

# Association rule algorithm

Created	@March 22, 2022 3:09 PM
Tags	Association rule algorithm

Apriori算法原理及基础代码实现

**关联分析**，主要是通过算法在大规模数据集中寻找频繁项集和关联规则。

支持度与置信度是实现Apriori算法的两个概念。

支持度和可信度的基础运算

Example：

算法原理

基础算法实现

从频繁项集中挖掘关联规则

计算可信度/递归计算频繁项集的规则/生成关联规则代码实现

Apriori支持度实现

基础算法思想

Apriori Algorithm in Machine Learning | Association Rule Mining

Recommender System

User-based nearest neighbour

item-based nearest neighbour

Association rule mining

Apriori Algorithm

使用FP-Growth算法来高效发现频繁项集

FP-Tree结构

代码解析

Federated Mining of Interesting Association Rules Over EHRs

关联规则算法汇总

有监督与无监督

例子(additional info)

高数据维度的挖掘方法

(additional info)

统计显著性检验(additional info)

跨特征俩方关联规则联邦方案

详细步骤

Association rule on vertically

partitioned datasets

基于俩方的方案

基于多方的方案 (Not solving yet)

## Apriori算法原理及基础代码实现

【机器学习】Apriori算法--原理及代码实现（Python版）\_慕课笔记

Apriori算法 Apriori算法在数据挖掘中应用较为广泛，常用来挖掘属性与结果之间的相关程度。对于这种寻找数据内部关联关系的做法，我们称之为：关联分析或者关联规则学习。而Apriori算法就是其中非常著名的算法之一。

🔥 <https://www.imooc.com/article/266009>

笔记

Apriori算法在数据挖掘中应用较为广泛，**常用来挖掘属性与结果之间的相关程度**，是关联分析（关联规则学习）的算法之一。

**关联分析**，主要是通过算法在大规模数据集中寻找频繁项集和**关联规则**。

**频繁项集**：经常出现在一起的物品或者属性的集合

**关联规则**：物品或者属性之间存在的内在关系

**支持度与置信度是实现Apriori算法的两个概念。**

**支持度**是用来寻找频繁项集在全体数据中所占的比例

$$support(X,Y) = P(X,Y) = \frac{number(X,Y)}{number(allsample)}$$

**置信度**是体现为一个数据出现后，另一个数据出现的概率

$$confidence(X \rightarrow Y) = P(Y|X) = \frac{P(X,Y)}{P(X)} = \frac{number(X,Y)}{number(X)}$$

**支持度和可信度的基础运算**

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

$$support(尿布, 葡萄酒) = \frac{3}{5}$$

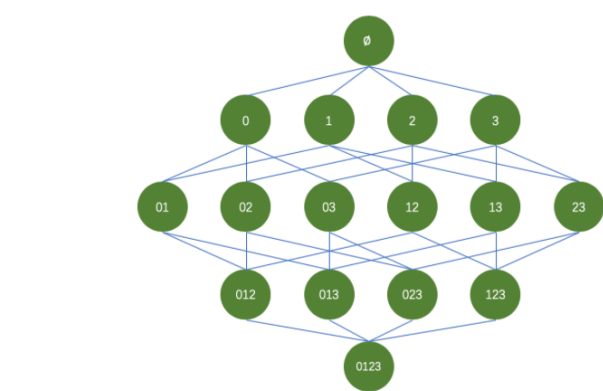
$$support(尿布) = \frac{4}{5}$$

$$confidence(尿布 \rightarrow 葡萄酒) = \frac{3/5}{4/5} = \frac{3}{4}$$

**Example：**

以商品购买为例子。假设一家商店，出售商品0，1，2，3。

我们希望通过挖掘买家购买商品的订单数据，来进行商品之间的组合促销或者说是摆放的位置设置。



针对这些商品，我们的目标是从大量购买数据中找到经常一起被购买的商品。在寻找频繁项集的过程中，我们采用支持度来过滤商品组合。在这四种商品之中，进行了15次轮询，才可以统计出每个频繁项集的支持度。

## Federated Mining of Interesting Association Rules Over EHRs

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7c8429d4-65ca-49f4-9bef-aed0951f3bd5/Federated\\_Mining\\_of\\_Interesting\\_Association\\_Rules\\_.pdf](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7c8429d4-65ca-49f4-9bef-aed0951f3bd5/Federated_Mining_of_Interesting_Association_Rules_.pdf)

笔记

Having federated methods means a step forward in the interoperability since any of source can benefit from the results calculated collaboratively.

FAIR principle

We have presented the need for explained federated mining methods and we have proposed a federated association rule mining algorithm that works with EHR data. It is able to deal with different number of source and data distributions without quality loose.

We have also defined a measure of the internet of an item-set. These federated techniques require a framework that integrates to take advantage of their potential. We plan to integrate the proposed methods with the EHRagg. As we haven mentioned in the previous section, the synchronous schema has some problems, so an asynchronous proposal that can build an incremental solution would also be interesting.

