華中科技大学

课 程 实 验 报 告

课程名称： 大数据分析

专业班级： **CS1804（交换）**

学　　号： **X2020I1007**

姓　　名： 刘日星

指导教师： 杨　驰

报告日期： **2021 年 12 月 18 日**

计算机科学与技术学院

目录

实验三 关系挖掘实验

3.1 实验目的

1、加深对 Apriori 算法的理解,进一步认识 Apriori 算法的实现；

2、分析 Apriori 算法的缺点，使用 pcy 等变式对 Apriori 算法进行优化。

3.2 实验内容

必做：

1. 实验内容

编程实现 Apriori 算法，要求使用给定的数据文件进行实验，获得频繁项集以及关联规则。

2. 实验要求

以 Groceries.csv 作为输入文件。

输出 1~3 阶频繁项集与关联规则,各个频繁项的支持度,各个规则的置信度，各阶频繁项集的数量以及关联规则的总数。

固定参数以方便检查，频繁项集的最小支持度为 0.005，关联规则的最小置信度为 0.5。

加分项：

1. 实验内容

在 Apriori 算法的基础上，要求使用 pcy 或 pcy 的几种变式 multiHash、multiStage 等算法对二阶频繁项集的计算阶段进行优化。

2. 实验要求

以 Groceries.csv 作为输入文件。

输出 1~4 阶频繁项集与关联规则，各个频繁项的支持度，各个规则的置信度，各阶频繁项集的数量以及关联规则的总数。

输出 pcy 或 pcy 变式算法中的 vector 的值，以 bit 位的形式输出。

参数不变，频繁项集的最小支持度为 0.005，关联规则的最小置信度为 0.5。

3.3 实验过程

3.3.1 编程思路

本次实验通过 4 个部分来实现，分别是使用算法对数据进行预处理、频繁项集的生成、关联规则的生成和调用 Python 库来证实算法的正确性。

第一部分：数据预处理部分

数据预处理部分，将 Groceries.csv 文件内的每一行数据内容通过使用 strip()函数和 split()函数进行格式化分割后读取进程序内，然后计算数据（物品 item）的个数并赋予每个物品 item 一个编号 id。通过字典实现物品名称 item 和 id 的映射转化，即 id2item 和 item2id 这两部分以供后续程序使用。

第二部分：频繁项集的生成

频繁项集生成部分，通过使用 combine 函数功能来根据上一阶的频繁项集构成该阶的候选频繁项集，combine 函数功能如下图所示：

```
def combine(l, n):
    '''
    给定n-1阶频繁项列表，返回待选n阶频繁项列表
    '''
    if n == 2:
        return list(itertools.combinations(l, n))
    if n > 2:
        tmp = []
        for i in range(len(l)-1):
            for j in range(i+1, len(l)):
                s = set(l[i]) & set(l[j])
                if s and len(list(s)) == n - 2:
                    tmp.append(tuple(sorted(set(l[i])|set(l[j]))))

        return list(set(tmp))   # 去重
```

然后通过 reduceFreq 函数和候选频繁项集实现计算该阶的频繁项和支持度。定义一个最小支持度 min_support （0.005），如果计算出来的频繁项的支持度小于 0.005，则将该频繁项舍弃，最后得出支持度大于 0.005 的频繁项集并按支持度从小到大输出。reduceFreq 函数功能具体实现如下图所示：

```python
def reduceFreq(groups_n, n, data, data_num, min_support):
    '''
    计算n阶频繁项及支持度
    '''
    support_n = {g:0 for g in groups_n}

    if n == 1:
        for i in range(data_num):
            if i % 100 == 0:
                print(f'1阶频繁项:  {i+1}/{data_num}...')
            for j in range(len(data[i])):
                k = data[i][j]
                support_n[k] = support_n[k] + 1.0 / data_num
    else:
        for i in range(data_num):
            if i % 100 == 0:
                print(f'{n}阶频繁项:  {i+1}/{data_num}...')
            for g in groups_n:
                if set(g).issubset(set(data[i])):
                    support_n[g] = support_n[g] + 1.0 / data_num

    # 对于低于最小支持度0.005的频繁项集舍弃
    for i in list(support_n.keys()):
        if support_n[i] < min_support:
            del support_n[i]

    return support_n
```

