DM-2019 lab2实验报告

161220096 欧阳鸿荣

DM-2019 lab2实验报告
- 1.实验说明
- 2.数据集介绍与数据获取
  - （1）Grocery Store 数据集
  - （2）UNIX_usage 数据集
  - （3）bftest 数据集
- 3.方法的介绍和代码的实现
  - (1) Apriori 算法
    - (a) 算法介绍
    - (b) 代码实现
      - (1) scanD函数
      - (2) aprioriGen函数
      - (3) apriori函数
  - (2) FP-Growth 算法
    - (a) 算法介绍
    - (b) 代码实现
      - (1) FP_Node
      - (2) FP-Tree
      - (3) find_frequent_itemsets
  - (3) Brute-Force 算法
    - (a) 算法介绍
    - (b) 代码介绍
- 4.代码执行与实验数据
  - 4.1 文件结构
  - 4.2 代码使用说明
    - (1) getApriori
    - (2) getFPGrowth
    - (3) getBrute
  - 4.3 实验结果和数据
    - (1) Grocery Store 数据集
    - (2) Unix 用户数据集
    - (3) bftest数据集
- 5.关联规则的挖掘与解读
  - 5.1 Grocery Store 数据集
  - 5.2 Unix 用户数据集
- 6.方法的比较与感想
- 7.参考资料

1.实验说明

（1）实验要求：对给定的数据集挖掘关联规则

（2）通过给定不同的支持度和置信度，比较 Apriori, FP-Growth 和 Brute-Force 蛮力搜索的方法下挖掘频繁项集，在生成的频繁项集数目，内存的使用和时间的消耗上的区别。

（3）尝试通过Apriori和FP-Growth挖掘一些有趣的关联规则，并且谈谈你关于这些规则的想法

2.数据集介绍与数据获取

对于数据集，我实现了一个`FileOption`类，针对数据集的不同特点，向不同算法的代码传递数据集

（1）Grocery Store 数据集

该数据集记录了在一个月里杂货店的交易记录，一共有9835条交易记录，包含169种商品，记录格式为

```
"id","{item1,item2,...,itemn}"
```

对于该数据集，通过如下代码进行解析，以下是核心代码：

```
def get_frozenset(self,filename):
    self.load_csv(filename) # 自己实现的load_csv文件，读取Groceries.csv内容
    out = []
    for lines in self.dataset_original:
        # 对数据进行清洗，去除一些无用符号
        lines = str(lines)
        lines = lines.strip('{}')  # 去除两端的符号
        lines = lines.replace('/', ' ')  # 把斜杠转化为空格
        transaction = lines.split(',')  # 按逗号分割
        self.transactions.append(transaction)  #记录每一条购买记录
        for item in transaction:
            if not [item] in out:
                out.append([item]) #计算数据集中的项数
                out.sort()
                self.transactions = list(map(set, self.transactions))
                # 使用frozenset是为了后面可以将这些值作为字典的键
                self.items = list(map(frozenset, out))
```

（2）UNIX_usage 数据集

该数据集记录了8个UNIX用户在课程中的命令历史记录，会话以`**SOF**`开始，以`**EOF**`结束

以该段会话为例：

```
# Start session 1
cd ~/private/docs
ls -laF | more
cat foo.txt bar.txt zorch.txt > somewhere
exit
# End session 1
```

在数据集中被解析和清理为：

```
**SOF**
cd
<1>         # one "file name" argument
ls
-laF
|
more
cat
<3>         # three "file" arguments
>
<1>
exit
**EOF**
```

文件名、用户名、目录结构、网址，主机名等均以`<数字>`的格式代替，在实际处理中对其清洗，核心代码如下：

```
def get_UNIX_data(self):
    """
    默认数据集放在 UNIX_usage/下
    """
    path0 = 'dataset/UNIX_usage/USER'
    path1 = '/sanitized_all.981115184025'
    out = []
    for i in range(9):
        filename = path0+str(i)+path1  #实际处理中，将8个用户的数据合并，试图发现用户的共性规律
        f = open(filename, 'r')
        lines = f.readlines()
        startFlag = "**SOF**"
        endFalg = "**EOF**"
        transaction = []

        for l in lines:  # 每次取出一行
            t = l.strip('\n')  # 去掉行尾的换行符
            if t == startFlag:
                continue
            elif t == endFalg:
```

```
                self.transactions.append(transaction)
                for item in transaction:
                    if not [item] in out:
                        out.append([item])
                transaction = []
            else:  #对形如<数字>的数据进行清洗
                cle = t.strip('<>')  # 去除两端的符号
                if cle.isdigit():
                    continue
                else:
                    transaction.append(t)

    ...  #后面同Grocery Store 数据集的处理
```

