菊安酱的机器学习第10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：2019-5-13

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

期数	算法
第1期	k-近邻算法
第2期	决策树
第3期	朴素贝叶斯
第4期	Logistic回归
第5期	支持向量机
第6期	AdaBoost 算法
第7期	线性回归
第8期	树回归
第9期	K-均值聚类算法
第10期	Apriori 算法
第11期	FP-growth 算法
第12期	降维算法PCA&SVD

Apriori算法

一、关联分析概述

大型超市有海量的交易数据，作为精明的商家肯定不会放弃对这些海量数据的应用，他们希望通过对这些交易数据的分析，了解顾客的购买行为。我们可以通过聚类算法寻找购买相似物品的人群，从而为特定人群提供更具个性化的服务。但是对于超市来讲，更有价值的是找出商品之间的隐藏关联，从而可以用于商品定价、市场促销、存货管理等一系列环节，来增加营业收入。那如何从这种海量而又繁杂的交易数据中找出商品之间的关联呢？当然，你可以使用穷举法，但是这是一种十分耗时且计算代价高的笨方法，所以需要用更智能的方法在合理的时间范围内找到答案。因此，关联分析就此诞生了~

1. 关联分析

关联分析（association analysis）是一种在大规模数据集中寻找有趣关系的非监督学习算法。这种关系可以有两种形式：频繁项集或者关联规则。**频繁项集**（frequent item sets）是经常出现在一块的物品的集合，**关联规则**（association rules）暗示两种物品之间可能存在很强的关系。

关联分析的一个典型例子是购物篮分析。下面我们一起来看一个简单例子。

交易号码	商品
001	豆奶，莴苣
002	莴苣，尿布，啤酒，甜菜
003	豆奶，尿布，啤酒，橙汁
004	莴苣，豆奶，尿布，啤酒
005	莴苣，豆奶，尿布，橙汁

交易号码代表交易流水号，商品代表一个顾客一次购买的全部商品。

在这里需要明确几个定义：

- **事务**：每一条交易称为一个事务。例如，在这个例子中包含了5个事务。
- **项**：交易的每一个物品称为一个项，例如豆奶、尿布等。
- **项集**：包含零个或者多个项的集合叫做项集，例如 {豆奶，莴苣}。
- **k-项集**：包含k个项的项集叫做k-项集。例如 {豆奶} 叫做1-项集，{豆奶，尿布，啤酒} 叫做3-项集。
- **前件**和**后件**：对于规则{尿布}→{啤酒}，{尿布} 叫做前件，{啤酒} 叫做后件。

啤酒与尿布的故事

关联分析中最有名的例子是"啤酒与尿布"。你打开百度搜索一下，就会发现很多人都在对“啤酒与尿布”的故事津津乐道，可以说100个人就有100个版本的 “啤酒与尿布”的故事。
故事的时间跨度从上个世纪80年代到本世纪初，甚至连故事的主角和地点都会发生变化——从美国跨越到欧洲。认真地查了一下资料，我们发现沃尔玛的 “啤酒与尿布”案例是正式刊登在1998年的《哈佛商业评论》上面的，这应该算是目前发现的最权威报道。
“啤酒与尿布”的故事产生于20世纪90年代的美国沃尔玛超市中，沃尔玛的超市管理人员分析销售数据时发现了一个令人难于理解的现象：在某些特定的情况下，“啤酒”与“尿布”两件看上去毫无关系的商品会经常出现在同一个购物篮中，这种独特的销售现象引起了管理人员的注意，经过后续调查发现，这种现象出现在年轻的父亲身上。在美国有婴儿的家庭中，一般是母亲在家中照看婴儿，年轻的父亲前去超市购买尿布。父亲在购买尿布的同时，往往会顺便为自己购买啤酒，这样就会出现啤酒与尿布这两件看上去不相干的商品经常会出现在同一个购物篮的现象。如果这个年轻的父亲在卖场只能买到两件商品之一，则他很有可能会放弃购物而到另一家商店，直到可以一次同时买到啤酒与尿布为止。
沃尔玛发现了这一独特的现象，开始在卖场尝试将啤酒与尿布摆放在相同的区域，让年轻的父亲可以同时找到这两件商品，并很快地完成购物；而沃尔玛超市也可以让这些客户一次购买两件商品、而不是一件，从而获得了很好的商品销售收入，这就是“啤酒与尿布”故事的由来。

2. 频繁项集的评估标准

频繁项集是经常出现在一块的物品的集合，那么问题就来了：第一，当数据量非常大的时候，我们无法凭肉眼找出经常出现在一起的物品，这就催生了关联规则挖掘算法，比如 Apriori、PrefixSpan、CBA 等。第二是我们缺乏一个频繁项集的标准。比如10条记录，里面A和B同时出现了三次，那么我们能不能说A和B一起构成频繁项集呢？因此我们需要一个评估频繁项集的标准。

常用的频繁项集的评估标准有支持度,置信度和提升度三个。

2.1 支持度

支持度就是几个关联的数据在数据集中出现的次数占总数据集的比重。或者说几个数据关联出现的概率。如果我们有两个想分析关联性的数据X和Y，则对应的支持度为:

$$Support(X,Y) = P(XY) = \frac{number(XY)}{num(AllSamples)}$$

比如上面例子中，在5条交易记录中{尿布, 啤酒}总共出现了3次，所以

$$Support(\text{尿布，啤酒}) = \frac{3}{5} = 0.6$$

以此类推，如果我们有三个想分析关联性的数据X，Y和Z，则对应的支持度为:

$$Support(X,Y,Z) = P(XYZ) = \frac{number(XYZ)}{num(AllSamples)}$$

一般来说，支持度高的数据不一定构成频繁项集，但是支持度太低的数据肯定不构成频繁项集。 另外，支持度是针对项集来说的，因此，可以定义一个最小支持度，而只保留满足最小支持度的项集，起到一个项集过滤的作用。

交易号码	商品
001	豆奶，莴苣
002	莴苣，尿布，啤酒，甜菜
003	豆奶，尿布，啤酒，橙汁
004	莴苣，豆奶，尿布，啤酒
005	莴苣，豆奶，尿布，橙汁

2.2 置信度

置信度体现了一个数据出现后，另一个数据出现的概率，或者说数据的条件概率。如果我们有两个想分析关联性的数据X和Y，X对Y的置信度为

$$Confidence(X \rightarrow Y) = P(Y|X) = P(XY)/P(X)$$

比如上面例子中，莴苣对豆奶的置信度=豆奶和莴苣同时出现的概率/莴苣出现的概率，则有

$$P(\text{豆奶} * \text{莴苣}) = \frac{3}{5} = 0.6$$

$$P(\text{莴苣}) = \frac{4}{5} = 0.8$$

$$Confidence(\text{莴苣} \rightarrow \text{豆奶}) = \frac{P(\text{豆奶} * \text{莴苣})}{P(\text{莴苣})} = \frac{0.6}{0.8} = 0.75$$

也可以以此类推到多个数据的关联置信度，比如对于三个数据X，Y，Z,则Y和Z对于X的置信度为：

$$Confidence(YZ \rightarrow X) = P(X|YZ) = P(XYZ)/P(YZ)$$

为什么使用支持度和置信度？支持度是一种重要度量，因为支持度很低的规则可能只是偶然出现。从商务角度来看，低支持度的规则多半也是无意义的，因为对顾客很少同时购买的商品进行促销可能并无益处。因此，支持度通常用来删去那些无意义的规则。此外，支持度还具有一种期望的性质，可以用于关联规则的有效发现。

另一方面，置信度度量通过规则进行推理具有可靠性。对于给定的规则*X*→*Y*，置信度越高，*Y*在包含*X*的事务中出现的可能性就越大。置信度也可以估计*Y*在给定*X*下的条件概率。

同时，应当小心解释关联分析的结果。由关联规则作出的推论并不必然蕴涵因果关系。它只表示规则前件和后件同时出现的一种概率。

2.3 提升度

接下来我们来讨论支持度置信度框架的局限性，现有的关联规则的挖掘算法需要使用支持度和置信度来除去没有意义的模式。支持度的缺点在于许多潜在的有意义的模式由于包含支持度小的项而被删去，置信度的缺点更加微妙，用下面的例子最适于说明。

