关联规则FP-growth算法实现

星点学习 星点学习 2018-06-12

子曰：温故而知新，可以为师矣

孔子说，温习旧知识从而得到新的理解和体会，这样的人可以成为老师了

老师是做不了，不过温习旧知识还是要滴

认知心理学告诉我们

人的知识记忆呈曲线下降

合理安排时间复习很关键

能够将短期记忆转化为长期记忆

说了这么多，就是怕你忘记上一次的数据挖掘Apriori算法实现

快快点进去温故知新吧

浏览完上次Apriori算法，这次也来学一个同样处理关联规则的经典算法

算法介绍

> FP-Growth算法是韩嘉炜等人在2000年提出的关联分析算法，它采取如下分治策略：将提供频繁项集的数据库压缩到一棵频繁模式树（FP-tree），但仍保留项集关联信息

上一次我们学习Apriori算法，其基本思路是找出1-频繁集，接着找2-频繁集，直至找出所有的频繁集

但是它需要多次扫描数据库，而且产生大量的候选频繁集，时间复杂度和空间复杂度都比较大

这次我们来学习一下效率更高FP-growth 算法

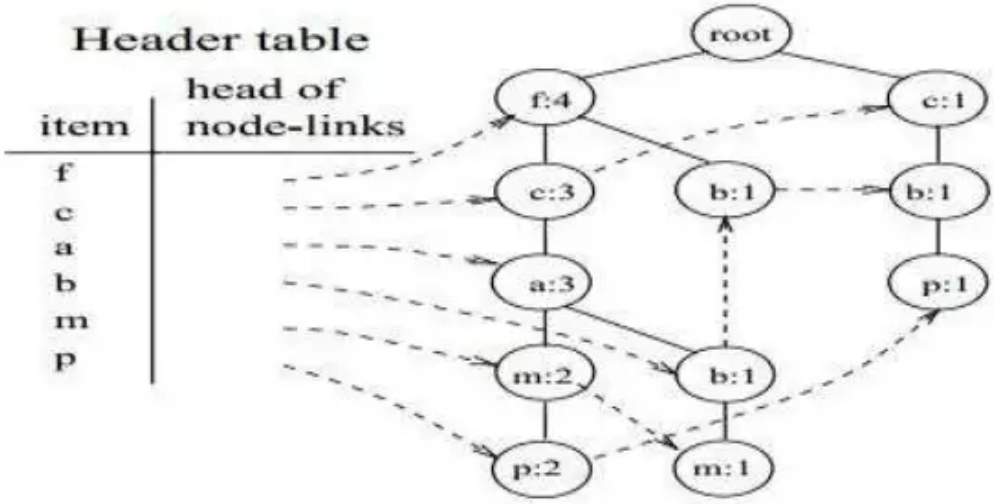
概念

又有几个概念需要了解一下

FP-Tree

FP-Growth算法 将交易集D中的所有信息都压缩到一棵树上

这就是 频繁模式树(frequent pattern) FP-Tree, 包括一个频繁项组成的头表(headerTable)



频繁项

单个项目的支持度大于或等于最小支持度则称为频繁项

频繁项头表

频繁项头表(headerTable) 由两个域组成，分别是名称name和指针node_link

指针node_link指向FP-Tree中具有与该表项相同name的第一个结点

如上图

项前缀子树

项前缀子树 可简单理解为 FP-Tree的节点，每个节点即为一颗子树

每个节点除了书中讲的3个域，分别是 name, count, node_link

还有 fa_point, child 域

其中 name是结点名字，count为交易个数， node_link指针域，

node_link 指向了下一个具有同样name域的节点，否则为None

fa_point也是指针，指向了父节点

child 则是一个字典，以key-value的形式存储子节点

Python代码如下

```
class Tree_node:
   def __init__(self, name, count, fa_point):
      self.name = name
      self.count = count
      self.fa_point = fa_point
      self.node_link = None
      self.child = {}
```

算法步骤

(1) 根据一个输入的交易记录集建立一颗FP-Tree
(2) 利用FP-Tree 来产生频繁集

构建FP-Tree

1 首先我们将数据库D扫描一遍，得到每个频繁项的集合L和每个频繁项的支持度 （L按支持度降序排序）

例如，我们有数据库D如下

Tid	商品Items	Tid	商品Items
1	bread cream milk tea	6	bread tea
2	bread cream milk	7	beer milk tea
3	cake milk	8	bread tea
4	milk tea	9	bread cream milk tea
5	bread cake milk	10	bread milk tea