以上是Apriori 算法数据的获取，FP-Growth算法由于数据结构有差异，因此基于上述方法得到的数据进行修饰后得到适合其的数据，这里不多赘述。

（3）bftest 数据集

用于测试Brute-Force方法的数据集，为Grocery Store 数据集的前7项

```
"","items"
"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}"
"6","{whole milk,butter,yogurt,rice,abrasive cleaner}"
"7","{rolls/buns}"
```

3.方法的介绍和代码的实现

(1) Apriori 算法

(a) 算法介绍

Apriori 算法通过限制候选项产生和发现频繁项集。其限制基于**先验性质：频繁项集的所有非空子集也一定是频繁的。**算法使用一种逐层搜索的迭代方法，其中k项集用于探索$k+1$项集。

step1: 扫描数据库，累计每个项的计数并搜集满足最小支持度的项，找出频繁1项集的集合。该集合记为L_1

step2: 使用L_1找出频繁2项集的集合L_2，使用L_2找出L_3

step3: 重复 step2，直到不能再找到频繁k项集

(b) 代码实现

该代码的实现参考了书籍《机器学习实战》。根据上述步骤，此处介绍Apriori 算法的三个核心函数：

(1) scanD函数

scanD函数主要对应step1并用于step2中的筛选，该函数有三个参数，分别是数据集C_k,包含候选集合的列表dataset和最小值支持度。该函数遍历每一条记录的每一项，并计算数据集中集合是否是候选集合的子集，依次计算支持度，并保留满足支持度的项。

```
def scanD(dataset, Ck, minSupport):
    """
    将不符合minSupport的集合删去，
    返回频繁项集列表:retList 所有元素的支持度Dict:supportData
    """
    ssCnt = defaultdict(lambda: 0) # 默认值为0的字典
    for trans in dataset:
        for item in Ck:
            # 判断can是否是tid的子集，并以此计算sup_count
            if item.issubset(trans):
                ssCnt[item] += 1

    dataLenth = dataset.__len__()

    retList = []  # 重新记录每一项的值
    supportData = {}  # 项的支持度

    for item in ssCnt:
        support = ssCnt[item] / dataLenth
        # 保留满足最小支持度要求的项
        if support >= minSupport:
            retList.append(item)
        supportData[item] = support

    return retList, supportData
```

(2) aprioriGen函数

aprioriGen函数主要对应上述的step2，主要完成连接工作，为找出L_k，通过将L_{k-1}与自身连接产生候选k项集

```
def aprioriGen(Lk, k):
    """
    当前k-2项相同时，将两个集合合并
    返回频繁项集列表Ck:res
    """
    resList = []
    for i in range(Lk.__len__()):  # 两层循环比较Lk中的每个元素与其它元素
        for j in range(i + 1, Lk.__len__()):
            L1 = (list(Lk[i])[0:k - 2])  # 取集合排序后的前k-1项
            L2 = (list(Lk[j])[0:k - 2])
            L1.sort()
            L2.sort()
            if L1 == L2:
                # 比较前k-1项，若前k-1项相同，则合并
                res = Lk[i] | Lk[j]
                resList.append(res)  # 求并集
    return resList
```

(3) apriori函数

该函数对应整个算法的的过程，首先得到L_1，然后通过aprioriGen函数不断将L_{k-1}与自身连接产生候选k项集，然后剪枝得到L_k，直到不能再找到频繁k项集

```
def apriori(items,transactions, minSupport=0.5):
    """
    返回 所有满足大于阈值的组合 集合支持度列表
    """
    L1, supportData = scanD(transactions, items, minSupport)  # 过滤数据,得到的L1列表中的每个单项至少出现在满足minSupport的记录中
    L = []  # 记录频繁项
    L.append(L1)
    k = 2
    while len(L[k - 2]) > 0:  # 若仍有满足支持度的集合则继续做关联分析
        Ck = aprioriGen(L[k - 2], k)  # Ck候选频繁项集
        Lk, supK = scanD(transactions, Ck, minSupport)  # Lk频繁项集
        supportData.update(supK)  # 把新出现的(trans,support)加入到supportData中
        L.append(Lk)
        k = k + 1
    return L, supportData
```