	咖啡	不喝咖啡	
茶	150	50	200
不喝茶	650	150	800
	800	200	1000

可以使用表中给出的信息来评估关联规则{茶}→{咖啡}。猛一看，似乎喜欢喝茶的人也喜欢喝咖啡，因为该规则的支持度（15%）和置信度（75%）都相当的高。这个推论也许是可以接受的，但是所有的人中，不管他是否喝茶，喝咖啡的人的比例为80%，而喝咖啡的饮茶者却只占75%。也就是说，一个人如果喝茶，则他喝咖啡的可能性由80%减到了75%。因此，尽管规则{茶}→{咖啡}有很高的置信度，但是它却是一个误导。所以说，置信度的缺陷在于该度量忽略了规则后件中项集的支持度。

茶与咖啡的例子表明，由于置信度度量忽略了规则后件中出现的项集的支持度，高置信度的规则有时可能出现误导。解决这个问题的一种方法是使用称作提升度（lift）的度量：

$$lift(X \to Y) = \frac{c(X \to Y)}{s(Y)}$$

提升度表示$X \to Y$的置信度与后件Y的支持度之比，即:

$$Lift(X \to Y) = \frac{P(Y|X)}{P(Y)} = \frac{Confidence(X \to Y)}{P(Y)}$$

提升度体先了X和Y之间的关联关系, 提升度大于1则$X \to Y$是有效的强关联规则， 提升度小于等于1则$X \to Y$是无效的强关联规则 。一个特殊的情况，如果X和Y独立，则有$Lift(X \to Y) = 1$，因为此时$P(Y|X) = P(Y)$。

那我们计算一下{茶}→{咖啡}的提升度：

$$Lift(\text{茶} \to \text{咖啡}) = \frac{P(\text{咖啡}|\text{茶})}{P(\text{咖啡})} = \frac{0.75}{0.8} \approx 0.94$$

一般来说，要选择一个数据集合中的频繁数据集，则需要自定义评估标准。最常用的评估标准是用自定义的支持度，或者是自定义支持度和置信度的一个组合。

3. 关联规则发现

给定事务的集合*T*，关联规则发现是指找出支持度大于等于$minsup$并且置信度大于等于 $minconf$ 的所有规则，其中 $minsup$ 和 $minconf$ 是对应的支持度和置信度阈值。挖掘关联规则的一种原始方法是：计算每个可能规则的支持度和置信度。但是这种方法的代价很高，令人望而却步，因为可以从数据集提取的规则的数目达指数级。更具体地说，从包含d个项的数据集提取的可能规则的总数为:

$$R = 3^d - 2^{d+1} + 1$$

【公式证明过程】可参考：https://blog.csdn.net/wxbmelisky/article/details/48268833

对于我们前面的小例子，里面一共有6中商品，提取的可能规则数为：$3^6 - 2^7 + 1 = 602$，也就是说对于只有6种商品的小数据集都需要计算602条规则的支持度和置信度。使用*minsup*= 20%和*minconf*=50%，80%以上的规则将被丢弃，使得大部分计算是无用的开销。为了避免进行不必要的计算，事先对规则剪枝，而无须计算它们的支持度和置信度的值将是有益的。 因此，大多数关联规则挖掘算法通常采用的一种策略是，将关联规则挖掘任务分解为如下两个主要的子任务。

- 频繁项集产生：

  其目标是发现满足最小支持度阈值的所有项集，这些项集称作频繁项集（frequent itemset）。

- 规则的产生：

  其目标是从上一步发现的频繁项集中提取所有高置信度的规则，这些规则称作强规则（strong rule）。

通常，频繁项集产生所需的计算开销远大于产生规则所需的计算开销。那有没有办法可以减少这种无用的计算呢？有。下面这两种方法可以降低产生频繁项集的计算复杂度：

（1）减少候选项集的数目M。

（2）减少比较次数。替代将每个候选项集与每个事务相匹配，可以使用更高级的数据结构，或者存储候选项集或者压缩数据集，来减少比较次数。

这些策略将在Apriori算法基本思想中进行讨论。

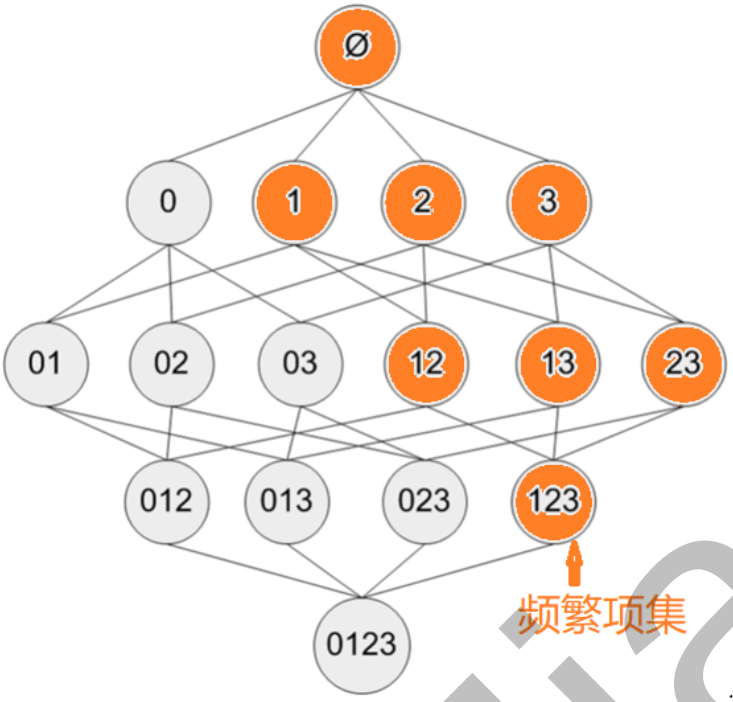
二、Apriori算法原理

先验原理 如果一个项集是频繁的，则它的所有子集一定也是频繁的。

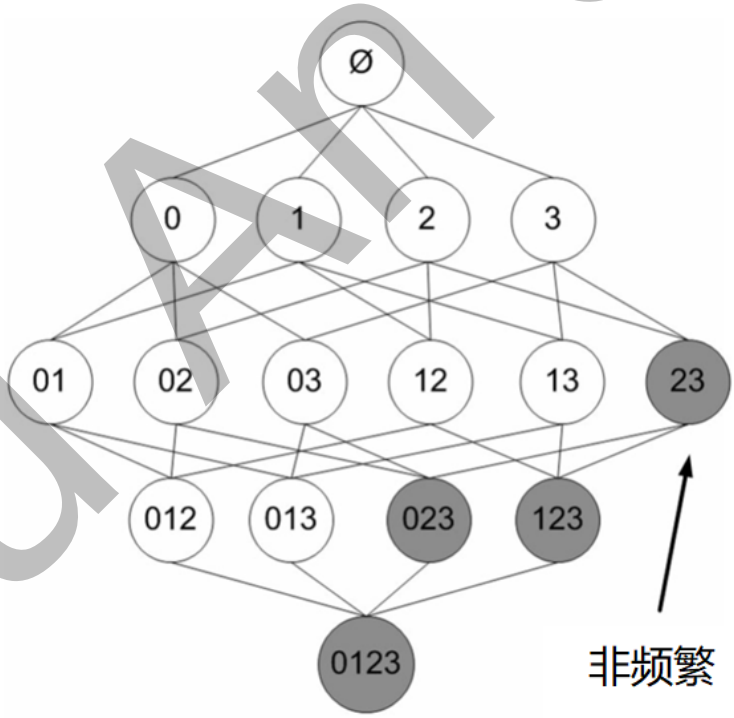
假设我们在经营一家商品种类并不多的杂货店，我们对那些经常在一起被购买的商品非常感兴趣。我们只有4种商品：商品0，商品1，商品2和商品3。那么所有可能被一起购买的商品组合都有哪些？这些商品组合可能只有一种商品，比如商品0，也可能包括两种、三种或者所有四种商品。我们并不关心某人买了两件商品0以及四件商品2的情况，我们只关心他购买了一种或多种商品。