## 关联规则算法汇总

### 有监督与无监督

- 无监督关联规则

**主要是频繁项集的挖掘，选取标准min\_support和confidence**

挖掘出频繁项后进行规则生产和选取

- 有监督关联规则

适用于所有的discriminative pattern, beam search/DFS/BFS等方法，都会生成大量的冗余项（local discriminative pattern），可以用一些方法来去除冗余项，在挖掘中或是挖掘后，得到global discriminative pattern

```
# BFS and DFS basic code

graph = {
    'A': ['B','C'],
    'B': ['A','C','D'],
    'C': ['A','B','D','E'],
    'D': ['B','C','E','F'],
    'E': ['C','D'],
    'F': ['D']
}

def BFS(graph, start):
    queue = []
    queue.append(start)
    seen = set()
    seen.add(start)

    while(len(queue) > 0):
        vertex = queue.pop(0)
        nodes = graph[vertex]
        for w in nodes:
            if w not in seen:
                queue.append(w)
                seen.add(w)
        print(vertex)

def DFS(graph, start):
    stack = []
    stack.append(start)
    seen = set()
    seen.add(start)

    while(len(stack) > 0):
        vertex = stack.pop()
```

那如果数据量较大并且商品种类在不止四种的情况下，这个轮询的方法肯定是不支持的。哪位了解这个复杂运算量的问题，就需要Apriori算法的支持了。

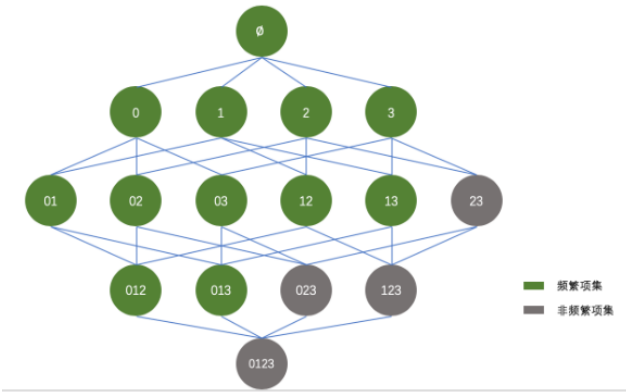
算法原理

如果某一个项集是频繁的，那么他的所有子集也是频繁的。

如果某一个项集是非频繁的，那么他的所有子集也是非频繁的。

在优化过后，根据Apriori算法。我们知道23的项集是非频繁的，那么他的所有超集也是非频繁的。

那么在计算的过程中，只要我们计算出23的支持度是不满足最小支持度的，他的超集肯定也是不满足的。



基础算法实现

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

# 发现频繁项集
def createC1(dataSet):
    C1=[]
    for transaction in dataSet:
        for item in transaction:
            if not [item] in C1:
                C1.append([item])
    C1.sort()
    return list(map(frozenset,C1))

def scanD(D,CK,minSupport):
    ssCnt = {}
    for tid in D:
        for can in CK:
            if can.issubset(tid):
                if not can in ssCnt:ssCnt[can]=1
                else:ssCnt[can]+=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

#频繁项集两两组合
def aprioriGen(Lk,k):
    retList=[]
    lenLk = len(Lk)
    for i in range(lenLk):
        for j in range(i+1,lenLk):
            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

def apriori(dataSet,minSupport=0.5):
    C1=createC1(dataSet)
    D=list(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

#规则计算的主函数
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):
                rulesFromConseq(freqSet,H1,supportData,bigRuleList,minConf)
            else:
                calcConf(freqSet,H1,supportData,bigRuleList,minConf)
    return bigRuleList

def calcConf(freqSet,H,supportData,brl,minConf=0.7):
    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
def rulesFromConseq(freqSet,H,supportData,brl,minConf=0.7):
    m = len(H[0])
    if (len(freqSet)>(m+1)):
        Hmp1 = aprioriGen(H,m+1)
        Hmp1 = calcConf(freqSet,Hmp1,supportData,brl,minConf)
        if (len(Hmp1)>1):
            rulesFromConseq(freqSet,Hmp1,supportData,brl,minConf)

if __name__=='__main__':
```

```
nodes = graph[vertex]
for w in nodes:
    if w not in seen:
        stack.append(w)
        seen.add(w)
    print(vertex)