(2) FP-Growth 算法

(a) 算法介绍

该算法使用频繁模式增长策略，目的是降低因指数爆炸和重复扫描数据集带来的非平凡开销。算法主要步骤如下：

step1: 算法第一次扫描，同Apriori算法，导出频繁1项集并按照支持度降序排列

step2: 将代表频繁项集的数据库压缩到一棵FP树上，该树保留项集的相关信息：

i. 第二次扫描数据库，对每个事务按照支持度递减处理，并为每个事务创建分支

ii.为方便遍历，创建一个项头表，使得每项通过一个节点链指向其在数中的位置

step2: 将压缩后的数据库划分成一组条件数据库，每个数据库关联模式段，并以此挖掘每个条件数据库

i. 从长度为1的频繁模式开始，构造其条件模式基

step3: 对每个模式片段，考察与他关联的数据集，得到频繁项：

i.模式增长通过后缀模式与条件FP树产生的频繁模式链接实现

(b) 代码实现

该代码的实现参考了Github上代码，报告未有参考链接。根据上述步骤，此处介绍FP-Growth算法的核心代码：

(1) FP_Node

FP_Node是FP-Tree的节点，用于构建FP-树

```
class FPNode(object):
    """
    FP-Tree 的节点
    """

    def __init__(self, tree, item, count=1):
        self._tree = tree       # 树节点
        self._item = item       # 项
        self._count = count     # 该项出现的次数
        self._parent = None     # 父节点
        self._children = {}     # 子节点
        self._neighbor = None   # 用于链接同类项的邻居结点

    def add(self, child):
        """
        为该节点增加一个子节点
        """
        if not child.item in self._children:
            self._children[child.item] = child
            child.parent = self

    def search(self, item):
        """
        查找该节点的子节点中项为item的节点
```

```
        """
        if item in self._children:
            return self._children[item]
        else:
            return None
    ...
```

(2) FP-Tree

FP-Tree由FP_Node节点组成

```
class FPTree(object):
    """
    FP-Growth Tree
    """

    def __init__(self):
        # 树的根节点
        self._root = FPNode(self, None, None)
        self._routes = {}  # FP树每个项的HeaderTable
```

(3) find_frequent_itemsets

该函数对应上述的算法介绍，其中做了一个取巧的操作，便是对一条数据进行数据清洗，去除掉非单项频繁项，这样对结果没有影响，也能增加代码执行的效率

```
def find_frequent_itemsets(transactions, minSup):

    # 统计数据集中每个项的数目
    items = {}
    for transaction in transactions:
        for item in transaction:
            if item in items:
                items[item] += 1
            else:
                items[item] = 1

    # 将不满足minSup的项去除
    newItems = {}
    for k,v in items.items():
        if v < minSup:
            continue
        newItems[k] = v
    items = newItems

    # 对一条数据进行数据清洗，去除掉非单项频繁项，这样对结果没有影响
    def clean_transaction(transaction):
        transaction = filter(lambda v: v in items, transaction)
        transaction_list = list(transaction)    # 为了防止变量在其他部分调用，这里引入临时变量
transaction_list
        transaction_list.sort(key=lambda v: items[v], reverse=True)
        return transaction_list
```

```
    master = FPTree()

    # 将清洗后的数据集加入FP Tree
    for trans in transactions:
        trans = clean_transaction(trans)
        master.add(trans)

    # 寻找前缀路径
    def find_with_suffix(tree, suffix):
        for item, nodes in tree.items():
            support = 0
            for n in nodes:
                support += n.count

            if support >= minSup and item not in suffix:
                # New winner!
                found_set = [item] + suffix
                yield (found_set, support)
                # 构造一颗FP条件树
                cond_tree = conditional_tree_from_paths(tree.prefix_paths(item))
                for s in find_with_suffix(cond_tree, found_set):
                    yield s

    for itemset in find_with_suffix(master, []):
        yield itemset
```

(3) Brute-Force 算法

(a) 算法介绍

该算法无他，就是暴力枚举。先通过对数据集计数，统计数据集中一共有多少项，然后得到这些项的非空子集的集合。对每个非空子集，与数据集中的序列逐一比较，看是否满足子集关系，并计算支持度。

(b) 代码介绍

这里主要介绍得到所有子集的代码，具体代码会在后文的代码使用说明中给出。

```
def PowerSetsBinary(items):
    '''
    传入items集合，返回items集合的所有非空子集
    :param items: 数据集中项的集合
    :return: items集合的所有非空子集
    '''
    N = items.__len__()
    retSubset=[]
    for i in range(2**N):
        combo = frozenset([])
        for j in range(N):
            if(i >> j ) % 2 == 1:
                combo = combo.union(items[j])
        if combo.__len__()!= 0:
            print("生成第", i, "个子集", combo)
            retSubset.append(combo)
    return retSubset
```

