



FP-growth演算法

組員：資工2A 00857034 陳冠宇
資工2A 00857028 張博堯
資工2A 00857014 徐易中

如何尋找頻繁項:

1. 窮舉法: 為了計算每種組合的支持讀, 需遍歷 $2^n - 1$ 次

2. Apriori 演算法:

原理: 如果某個項集是頻繁的, 那麼它所有的子集也是頻繁的。如果一個元素項是不頻繁的那麼包含該元素的超集也是不頻繁的。Apriori演算法從單元素項集合開始, 通過組合滿足最小支援度要求的項集來形成更大的集合。

3. FP-growth 演算法:

原理: 精隨是**建構一個FP-tree**, 他對於資料的挖掘並不是針對全量資料, **只針對FP-tree上的頻繁集**

Apriori 與 FP-growth演算法比較:

FP-growth演算法更快, 只需對DataSet掃描兩次直接得到頻繁集, 而Apriori需多次掃描, 每次利用候選頻繁集產生頻繁集, 故效率低

Apriori -----> 優點: 易編碼實現

缺點: 掃描次數過多, 大資料集上會較慢

適用資料類別: 稀疏型態資料庫

FP-growth ----> 優點: 速度較快

缺點: 需以遞迴生成條件數據庫和FP-tree, 內存開銷大

適用資料類別: 密集型態資料庫

mushroom.dat介紹

mushroom.dat資料集是關於肋形蘑菇的23種特徵的資料集，每個特徵都包含一個標稱資料值

目的: 找出毒蘑菇的相似特徵

第一個特徵:表示有毒或者可食用。如果某樣本有毒,則值為2。如果可食用,則值為1。

第二個特徵:蘑菇傘的形狀,有六種可能的值, 分別用整數3-8來表示。

(3:bell鐘型, 4:conical圓錐形, 5:convex凸面, 6:flat平面, 7: knobbed旋鈕 ,8:sunken凹陷)

總共有128種特徵

使用找出頻繁項演算法，尋找特徵為毒蘑菇的最關聯特徵，我們可以看出毒蘑菇最常見的特徵。

1.使用Python資料結構 (list,dict,set等)加速計算

python的各種資料結構各對於存取或是搜尋都有不同運作方式

若是可以運用其各種的特性，在數據量極大的時候，處理時間會有很大的差距

測試

右圖為使用不同資料結構型態時，個別去計算其搜尋的時間

執行結果為：

```
set: 0.01762632617301519
dict: 0.021149536796960248
...
.....
```

至於list那一個 等了不知道多久都沒有出來

一般來說: 速度set>dict>>list

就這樣看來，
假如要處理及大量的數據，還是要避免使用list來進行搜尋

```
import numpy
import time
l=[]
sl=set()
dl=dict()
r=numpy.random.randint(0,10000000,100000)
for i in range(0,100000):
    l.append(r[i])
    sl.add(r[i])
    dl.setdefault(r[i],1)
#生成3種數據結構供查找，常規的list,集合sl,字典dl.
# 裡面的元素都是隨機生成的，
# 為什麼要隨機生成元素？這是防止某些結構對有序數據的偏倚導致測試效果不客觀。

start=time.clock()
for i in range(100000):
    t=i in sl
end=time.clock()
print("set:",end-start)

start=time.clock()
for i in range(100000):
    t=i in dl
end=time.clock()
print("dict:",end-start)

start=time.clock()
for i in range(100000):
    t=i in l
end=time.clock()
print("list:",end-start)
```