下图显示了物品之间所有可能的组合。为了让该图更容易懂，图中使用物品的编号0来取代物品0本身。另外，图中从上往下的第一个集合是ϕ，表示空集或不包含任何物品的集合。物品集合之间的连线表明两个或者更多集合可以组合形成一个更大的集合。

根据先验原理，假如 {1,2,3} 是频繁项集，那么它的所有子集（下图中橘色项集）一定也是频繁的。



这个先验原理直观上并没有什么帮助，但是反过来看就有用了，也就是说**如果一个项集是非频繁项集，那么它的所有超集也是非频繁的**（如下图所示）。

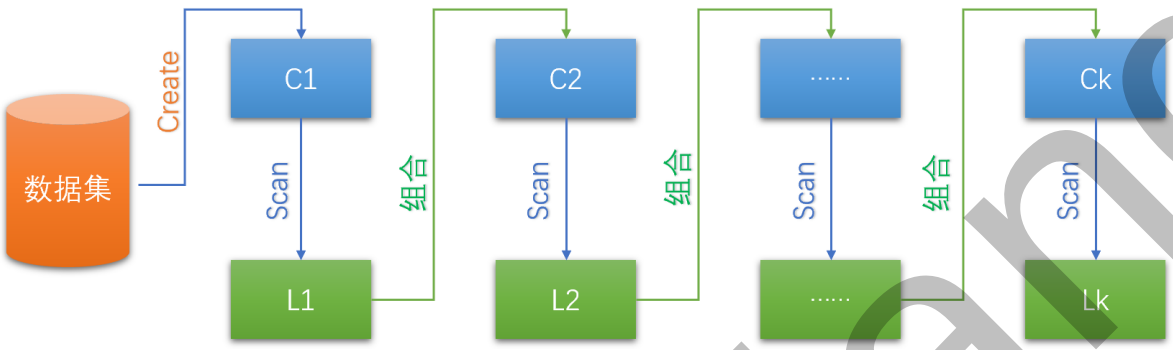


上图中，已知阴影项集 {2,3} 是非频繁的。利用这个知识，我们就知道项集{0,2,3} ，{1,2,3}以及{0,1,2,3}也是非频繁的。这也就是说，一旦计算出了{2,3}的支持度，知道它是非频繁的之后，就不需要再计算{0,2,3}、{1,2,3}和{0,1,2,3}的支持度，因为我们知道这些集合不会满足我们的要求。使用该原理就可以避免项集数目的指数增长，从而在合理时间内计算出频繁项集。

三、使用Apriori算法来发现频繁项集

关联分析的目的包括两项：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。

Apriori 算法过程



C1，C2，...，Ck分别表示1-项集，2-项集，...，k-项集；

L1，L2，...，Lk分别表示有k个数据项的频繁项集。

Scan表示数据集扫描函数。该函数起到的作用是支持度过滤，满足最小支持度的项集才留下，不满足最小支持度的项集直接舍掉。

下面我们用python代码来实现这一过程~

1. 生成候选项集

在使用python来对整个程序编码之前，需要创建一些辅助函数。

数据集扫描的伪代码如下：

```
对数据集中的每条交易记录transaction
对每个候选项集can:
    检查can是否是transaction的子集:
    如果是，增加can的计数
对每个候选项集:
    如果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
```

首先，我们创建一个简单数据集，来帮助我们建模

函数1：创建一个用于测试的简单数据集

```
#导入包
import pandas as pd
import numpy as np

#创建生成数据集的函数
def loadDataSet():
    dataSet = [[1,3,4],[2,3,5],[1,2,3,5],[2,5]]
    return dataSet
```

运行函数，结果如下：

```
dataSet = loadDataSet()
dataSet
```

函数2：构建第一个候选集合C1。

由于算法一开始是从输入数据集中提取候选项集列表，所以这里需要一个特殊的函数来处理——frozenset类型。frozenset是指被“冰冻”的集合，也就是用户不能修改他们。这里必须要用frozenset而非set类型，是因为后面我们要将这些集合作为字典键值使用。

```
s = set('hello')          #生成集合，集合会自动去重
s.add('a')                #集合允许修改
f = frozenset('hello')    #创建frozenset类型，也有去重功能
f.add('a')                #frozdenset类型不允许修改
```

该函数流程是：首先创建一个空列表，用来储存所有不重复的项值。接下来遍历数据集中所有的交易记录，对每一条交易记录，遍历记录中的每一个项。如果该项没有在C1中出现过，那么就把它添加到C1中，这里需要注意的是，并非简单地添加物品项，而是添加只包含该物品项的一个集合（此处用集合或者列表都可以）。循环完毕后，对整个C1进行排序并将其中每个单元素集合映射到frozenset()，最后返回frozenset的列表。

```
"""
函数功能：生成第一个候选集合C1
参数说明：
    dataSet：原始数据集
返回：
    frozenset形式的候选集合C1
"""
def createC1(dataSet):
    C1 = []
    for transaction in dataSet:  #数据集中每一条事务
        for item in transaction: #事务中的每一个项集
            if not {item} in C1: #判断每一个项集在不在候选集合中
                C1.append({item})#添加不在候选集中的那些项集
    C1.sort() #排序
    return list(map(frozenset, C1))
```

运行函数，查看结果：

```
C1=createC1(dataSet)
C1
```

在上面这个函数中有两个点需要注意：

- .sort()函数起到排序的作用，默认从小到大排序。

```
#对于列表套列表这样的数据格式
#sort排序规则是先以第一个元素进行排序，若第一个元素相同则以第二个元素进行排序，以此类推
a=[[2,6,1],[2,4,2],[1,5,4,3],[2]]
a.sort()
a
```

- map()是python内置的高阶函数，它接收一个函数f 和一个列表list，它的功能是把函数 f 依次作用到list 的每个元素上，得到一个新的list然后返回。

```
l = [1, 2, 3]
def square(x):
    return x ** 2
ml = map(square, l)  #返回的是map对象 ，非显示对象
list(ml)             #若要显示结果，需要用list将其还原，print无法起作用
```

函数3：生成满足最小支持度的频繁项集L1。

C1是大小为1的所有候选项集的集合，Apriori算法首先构建集合C1，然后扫描数据集来判断这些只有一个元素的项集是否满足最小支持度的要求。那些满足最低要求的项集构成集合L1。

```
"""
函数功能：生成满足最小支持度的频繁项集L1
参数说明：
    D:原始数据集
    Ck:候选项集
    minSupport:最小支持度
返回：
    retList：频繁项集
    supportData：所有候选项集的支持度
"""
def scanD(D, Ck, minSupport):
    ssCnt = {}    #字典的形式存放项集及其出现次数，{项集:次数}
    for tid in D:  #每一条事务集
        for can in Ck: #候选集中每一个项集
            if can.issubset(tid):  #判断can是否是tid的子集，返回的是布尔型数据
                ssCnt[can] = ssCnt.get(can, 0) + 1 #如果是子集则字典值加1
    numItems = float(len(D)) #数据集长度
    retList= []              #频繁项集
    supportData = {}         #候选集项Ck的支持度字典(key:候选项，  value:支持度)
    for key in ssCnt:
        support = ssCnt[key] / numItems #计算每一项集的支持度
        supportData[key] = support  #将所有项集及其支持度以字典的形式放入supportData
        if support >= minSupport: #如果支持度满足最小支持度要求（≥最小支持度）
```

```
            retList.append(key)    #把满足支持度的项集放入频繁项集retList
    return retList, supportData    #返回频繁项集和所有项集及其支持度
```

将数据集dataSet带入函数，查看运行结果：

```
L1, supportData = scanD(dataSet, C1, 0.5)
L1
supportData
```

```
In [11]: L1, supportData = scanD(dataSet, C1, 0.5)

In [12]: L1

Out[12]: [frozenset({1}), frozenset({2}), frozenset({3}), frozenset({5})]

In [13]: supportData

Out[13]: {frozenset({1}): 0.5,
          frozenset({2}): 0.75,
          frozenset({3}): 0.75,
          frozenset({5}): 0.75,
          frozenset({4}): 0.25}
```

