菊安酱的机器学习第9期

菊安酱的直播间：https://live.bilibili.com/14988341

每周一晚8:00 菊安酱和你不见不散哦~(^o^)/~

更新日期：2019-1-14

作者：菊安酱

课件内容说明：

- 本文为作者原创，转载请注明作者和出处
- 如果想获得此课件及录播视频，可扫描左边二维码，回复"k"进群
- 如果想获得2小时完整版视频，可扫描右边二维码或点击如下链接
- 若有任何疑问，请给作者留言。

交流群二维码

完整版视频及课件

直播视频及课件：http://www.peixun.net/view/1278.html

完整版视频及课件：http://edu.cda.cn/course/966

12期完整版课纲

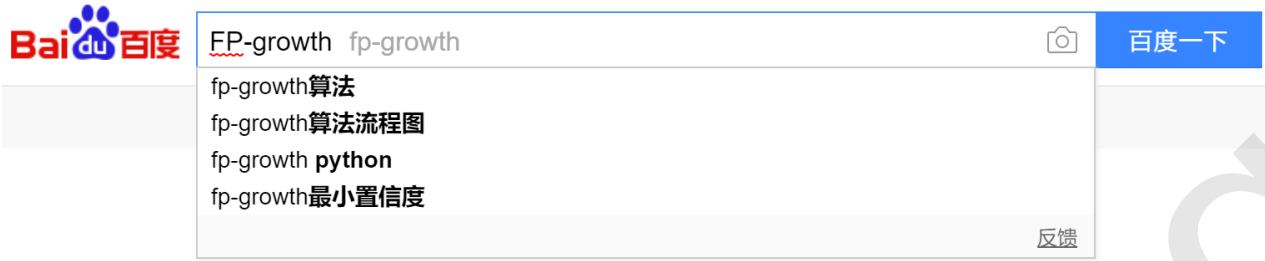
直播时间：每周一晚8:00

直播内容：

时间	期数	算法
2018/11/05	第1期	k-近邻算法
2018/11/12	第2期	决策树
2018/11/19	第3期	朴素贝叶斯
2018/11/26	第4期	Logistic回归
2018/12/03	第5期	支持向量机
2018/12/10	第6期	AdaBoost 算法
2018/12/17	第7期	线性回归
2018/12/24	第8期	树回归
2018/12/28	第9期	K-均值聚类算法
2019/01/07	第10期	Apriori 算法
2019/01/14	第11期	FP-growth 算法
2019/01/21	第12期	奇异值分解SVD

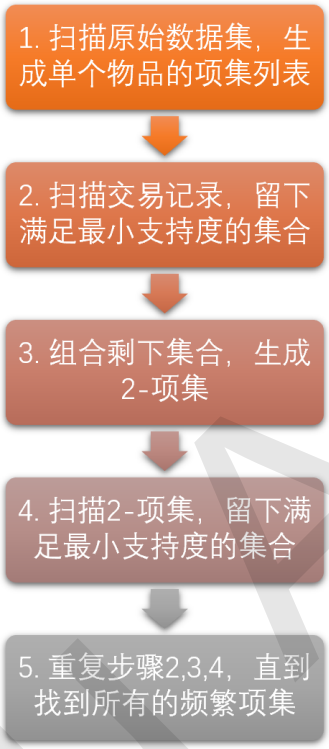
FP-growth算法

相信大家都用过百度搜索，当你输入一个词语的时候，搜索引擎就会帮你自动补全你可能要查询的词项。比如，下图中所示，当我输入"FP-growth"时，百度自动给出了"fp-growth算法"、"fp-growth算法流程图"、"fp-growth python"、"fp-growth最小置信度"这四个词条（你搜出来的结果有可能跟我的不一样）供我们参考。



对于百度的研究人员来说，想要给出这些推荐查询的词项，他们需要查看互联网上的用词并快速地找出经常出现在一起的词对，因此他们需要一种更加高效的发现频繁项集的方法。

上一期讲的Apriori算法是一种很好的发现频繁项集的方法，但是每次增加频繁项集的大小，Apriori算法都会重新扫描整个数据集，当数据量很大的时候，这毫无疑问会成为Apriori算法最大的缺点——频繁项集发现的速度太慢。FP-growth算法其实是在Apriori算法基础上进行了优化得到的算法，下面看一下两者的对比：



图a Apriori算法挖掘频繁项集步骤

图b FP-growth算法挖掘频繁项集步骤

FP-growth算法只需要对数据库进行了两次扫描，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，因此FP-growth算法的速度要比Apriori算法快。在小规模数据集上，这不是什么问题，但是当处理大规模数据集时，就会产生很大的区别。