4.代码执行与实验数据

4.1 文件结构

本次实验代码由以下文件组成：

文件名	作用
dataset	存放该实验的数据集，共3个，分别是Groceries，UNIX和暴力专用小数据集
FileOption.py	用于文件读取和数据集获取和清洗的类
Apriori.py	实现Apriori算法并提供对外接口的文件
FPGrowth.py	实现FP-Growth算法并提供对外接口的文件
BruteForce.py	实现暴力算法并提供对外接口的文件
freqItems.py	测试脚本，可以在此调用上述方法，并得到频繁项集，关联规则，内存和时间消耗

4.2 代码使用说明

运行freqItems.py即可对代码进行测试，代码参数如下：

参数名	含义
methodType	使用的方法，0为Apriori法，1为FPGrowth法，2为暴力法（暴力法只能使用datatye=3的数据集，不然会炸）
datatype	挖掘的数据集，0为Groceies数据集，1为UNIX数据集, 2为测试暴力算法测试集
minSup	最小支持度（默认为0-1的小数）
minConf	最小置信度（默认为0-1的小数）
getFreqitems	是否输出频繁项集（暴力法只能输出频繁项集）
getRules	是否输出关联规则

根据`methodType`，代码调用下列三个接口，通过修改上述参数，即可对代码进行测试

```
if methodType == 0:
    ap.getApriori(datatype,minSup,minConf,getFreqitems,getRules)
elif methodType == 1:
    fp.getFPGrowth(datatype,minSup,minConf,getFreqitems,getRules)
elif methodType == 2:
    if datatype != 2:
        print("暴力方法只能处理小规模数据")
        sys.exit(-1)
    bf.getBrute(datatype,minSup,minConf,getFreqitems,getRules)
else:
    print("请选择正确的方法类型")
    sys.exit(-1)
```

下面分别给出三个对外接口的代码：

(1) getApriori

```
def getApriori(datatype=0,minSup=0.5,minConf=0.7,getFreqitems=True,getRules=False):
    '''
    使用 Apriori算法得到频繁项集和规则
    :param datatype: 挖掘的数据集，0为Groceies数据集，1为UNIX数据集， 2为测试暴力算法测试集
    :param getFreqitems: 是否得到频繁项
    :param getRules: 是否得到频繁规则
    :return:
    '''
    fop = FileOption()
    items = []
    transactions = []

    if datatype == 0:
        items, transactions = fop.get_data('dataset/Groceries.csv')
    elif datatype == 1:
        items, transactions = fop.get_UNIX_data()
    elif datatype == 2:
        items, transactions = fop.get_data('dataset/bftest.csv')
    else:
        print("数据集类型出错")
        return

    minSupport = minSup
    minConf = minConf

    print("Apriori 开始")
    L, supportData = apriori(items, transactions, minSupport=minSupport)

    count = 0  # 频繁项集数
    if getFreqitems:
        for Li in L:
            count += Li.__len__()
            Llist = []
            for items in Li:
                Llist.append(list(set(items)))
```

```
            result = sorted(Llist, key=lambda i: i[0])  # 排序后输出
            for its in result:
                print(str(its) + "->" + str(supportData.get(frozenset(its))))  # 输出每个频繁
项集的支持度
    print("频繁项数为: "+str(count))

    rules = generateRules(L, supportData, minConf=minConf)
    if getRules:
        print("一共有" + str(rules.__len__()) + "条满足置信度的规则，如下所示")
        for rule in rules:
            print(str(rule[0]) + "->" + str(rule[1]) + ":" + str(rule[2]))


    print("频繁项集个数", count)
    print("挖掘到规则数", rules.__len__())
```