从运行结果中可以看出，设定最小支持度为0.5时，frozenset({4})支持度为0.25<0.5，就被舍弃了，没有放入到L1中。

2. 项集迭代函数

根据前述Apriori算法工作流程，我们已经有了频繁一项集的生成函数，但是我们还缺少由频繁k项集生成候选k+1项集的函数，下面我们就来构建这个函数。构建这个函数的思路是：将不同的频繁k项集两两合并生成候选k+1项集。在这个函数的构造过程中，有两点需要注意：

1. **Ck中不会放重复的项集**，这样会增加计算量。也就是说在生成候选k+1项集之后需要判断，这个候选k-1项集是否已经存在Ck中，如果没有的话再把它放入Ck，反之则舍弃。
2. 判断两两合并之后得到的项集是不是**长度为k+1**,如果是则放入Ck，反之则舍弃。

运算符 | 在python中表示集合的合并，frozenset集合也可以执行合并操作，并且合并结果仍为frozenset类型。

```
L1[1]|L1[2]
```

```
In [22]: L1[1]|L1[2]

Out[22]: frozenset({2, 3})
```

```
"""
函数功能：由频繁k项集生成候选k+1项集
参数说明：
```

```
    Lk：频繁k项集
返回：
    Ck：候选k+1项集
"""
def aprioriGen(Lk):
    Ck = []
    lenLk = len(Lk)
    for i in range(lenLk):
        for j in range(i+1, lenLk):
            L1 = Lk[i]
            L2 = Lk[j]
            C = Lk[i] | Lk[j]
            if not C in Ck and (len(C)==len(Lk[0])+1):
                Ck.append(Lk[i] | Lk[j])
    return Ck
```

由前面运行结果得知L1中共有4个频繁1项集，那两两组合的话，可以得到$C_4^2 = \frac{4*3}{2} = 6$个候选2项集。现在我们用aprioriGen函数由频繁1项集生成候选2项集，查看运行结果：

```
aprioriGen(L1)
```

[25]:
```
C2 = aprioriGen(L1,2)
C2
```

[25]:
```
[frozenset({1, 3}),
 frozenset({1, 2}),
 frozenset({1, 5}),
 frozenset({2, 3}),
 frozenset({3, 5}),
 frozenset({2, 5})]
```

接下来可以看一下由频繁2项集生成候选3项集，并由候选3项集生成频繁3项集的过程。在这个过程中，首先用scanD函数根据候选2项集C2生成频繁2项集L2，再用aprioriGen函数生成候选3项集C3，之后再次使用scanD函数生成频繁3项集L3。

```
L2, supportData= scanD(dataSet, C2, 0.5)
C3 = aprioriGen(L2)
L3, supportData= scanD(dataSet, C3, 0.5)
```

[22]:
```
L2, supportData= scanD(dataSet, C2, 0.5)
L2
```

[22]:
```
[frozenset({1, 3}), frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5})]
```

[23]:
```
C3 = aprioriGen(L2)
C3
```

[23]:
```
[frozenset({1, 2, 3}), frozenset({1, 3, 5}), frozenset({2, 3, 5})]
```

[24]:
```
L3, supportData= scanD(dataSet, C3, 0.5)
L3
```

[24]:
```
[frozenset({2, 3, 5})]
```

函数2：根据数据集和支持度，返回所有的频繁项集，以及所有项集的支持度。

```
"""
Apriori主函数：根据输入数据集，返回所有频繁项集和所有项集及其支持度
参数说明:
    D：原始数据集
    minSupport：最小支持度
返回:
    L：所有频繁项集
    supportData：所有项集及其支持度
"""
def apriori(D, minSupport = 0.5):
    C1 = createC1(D) #生成候选1项集
    L1, supportData = scanD(D, C1, minSupport) #生成频繁1项集和支持度列表
    L = [L1] #将频繁1项集放入L中
    k=2      #项集数，初始值设为2
    while (len(L[-1]) > 0): #如果L中最后项集中的元素数不为0则持续循环
        Ck = aprioriGen(L[-1]) #生成候选k项集
        Lk, supK = scanD(D, Ck, minSupport) #根据候选k项集，生成频繁k项集和支持度列表
        supportData.update(supK) #更新支持度列表
        L.append(Lk) #更新频繁项集L
        k=k+1
    return L, supportData
```

运行函数，查看结果：

```
dataset = loadDataSet()
L, supportData = apriori(dataset, minSupport=0.5)
```

In [27]:
```
dataset = loadDataSet()
L, supportData = apriori(dataset, minSupport=0.5)
```

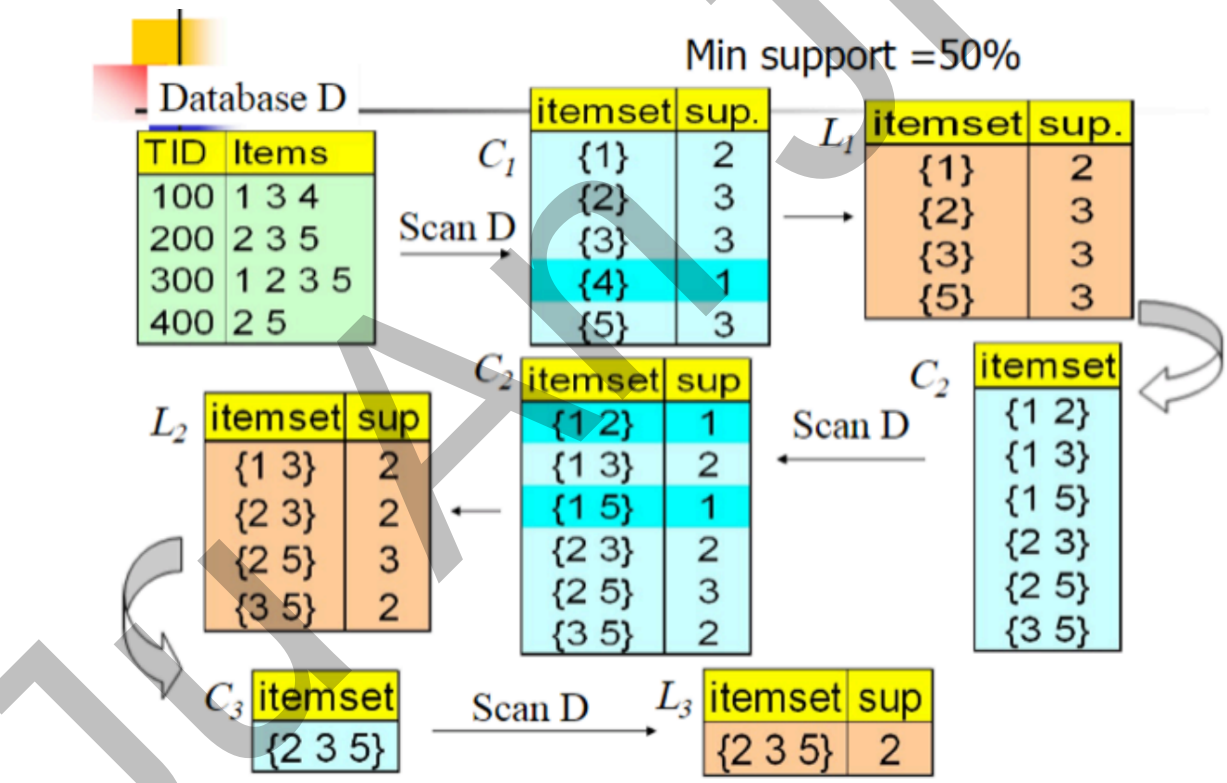
In [28]:
```
L
```

Out[28]:
```
[[frozenset({1}), frozenset({3}), frozenset({2}), frozenset({5})],
 [frozenset({1, 3}), frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5})],
 [frozenset({2, 3, 5})],
 []]
```

In [29]:
```
supportData
```

Out[29]:
```
{frozenset({1}): 0.5,
 frozenset({3}): 0.75,
 frozenset({4}): 0.25,
 frozenset({2}): 0.75,
 frozenset({5}): 0.75,
 frozenset({1, 3}): 0.5,
 frozenset({2, 3}): 0.5,
 frozenset({3, 5}): 0.5,
 frozenset({2, 5}): 0.75,
 frozenset({1, 2}): 0.25,
 frozenset({1, 5}): 0.25,
 frozenset({2, 3, 5}): 0.5}
```

