实验报告

一、封面页

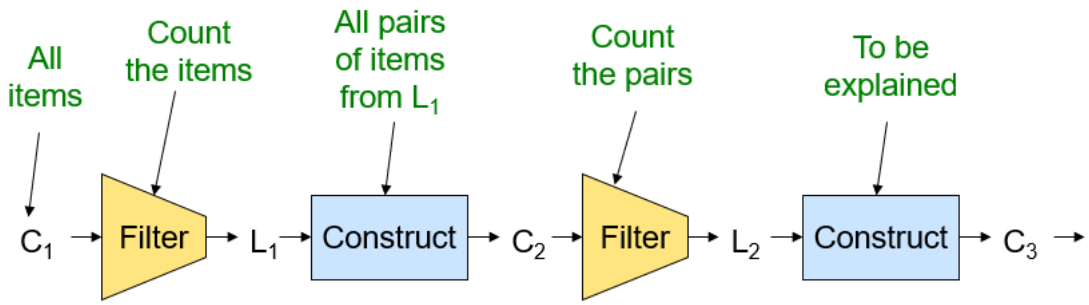
项目标题	实现Apriori算法的频繁项集挖掘
项目编号	5
实验人	毕泽同_21307129
截止日期	2024年5月12日24：00
提交日期	2024年5月12日

摘要

在本次实验中，我实现了Apriori算法，在给定的GroceryStore购买记录数据集(存储在Groceries.csv)，完成了算法测试，并对不同的support threshold下的结果进行了分析。

二、理论基础

Apriori是一种常用的数据关联规则挖掘方法，它可以用来找出数据集中频繁出现的数据集合。找出这样的一些频繁集合有利于决策，例如通过找出超市购物车数据的频繁项集，可以更好地设计货架的摆放。需要注意的是它是一种逐层迭代的方法，先找出频繁1项集L1，再利用L1找出频繁2项集，以此类推……



首先，我们应明确一些基本概念:

1. 关联规则
   关联规则是形如X=>Y的蕴含式，其中X、Y分别是一事务的真子集，且$X \cap Y = \phi$。X称为规则的前提，Y称为规则的结果。关联规则反映出X中的项目在事务中出现时，Y中的项目也跟着出现的规律。
2. 支持度
   关联规则的支持度是事务集中同时包含X和Y的事务数量与所有事务数量之比，它反映了X和Y中所含的事务的项在事务集中同时出现的频率，记为$support(X => Y)$，即$support(X => Y) = support(X \cup Y) = P(XY)$
3. 置信度
   关联规则的置信度是事务集中同时包含X和Y的事务数量与包含X的事务数量之比，记为$confidence(X => Y)$，置信度反映了包含X的事务中出现Y的条件概率。即$confidence(X => Y) = support(X \cup Y)/support(X) = P(XY)$
4. 频繁项集
   设$U \in I$，项目集U在事务集T上的支持度是包含U的事务在T中所占的百分比，即$support(U) = ||t \in T|U \in t||/||T||$。式中，|| ... ||表示集合中的元素数目。对项目集I，在事务数据库T中所有满足用户指定的最小支持度的项目集，即不小于最小支持度阀值的I的非空子集，称为频繁项目集或大项目集。
   频繁项集性质：1. 频繁项集的所有非空子集也为频繁项集； 2. 若A项集不是频繁项集，则其他项集或事务与A项集的并集也不是频繁项集

在本次实现的Apriori算法中，我们搜索频繁项集的流程如下所示:

1. 生成候选项集
2. 通过遍历每一个buckets，得到每一个候选项集的支持度
3. 通过最小支持度阈值过滤掉部分候选项集，剩下的集合即为频繁项集，可以进行保存
4. 将刚刚得到的频繁项集逐个，与所有长度为1的频繁项集进行拼接，即可得到长度加1的新的候选项集