根据附加项的要求，对 2 阶频繁项集生成和支持度的计算使用 Apriori 改进算法——PCY 算法进行改进。为频繁项对创建一个哈希 hash 表，该表只统计 hash 到本桶的项对的个数；然后筛选桶里的数据（频繁项对），对支持度小于 0.005 的候选频繁项进行舍弃，PCY 算法实现如下图：

```python
def PCY(nSub1Freq, n, data, data_num, min_support, nBuckets):
    '''
    利用PCY算法对2阶频繁项及支持度算法计算进行改进——加分项
    '''
    cnt = [0] * nBuckets
    bitmap = [0] * nBuckets
    pairs = []
    pairs2hash = {}

    # Pass 1
    for i in range(data_num):
        for j in range(len(data[i])-1):
            for k in range(j+1, len(data[i])):
                g = tuple([data[i][j], data[i][k]])
                f = hash(g[0], g[1], nBuckets)
                cnt[f] += 1
                if g[0] in nSub1Freq and g[1] in nSub1Freq and g not in pairs:
                    pairs.append(g)
                    pairs2hash[g] = f

    pairs = sorted(pairs)       # 对pairs进行排序
```

PCY 算法第一部分

```
    # Pass 2
    min_cnt = min_support * data_num
    bitmap = [1 if cnt[i] >= min_cnt else 0 for i in range(nBuckets)]
    print(bitmap)

    candidateFreq = []
    for i in range(len(pairs)):
        g = pairs[i]
        if bitmap[pairs2hash[g]] == 1:
            candidateFreq.append(pairs[i])

    support_2 = {}
    support_2_backup = {}
    candidateFreq_num = len(candidateFreq)
    for i in range(candidateFreq_num):
        if i % 100 == 0:
            print(f'2阶频繁项:  {i+1}/{candidateFreq_num}...')
        support = 0
        for j in range(data_num):
            if set(candidateFreq[i]).issubset(set(data[j])):
                support += 1.0 / data_num
        if support >= min_support:
            support_2[candidateFreq[i]] = support

    return support_2
```

PCY 算法第二部分

桶的具体表现为 vector 值 0 和 1，以 bit 位形式输出。通常来说，大规模的数据使用 PCY 算法的计算效率更高。

第三部分：关联规则生成

关联规则生成步骤如下图：

如何挖掘关联规则呢， 分成 下面的步骤：

- **1. 找出所有的频繁项集 I, （相关具体过程我们后面介绍）**

- **2. 规则的产生：**

对于频繁项集的每一个子集 A， 产生一个规则 记作 $A \rightarrow I \setminus A$

根据频繁项集的的点，我们知道，因为I是频繁的，它的任何一个子集A也一定是频繁的

方法1：单次遍历计算规则的置信度， 根据置信度的定义，我们可以计算出所有规则的置信度 例如 confidence(**A,B→C,D**) = support(**A,B,C,D**) / support(**A,B**)

方法2： 观察如果 某一个规则低于置信度阈值，则其对应所有子集也是低于置信度的， 不断迭代

输出： 所有找到的置信度高于阈值的关联规则

通过使用置信度公式计算关联规则的置信度。定义一个最小置信度min_confidence为0.5，若频繁项集的关联规则的置信度计算出来小于最小置信度则将该规则舍弃，算法实现具体如下图：