关于FP-growth算法需要注意的两点是：

（1）该算法采用了与Apriori完全不同的方法来发现频繁项集

（2）该算法虽然能更为高效地发现频繁项集，但不能用于发现关联规则。

从FP-growth算法挖掘频繁项集这个流程图中可以看出，FP-growth算法主要有两个步骤：

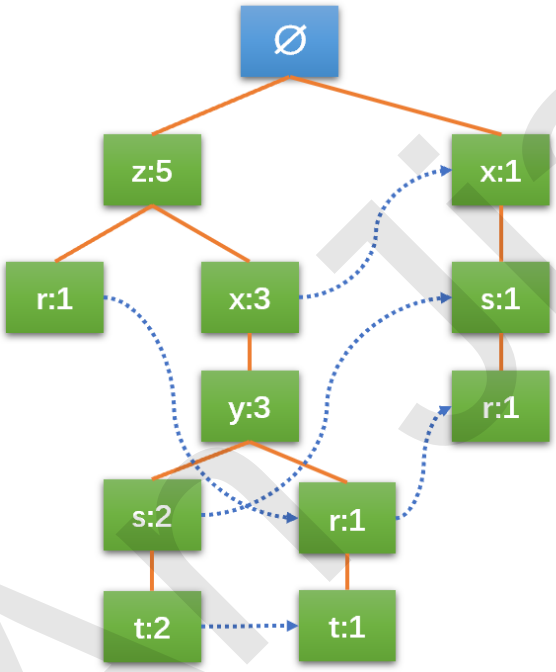
（1）构建FP树

（2）从FP树中挖掘频繁项集

一、FP树

1. FP树表示方式

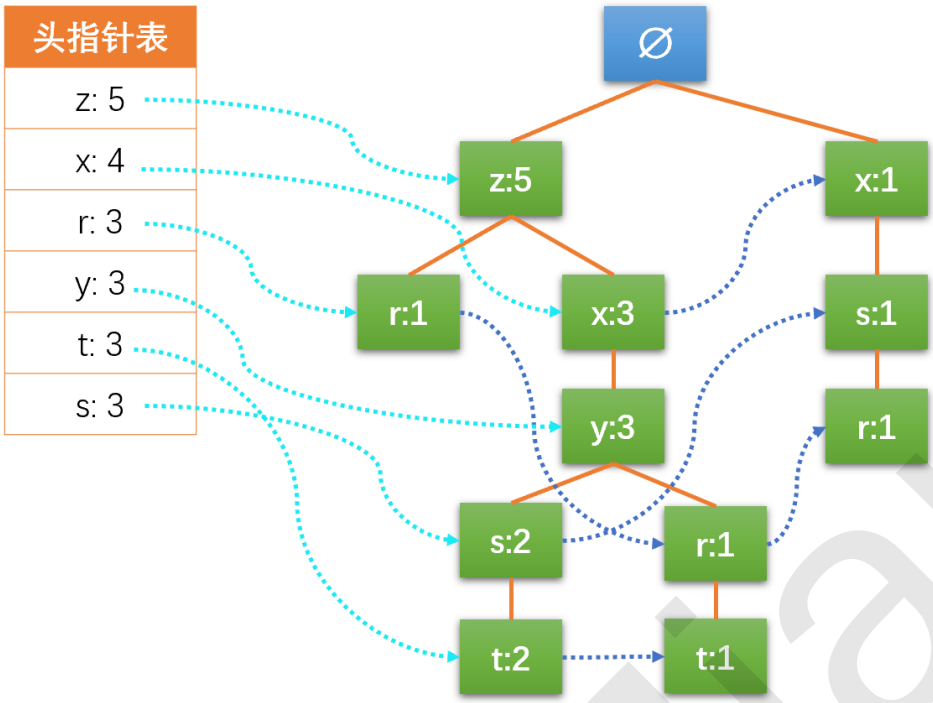
FP是**频繁模式**（Frequent Pattern）的缩写。FP-growth算法将数据存储在一种称为FP树的的紧凑数据结构中，并直接从该结构中提取频繁项集。一棵FP树的结构如下图所示，和其他树结构不同之处是FP树通过链接（link）来连接相似元素，被连起来的元素项可以看成一个链表。可以说，FP树是一种用于编码数据集的有效方式。



图c　一棵FP树

序号	FP树的特点
1	一个元素项可以在一棵FP树中出现**多次**
2	项集以**路径+频率**的方式存储在FP树中
3	树节点中的频率表示的是集合中单个元素在其**序列中出现的次数**
4	一条路径表示的是一个事务，如果事务完全一致，则**路径可以重合**
5	相似项之间的链接即为**节点链接**（link），主要是用来快速发现相似项的位置

为了快速访问树中的相同项，还需要维护一个连接具有相同项的节点的指针列表（headTable），每个列表元素包括：数据项、该项的全局最小支持度、指向FP树中该项链表的表头的指针。



