实验介绍

1.实验内容

本实验介绍Apriori算法。并使用python实现该算法。

2.实验目标

通过本实验掌握Apriori算法。

3.实验知识点

- Apriori算法

4.实验环境

- python 3.6.5

5.预备知识

- 初等数学知识
- Linux命令基本操作
- Python编程基础

【原理】Apriori算法原理

Apriori算法是一种用于关联规则挖掘的代表性算法。从本节开始，我们已经进入了机器学习和数据挖掘相交叉的地带。

数据挖掘与机器学习

数据挖掘和机器学习的关系就好比，机器学习是数据挖掘的弹药库中一类相当庞大的弹药集。既然是一类弹药，其实也就是在说数据挖掘中肯定还有其他非机器学习范畴的技术存在。Apriori算法就属于一种非机器学习的数据挖掘技术。
在非机器学习的数据挖掘技术中，我们并不会去建立这样一个模型，而是直接从原数据集入手，设法分析出隐匿在数据背后的某些信息或知识。在后续介绍Apriori算法时，你会相当明显地感受到这一特点。

Apriori算法

我们先从一个例子了解一下apriori原理。被大家所熟知的"啤酒尿布"的购买行为问题，其实就是一个具有关联性的行为。而发现这种关联性行为的方式，可以用apriori原理来实现。
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”或者“关联规则学习”。而我们需要关注的问题是，如何使用更智能的方法，在更合理的时间范围内找到我们所需要的关联规则。
以此，引出了我们的aprori算法。

我们先来介绍几个概念。

关联分析，是一种在大规模数据集中寻找有趣关系的任务。在这种关系中，存在着两种形式：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。频繁项集是我们对原始数据格式化后的源数据集，而关联规则则是寻找源数据集关系得到的结果数据集。{}为集合标识。在下图中，{葡萄酒，尿布，豆奶}就是频繁项集的一个例子。在频繁项集中，我们也可以找到诸如“{尿布}->{葡萄酒}”的关联规则。这意味着如果有人买了尿布，那么他很可能也会买葡萄酒。

交易号码	商品
0	豆奶，莴苣
1	莴苣，尿布，葡萄酒，甜菜
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11-1　一个来自Hole Foods天然食品店的简单交易清单

那么应该如何定义这些有趣的关系？谁来定义什么是有趣？当寻找频繁项集时，频繁的定义又是什么？

一个项集的支持度(support)被定义为数据集中包含该项集的记录所占的比例。支持度是针对项集，我们可以定义一个最小支持度，来保留满足最小支持度的项集。

如在上图中，{豆奶}的支持度为4/5，而{豆奶，尿布}的支持度为3/5

可信度又称为置信度(confidence)是针对一条关联规则定义的。

对于关联规则“{尿布}->{葡萄酒}”，它的可信度被定义为“支持度({尿布，葡萄酒})/支持度({尿布})”，计算该规则可信度为0.75，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。

在上面介绍的支持度和可信度被用来量化关联分析是否成功。但在实际操作的过程中，我们需要对物品所有的组合计算其支持度和可信度，当物品量上万时，上述的操作会非常非常慢。

我们该如何解决这种问题呢?