其实上面这一系列函数实现的功能可以用下图表示：

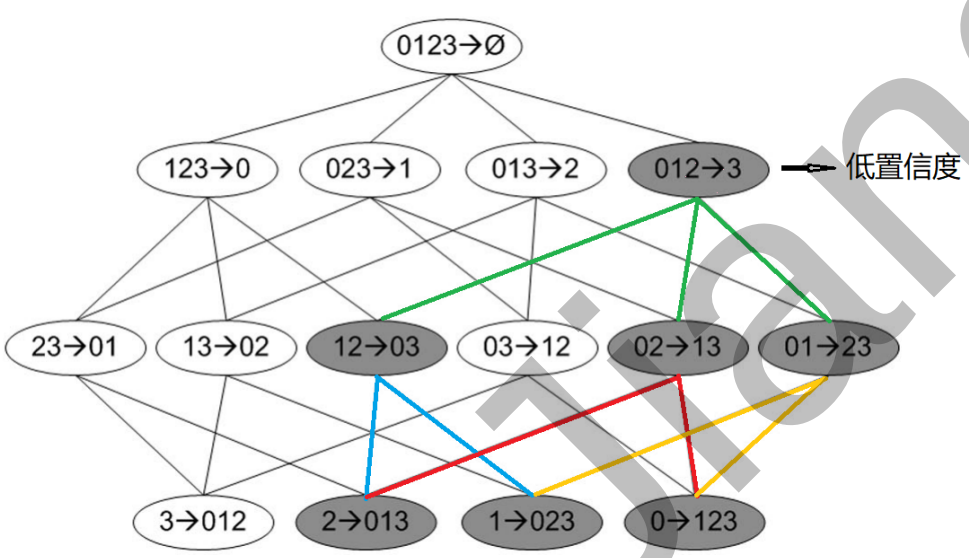


四、Apriori关联规则挖掘

1. 挖掘关联规则的流程

完成频繁项集的挖掘之后，我们就可以根据这些频繁项集来寻找关联规则了。对于频繁项集，上面给出了量化定义，即它满足最小支持度要求。那么，对于关联规则，我们也有类似的量化方法——置信度。对于一条规则P→H的置信度可以表示为：$support(H|P)/support(P)$ 。在python中，操作符|表示集合的并操作，数学上集合并的符号是∪。

类似于频繁项集的生成，每一个频繁项集也可以生成许多条关联规则。如下图，列出了频繁项集{0,1,2,3}可以生成的全部规则。**如果某一条规则不满足最小置信度要求，那么该规则的所有子集也不会满足最小置信度的要求**（如图中阴影部分所示）。我们可以利用这条性质来减少需要计算的规则数目，一旦发现某条规则不满足最小置信度的要求，那么这条规则的所有子集也就不用再去计算了。



频繁项集{0,1,2,3}的关联规则网格示意图

挖掘关联规则的流程为：首先从一个频繁项集开始，创建一个规则列表，其中规则右部只包含一个元素，然后计算这些规则的置信度。接下来合并所有剩余规则，创建一个新的规则列表，其中右部包含两个元素，计算这些规则的置信度……如此循环，遍历所有的规则。这种方法也被称为**分级法** 。

了解流程之后，我们尝试用python实现这一流程。

2. 关联规则的python实现

我们直接在Apriori()函数挖掘出来的频繁项集的基础上来进一步挖掘满足置信度的关联规则，首先再来回顾一下Apriori()函数返回的结果：

```
In [27]: dataset = loadDataSet()
         L, supportData = apriori(dataset, minSupport=0.5)

In [28]: L

Out[28]: [[frozenset({1}), frozenset({3}), frozenset({2}), frozenset({5})],
          [frozenset({1, 3}), frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5})],
          [frozenset({2, 3, 5})],
          []]

In [29]: supportData

Out[29]: {frozenset({1}): 0.5,
          frozenset({3}): 0.75,
          frozenset({4}): 0.25,
          frozenset({2}): 0.75,
          frozenset({5}): 0.75,
          frozenset({1, 3}): 0.5,
          frozenset({2, 3}): 0.5,
          frozenset({3, 5}): 0.5,
          frozenset({2, 5}): 0.75,
          frozenset({1, 2}): 0.25,
          frozenset({1, 5}): 0.25,
          frozenset({2, 3, 5}): 0.5}
```

其中，L是一个由频繁项集组成的list，而supportData则包含了所有项集及其支持度。对于L而言，第一个元素就是频繁一项集的list，第二个元素就是频繁二项集的list，以此类推。大家已经知道，置信度表示的是一个数据出现后另一个数据出现的概率（即条件概率），所以对于频繁一项集而言并无置信度的概念。频繁二项集的置信度相较于频繁多项集（三项集及以上）更为简单，因此在计算置信度时，我们对频繁二项集和频繁多项集分别计算。

首先考虑频繁二项集，返回的L中包含了所有的频繁项集，直接取出频繁二项集L[1]

```
L2 = L[1]
L2
```

```
[31]: L2=L[1]
      L2

[31]: [frozenset({1, 3}), frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5})]
```

然后考虑计算其中的每个频繁二项集的支持度，以frozenset({1, 3})为例。

```
freqSet = L2[0]
freqSet

#置信度的计算过程非常简单，借助集合运算以及Apriori的supportData计算结果，用频繁项集的支持度除以前驱项的支持度即可，例如计算1→3的置信度：
supportData[freqSet]/supportData[freqSet-frozenset({3})]

#此处用到了集合减法运算求差集
freqSet
freqSet - frozenset({3})
```

将上述过程封装成函数：

```
"""
```

```
函数功能：根据单个频繁项集生成满足最小置信度的关联规则
参数说明：
    freqSet：单个频繁项集
    H：可以出现在关联规则右部的元素列表
    supportData：所有项集及其支持度
    brl：强关联规则列表
    minConf：最小置信度
返回：
    prunedH：关联规则右部元素列表
"""
def calcConf(freqSet, H, supportData, brl, minConf=0.5):
    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
```

测试函数运行结果：

```
brl=[]
H1 = [frozenset([item]) for item in freqSet]
calcConf(freqSet, H1, supportData, brl, minConf=0.5)
```

[41]:
```
brl=[]
H1 = [frozenset([item]) for item in freqSet]
calcConf(freqSet, H1, supportData, brl, minConf=0.5)
```

```
frozenset({3}) --> frozenset({1}) conf: 0.6666666666666666
frozenset({1}) --> frozenset({3}) conf: 1.0
```

[41]:
```
[frozenset({1}), frozenset({3})]
```

频繁多项集生成函数

对于多项集，生成置信度规则的过程会更加复杂，在上述例子中，只有一个频繁三项集{2,3,5}，接下来考虑计算该项集所能生成的规则置信度。首先仍然是将该频繁项集赋值给一个单独的变量，然后将其内部的每个元素单独保存为一个frozenset。

```
freqSet = L[2][0]
freqSet

H2 = [frozenset([item]) for item in freqSet]
H2

#采用上述二项集置信度计算函数来计算前驱项只有一项的关联规则
calcConf(freqSet, H2, supportData, brl, minConf=0.5)

#然后用aprioriGen函数由一项集生成二项集
H3 = aprioriGen(H2)
```

```
H3

calcConf(freqSet, H3, supportData, brl, minConf=0.5)
```

将上述过程封装成函数

```
"""
函数功能：根据单个频繁项集生成满足最小置信度的关联规则
参数说明：
    freqSet：单个频繁项集
    H：频繁项集中的所有子项，可以放在规则右部的元素列表
    brl：存放关联规则的容器
"""
def rulesFromConseq(freqSet, H, supportData, brl, minConf=0.7):
    Hmp = True
    while Hmp:
        Hmp = False
        H = calcConf(freqSet, H, supportData, brl, minConf)
        H = aprioriGen(H)
        Hmp = not(H == [] or len(H[0]) == len(freqSet))
```

测试函数运行结果：

```
freqSet = L[2][0]
H2 = [frozenset([item]) for item in freqSet]
brl = []
rulesFromConseq(freqSet, H2, supportData, brl, minConf=0.5)
```