图c　一棵FP树

2. FP树构建过程

为了更好地理解FP树，我们一起来通过一个小例子来构建图c中的这课FP树

下表这个数据集就是用于生成图c中FP树的原始数据集

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

步骤一：统计原始事务集中各元素项出现的频率

这一步主要作用是生成FP树的树节点，统计好的元素项及其频率即为我们所需要的树节点。根据原始事务集，我们可以计算中共有17个元素项，然后分别计算每个元素项在事务集中一共出现了多少次。统计结果如下表所示：

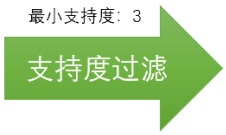
元素项	出现频率
r	3
z	5
h	1
j	1
p	2
y	3
x	4
w	1
v	1
u	1
t	3
s	3
n	1
o	1
q	2
e	1
m	1

步骤二：支持度过滤

上表计算出来的所有树节点并非全部都是我们需要的树节点。我们构建FP树的主要目的是为了挖掘频繁项集，所以为了减少后续的计算量，在这一步我们就可以把不满足最小支持度的元素项给剔除了。因为我们知道，如果一个元素项是不频繁的，那么包含该元素项的超集也是不频繁的，所以就不需要考虑这些超集了。

假设这里我们定义最小支持度为3，那么进行支持度过滤之后得到的元素项为：

元素项	出现频率
r	3
z	5
h	1
j	1
p	2
y	3
x	4
w	1
v	1
u	1
t	3
s	3
n	1
o	1
q	2
e	1
m	1



元素项	出现频率
r	3
z	5
y	3
x	4
t	3
s	3

从上表可以看到，经过支持度过滤之后，元素项由原来的17个变为了6个，可以大大减少后续的计算量。

步骤三：排序

根据频率大小，将支持度过滤后的元素项（即频繁元素项）进行排序，并根据排序后的元素项，将原始事务集也进行排序。这样做的原因是，我们构建的FP树中，相同项只会出现一次，但是大家都知道集合是无序的，对于集合{x, y, z}和集合{z, y, x}我们可以一眼看出这两个集合是一样的，但是对于FP树来说，它会把这两个集合当做两个不同的集合，然后生成两条路径，这样的结果不是我们想要的。所以为了解决这样的问题，我们需要将集合添加到树之前对它们进行排序。

频繁元素项排序结果：

元素项	出现频率
z	5
x	4
r	3
y	3
t	3
s	3

原始事务集支持度过滤及重排序后的结果：

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

支持度过滤

事务ID	过滤后的事务
001	r, z
002	z, y, x, t, s
003	z
004	r, x, s
005	y, r, x, z, t
006	y, z, x, s, t

排序

事务ID	过滤及重排序后的事务
001	z, r
002	z, x, y, t, s
003	z
004	x, r, s
005	z, x, y, t, r
006	z, x, y, t, s

