

Data Mining Assignment 2

- 姓名：潘国盛 计算机系(校外本科保送)
- 原本科学号：3014218157

Problem & Data Set

需要在给定的数据集上使用穷举、Apriori、FP-growth等方法挖掘频繁项集以及关联规则，并测定不同参数下消耗的时间、产生的项集数，并发现一些数据集中的规律

- GroceryStore
包含一些食品杂货店的商品交易记录，每个交易记录是一样或多样商品的集合，总共有9835条交易记录169个不同的商品；
- UNIX_usage
格式化清洗过的Unix用户数据，记录了8个用户最多两年的Unix命令使用记录，命令进行了切割和转义，`<3>`代表三个文件长度的路径名称

Code

• 数据处理

进行挖掘前需要对数据进行处理，将item进行编码然后存在二维数组中，并保存映射关系
首先将数据替换为如下样式

```
1 1\citrus fruit,semi-finished bread,margarine,ready soups
2 2\tropical fruit,yogurt,coffee
3 3\whole milk
4 4\pip fruit,yogurt,cream cheese ,meat spreads
5 5\other vegetables,whole milk,condensed milk,long life bakery product
```

然后读取后编码用json格式进行存储

```
1 import numpy as np
2 import json
3
4 X = np.loadtxt("raw/Groceries.txt", delimiter='\\', dtype=np.str)[1:, 1]
5
6 data_set = []
7
8 int_data_set = []
9
10 for i in range(len(X)):
11     arr = X[i].split(',')
12     data_set.extend(arr)
13     int_data_set.append(arr)
14
15 data_set = set(data_set)
```

```

16
17 thing_to_int = dict([(j, i) for (i, j) in enumerate(data_set)])
18 int_to_thing = dict([(i, j) for (i, j) in enumerate(data_set)])
19
20 thing_json = json.dumps(int_to_thing)
21 print(thing_json)
22 with open('data/Groceries_to_int.txt', 'w') as f:
23     print(thing_json, file=f)
24     f.close()
25
26 with open('data/Groceries.txt', 'w') as f:
27     for i in range(len(int_data_set)):
28         # tmpstr = ''
29         for j in range(len(int_data_set[i])):
30             int_data_set[i][j] = thing_to_int[int_data_set[i][j]]
31             # tmpstr += str(int_data_set[i][j]) + " "
32             # print(tmpstr, file=f)
33
34         # print(int_data_set, file=f)
35
36     int_data_set = json.dumps(int_data_set)
37     print(int_data_set, file=f)
38
39     f.close()

```

- **Apriori** 代码参考自《机器学习实战》，做少许改动以兼容python3并应对本问题，相对复杂的部分有详细注释

读取数据

```

1 def loadDataSet():
2     thing_arr = []
3
4     with open('data/Groceries.txt', 'r') as f:
5         X = f.read()
6         thing_arr = json.loads(X)
7         f.close()
8     return thing_arr

```

首先生成 C_1 集合，将1-项集都加入到 C_1 中

```

1 def createC1(dataSet):
2     '''
3     生成C1
4     :param dataSet:
5     :return:
6     '''
7     C1 = []
8     for transaction in dataSet:
9         for item in transaction:
10             if not [item] in C1:
11                 C1.append([item])
12     C1.sort()

```

```

13     #将项集列表转换为不可变集和
14     return [frozenset(item) for item in C1]

```

扫描数据集计数，去除 C_k 中的非频繁项集，生成 L_k

```

1  def scanD(D, Ck, minSupport = 50):
2      '''
3      扫描事务集D过滤Ck
4      :param D:
5      :param Ck:
6      :param minSupport:
7      :return:
8      '''
9      ssCnt = {}
10     for tid in D:
11         for can in Ck:
12             if can.issubset(tid):
13                 if can not in ssCnt.keys() : ssCnt[can] = 1
14                 else: ssCnt[can] += 1
15
16     retList = []
17     supportData = {}
18     for key in ssCnt:
19         support = ssCnt[key]
20         if support >= minSupport:
21             retList.insert(0, key)
22             supportData[key] = support
23     return retList, supportData