[49]:
```
brl = []
rulesFromConseq(freqSet, H2, supportData, brl, minConf=0.5)
```

```
frozenset({3, 5}) --> frozenset({2}) conf: 1.0
frozenset({2, 5}) --> frozenset({3}) conf: 0.6666666666666666
frozenset({2, 3}) --> frozenset({5}) conf: 1.0
frozenset({5}) --> frozenset({2, 3}) conf: 0.6666666666666666
frozenset({3}) --> frozenset({2, 5}) conf: 0.6666666666666666
frozenset({2}) --> frozenset({3, 5}) conf: 0.6666666666666666
```

生成关联规则的主函数

```
"""
函数功能：生成所有满足最小置信度的关联规则
参数说明：
    L：频繁项集
    supportData：所有项集及其支持度
    minConf：最小置信度
返回：
    bigRuleList：满足最小置信度的关联规则（即强关联规则）
"""
def generateRules(L, supportData, minConf=0.7):
    bigRuleList = [] #强关联规则容器
```

```
    for i in range(1, len(L)): #对所有频繁项集循环操作，这里需要注意的是频繁1项集不存在关联规则
        for freqSet in L[i]: # 对频繁项集中的每个子集关联规则进行挖掘
            H1 = [frozenset([item]) for item in freqSet]
            if (i > 1):
                rulesFromConseq(freqSet, H1, supportData, bigRuleList,minConf) #频繁多项集
            else:
                calcConf(freqSet, H1, supportData, bigRuleList, minConf) #频繁二项集
    return bigRuleList
```

函数运行结果：

```
brl = generateRules(L, supportData, minConf=0.5)
```

[54]:
```
brl = generateRules(L, supportData, minConf=0.5)
```

```
frozenset({3}) --> frozenset({1}) conf: 0.6666666666666666
frozenset({1}) --> frozenset({3}) conf: 1.0
frozenset({3}) --> frozenset({2}) conf: 0.6666666666666666
frozenset({2}) --> frozenset({3}) conf: 0.6666666666666666
frozenset({5}) --> frozenset({3}) conf: 0.6666666666666666
frozenset({3}) --> frozenset({5}) conf: 0.6666666666666666
frozenset({5}) --> frozenset({2}) conf: 1.0
frozenset({2}) --> frozenset({5}) conf: 1.0
frozenset({3, 5}) --> frozenset({2}) conf: 1.0
frozenset({2, 5}) --> frozenset({3}) conf: 0.6666666666666666
frozenset({2, 3}) --> frozenset({5}) conf: 1.0
frozenset({5}) --> frozenset({2, 3}) conf: 0.6666666666666666
frozenset({3}) --> frozenset({2, 5}) conf: 0.6666666666666666
frozenset({2}) --> frozenset({3, 5}) conf: 0.6666666666666666
```

读取外部数据进行关联规则挖掘

```
gr = open('groceries.txt')
gro = gr.readlines()
gro
```

清洗一条数据：

```
gro[1].strip('\n').split(' ')
```

清洗所有的数据

```
tran = [gro[i].strip('\n').split(' ') for i in range(len(gro))]
tran
```

将清洗后的数据带入apriori()函数中，生成频繁一项集和所有项集的支持度

```
L, supportData = apriori(tran, minSupport = 0.5)
L
supportData
```

生成所有的关联规则：

```
brl = generateRules(L, supportData, minConf=0.5)
```

[63]:
```
brl = generateRules(L, supportData, minConf=0.5)
```

```
frozenset({'milk'}) --> frozenset({'bread'}) conf: 0.7499999999999999
frozenset({'bread'}) --> frozenset({'milk'}) conf: 0.7499999999999999
frozenset({'diaper'}) --> frozenset({'bread'}) conf: 0.7499999999999999
frozenset({'bread'}) --> frozenset({'diaper'}) conf: 0.7499999999999999
frozenset({'beer'}) --> frozenset({'diaper'}) conf: 1.0
frozenset({'diaper'}) --> frozenset({'beer'}) conf: 0.7499999999999999
frozenset({'milk'}) --> frozenset({'diaper'}) conf: 0.7499999999999999
frozenset({'diaper'}) --> frozenset({'milk'}) conf: 0.7499999999999999
```

五、案例：发现美国国会投票中的模式

这里我们使用的是1984年美国国会投票的数据，这个数据集中包含了每个美国众议院议员对CQA确定的16个主要选票的投票。CQA列出了九种不同类型的投票：投票赞成、配对赞成和宣布赞成（这三个简化为yea），投票反对、配对反对、宣布反对（这三个简化为nay），投票、为避免利益冲突而投票、弃权或以其他方式表明立场（这三个简化为未知意向）

数据集来源于UCI数据库：http://archive.ics.uci.edu/ml/datasets/Congressional+Voting+Records

数据集包含了435条记录17个特征，每条记录（即每个事务）包含议员的党派信息以及他对16个关键问题的投票记录，每个特征描述如下：

特征索引	特征名称
0	党派（民主党，共和党）
1	残疾人婴幼儿提案
2	水项目费用分摊
3	预算决议案
4	医生费用冻结决议案
5	萨尔瓦多援助
6	校园宗教团体决议
7	反卫星禁试决议
8	援助尼加拉瓜反政府
9	MX导弹决议案
10	移民决议案
11	合成燃料公司削减决议
12	教育支出决议
13	超级基金起诉权
14	犯罪决议案
15	免税出口决议案
16	南非出口管理决议案

1. 导入数据集并进行简单数据处理

```
votes = pd.read_csv('house-votes-84.data.txt',sep=',',header=None)
votes.head()

votes.shape #(435, 17)

#将列名修改为中文
votes.columns = ['党派','残疾人婴幼儿提案','水项目费用分摊','预算决议案'
                 ,'医生费用冻结决议案','萨尔瓦多援助','校园宗教团体决议'
                 ,'反卫星禁试决议','援助尼加拉瓜反政府','MX导弹决议案'
                 ,'移民决议案','合成燃料公司削减决议','教育支出决议'
                 ,'超级基金起诉权','犯罪决议案','免税出口决议案','南非出口管理决议案']

#将列名和投票结果合并
for i in range(1,votes.shape[1]):
    votes.iloc[:,i] = votes.iloc[:,i].apply(lambda x:x+'-'+votes.columns[i])
```

```
#将数据变为列表
voteslist=votes.values.tolist()
```

2. 挖掘所有的频繁项集

利用前面已经写好的Apriori算法来进行频繁项集的挖掘，这里设定默认的支持度阈值50%

```
L,suppData = apriori(voteslist, minSupport = 0.5)
```

从运行结果中，可以看出，当支持度阈值设定为50%的时候，产生了12个频繁1项集，4个频繁2项集，1个频繁3项集。如果降低支持度阈值，你会发现有更多的频繁项集被挖掘出来，感兴趣的小伙伴可以自行尝试~

3. 挖掘强关联规则

这里设定置信度阈值为90%，利用generateRules函数可以挖掘出的关联规则有：

```
brl = generateRules(L, suppData, minConf=0.9)
```

运行结果中出现了以下几条规则：

If	Then	可信度
y-预算决议案	democrat	91.3%
n-医生费用冻结决议案	democrat	99.1%
democrat	n-医生费用冻结决议案	91.8%
y-援助尼加拉瓜反政府	democrat	90.0%
y-预算决议案，n-医生费用冻结决议案	democrat	100.0%
y-预算决议案，democrat	n-医生费用冻结决议案	94.8%

这些强规则是在支持度为50%的情况下挖掘出来的，也就是说，这些规则至少在50%以上的记录中出现过。对于{y-预算决议案，n-医生费用冻结决议案}→{democrat}这条规则，在100%的情况下是成立的，也就是说，这是一个必然事件，只要他给预算决议案投了赞成票，并且医生费用冻结决议案投了反对票，那么这个人一定是民主党。大家感兴趣的话，可以回到原数据集中，查看那些给预算决议案投了赞成票、医生费用冻结决议案投了反对票的人是不是全部都是民主党。剩下的几条也都是很有意思的规则。