我们已经知道，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）。

Apriori算法是生成频繁集的一种算法。

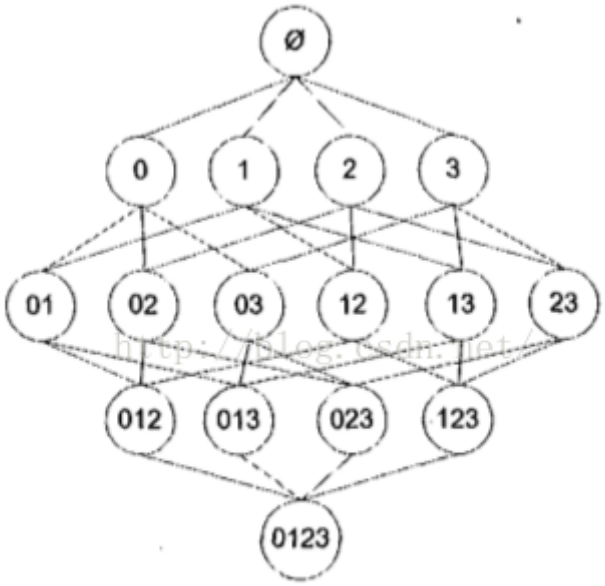

图11-2　集合{0，1，2，3}中所有可能的项集组合

上图显示了4种商品所有可能的组合。对给定的集合项集{0,3}，需要遍历每条记录并检查是否同时包含0和3,扫描完后除以记录总数即可得支持度。对于包含N种物品的数据集共有2的N次方-1种项集组合，即使100种，也会有1.26×10的30次方种可能的项集组成。

为降低计算时间，可用Apriori原理：如果某个项集是频繁的，那么它的所有子集也是频繁的。而我们需要使用的则是它的逆反定义：如果一个项集是非频繁集，那么它的所有超集也是非频繁的。

在下图中，已知阴影项集{2,3}是非频繁的，根据Apriori原理，我们知道{0,2,3},{1,2,3}以及{0,1,2,3}也是非频繁的，我们就不需要计算其支持度了。

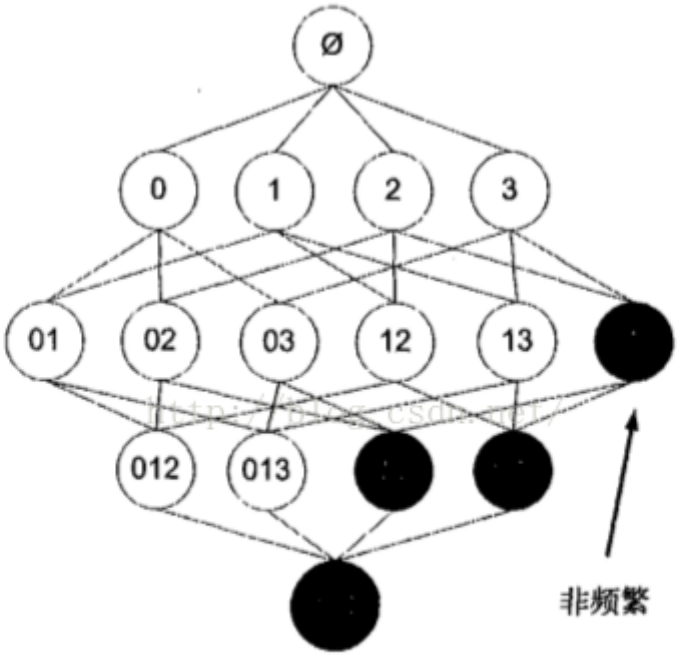

图11-3　图中给出了所有可能的项集，其中非频繁项集用灰色表示。由于集合{2,3}是非频繁的，因此{0,2,3}、{1,2,3}和{0,1,2,3}也是非频繁的，它们的支持度根本不需要计算

这样就可以避免项集数目的指数增长，从而找到频繁项集。

【实验】python实现Apriori算法——发现频繁集

我们已经知道关联分析的目标分为：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。本节我们将重点放在如何发现频繁项集上。

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

而集合L1中的元素相互组合构成C2，C2再进一步过滤变成L2,以此循环直到Lk为空。

在上述对Apriori算法的描述中，我们可以很清楚的看到，需要构建相应的功能函数来实现该算法：

1. createC1 -构建集合C1
2. scanD -过滤集合C1，构建L1
3. aprioriGen -对集合Lk中的元素相互组合构建Ck+1
4. apriori -集成函数

