昵称：qwertWZ
园龄：4年4个月
粉丝：9
关注：16
+加关注

< 2015年12月 >

日	一	二	三	四	五	六
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

我的标签

转载(13)
版本控制(6)
Python(5)
SQL Server 2005 编程入门经典(5)
机器学习(5)
机器学习实战(5)
操作指南(4)
Git(4)
备忘(3)
版本控制之道：使用Git(2)

使用Apriori算法和FP-growth算法进行关联分析

目录

系列文章：《机器学习实战》学习笔记

最近看了《机器学习实战》中的第11章（使用Apriori算法进行关联分析）和第12章（使用FP-growth算法来高效发现频繁项集）。正如章节标题所示，这两章讲了无监督机器学习方法中的关联分析问题。关联分析可以用于回答"哪些商品经常被同时购买？"之类的问题。书中举了一些关联分析的例子：

- 通过查看哪些商品经常在一起购买，可以帮助商店了解用户的购买行为。这种从数据海洋中抽取的知识可以用于商品定价、市场促销、存活管理等环节。
- 在美国国会投票记录中发现关联规则。在一个国会投票记录的数据集中发现议案投票的相关性，（原文：这里只是出于娱乐的目的，不过也可以.....）使用分析结果来为政治竞选活动服务，或者预测选举官员会如何投票。
- 发现毒蘑菇的相似特征。这里只对包含某个特定元素（有毒性）的项集感兴趣，从中寻找毒蘑菇中的一些公共特征，利用这些特征来避免吃到那些有毒蘑菇。
- 在Twitter源中发现一些共现词。对于给定搜索词，发现推文中频繁出现的单词集合。
- 从新闻网站点击流中挖掘新闻流行趋势，挖掘哪些新闻广泛被用户浏览到。
- 搜索引擎推荐，在用户输入查询词时推荐同相关的查询词项。

从大规模数据集中寻找物品间的隐含关系被称作关联分析（**association analysis**）或者关联规则学习（**association rule learning**）。这里的主要问题在于，寻找物品的不同组合是一项十分耗时的任务，所需的计算代价很高，蛮力搜索方法并不能解决这个问题，所以需要用更智能的方法在合理的时间范围内找到频繁项集。本文分别介绍如何使用Apriori算法和FP-growth算法来解决上述问题。

回到顶部

1. 关联分析

关联分析是在大规模数据集中寻找有趣关系的任务。这些关系可以有两种形式：

- 频繁项集
- 关联规则

频繁项集（**frequent item sets**）是经常出现在一块儿的物品的集合，关联规则（**association rules**）暗示两种物品之间可能存在很强的关系。

下面用一个例子来说明这两种概念：图1给出了某个杂货店的交易清单。

交易号码	商品
0	豆奶，莴苣
1	莴苣，尿布，葡萄酒，甜菜

更多

随笔档案(99)

2015年11月 (1)

2015年9月 (1)

2015年8月 (3)

2015年6月 (4)

2015年5月 (2)

2015年4月 (1)

2014年5月 (1)

2014年4月 (2)

2014年2月 (1)

2013年11月 (3)

2013年5月 (2)

2013年4月 (6)

2013年3月 (4)

2013年2月 (2)

2013年1月 (17)

2012年12月 (19)

2012年11月 (1)

2012年9月 (1)

2012年7月 (28)

阅读排行榜

1. SQL Server 2005 存储过程(7191)

2. Git忽略规则(3828)

3. Subversion命令汇总(2480)

4. 当前网络存在的安全问题(2431)

5. Android ListView 横向滑动删除 Item(2352)

6. 《SQL Server 2005 编程入门经典》第一到十二章(2241)

7. 使用Apriori算法和FP-growth算法进行关联分析(1765)

8. Windows 8下任何Windows Store

2	豆奶，尿布，葡萄酒，橙汁
3	莴苣，豆奶，尿布，葡萄酒
4	莴苣，豆奶，尿布，橙汁

图1 某杂货店交易清单

频繁项集是指那些经常出现在一起的商品集合，图中的集合{葡萄酒,尿布,豆奶}就是频繁项集的一个例子。从这个数据集中也可以找到诸如尿布->葡萄酒的关联规则，即如果有人买了尿布，那么他很可能也会买葡萄酒。

我们用支持度和可信度来度量这些有趣的关系。一个项集的支持度（**support**）被定义数据集中包含该项集的记录所占的比例。如上图中，{豆奶}的支持度为4/5，{豆奶,尿布}的支持度为3/5。支持度是针对项集来说的，因此可以定义一个最小支持度，而只保留满足最小值尺度的项集。

可信度或置信度（**confidence**）是针对关联规则来定义的。规则{尿布}→{啤酒}的可信度被定义为"支持度({尿布,啤酒})/支持度({尿布})"，由于{尿布,啤酒}的支持度为3/5，尿布的支持度为4/5，所以"尿布→啤酒"的可信度为3/4。这意味着对于包含"尿布"的所有记录，我们的规则对其中75%的记录都适用。

回到顶部

2. Apriori原理

假设我们有一家经营着4种商品（商品0，商品1，商品2和商品3）的杂货店，2图显示了所有商品之间所有的可能组合：

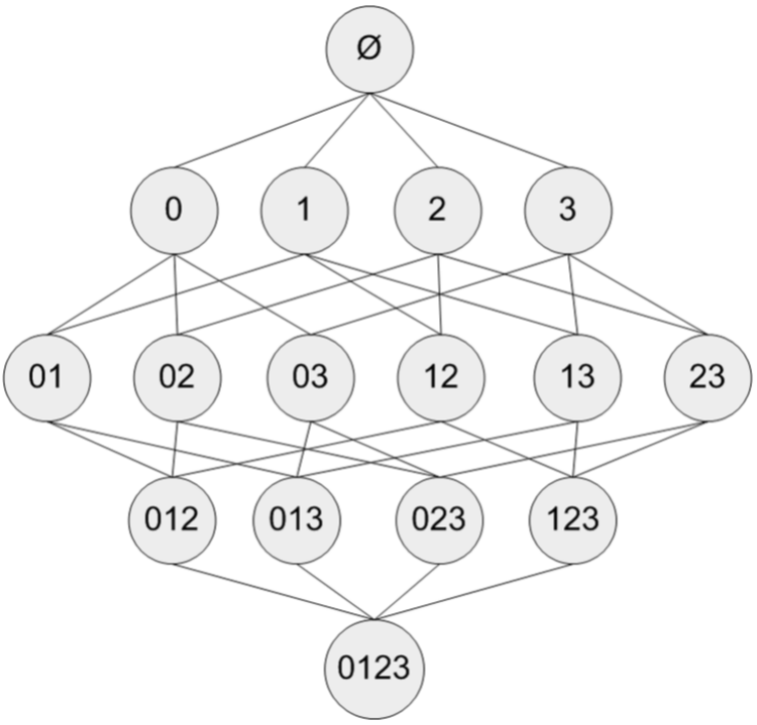


图2 集合{0,1,2,3,4}中所有可能的项集组合

对于单个项集的支持度，我们可以通过遍历每条记录并检查该记录是否包含该项集来计算。对于包含N中物品的数据集共有$2^N - 1$种项集组合，重复上述计算过程是不现实的。