(2) getFPGrowth

```
def getFPGrowth(datatype=0,minSup=0.5,minConf=0.7,getFreqitems=True,getRules=False):
    '''
    使用 FPGrowth算法得到频繁项集和规则
    :param datatype: 挖掘的数据集，0为Groceies数据集，1为UNIX数据集，2为测试暴力算法测试集
    :param getFreqitems: 是否得到频繁项
    :param getFreqitems: 是否得到频繁项
    :param getRules: 是否得到频繁规则
    :return:
    '''

    fop = FileOption()
    dataset = []
    if datatype == 0:
        dataset= fop.get_data_FP_new('dataset/Groceries.csv')
    elif datatype == 1:
        dataset = fop.get_data_FP_UNIX()
    elif datatype == 2:
        dataset = fop.get_data_FP_new('dataset/bftest.csv')
    else:
        print("数据集类型出错")
        return
    minSupport = dataset.__len__() * minSup
    minConf = minConf

    print("FP-Growth 开始")
    frequent_itemsets = find_frequent_itemsets(dataset, minSup=minSupport)
    # print(type(frequent_itemsets))   # print type

    result = []
    for itemset, support in frequent_itemsets:    # 将generator结果存入list
        result.append((itemset, support))

    result = sorted(result, key=lambda i: i[0])   # 排序后输出
    if getFreqitems:
        for itemset, support in result:
```

```
            print(str(itemset) + ' ' + str(support/dataset.__len__()))
    print("频繁项集个数",result.__len__())

    L, supportData = FP_rule_adapter(result=result,datasetlenth=dataset.__len__())

    rules = generateRules(L, supportData, minConf=minConf)

    if getRules:
        for rule in rules:
            print(str(rule[0]) + "->" + str(rule[1]) + ":" + str(rule[2]))

    print("频繁项集个数", result.__len__())
    print("挖掘到规则数", rules.__len__())
```

(3) getBrute

```
def getBrute(datatype=0,minSup=0.5,minConf=0.7,getFreqitems=True,getRules=False):
    '''
    使用 Apriori算法得到频繁项集和规则
    :param datatype: 挖掘的数据集，0为Groceies数据集，1为UNIX数据集，2为测试暴力算法测试集
    :param getFreqitems: 是否得到频繁项
    :param getRules: 是否得到频繁规则
    :return:
    '''
    fop = FileOption()
    items = []
    transactions = []

    if datatype == 0:
        items, transactions = fop.get_data('dataset/Groceries.csv')
    elif datatype == 1:
        items, transactions = fop.get_UNIX_data()
    elif datatype == 2:
        items, transactions = fop.get_data('dataset/bftest.csv')
    else:
        print("数据集类型出错")
        return

    minSup = minSup * transactions.__len__()

    print("Brute Force 开始")

    subSets = PowerSetsBinary(items)  # 得到items的所有子集

    print("一共有",subSets.__len__(),"个子集")
    supDict = {}
    index = 1
    for subset in subSets:
        if index%10000 == 0:
            print("处理第",index,"项")
        index += 1
        supCnts = 0
        for trans in transactions:
```

```
            if subset.issubset(trans):
                supCnts += 1
        if supCnts >= minSup:
            supDict[subset] = supCnts/transactions.__len__()

    for k,v in supDict.items():
        print(list(k),"->",v)

    print("频繁项集个数", supDict.__len__())
```

4.3 实验结果和数据

分别更改最小值支持度minSup和最小值置信度minConf，得到如下实验对比结果：

(1) Grocery Store 数据集

	minSup	minConf	频繁项集数	规则数	时间(s)	内存(MB)
Apriori	0.05	0.05	31	6	0.8617	53.34
FP-Growth			31	6	0.7221	62.68
Apriori	0.1	0.05	8	0	0.3730	53.45
FP-Growth			8	0	0.1974	54.49
Apriori	0.02	0.05	122	128	4.1768	54.47
FP-Growth			122	128	1.2437	72.08
Apriori	0.01	0.05	333	499	7.3703	53.73
FP-Growth			333	499	1.9608	66.59
Apriori	0.002	0.05	4223	6506	62.30	55.61
FP-Growth			4226	6506	4.0152	72.45
Apriori	0.002	0.1	4223	3089	70.93	54.95
FP-Growth			4226	3089	3.7968	71.59

(2) Unix 用户数据集

	minSup	minConf	频繁项集数	规则数	时间(s)	内存(MB)
Apriori	0.05	0.05	52	68	6.2919	59.65
FP-Growth			52	68	2.8713	63.88
Apriori	0.1	0.05	13	11	5.3157	59.97
FP-Growth			13	11	2.2290	60.41
Apriori	0.02	0.05	288	874	8.527	60.36
FP-Growth			288	874	3.619	70.98
Apriori	0.02	0.02	288	874	9.113	60.00
FP-Growth			288	874	3.017	70.96
Apriori	0.005	0.02	5788	57028	102.69	61.35
FP-Growth			5788	50728	8.2379	72.07

