数据科学基础第五次作业

王晨曦

2018 年 5 月 27 日

1 问题重述

研究数据集的关联规则。

- *Python* 编程实现 Apriori 算法，能够从交易数据集发现频繁项集，并生成关联规则；
- 生成一个交易数据集，验证算法实现的正确性。

2 Apriori 算法

Apriori 算法是数据挖掘领域的经典算法，用来寻找商品交易中的关联规则，与朴素的蛮力算法相比，Apriori 算法**通过剪枝有效的缩小了搜索空间的范围，从而大大缩短了程序运行的时间**，其详细过程如下：

- 从数据集 $\mathcal{D}$ 中找出频繁 1-项集（全体记为 L_1）作为初始项集。
- 对于频繁 k-项集 L_k，通过自连接得到频繁 $(k+1)$-项集的候选集合 C_{k+1}，连接方法如下：将 L_k 中的所有项按照字典序排列，将所有满足仅最后一项不同的两个项集连接起来得到 C_{k+1}。
- 扫描原始数据集，找出 C_{k+1} 中所有频繁 $(k+1)$-项集，构成 L_{k+1}。
- 重复以上两步，直到找出所有频繁项集。
- 对于找出的频繁项集，生成可能的关联规则，计算置信度，保留大于阈值的结果，得到数据集 $\mathcal{D}$ 的关联规则。

与蛮力算法相比，Apriori 算法大大减少了程序的时间复杂度和空间复杂度，下面我们将分别在自建数据集和公开数据集上运用 Apriori 算法分析数据，并与蛮力算法进行对比。

3 数据集介绍

3.1 自建数据集

为了验证 Apriori 算法的效果，我首先自建了一个数据集，包含了十种类别的商品：cup、thermometer、calculator、pencil、notebook、egg、soap、shampoo、cake 和 milk，均为商店中的常见物品。为了使订单中的商品关联更符合实际，根据日常生活的常识，我在生成订单的时候设置了一个后验概率分布表，基于当前购买的商品和后验概率值选择下一个商品，进

而生成完整的订单。如图1中的热度图所示，方格 (i,j) 中的颜色代表购买第 i 个商品后购买第 j 个商品的概率，颜色越红，概率越大，无色方格表示后验概率为0。详细的概率分布表见附录 B。

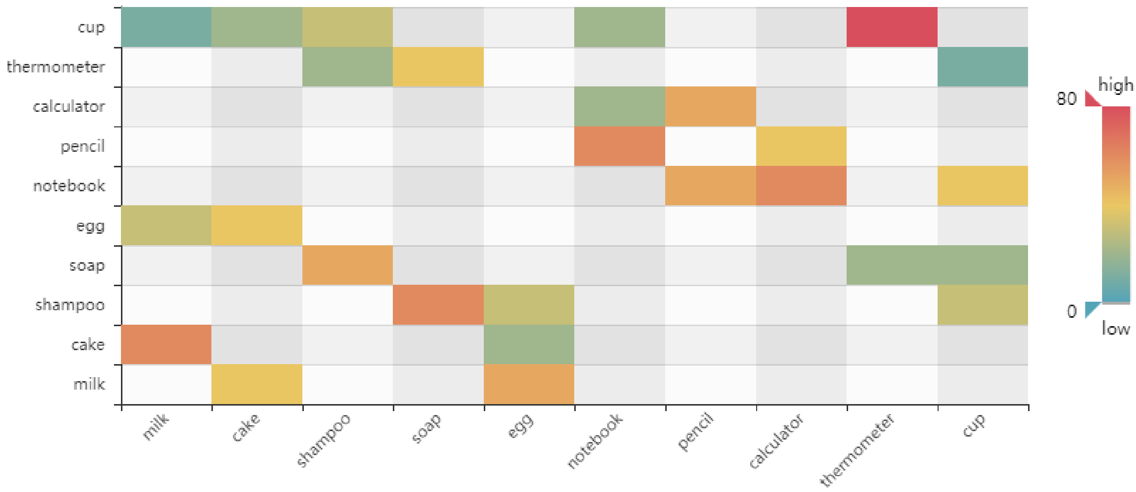

图 1: 商品后验概率分布

模拟订单的生成在概率分布表的基础上添加了随机购买商品的逻辑，保证生成订单的多样性，一条订单的生成流程如下：

- 随机生成此条订单中商品种类总数 $k(1 \leqslant k \leqslant 10)$。
- 随机产生第一个商品。
- 生成下一个商品：其中有 p 的概率根据后验分布表的规律生成商品，$1-p$ 的概率从所有商品中随机挑选一种商品。
- 持续生成商品，直至订单中有 k 种不同的商品。