- **Association Rules:**
  If-then rules about the contents of baskets
- $\{i_1, i_2,\ldots,i_k\} \rightarrow j$ means: "if a basket contains all of $i_1,\ldots,i_k$ then it is ***likely*** to contain j"
- **In practice there are many rules, want to find significant/interesting ones!**
- ***Confidence*** of this association rule is the probability of j given $I = \{i_1,\ldots,i_k\}$

$$\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$$

generateRules 函数作用是遍历每个频繁项，对于每个非单元素的元组调用appendRule 函数，因为单元素不可产生关联规则。

```
def generateRules(support, min_confidence):
    '''
    生成关联规则
    '''
    rules = {}
    keys = list(support.keys())
    for k in keys:
        if type(k) != int:      # 不需要考虑单元素问题
            rules = {**rules, **appendRule([set(k), set()], support, min_confidence)}
    return rules
```

appendRule 函数接收k、support、min_confidence 三个参数，其中k 的形式举例如下：[{0,1,2,3}, {}]，即由两个set 组成的列表，两个set 分别为前件和后件，在appendRule 函数内部，会通过循环遍历所有将前件中的元素移动到后件的可能，并计算前件->后件的置信度，如果满足min_confidence 的要求，才会进一步递归调用，从而实现剪枝的思路。进一步举例，k 从前件中将2 移动到后件，则 tmp 为[{0,1,3},{2}]，经过计算，013->2 的置信度满足要求，那么tmp 将作为形参k递归调用。

```python
def appendRule(k, support, min_confidence):
    '''
    对列表k生成全部高于置信度0.5的关联规则
    '''
    if len(k[0]) == 1:
        return {}
    rules = {}
    union = tuple(sorted(k[0] | k[1]))
    if union not in support.keys():
        return {}
    for i in k[0]:
        tmp = copy.deepcopy(k)
        tmp[0].remove(i)
        tmp[1].add(i)
        tmp[0], tmp[1] = tuple(sorted(tmp[0])), tuple(sorted(tmp[1]))

        # 将单元素元组转化为int整型作为key
        tmp[0] = tmp[0][0] if len(tmp[0]) == 1 else tmp[0]
        tmp[1] = tmp[1][0] if len(tmp[1]) == 1 else tmp[1]

        if tmp[0] not in support.keys():
            continue

        confidence = support[union] / support[tmp[0]]
        if confidence >= min_confidence - 1e-6:
            rules[(tmp[0], tmp[1])] = confidence
            tmp[0] = tuple([tmp[0]]) if type(tmp[0]) == int else tmp[0]
            tmp[1] = tuple([tmp[1]]) if type(tmp[1]) == int else tmp[1]

            rules = {**rules, **appendRule([set(tmp[0]), set(tmp[1])], support, min_confidence)}
    return rules
```

第四部分：调用 Python 库 mlxtend 进行验证

调用 python 的 mlxtend 库对 Groceries.csv 文件进行计算生成最终的总频繁项集数和关联规则数，用该两值与 Apriori 算法计算出来的频繁项集数和关联规则数进行对比验证是否相同，具体实现如下：

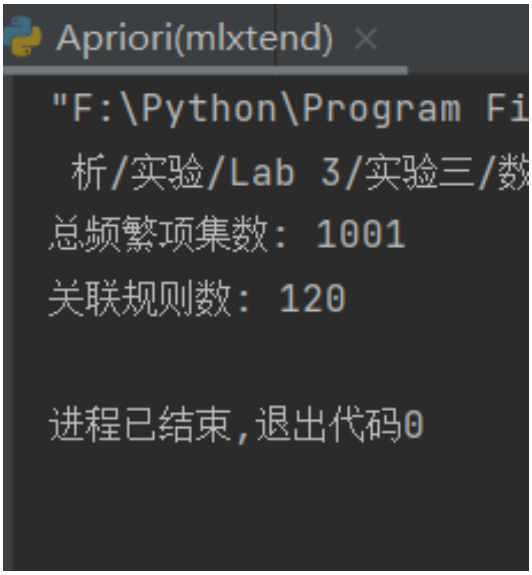
```python
# 求频繁项集
freq = apriori(df, min_support=0.005, use_colnames=True, max_len=4)
freq.sort_values(by='support', ascending=False, inplace=True)
print(f'总频繁项集数: {len(freq)}')

# 求关联规则
rules = association_rules(freq, metric='confidence', min_threshold=0.5)
rules.sort_values(by='confidence', ascending=False, inplace=True)
#rules.to_csv('rules.csv', index=False)
print(f'关联规则数: {len(rules)}')
```

