#-\*- coding:utf-8 –\*-

\_\_author\_\_ = *'Dodd'*

import fptree

import tree

class **FP\_Grow\_tree**:

def **\_\_init\_\_**(*self*,datas,a,support):

//调用fptree函数，传入数据集和支持度阈值，转入该函数

*self*.f=fptree.fptree(datas,support)

//将所有出现次数大于支持度阈值的项集联通他们出现的次数经过排序存入f.pretable={}中

*self*.f.fp\_tree()

//执行tree函数，传入 *self*.f.getRootTree()为一颗由全部数据构成的树，*self*.f.headtable为纵向头列表，还有支持度阈值和a *self*.f.tree=tree.tree(*self*.f.getRootTree(),*self*.f.headtable,support,a)

#print(a)

*self*.f.tree.FP\_growth(*self*.f.headnode,*self*.f.headtable)

pass

def **printfrequent**(*self*):

*self*.f.tree.printfrequent()

pass

#-\*- coding:utf-8 –\*-

\_\_author\_\_ = *'Dodd'*

import node

import tree

from operator import itemgetter, attrgetter

class **fptree**:

def **\_\_init\_\_**(*self*,datas,support):

*self*.pretable={}

*self*.datas=datas//数据集传入

*self*.support=support//支持度阈值传入

*self*.headnode=node.node(*'null'*,None)//创建一个节点作为头节点

*self*.tree=None

*self*.headtable={}

def **fp\_tree**(*self*):

*self*.pretable=*self*.getpretable()

//将所有出现次数大于支持度阈值的项集筛选出来

*self*.pretable=[i for i in *self*.pretable if i[1] >= *self*.support]

#print(self.pretable)

//该函数可以将数据整个存入pretable字典，字典中的value值为该字段出现的次数

def **getpretable**(*self*):

pretable={}

for t in *self*.datas:

for item in t://遍历数据集

//.setdefault(item,0)意思是遍历整个字典，如果能找到item则返回改key对应的值，若找不到，则添加key=item，value=0的新值，在这里用做将数据添加的操作

pretable.setdefault(item,0);

pretable[item]+=1

//sorted函数用于对元祖进行排序，pretable.items()将字典转换成元祖，key=itemgetter(1,0)代表先按照第二维排序，再按照第一维排序，reverse = True代表由大到小排列

return sorted(pretable.items(),key=itemgetter(1,0),reverse = True)

//将所有的数据压缩成一棵树，代码结构完美

def **getRootTree**(*self*):

nowheadtable={}

//遍历数据

for t in *self*.datas:

//每遍历完一条数据头结点变成最初的那个节点所以对于每条数据来说，她都是从最上方的头结点开始遍历的

headnode=*self*.headnode

//对于每一行数据都进行，遍历pretable字典的操作

for item in *self*.pretable:

//这个循环很关键，我的理解是将在这条数据中所有支持度大于阈值的项集按出现次数从大到小参与下面的运算

if item[0] in t:

//如果当前节点的孩子中已经有item[0]了，那让item[0]的出现次数加一，如果没有则给当前头结点添加item[0]节点，并且令item[0]节点变成头结点（优美迭代）

thenode=headnode.findchildnode(item[0])

if not thenode:

thenode=node.node(item[0],headnode)

headnode.child.append(thenode)

*self*.headtable.setdefault(item[0],thenode)

if item[0] in nowheadtable.keys():

nowheadtable[item[0]].next=thenode

nowheadtable[item[0]]=thenode

nowheadtable.setdefault(item[0],thenode)

thenode.count+=1

headnode=thenode

#print(headnode)

#print('fds')

return *self*.headnode

class **node**:

def **\_\_init\_\_**(*self*,name,parent):

*self*.name=name

*self*.parent=parent

*self*.child=[]

*self*.next=None

*self*.count=0

pass

//遍历所有孩子节点，若找到指定姓名的节点，则返回该节点，否则返回空

def **findchildnode**(*self*,item):

for nodes in *self*.child:

if nodes.name==item:

return nodes

pass

return None

pass

class **tree**:

frequent=[]

def **\_\_init\_\_**(*self*,headnode,headtable,support,a):

*self*.a=a

*self*.headnode=headnode

*self*.headtable=headtable

*self*.support=support

#print('ji:')

#print(a)

#tree.printTree(self.headnode)

#tree.printheadtable(self.headtable)

pass

def **printfrequent**(*self*):

y=sorted(tree.frequent,key=lambda x:x[1],reverse=True)

for x in y:

print(x)

pass

print(len(y))

def **FP\_growth**(*self*,headnode,headtable):

a=*self*.a

//调用checkTreeOneWay函数，并将当前树头结点传入

if tree.checkTreeOneWay(headnode):

//调用unit.generateCombination（也是一个自定义函数，见下文）

add=unit.generateCombination(headtable,a,*self*.support)

//将add加入最终的结果输出数组

if len(add)>0:

tree.frequent+=add

#print('frequent')