我们按照该逻辑来实现我们的apriori算法，并找到频繁项集。

在当前工作目录下，添加如下代码：

In [1]:

```
from numpy import *
def loadDataSet():
    """
    函数说明：加载数据集
    """
    return [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]
def createC1(dataSet):
    """
    函数说明：构建集合C1。即所有候选项元素的集合。

    parameters:
        dataSet -数据集
    Returns:
        frozenset列表
    output:
        [forzenset([1]),forzenset([2]),……]
    """
    C1 = []                         #创建一个空列表
    for transaction in dataSet: #对于数据集中的每条记录
        for item in transaction:#对于每条记录中的每一个项
            if not [item] in C1:       #如果该项不在C1中，则添加
                C1.append([item])
    C1.sort()                       #对集合元素排序
    return list(map(frozenset,C1))    #将C1的每个单元列表元素映射到forzenset()
def scanD(D,Ck,minSupport):
    """
    函数说明：构建符合支持度的集合Lk
    parameters:
        D -数据集
        Ck -候选项集列表
        minSupport -感兴趣项集的最小支持度
    Returns:
        retList -符合支持度的频繁项集合列表L
        supportData -最频繁项集的支持度
    """
    ssCnt = {}                                                  #创建空字典
    for tid in D:                                               #遍历数据集中的所有交易记录
        for can in Ck:                                          #遍历Ck中的所有候选集
            if can.issubset(tid):                               #判断can是否是tid的子集
                if not can in ssCnt:  ssCnt[can] = 1  #如果是记录的一部分，增加字典中对应的计数值。
                else: ssCnt[can] += 1
    numItems = float(len(D))                                    #得到数据集中交易记录的条数
    retList = []                                                #新建空列表
    supportData = {}                                            #新建空字典用来存储最频繁集和支持度
    for key in ssCnt:
        support = ssCnt[key] / numItems                         #计算每个元素的支持度
```

```
        if support >= minSupport:                    #如果大于最小支持度则添加到retList中
            retList.insert(0,key)
        supportData[key] = support                   #并记录当前支持度，索引值即为元素值
    return retList,supportData

if __name__ == '__main__':
    dataSet = loadDataSet()
    print("数据集:\n",dataSet)
    C1 = createC1(dataSet)
    print("候选集C1:\n",C1)
    D = list(map(set,dataSet))  #将数据转换为集合形式
    L1,supportData = scanD(D,C1,0.5)
    print("满足最小支持度为0.5的频繁项集L1: \n",L1)
```

```
数据集:
 [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]
候选集C1:
 [frozenset({1}), frozenset({2}), frozenset({3}), frozenset({4}), frozenset({5})]
满足最小支持度为0.5的频繁项集L1:
 [frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})]
```

其中forzenset为不可变集合，它的值是不可变的，好处是它可以作为字典的key，也可以作为其他集合的元素。我们这里必须使用forzenset而不是set，就是因为我们需要将这些集合作为字典键值使用。

至此，我们完成了C1和L1的构建。那么对于L1中元素的组合，我们需要先引入一个组合定义。
假定我们的L1中元素为[{0},{1},{2}],我们想要aprioriGen函数生成元素的组合C2，即[{0,1},{0,2},{1,2}]。这是单个项的组合。那么我们考虑一下，如果想利用[{0,1},{0,2},{1,2}]来创建三元素的候选项集C3呢？如果依旧按照单个项的组合方法，将每两个集合合并，就会得到{0,1,2},{0,1,2},{0,1,2}。也就是说，我们重复操作了3次。

接下来还需要扫描该结果来得到非重复结果，但我们要确保的是遍历列表的次数最少。那么如何解决这个问题呢？我们观察到，如果比较集合{0,1},{0,2},{1,2}的第一个元素并只对第一个元素相同的集合求并操作，我们会得到{0,1,2}相同的结果，但只操作了1次！这样的话就不需要遍历列表来寻找非重复值。也能够实现apriori算法的原理。当我们创建三元素的候选集时，k=3，但此时待组合的频繁项集元素为f=2，即我们只需比较前f-1个元素，也就是前k-2个元素。

