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： 物联网 1801 班

学　　号： U201814500

姓　　名： 王英嘉

指导教师： 崔金华

报告日期： 2020.12.16

计算机科学与技术学院

目录

实验三 关系挖掘实验

3.1 实验内容

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

3.2 实验过程

3.2.1 编程思路

本实验主要包括三部分，分别是数据预处理、频繁项集的生成和关联规则的生成。

数据预处理方面，将 Groceries.csv 中的数据每一行进行拆分成 item 组成的列表并映射成唯一 id，维护 id2item 和 item2id 两个字典供后续使用。

频繁项集生成方面，首先根据上一阶的频繁项集构成该阶的备选频繁项集，这里使用到了实现的 combine 函数，代码部分如图 1 所示。

```python
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
```

图 1 combine 函数

之后利用生成的备选频繁项集计算频繁项及支持度，这里通过 reduceFreq 函数实现，代码部分见图 2，最后把低于 min_support 的备选频繁项集舍弃即为所求的 n 阶频繁项集。

```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('reduceFreq1 %d/%d ... ' % (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('reduceFreq%d %d/%d ... ' % (n, 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

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

    return support_n
```

图 2 reduceFreq 函数

根据附加要求可以使用 PCY 算法进行改进，PCY 算法的基本原理为：将不同的频繁项组成的有序对通过 hash 函数映射不同的桶里并计数，如果桶的支持度低于最小支持度，那么桶里的数据一定都不频繁，从而实现备选频繁项集的初次筛选，之后思想与普通算法一致，对筛选后的备选频繁项集计算支持度并舍弃低于 min_support 的部分。

```python
def PCY(nSub1Freq, n, data, data_num, min_support, nBuckets):
    '''
    计算2阶频繁项及支持度(经过PCY算法改进)
    '''
    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排序
```

图 3 PCY 函数第一部分

```python
# 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('reduceFreq2 %d/%d ...' % (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
```

图 4 PCY 函数第二部分

值得注意的是，桶的数目 nBuckets 是一个超参，不同值对结果影响较大，生成的 bitmap 可以反映桶的数目是否合适。考虑极端情况，桶太满或太空效果都不好，如果所有桶的 bit 都为 1，相当于没有 item 被筛选出去，如果只有一个桶的 bit 为 1，也说明几乎所有 item 都映射到了这个桶。当 nBuckets=1000 时 bit vector 示例如图 5，此时 1 和 0 相对均衡，效果也比较好。