BFS(graph, 'A')
print('new line')
DFS(graph, 'A')
```

挖掘出来的pattern存在统计意义上的不显著，需要做significant test。

- ▼ Association Rule
  - ▼ Unsupervised
    - frequent item-sets(patterns)
    - quality measure: min\_sup/confidence
  - ▼ Supervised
    - discriminative pattern
    - subgroup discovery (frequent item-sets → rules cover subgroup)
    - emerging patterns
    - contrast patterns

例子(additional info)

无监督关联规则	原理	特点	缺点
Apriori	多次扫描数据集，不断迭代寻找frequent item-sets	从k-item-sets迭代搜索k+1-item-sets	产生很多k-item-sets，多次扫描数据集来计算支持值
FP-Growth	用分枝的方法把frequent item-sets放入FP-Tree中并保留关联信息	保留所有item-sets的关联信息，缩减需要搜索的数据	数据集很大时构建FP-Tree会很耗时
Tree-Projection	建立字典树，用DFS/BFS方式进行搜索	只有部分包含frequent item-sets的交易被扫描，会搜索的更快	字典树的不同表达方式在memory小号上会有瓶颈
COFI			
TM			
P-Mine			
LP-Growth			
Can-Mining			
EXTRACT			
ECLaT			

无监督关联规则	原理	特点	缺点
Apriori_SD(Exhaustive)	基于Apriori搜索包含Label的规则，最后根据WRAcc选择最优的N best rule	生成更晓得谷子饿集合WRAcc	基于Apriori需要多次进行扫描数据
SMP(Exhaustive)			
DDPMine(Exhaustive)			
CIMCP(Top-ranking)			
SSDP(Top-ranking)			
CN2-SD(Heuristic)			
SDIGA(Heuristic)			
NMEEF-SD(Heuristic)			
GSD(Global)			
DSSD(Global)			
PLCM/MT-Closed/PARAMiner(Global)			

高数据维度的挖掘方法(additional info)

Carpenter, Cobbler, TD-close, TTD-close, Farmer, TopArgs, PARMA, Map-reduce related algorithms, gpu-related algorithms

统计显著性检验(additional info)

方法	multiple hypothesis testing	原理
FastWY	Westfall-Young procedure	效率低，用随机排序的数据集来寻找adjusted significance level
Westfall-Young light	Westfall-Young procedure	用branch-and-bound找到所有显著的pattern，时间和内存损耗优于FastWY
LAMP	LAMP	排除无意义的非平凡项集，耗时少，适合大数据
FACS	Tarone's testability criterion	调整显著水平，用CMH tes测试discriminative pattern和label的条件关联性

跨特征俩方关联规则联邦方案

详细步骤

1. 假设俩个参与方A/B和一个中间方

参数：

最小支持度（min\_support），最小置信度（min\_confidence），最大关联规则的长度（关联规则中频繁项集的长度）max\_len



```
dataSet=loadDataSet()
L, supportData=apriori(dataSet)
rules = generateRules(L, supportData, minConf=0.7)
```

## 从频繁项集中挖掘关联规则

https://ailearning.apachecn.org/##docs/ml/11

要找到关联规则，首先要从一个频繁项集开始。我们知道集合中的元素是不重复的，但我们想知道基于这些元素能否获得其他内容。某个元素或者某个元素集合可能会推导出另一个元素。

从杂货店的例子来举例说明，如果有一个频繁项集是豆奶和莴苣。那么就可能有一条关联规则是豆奶→莴苣，这就说明有人买了豆奶就有较大的可能性会去买莴苣。但是一旦条件反过来就不是永远都会成立。

频繁项集的量化定义就是满足最小支持度要求。

关联规则的量化指标就是可信度。

## 计算可信度/递归计算频繁项集的规则/生成关联规则代码实现

```
# 计算可信度 (confidence)
def calcConf(freqSet, H
, supportData, brl, minConf=0.7):
    """calcConf (对两个元素的频繁项, 计算可信度, 例如:  {1,2}/{1} 或者 {1,2}/{2} 看是否满足条件)

    Args:
        freqSet 频繁项集中的元素, 例如: frozenset([1, 3])
        H 频繁项集中的元素的集合, 例如: [frozenset([1]), frozenset([3])]
        supportData 所有元素的支持度的字典
        brl 关联规则列表的空数组
        minConf 最小可信度

    Returns:
        prunedH 记录 可信度大于阈值的集合

    """
    # 记录可信度大于最小可信度 (minConf) 的集合
    prunedH = []
    for conseq in H: # 假设 freqSet = frozenset([1, 3]), H = [frozenset([1]), frozenset([3])]
        # print 'confData=', freqSet, H, conseq, freqSet-conseq
        conf = supportData[freqSet]/supportData[freqSet-conseq]
        # 支持度定义: a -> b = support(a | b) / support(a). 假设 freqSet = frozenset([1, 3]), conseq = [frozenset([1])], 那么 frozenset([1]) 至 frozenset([3]) 的可信度为 = sup
        if conf >= minConf:
            # 只要买了 freqSet-conseq 集合, 一定会买 conseq 集合 (freqSet-conseq 集合和 conseq 集合是主集)
            print (freqSet-conseq, '->', conseq, 'conf:', conf)
            brl.append((freqSet-conseq, conseq, conf))
            prunedH.append(conseq)

    return prunedH

# 递归计算频繁项集的规则
def rulesFromConseq(freqSet, H, supportData, brl, minConf=0.7):
    """rulesFromConseq

    Args:
        freqSet 频繁项集中的元素, 例如: frozenset([2, 3, 5])
        H 频繁项集中的元素的集合, 例如: [frozenset([2]), frozenset([3]), frozenset([5])]
        supportData 所有元素的支持度的字典
        brl 关联规则列表的数组
        minConf 最小可信度

