DM-2019 lab2实验报告

161220096 欧阳鸿荣

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

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.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
```

(1) 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
```