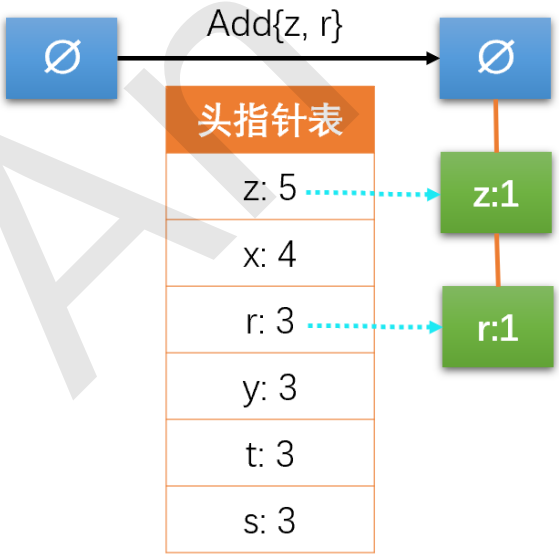
这里需要注意的是：事务集在重排序的时候，我们进行了两步操作：

第一步：按照频率大小对元素项进行排序；

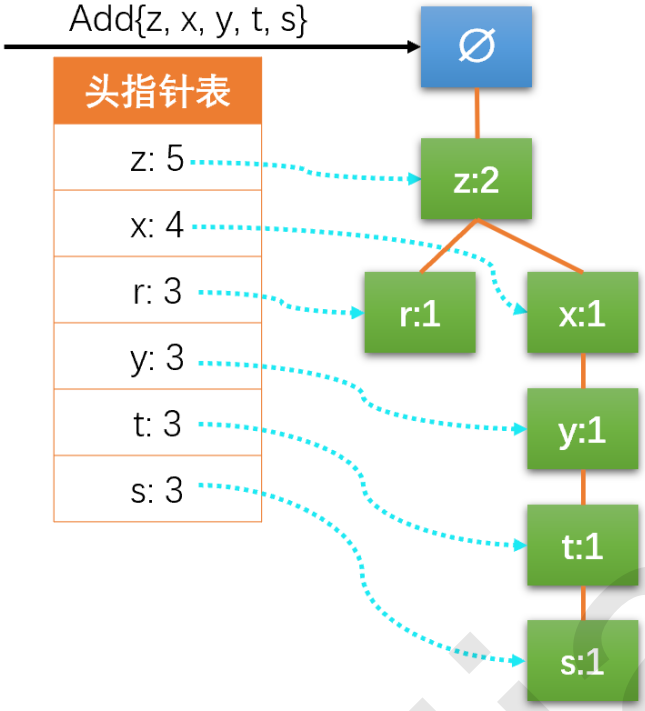
第二步：对于相同频率的元素项，则对关键字进行降序排列。

步骤四：构建FP树

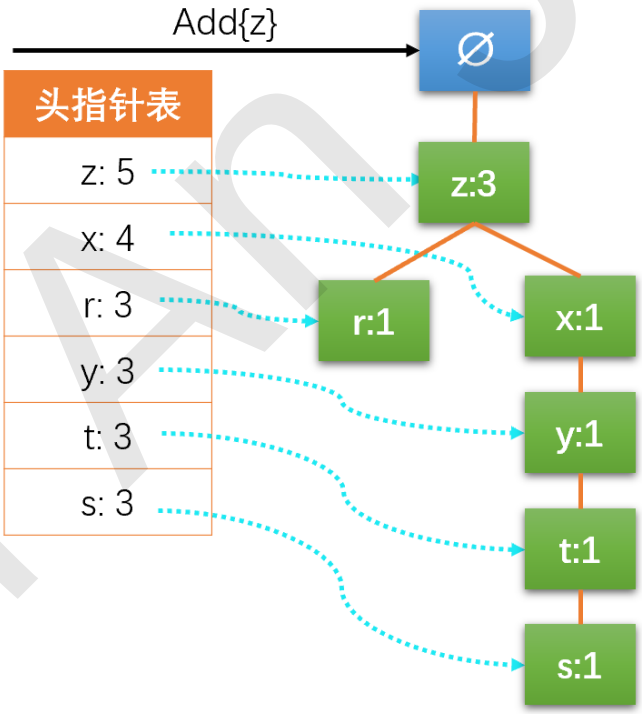
在对事务集过滤和重排序后，就可以构建FP树了。首先从空集（∅）开始，向其中不断添加频繁项集。如果树中已存在现有元素，则增加现有元素的值。如果现有元素不存在，则向树中添加一个分支。具体过程如下所示：



