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

Probelm & 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\citrus fruit,semi-finished bread,margarine,ready soups
2\tropical fruit,yogurt,coffee
3\whole milk
4\pip fruit,yogurt,cream cheese ,meat spreads
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

```python
import numpy as np
import json

X = np.loadtxt("raw/Groceries.txt", delimiter='\\', dtype=np.str)[1:, 1]

data_set = []

int_data_set = []

for i in range(len(X)):
    arr = X[i].split(',')
    data_set.extend(arr)
    int_data_set.append(arr)

data_set = set(data_set)
```

```python

thing_to_int = dict([(j, i) for (i, j) in enumerate(data_set)])
int_to_thing = dict([(i, j) for (i, j) in enumerate(data_set)])

thing_json = json.dumps(int_to_thing)
print(thing_json)
with open('data/Groceries_to_int.txt', 'w') as f:
    print(thing_json, file=f)
    f.close()

with open('data/Groceries.txt', 'w') as f:
    for i in range(len(int_data_set)):
        # tmpstr = ''
        for j in range(len(int_data_set[i])):
            int_data_set[i][j] = thing_to_int[int_data_set[i][j]]
        #     tmpstr += str(int_data_set[i][j]) + " "
        # print(tmpstr, file=f)

    # print(int_data_set, file=f)

    int_data_set = json.dumps(int_data_set)
    print(int_data_set, file=f)

    f.close()
```

- **Baseline**

  用递归的方法遍历所有transaction的子集进行统计

```python
def iterSet(prefix, trans, pos, result, put):
    if pos >= len(trans):
        return
    prefixCopy = prefix.copy()
    if put == True:
        prefixCopy.add(trans[pos])
        froz = frozenset(prefixCopy)
        if froz not in result.keys():
            result[froz] = 1
        else:
            result[froz] = result[froz] + 1
    iterSet(prefixCopy, trans, pos + 1, result, True)
    iterSet(prefixCopy, trans, pos + 1, result, False)

if __name__ == '__main__':

    dataSet = loadDataSet()
    result = {}
    for trans in dataSet:
        iterSet(set([]), trans, 0, result, True)
        iterSet(set([]), trans, 0, result, False)
    minSup = 2
    for itemSet in list(result.keys()):

        if result[itemSet] < minSup:
```

```
            del(result[itemSet])
    # print(result)
    print(result.keys())
```

- **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

```
def loadDataSet():
    thing_arr = []

    with open('data/Groceries.txt', 'r') as f:
        X = f.read()
        thing_arr = json.loads(X)
        f.close()
    return thing_arr
```

  首先生成C_1集合，将1-项集都加入到C_1中

```
def createC1(dataSet):
    '''
    生成C1
    :param dataSet:
    :return:
    '''
    C1 = []
    for transaction in dataSet:
        for item in transaction:
            if not [item] in C1:
                C1.append([item])
    C1.sort()
    #将项集列表转换为不可变集和
    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

```
def scanD(D, Ck, minSupport = 50):
    '''
    扫描事务集D过滤Ck
    :param D:
    :param Ck:
    :param minSupport:
    :return:
    '''
    ssCnt = {}
    for tid in D:
        for can in Ck:
            if can.issubset(tid):
                if can not in ssCnt.keys() : ssCnt[can] = 1
                else: ssCnt[can] += 1

```

```
    retList = []
    supportData = {}
    for key in ssCnt:
        support = ssCnt[key]
        if support >= minSupport:
            retList.insert(0, key)
        supportData[key] = support
    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

```
def aprioriGen(Lk, k):
    '''
    生成Ck
    :param Lk:
    :param k:
    :return:
    '''
    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
```

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

```
def apriori(dataSet, minSupport = 50):
    C1 = createC1(dataSet=dataSet)
    D = [set(item) for item in 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
```

- **FP-Growth**

  FP-Growth实现起来相对复杂，需要定义树结点结构

  class treeNode:

```python
def __init__(self, nameValue, numOccur, parentNode):
    # 值
    self.name = nameValue
    # 计数
    self.count = numOccur
    # 下一个相同值的结点
    self.nodeLink = None
    # 父节点
    self.parent = parentNode
    # 孩子结点
    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)
```

加载数据集代码同Apriori

初始化transaction计数

```python
def createInitSet(dataSet):
    retDict = {}
    for trans in dataSet:
        retDict[frozenset(trans)] = 1
    return retDict
```