```

从 L_{k-1} 集中生成 C_K 集合，需要将 L_{k-1} 项集进行排序，然后将前k-2项相同的集合进行合并

```

1  def aprioriGen(Lk, k):
2      '''
3      生成Ck
4      :param Lk:
5      :param k:
6      :return:
7      '''
8      retList = []
9      lenLk = len(Lk)
10     for i in range(lenLk):
11         for j in range(i+1, lenLk):
12             L1 = list(Lk[i]):k-2]
13             L2 = list(Lk[j]):k-2]
14             L1.sort()
15             L2.sort()
16             if L1 == L2:
17                 retList.append(Lk[i] | Lk[j])
18     return retList

```

将上述步骤结合起来，就可以生成所有的频繁项集，直到为空终止算法

```

1 def apriori(dataSet, minSupport = 50):
2     C1 = createC1(dataSet=dataSet)
3     D = [set(item) for item in dataSet]
4     L1, supportData = scanD(D, C1, minSupport)
5     L = [L1]
6     k = 2
7     while(len(L[k-2]) > 0):
8         Ck = aprioriGen(L[k-2], k)
9         Lk, supK = scanD(D, Ck, minSupport)
10        supportData.update(supK)
11        L.append(Lk)
12        k += 1
13    return L, supportData

```

• FP-Growth

FP-Growth实现起来相对复杂，需要定义树结点结构

class treeNode:

```

1 def __init__(self, nameValue, numOccur, parentNode):
2     # 值
3     self.name = nameValue
4     # 计数
5     self.count = numOccur
6     # 下一个相同值的结点
7     self.nodeLink = None
8     # 父节点
9     self.parent = parentNode
10    # 孩子结点
11    self.children = {}
12
13    def inc(self, numOccur):
14        self.count += numOccur
15
16    def disp(self, ind=1):
17        print(" "*ind, self.name, ' ', self.count)
18        for child in self.children.values():
19            child.disp(ind+1)

```

加载数据集代码同Apriori

初始化transaction计数

```

1 def createInitSet(dataSet):
2     retDict = {}
3     for trans in dataSet:
4         retDict[frozenset(trans)] = 1
5     return retDict

```

建立FP树

```

1 def createTree(dataSet, minSup = 1):
2     '''
3     创建根结点以及搜索链表表头
4     :param dataSet:
5     :param minSup:
6     :return:
7     '''
8
9     # 搜索链表头
10    headerTable = {}
11    # 在搜索用的链表头除记录每个item的频数
12    for trans in dataSet:
13        for item in trans:
14            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]
15
16    # 小于最小支持度的item不用考虑
17    for k in list(headerTable.keys()):
18        if headerTable[k] < minSup:
19            del(headerTable[k])
20    freqItemSet = set(headerTable.keys())
21    # 如果不存在频繁项集则直接返回空
22    if len(freqItemSet) == 0:
23        return None, None
24    # 为每个结点增加一个指向下一个同值结点的指针
25    for k in headerTable.keys():
26        headerTable[k] = [headerTable[k], None]
27    # 树根
28    retTree = treeNode('Null Set', 1, None)
29
30    for tranSet, count in dataSet.items():
31        localD = {}
32        for item in tranSet:
33            if item in freqItemSet:
34                localD[item] = headerTable[item][0]
35            if len(localD) > 0:
36                # 每个transaction中的item按出现的次数从高到低排
37                orderedItems = [v[0] for v in sorted(localD.items(), key=lambda p: p[1],
reverse=True)]
38                # 建树
39                updateTree(orderedItems, retTree, headerTable, count)
40    return retTree, headerTable
41
42 def updateTree(items, inTree, headerTable, count):
43     '''
44     每个transaction递归更新到树上，并更新搜索链表
45     :param items:
46     :param inTree:
47     :param headerTable:
48     :param count: 每个transaction的出现次数
49     :return:
50     '''
51    # 每个transaction的最高出现词数item直接接在root上
52    if items[0] in inTree.children:

```

```

53     # 有该元素项时计数值+1
54     inTree.children[items[0]].inc(count)
55 else:
56     # 没有这个元素项时创建一个新节点
57     inTree.children[items[0]] = treeNode(items[0], count, inTree)
58     # 更新头指针表或前一个相似元素项节点的指针指向新节点
59     if headerTable[items[0]][1] == None:
60         headerTable[items[0]][1] = inTree.children[items[0]]
61     else:
62         updateHeader(headerTable[items[0]][1], inTree.children[items[0]])
63
64     # 递归建树
65     if len(items) > 1:
66         # 对剩下的元素项迭代调用updateTree函数
67         updateTree(items[1:], inTree.children[items[0]], headerTable, count)
68
69
70 def updateHeader(nodeToTest, targetNode):
71     '''
72     找到链表尾加上一个
73     :param nodeToTest:
74     :param targetNode:
75     :return:
76     '''
77     while (nodeToTest.nodeLink != None):
78         nodeToTest = nodeToTest.nodeLink
79     nodeToTest.nodeLink = targetNode