(3) bftest数据集

bftest数据集是从Groceries数据集中选取前7项得到的数据集，主要为了测试暴力算法。因为暴力算法真的太暴力了，指数增长的子集，一旦数据集稍微大了一些，就会被系统killed

	minSup	频繁项集数	时间(s)	内存(MB)
Apriori	0.05	81	0.011	48.13
FP-Growth		81	0.013	48.26
BruteForce		81	18.75	48.07
Apriori	0.01	81	0.0090	48.09
FP-Growth		81	0.0089	48.27
BruteForce		81	19.00	47.97

5.关联规则的挖掘与解读

5.1 Grocery Store 数据集

在该数据集上，经过对`minSup`和`minConf`的调试，找到了一组较为合适的取值

```
minSup = 0.05,minConf = 0.05
#  规则如下
{'other vegetables'} -> {'whole milk'} : 38.68%
{'whole milk'}->{'other vegetables'}:29.28%
{'rolls buns'}->{'whole milk'}:30.79%
{'whole milk'}->{'rolls buns'}:22.16%
{'yogurt'}->{'whole milk'}:40.16%
{'whole milk'}->{'yogurt'}:21.92%
```

观察得到的规则，可以看出基本集中在other vegetables，whole milk，rolls buns，yogurt四项中，分别是蔬菜，牛奶，面包和酸奶，都是食品类的消费品。观察上述结果，不难发现买牛奶的有很大概率会买其他三种，因此可以考虑将这三种商品放在一起，并且将牛奶放在显眼的位置。

5.2 Unix 用户数据集

在该数据集上，经过对`minSup`和`minConf`的调试，找到了一组较为合适的取值

```
minSup = 0.1,minConf = 0.1
#  规则如下，共11条
{'cd'}->{'ls'}:63.87%
{'ls'}->{'cd'}:83.09%
{'rm'}->{'cd'}:87.5%
{'cd'}->{'rm'}:34.28%
{'vi'}->{'cd'}:87.87%
{'cd'}->{'vi'}:46.51%
{'vi'}->{'ls'}:68.55%
{'ls'}->{'vi'}:47.20%
{'vi'}->{'ls', 'cd'}:62.95%
{'cd'}->{'ls', 'vi'}:33.32%
{'ls'}->{'cd', 'vi'}:43.35%
```

观察得到的规则，可以看出基本集中在cd，ls，rm，vi四项中，分别是切换目录，展示，删除和用vi编辑，都是Unix系统中的常见命令。观察上述结果，不难发现在这11条规则中，都是很常见的一些操作。其中置信度最高的三者分别为 vi -> cd,rm->cd,ls->cd，都是在进行编辑、删除、展示目录后切换目录，而且cd和ls的出现频率极高，可以看出Unix命令行界面下，每切换目录都得看看目录下是什么的真实场景。

其次，由最后三项规则，可以归纳出三种行为：

- vi->ls, cd：编辑后先看看当前目录，然后切换
- cd->ls, vi：切换目录后，看文件名，编辑
- ls->cd,vi：展示得到目录后，切换，编辑

显然，这的确是Unix下的常见行为

6.方法的比较与感想

在对于频繁项的挖掘上，由上述Brute-Force同另外两种方法的比较可以看出，直接用项集的所有子集去匹配根本就是天方夜谭。当数据量达到7时，就可以看出速度明显区别，更不用说当面对100以上的数据时，暴力求解基本是不可能完成的。因此才有了Apriori和FP-Growth两种算法的诞生。

对比这两种算法，不难发现在支持度较高的情况下，FP-Growth的速度明显快于Apriori算法，这得益于FP-Growth的频繁模式增长。然而当支持度较低时，若数据量比较小，建树的时间开销也许会让FP-Growth算法慢一些，但两者此时都是在很小段时间完成的，因此差距不大。另外，FP-Growth的内存开销大部分情况下总是高于Apriori算法，这应该是因为建树和不断递归造成的内存资源开销较大。

总体来说，FP-Growth是效率更优的算法，在数据集较大且阈值较小时，基本至少差一个数量级。但是在代码中也可以看到，Apriori的算法扩展性较好，可以用于并行计算，而FP-Growth的树结构容易在并行时出现数据不一致问题。

7.参考资料

[1] 《机器学习实战》人民邮电出版社 Peter Harrington

[2] https://blog.csdn.net/qq_36523839/article/details/82191677

[3] https://blog.csdn.net/qq_36523839/article/details/82250748

[4] https://github.com/Nana0606/python3-fp-growth

[5] https://www.jianshu.com/p/1b3b3a13b558

[6] https://blog.csdn.net/luoganttcc/article/details/80785149