研究人员发现一种所谓的Apriori原理，可以帮助我们减少计算量。**Apriori**原理是说如果某个项集是频繁的，那么它的所有子集也是频繁的。更常用的是它的逆否命题，即如果一个项集是非频繁的，那么它的所有超集也是非频繁的。

在图3中，已知阴影项集{2,3}是非频繁的。利用这个知识，我们就知道项集{0,2,3}，{1,2,3}以及{0,1,2,3}也是非频繁的。也就是说，一旦计算出了{2,3}的支持度，知道它是非频繁的后，就可以紧接着排除{0,2,3}、{1,2,3}和{0,1,2,3}。

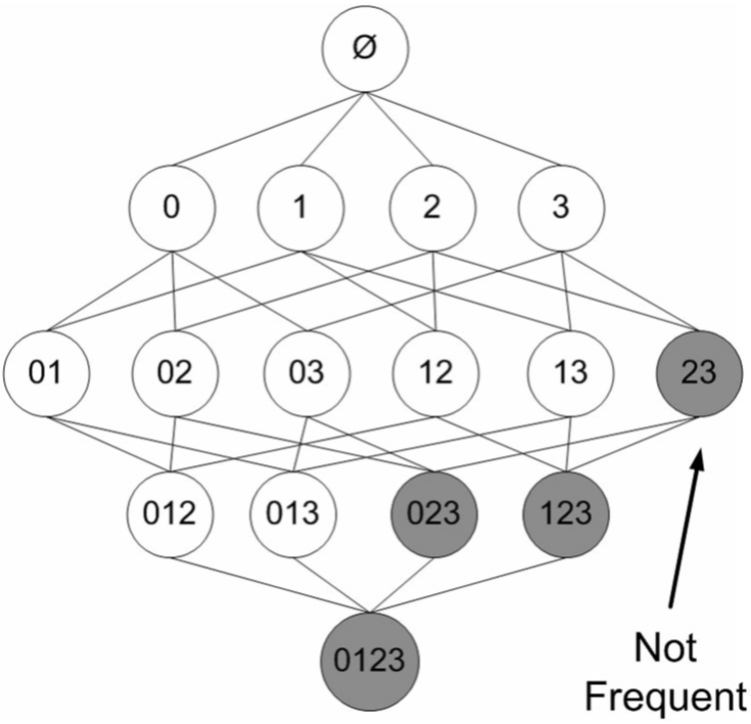


图3 图中给出了所有可能的项集，其中非频繁项集用灰色表示。

回到顶部

3. 使用Apriori算法来发现频繁集

前面提到，关联分析的目标包括两项：发现频繁项集和发现关联规则。首先需要找到频繁项集，然后才能获得关联规则（正如前文所讲，计算关联规则的可信度需要用到频繁项集的支持度）。

Apriori算法是发现频繁项集的一种方法。Apriori算法的两个输入参数分别是最小支持度和数据集。该算法首先会生成所有单个元素的项集列表。接着扫描数据集来查看哪些项集满足最小支持度要求，那些不满足最小支持度的集合会被去掉。然后，对剩下来的集合进行组合以生成包含两个元素的项集。接下来，再重新扫描交易记录，去掉不满足最小支持度的项集。该过程重复进行直到所有项集都被去掉。

3.1 生成候选项集

数据集扫描的伪代码大致如下：

```
对数据集中的每条交易记录tran:
    对每个候选项集can:
        检查can是否是tran的子集
        如果是，则增加can的计数
对每个候选项集：
    如果其支持度不低于最小值，则保留该项集
返回所有频繁项集列表
```

下面看一下实际代码，建立一个apriori.py文件并加入一下代码：

```
# coding=utf-8
from numpy import *

def loadDataSet():
    return [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]
```

其中numpy为程序中需要用到的Python库。

```
def createC1(dataSet):
    C1 = []
    for transaction in dataSet:
        for item in transaction:
            if not [item] in C1:
                C1.append([item])
    C1.sort()
    return map(frozenset, C1)
```

其中C1即为元素个数为1的项集（非频繁项集，因为还没有同最小支持度比较）。map(frozenset, C1)的语义是将C1由Python列表转换为不变集合（frozenset，Python中的数据结构）。

```
def scanD(D, Ck, minSupport):
    ssCnt = {}
    for tid in D:
        for can in Ck:
            if can.issubset(tid):
                ssCnt[can] = ssCnt.get(can, 0) + 1
    numItems = float(len(D))
    retList = []
    supportData = {}
    for key in ssCnt:
        support = ssCnt[key] / numItems
        if support >= minSupport:
            retList.insert(0, key)
        supportData[key] = support
    return retList, supportData
```

其中D为全部数据集，Ck为大小为k（包含k个元素）的候选项集，minSupport为设定的最小支持度。返回值中retList为在Ck中找出的频繁项集（支持度大于minSupport的），supportData记录各频繁项集的支持度。

retList.insert(0, key)一行将频繁项集插入返回列表的首部。

3.2 完整的Apriori算法

整个Apriori算法的伪代码如下：

> 当集合中项的个数大于0时：
>
> 构建一个由k个项组成的候选项集的列表（k从1开始）
>
> 计算候选项集的支持度，删除非频繁项集
>
> 构建由k+1项组成的候选项集的列表

程序代码如下：

```
def aprioriGen(Lk, k):
    retList = []
    lenLk = len(Lk)
    for i in range(lenLk):
        for j in range(i + 1, lenLk):
            # 前k-2项相同时，将两个集合合并
            L1 = list(Lk[i])[:k-2]; L2 = list(Lk[j])[:k-2]
            L1.sort(); L2.sort()
            if L1 == L2:
                retList.append(Lk[i] | Lk[j])
    return retList
```

该函数通过频繁项集列表L_k和项集个数k生成候选项集C_{k+1}。

注意其生成的过程中，首选对每个项集按元素排序，然后每次比较两个项集，只有在前k-1项相同时才将这两项合并。这样做是因为函数并非要两两合并各个集合，那样生成的集合并非都是k+1项的。在限制项数为k+1的前提下，只有在前k-1项相同、最后一项不相同的情况下合并才为所需要的新候选项集。

由于Python中使用下标0表示第一个元素，因此代码中的**[:k-2]**的实际作用为取列表的前k-1个元素。

```
def apriori(dataSet, minSupport=0.5):
    C1 = createC1(dataSet)
    D = map(set, dataSet)
    L1, supportData = scanD(D, C1, minSupport)
    L = [L1]
    k = 2
    while (len(L[k-2]) > 0):
        Ck = aprioriGen(L[k-2], k)
        Lk, supK = scanD(D, Ck, minSupport)
        supportData.update(supK)
        L.append(Lk)
        k += 1
```

```
    return L, supportData
```

该函数为Apriori算法的主函数，按照前述伪代码的逻辑执行。Ck表示项数为k的候选项集，最初的C1通过createC1()函数生成。Lk表示项数为k的频繁项集，supK为其支持度，Lk和supK由scanD()函数通过Ck计算而来。

函数返回的L和supportData为所有的频繁项集及其支持度，因此在每次迭代中都要将所求得的Lk和supK添加到L和supportData中。

代码测试（在Python提示符下输入）：

```
>>> import apriori
>>> dataSet = apriori.loadDataSet()
>>> dataSet
>>> C1 = apriori.createC1(dataSet)
>>> D = map(set, dataSet)
>>> D
>>> L1, suppDat = apriori.scanD(D, C1, 0.5)
>>> L1
>>> L, suppData = apriori.apriori(dataSet)
>>> L
>>> L, suppData = apriori.apriori(dataSet, minSupport=0.7)
>>> L
```