5. 继续计算新的候选项集的支持度，直到得不到频繁项集。

通过上面的步骤，最终我们就能得到各个长度的频繁项集。
在实现时，可以考虑还可以考虑利用频繁项集的性质进行剪枝。

三、代码解析

```
def apriori(dataset, min_support):
    # 参数列表
    frequent_itemsets = []
    # 候选项集 (n,1)
    candidate_itemset = np.arange(dataset.elements_num).reshape(-1, 1)
    while True:
        # 支持度汇总(n,)
        sum_support = np.zeros(candidate_itemset.shape[0])
        # 遍历每个bucket，得到每个候选项集的支持度
        for event in tqdm(
            dataset,
            desc=f"Getting support scores, candidate len={len(frequent_itemsets)+1}",
        ):
            sum_support += np.all(np.isin(candidate_itemset, event), axis=1)
        # 提取出支持度大于阈值的候选项集作为频繁项集
        k_itemsets = candidate_itemset[sum_support >= min_support]
        if len(k_itemsets) == 0:
            break
        else:
            frequent_itemsets.append(k_itemsets)
            # 构建新的候选项集集合，其首先为set，便于剔除重复的候选项集
            candidate_itemset = set()
            # 遍历每个刚得到的频繁项集，构建新的候选项集
            for k_itemset in tqdm(
                k_itemsets,
                desc=f"Getting candidate itemsets,new candidate len={len(frequent_itemsets)+1}",
            ):
                # 找到不在该频繁项集中的元素
                new_elements = np.setdiff1d(frequent_itemsets[0], k_itemset)
                # 利用这些元素，构建新的候选项集
                for new_element in new_elements:
                    new_cadidate = sorted(np.append(k_itemset, new_element))    # 对候选项集进行内部排序，来去除重复的候选项集
                    # 这里可以利用不成功的项集进行剪枝，而不是直接导入
                    candidate_itemset.add(tuple(new_cadidate))
            # 将每个元组转换为 NumPy 数组并添加到列表中
            candidate_itemset = [np.array(tensor) for tensor in candidate_itemset]

            # 将列表转换为 NumPy 张量
            candidate_itemset = np.array(candidate_itemset)

    return frequent_itemsets
```

核心代码如上所示，在具体实现时需要将数据的I/O及预处理操作集成到dataset中。

四、实验结果

4.1 不同阈值下的结果差异

频繁项集数目分布情况

阈值	长度范围	总计	长度=1	长度=2	长度=3	长度=4	长度=5	长度=6	长度=7	长度=8	长度=9	长度=10
3	[1,10]	166682	166	6246	37848	62293	40850	14802	3711	675	85	6
4	[1,9]	80388	166	5425	25985	31620	13940	2896	335	20	1	0

阈值	长度范围	总计	长度=1	长度=2	长度=3	长度=4	长度=5	长度=6	长度=7	长度=8	长度=9	长度=10
5	[1,7]	49580	164	4782	19226	18597	5988	785	38	0	0	0

阈值提升，频繁项集数目减少比例

阈值	长度范围变化	总计	长度=1	长度=2	长度=3	长度=4	长度=5	长度=6	长度=7	长度=8	长度=9	
3-4	1	51.772%	0.0%	13.144%	31.344%	49.24%	65.875%	80.435%	90.973%	97.037%	98.824%	1
4-5	2	38.324%	1.205%	11.853%	26.011%	41.186%	57.044%	72.894%	88.657%	100.0%	100.0%	

在频繁项集数目方面，我们可以观察到以下现象:

1. 随着长度的增加，每种长度减少的比例则在增加
2. 整体减少的趋势比较平稳，没有出现突变的情况
3. 在数目分布上，中等长度(3,4,5)占的比例非常高，这意味着计算这些长度的频繁项集时要花费更多的时间。但同时阈值提升时对这些长度的数目也比较大，可以减少近一半的数目，这可能是减少程序运行时间的关键因素。

运行时间

阈值	3	4	5
时间	17201.13s	6061.59s	4246.56s
	4.778h	1.683h	1.179h
频繁项集数目	166682	80388	49580
频繁项集数目/时间	34,885.30	47,764.70	42,052.58

可以看到程序性能并不高，在阈值为3时的性能表现非常差，这有可能是过大的规模导致的，规模较大时其剪枝所需要耗费的时间也越多，所以减低了其性能。

4.2 剪枝算法探索

上面的性能是非常难以接受的，为了缩短算法运行时间，我们可以考虑两种方法：

1. 改变时间方式，使用其他语言或其他方法
2. 引入剪枝方法

在本次实验中，我采用了第二种方法。

这里，我考虑到了在构建新候选项集时的优化可行性。在构建新候选项集时，我们依次将频繁项集中没有的项与频繁项集进行结合，尝试构建新的候选项集。但此时我们所构建的新候选项集的子集可能已经被确认为了非频繁项集。根据频繁项集的性质，这就排除了其为频繁项集的可能性。同时也为我们剪枝提供了可能性。

在具体实现层面，我们需要将所有非频繁项集保存起来，并在生成候选项集时与这些非频繁项集进行比对，如果有非频繁项集是候选项集的子集，那么我们就可以直接进行剪枝。

这样做可以减少候选项集的数目，减小了时间和空间方面的开销。并且可以论证所记录的长度较短的非频繁项集一定不是长度更长的非频繁项集的子集，这样对于非频繁项集的存储开销也有了一定程度的保证。

阈值	3	4	5
未使用剪枝方法	17201.13s	6061.59s	4246.56s
使用剪枝方法	9623.68s	3814.44s	2217.97s
加速比	44.052%	37.072%	47.77%

可以看到使用剪枝算法以后，程序性能有了40%的提升，且实验结果与未剪枝的方法完全一致，这足以说明剪枝方法的有效性。

五、总结

在本次实验中，我实现了Apriori算法，通过设置不同的支持度阈值对频繁项集结果进行了分析，同时通过一定的剪枝算法对程序运行效率进行了改进。

六、附录

关联规则挖掘算法——Apriori算法原理