#print(tree.frequent)

pass

else:

for item in headtable:

#datas为条件模式基

//这里调用了另一个函数generateSubset,该函数和generateCombination作用其实是一致的，只不过在查询效率上较上面的慢一点，下面在介绍这个函数的时候会细说

datas=unit.generateSubset(headtable,item,*self*.a,tree.frequent)

if datas:

#print(item)

if item:

x=a[:]

x.append(item)

f=FP\_Grow\_tree.FP\_Grow\_tree(datas,x,*self*.support)

#print('----------------ddddd-')

#print(f.f.pretable)

for jix in f.f.pretable:

xx=a[:]

xx.append(item)

xx.append(jix[0])

tree.frequent.append((*","*.join(str(i) for i in xx),jix[1]))

pass

pass

pass

pass

//这个就是上面的那个判定函数了，这个函数的作用就是判断当前的树有没有两个或两个以上的分支，如果没有任何分支，这个树直接以队列的形式结束，那么调用generateCombination函数，如果存在两个以上分支，则调用generateSubset

def **checkTreeOneWay**(nodex):

nodesx=nodex

#print(nodesx)

while nodesx:

#print(nodesx)

if len(nodesx.child)>1:

return False

if len(nodesx.child)>0:

nodesx=nodesx.child[0]

if len(nodesx.child)==0:

break

nodesx=nodesx.child[0]

return True

//该函数是我为了体现基于置信度的频繁规则输出加入的，代码可能没有其他的优美简介，但是很容易看懂

def **printconfident**(*self*,con):

for i in tree.frequent:

a=i[0].split(*','*)

alen=len(a)

for j in tree.frequent:

flag=1

b=j[0].split(*','*)

blen=len(b)

if blen>alen:

str1=i[0]

str2=[]

for x in a:

if x not in b:

flag=0

if flag==1:

for x2 in a:

b.remove(x2)

str2=*","*.join(str(d) for d in b)

str2=str1+*'-->>'*+str2

num=float(j[1])/float(i[1])

tree.confident.append((str2,num))

File = open(*"hello2"*, *"w"*)

y=sorted(tree.confident,key=lambda x:x[1],reverse=True)

count=0

for x in y:

if x[1]>=con:

count+=1

print(x)

File.write(str(x) + *"\n"*)

pass

print(count)

File.write(str(count) + *"\n"*)

File.close()

//这个函数应该是测试使用的，在实际没有用到

def **printTree**(node):

if len(node.child)!=0:

print(node.name+str(node.count)+*'p '*+node.parent.name if node.parent else *'not'*)

for nodes in node.child:

tree.printTree(nodes)

pass

else:

print(node.name+str(node.count)+*'p '*+node.parent.name if node.parent else *'not'*)

print(*'--------------'*)

//同上

def **printheadtable**(headtable):

print(headtable)

for x in headtable:

print(headtable[x])

y=headtable[x]

i=0

print(x)

while y.next:

y=y.next

print(y)

i+=1

pass

print(i)

pass

def **generateCombination**(headtable,a,support):

lis=[]

lisc=[]

ans=[]

用逗号链接a中的个元素然后存入base

base=str(*","*.join(str(i) for i in a))

//如果纵向头表中项集次数大于支持度阈值，则将其加入lis列表

for x in headtable.keys():

if(headtable[x].count>=support):

lis.append(x)

lenlis=len(lis)

// combinations(iterable,r);创建一个迭代器，返回iterable中所有长度为r的子序列，返回的子序列中的项按输入iterable中的顺序排序，例如

for i in combinations([1, 2, 3], 2):

    print i

(1, 2)

(1, 3)

(2, 3)

---------------------------------------------------------------------------

for i in range(2,lenlis+1):

lisc+=list(combinations(lis,i))

#print(lisc)

st=None

for x in lisc:

//这个count好像是为了内存限制把，我没太发现这个变量什么关键的作用，各位如果发现了请给我说一声

count=999

st=base

if x:

for i in x:

st=st+*","*+i

count=min(count,headtable[i].count)

pass

if st:

ans.append((st,count))

return ans

//该函数就是找出当前频繁项的条件模式基

def **generateSubset**(headtable,item,a,frequent):

datas=[]

//首先找到定位到item这个频繁项

node=headtable[item]

//挨个遍历这个频繁项的各个子节点，至于这些子节点是怎么串联的，在fptree.Getroottree里有所定义

//if item[0] in nowheadtable.keys():

//nowheadtable[item[0]].next=thenode

//nowheadtable[item[0]]=thenode

----------------------------------------------------------

while node:

l=[]

count=node.count

lis=node

while lis.parent.name!=*'null'*:

#print(lis.parent.name)

lis=lis.parent

l.append(lis.name)

node=node.next

if len(l)>0:

for x in range(0,count):

datas.append(l)

pass

#print(item)

#print(datas)

return datas

pass