L返回的值为frozenset列表的形式：

```
[[frozenset([1]), frozenset([3]), frozenset([2]), frozenset([5])],
[frozenset([1, 3]), frozenset([2, 5]), frozenset([2, 3]), frozenset([3,
5])],
[frozenset([2, 3, 5])], []]
```

即L[0]为项数为1的频繁项集：

```
[frozenset([1]), frozenset([3]), frozenset([2]), frozenset([5])]
```

L[1]为项数为2的频繁项集：

```
[frozenset([1, 3]), frozenset([2, 5]), frozenset([2, 3]), frozenset([3, 5])]
```

依此类推。

suppData为一个字典，它包含项集的支持度。

3.3 从频繁集中挖掘相关规则

解决了频繁项集问题，下一步就可以解决相关规则问题。

要找到关联规则，我们首先从一个频繁项集开始。从杂货店的例子可以得到，如果有一个频繁项集{豆奶, 莴苣}，那么就可能有一条关联规则“豆奶➞莴苣”。这意味着如果有人购买了豆奶，那么在统计上他会购买莴苣的概率较大。注意这一条反过来并不总是成立，也就是说，可信度(“豆奶➞莴苣”)并不等于可信度(“莴苣➞豆奶”)。

前文也提到过，一条规则P➞H的可信度定义为support(P | H)/support(P)，其中“|”表示P和H的并集。可见可信度的计算是基于项集的支持度的。

图4给出了从项集{0,1,2,3}产生的所有关联规则，其中阴影区域给出的是低可信度的规则。可以发现如果{0,1,2}➞{3}是一条低可信度规则，那么所有其他以3作为后件（箭头右部包含3）的规则均为低可信度的。

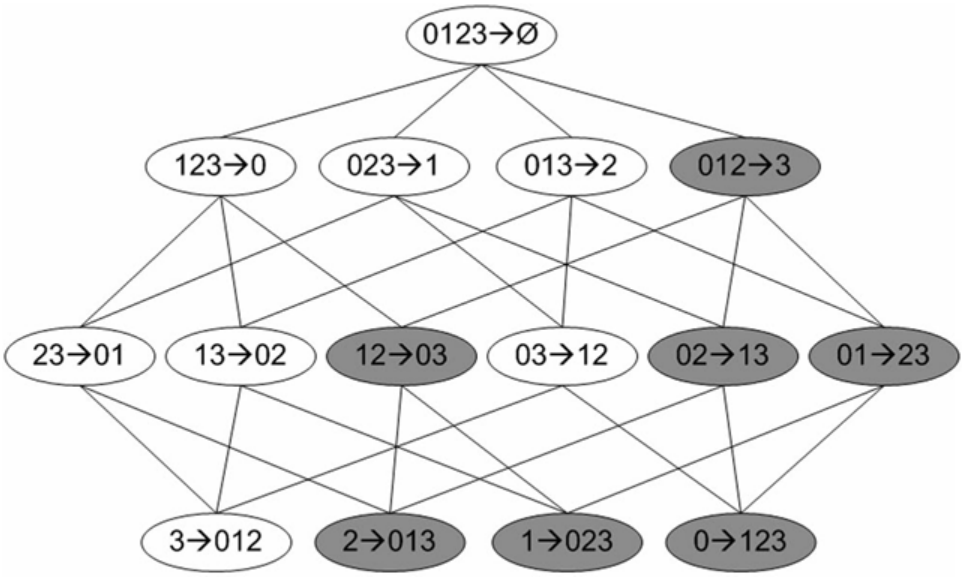


图4 频繁项集{0,1,2,3}的关联规则网格示意图

可以观察到，如果某条规则并不满足最小可信度要求，那么该规则的所有子集也不会满足最小可信度要求。以图4为例，假设规则{0,1,2} → {3}并不满足最小可信度要求，那么就知道任何左部为{0,1,2}子集的规则也不会满足最小可信度要求。可以利用关联规则的上述性质属性来减少需要测试的规则数目，类似于Apriori算法求解频繁项集。

1 书中的原始代码

1 关联规则生成函数：

```
def generateRules(L, supportData, minConf=0.7):
    bigRuleList = []
    for i in range(1, len(L)):
        for freqSet in L[i]:
            H1 = [frozenset([item]) for item in freqSet]
            if (i > 1):
                # 三个及以上元素的集合
                rulesFromConseq(freqSet, H1, supportData, bigRuleList, minConf)
            else:
                # 两个元素的集合
                calcConf(freqSet, H1, supportData, bigRuleList, minConf)
    return bigRuleList
```

这个函数是主函数，调用其他两个函数。其他两个函数是rulesFromConseq()和calcConf()，分别用于生成候选规则集合以及对规则进行评估（计算支持度）。

函数generateRules()有3个参数：频繁项集列表L、包含那些频繁项集支持数据的字典supportData、最小可信度阈值minConf。函数最后要生成一个包含可信度的规则列表bigRuleList，后面可以基于可信度对它们进行排序。L和supportData正好为函数apriori()的输出。该函数遍历L中的每一个频繁项集，并对每个频繁项集构建只包含单个元素集合的列表H1。代码中的i指示当前遍历的频繁项集包含的元素个数，freqSet为当前遍历的频繁项集（回忆L的组织结构是先把具有相同元素个数的频繁项集组织成列表，再将各个列表组成一个大列表，所以为遍历L中的频繁项集，需要使用两层for循环）。

2 辅助函数——计算规则的可信度，并过滤出满足最小可信度要求的规则

```
def calcConf(freqSet, H, supportData, brl, minConf=0.7):
    ''' 对候选规则集进行评估 '''
    prunedH = []
    for conseq in H:
        conf = supportData[freqSet] / supportData[freqSet - conseq]
        if conf >= minConf:
            print freqSet - conseq, '-->', conseq, 'conf:', conf
            brl.append((freqSet - conseq, conseq, conf))
            prunedH.append(conseq)
    return prunedH
```

计算规则的可信度以及找出满足最小可信度要求的规则。函数返回一个满足最小可信度要求的规则列

表，并将这个规则列表添加到主函数的bigRuleList中（通过参数brl）。返回值prunedH保存规则列表的右部，这个值将在下一个函数rulesFromConseq()中用到。

3 辅助函数——根据当前候选规则集H生成下一层候选规则集

```
def rulesFromConseq(freqSet, H, supportData, brl, minConf=0.7):
    ''' 生成候选规则集 '''
    m = len(H[0])
    if (len(freqSet) > (m + 1)):
        Hmpl = aprioriGen(H, m + 1)
        Hmpl = calcConf(freqSet, Hmpl, supportData, brl, minConf)
        if (len(Hmpl) > 1):
            rulesFromConseq(freqSet, Hmpl, supportData, brl, minConf)
```

从最初的项集中生成更多的关联规则。该函数有两个参数：频繁项集freqSet，可以出现在规则右部的元素列表H。其余参数：supportData保存项集的支持度，brl保存生成的关联规则，minConf同主函数。函数先计算H中的频繁项集大小m。接下来查看该频繁项集是否大到可以移除大小为m的子集。如果可以的话，则将其移除。使用函数aprioriGen()来生成H中元素的无重复组合，结果保存在Hmp1中，这也是下一次迭代的H列表。