```

在每次查询条件FP树时，需要递归寻找其条件前缀路径

```

1  def ascendTree(leafNode, prefixPath):
2      '''
3      递归寻找父节点
4      :param leafNode:
5      :param prefixPath:
6      :return:
7      '''
8      if leafNode.parent != None:
9          prefixPath.append(leafNode.name)
10         ascendTree(leafNode.parent, prefixPath)
11
12 def findPrefixPath(basePat, treeNode):
13     condPats = {}
14     while treeNode != None:
15         prefixPath = []
16         # 获得某个叶子节点的前缀路径
17         ascendTree(treeNode, prefixPath)
18         if len(prefixPath) >= 2:
19             # 去掉自己获得前缀路径，且权重为当前结点的权重，用于建立条件前缀树
20             condPats[frozenset(prefixPath[1:])] = treeNode.count
21         treeNode = treeNode.nodeLink
22     return condPats

```

进行频繁项挖掘

```
1 def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
2     ...
3     递归查找频繁项集
4     :param inTree: FP树
5     :param headerTable:
6     :param minSup:
7     :param preFix: 当前前缀
8     :param freqItemList: 存储频繁项集
9     :return:
10    ...
11    # 从出现次数少的开始找
12    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: p[1][0])]
13
14    for basePat in bigL:
15        newFreqSet = preFix.copy()
16        newFreqSet.add(basePat)
17        freqItemList.append(newFreqSet)
18        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
19        CondTree, Header = createTree(condPattBases, minSup)
20
21        if Header != None:
22            mineTree(CondTree, Header, minSup, newFreqSet, freqItemList)
```

Experiment

实验将从时间消耗角度和频繁项集角度数量入手比较两个算法

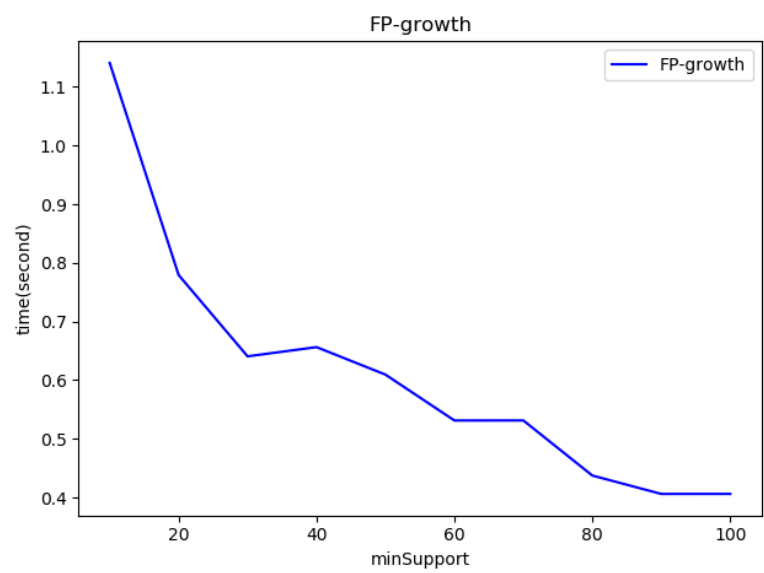
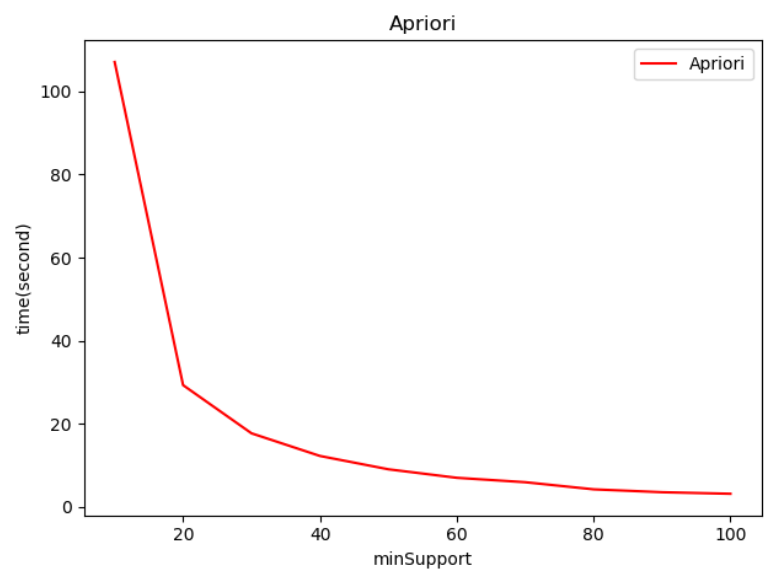
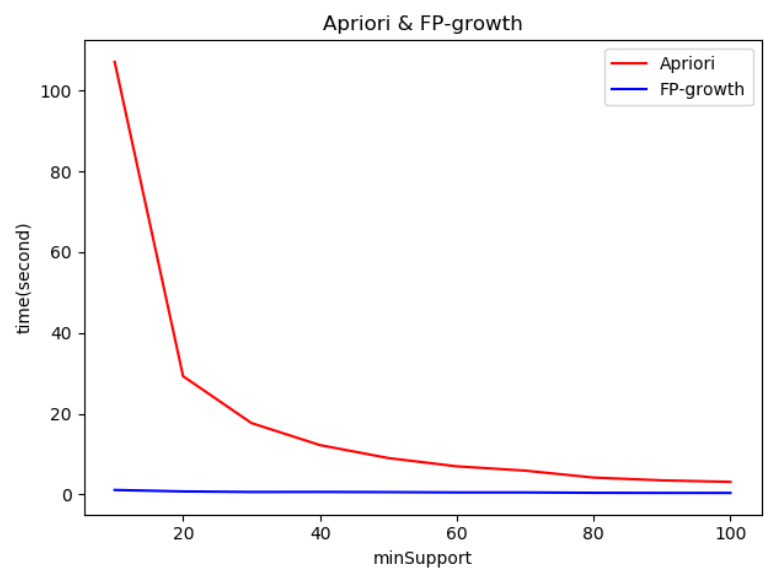
数据会以前面预处理所用的编码暂时存储，最后将数据映射回原来的字符串进行观察