    """
    # H[0] 是 freqSet 的元素组合的第一个元素, 并且 H 中所有元素的长度都一样, 长度由 aprioriGen(H, m+1) 这里的 m+1 来定的, r+1。
    # 该函数递归时, H[0] 的长度从 1 开始增长 1 2 3 ...
    # 假设 freqSet = frozenset([2, 3, 5]), H = [frozenset([2]), frozenset([3]), frozenset([5])]
    # 那么 m = len(H[0]) 的递归的值依次为 1 2
    # 在 m = 2 时, 跳出该递归。假设再递归一次, 那么 H[0] = frozenset([2, 3, 5]), freqSet = frozenset([2, 3, 5]), 那么现在需要求出 frozenset([1]) -> frozenset([3]) 的可信度和 frozenset([3]) -> fr
    m = len(H[0])
    if (len(freqSet) > (m + 1)):
        print ('freqSet*****', len(freqSet), m + 1, freqSet, H, H[0])
        # 生成 m+1 个长度的所有可能的 H 中的组合, 假设 H = [frozenset([2]), frozenset([3]), frozenset([5])]
        # 第一次递归调用时生成 [frozenset([2, 3]), frozenset([2, 5]), frozenset([3, 5])]
        # 第二次 ... 没有第二次, 递归条件判断时已经退出了
        Hmp1 = aprioriGen(H, m+1)
        # 返回可信度大于最小可信度的集合
        Hmp1 = calcConf(freqSet, Hmp1, supportData, brl, minConf)
        print ('Hmp1=', Hmp1)
        print ('len(Hmp1)=', len(Hmp1), 'len(freqSet)=', len(freqSet))
        # 计算可信度后, 还有数据大于最小可信度的话, 那么继续递归调用, 否则跳出递归
        if (len(Hmp1) > 1):
            print ('-----', Hmp1)
            # print len(freqSet), len(Hmp1[0]) + 1
            rulesFromConseq(freqSet, Hmp1, supportData, brl, minConf)

# 生成关联规则
def generateRules(L, supportData, minConf=0.7):
    """generateRules

    Args:
        L 频繁项集列表
        supportData 频繁项集支持度的字典
        minConf 最小置信度

    Returns:
        bigRuleList 可信度规则列表 (关于 (A->B+置信度) 3个字段的组合)