实验中设定 $p=0.8$，生成了 100 条交易订单。可以看出生成的数据集规模比较小，即使用蛮力算法也能快速跑出结果，便于检验 Apriori 算法的正确性。

3.2 公开数据集

除了规模较小的自建数据集，我还采用了公开数据集 Groceries 测试 Apriori 算法的结果。Groceries 数据集是 R 语言包自带数据集，包含了 9835 条交易订单数据，涵盖了 169 个商品种类，更加贴近真实的交易情况，因此可以用来检测关联规则算法的有效性。在实验中，我将 Groceries 数据集导出成 csv 文件，再读近程序进行解析，以便后续的实验和分析。

4 问题解答

4.1 算法实现及结果展示

我用 *Python* 实现了 Apriori 算法和蛮力算法，以便后续的实验和对比，Apriori 算法和蛮力算法尽在寻找频繁项集这一步有所差别，详细代码见附录 A，所有实验代码和结果保存

在 hw5.ipynb 中。

下面两小节分别展示了 Apriori 算法在自建数据集和 Groceries 数据集上的表现。

4.1.1 自建数据集

首先来验证实现的 Apriori 算法的正确性。在这里我选择了支持度阈值为 0.35，置信度阈值为 0.85 进行实验。实验结果表示，运行 Apriori 算法和蛮力算法得到了相同的关联规则，验证了解答的正确性，结果如表1所示，其中括号内的数值表示对应频繁项集的支持度。

表 1: 自建数据集关联规则

关联规则	置信度
soap (0.56) $\Rightarrow$ shampoo (0.62)	0.8571
notebook, shampoo (0.41) $\Rightarrow$ cup (0.65)	0.8780
egg, soap (0.40) $\Rightarrow$ cup (0.65)	0.8750
egg, shampoo (0.43) $\Rightarrow$ cup (0.65)	0.8605
cake, milk (0.42) $\Rightarrow$ egg (0.59)	0.8810
egg, milk (0.42) $\Rightarrow$ cake (0.58)	0.8810
egg, notebook (0.38) $\Rightarrow$ cup (0.65)	0.9211
soap, thermometer (0.40) $\Rightarrow$ cup (0.65)	0.8750
notebook, soap (0.40) $\Rightarrow$ cup (0.65)	0.8750
notebook, soap (0.40) $\Rightarrow$ shampoo (0.62)	0.8750
notebook, shampoo (0.41) $\Rightarrow$ soap (0.56)	0.8537
egg, thermometer (0.37) $\Rightarrow$ cup (0.65)	0.9459

可以发现，即使置信度阈值高达 0.85，还是能找到如此多的关联规则，这主要是由于数据集规模和制定的生成规则所致。由于数据集规模太小，商品种类有限，而后验概率表使得购买商品的规则性太强，所以商品之间的关联很大。在真实的数据集里，这种现象很难找到。

