FP-growth高效率频繁项集发现

FP-growth高效频繁项集发现

FP-growth

算法优缺点：

优点：一般快于**Apriori**
缺点：实现比较困难，在某些数据上性能下降
适用数据类型：标称型数据

算法思想：

FP-growth算法是用来解决频繁项集发现问题的，这个问题再前面我们可以通过Apriori算法来解决，但是虽然利用Apriori原理加快了速度，仍旧是效率比较低的。FP-growth算法则可以解决这个问题。
FP-growth算法使用了频繁模式树（Frequent Pattern Tree）的数据结构。FP-tree是一种特殊的前缀树，由频繁项头表和项前缀树构成。所谓前缀树，是一种存储候选项集的数据结构，树的分支用项名标识，树的节点存储后缀项，路径表示项集。
FP-growth算法生成频繁项集相对Apriori生成频繁项集的主要好处就是速度快，能快到几个数量级；另一个好处就是用FP树存储数据可以减少存储空间，因为关联挖掘的数据集往往是重复性很高的，这就能带来很高的压缩比。

算法可以分成一下几个部分：

构建FP树

> 首先我们要统计出所有的元素的频度，删除不满足最小支持度的（Apriori原理）
> 然后我们要根据频度对所有的项集排序(保证我们的树是最小的)
> 最后根据排序的项集构建FP树

从FP树挖掘频繁项集：

> 生成条件模式基
> 生成条件FP树

算法的执行过程这篇文章有个很好的示例程序

函数：

`loadSimpDat()`
创建数据集
`createInitSet(dataSet)`
将数据集处理成字典的形式
`createTree(dataSet, minSup=1)`
创建FP树的主函数。首先生成单元素的频繁项，然后对每个项集进行以频繁项的频度为基准的排序。
`updateTree(items, inTree, headerTable, count)`
根据每一个项集和对应的频数，更新FP树。并同时建立表头
`updateHeader(nodeToTest, targetNode)`
当指针已经初始化的时候，调用这个函数把新的点加到链表的最后面
`ascendTree(leafNode, prefixPath)`
向上遍历移植到根节点，将经过的节点都加到前缀路径中，得到整条每个频繁项的前缀路径
`findPrefixPath(basePat, treeNode)`
生成条件模式基
`mineTree(inTree, headerTable, minSup, preFix, freqItemList)`
递归调用生成条件FP树和频繁项集。创建条件FP树的过程可以重用前面createTree的代码

```
1 #coding=utf-8
```

```
import time
class treeNode(object):
    """docstring for treeNode"""
    def __init__(self, nameValue, numOccur, parentNode):
        super(treeNode, self).__init__()
        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():
            child.disp(ind+1)
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
def createTree(dataSet, minSup=1):
    headerTable = {}
    #frequency of each item
    for trans in dataSet:
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]#some trans may same
    #remove items not meeting minSup
    for k in headerTable.keys():
        if headerTable[k] < minSup:
            del(headerTable[k])
    freqItemSet = set(headerTable.keys())
    if len(freqItemSet) == 0:#no frequent item
        return None, None
    for k in headerTable:#add a point field
        headerTable[k] = [headerTable[k], None]

    retTree = treeNode('Null set', 1, None)
    for tranSet, count in dataSet.items():
        localD = {}
        for item in tranSet:#把每一个项集的元素提取出来，并加上统计出来的频率
            if item in freqItemSet:
                localD[item] = headerTable[item][0]
        if len(localD) > 0:#排序，并更新树
            orderdItem = [v[0] for v in sorted(localD.items(),key=lambda p:p[1],reverse=True)]
            updateTree(orderdItem, retTree, headerTable, count)
    return retTree, headerTable
def updateTree(items, inTree, headerTable, count):
    #将新的节点加上来
    if items[0] in inTree.children:
        inTree.children[items[0]].inc(count)
    else:
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        #更新指针
        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:
        updateTree(items[1::],inTree.children[items[0]],headerTable,count)
def updateHeader(nodeToTest, targetNode):
    while nodeToTest.nodeLink != None:
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode

def ascendTree(leafNode, prefixPath): #ascends from leaf node to root
    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
def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: p[1])]#(sort header table)
    #print bigL
    for basePat in bigL:  #start from bottom of header table
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)
        print 'finalFrequent Item: ',newFreqSet    #append to set
        freqItemList.append(newFreqSet)
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        print 'condPattBases :',basePat, condPattBases
        #2. construct cond FP-tree from cond. pattern base
        myCondTree, myHead = createTree(condPattBases, minSup)
        print 'head from conditional tree: ', myHead
        if myHead != None: #3. mine cond. FP-tree
            print 'conditional tree for: ',newFreqSet
            myCondTree.disp(1)
            mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)
def main():
    if True:
        simpDat = loadSimpDat()
        initSet = createInitSet(simpDat)
        myFP, myHeadTable = createTree(initSet,3)
        myFP.disp()
        freqItems = []
        mineTree(myFP,myHeadTable,3,set([]),freqItems)
        print freqItems
    if False:
        t1 = time.clock()
        parsedDat = [line.split() for line in open('kosarak.dat').readlines()]
        initSet = createInitSet(parsedDat)
        myFP,myHeadTable = createTree(initSet,100000)
        myfreq = []
        mineTree(myFP,myHeadTable,100000,set([]),myfreq)
        t2 = time.clock()
        print 'time=', t2-t1
        print myfreq
if __name__ == '__main__':
    main()

```

使用FP算法对一个近100万行的数据进行分析，耗时不过十来秒：

```
time= 12.5076159007
[set(['1']), set(['1', '6']), set(['3']), set(['11', '3']), set(['11', '3', '6']), set(['3', '6']), set(['11']), set(['11', '6']), set(['6'])]
[Finished in 13.2s]
```


而如果采用Apriori的频繁集发现算法我跑了四分多种没出结果然后就强制关掉了。。。

```
[Finished in 246.3s]
```

事实证明这个算法确实能够提高数量级的速度啊。

来自为知笔记(Wiz)

上一篇：算法竞赛入门经典_第二章:循环结构程序设计_上机练习题_MyAnswer
下一篇：《数据结构与算法分析:C语言描述_原书第二版》CH3表、栈跟队列_reading notes