    """
    bigRuleList = []
    # 假设 L = [[frozenset([1]), frozenset([3]), frozenset([2]), frozenset([5])], [frozenset([1, 3]), frozenset([2, 5]), frozenset([2, 3]), frozenset([3, 5])], [frozenset([
    for i in range(1, len(L)):
        # 获取频繁项集中每个组合的所有元素
        for freqSet in L[i]:
            # 假设: freqSet= frozenset([1, 3]), H1=[frozenset([1]), frozenset([3])]
            # 组合总的元素并遍历子元素, 并转化为 frozenset 集合, 再存放到 list 列表中
            H1 = [frozenset([item]) for item in freqSet]
            # 2 个的组合, 走 else, 2 个以上的组合, 走 if
            if (i > 1):
                rulesFromConseq(freqSet, H1, supportData, bigRuleList, minConf)
            else:
                calcConf(freqSet, H1, supportData, bigRuleList, minConf)
    return bigRuleList
```

## Apriori支持度实现

2. 跨特征场景下，每个参与方可先从本地使用FP-Growth算法去得到满足参数条件的频繁项集

$A\ frequent\ itemsets = \{(x_1, x_2) : \{support = 0.5\}, (x_1) : \{support = 0.6\} \dots\}$

$B\ frequent\ itemsets = \{(y_1, y_2) : \{support = 0.43\}, (y_1) : \{support = 0.5\} \dots\}$

3. A和B各自将本方的frequent\_item-sets的key传给对方。

A将 $[(x_1, x_2), (x_1), \dots]$ 传给B

B将 $[(y_1, y_2), (y_1), \dots]$ 传给A

那么两方都知道了彼此的频繁项集但并不知道频繁项集的支持度

4. 针对A和B俩方的频繁项集进行循环，

从A中抽取一个频繁项集比如 $ft_1 = (x_1, x_2)$

从B中抽取一个频繁项集比如 $ft_2 = (y_1)$

利用多方安全计算MPC的矩阵乘法，A和B都得到联邦频繁项集 $(x_1, x_2, y_1)$ 的支持度为sup

5. A计算sup是否大于min\_support，若大于

计算 $(x_1, x_2) \rightarrow (y_1)$ 的置信度，若大于min\_confidence

且 $len(x_1, x_2, y_1) \leq max(len)$

那么 $(x_1, x_2) \rightarrow (y_1)$ 为一条满足参数条件的关联规则

6. B计算sup是否大于min\_support，若大于

计算 $(y_1) \rightarrow (x_1, x_2)$ 的置信度，若大于min\_confidence

且 $len(x_1, x_2, y_1) \leq max(len)$

那么 $(y_1) \rightarrow (x_1, x_2)$ 为一条满足参数条件的关联规则

7. 对A和B的所有繁琐项集俩俩重复步骤456，找出所有满足条件的联邦关联规则，最后输出

## Association rule on vertically partitioned datasets

### 基于俩方的方案

俩方样本量一样的 $n$

步骤：

1. 参与方用FP-Growth算法得到各自frequent item-sets和对应的support值。

$Im(item, item_{sup})$

2. 双方需要对在随即种子以及随即数的生成方式上达成一致，

另双方可得到相同的随即矩阵 $C$ 大小为 $n * m$

3. 从 $Im_A$ 中选取特征 $X$ ，从 $Im_B$ 中选取特征 $Y$ 。 $A$ 生成随机数 $R_{n+1}$ ， $B$ 生成随机数 $R_{r+1}$ 。

•  $A$ 计算 $X' = X + C * R$ 并将 $X'$ 发送给 $B$

•  $B$ 接收 $X'$ 并计算 $S = X' * Y$ 和 $v = C * X + R'$ ，然后将 $S$ 和 $v$ 发送给 $A$

•  $A$ 接收 $v, S$ ，计算

$temp = S - R * v$ 和

$sub_R = [R_1 + R_2 + \dots + R_{n/r}, R_{n/(r+1)}, R_{n/(r+2)} + \dots + R_{2n/r}, \dots]$

将 $temp, sub_R$ 发送给 $B$

•  $B$ 接收 $temp, sub_R$ ，然后计算

$sup(XY) = temp - sub_R * R'$

将计算结果发送给 $A$ 。

$B$ 计算 $conf(Y \rightarrow X) = \frac{sup(XY)}{sup(Y)}$

•  $A$ 接收 $sup(XY)$ 并计算 $conf(X \rightarrow Y) = \frac{sup(XY)}{sup(X)}$ 是否满足1和2


4. 重复步骤三中的所有数据特征

### 基于多方的方案（Not solving yet）

...

apriori支持度实现\_ | 小目标 | 的博客-CSDN博客\_apriori最小支持度

算法思想 该算法的基本思想 是:首先找出所有的频集, 这些项集出现的频繁性至少和预定义的最小支持度一样。然后由频集产生强关联规则, 这些规则必须满足最小支持度和最小可信度。然后使用第1步找到的频集产生期望的规

 [https://blog.csdn.net/qq\\_43073162/article/details/106024766](https://blog.csdn.net/qq_43073162/article/details/106024766)

原创

## 基础算法思想

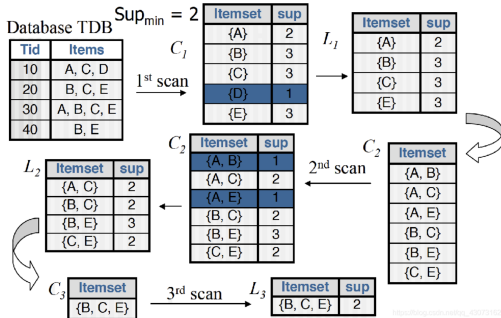
首先找出所有的频集, 这些项集出现的频繁性至少和预定义的最小支持度一样。

然后由频集产生强关联规则, **这些规则必须满足最小支持度和最小可信度。**

然后再使用第一部找到的频集产生期望的规则, 产生只包含集合的项的所有规则。其中每一条规则的右边只有一项, 这里采用的是中规则定义。

一旦这些规则被生成, 那没有那些大于给定的最小可信度的规则才能被留下来。

为了生成所有频集, 使用了递归方法。



```
##*- coding: UTF-8 -*-

# 加载数据
def loadDataSet():
    return [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5], [3, 1, 5, 2], [1, 2, 3]]

# 将所有元素转换为frozenset型字典, 存放在列表中
def createC1(dataSet):
    C1 = []
    for transaction in dataSet:
        for item in transaction:
            if not [item] in C1:
                C1.append([item])
    C1.sort()
    return list(map(frozenset, C1))

# 过滤掉支持度不符合的集合 即剪枝
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
    retList = []
    supportData = {}
    for key in ssCnt:
        support = ssCnt[key] / len(D)
        if support >= minSupport:
            retList.append(key)
            supportData[key] = support
    return retList, supportData

# 生成所有可以组合的集合
# 频繁项集列表Lk
# 连接后项集元素个数k [frozenset({2, 3}), frozenset({3, 5})] -> [frozenset({2, 3, 5})]
def aprioriGen(Lk, k):
    retList = []
    lenLk = len(Lk)
    for i in range(lenLk):
        for j in range(i + 1, lenLk):
            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