建立FP树

```python
def createTree(dataSet, minSup = 1):
    '''
    创建根结点以及搜索链表表头
    :param dataSet:
    :param minSup:
    :return:
    '''

    # 搜索链表头
    headerTable = {}
    # 在搜索用的链表头除记录每个item的频数
    for trans in dataSet:
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]

    # 小于最小支持度的item不用考虑
    for k in list(headerTable.keys()):
        if headerTable[k] < minSup:
            del(headerTable[k])

    freqItemSet = set(headerTable.keys())
```

```python
    # 如果不存在频繁项集则直接返回空
    if len(freqItemSet) == 0:
        return None, None
    # 为每个结点增加一个指向下一个同值结点的指针
    for k in headerTable.keys():
        headerTable[k] = [headerTable[k], None]
    # 树根
    retTree = treeNode('Null Set', 1, None)

    for tranSet, count in dataSet.items():
        localD = {}
        for item in tranSet:
            if item in freqItemSet:
                localD[item] = headerTable[item][0]
        if len(localD) > 0:
            # 每个transaction中的item按出现的次数从高到低排
            orderedItems = [v[0] for v in sorted(localD.items(), key=lambda p: p[1], reverse=True)]
            # 建树
            updateTree(orderedItems, retTree, headerTable, count)
    return retTree, headerTable

def updateTree(items, inTree, headerTable, count):
    '''
    每个transaction递归更新到树上，并更新搜索链表
    :param items:
    :param inTree:
    :param headerTable:
    :param count: 每个transaction的出现次数
    :return:
    '''
    # 每个transaction的最高出现词数item直接接在root上
    if items[0] in inTree.children:
        # 有该元素项时计数值+1
        inTree.children[items[0]].inc(count)
    else:
        # 没有这个元素项时创建一个新节点
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        # 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
        if headerTable[items[0]][1] == None:
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], inTree.children[items[0]])

    # 递归建树
    if len(items) > 1:
        # 对剩下的元素项迭代调用updateTree函数
        updateTree(items[1:], inTree.children[items[0]], headerTable, count)


def updateHeader(nodeToTest, targetNode):
    '''

    找到链表尾加上一个
```

```
    :param nodeToTest:
    :param targetNode:
    :return:
    '''
    while (nodeToTest.nodeLink != None):
        nodeToTest = nodeToTest.nodeLink
    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

```
def ascendTree(leafNode, prefixPath):
    '''
    递归寻找父节点
    :param leafNode:
    :param prefixPath:
    :return:
    '''
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)

def findPrefixPath(basePat, treeNode):
    condPats = {}
    while treeNode != None:
        prefixPath = []
        # 获得某个叶子节点的前缀路径
        ascendTree(treeNode, prefixPath)
        if len(prefixPath) >= 2:
            # 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
            condPats[frozenset(prefixPath[1:])] = treeNode.count
        treeNode = treeNode.nodeLink
    return condPats
```

进行频繁项挖掘

```
def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    '''
    递归查找频繁项集
    :param inTree: FP树
    :param headerTable:
    :param minSup:
    :param preFix: 当前前缀
    :param freqItemList: 存储频繁项集
    :return:
    '''
    # 从出现次数少的开始找
    bigL = [v[0] for v in sorted(headerTable.items(), key=lambda p: p[1][0])]

    for basePat in bigL:
        newFreqSet = preFix.copy()
        newFreqSet.add(basePat)

        freqItemList.append(newFreqSet)
```

```
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        CondTree, Header = createTree(condPattBases, minSup)

        if Header != None:
            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

