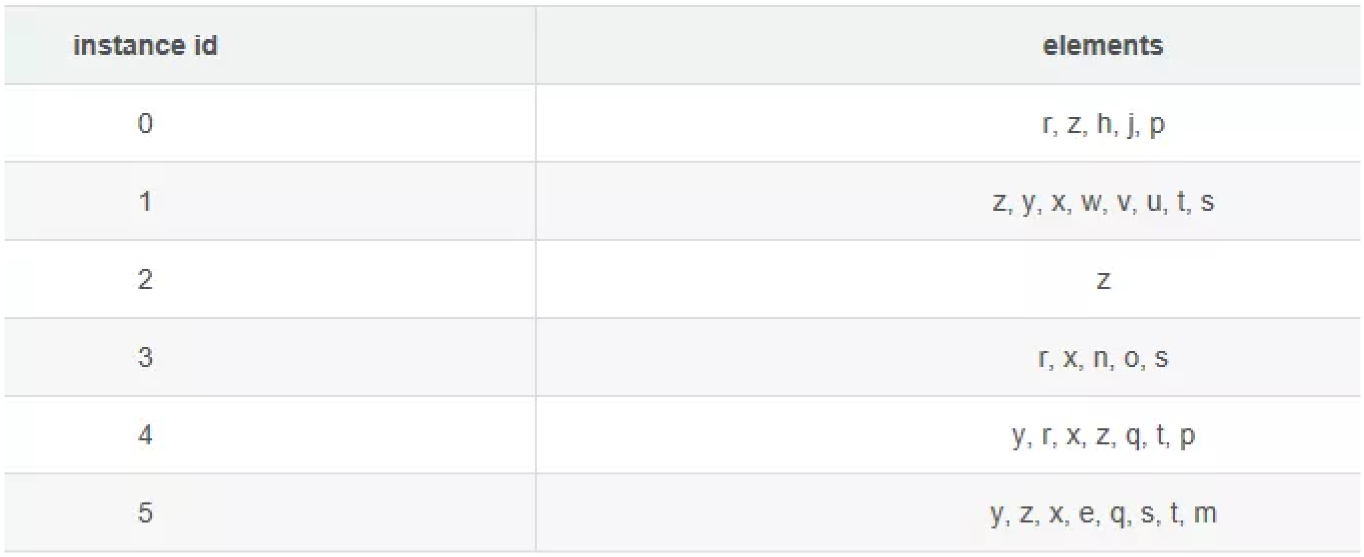
FP-growth算法的python实现

wsp Python可视化编程机器学习OpenCV 2019-10-24

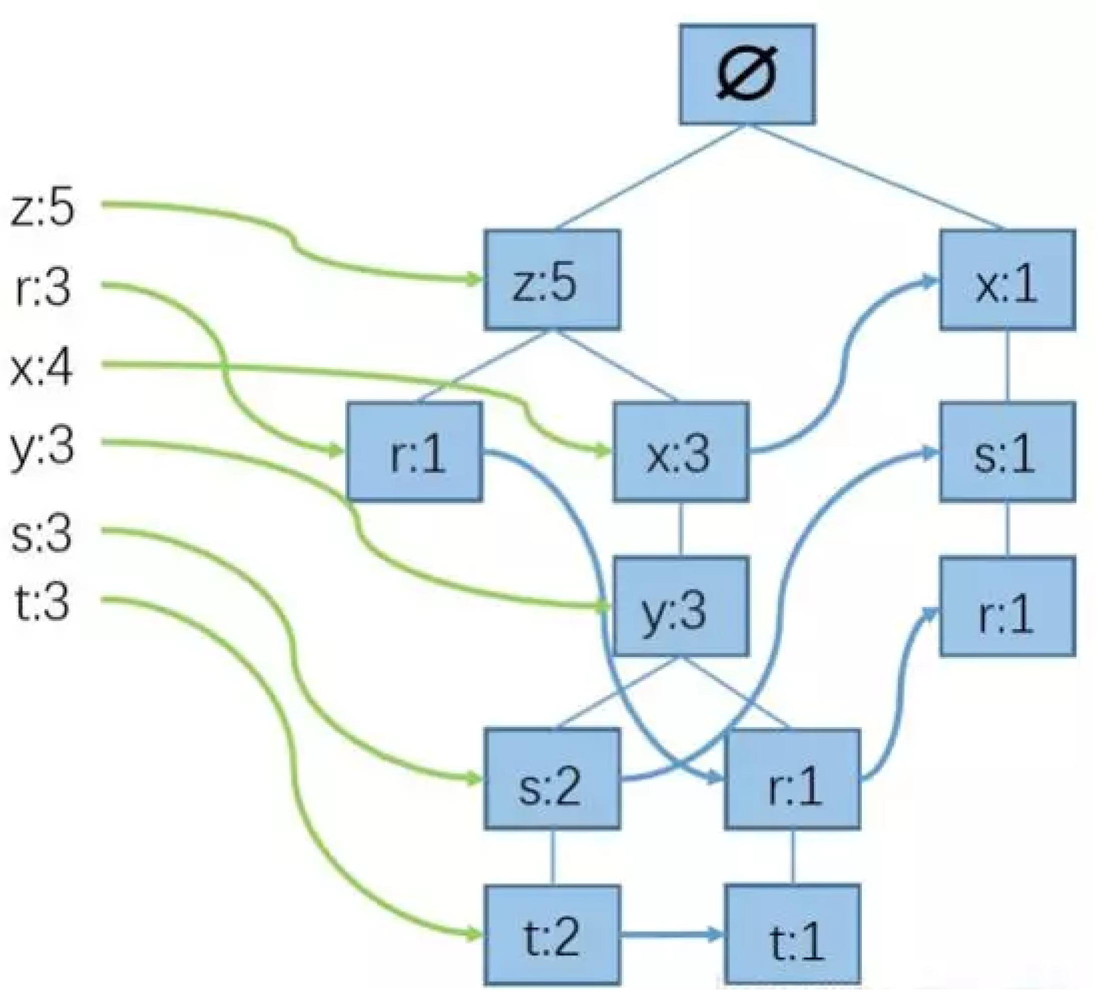
FP-growth算法是一种用于发现数据集中频繁模式的有效方法。Apriori算法在产生频繁模式完全集前需要对数据库进行多次扫描，同时产生大量的候选频繁集，这就使Apriori算法时间和空间复杂度较大。FP-growth算法由Apriori算法产生候选项集，然后扫描数据集来检查它们是否频繁。由于只对数据集扫描两次，因此它比Apriori算法速度要快，通常性能要好两个数量级以上。

在FP-growth算法中，数据集存储在一个称为FP(Frequent Pattern)树的结构中。FP树构建完成后，可以通过查找元素项的条件基以及构建条件FP树来发现频繁集。该过程不断以更多的元素为条件重复进行，知道FP树只包含一个元素为止。

下面仅以这个简单的数据集为例子--实际上，既使在多达百万条记录的大数据集上，FP-growth算法也能快速运行。



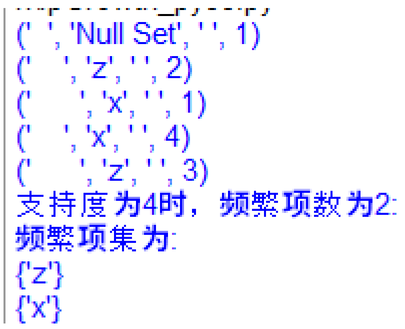
instance id	elements
0	r, z, h, j, p
1	z, y, x, w, v, u, t, s
2	z
3	r, x, n, o, s
4	y, r, x, z, q, t, p
5	y, z, x, e, q, s, t, m



python代码：

```python
'''
FP-Growth FP means frequent pattern
the FP-Growth algorithm needs:
1. FP-tree (class treeNode)
2. header table (use dict)
This finds frequent itemsets similar to apriori but does not
find association rules.
@author: Peter
'''
def loadSimpDat():
    simpDat = [['r', 'z', 'h', 'j', 'p'],
               ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
               ['z','p','x'],
               ['r', 'x', 'n', 'o', 's'],
               ['y', 'r', 'x', 'z', 'q', 't', 'p'],
               ['y', 'z', 'x', 'e', 'q', 's', 't', 'm']]
    return simpDat