按最小支持度min_sup为3的标准扫描后得到一个频繁项列表L

如下

项目	支持度
milk	8
bread	7
tea	7
cream	3

2 我们将数据库D中的项目按频繁项列表L上顺序排序

这里先预处理一下，将数据集中相同项累加，用count来保存相同项集的个数

并将项集中非频繁项剔除掉

结果如下

```
count    item-set
1       ['milk', 'bread', 'cream']
2       ['tea', 'bread']
1       ['milk', 'tea', 'bread']
1       ['milk', 'tea']
2       ['milk', 'tea', 'bread', 'cream']
1       ['milk', 'bread']
1       ['milk', 'tea']
1       ['milk']
```

3 接着创建一个root节点，作为树的根节点

用项集元素来更新或建立树的节点

例如：用第一条记录来更新树节点

```
1 ['milk', 'bread', 'cream']
```

当前树只有一个根节点root

所以root节点下不存在milk子节点

所以创建一个新节点milk,连上root

接着以milk为根，继续以上步骤，直到记录遍历完毕

得到

```
root -> milk:1 -> bread:1 -> cream:1
```

接着第二条记录

得到

```
root -> milk:1 -> bread:1 -> cream:1
     -> tea:2 -> bread:2
```

遍历所有记录，得到FP-Tree 如下

```
root -> milk:8 -> bread:2 -> cream:1
               -> tea:5  -> bread:2 -> cream:2
     -> tea:2 -> bread:2
```

产生频繁集

有了上面的操作后，我们得到一个头表和一个频繁模式树

我们已经把所有信息都压缩在一个FP-Tree上了

现在可以将所有的频繁集挖出来了

挖掘过程

由于操作的是一颗多叉树，所以挖掘信息是一个递归的过程

从底向上循环取出当前头表的项item，利用节点的node_link域找到树节点的位置

找到树中所有关于item的路径，简化成数据集形式

再生成关于item的FP-Tree

递归搜索此路径，组合成频繁集，其支持度为当前头表item项的支持度

结束

至此，就得到了所有的频繁集以及支持度

样例产生的频繁集如下

```
[(set(['cream']),        3),
 (set(['bread']),        7),
 (set(['tea']),         7),
 (set(['milk']),         8),
 (set(['cream', 'milk']),  3),
 (set(['bread', 'cream']), 3),
 (set(['bread', 'tea']),   5),
 (set(['bread', 'milk']),  5),
 (set(['milk', 'tea']),    5),
 (set(['bread', 'cream', 'milk']), 3),
 (set(['bread', 'milk', 'tea']),  3)]
```