另外经过一些测试发现，PCY 算法效果还和数据规模有很大关系，对大规模数据来说 PCY 算法效果更好一些。

图 5 bitmap vector

关联规则生成方面，流程见图 6 所示，此处不进行赘述，代码部分见图 7 和图 8 所示。

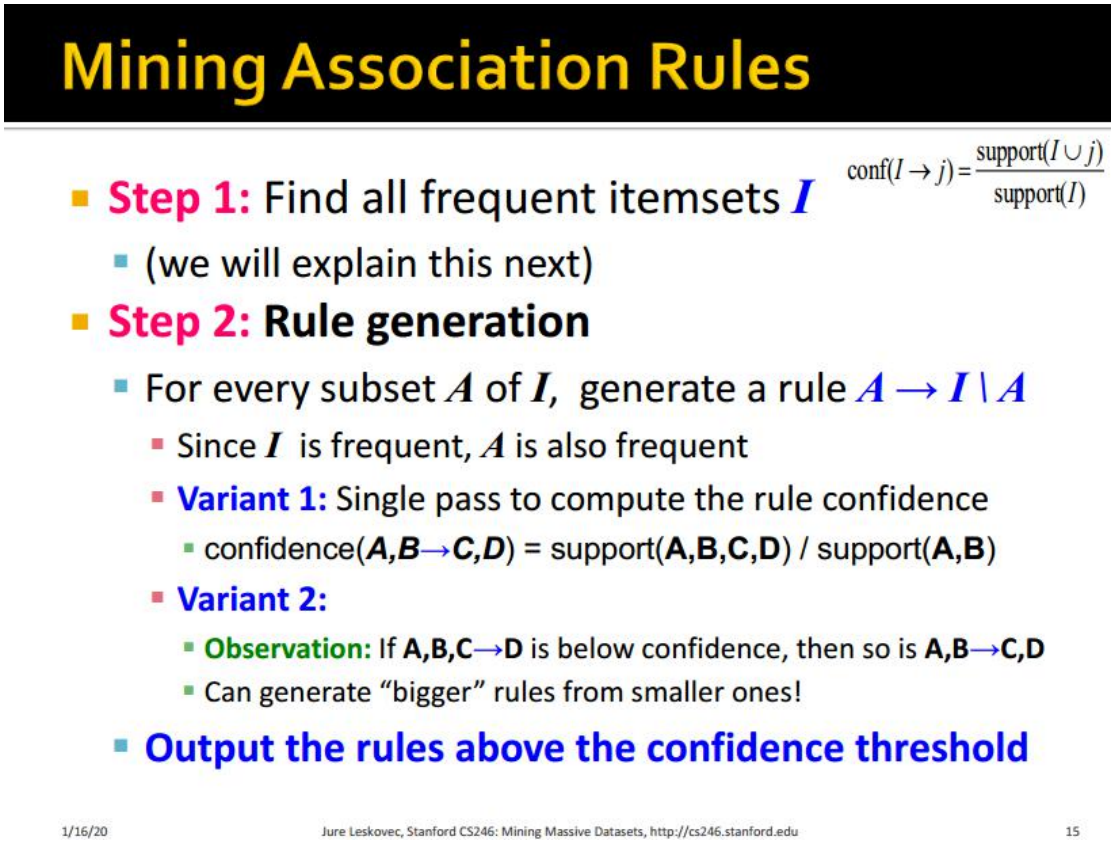

图 6 关联规则生成流程

给定一个频繁项到支持度的字典 support 和最小置信度 min_confidence，generateRules 函数遍历频繁项，对于每一个不是单元素的元组（显然单元素不可能产生关联规则），调用 appendRule 函数。

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

    return rules
```

图 7 generateRules 函数

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作为键
        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: # 很重要！！！！
            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
```

图 8 appendRule 函数

3.2.2 遇到的问题及解决方式

这个实验中出现了比较多的问题，下面举三个典型问题进行说明：

其一，由于频繁项-支持度字典的键是用 tuple 形式表示的 id 组合，如(25,36,48)，那么(36,48,25)和前者是不一样的，所以对于每个 tuple 都要小心维护好有序状态，否则在比较中会出现问题。

其二，在 combine 函数的实现中，不同的 n-1 阶频繁项组合可能生成了相同的 n 阶频繁项，所以结果要进行去重，否则得到的频繁项数目不正确。

其三，关联规则生成方面，一开始使用暴力搜索得到了不正确的结果，第二

次使用了剪枝树来实现仍然得到了不正确的结果，最后经过了大量调试发现算法思想本身没有问题，而是因为下面这行语句出现了问题。

```
if confidence ≥ min_confidence - 1e-6: # 很重要！！！
```

图 9 问题语句

猜测由于计算机中浮点数的表示方式的原因，此处比较时即使 confidence 刚好等于 0.5 也可能进不了 if 条件，因此一定要给程序留出一定的“余地”。

3.2.3 实验测试与结果分析

首先输出 1-4 阶频繁项集的数量以及关联规则的总数如图 10 所示，结果可以与图 11 中 mlxtend 库生成的结果进行对比，可知算法实现基本正确。

```
终端  问题  输出  调试控制台

reduceFreq4 9401/9835...
reduceFreq4 9501/9835...
reduceFreq4 9601/9835...
reduceFreq4 9701/9835...
reduceFreq4 9801/9835...
120 605 264 12
1001
120
```

图 10 Apriori.py 终端输出

```
(torch) yingjia@yingjia-Vostro-5370:~/文档/vscode/python/lab3$ python Apri
ori\(mlxtend\).py
1001
120
```

图 11 Apriori(mlxtend).py 终端输出

1~4 阶频繁项集与关联规则，包括各个频繁项的支持度，各个规则的置信度已经输出到了 output.txt 中，此处不再展示。

3.3 实验总结

关联规则实验我觉得是四个实验中最难的一个实验，一方面在于本身实验原理之前没有接触过，PCY 更是资料特别少，另一方面实验对 python 需要很高的熟练度，踩了不少坑，即使完成了之后性能也和 mlxtend 库差很多。为此我还特

意去查看了 mlxtend 中的源码，不过这使我对原理的掌握以及 python 的熟练度都有了不小的提高。

最后感谢助教对我实验过程中的启发与帮助。