序列	list	deque	dict	set
insert	$\sqrt{O(n)}$	\times	\times	\times
append	$\sqrt{O(1)}$	$\sqrt{O(1)}$	\times	\times
appendleft	\times	$\sqrt{O(1)}$	\times	\times
extend	$\sqrt{O(k)}$	$\sqrt{O(1)}$	\times	\times
extendleft	\times	$\sqrt{O(1)}$	\times	\times
add	\times	\times	\times	$\sqrt{O(1)}$
update	\times	\times	$\sqrt{O(1)}$	$\sqrt{O(1)}$
remove	$\sqrt{O(n)}$	$\sqrt{O(n)}$	\times	$\sqrt{O(n)}$
clear	\times	$\sqrt{O(1)}$	$\sqrt{O(1)}$	$\sqrt{O(1)}$
del	$\sqrt{O(n)}$	$\sqrt{O(1)}$	$\sqrt{O(1)}$	$\sqrt{O(1)}$
popleft	\times	$\sqrt{O(1)}$	\times	\times
pop last (pop()[list])	$\sqrt{O(1)}$	$\sqrt{O(1)}$	\times	\times
pop (index[list]/key[dict])	$\sqrt{O(k)}$	$\sqrt{O(1)}$	$\sqrt{O(1)}$	\times
popitem()	\times	\times	$\sqrt{O(1)}$	\times
Iteration(迭代)	$\sqrt{O(n)}$	$\sqrt{O(n)}$	$\sqrt{O(n)}$	$\sqrt{O(n)}$
x in s (查找)	$\sqrt{O(n)}$	$\sqrt{O(n)}$	$\sqrt{O(1)}$	$\sqrt{O(1)}$

在查找方面, dict和set明顯比list有效率許多

2.方便完成此分組作業的Python特性或基本模組

1. 語法簡單
2. 資料型態list, dict, set上轉換方便, 可迭代, 功能多, 易使用
3. 擁有frozenset型態, 不能修改, 添加, 刪除元素特性, 可當dict的key
4. 使用time模組計算時間
5. itertools 的 combinations 函式, 可輕鬆找出排列組合

3.重要程式片段說明

#####建FP樹#####

```
def createFPTree(frozenDataSet, minSupport):
    headPointTable = {}
    for items in frozenDataSet:
        for item in items:
            headPointTable[item] = headPointTable.get(item, 0) + frozenDataSet[items] ##統計次數
    headPointTable = {k:v for k,v in headPointTable.items() if v >= minSupport} ##篩選minSupport
    frequentItems = set(headPointTable.keys())
    if len(frequentItems) == 0: return None, None
```

1. f:1,a:1,c:1,d:1,g:1,i:1,m:1,p:1
2. a:1,b:1,c:1,f:1,l:1,m:1,o:1
3. b:1,f:1,h:1,j:1,o:1
4. b:1,c:1,k:1,s:1,p:1
5. a:1,f:1,c:1,e:1,l:1,p:1,m:1,n:1

Item	count	Item	count
f	4	d	1
c	4	g	1
a	3	i	1
b	3	o	2
m	3	h	1
p	3	j	1
e	1	s	1
l	1	n	1
k	1		

Item	Count
f	4
c	4
a	3
b	3
m	3
p	3

```

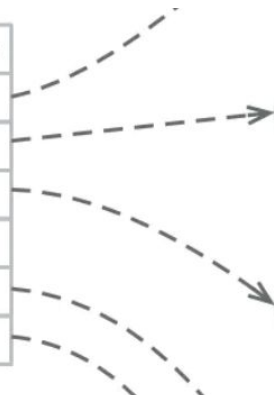
for k in headPointTable:
    headPointTable[k] = [headPointTable[k], None] ##將headPointTable value轉乘list，第一格放次數，第二格放nextSimilarItem
fptree = TreeNode("null", 1, None)

for items,count in frozenDataSet.items():
    frequentItemsInRecord = {}
    for item in items:
        if item in frequentItems:
            frequentItemsInRecord[item] = headPointTable[item][0] ##過濾data
    if len(frequentItemsInRecord) > 0:
        #orderedFrequentItems = [v[0] for v in sorted(frequentItemsInRecord.items(), key=lambda v:v[1], reverse = True)]
        orderedFrequentItems = [v[0] for v in sorted(frequentItemsInRecord.items(), key=lambda v:(v[1], v[0]), reverse = True)] ##排序dataSet
        updateFPTree(fptree, orderedFrequentItems, headPointTable, count)

return fptree, headPointTable


