# 实验四：关联挖掘之FP-growth算法

## 一、目的与要求

FP-growth算法的应用我们经常接触到。比如，你在百度的搜索框内输入某个字或词，搜索引擎会自动补全查询词项，而这些词项都是和搜索词经常一起出现的。FP-growth算法被用来挖掘频繁项集，也就是说从已给的多条数据记录中挖掘出哪些项是频繁一起出现的。该算法适用于标称型数据，即离散型数据。它比Apriori算法更高效，因为该算法只需要对数据库进行两次扫描，而Apriori算法对于每个潜在的频繁项集都会扫描数据集判定给定模式是否频繁。本实验将简单地利用Python实现FP-growth算法，帮助同学们学习。

## 二、操作环境

实验采用头歌实践教学平台（[www.educoder.net](http://www.educoder.net)）进行自动测评，课程团队通过平台提供的公用云和公共镜像的方式提前在平台配置好软硬件环境及实验任务，操作系统为Ubuntu 16.04.3 LTS，并配置Python 3.6编程环境。

## 三、实验内容

### 第1关：判断项集是否满足先验性质

#### 任务描述

本关任务：创建FP树，并且更新FP树，向其中插入新节点。

#### 相关知识

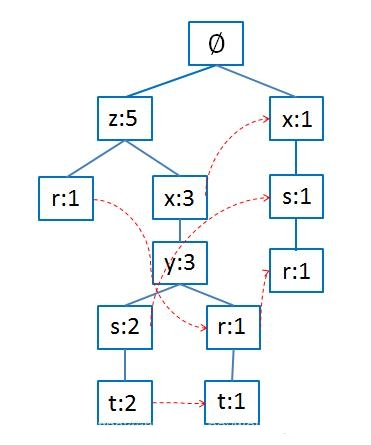
事务ID 事务中的元素项：

001 r,z,h,j,p  
002 z,y,x,w,v,u,t,s  
003 z  
004 r,x,n,o,s  
005 y,r,x,z,q,t,p  
006 y,z,x,e,q,s,t,m

这份数据一共有6条记录，每条记录中的元素就是项，第1条记录中有5个项，分别为：r，z，h，j，p。项的集合就是项集，比如，[r]是一个项集，[r，z]是一个项集，[r，z，h，j，p]也是一个项集，项集是指项的任意组合。而频繁项集是指，那些在记录中经常一起出现的项组合成的集合。那么，“经常”是怎么衡量的呢？这里就涉及到支持度的概念。支持度是说出现的次数，它可以针对单个项，也可以针对项的组合，在这6条数据记录中，r一共出现了3次，所以r的支持度是3，项集（r，x）出现了2次，所以（r，x）的支持度是2。

#### 构建FP树的准备工作

FP代表频繁模式（Frequent Pattern）。我们先看看FP树长什么样子。以下这棵FP树是根据上面那份数据记录建立的。



可以看出，一棵FP树看上去与计算机科学中的其他树结构类似，但是它包含着连接相似节点的链接（图中的红色虚线部分）。 相似节点是指前缀路径不同的项，如在上面的FP树中r的前缀路径有3个，分别为（z），（z，x，y），（x，s），于是，这些不同的r就叫做相似节点。用Python构建FP树时会创建一个字典结构存储这些相似元素。

##### 构造树节点

在构建FP树之前，我们要先定义一个类，用来保存树的每一个节点。

class treeNode:#构造树节点   
 def \_\_init\_\_(self, nameValue, numOccur, parentNode):   
 self.name = nameValue#节点名称   
 self.count = numOccur #节点出现的次数   
 self.nodeLink = None #链接指向的下一个节点   
 self.parent = parentNode#父节点   
 self.children = {}#子节点   
   
 def inc(self, numOccur):#该函数用来增加节点出现的次数   
 self.count += numOccur   
   
 def disp(self, ind=1):#展示节点名称和出现的次数   
 print(' ' \* ind, self.name, ' ', self.count)   
 for child in self.children.values():   
 child.disp(ind + 1)   
运行下面这段代码：   
 rootNode=treeNode('pyramid',9,None) #创建节点   
 rootNode.children['eye']=treeNode('eye',13,None) #增加子节点   
 rootNode.children['phoenix']=treeNode('phoenix',3,None) #增加另一个子节点   
 rootNode.disp() #展示树