实际运行效果：

```
>>> import apriori
>>> dataSet = apriori.loadDataSet()
>>> L, suppData = apriori.apriori(dataSet, minSupport=0.5)
>>> rules = apriori.generateRules(L, suppData, minConf=0.7)
>>> rules
```

```
frozenset([1]) --> frozenset([3]) conf: 1.0
frozenset([5]) --> frozenset([2]) conf: 1.0
frozenset([2]) --> frozenset([5]) conf: 1.0
```

```
>>> rules = apriori.generateRules(L, suppData, minConf=0.5)
>>> rules
```

```
frozenset([3]) --> frozenset([1]) conf: 0.6666666666667
frozenset([1]) --> frozenset([3]) conf: 1.0
frozenset([5]) --> frozenset([2]) conf: 1.0
frozenset([2]) --> frozenset([5]) conf: 1.0
frozenset([3]) --> frozenset([2]) conf: 0.6666666666667
frozenset([2]) --> frozenset([3]) conf: 0.6666666666667
frozenset([5]) --> frozenset([3]) conf: 0.6666666666667
frozenset([3]) --> frozenset([5]) conf: 0.6666666666667
frozenset([5]) --> frozenset([2, 3]) conf: 0.6666666666667
frozenset([3]) --> frozenset([2, 5]) conf: 0.6666666666667
frozenset([2]) --> frozenset([3, 5]) conf: 0.6666666666667
```

到目前为止，如果代码同书中一样的话，输出就是这样。在这里首先使用参数最小支持度minSupport = 0.5计算频繁项集L和支持度suppData，然后分别计算最小可信度minConf = 0.7和minConf = 0.5的关联规则。

2 关于**rulesFromConseq()**函数的问题

如果仔细看下上述代码和输出，会发现这里面是一些问题的。

1 问题的提出

频繁项集L的值前面提到过。我们在其中计算通过{2, 3, 5}生成的关联规则，可以发现关联规则{3, 5}→{2}和{2, 3}→{5}的可信度都应该为1.0的，因而也应该包括在当minConf = 0.7时的rules中——但是这在前面的运行结果中并没有体现出来。minConf = 0.5时也是一样，{3, 5}→{2}的可信度为1.0，{2, 5}→{3}的可信度为2/3，{2, 3}→{5}的可信度为1.0，也没有体现在rules中。

通过分析程序代码，我们可以发现：

- 当i = 1时，generateRules()函数直接调用了calcConf()函数直接计算其可信度，因为这时L[1]中的频繁项集均包含两个元素，可以直接生成和判断候选关联规则。比如L[1]中的{2, 3}，生成的候选关联规则为{2}→{3}、{3}→{2}，这样就可以了。

- 当i > 1时，generateRules()函数调用了rulesFromConseq()函数，这时L[i]中至少包含3个元素，如{2, 3, 5}，对候选关联规则的生成和判断的过程需要分层进行（图4）。这里，将初始的H1（表示初始关联规则的右部，即箭头右边的部分）作为参数传递给了rulesFromConseq()函数。

例如，对于频繁项集{a, b, c, ...}，H1的值为[a, b, c, ...]（代码中实际为frozenset类型）。如果将H1带入计算可信度的calcConf()函数，在函数中会依次计算关联规则{b, c, d, ...}➞{a}、{a, c, d, ...}➞{b}、{a, b, d, ...}➞{c}.....的支持度，并保存支持度大于最小支持度的关联规则，并保存这些规则的右部（prunedH，即对H的过滤，删除支持度过小的关联规则）。

当i > 1时没有直接调用calcConf()函数计算通过H1生成的规则集。在rulesFromConseq()函数中，首先获得当前H的元素数m = len(H[0])（记当前的H为H_m）。当H_m可以进一步合并为m+1元素数的集合H_{m+1}时（判断条件：len(freqSet) > (m + 1)），依次：

- 生成H_{m+1}：Hmpl = aprioriGen(H, m + 1)
- 计算H_{m+1}的可信度：Hmpl = calcConf(freqSet, Hmpl, ...)
- 递归计算由H_{m+1}生成的关联规则：rulesFromConseq(freqSet, Hmpl, ...)

所以这里的问题是，在i>1时，rulesFromConseq()函数中并没有调用calcConf()函数计算**H1**的可信度，而是直接由H1生成H2，从H2开始计算关联规则——于是由元素数>3的频繁项集生成的{a, b, c, ...}➞{x}形式的关联规则（图4中的第2层）均缺失了。由于代码示例数据中的对H1的剪枝prunedH没有删除任何元素，结果只是"巧合"地缺失了一层。正常情况下如果没有对H1进行过滤，直接生成H2，将给下一层带入错误的结果（如图4中的012➞3会被错误得留下来）。

2 对问题代码的修改

在i>1时，将对H1调用calcConf()的过程加上就可以了。比如可以这样：

```
def generateRules2(L, supportData, minConf=0.7):
    bigRuleList = []
    for i in range(1, len(L)):
        for freqSet in L[i]:
            H1 = [frozenset([item]) for item in freqSet]
            if (i > 1):
                # 三个及以上元素的集合
                H1 = calcConf(freqSet, H1, supportData, bigRuleList, minConf)
                rulesFromConseq(freqSet, H1, supportData, bigRuleList, minConf)
            else:
                # 两个元素的集合
                calcConf(freqSet, H1, supportData, bigRuleList, minConf)
    return bigRuleList
```

这里就只需要修改generateRules()函数。这样实际运行效果中，刚才丢失的那几个关联规则就都出来了。

进一步修改：当i=1时的else部分并没有独特的逻辑，这个if语句可以合并，然后再修改rulesFromConseq()函数，保证其会调用calcConf(freqSet, H1, ...)：

```
def generateRules3(L, supportData, minConf=0.7):
    bigRuleList = []
    for i in range(1, len(L)):
        for freqSet in L[i]:
            H1 = [frozenset([item]) for item in freqSet]
            rulesFromConseq2(freqSet, H1, supportData, bigRuleList, minConf)
    return bigRuleList

def rulesFromConseq2(freqSet, H, supportData, brl, minConf=0.7):
    m = len(H[0])
    if (len(freqSet) > m): # 判断长度改为 > m，这时即可以求H的可信度
        Hmpl = calcConf(freqSet, H, supportData, brl, minConf)
        if (len(Hmpl) > 1): # 判断求完可信度后是否还有可信度大于阈值的项用来生成下一层H
            Hmpl = aprioriGen(Hmpl, m + 1)
            rulesFromConseq2(freqSet, Hmpl, supportData, brl, minConf) # 递归计算，不变
```

运行结果和generateRules2相同。

进一步修改：消除rulesFromConseq2()函数中的递归项。这个递归纯粹是偷懒的结果，没有简化任何

逻辑和增加任何可读性，可以直接用一个循环代替：

```
def rulesFromConseq3(freqSet, H, supportData, brl, minConf=0.7):
    m = len(H[0])
    while (len(freqSet) > m): # 判断长度 > m，这时即可求H的可信度
        H = calcConf(freqSet, H, supportData, brl, minConf)
        if (len(H) > 1): # 判断求完可信度后是否还有可信度大于阈值的项用来生成下一层H
            H = aprioriGen(H, m + 1)
            m += 1
        else: # 不能继续生成下一层候选关联规则，提前退出循环
            break
```