# 调用上述函数
# 返回 所有满足大于阈值的组合 集合支持度列表
def apriori(dataSet, minSupport):
    D = list(map(set, dataSet))
    C1 = createC1(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)
        L.append(Lk)
        supportData.update(supK)
        k = k + 1
    support = {}
    for i in L[k - 3]:
        support[i] = supportData.get(i)
    return support

if __name__ == '__main__':
    while int(input('请输入序号: 1 测试 0 退出\n')):
        print('输入支持度(0->1)')
        support = float(input())
        print(support)
        supp = apriori(loadDataSet(), support)
        print(supp)
```

输入支持度(0->1)


0.777

{frozenset({3}): 0.8333333333333334, frozenset({2}): 0.8333333333333334}

# Apriori Algorithm in Machine Learning | Association Rule Mining

Apriori Algorithm | Apriori Algorithm In Machine Learning | Association Rule Mining | Simplilearn

Apriori algorithm is a popular machine learning technique used for building recommendation systems. This video will make you understand what recommender syst...

 <https://www.youtube.com/watch?v=wryiEKQ0XhQ>



## Recommender System

predicts users interests and recommends products that the users may be interested in.

Gathers data from:

- Explicit ratings that users provide
- Implicit search engine queries and purchase histories
- Other source of knowledge about the users or items

is an information filtering techniques that provides users with recommendations for which they might be interested in.

purpose:

- Prediction perception
- Conversion perspective
- Recommendation perspective
- Interaction perspective
- Retrival perspective

## User-based nearest neighbour

Recommends item by finding users similar to the active user

Measure user similarity: **Pearson Correlation**

- A measure of how strong a relationship is between 2 variables.
- Degree of linearity can be determined using Pearson Correlation
- it determines linear components of association between 2 continuous variables.

It has no assumptions about linearity

Pearson's correlation is not used to determine the strength

$$sim(a,b)=\frac{\sum_{p\in P}(r_{a,p}-\bar{r}_a)(r_{b,p}-\bar{r}_b)}{\sqrt{\sum_{p\in P}(r_{a,p}-\bar{r}_a)^2}\sqrt{\sum_{p\in P}(r_{b,p}-\bar{r}_b)^2}}$$

- $a, b$ : users
- $r_{a,p}$ : rating of user  $a$  for item  $p$
- $P$ : set of items, rated both by  $a$  and  $b$
- Possible similarity values between  $-1$  and  $1$  (highly similar)

## item-based nearest neighbour

is easy to scale

can be computed offline and served without constant re-training

can be implemented best through KNN model

similarity is measured based on 2 algorithm:

### Cosine similarity

- produces better results in item-to-item filtering
- ratings are seen as vectors in n-dimensional space
- similarity is calculated based on the angle between the vectors

$$sim(\vec{a},\vec{b})=\frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}$$

### Adjusted Cosine Similarity

- takes average user ratings into account
- Transforms the original ratings

$$sim(a,b)=\frac{\sum_{p\in P}(r_{a,p}-\bar{r}_a)(r_{b,p}-\bar{r}_b)}{\sqrt{\sum_{p\in P}(r_{a,p}-\bar{r}_a)^2}\sqrt{\sum_{p\in P}(r_{b,p}-\bar{r}_b)^2}}$$

## Association rule mining

uses machine learning modes to analyse data for patterns or co-occurrence in a database.

### Support:

- indicates how frequently the items appear in the data
- provides fractions of transactions that contain  $X$  and  $Y$

### Confidence:

- indicates the number of times the if-then statements are found true
- indicates how often  $X$  and  $Y$  occur together, given the no. of times  $X$  occurs

### Lift:

- compare the actual confidence with the expected confidence
- indicates the strength of a rule over the random co-occurrence of  $X$  and  $Y$

$$Y = \frac{\sigma(x \cup y)}{\sigma(x) * \sigma(y)}$$

Apriori Algorithm

uses frequent item-sets to generate association rules

support value of frequent item-sets is greater than the threshold value

The algorithm reduces the number of candidates being considered by only exploring the item-sets whose support count is greater than the minimum support count.

使用FP-Growth算法来高效发现频繁项集

https://ailearning.apachecn.org/##docs/ml/12

一种发现频繁项集的算法。

基于Apriori算法，但是数据结构不同，使用FPTree的数据结构来储存集合。

FP-growth算法理解和实现\_木百栢的博客-CSDN博客\_fp-growth算法  
FP-growth(Frequent Pattern Tree, 频繁模式树),是韩家炜老师提出的挖掘频繁项集的方法，是将数据集存储在一个特定的称作FP树的结构之后发现频繁项集或频繁项对，即常在一块出现的元素项的集合FP树。FP-growth算法比

https://blog.csdn.net/baixiangxue/article/details/80335469



FP-Growth算法比Apriori算法效率更高，在整个算法执行过程中，只需要遍历数据集俩次，就能够完成频繁模式发现，基本的操作就是：

- 1. 构建FP-Tree
- 2. 从FP-Tree中挖掘频繁项集

FP-Tree结构