这里不知道大家是否发现，当设置支持度阈值为50%，置信度阈值为90%的时候，我们挖掘出来的强关联规则全部是跟民主党相关的，为什么会出现这样的情况呢？

```
votes['党派'].value_counts() #统计各党派数量

votes['党派'].value_counts()/votes.shape[0] #计算各党派所占比例
```

从上述代码运行结果可以看出，共和党所占比例仅为38.6%，如果设定支持度为50%，它根本就不会进入频繁项集。如果我们一定想要查看共和党的一些特征，挖掘与共和党相关的一些强关联规则，那么我们可以将支持度降低一点，比如30%，然后再来进行频繁项集的挖掘、关联规则的提取。但是需要提醒的是，如果降低了支持度，那频繁项集的数量一定会很大程度的增加，那么置信度的设定就要更高一点，否则数据量大的话很容易卡死~

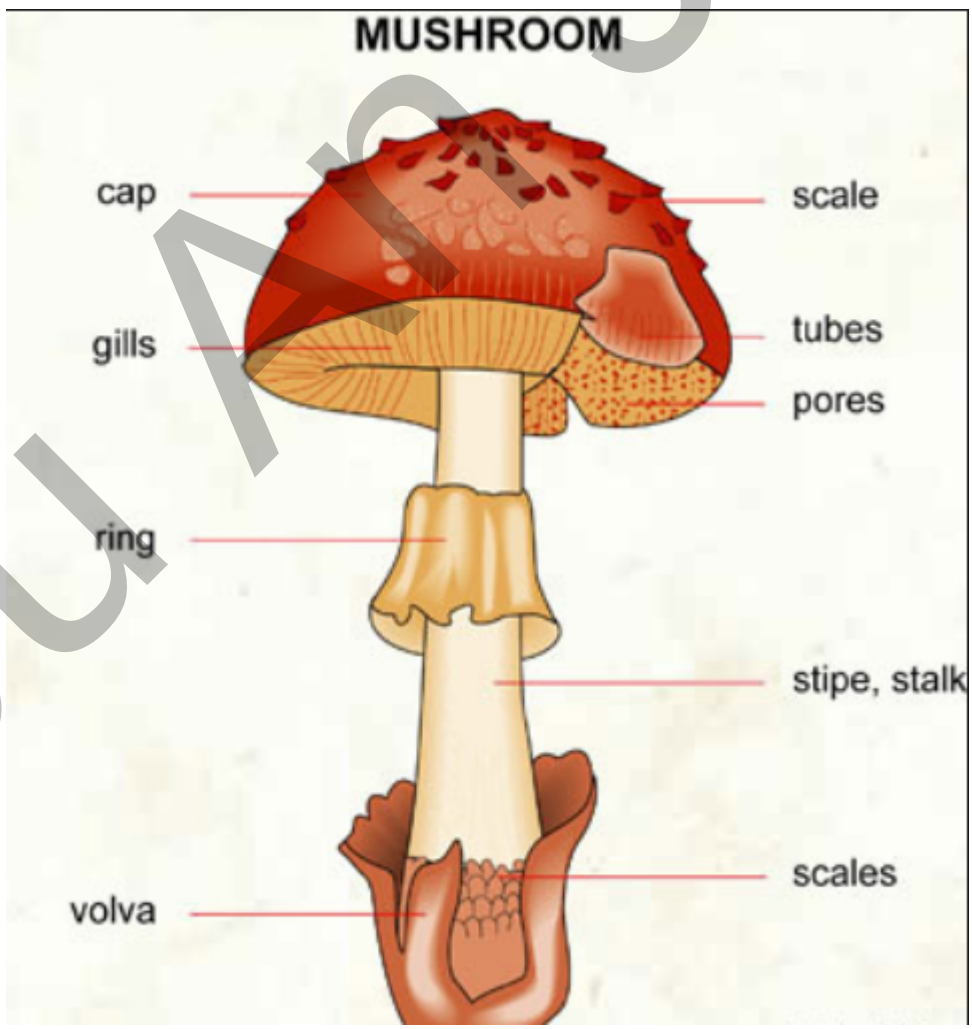
```
L,suppData = apriori(voteslist, minSupport = 0.3) #设定最小支持度为30%
brl = generateRules(L, suppData, minConf=0.99) #设定最小置信度为99%

#运行出来，会发现共有973个频繁项集，259条强关联规则

#我们可以进一步探索，与民主党和共和党相关的强关联规则
for i in range(len(brl)):
    if frozenset({'republican'}) in brl[i] :
        print(brl[i])
    if frozenset({'democrat'}) in brl[i]:
        print(brl[i])
```

六、案例：发现毒蘑菇的相似特征

现实生活中，大多数时候，我们并不关心所有的频繁项集，而是对某些特定元素的项集感兴趣。比如说毒蘑菇研究中就只对毒蘑菇相关的特性感兴趣。



毒蘑菇又称毒蕈，是指大型真菌的子实体食用后对人体产生中毒反应的物种。世界上每年都会有很多因为误食毒蘑菇而引发的中毒事件，主要是由于有些毒菌和食用菌的宏观特征没有明显区别。所以长期以来，鉴别毒蘑菇是人们十分关心的事情。

最后的这个例子，就用Apriori算法挖掘出毒蘑菇的一些公共特征，利用这些特征就可以避免迟到那些有毒的蘑菇。这个数据集来自于UCI的机器学习数据集：https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/

这个数据集中共有8124条样本，23个特征，如下所示：

序号	英文特征名	特征名	特征说明
0	class	蘑菇类型	edible:食用菌,poisonous:毒菌
1	cap-shape	帽形状	b:钟形,c:锥形,x:凸形,f:扁平,k:把手形,s:凹陷形
2	cap-surface	帽表面	f:纤维,g:凹槽,y:鳞片状,s:光滑
3	cap-color	帽颜色	n:棕色,b:浅黄色,c:肉桂色,g:灰色,r:绿色,p:粉红色,u:紫色,e:红色,w:白色,y:黄色
4	bruises?	有无挫伤	t:有,f:无
5	odor	气味	a杏仁,l:茴香,c:杂酚油,y:腥味,f:恶臭,m:霉味,n:无味,p:刺鼻,s:辣
6	gill-attachment	菌褶附属物	a:附着,d:降落,f:free,n:缺口
7	gill-spacing	菌褶间距	c:闭合,w:紧凑,d:稀疏
8	gill-size	菌褶尺寸	b:宽,n:窄
9	gill-color	菌褶颜色	k:黑色,n:棕色,b:浅黄色,h:巧克力色,g:灰色,r:绿色,o:橘黄色,p:粉色,u:紫色,e:红色,w:白色,y:黄色
10	stalk-shape	茎形状	e:扩展形,t:锥形
11	stalk-root	茎根部	b:球根状,c:团状,u:杯状,e:相等,z:根状菌索,r:根状,?:缺失值
12	stalk-surface-above-ring	蕈圈上部茎表面	f:纤维,y:鳞片状,k:柔滑,s:光滑
13	stalk-surface-below-ring	蕈圈下部茎表面	f:纤维,y:鳞片状,k:柔滑,s:光滑
12	stalk-color-above-ring	蕈圈上部茎颜色	n:棕色,b:浅黄色,c:肉桂色,g:灰色,o:橘黄色,p:粉色,e:红色,w:白色,y:黄色
13	stalk-color-below-ring	蕈圈下部茎颜色	n:棕色,b:浅黄色,c:肉桂色,g:灰色,o:橘黄色,p:粉色,e:红色,w:白色,y:黄色
16	veil-type	网类型	p:局部,u:常规
17	veil-color	网颜色	n:棕色,o:橘黄色,w:白色,y:黄色
18	ring-number	蕈圈数量	n:无,o:1个,t:2个
19	ring-type	蕈圈类型	c:蛛网状,e: 消逝波形,f:火光闪耀形,l:大的,n:无,p:垂状,s:覆盖形,z:区域形
20	spore-print-color	孢子颜色	k:黑色,n:棕色,b:浅黄色,h:巧克力色,r:绿色,o:橘黄色,u:紫色,w:白色,y:黄色
21	population	数量	a:丰富,c:簇,n:大量,s:分散,v:几个,y:单个
22	habitat	栖息地	g:草地,l:树叶,m:草甸,p:小路,u:城市,w:废物堆,d:树林