另一个主要的区别是去掉了多余的Hmpl变量。运行的结果和generateRules2相同。

至此，一个完整的Apriori算法就完成了。

3.4 小结

关联分析是用于发现大数据集中元素间有趣关系的一个工具集，可以采用两种方式来量化这些有趣的关系。第一种方式是使用频繁项集，它会给出经常在一起出现的元素项。第二种方式是关联规则，每条关联规则意味着元素项之间的"如果.....那么"关系。

发现元素项间不同的组合是个十分耗时的任务，不可避免需要大量昂贵的计算资源，这就需要一些更智能的方法在合理的时间范围内找到频繁项集。能够实现这一目标的一个方法是Apriori算法，它使用Apriori原理来减少在数据库上进行检查的集合的数目。Apriori原理是说如果一个元素项是不频繁的，那么那些包含该元素的超集也是不频繁的。Apriori算法从单元素项集开始，通过组合满足最小支持度要求的项集来形成更大的集合。支持度用来度量一个集合在原始数据中出现的频率。

关联分析可以用在许多不同物品上。商店中的商品以及网站的访问页面是其中比较常见的例子。

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。当数据集很大时，这会显著降低频繁项集发现的速度。下面会介绍FP-growth算法，和Apriori算法相比，该算法只需要对数据库进行两次遍历，能够显著加快发现频繁项集的速度。

回到顶部

4. 使用FP-growth算法来高效发现频繁项集

FP-growth算法基于Apriori构建，但采用了高级的数据结构减少扫描次数，大大加快了算法速度。FP-growth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，因此FP-growth算法的速度要比Apriori算法快。

FP-growth算法发现频繁项集的基本过程如下：

- 构建FP树
- 从FP树中挖掘频繁项集

> FP-growth算法
>
> - 优点：一般要快于Apriori。
> - 缺点：实现比较困难，在某些数据集上性能会下降。
> - 适用数据类型：离散型数据。

4.1 FP树：用于编码数据集的有效方式

FP-growth算法将数据存储在一种称为FP树的紧凑数据结构中。FP代表频繁模式（Frequent Pattern）。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。图5给出了FP树的一个例子。

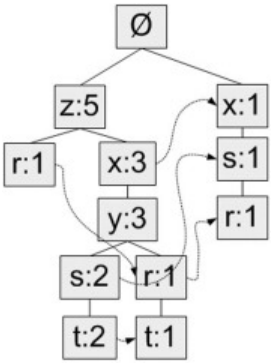


图5 一棵FP树，和一般的树结构类似，包含着连接相似节点（值相同的节点）的连接

与搜索树不同的是，一个元素项可以在一棵FP树种出现多次。FP树辉存储项集的出现频率，而每个项集会以路径的方式存储在数中。存在相似元素的集合会共享树的一部分。只有当集合之间完全不同时，树才会分叉。 树节点上给出集合中的单个元素及其在序列中的出现次数，路径会给出该序列的出现次数。

相似项之间的链接称为节点链接（node link），用于快速发现相似项的位置。

举例说明，下表用来产生图5的FP树：

用于生成图5中FP树的事务数据样例

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

对**FP**树的解读：

图5中，元素项z出现了5次，集合{r, z}出现了1次。于是可以得出结论：z一定是自己本身或者和其他符号一起出现了4次。集合{t, s, y, x, z}出现了2次，集合{t, r, y, x, z}出现了1次，z本身单独出现1次。就像这样，FP树的解读方式是读取某个节点开始到根节点的路径。路径上的元素构成一个频繁项集，开始节点的值表示这个项集的支持度。根据图5，我们可以快速读出项集{z}的支持度为5、项集{t, s, y, x, z}的支持度为2、项集{r, y, x, z}的支持度为1、项集{r, s, x}的支持度为1。FP树中会多次出现相同的元素项，也是因为同一个元素项会存在于多条路径，构成多个频繁项集。但是频繁项集的共享路径是会合并的，如图中的{t, s, y, x, z}和{t, r, y, x, z}

和之前一样，我们取一个最小阈值，出现次数低于最小阈值的元素项将被直接忽略。图5中将最小支持度设为3，所以q和p没有在FP中出现。

FP-growth算法的工作流程如下。首先构建FP树，然后利用它来挖掘频繁项集。为构建FP树，需要对原始数据集扫描两遍。第一遍对所有元素项的出现次数进行计数。数据库的第一遍扫描用来统计出现的频率，而第二遍扫描中只考虑那些频繁元素。

4.2 构建**FP**树

1 创建**FP**树的数据结构

由于树节点的结构比较复杂，我们使用一个类表示。创建文件fpGrowth.py并加入下列代码：

```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue
        self.count = numOccur
        self.nodeLink = None
        self.parent = parentNode
        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():
```

```
15                child.disp(ind + 1)
```

每个树节点由五个数据项组成：

- name：节点元素名称，在构造时初始化为给定值
- count：出现次数，在构造时初始化为给定值
- nodeLink：指向下一个相似节点的指针，默认为None
- parent：指向父节点的指针，在构造时初始化为给定值
- children：指向子节点的字典，以子节点的元素名称为键，指向子节点的指针为值，初始化为空字典

成员函数：

- inc()：增加节点的出现次数值
- disp()：输出节点和子节点的FP树结构

测试代码：

```
>>> import fpGrowth
>>> rootNode = fpGrowth.treeNode('pyramid', 9, None)
>>> rootNode.children['eye'] = fpGrowth.treeNode('eye', 13, None)
>>> rootNode.children['phoenix'] = fpGrowth.treeNode('phoenix', 3, None)
>>> rootNode.disp()
```

2 构建FP树

头指针表

FP-growth算法还需要一个称为头指针表的数据结构，其实很简单，就是用来记录各个元素项的总出现次数的数组，再附带一个指针指向FP树中该元素项的第一个节点。这样每个元素项都构成一条单链表。图示说明：