```

Item	Count
f	4
c	4
a	3
b	3
m	3
p	3



上面的for迴圈

Items bought	(Ordered) frequent items
{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
{a, b, c, f, l, m, o}	{f, c, a, b, m}
{b, f, h, j, o}	{f, b}
{b, c, k, s, p}	{c, b, p}
{a, f, c, e, l, p, m, n}	{f, c, a, m, p}



下面的巢狀迴圈

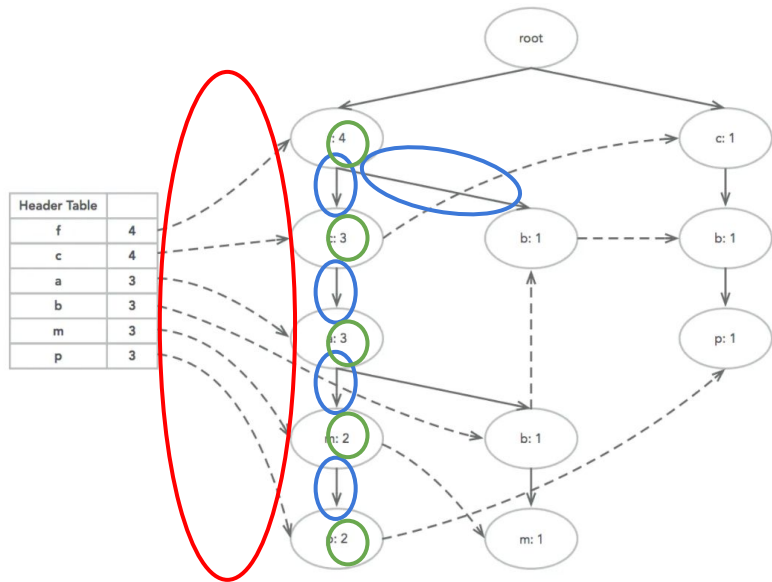
```
def updateFPtree(fptree, orderedFrequentItems, headPointTable, count): ##更新樹
    #handle the first item
    if orderedFrequentItems[0] in fptree.children:
        fptree.children[orderedFrequentItems[0]].increaseC(count) ##count增加
    else:
        fptree.children[orderedFrequentItems[0]] = TreeNode(orderedFrequentItems[0], count, fptree) ##新增新的node
    if headPointTable[orderedFrequentItems[0]][1] == None: ##更新headPointTable
        headPointTable[orderedFrequentItems[0]][1] = fptree.children[orderedFrequentItems[0]] ##指到這個
    else:
        updateHeadPointTable(headPointTable[orderedFrequentItems[0]][1], fptree.children[orderedFrequentItems[0]]) ##找到最後一個，再把這個塞入最後一個

    if(len(orderedFrequentItems) > 1):
        updateFPtree(fptree.children[orderedFrequentItems[0]], orderedFrequentItems[1:], headPointTable, count) ##遞迴，往下一個
```

該數字+1

全部實線

其他虛線



```
def updateHeadPointTable(headPointBeginNode, targetNode): ##更新headPointTable需要的函式
    while(headPointBeginNode.nextSimilarItem != None): ##找到最後一個，再把targetNode塞入最後一個
        headPointBeginNode = headPointBeginNode.nextSimilarItem
    headPointBeginNode.nextSimilarItem = targetNode
```