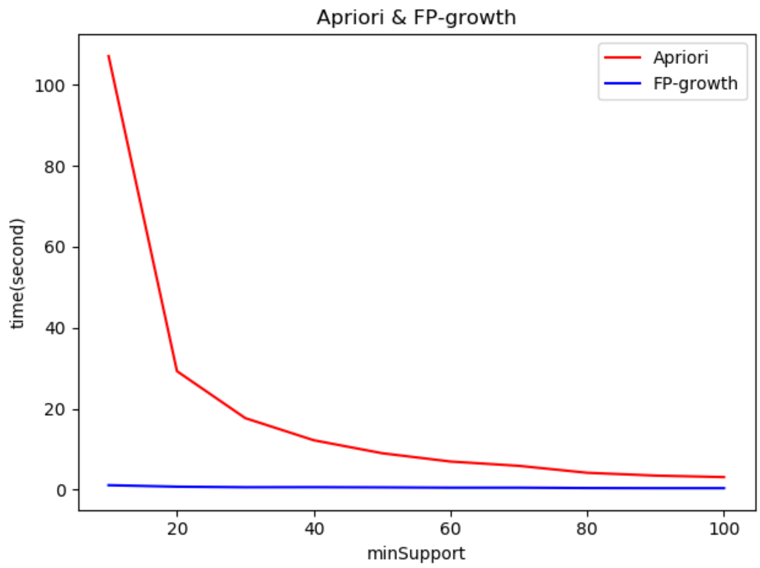
数据会以前面预处理所用的编码暂时存储，最后将数据映射回原来的字符串进行观察

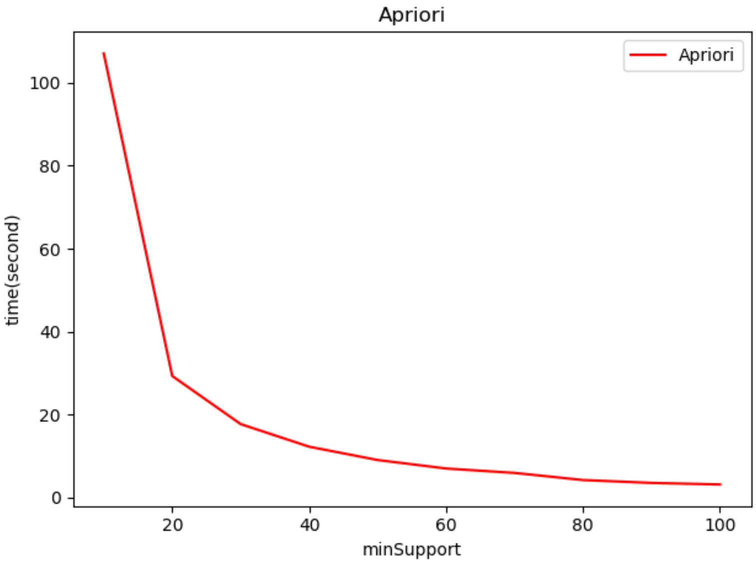
Result and Discussion

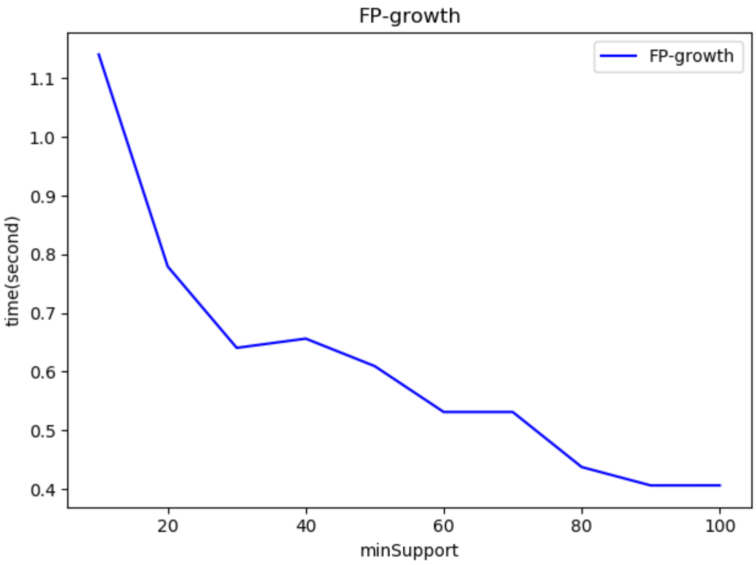
时间效率

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





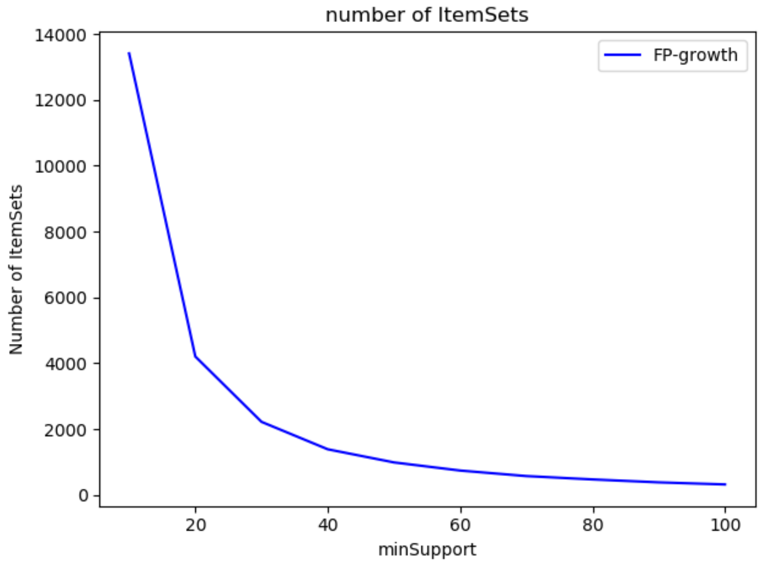


频繁项集数量

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



频繁项集挖掘结论

- Groceries

  选择不同的的最小支持度，并提取至少两个项的频繁项集，查看结果

```
tropical fruit + whole milk
root vegetables + other vegetables
root vegetables + whole milk
other vegetables + yogurt
whole milk + yogurt
whole milk + soda
rolls/buns + other vegetables
rolls/buns + whole milk
whole milk + other vegetables
```

  选择minSupport=350可以得到上述结果。发现牛奶的购买频数很高，且经常与水果、蔬菜等类别的商品一同购买

- UNIX

  以Unix0数据为例，选择不同的的最小支持度，并提取至少两个项的频繁项集，查看结果

```
cd + ls
cd + <1> + ls
cd + <1>
finger + <1>
elm + exit
<1> + elm + exit
elm + <1>
exit + ls
<1> + exit + ls
<1> + ls
<1> + exit
```

选择minSupport=120可以得到上述结果。可以发现`cd`经常与`ls`联用，`cd` `exit` `ls`等命令用得较多

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