打开apriori.py文件，修改代码如下：

In [2]:

```
def aprioriGen(Lk,k):
    """
    函数说明:构建集合Ck
    parameters:
        Lk -频繁项集列表L
        k -候选集的列表中元素项的个数
    return:
        retList -候选集项列表Ck
    """
    retList = []                                        #创建一个空列表
    lenLk = len(Lk)                                     #得到当前频繁项集合列表中元素的个数
    for i in range(lenLk):                              #遍历所有频繁项集合
        for j in range(i+1,lenLk):                      #比较Lk中的每两个元素，用两个for循环实现
            L1 = list(Lk[i])[:k-2]; L2 = list(Lk[j])[:k-2]  #取该频繁项集合的前k-2个项进行比较
            #[注]此处比较了集合的前面k-2个元素，需要说明原因
            L1.sort(); L2.sort()                        #对列表进行排序
            if L1 == L2:
                retList.append(Lk[i]|Lk[j])             #使用集合的合并操作来完成 e.g.: [0,1],[0,2]->[0,1,2]
    return retList
def apriori(dataSet,minSupport=0.5):
    """
    函数说明: apriori算法实现
    parameters:
        dataSet -数据集
        minSupport -最小支持度
    return:
        L -候选项集的列表
        supportData -项集支持度
    """
    C1 = createC1(dataSet)
    D = list(map(set,dataSet))                          #将数据集转化为集合列表
    L1, supportData = scanD(D,C1,minSupport)            #调用scanD()函数，过滤不符合支持度的候选项集
    L = [L1]                                            #将过滤后的L1放入L列表中
    k = 2                                               #最开始为单个项的候选集，需要多个元素组合
    while len(L[k - 2])>0:
        Ck = aprioriGen(L[k-2],k)                       #创建Ck
        Lk, supK = scanD(D,Ck,minSupport)               #由Ck得到Lk
        supportData.update(supK)                        #更新支持度
        L.append(Lk)                                    #将Lk放入L列表中
        k += 1                                          #继续生成L3, L4....
    return L, supportData

if __name__ == '__main__':
    dataSet = loadDataSet()
    print("数据集:\n",dataSet)
    C1 = createC1(dataSet)
```

```
    print("候选集C1:\n",C1)
    D = list(map(set,dataSet))  #将数据转换为集合形式
    L1,supportData = scanD(D,C1,0.5)
    print("满足最小支持度为0.5的频繁项集L1: \n",L1)
    L,suppData = apriori(dataSet)
    print("满足最小支持度为0.5的频繁项集列表L: \n",L)
    print("L2:\n",L[1])
    print("使用L2生成的C3:\n",aprioriGen(L[1],3))

    L,suppData = apriori(dataSet)
    print("满足最小支持度为0.7的频繁项集列表L: \n",L)
```

```
数据集:
 [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]
候选集C1:
 [frozenset({1}), frozenset({2}), frozenset({3}), frozenset({4}), frozenset({5})]
满足最小支持度为0.5的频繁项集L1:
 [frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})]
满足最小支持度为0.5的频繁项集列表L:
 [[frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})], [frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5}), frozenset
({1, 3})], [frozenset({2, 3, 5})], []]
L2:
 [frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5}), frozenset({1, 3})]
使用L2生成的C3:
 [frozenset({2, 3, 5})]
满足最小支持度为0.7的频繁项集列表L:
 [[frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})], [frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5}), frozenset
({1, 3})], [frozenset({2, 3, 5})], []]
```

可以看到，使用L2生成的候选项集C3，只对k-2个项相同的元素合并。那其实不生成{1,3,5}的原因是，由于{1,5}是非频繁项，那么其超集均为非频繁项。故此极大减少计算量。
到此我们也得到了我们的最频繁项集列表。