這些程式只有含建樹而已喔~

接下來是挖掘frequentPatterns

挖掘frequentPatterns

```
def mineFPTree(headPointTable, prefix, frequentPatterns, minSupport, maxPatLen):
    if len(prefix) >= maxPatLen: return ##大於maxPatLen, return
    #for each item in headPointTable, find conditional prefix path, create conditional fptree, then iterate
    headPointItems = [v[0] for v in sorted(headPointTable.items(), key = lambda v:v[1][0])] ##排序, 轉為list
    print(headPointItems)
    if(len(headPointItems) == 0): return

    for headPointItem in headPointItems: ##找所有headPointItems的路徑
        newPrefix = prefix.copy()
        newPrefix.add(headPointItem)
        frequentPatterns[frozenset(newPrefix)] = headPointTable[headPointItem][0] ##key為frozenset型態的newPrefix

    prefixPath = getPrefixPath(headPointTable, headPointItem)
    if(prefixPath != {}):
        conditionalFPTree, conditionalHeadPointTable = createFPTree(prefixPath, minSupport) ##創造新樹
        if conditionalHeadPointTable != None:
            mineFPTree(conditionalHeadPointTable, newPrefix, frequentPatterns, minSupport, maxPatLen)
```

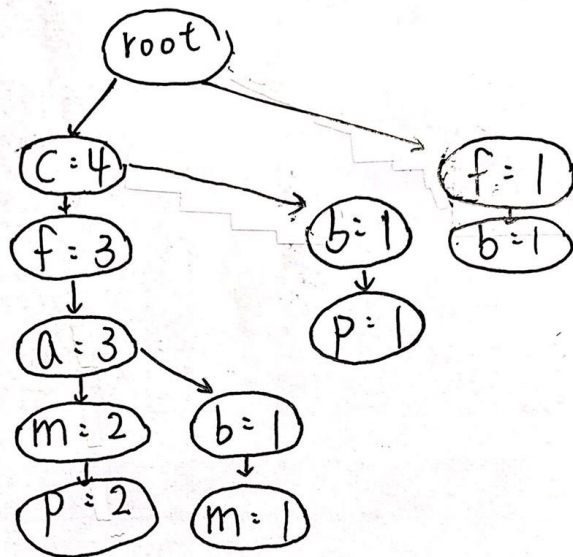
prefixpath

```
def getPrefixPath(headPointTable, headPointItem):  ##找nextSimilarItem 再傳入ascendTree函式找路徑
    prefixPath = {}
    beginNode = headPointTable[headPointItem][1]
    prefixes = ascendTree(beginNode)
    if((prefixs != [])):
        prefixPath[frozenset(prefixs)] = beginNode.count  ##路徑為frozenset型態的key，路徑數量為value

    while(beginNode.nextSimilarItem != None):  ##nextSimilarItem不為空，則往下一個nextSimilarItem走
        beginNode = beginNode.nextSimilarItem
        prefixes = ascendTree(beginNode)
        if (prefixs != []):
            prefixPath[frozenset(prefixs)] = beginNode.count
    return prefixPath

def ascendTree(treeNode):  ##往上找路徑
    prefixes = []
    while((treeNode.nodeParent != None) and (treeNode.nodeParent.nodeName != 'null')):
        treeNode = treeNode.nodeParent  ##往父節點走
        prefixes.append(treeNode.nodeName)
    return prefixes
```

c 4
f 4
b 3
p 3
m 3
a 3



(小 → 大) (先發現 ~ 後發現)
[a, m, p, b, c, f]

```

PS C:\Users\chonyy\Desktop\git\frequent-pattern> python .\fpgrowth.py
Null 1
  c 4
    f 3
      a 3
        m 2
          p 2
            b 1
              m 1
                b 1
                  p 1
                    f 1
                      b 1
Finish sorting: ['a', 'm', 'p', 'b', 'c', 'f']
Checking item: a
in ['a', 'm', 'p', 'b', 'c', 'f']
Adding new frequent set: {'a'}
Conditional tree for item: a
  Null 1
    f 3
      c 3
Finish sorting: ['f', 'c']
Checking item: f
in ['f', 'c']
Adding new frequent set: {'f', 'a'}
Checking item: c
in ['f', 'c']
Adding new frequent set: {'c', 'a'}
Conditional tree for item: c
  Null 1
    f 3
Finish sorting: ['f']
Checking item: f
in ['f']
Adding new frequent set: {'c', 'f', 'a'}
Checking item: m
in ['a', 'm', 'p', 'b', 'c', 'f']
Adding new frequent set: {'m'}
Conditional tree for item: m
  Null 1
    a 3
      f 3
        c 3
Finish sorting: ['a', 'f', 'c']
Checking item: a
  