```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue          # 节点名称
        self.count = numOccur          # 节点出现次数
        self.nodeLink = None           # 不同项集的相同项通过nodeLink连接在一起
        # needs to be updated
        self.parent = parentNode        # 指向父节点
        self.children = {}              # 存储叶子节点
```

步骤1:

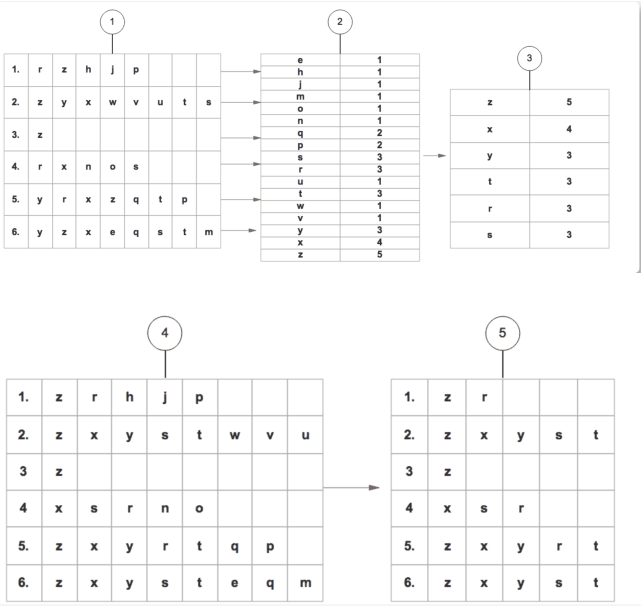
遍历所有的数据集，计算所有项的支持度

丢弃非频繁的项

基于支持度降序排序所有的项

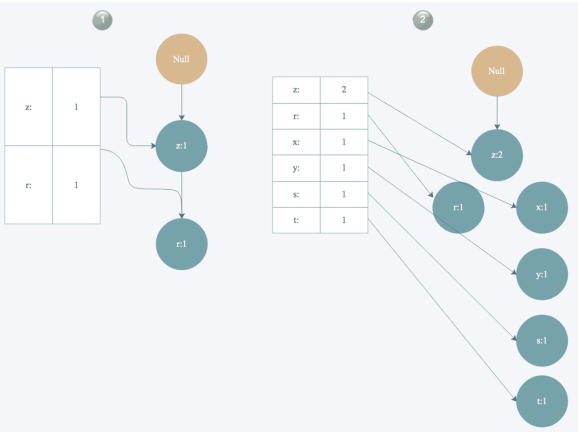
所有数据集按照得到的顺序重新整理

重新整理完之后，丢弃每个集合末尾非频繁的项

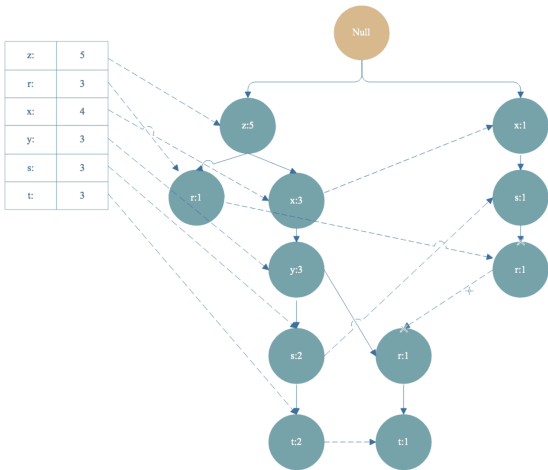


步骤2:

读取每个集合插入FP-Tree中，同时用一个头部链表数据结构维护不同集合的相同项



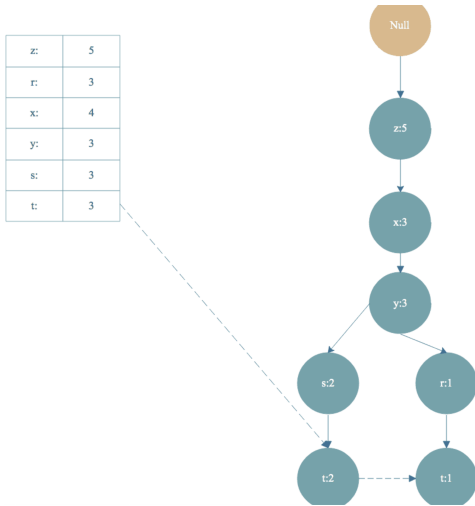
得到最终的树



步骤3:

对头部链表进行降序排序

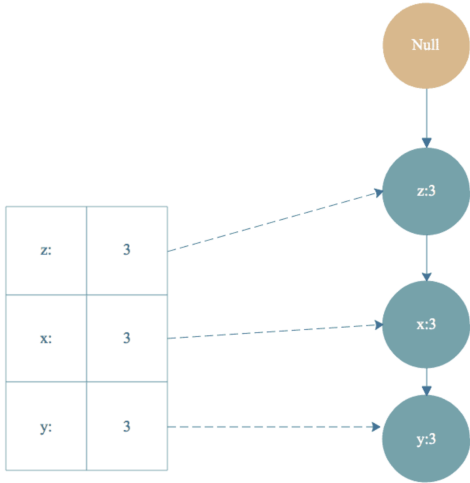
对头部链表节点从小到大遍历，得到条件模式基，同时获得一个频繁项集



去掉FP-Tree中的t节点，得到条件模式基于<左边路径，右边值数>。条件模式基的值取决于末尾节点t，因为t的出现次数最小，一个频繁项集的支持度由支持度最小的项决定。所以t节点的条件模式基的值可以理解为对于以t节点为末尾的前缀路径出现的次数。

条件模式继续构造条件FP-Tree，得到频繁项集，和之前的频繁项组合起来，这是一个递归遍历头部链表生成FP-Tree的过程，递归截止的条件是生成的FP-Tree的头部链表为空集。

计算支持度，去除非频繁项集，集合按照支持度降序排序，重复上面构造FP-Tree的步骤，最后得到：



### 代码解析