【实验】python实现Apriori算法——挖掘关联规则

上一节介绍了如何使用Apriori算法来发现频繁集，现在需要解决的问题则是如何找出关联规则。

在原理讲解中，我们找到诸如“{尿布}->{葡萄酒}”的关联规则。这意味着如果有人买了尿布，那么他很可能也会买葡萄酒。但是，这一条反过来，却不一定成立。有可能买葡萄酒的人，根本不会去买尿布。那么，成立的定义是什么呢？还记得我们介绍的可信度度量值，对于关联规则的量化方法，则是使用可信度。

对于关联规则“P->H”，它的可信度被定义为“支持度(P | H)/支持度(P)”。其中|符合在python中为合并符。在找出关联规则的过程中，首先从一个频繁项集开始，接着创建一个规则列表，其中规则右部只包含一个元素，然后对这些规则进行测试。接下来，合并所有剩余规则来创建一个新的规则列表，其中规则右部包含两个元素。直到规则左右剩余一个元素。

同样，我们需要创建相应的功能函数来实现：

1. rulesFromConseq -从频繁项集生成规则列表
2. calcConf -对规则进行测试并过滤
3. generateRules -集成函数

我们来看下这种方法的实际效果，对代码作出如下修改：

In [3]:
```python
def rulesFromConseq(freqSet,H,supportData,brl,minConf=0.7):
    """
    函数说明：规则构建函数
    parameters:
        freqSet -频繁项集合
        H -可以出现在规则右部的元素列表
        supportData -支持度字典
        brl -规则列表
        minConf -最小可信度
    return:
        null
    """
    m = len(H[0])                                          #得到H中的频繁集大小m
    if len(freqSet) > (m + 1):                             #查看该频繁集是否大到可以移除大小为m的子集
        Hmp1 = aprioriGen(H, m+1)                          #构建候选集Hm+1，Hmp1中包含所有可能的规则
        Hmp1 = calcConf(freqSet,Hmp1,supportData,brl,minConf)    #测试可信度以确定规则是否满足要求
        if len(Hmp1)>1:                                    #如果不止一条规则满足要求，使用函数迭代判断是否能进一步组合这些规则
            rulesFromConseq(freqSet,Hmp1,supportData,brl,minConf)

def calcConf(freqSet,H,supportData,brl,minConf=0.7):
    """
    函数说明：计算规则的可信度，找到满足最小可信度要求的规则
    parameters:
        freqSet -频繁项集合
        H -可以出现在规则右部的元素列表
        supportData -支持度字典
        brl -规则列表
        minConf -最小可信度
    return:
        prunedH -满足要求的规则列表
    """
    prunedH = []                                           #为保存满足要求的规则创建一个空列表
    for conseq in H:
        conf = supportData[freqSet]/supportData[freqSet-conseq] #可信度计算[support(PUH)/support(P)]
        if conf>=minConf:
            print(freqSet-conseq,'-->',conseq,'可信度为:',conf)
            brl.append((freqSet-conseq,conseq,conf))       #对bigRuleList列表进行填充
            prunedH.append(conseq)                         #将满足要求的规则添加到规则列表
    return prunedH
def generateRules(L,supportData,minConf=0.7):
    """
    函数说明：关联规则生成函数
    parameters:
        L -频繁项集合列表
        supportData -支持度字典
        minConf -最小可信度
```

```
    return:
        bigRuleList -包含可信度的规则列表
    """
    bigRuleList = []                                        #创建一个空列表
    for i in range(1,len(L)):                               #遍历频繁项集合列表
        for freqSet in L[i]:                                #遍历频繁项集合
            H1 = [frozenset([item]) for item in freqSet]    #为每个频繁项集合创建只包含单个元素集合的列表H1
            if i>1:                                        #要从包含两个或者更多元素的项集开始规则构建过程
                rulesFromConseq(freqSet,H1,supportData,bigRuleList,minConf)