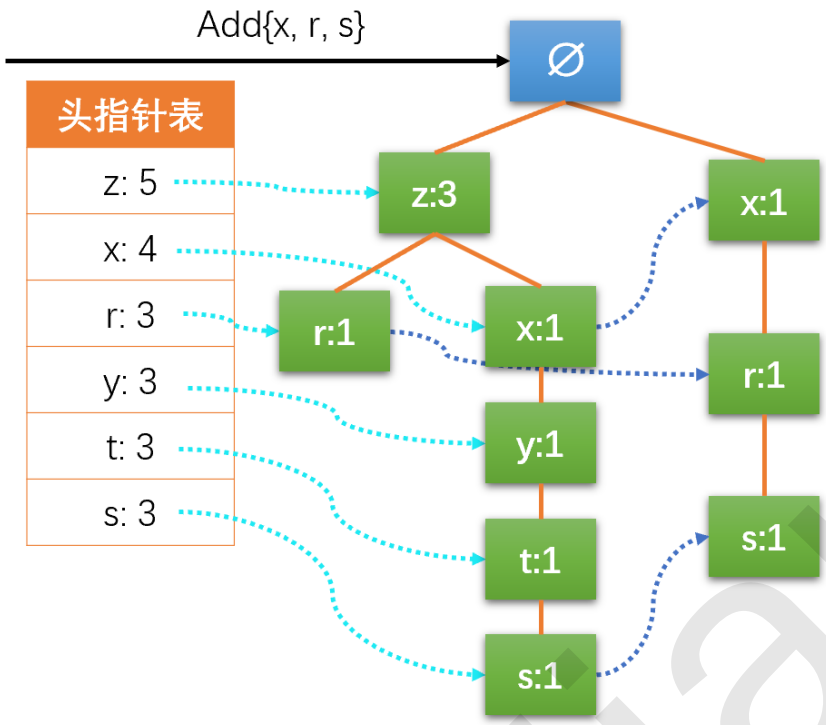
事务001，{z, r}



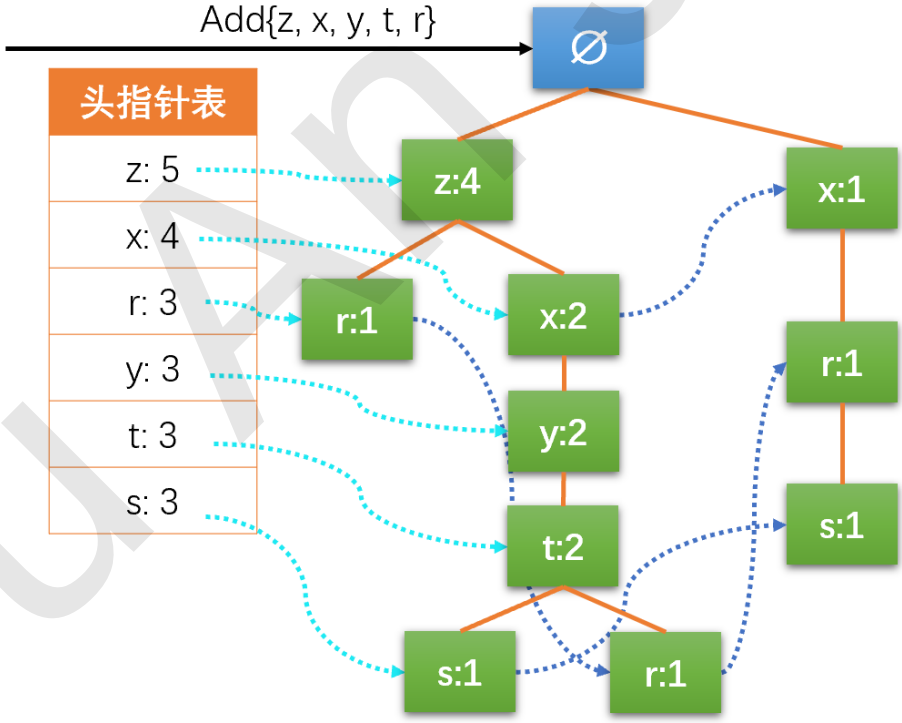
事务002，{z, x, y, t, s}



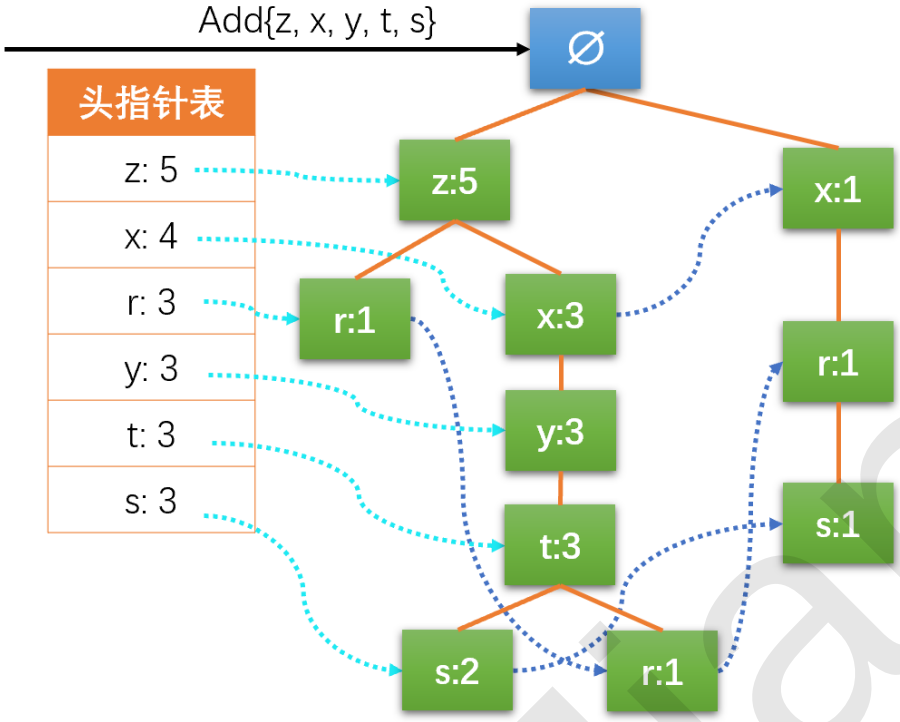
事务003，{z}



事务004，{x, r, s}

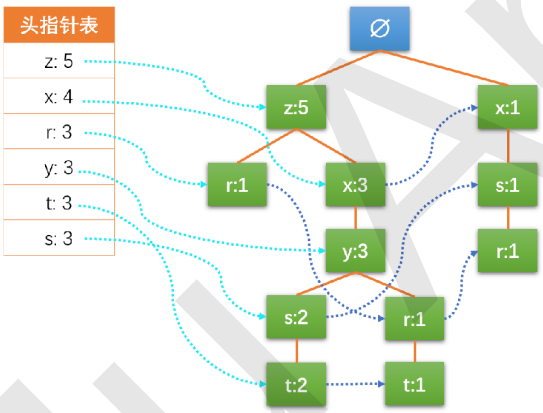


事务005，{z, x, y, t, r}

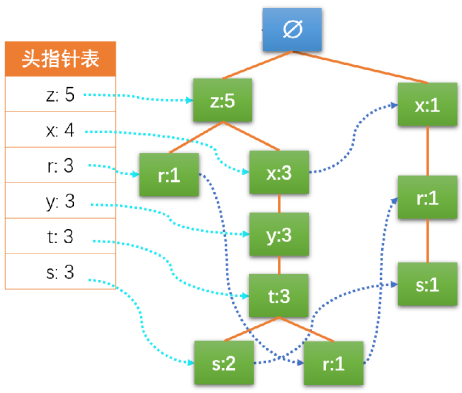


事务006，{z, x, y, t, s}

把六个事务都添加进去之后，我们的FP树就建立好了。细心的你肯定发现了，这棵树跟我们一开始示意的FP树不太一样。原因是：我们对频率相同的元素项进行了关键字排序处理。这说明，对项的关键字排序将会影响FP树的结构。进而，树的结构也将影响后续发现频繁项的结果。



图c　一棵FP树



最终的FP树

二、FP树的python实现

在了解FP树的详细建树过程之后，我们就可以根据这个步骤用python代码来建立FP树了。

1. 创建FP树的数据结构

由于FP树比之前建立的决策树要复杂，所以我们这里创建一个类来保存树的每一个节点。类中包含用于存放节点名字的变量和1个计数值，nodeLink变量用于链接相似的元素项，parent变量用于指向当前父节点（如果从上到下迭代访问节点的话不需要这个变量，此处使用是为后面的内容做铺垫），空字典变量用于存放节点的子节点。

另外，这个类中包含了两个方法：

- inc()方法的功能是对count变量增加给定值；
- disp()方法的功能是将树以文本形式显示，主要是方便调试。

```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue       #名字变量
        self.count = numOccur       #计数变量（频率）
        self.nodeLink = None        #链接相似元素项
        self.parent = parentNode    #当前父节点
        self.children = {}          #用于存放子节点

    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)         #子节点向右缩减
```

测试函数运行结果：

```
#创建一个单节点
rootNode = treeNode('这是父节点',9,None)
#创建一个子节点
rootNode.children['这是子节点'] = treeNode('这是子节点',13,None)
#显示节点
rootNode.disp()

#再增加一个子节点
rootNode.children['这是另一个子节点'] = treeNode('这是另一个子节点',3,None)
#显示节点
rootNode.disp()
```

In [2]:
```
#创建一个单节点
rootNode = treeNode('这是父节点',9,None)
#创建一个子节点
rootNode.children['这是子节点'] = treeNode('这是子节点',13,None)
#显示节点
rootNode.disp()
```

```
   这是父节点    9
     这是子节点    13
```

In [3]:
```