```
# FP_tree的类定义
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue #节点名字
        self.count = numOccur #节点计数值
        self.nodeLink = None #用于链接相似的元素项
        self.parent = parentNode #needs to be updated
        self.children = {} #子节点

    def inc(self, numOccur):
        '''
        对count变量增加给定值
        '''
        self.count += numOccur

    def disp(self, ind=1):
        '''
        将树以文本形式展示
        '''
        print (' '*ind, self.name, ' ', self.count)
        for child in self.children.values():
            child.disp(ind+1)

# FP_tree构造函数
def createTree(dataSet, minSup=1):
    '''
```

```

创建FP树
'''
headerTable = {}
#第一次扫描数据集
for trans in dataSet:#计算item出现频数
    for item in trans:
        headerTable[item] = headerTable.get(item, 0) + dataSet[trans]
headerTable = {k:v for k,v in headerTable.items() if v >= minSup}
freqItemSet = set(headerTable.keys())
#print ('freqItemSet: ',freqItemSet)
if len(freqItemSet) == 0: return None, None #如果没有元素项满足要求, 则退出
for k in headerTable:
    headerTable[k] = [headerTable[k], None] #初始化headerTable
#print ('headerTable: ',headerTable)
#第二次扫描数据集
retTree = treeNode('Null Set', 1, None) #创建树
for tranSet, count in dataSet.items():
    localD = {}
    for item in tranSet: #put transaction items in order
        if item in freqItemSet:
            localD[item] = headerTable[item][0]
    if len(localD) > 0:
        orderedItems = [v[0] for v in sorted(localD.items(), key=lambda p: p[1], reverse=True)]
        updateTree(orderedItems, retTree, headerTable, count)#将排序后的item集合填充的树中
return retTree, headerTable #返回树型结构和头指针表

def updateTree(items, inTree, headerTable, count):
    if items[0] in inTree.children:#检查第一个元素项是否作为子节点存在
        inTree.children[items[0]].inc(count) #存在, 更新计数
    else: #不存在, 创建一个新的treeNode, 将其作为一个新的子节点加入其中
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        if headerTable[items[0]][1] == None: #更新头指针表
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], inTree.children[items[0]])
    if len(items) > 1:#不断迭代调用自身, 每次调用都会删掉列表中的第一个元素
        updateTree(items[1:], inTree.children[items[0]], headerTable, count)

def updateHeader(nodeToTest, targetNode):
    '''
    this version does not use recursion
    Do not use recursion to traverse a linked list!
    更新头指针表, 确保节点链接指向树中该元素项的每一个实例
    '''
    while (nodeToTest.nodeLink != None):
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode

# 抽取条件模式基
def ascendTree(leafNode, prefixPath): #迭代上溯整棵棵树
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)

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

# 递归查找频繁项集
def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: p[1][0])]# 1.排序头指针表
    for basePat in bigL: #从头指针表的底端开始
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        print ('finalFrequent Item: ',newFreqSet) #添加的频繁项列表
        freqItemList.append(newFreqSet)
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        print ('condPattBases :',basePat, condPattBases)
        # 2.从条件模式基创建条件FP树
        myCondTree, myHead = createTree(condPattBases, minSup)
    #
    print ('head from conditional tree: ', myHead)
    if myHead != None: # 3.挖掘条件FP树
        print ('conditional tree for: ',newFreqSet)
        myCondTree.disp(1)
        mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)

# 测试
def loadSimpDat():
    simpDat = [
        ['I1', 'I2', 'I5'],
        ['I2', 'I4'],
        ['I2', 'I3'],
        ['I1', 'I2', 'I4'],
        ['I1', 'I3'],
        ['I2', 'I3'],
        ['I1', 'I3'],
        ['I1', 'I2', 'I3', 'I5'],
        ['I1', 'I2', 'I3']
    ]
    return simpDat

'''def loadSimpDat():
    simpDat = [['r', 'z', 'h', 'j', 'p'],
        ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
        ['z'],
        ['r', 'x', 'n', 'o', 's'],
        # ['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)] = retDict.get(frozenset(trans), 0) + 1 #若无相同事项, 则为1; 若有相同事项, 则加1
    return retDict

minSup = 2
simpDat = loadSimpDat()
initSet = createInitSet(simpDat)
myFPtree, myHeaderTab = createTree(initSet, minSup)
myFPtree.disp()
myFreqList = []
mineTree(myFPtree, myHeaderTab, minSup, set([]), myFreqList)

myFreqList

```