运行结果：

文本

中度可信度描述已自动生成

由于“eye”和“phoenix”都是”pyramid“的子节点，所以在展示树的结构时，“eye”和“phoenix”的缩进深度相同，都比”pyramid“的缩进深度更深一级。

##### 把原始事务数据集处理成字典的形式

除此之外，我们还需要把原始事务数据集处理成字典的形式，方面后面的函数调用。  
定义两个函数，如下：

from collections import OrderedDict  
def loadSimpDat():  
 simpDat=[['r','z','h','j','p'],  
['z','y','x','w','v','u','t','s'],['z'], ['r','x','n','o','s'],['y','r','x','z','q','t','p'],['y','z','x','e','q','s','t','m']]   
 return simpDat  
   
def createInitSet(dataSet):   
 retDict=OrderedDict()  
 for trans in dataSet:   
 retDict[frozenset(trans)]=1   
 return retDict

函数loadSimpDat()把多条数据记录存储成列表的形式，函数createInitSet(dataSet)把每条数据记录冻结（frozenset函数）后作为字典的键，而每个键对应的值都是1。

我们运行一下看看结果。

simpDat=loadSimpDat()  
simpDat

图片包含 散点图

描述已自动生成

initSet=createInitSet(simpDat)  
initSet

文本

中度可信度描述已自动生成

##### 第一次遍历数据集

FP树第一次扫描数据库是为了获得每个元素项的出现频率。实现这一步的代码如下（注：代码中的dataSet是经过上面所说的createInitSet（dataSet）函数处理后的数据结果，即一个字典结构）：

headerTable={}#用来存储每项元素及其出现次数  
for trans in dataSet:#遍历每条记录   
 for item in trans:#遍历每条记录的每项元素   
 headerTable[item]=headerTable.get(item,0)+dataSet[trans]#计算每项元素的出现次数  
print headerTable  
print ("headerTable's length: %s" % len(headerTable))

这时headerTable包含17个元素：



接下来，去掉不满足最小支持度的元素项。

for k in headerTable.keys():  
 if headerTable[k]<3:#这里的3是指最小支持度的取值，可根据实际情况改变  
 del(headerTable[k])#如果某项元素的支持度小于最小支持度，从headerTable中删掉该元素  
freqItemSet=set(headerTable.keys())#freqItemSet中的每一项元素的支持度均大于或等于最小支持度  
print headerTable  
print ("headerTable's length: %s" % len(headerTable))

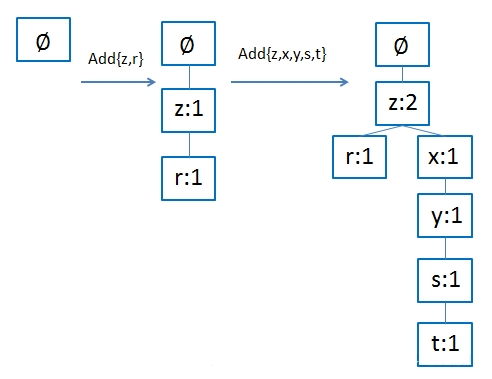
##### 存储符合条件的元素

freqItemSet存储的都是符合条件的元素。如果freqItemSet非空，证明确实有符合条件的元素。从前面的运行结果可以知道，headerTable里面每一个键对应的值是该元素在所有的事务数据记录中出现的次数。现在，我们在每个键对应的值中增加一个“None”，为后面的存储相似元素做准备。  
 if len(freqItemSet)!=0:   
 for k in headerTable:   
 headerTable[k]=[headerTable[k],None]  
 headerTable

#### 构建FP树

构建FP树的思路是这样的：读入每个项集也就是每条记录，并将其添加到一条已经存在的路径中。如果该路径不存在，则创建一条新路径。假设有集合{z,x,y}和{y,z,r}，为了保证相同项只出现一次，需要对每条记录里的元素项进行排序。在每条记录中，这种排序是根据每个元素出现的次数进行的，也就是说出现次数越多，排位越前。