#再增加一个子节点
rootNode.children['这是另一个子节点'] = treeNode('这是另一个子节点',3,None)
#显示节点
rootNode.disp()
```

```
   这是父节点    9
     这是子节点    13
     这是另一个子节点    3
```

现在FP树所需数据结构已经建好，下面就可以来构建FP树了。

2. 创建简单数据集

首先我们创建一个简单数据集（上面所用例子）。

```
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
```

运行函数，查看运行结果：

```
simpDat = loadSimpDat()
simpDat
```

3. 创建FP树

函数1：数据格式化。

得到数据集后，我们需要遍历每一条交易数据，并把它们变成固定格式（frozenset）。这里使用了一个.setdefault()函数，该函数的功能是**返回指定键的值，如果键不在字典中，将会添加键并将值设置为一个指定值，默认为None**，该函数与get()函数功能类似，但是不同的是setdefault() 返回的键如果不在字典中，会添加键（更新字典），而 get() 不会添加键。

```
def createInitSet(dataSet):
    retDict = {}
    for trans in dataSet:
        #print(trans)
        fset = frozenset(trans)
        #print(fset)
        retDict.setdefault(fset, 0) #返回指定键的值，如果没有则添加一个键
        #print(retDict)
        retDict[fset] += 1
        #print(retDict)
    return retDict
```

运行函数，查看运行结果：

```
retDict = createInitSet(simpDat)
retDict
```

```
In [16]: retDict = createInitSet(simpDat)
         retDict

Out[16]: {frozenset({'h', 'j', 'p', 'r', 'z'}): 1,
          frozenset({'s', 't', 'u', 'v', 'w', 'x', 'y', 'z'}): 1,
          frozenset({'z'}): 1,
          frozenset({'n', 'o', 'r', 's', 'x'}): 1,
          frozenset({'p', 'q', 'r', 't', 'x', 'y', 'z'}): 1,
          frozenset({'e', 'm', 'q', 's', 't', 'x', 'y', 'z'}): 1}
```

函数2：更新头指针表。

updateHeader()函数的功能是确保节点链接指向树中该元素的每一个实例。从头指针表的nodeLink开始，一直沿着nodeLink知道到达链表末尾。

```
def updateHeader(nodeToTest, targetNode):
    while (nodeToTest.nodeLink != None):
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode
```