            else:                                           #如果项集中只有两个元素，则计算可信度值，（len(L)=2)
                calcConf(freqSet,H1,supportData,bigRuleList,minConf)
    return bigRuleList

if __name__ == '__main__':
    dataSet = loadDataSet()
    print("数据集:\n",dataSet)
    C1 = createC1(dataSet)
    print("候选集C1:\n",C1)
    L,suppData = apriori(dataSet)
    print("满足最小支持度为0.5的频繁项集列表L: \n",L)
    print("满足最小可信度为0.7的规则列表为:")
    rules = generateRules(L,suppData,0.7)
    print(rules)
    print("满足最小可信度为0.5的规则列表为:")
    rules1 = generateRules(L,suppData,0.5)
    print(rules1)
```

```
数据集:
 [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]
候选集C1:
 [frozenset({1}), frozenset({2}), frozenset({3}), frozenset({4}), frozenset({5})]
满足最小支持度为0.5的频繁项集列表L:
 [[frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})], [frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5}), frozenset
({1, 3})], [frozenset({2, 3, 5})], []]
满足最小可信度为0.7的规则列表为:
frozenset({5}) --> frozenset({2}) 可信度为: 1.0
frozenset({2}) --> frozenset({5}) 可信度为: 1.0
frozenset({1}) --> frozenset({3}) 可信度为: 1.0
[(frozenset({5}), frozenset({2}), 1.0), (frozenset({2}), frozenset({5}), 1.0), (frozenset({1}), frozenset({3}), 1.0)]
满足最小可信度为0.5的规则列表为:
frozenset({3}) --> frozenset({2}) 可信度为: 0.6666666666666666
frozenset({2}) --> frozenset({3}) 可信度为: 0.6666666666666666
frozenset({5}) --> frozenset({3}) 可信度为: 0.6666666666666666
frozenset({3}) --> frozenset({5}) 可信度为: 0.6666666666666666
frozenset({5}) --> frozenset({2}) 可信度为: 1.0
frozenset({2}) --> frozenset({5}) 可信度为: 1.0
frozenset({3}) --> frozenset({1}) 可信度为: 0.6666666666666666
frozenset({1}) --> frozenset({3}) 可信度为: 1.0
frozenset({5}) --> frozenset({2, 3}) 可信度为: 0.6666666666666666
frozenset({3}) --> frozenset({2, 5}) 可信度为: 0.6666666666666666
frozenset({2}) --> frozenset({3, 5}) 可信度为: 0.6666666666666666
[(frozenset({3}), frozenset({2}), 0.6666666666666666), (frozenset({2}), frozenset({3}), 0.6666666666666666), (frozenset({5}), frozenset
({3}), 0.6666666666666666), (frozenset({3}), frozenset({5}), 0.6666666666666666), (frozenset({5}), frozenset({2}), 1.0), (frozenset({2}),
frozenset({5}), 1.0), (frozenset({3}), frozenset({1}), 0.6666666666666666), (frozenset({1}), frozenset({3}), 1.0), (frozenset({5}), frozen
set({2, 3}), 0.6666666666666666), (frozenset({3}), frozenset({2, 5}), 0.6666666666666666), (frozenset({2}), frozenset({3, 5}), 0.666666666
6666666)]
```

更改最小置信度后，可以获得更多的规则。

实验总结

本节我们介绍了Apriori算法，并介绍了如何使用python实现Apriori算法，您应该能达到以下两个目标：

1. 掌握Apriori原理。
2. 学会使用python实现相应算法。

参考文献与延伸阅读

参考资料:

1.哈林顿，李锐. 机器学习实战 : Machine learning in action[M]. 人民邮电出版社, 2013.
2.周志华. 机器学习:Machine learning[M]. 清华大学出版社, 2016.

延伸阅读

1.李航. 统计学习方法[M]. 清华大学出版社, 2012.