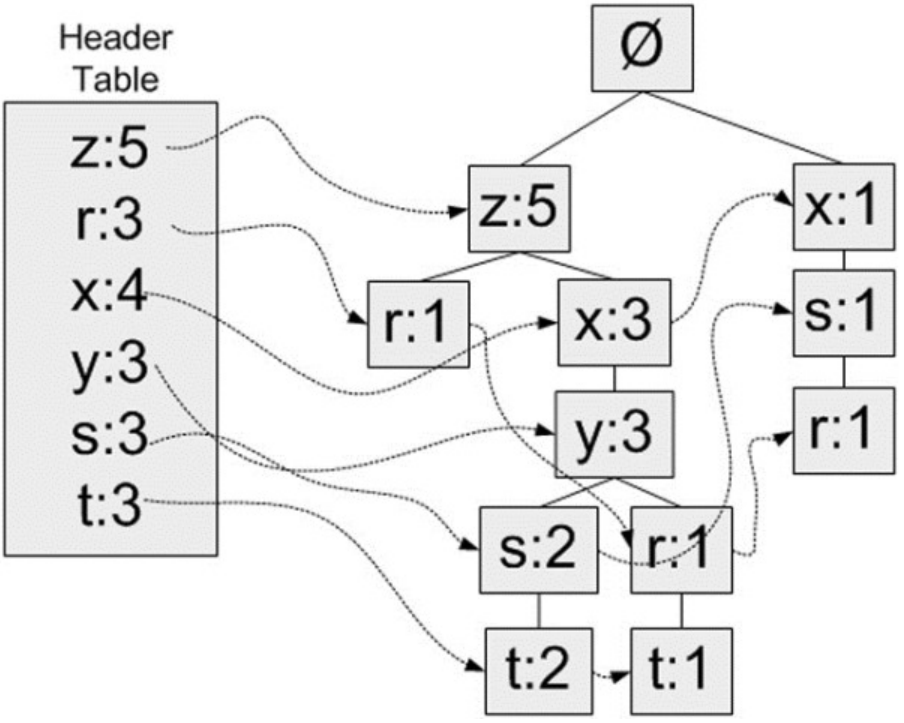


图6 带头指针表的FP树，头指针表作为一个起始指针来发现相似元素项

这里使用Python字典作为数据结构，来保存头指针表。以元素项名称为键，保存出现的总次数和一个指向第一个相似元素项的指针。

第一次遍历数据集会获得每个元素项的出现频率，去掉不满足最小支持度的元素项，生成这个头指针表。

元素项排序

上文提到过，FP树会合并相同的频繁项集（或相同的部分）。因此为判断两个项集的相似程度需要对项集中的元素进行排序（不过原因也不仅如此，还有其它好处）。排序基于元素项的绝对出现频率（总的

出现次数）来进行。在第二次遍历数据集时，会读入每个项集（读取），去掉不满足最小支持度的元素项（过滤），然后对元素进行排序（重排序）。

对示例数据集进行过滤和重排序的结果如下：

事务ID	事务中的元素项	过滤及重排序后的事务
001	r, z, h, j, p	z, r
002	z, y, x, w, v, u, t, s	z, x, y, s, t
003	z	z
004	r, x, n, o, s	x, s, r
005	y, r, x, z, q, t, p	z, x, y, r, t
006	y, z, x, e, q, s, t, m	z, x, y, s, t

构建**FP**树

在对事务记录过滤和排序之后，就可以构建FP树了。从空集开始，将过滤和重排序后的频繁项集一次添加到树中。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，则向树添加一个分支。对前两条事务进行添加的过程：

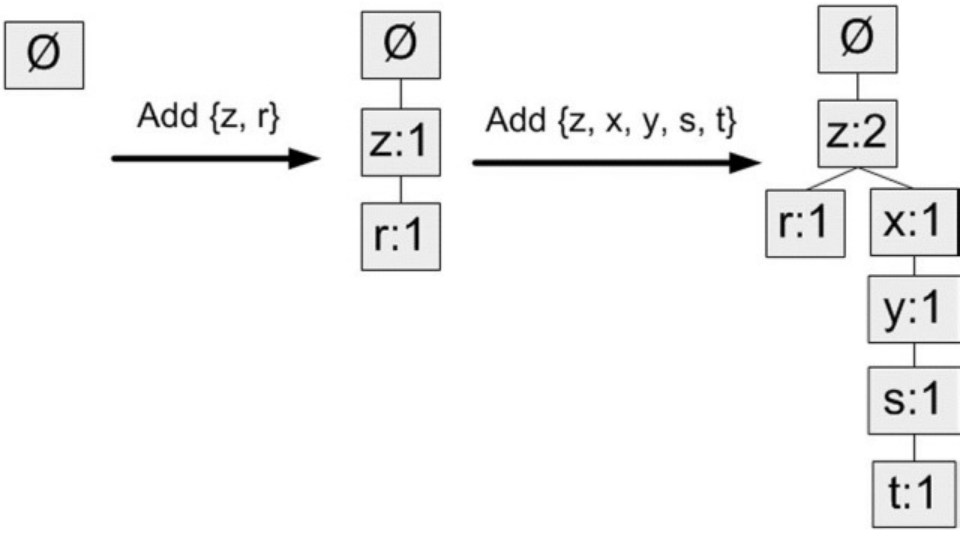


图7 FP树构建过程示意（添加前两条事务）

算法：构建FP树

输入：数据集、最小值尺度
输出：FP树、头指针表
1. 遍历数据集，统计各元素项出现次数，创建头指针表
2. 移除头指针表中不满足最小值尺度的元素项
3. 第二次遍历数据集，创建FP树。对每个数据集中的项集：
    3.1 初始化空FP树
    3.2 对每个项集进行过滤和重排序
    3.3 使用这个项集更新FP树，从FP树的根节点开始：
        3.3.1 如果当前项集的第一个元素项存在于FP树当前节点的子节点中，则更新这个子节点的计数值
        3.3.2 否则，创建新的子节点，更新头指针表
        3.3.3 对当前项集的其余元素项和当前元素项的对应子节点递归3.3的过程

代码（在fpGrowth.py中加入下面的代码）：

1 总函数：createTree

```
def createTree(dataSet, minSup=1):
    ''' 创建FP树 '''
    # 第一次遍历数据集，创建头指针表
    headerTable = {}
    for trans in dataSet:
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]
    # 移除不满足最小支持度的元素项
```

```
    for k in headerTable.keys():
        if headerTable[k] < minSup:
            del(headerTable[k])
    # 空元素集，返回空
    freqItemSet = set(headerTable.keys())
    if len(freqItemSet) == 0:
        return None, None
    # 增加一个数据项，用于存放指向相似元素项指针
    for k in headerTable:
        headerTable[k] = [headerTable[k], None]
    retTree = treeNode('Null Set', 1, None) # 根节点
    # 第二次遍历数据集，创建FP树
    for tranSet, count in dataSet.items():
        localD = {} # 对一个项集tranSet，记录其中每个元素项的全局频率，用于排序
        for item in tranSet:
            if item in freqItemSet:
                localD[item] = headerTable[item][0] # 注意这个[0]，因为之前加过一个数据项
        if len(localD) > 0:
            orderedItems = [v[0] for v in sorted(localD.items(), key=lambda p: p[1],
reverse=True)] # 排序
            updateTree(orderedItems, retTree, headerTable, count) # 更新FP树
    return retTree, headerTable
```

（代码比较宽，大家的显示器都那么大，应该没关系吧……）

需要注意的是，参数中的dataSet的格式比较奇特，不是直觉上得集合的list，而是一个集合的字典，以这个集合为键，值部分记录的是这个集合出现的次数。于是要生成这个dataSet还需要后面的createInitSet()函数辅助。因此代码中第7行中的`dataSet[trans]`实际获得了这个trans集合的出现次数（在本例中均为1），同样第21行的"`for tranSet, count in dataSet.items():`"获得了tranSet和count分别表示一个项集和该项集的出现次数。——这样做是为了适应后面在挖掘频繁项集时生成的条件FP树。

2 辅助函数：updateTree

```
def updateTree(items, inTree, headerTable, count):
    if items[0] in inTree.children:
        # 有该元素项时计数值+1
        inTree.children[items[0]].inc(count)
    else:
        # 没有这个元素项时创建一个新节点
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        # 更新头指针表或前一个相似元素项节点的指针指向新节点
        if headerTable[items[0]][1] == None:
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], inTree.children[items[0]])

    if len(items) > 1:
        # 对剩下的元素项迭代调用updateTree函数
        updateTree(items[1::], inTree.children[items[0]], headerTable, count)
```