3.3.2 遇到的问题及解决方式

在本次实验中出现过一些问题，例如对数据预处理时，在 combine 函数内，n-1 阶频繁项集组合可能会生成相同阶的频繁项，导致频繁项数目不正确，所以在最后做了个去重的动作。其次时频繁项的支持度字典的键 key 需要使用到元组 tuple 的形式去表示 id 的组合，例如（25,36,48），那么（36,48,25）与前者不相同，所以每个元组 tuple 项都需要维护好有序的状态。

3.3.3 实验测试与结果分析

分别输出 1-4 阶的频繁项集数以及关联规则的总数，将结果与 python 的 mlxtend 库生成的结果进行对比，如下两图：



最后将 1~4 阶的频繁项集与关联规则（包含各频繁项的支持度和各规则的置信度）按升序输出到 output.txt 文件内，如下两图：

```
----- 1阶频繁项集数: 120 -----
liver loaf : 0.005083884087442804
cleaner : 0.005083884087442804
curd cheese : 0.005083884087442804
spices : 0.00518556176919166
jam : 0.005388917132689372
sauces : 0.0054905948144382275
softener : 0.0054905948144382275
sparkling wine : 0.005592272496187083
cereals : 0.005693950177935939
dental care : 0.005795627859684795
kitchen towels : 0.005998983223182507
female sanitary products : 0.006100660904931362
vinegar : 0.006507371631926786
finished products : 0.006507371631926786
soups : 0.006812404677173353
```

```
----- 关联规则数: 120 -----
whipped/sour cream, root vegetables,  -> other vegetables : 0.4999999999999985
brown bread, other vegetables,  -> whole milk : 0.4999999999999986
rolls/buns, beef,  -> whole milk : 0.49999999999999867
pork, root vegetables,  -> whole milk : 0.49999999999999867
yogurt, root vegetables,  -> other vegetables : 0.499999999999999
rolls/buns, frozen vegetables,  -> whole milk : 0.49999999999999967
root vegetables, frankfurter,  -> whole milk : 0.49999999999999967
rolls/buns, root vegetables,  -> other vegetables : 0.502092050209204
yogurt, tropical fruit, whole milk,  -> other vegetables : 0.503355704697985
root vegetables, newspapers,  -> whole milk : 0.5044247787610612
root vegetables, curd,  -> other vegetables : 0.5046728971962611
yogurt, fruit/vegetable juice,  -> whole milk : 0.5054347826086942
whipped/sour cream, other vegetables,  -> whole milk : 0.5070422535211259
tropical fruit, pastry,  -> whole milk : 0.5076923076923064
rolls/buns, root vegetables, other vegetables,  -> whole milk : 0.5083333333333323
whipped/sour cream, domestic eggs,  -> other vegetables : 0.5102040816326527
oil, other vegetables,  -> whole milk : 0.5102040816326527
root vegetables, domestic eggs,  -> other vegetables : 0.5106382978723389
butter, root vegetables,  -> other vegetables : 0.511811023622046
yogurt, other vegetables,  -> whole milk : 0.5128805620608942
whipped/sour cream, yogurt, whole milk,  -> other vegetables : 0.5140186915887844
tropical fruit, curd,  -> other vegetables : 0.5148514851485144
tropical fruit, curd,  -> yogurt : 0.5148514851485144
```

3.4 实验总结

本次实验我认为是大数据分析前三个实验里，算法最难实现的。因为对 PCY 算法的不熟悉，加上网上的 PCY 算法资源特别少，就很难将实验进行下去。到后面参考了大量资料后，一点一点调试修改完成出来，但是性能比 mlxtend 库差了很多。所以在后面的实验里还需再接再厉，提高自己对 python 编程语言的掌握和熟练度，对大数据分析的算法加深了解。