```

Level 1

Level 2

Level 3

Pattern Growth

associaton_rules

```
#####挖掘關聯規則#####  
  
def associaton_rules(frequentPatterns, minConf):  
    total = 0  
    for frequenceItem in frequentPatterns:  
        subSets = [c for n in range(1, len(frequenceItem)) for c in combinations(frequenceItem, n)] ##從頻繁項集生成所有排列組合子集合  
  
        for subSet in subSets:  
            confidence = float(frequentPatterns[frozenset(frequenceItem)] / frequentPatterns[frozenset(subSet)])  
  
            if (confidence >= minConf):  
                total += 1  
                #yield set(subSet), set(frequenceItem) - set(subSet), confidence ##association rules  
    print("association rules total:", total)
```

註解:frequentPatterns={frozenset(子集合):所對應的出現次數,.....}

主程式

```
if __name__ == '__main__':
    begin = time.time()
    dataSet = loadDataSet_str()
    #print(dataSet)
    print("data num:", len(dataSet))
    minSupport = (len(dataSet)+9)//10
    minConf = 0.8
    maxPatLen = 5
    print("minSupport=", minSupport) ##support限制
    print("minConf=", minConf) ##confidence限制
    print("maxPatLen num:", maxPatLen) ##frequency patterns rules長度限制
    frozenDataSet = transfer2FrozenDataSet(dataSet) ##換成frozenset
    fptree, headPointTable = createFPTree(frozenDataSet, minSupport) ##建樹

    frequentPatterns = {}
    prefix = set([])
    mineFPTree(headPointTable, prefix, frequentPatterns, minSupport, maxPatLen) ##挖掘
    #print("frequent patterns:")
    #fp_print(frequentPatterns)
    count_frequency_item_set(frequentPatterns) ##計算frequency item set
    associaton_rules(frequentPatterns, minConf)
    end = time.time()
    print("execute time:", end-begin)
```


執行結果與計算時間

```
data num: 8124  
minSupport= 813  
minConf= 0.8  
maxPatLen num: 5  
|L^1|=56  
|L^2|=763  
|L^3|=4593  
|L^4|=16150  
|L^5|=38800  
association rules total: 394175  
execute time: 2.485290765762329
```

心得

在寫程式碼的時候，找了些現成程式碼資料，但有些是程式碼是錯的，有些是跑得出結果，但答案是錯的，我便去理解每個人寫的函式的差異，從建樹階段，挖掘階段，到其附屬函式，都去比對過，都找不出錯在哪，還回頭去看讀檔的部分，是否型態有錯。

最後去理性分析，比較可能出錯的部分，最後找到是sorted的問題。再來就是association rules部分，原本函式找排列組合會超時且跑不出結果，之後才想到itertools有combinations自動尋找排列組合，使用後才解決了時間問題。

參考資料

AprioriAlgorithm:<https://medium.com/bandai%E7%9A%84%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92%E7%AD%86%E8%A8%98/%E6%89%8B%E6%8A%8A%E6%89%8B%E7%A8%8B%E5%BC%8F%E5%AF%A6%E4%BD%9C%E5%88%86%E4%BA%AB%E7%B3%BB%E5%88%97-%E5%85%88%E9%A9%97%E6%BC%94%E7%AE%97%E6%B3%95-apriori-algorithm-%E9%97%9C%E8%81%AF%E8%A6%8F%E5%89%87%E5%88%86%E6%9F%90-64e0c8506413>

關聯分析算法:FPGrowth:<https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/733255/>

搜索记录频繁模式挖掘:<https://github.com/CLDXiang/Mining-Frequent-Pattern-from-Search-History>