3 辅助函数：updateHeader

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

这个函数其实只做了一件事，就是获取头指针表中该元素项对应的单链表的尾节点，然后将其指向新节点targetNode。

生成数据集

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

生成的样例数据同文中用得一样。这个诡异的输入格式就是createInitSet()函数中这样来得。

测试代码

```
>>> import fpGrowth
>>> simpDat = fpGrowth.loadSimpDat()
>>> initSet = fpGrowth.createInitSet(simpDat)
>>> myFPtree, myHeaderTab = fpGrowth.createTree(initSet, 3)
>>> myFPtree.disp()
```

结果是这样的（连字都懒得打了，直接截图……）：

```
>>> myFPtree.disp()
   Null Set   1
     x   1
       s   1
         r   1
     z   5
       x   3
         y   3
           s   2
             t   2
           r   1
             t   1
       r   1
```

得到的FP树也和图5中的一样。

4.3 从一棵FP树种挖掘频繁项集

到现在为止大部分比较困难的工作已经处理完了。有了FP树之后，就可以抽取频繁项集了。这里的思路与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。

从FP树中抽取频繁项集的三个基本步骤如下：

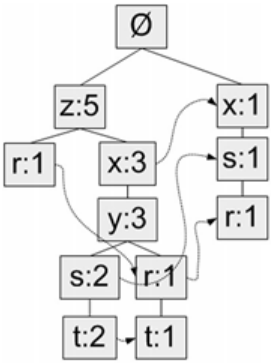
1. 从FP树中获得条件模式基；
2. 利用条件模式基，构建一个条件FP树；
3. 迭代重复步骤1步骤2，直到树包含一个元素项为止。

1 抽取条件模式基

（什么鬼……）

首先从头指针表中的每个频繁元素项开始，对每个元素项，获得其对应的条件模式基（conditional pattern base）。条件模式基是以所查找元素项为结尾的路径集合。每一条路径其实都是一条前缀路径（**prefix path**）。简而言之，一条前缀路径是介于所查找元素项与树根节点之间的所有内容。

将图5重新贴在这里：



则每一个频繁元素项的所有前缀路径（条件模式基）为：

频繁项	前缀路径
z	{}: 5
r	{x, s}: 1, {z, x, y}: 1, {z}: 1
x	{z}: 3, {}: 1
y	{z, x}: 3
s	{z, x, y}: 2, {x}: 1
t	{z, x, y, s}: 2, {z, x, y, r}: 1

发现规律了吗，z存在于路径{z}中，因此前缀路径为空，另添加一项该路径中z节点的计数值5构成其条件模式基；r存在于路径{r, z}、{r, y, x, z}、{r, s, x}中，分别获得前缀路径{z}、{y, x, z}、{s, x}，另添加对应路径中r节点的计数值（均为1）构成r的条件模式基；以此类推。

前缀路径将在下一步中用于构建条件FP树，暂时先不考虑。如何发现某个频繁元素项的所在的路径？利用先前创建的头指针表和FP树中的相似元素节点指针，我们已经有了每个元素对应的单链表，因而可以直接获取。

下面的程序给出了创建前缀路径的代码：

1 主函数：findPrefixPath

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

该函数代码用于为给定元素项生成一个条件模式基（前缀路径），这通过访问树中所有包含给定元素项的节点来完成。参数basePet表示输入的频繁项，treeNode为当前FP树种对应的第一个节点（可在函数外部通过headerTable[basePat][1]获取）。函数返回值即为条件模式基condPats，用一个字典表示，键为前缀路径，值为计数值。

2 辅助函数：ascendTree

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

这个函数直接修改prefixPath的值，将当前节点leafNode添加到prefixPath的末尾，然后递归添加其父节点。最终结果，prefixPath就是一条从treeNode（包括treeNode）到根节点（不包括根节点）的路径。在主函数findPrefixPath()中再取prefixPath[1:]，即为treeNode的前缀路径。

测试代码：

```
>>> fpGrowth.findPrefixPath('x', myHeaderTab['x'][1])
>>> fpGrowth.findPrefixPath('z', myHeaderTab['z'][1])
>>> fpGrowth.findPrefixPath('r', myHeaderTab['r'][1])
```

2 创建条件FP树

（又是什么鬼……）

对于每一个频繁项，都要创建一棵条件FP树。可以使用刚才发现的条件模式基作为输入数据，并通过相同的建树代码来构建这些树。例如，对于r，即以“{x, s}: 1, {z, x, y}: 1, {z}: 1”为输入，调用函数createTree()获得r的条件FP树；对于t，输入是对应的条件模式基“{z, x, y, s}: 2, {z, x, y, r}: 1”。

代码（直接调用createTree()函数）：

```
condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
myCondTree, myHead = createTree(condPattBases, minSup)
```

示例：t的条件FP树

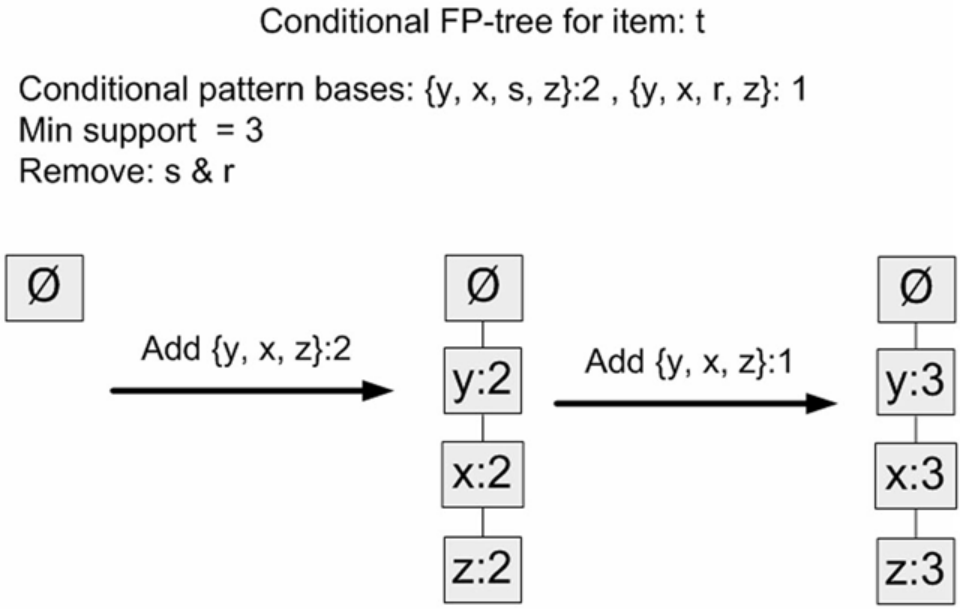


图8 t的条件FP树的创建过程

在图8中，注意到元素项s以及r是条件模式基的一部分，但是它们并不属于条件FP树。因为在当前的输入中，s和r不满足最小支持度的条件。

3 递归查找频繁项集

有了FP树和条件FP树，我们就可以在前两步的基础上递归得查找频繁项集。

递归的过程是这样的：

> 输入：我们有当前数据集的FP树（inTree，headerTable）
> 1. 初始化一个空列表preFix表示前缀
> 2. 初始化一个空列表freqItemList接收生成的频繁项集（作为输出）
> 3. 对headerTable中的每个元素basePat（按计数值由小到大），递归：
> 3.1 记basePat + preFix为当前频繁项集newFreqSet
> 3.2 将newFreqSet添加到freqItemList中
> 3.3 计算t的条件FP树（myCondTree、myHead）
> 3.4 当条件FP树不为空时，继续下一步；否则退出递归
> 3.4 以myCondTree、myHead为新的输入，以newFreqSet为新的preFix，外加freqItemList，递归这个过程