1.导入数据集并做预处理

首先导入所需的包和蘑菇数据集

```
import pandas as pd
import numpy as np

df = pd.read_csv('agaricus-lepiota.data.txt',header=None)
df.head()

df.shape #(8124, 23)
```

简单进行数据处理：

1. 更改特证名
2. 删除无用特征
3. 将特证名和值连起来
4. 将DF变为列表

```
#更改特证名
names = ['蘑菇类型','帽形状','帽表面','帽颜色','有无挫伤','气味'
         ,'菌褶附属物','菌褶间距','菌褶尺寸','菌褶颜色','茎形状'
```

```
          ,'茎根部','蕈圈上部茎表面','蕈圈下部茎表面','蕈圈上部茎颜色'
          ,'蕈圈下部茎颜色','网类型','网颜色','蕈圈数量'
          ,'蕈圈类型','孢子颜色','数量','栖息地']
df.columns = names

#如果一个特征中所有的值都是一样，说明可食用蘑菇和毒蘑菇在这个特征上表现是一致的
#那说明这个特征中所含的有用信息为0，可以直接剔除
for i in range(df.shape[1]):
    col = df.iloc[:,i].value_counts()
    if len(col)<=1:
        print(col)

df['网类型'].value_counts()
df.drop('网类型',axis=1,inplace=True) #删除特征

#由于不同特征中包含的值有些是相同，所以将特征名和值连起来使其作为独一无二的项
df_ = df.copy()
for i in range(1,df_.shape[1]):
    df_.iloc[:,i] = df_.iloc[:,i].apply(lambda x:df_.columns[i]+'-'+x)

#将DF变为列表
mlist = df_.values.tolist()
```

2. 进行频繁项集的挖掘

由于是“人命关天”的大事，所以我们采取保守策略，设定最小支持度为30%

```
L,suppData= apriori(mlist, minSupport=0.3)

#查看一共挖掘出了多少条频繁项集
for i in range(len(L)):
    n+=len(L[i])
n

#查看这些频繁项集中有哪些是与毒蘑菇相关的
L0 = []
for i in range(len(L)):
    for item in L[i]:
        if item.intersection('p'):
            L0.append(item)
L0

#创建有毒蘑菇的集合
re = []
for i in range(len(L0)):
    for item in L0[i]:
        if item not in re:
            re.append(item)
re
```

根据代码运行的结果可以得出，在频繁项集中，出现的毒蘑菇的特征主要有：

['p', '菌褶附属物-f', '菌褶间距-c', '网颜色-w', '蕈圈数量-o', '数量-v', '有无挫伤-f']

以后在辨识蘑菇的时候，要格外注意这几种特征，如果出现了这些特征中的一种或者多种，请坚决不要吃！！！

3. 挖掘强关联规则

这里我们重点关注的是与毒蘑菇有关的关联规则，所以在挖掘出满足最小置信度的关联规则之后，还要多做一步就是找出与毒蘑菇有关的规则。

```
#挖掘满足最小置信度为95%的强关联规则
rules= generateRules(L,suppData, minConf=0.95)

len(rules) #共有4332条满足要求的规则

#从所有满足要求的规则中找出有毒蘑菇有关的规则
for i in range(len(rules)):
    if frozenset({'p'}) in rules[i]:
        print(rules[i])
```

运行结果再一次向我们证明了：这六种蘑菇特性与毒蘑菇强相关**['菌褶附属物-f', '菌褶间距-c', '网颜色-w', '蕈圈数量-o', '数量-v', '有无挫伤-f']**

在上述频繁项集的挖掘过程中，我们的处理方法是把特证名和特征值放到一起作为一个项。在《机器学习实战》这本书中，给出的方法是先对原数据集进行解析，把每一个项编码成独一无二的数字，然后再带入模型进行频繁项的挖掘。编码的思想是：一列一列操作，如果第一列中有两类值，则分别编码为1,2，第二列中有6类值，则编码为3,4,5,6,7,8，以此类推。下面我们一起来看一下操作过程。

```
#导入原始数据集
df = pd.read_csv('agaricus-lepiota.data.txt',header=None)

#复制原始数据集，然后在复制数据集上进行操作，避免直接在原数据集上修改
df1 = df.copy()

#对数据集进行编码
ret =[]     #放置所有列编码的结果（字典形式）
a=0
for i in range(df1.shape[1]): #对每一列进行编码操作
    enc = {}  #每放置一列编码的结果
    col= df1.iloc[:,i].value_counts() #对每一列的值进行统计
    for ind,key in enumerate(col.index): #提取每一个值及其对应索引
        enc[key] = ind+a+1  #编码从1开始，所以+1
    ret.append(enc) #将此列编码结果放置到re中
    for i in enc.values():a=i #当列编码的最后一个值

ret #re中存放的是每一列编码的结果

#把编码结果映射到数据集中
```

```
for i in range(len(ret)):
    df1.iloc[:,i] = df1.iloc[:,i].map(ret[i])

#这里需要注意的是，编码之后，数据集中所有的值均为int类型
#但是我们需要的是字符串类型，所以还需要进一步转换
for i in range(df1.shape[1]):
    df1.iloc[:,i] = df1.iloc[:,i].apply(lambda x:str(x))

#到这里，我们就将前期的数据处理工作做好了~~~
#接下来就可以带入模型啦
mushdata = df1.values.tolist()
L,suppData= apriori(mushdata, minSupport=0.3) #挖掘频繁项集

#找到频繁项集中与毒蘑菇有关的项集
L0 = []
for i in range(len(L)):
    for item in L[i]:
        if item.intersection('2'):
            L0.append(item)

#创建毒蘑菇特征集合
re = []
for i in range(len(L0)):
    for item in L0[i]:
        if item not in re:
            re.append(item)

#还原这些数字代表的原意义
ssc = {}
for i in range(df1.shape[0]):
    for j in range(df1.shape[1]):
        if df1.iloc[i,j] in re[1:]:
            ssc[df.columns[j]]=df.iloc[i,j]

#挖掘关联规则
rules= generateRules(L,suppData, minConf=0.95)
#提取与毒蘑菇有关的规则
for i in range(len(rules)):
    if frozenset({'2'}) in rules[i]:
        print(rules[i])
```

除了上述自己动手编码数据之外，还可以借助别人已经编码好的数据集，直接进行操作。《机器学习实战》书上也给我们提供了解析好的数据集。我们拿来直接跑一便模型，看与我们上述过程得出的结果是否一致.

这里需要注意的是，mushroom.dat数据集中把毒蘑菇poisonous编码为1，食用菌edible编码为2 所以在找毒蘑菇特性的时候，应该找与'1'相关的.

```
mushDatSet = [line.split() for line in open('mushroom.dat').readlines()] #导入编码好的数据集
L,suppData= apriori(mushDatSet, minSupport=0.3) #挖掘频繁项集
rules= generateRules(L,suppData, minConf=0.95)  #挖掘关联规则
```

```
#从频繁项集中提取出与毒蘑菇有关的项集
L0 = []
for i in range(len(L)):
    for item in L[i]:
        if item.intersection('1'):
            L0.append(item)

#将与毒蘑菇有关的编码项集整合
re = []
for i in range(len(L0)):
    for item in L0[i]:
        if item not in re:
            re.append(item)

#将与毒蘑菇有关的项集编码还原到原数据中
ssc = {}
for i in range(len(mushDatSet)):
    for j in range(len(mushDatSet[0])):
        if mushDatSet[i][j] in re[1:]:
            ssc[df.columns[j]]=df.iloc[i,j]

#提取与毒蘑菇有关的关联规则
for i in range(len(rules)):
    if frozenset({'1'}) in rules[i]:
        print(rules[i])
```

上述这三种方法都可以实现毒蘑菇的频繁项集挖掘。这里需要说明的一点是，我们挖掘出来的与毒蘑菇有关的频繁项集，可以帮助我们确定，如果有这些特征，那么这些蘑菇就不要吃了。但是，没有这些特征的蘑菇也不一定就是能食用的。如果吃错了蘑菇，可能真的会因此丧命。所以，请大家要珍爱生命哦~