函数3：FP树的生长函数

该函数的主要功能是让树生长（这就是FP-growth中growth的来源）。首先测试事务中第一个元素项是不是子节点，如果是子节点，则更新count参数；如果不是子节点，则创建一个新的treeNode作为子节点添加到树中。此时，头指针表也要跟着更新以指向新的节点，这个更新需要调用updateHeader()函数。如果item中不止一个元素项的话，则将剩下的元素项作为参数进行迭代，注意：迭代的时候每次调用会去掉列表中的第一个元素（因为我们在前面判断过了）

```
def updateTree(items, myTree, headerTable, count):
    if items[0] in myTree.children:
        myTree.children[items[0]].inc(count)
    else:
        myTree.children[items[0]] = treeNode(items[0], count, myTree)
        if headerTable[items[0]][1] == None:
            headerTable[items[0]][1] = myTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], myTree.children[items[0]])
    if len(items) > 1:
        updateTree(items[1:], myTree.children[items[0]], headerTable, count)
```

函数4：创建FP树

该函数主要功能是创建FP树，树构建过程中会遍历数据集两次。第一次遍历数据集并统计每个元素项出现的次数。这些信息被存储在头指针表中。接下来扫描头指针表，删掉那些不满足最小支持度minSup的元素项。在头指针表中增加一列，用来存放指向每种类型第一个元素项的指针。然后创建只包含空集合∅的根节点。第二次遍历数据集，此次只考虑频繁项集。对频繁项集进行排序，最后调用updateTree() 让树生长。

【python filter()函数】：
用于过滤序列，过滤掉不符合条件的元素，返回由符合条件元素组成的新列表。
该函数接收两个参数，第一个为函数，第二个为序列，序列的每个元素作为参数传递给函数进行判断，
然后返回 True 或 False，最后将返回 True 的元素放到新列表中。

```
filter(function, iterable)
function -- 判断函数
iterable -- 可迭代对象
```

【注意】python3中，最后返回的对象是迭代器对象，如果要转换为列表，可以使用 list() 来转换。

函数代码：

```
def createTree(dataSet, minSup=1):
    headerTable = {}
    #第一次遍历数据集， 记录每个数据项的支持度
    for trans in dataSet:
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + 1

    #根据最小支持度过滤
    lessThanMinsup = list(filter(lambda k:headerTable[k] < minSup, headerTable.keys()))
    for k in lessThanMinsup:
        del(headerTable[k])
    freqItemSet = set(headerTable.keys())

    #如果所有数据都不满足最小支持度，返回None， None
    if len(freqItemSet) == 0:
        return None, None
    for k in headerTable:
        headerTable[k] = [headerTable[k], None]
```

```
    myTree = treeNode('ϕ', 1, None)

    #第二次遍历数据集，构建fp-tree
    for tranSet, count in dataSet.items():
        #根据最小支持度处理一条训练样本，key:样本中的一个样例，value:该样例的的全局支持度
        localD = {}
        for item in tranSet:
            if item in freqItemSet:
                localD[item] = headerTable[item][0]
        if len(localD) > 0:
            #根据全局频繁项对每个事务中的数据进行排序,等价于 order by p[1] desc, p[0] desc
            orderedItems = [v[0] for v in sorted(localD.items(), key=lambda p:
(p[1],p[0]), reverse=True)]
            updateTree(orderedItems, myTree, headerTable, count)
    return myTree, headerTable
```

设定最小支持度minSup=3，运行函数，查看运行结果：

```
myTree, headerTable = createTree(retDict,minSup=3)
```