Result and Discussion

时间效率

以杂货店数据集作为测试时间效率的数据集。

运行时间随最小支持度的减小的变化，下图展示了时间随支持度增长的变化，时间消耗都回随最小支持度的增长而增大，但是Apriori增大的幅度要远大于FP-growth

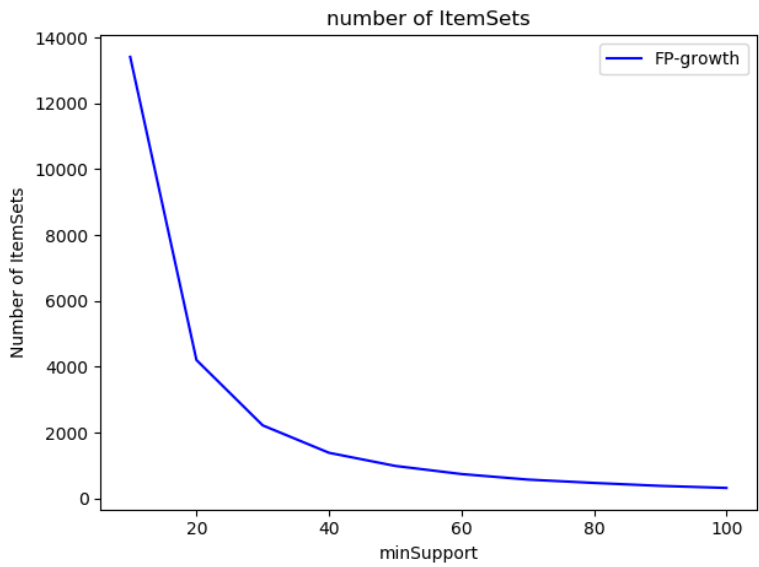


频繁项集数量

以杂货店数据集作为测试频繁项集数的数据集。

数据如下表

最小支持度	100	90	80	70	60	50	40	30	20	10
频繁项集数	13413	4206	2221	1388	989	742	574	472	383	320



Conclusions

Apriori算法在频繁模式挖掘的过程中，需要重复的查询原数据集，导致效率随着transaction的增多而大大降低

FP-growth算法减少了很多重复的查询，但是需要消耗大量的内存来建立FP树以及条件PF树，面对大数据集时仍然需要在空间上做优化与调整