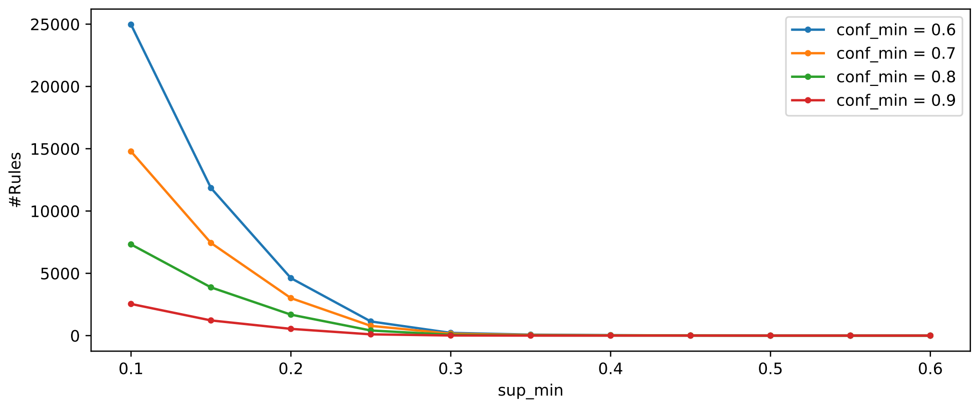

图 2: 关联规则数随支持度和置信度阈值的变化曲线

图2展示了关联规则数随支持度阈值和置信度阈值变化的情况。可以看到，规则数的下降程度随着支持度阈值升高变化幅度很大，不论置信度阈值如何，当支持度阈值超过 0.4 以

后就很难找到符合要求的关联规则了。随着置信度阈值的升高，符合要求的项集对越来越少，因而能找到的关联规则也会逐渐下降。

4.1.2 公开数据集

对于 Groceries 数据集，我在支持度阈值为 0.05，置信度阈值为 0.1 时运行 Apriori 算法得到了如表2所示的结果，括号内的数值代表了频繁项集的支持度。

表 2: 自建数据集关联规则

关联规则	置信度
other vegetables (0.19) ⇒ whole milk (0.26)	0.3868
whole milk (0.26) ⇒ other vegetables (0.19)	0.2929
rolls/buns (0.18) ⇒ whole milk (0.26)	0.3079
whole milk (0.26) ⇒ rolls/buns (0.18)	0.2216
yogurt (0.14) ⇒ whole milk (0.26)	0.4016
whole milk (0.26) ⇒ yogurt (0.14)	0.2193

对比表1与表2，我们可以看出在 Groceries 数据集上频繁项集的支持度和关联规则的置信度大幅降低。为了进一步验证这个现象，可以在不同的置信度阈值和支持度阈值下统计关联规则的数量，如图4所示。

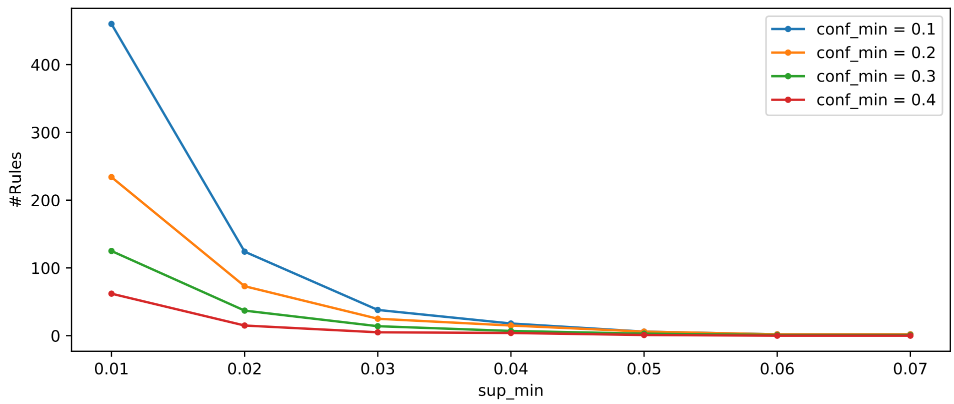

图 3: 关联规则数随支持度和置信度阈值的变化曲线

可以看出，后者生成的关联规则远不如前者，当支持度阈值超过 0.1 时就很难再找到符合条件的关联规则了。这种现象产生的原因是因为数据集规模太大，商品出现的频率不如自建数据集，在真实情况下商品之间的关联程度受各方面复杂因素影响，从而使得商品的支持度和规则的置信度都大大降低了。

4.2 商品关联程度

对于自建数据集，我们可以进一步探索 Apriori 算法得到的关联规则与预先设定好的后验概率之间的关系。在 Apriori 算法中，将置信度阈值设置为 0，即可得到频繁项集间所有的关联规则置信度，选择其中的一对一规则，将置信度绘制成热力图，可以得到如图4所示的结果。