FP-growth 算法实现

实现代码

```
#coding=utf-8
import pprint
class Tree_node:
    def __init__(self, name, count, fa_point):
        self.name = name
        self.count = count
        self.fa_point = fa_point
        self.node_link = None
        self.child = {}
    def show(self):
        '''
        打印整课树
        :return:
        '''
        pprint.pprint(self.name + " " + str(self.count))
        for item in self.child.values():
            item.show()
        print("==================")
def data_set(D):
    '''
    将交易集转换成 key-value 形式 方便处理
    :param D: 交易集
    :return: key-value形式交易集
    '''
    dataSet = {}
    for t in D:
        key = frozenset(t)
        if key in dataSet.keys():
```

```
            dataSet[key] += 1
        else:
            dataSet[key] = 1
    return dataSet
def createHeaderTable(dataSet, min_sup):
    '''
    根据交易集和最小支持度，建立头表
    :param dataSet: 交易集
    :param min_sup: 最小支持度
    :return:  头表
    '''
    headerTable = {}
    for trans in dataSet:
        for item in trans:
            headerTable[item] = headerTable.get(item,0) + dataSet[trans]
            # get(key, default)
    for key in headerTable.keys():
        if headerTable[key] < min_sup:
            del(headerTable[key])
    for key in headerTable:
        headerTable[key] = [headerTable[key], None]  # [频度， node_link]
    return headerTable
def create(dataSet, min_sup):
    '''
    建立一个FP-Tree
    :param dataSet: 交易集
    :param min_sup: 最小支持度
    :return: FP树，头表
    '''
    headerTable = createHeaderTable(dataSet, min_sup) # 建立头表
    # pprint.pprint(headerTable)
    root = Tree_node("root", 0, None)
    frequent_item = set(headerTable.keys())
    if len(frequent_item) == 0: return None,None
    for trans, count in dataSet.items():
        temp_D = {}
        for item in trans:
            if item in frequent_item:
                temp_D[item] = headerTable[item][0]
        # pprint.pprint(temp_D)
        if temp_D:
            sortItem = [v[0] for v in sorted(temp_D.items(), key = lambda p:p[1], reverse = True)]
            # pprint.pprint(sortItem)
            print count,"\t\t\t",sortItem
            # 按照频繁项大小排序
            buildTree(root, headerTable, sortItem, count)
    return root,headerTable
def buildTree(root, headerTable, sortItem, count):
    '''
    用项集元素来更新或建立树的节点
    :param root: 根节点（父节点）
    :param headerTable: 头表
    :param sortItem: 排序后的项集
    :param count: 项集的次数
    :return:
    '''
    key  = sortItem[0]
    if key in root.child.keys():
        root.child[key].count += count           #增加计数
    else:
        root.child[key] = Tree_node(key, count, root)  #新节点
        if headerTable[key][1] == None:          # 头表未连接
            headerTable[key][1] = root.child[key]     # 将头表和树节点连接起来
        else:                           #用node_link 将相同的name的树节点连接起来
            cur_node = headerTable[key][1]         # headerTable[key][1] 中存储着树节点地址
            while (cur_node.node_link != None):
                cur_node = cur_node.node_link        # 指向下一节点
            cur_node.node_link = root.child[key]      # 连接新节点
    if len(sortItem) > 1:
```

```
            buildTree(root.child[key], headerTable, sortItem[1:], count) # 递归建树
    def findPath(table_item):
        '''
        根据表项寻找路径，并将路径组合成新的数据集
        :param table_item:
        :return:
        '''
        data_set = {}
        tree_node = table_item
        while tree_node != None:
            path = []
            father = tree_node.fa_point
            while father.fa_point != None:
                path.append(father.name)
                father = father.fa_point
            if path:
                pprint.pprint(path)
                data_set[frozenset(path)] = tree_node.count
            tree_node = tree_node.node_link
        return data_set
    def mining(root, headerTable, min_sup, fre_set, fre_list):
        '''
        从FP-Tree中挖掘出所有的频繁集
        :param root: FP树
        :param headerTable: 头表
        :param min_sup: 最小支持度
        :param fre_set: 频繁集
        :param fre_list: 频繁集列表
        :return:
        '''
        items = [v[0] for v in sorted(headerTable.items(),key=lambda p:p[1])] #头表降序排序
        # pprint.pprint(items)
        for item in items:
            newFreqSet = fre_set.copy()          # 深拷贝
            newFreqSet.add(item)                 #
            fre_list.append((newFreqSet, headerTable[item][0]))   # 表头元素的支持度 为当前 频繁集 的支持度
            # pprint.pprint(item)
            data_set = findPath(headerTable[item][1])         # 从表项得到一个新的数据集
            # pprint.pprint(data_set)
            tree,header = create(data_set, min_sup)          # 建立一个新的FP-Tree，头表
            if header != None:
                mining(tree, header, min_sup, newFreqSet, fre_list)   # 递归 挖掘
    if __name__ == '__main__':
    # 交易集
        D = [["bread","cream","milk","tea"],
             ["bread","milk","cream"],
             ["cake","milk"],
             ["tea","milk"],
             ["bread","cake","milk"],
             ["bread","tea"],
             ["bread","tea"],
             ["beer","milk","tea"],
             ["bread","cream","milk","tea"],
             ["bread","milk","tea"],
            ]
    # 最小支持度
        min_sup = 3
        dataSet = data_set(D)
        pprint.pprint(dataSet)
        root,headerTable = create(dataSet, min_sup)
        root.show()
        frequent_list = []
        mining(root, headerTable, min_sup, set([]), frequent_list)
        # pprint.pprint(frequent_list)
        pprint.pprint(sorted(frequent_list, key = lambda p:len(p[0])))
```