```

```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue
        self.count = numOccur
        self.nodeLink = None
        self.parent = parentNode      #needs to be updated
        self.children = {}

    def inc(self, numOccur):
        self.count += numOccur

    def disp(self, ind=1):
        print (('  '*ind, self.name, ' ', self.count))
        for child in self.children.values():
            child.disp(ind+1)
    #def __lt__(self, other):#定义 "<"用于sorted()
        #return self.count < other.count

def createTree(dataSet, minSup=1): #create FP-tree from dataset but don't m
    headerTable = {}
    #go over dataSet twice
    for trans in dataSet:#first pass counts frequency of occurance
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]

    for k in list(headerTable.keys()):  #remove items not meeting minSup
        if headerTable[k] < minSup:
            headerTable.pop(k)
    freqItemSet = set(headerTable.keys())

    #print 'freqItemSet: ',freqItemSet
    if len(freqItemSet) == 0: return None, None  #if no items meet min suppo
    for k in headerTable:
        headerTable[k] = [headerTable[k], None] #reformat headerTable to use
    #print 'headerTable: ',headerTable
    retTree = treeNode('Null Set', 1, None) #create tree
    for tranSet, count in dataSet.items():  #go through dataset 2nd time
        localD = {}
        for item in tranSet:  #put transaction items in order
            if item in freqItemSet:
                    localD[item] = headerTable[item][0]
        if len(localD) > 0:
            orderedItems = [v[0] for v in sorted(localD.items(), key=lambda
            updateTree(orderedItems, retTree, headerTable, count)#populate t
    return retTree, headerTable #return tree and header table

def updateTree(items, inTree, headerTable, count):
    if items[0] in inTree.children:#check if orderedItems[0] in retTree.chil
        inTree.children[items[0]].inc(count) #incrament count
    else:   #add items[0] to inTree.children
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        if headerTable[items[0]][1] == None: #update header table
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], inTree.children[items[0]]
    if len(items) > 1:#call updateTree() with remaining ordered items
        updateTree(items[1::], inTree.children[items[0]], headerTable, count

def updateHeader(nodeToTest, targetNode):   #this version does not use recur
    while (nodeToTest.nodeLink != None):    #Do not use recursion to travers
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode

def ascendTree(leafNode, prefixPath): #ascends from leaf node to root
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)

def findPrefixPath(basePat, treeNode): #treeNode comes from header table
    condPats = {}
    while treeNode != None:
        prefixPath = []
        ascendTree(treeNode, prefixPath)
        if len(prefixPath) > 1:
            condPats[frozenset(prefixPath[1:])] = treeNode.count
        treeNode = treeNode.nodeLink
    return condPats

def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    bigL = [k for k,v in sorted(headerTable.items(), key=lambda p: p[1][0])]
    for basePat in bigL:  #start from bottom of header table
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        #print 'finalFrequent Item: ',newFreqSet    #append to set
        freqItemList.append(newFreqSet)
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        #print 'condPattBases :',basePat, condPattBases
        #2. construct cond FP-tree from cond. pattern base
        myCondTree, myHead = createTree(condPattBases, minSup)
        #print 'head from conditional tree: ', myHead
        if myHead != None: #3. mine cond. FP-tree
            #print 'conditional tree for: ',newFreqSet
            #myCondTree.disp(1)
            mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)#

def createInitSet(dataSet):
    retDict = {}
    for trans in dataSet:
        retDict[frozenset(trans)] = 1
    return retDict


minSup = 4
simpDat = loadSimpDat()
initSet = createInitSet(simpDat)
myFPtree, myHeaderTab = createTree(initSet, minSup)
myFreqList = []
if myFPtree is not None:
    myFPtree.disp()
    mineTree(myFPtree, myHeaderTab, minSup, set([]), myFreqList)
print("支持度为%d时，频繁项数为%d:"%(minSup, len(myFreqList)))
print("频繁项集为:")
for item in myFreqList:
    print(item)
```



```
(' ', 'Null Set', ' ', 1)
('   ', 'z', ' ', 2)
('     ', 'x', ' ', 1)
('   ', 'x', ' ', 4)
('     ', 'z', ' ', 3)
支持度为4时，频繁项数为2:
频繁项集为:
{'z'}
{'x'}
```