在对事务数据过滤和排序之后，就可以构建FP树了。从空集（符号为Φ）开始，依次遍历过滤，排序后的每一条记录，如果树中已经存在记录中的某个元素，那么已存在的元素增加值即可，如果树中不存在记录中的元素，那么增加一个分支。我们通过添加前两条事务为例，加深对建树的过程理解。



构建FP树的思路是这样的：读入每个项集也就是每条记录，并将其添加到一条已经存在的路径中。如果该路径不存在，则创建一条新路径。假设有集合{z,x,y}和{y,z,r}，为了保证相同项只出现一次，需要对每条记录里的元素项进行排序。在每条记录中，这种排序是根据每个元素出现的次数进行的，也就是说出现次数越多，排位越前。

#### 编程要求

请仔细阅读右侧代码，结合相关知识，在Begin-End区域内进行代码补充，构建FP树。

#### 测试要求

测试程序会使用测试集对你编写的代码进行测试

### 第2关 从FP树中挖掘频繁项集

#### 任务描述

本关任务：从一棵FP树中挖掘频繁项集。

#### 相关知识

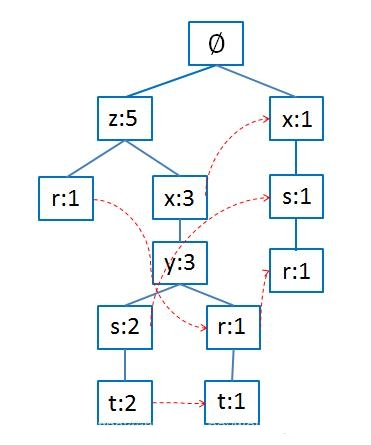
从一棵FP树中挖掘频繁项集的三个基本步骤如下：

1. 从FP树中获得条件模式基；
2. 条件模式基，构建一个条件FP树；
3. 迭代重复步骤1和2，直到树包含一个元素项为止。

##### 条件模式基

条件模式基是相对于树中的某个元素来说的，它指的是该元素在树中的前缀路径，是介于该元素与树根节点之间的所有内容。

比如，在我们构建好的FP树中：



每个元素的前缀路径如下表：

图形用户界面, 表格

描述已自动生成

以下的代码实现查找一棵FP树中某个元素节点及其全部相似元素节点的前缀路径。把前缀路径冻结作为字典的键，前缀路径的计数取值与该元素节点的计数一样。

def ascendTree(leafNode,prefixPath):#该函数找出元素节点leafNode的所有前缀路径，并把包括该leafNode及其前缀路径的各个节点的名称保存在prefixPath中  
 if leafNode.parent!=None:  
 prefixPath.append(leafNode.name)  
 ascendTree(leafNode.parent,prefixPath)  
 def findPrefixPath(basePat,treeNode):  
 condPats={}  
 while treeNode!=None:  
 prefixPath=[]  
 ascendTree(treeNode,prefixPath)  
 if len(prefixPath)>1:  
 condPats[frozenset(prefixPath[1:])]=treeNode.count #某个元素的前缀路径不包括该元素本身  
 treeNode=treeNode.nodeLink #下一个相似元素  
 return condPats #condPats存储的是元素节点treeNode及其所有相似元素节点的前缀路径和它的计数

把FP树中所有元素的前缀路径都找出来：

for item in freqItemSet:  
 condPats=findPrefixPath(item,myHeaderTab[item][1])  
 print item  
 print condPats

结果：

文本

描述已自动生成

这些元素的前缀路径与前面总结的一样，区别是当该元素的前缀路径为空时不显示，当然，还有一点，前缀路径的元素没有按顺序排列，但是不会影响后面的构建条件FP树。  
t的条件FP树如下图：

图示

描述已自动生成

#### 编程要求

请仔细阅读右侧代码，结合相关知识，在Begin-End区域内进行代码补充，实现从FP树中挖掘频繁项集。

#### 测试要求

测试程序会使用测试集对你编写的代码进行测试。