \_headerTable, freq\_items\_dict\_

這個步驟延續get\_fp\_tree函數內容。首先將headertable額外儲存每個item在FP tree的相同item鍵結。後續將itemsets中每筆itemset利用疊代的方式插入至該fp tree中，過程中將會更新每個item的support以及其鏈結點，整體建立該itemsets的FP tree。

def insert\_to\_tree(\_sorted\_transaction, \_root\_node, Header\_table, count):

if len(\_sorted\_transaction) == 0:

return

. . .

\_root\_node.children[\_sorted\_transaction[0]], Header\_table, count)

def update\_header(pointed\_node, linked\_node):

while(pointed\_node.nodelink != None):

pointed\_node = pointed\_node.nodelink # 指向node之連結node#

pointed\_node.nodelink = linked\_node #指向node之連結node和目標node連結形成連結鍊#

此步驟僅是建立fp tree過程中會利用的function，其中Insert\_to\_tree將給入itemset依照該itemset中的item依順序及規定的插入tree中，而update\_header為更新header table中每個item儲存在tree中的鏈結點。

def get\_single\_prefix\_path(single\_node, prefix\_path): #從leaf node向上找出單條條件基, 將其 prefix\_path(list) 更新為條件基#

if single\_node.parent != None:

prefix\_path.append(single\_node.name)

get\_single\_prefix\_path(single\_node.parent, prefix\_path)

def find\_all\_prefixPath(item\_node): #針對所有該 K1 item 鏈結之 nodes 各向上找 single prefix path，output為dict(key為條件基、value為條件基support)#

conditional\_patterns\_dict = {}

while item\_node != None:

prefixPath = []

get\_single\_prefix\_path(item\_node, prefixPath)

if len(prefixPath) > 1: #有條件基(不包含自己)

conditional\_patterns\_dict[frozenset(prefixPath[1:])] = item\_node.count

item\_node = item\_node.nodelink # 更新為下一個鏈結點

return conditional\_patterns\_dict

這段函數是針對transaction中的各個item在fp tree中的所有節點(包含連結點)找出該item的所有條件基（conditional pattern path)。

def mineTree(TreeClass, headerTable\_dict, min\_support, prefixPath, freq\_item\_list, freq\_pattern\_dict):

item\_sorted = [x[0] for x in sorted(headerTable\_dict.items(),key= lambda x:x[1][0],reverse = False)] #將item依照其support降冪排序#

for item in item\_sorted :

. . .

condTree, condHeader, \_ = get\_fp\_tree(condPatternBases\_dict,min\_support) #利用該item的所有條件基建立fp tree

if condHeader != None:

mineTree(condTree, condHeader, min\_support, new\_freq\_set,freq\_item\_list,freq\_pattern\_dict)

return freq\_item\_list

此段程式碼主要功能是利用 header table 針對各個找出 fp tree 中的所有條件基，而對於每個item，利用疊代的方式利用條件基中的 conditional header table 建立條件基樹(conditional fp tree)，以此方式找出所有符合minimum support 的 frequent patterns 與 其 support值

def rule\_generator(freq\_itemset, \_candidate\_sets, init\_num, min\_confidence, rules\_list):

if len(freq\_itemset) == 1 :

return

. . . candidate\_sets.append(pruned\_subsets[i][0].intersection(pruned\_subsets[j][0]))

candidate\_sets = [frozenset(candidate) for candidate in candidate\_sets]

candidate\_sets = list(dict.fromkeys(candidate\_sets))

. . .

這段為rule generation的階段，和先前hash tree的方法相同，不再贅述

找出rules後，關於後續的程式碼也和Hash tree的部分大致相同，主要只是寫入檔案和整理list而已。

## 資料集

IBM\_transaction.csv : 共 6956 筆交易

BreadBasket\_list.csv：此資料集為kaggle之資料集，已自己整理成transaction的list形式，共9465筆交易

## 結果

程式結果皆存為csv檔並附上。BreadBasket\_freq.csv和 BreadBasket\_rules.csv 分別為BreadBasket資料集利用min\_support = 10 & min\_conf = 0.5 的frequent patterns 與 rules；而IBM\_freq 與 IBM\_rules 分別為 IBM 資料集利用min\_support = 10 & min\_conf = 0.6 的frequent patterns 與 rules。

# 心得 & 研究

## Hash Tree 與 FP growth 方法運算時間

兩種方法在運算的時間上有明顯得差異。針對IBM 這筆資料集，在設定min\_support為10且min\_confidence 為0.6 時，Hash Tree的運算時間大約為8秒；而利用 FP growth 的方法僅需0.8秒左右，可以發現FP growth演算法的運算時間明顯快許多（近10倍），原因是 FP growth 的算法中僅需掃描資料集兩次，而在hash tree 算法中在Ck候選集生成Lk時，須每次從資料集取出可能的組合，再加上從Lk 找出 Ck+1 時皆須運用較多的計算。

## Minimum Support 與 Minimum Confidence 的決定

Minimum Support: 實作的過程中，min support 若設定較小，則大部分的pattern皆會是frequent，結果中可能會出現很多雜訊，而且運算時間明顯會多許多；反之min support 若設定太大，雖然計算較快，但許多pattern皆會是infrequent，如此可能會遺失資料重要資訊。

Minimum Confidence：最小信心值的設定取決於運算的大小與個人期望的rule信心水準。愈高的Min confidence會讓計算時間減少（幅度較少），且rules結果有更高的信心水準；但針對BreadBasket資料集遇到的情況，會發現若Minimum Confidence若設太高，則大部分的規則會是imply {Coffee}，主要原因是有coffee的frequency非常高，所以最後將min\_confidence調成0.5，可以發現除了Imply 咖啡以外的rules。

## Hash Tree中minimum leaf counts (最小葉節點數量) 參數設定

此參數為hash tree中每個節點最多能存放的itemset數量。針對IBM資料集，在設定min\_leaf\_count 為 20情況下hash tree的mining總共為7.64秒，當參數調為5時，時間變為7.74秒，當使參數升為100時，可發現整體時間快了近0.3秒，雖然不是很大的差異，但可以發現透過調整此參數有利於整體演算法速度的進步。

## Pruning Stage:

在建構rule時，中間的過程原本是利用暴力的方式找出所有關係再過濾掉不符合min conf的rule，但再利用疊代的方式針對每筆交易的rules做pruning後，有發現rule generation那個步驟的計算有變快許多，可知pruning在關聯演算法中的重要性。

## Itemset順序

在建立hash tree和fp tree時因為在將transaction插入至tree時沒有將其排序，所以一直找不到正確的frequent pattern和rules，而再進一步了解後，才發現transaction在tree的插入階段非常重要，因為相同的itemset若沒有排序可能會造成後續計算support的問題，最後終於成功debug確定兩種tree皆正常運作。

## 最後心得

這次作業花了非常大量的時間撰寫（應該超過40個小時…），不過學到非常多關於關聯法則的概念，而且將演算法以很底層的方式寫出來，感覺很紮實。會了關聯法則後對於後續資料的分析或是前處理有很大的幫助，能對資料有更深的洞見，這次作業雖然較難，但真的也學到很多。比較可惜的是IBM的item只是數字，對於rules 比較沒辦法有具體的推論。再次謝謝助教，辛苦了！