函数如下：

```
def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: p[1])]
    for basePat in bigL:
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        freqItemList.append(newFreqSet)
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        myCondTree, myHead = createTree(condPattBases, minSup)

        if myHead != None:
            # 用于测试
            print 'conditional tree for:', newFreqSet
            myCondTree.disp()
```

```
14
15                mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)
```

输入参数：

- inTree和headerTable是由createTree()函数生成的数据集的FP树
- minSup表示最小支持度
- preFix请传入一个空集合（set([])），将在函数中用于保存当前前缀
- freqItemList请传入一个空列表（[]），将用来储存生成的频繁项集

测试代码：

```
>>> freqItems = []
>>> fpGrowth.mineTree(myFPtree, myHeaderTab, 3, set([]), freqItems)
>>> freqItems
```

```
[set(['y']), set(['y', 'x']), set(['y', 'z']), set(['y', 'x', 'z']), set(['s']), set(['x', 's']),
set(['t']), set(['z', 't']), set(['x', 'z', 't']), set(['y', 'x', 'z', 't']), set(['y', 'z', 't']), set(['x',
't']), set(['y', 'x', 't']), set(['y', 't']), set(['r']), set(['x']), set(['x', 'z']), set(['z'])]
```

想这一段代码解释清楚比较难，因为中间涉及到很多递归。直接举例说明，我们在这里分解输入myFPtree和myHeaderTab后，“`for basePat in bigL:`”一行当basePat为**‘t’**时的过程：

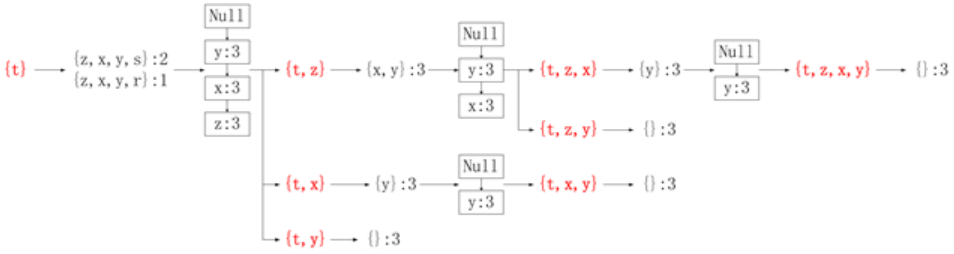


图9 mineTree函数解构图（basePat = ‘t’）

图中红色加粗的部分即实际添加到freqItemList中的频繁项集。

4 封装

至此，完整的FP-growth算法已经可以运行。封装整个过程如下：

```
def fpGrowth(dataSet, minSup=3):
    initSet = createInitSet(dataSet)
    myFPtree, myHeaderTab = createTree(initSet, minSup)
    freqItems = []
    mineTree(myFPtree, myHeaderTab, minSup, set([]), freqItems)
    return freqItems
```

测试代码：

```
>>> import fpGrowth
>>> dataSet = fpGrowth.loadSimpDat()
>>> freqItems = fpGrowth.fpGrowth(dataSet)
>>> freqItems
```

和之前的输出相同。

5 总结

FP-growth算法是一种用于发现数据集中频繁模式的有效方法。FP-growth算法利用Apriori原则，执行更快。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FP-growth算法执行更快。在FP-growth算法中，数据集存储在一个称为FP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及构建条件FP树来发现频繁项集。该过程不断以更多元素作为条件重复进行，直到FP树只包含一个元素为止。

FP-growth算法还有一个map-reduce版本的实现，它也很不错，可以扩展到多台机器上运行。Google使用该算法通过遍历大量文本来发现频繁共现词，其做法和我们刚才介绍的例子非常类似（参见扩展阅读：FP-growth算法）。

回到顶部

5. 示例：从新闻网站点击流中挖掘新闻报道

书中的这两章有不少精彩的示例，这里只选取比较有代表性的一个——从新闻网站点击流中挖掘热门新闻报道。这是一个很大的数据集，有将近100万条记录（参见扩展阅读：kosarak）。在源数据集合保存在文件kosarak.dat中。该文件中的每一行包含某个用户浏览过的新闻报道。新闻报道被编码成整数，我们可以使用Apriori或FP-growth算法挖掘其中的频繁项集，查看那些新闻ID被用户大量观看到。

首先，将数据集导入到列表：

```
>>> parsedDat = [line.split() for line in open('kosarak.dat').readlines()]
```

接下来需要对初始集合格式化：

```
>>> import fpGrowth
>>> initSet = fpGrowth.createInitSet(parsedDat)
```

然后构建FP树，并从中寻找那些至少被10万人浏览过的新闻报道。

```
>>> myFPtree, myHeaderTab = fpGrowth.createTree(initSet, 100000)
```

下面创建一个空列表来保存这些频繁项集：

```
>>> myFreqList = []
>>> fpGrowth.mineTree(myFPtree, myHeaderTab, 100000, set([]), myFreqList)
```

接下来看下有多少新闻报道或报道集合曾经被10万或者更多的人浏览过：

```
>>> len(myFreqList)
```

```
9
```

总共有9个。下面看看都是那些：

```
>>> myFreqList
```

```
[set(['1']), set(['1', '6']), set(['3']), set(['11', '3']), set(['11', '3', '6']), set(['3', '6']), set(['11']), set(['11', '6']), set(['6'])]
```

回到顶部

扩展阅读

在看这两章的过程中和之后又看到的一些相关的东西：

- 尿布与啤酒：http://web.onetel.net.uk/~hibou/Beer and Nappies.html
- Association Analysis: Basic Concepts and Algorithms[PDF]
- FP-growth算法：H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Chang, "PFP: Parallel FP-Growth for Query Recommendation," RecSys 2008, Proceedings of the 2008 ACM Conference on Recommender Systems; http://portal.acm.org/citation.cfm?id=1454027.
- kosarak：Hungarian online news portal clickstream retrieved July 11, 2011; from Frequent Itemset Mining Dataset Repository, http://fimi.ua.ac.be/data/, donated by Ferenc Bodon.

注：

- 获取kosarak.dat文件，请参考文章目录：《机器学习实战》学习笔记。
- 如果需要在Python源代码中插入Unicode字符（汉字）注释最好在文件第一行添加"# coding=utf-8"。

作者：王哲

标签：机器学习实战，Python，无监督学习，关联分析，关联规则学习，Apriori 算法，FP-growth 算法

好文要顶 关注我 收藏该文

qwertWZ
关注 - 16
粉丝 - 9
+加关注

0 0

(请您对文章做出评价)

« 上一篇：An ffmpeg and SDL Tutorial
» 下一篇：《机器学习实战》学习笔记

posted @ 2015-05-17 23:25 qwertWZ 阅读(1765) 评论(1) 编辑 收藏

评论列表

#1楼 2015-07-10 20:26 Mr.Quan

请问，Apriori算法可以用于空气质量预测吗

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新知识库文章:
· Git协作流程
· 企业计算的终结
· 软件开发的核心
· Linux概念架构的理解
· 从涂鸦到发布——理解API的设计过程
» 更多知识库文章...