从运行结果可以看出，这棵树与我们之前画出来的树一样，说明我们的函数没有问题。

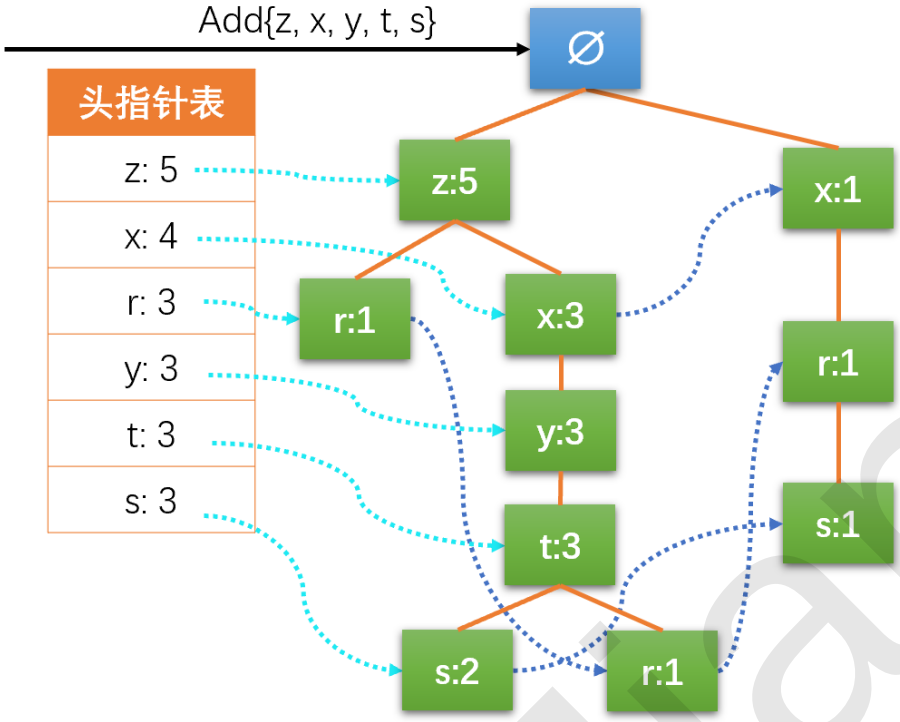
In [16]:
```
myTree, headerTable = createTree(retDict,minSup=3)
```

In [17]:
```
headerTable
```

Out[17]:
```
{'z': [5, <__main__.treeNode at 0x17b1d552048>],
 'r': [3, <__main__.treeNode at 0x17b1d552d68>],
 'x': [4, <__main__.treeNode at 0x17b1d552978>],
 's': [3, <__main__.treeNode at 0x17b1d552a58>],
 'y': [3, <__main__.treeNode at 0x17b1d5529b0>],
 't': [3, <__main__.treeNode at 0x17b1d5529e8>]}
```

In [18]:
```
myTree.disp()
```

```
  ϕ   1
    z   5
      r   1
      x   3
        y   3
          t   3
            s   2
            r   1
    x   1
      s   1
        r   1
```



事务006，{z, x, y, t, s}

【完整版】三、从FP树中挖掘频繁项集

有了FP树之后，我们就可以来挖掘频繁项集了。这里的思路和Apriori算法大致类似，首先从单元素项集合开始，然后在此基础上逐步构建更大的集合。当然，这里使用的是FP树而不是原始数据集。

从FP树中挖掘频繁项集的步骤：

1）从FP树中获得条件模式基；

2）利用条件模式基，构建一个条件FP树；

3）迭代：重复上述两个步骤，知道树包含一个元素项为止。

1. 抽取条件模式基

条件模式基（conditional pattern base）就是以所查找的元素项为结尾的路径集合。每一条路径其实都是一天前缀路径（prefix path）。什么叫前缀路径呢？其实就是介于所查找元素项与根节点之间的所有内容。

下面我们根据FP树写出每个频繁项的前缀路径：

每个频繁项集的前缀路径

频繁项	出现频率	前缀路径
z	5	{}5
x	4	{z}3，{}1
r	3	{z}1，{z, x, y, t}1，{x}1
y	3	{z, x}3
t	3	{z, x, y}3
s	3	{z, x, y, t}2，{x, r}1

2. 条件模式基的python实现

```
def ascendTree(leafNode, prefixPath):
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)
```

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

运行函数，查看结果：

```
simpDat = loadSimpDat()
dictDat = createInitSet(simpDat)
myFPTree,myheader = createTree(dictDat, 3)
myFPTree.disp()

condPats = findPrefixPath('z', myheader)
print('z', condPats)
condPats = findPrefixPath('x', myheader)
print('x', condPats)
condPats = findPrefixPath('y', myheader)
print('y', condPats)
condPats = findPrefixPath('t', myheader)
print('t', condPats)
condPats = findPrefixPath('s', myheader)
print('s', condPats)
```

```
condPats = findPrefixPath('r', myheader)
print('r', condPats)
```

3. 创造条件FP树

```
def mineTree(inTree, headerTable, minSup=1, preFix=set([]), freqItemList=[]):
    #排序minSup asc, value asc
    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: (p[1][0],p[0]))]
    for basePat in bigL:
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        freqItemList.append(newFreqSet)
        # 通过条件模式基找到的频繁项集
        condPattBases = findPrefixPath(basePat, headerTable)
        myCondTree, myHead = createTree(condPattBases, minSup)
        if myHead != None:
            print('condPattBases: ', basePat, condPattBases)
            myCondTree.disp()
            print('*' * 30)
            mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)
```

运行函数：

```
mineTree(myFPTree, myheader, 2)
```

【完整版】四、在Twitter源中发现一些共现词

【完整版】五、从新闻点击流中挖掘

其他

- 菊安酱的直播间：https://live.bilibili.com/14988341
- 下周一（2019/1/21）**最后一期**将讲解**奇异值分解SVD**，欢迎各位进入菊安酱的直播间观看直播
- 如有问题，可以给我留言哦~