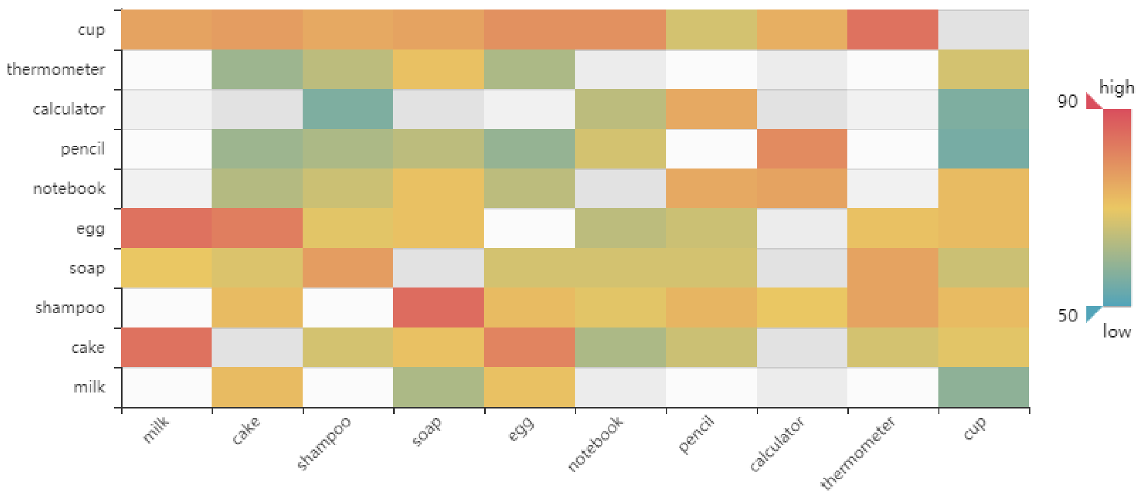

图 4: 商品关联程度

对比图1和图4，我们可以看到生成的关联规则中，置信度较高的项大多在后验分布中的概率也很大，从这个间接说明了购买商品时的因果关系会影响得到的关联规则。

4.3 算法运行时间

Apriori 算法相较于蛮力算法，最突出的优势在于通过剪枝，减少了搜索空间的范围，从而即为有效的降低了算法运行的时间。为了验证这个推论，我利用数据集生成算法产生了交易数不同的数据集，分别用两种算法搜索关联规则，并记录程序运行的时间，得到了如图5所示的结果。

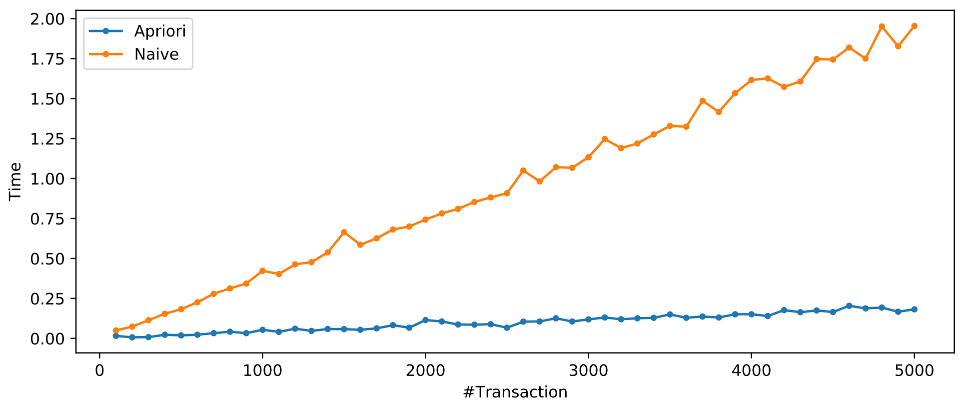

图 5: 算法运行时间对比

可以发现，随着交易数量的大幅增加，蛮力算法的运行时间显著增长，而 Apriori 算法运行时间变化幅度较小，达到了跟理论分析相符的效果。

5 结论

本次实验中，我实现了 Apriori 算法，通过与蛮力算法的运行结果相比较，验证了程序的正确性，得到了数据集中的关联规则。关联规则与置信度阈值和支持度阈值有很大的关系，在实验中通过改变这两个参数，得到了变化的曲线图。通过 Apriori 算法，还可以发现交易订单数据中商品之间的联系紧密程度。Apriori 算法通过剪枝大大降低了时间复杂度，这一点通过与蛮力算法对比运行时间也得到了验证。

A 算法程序源码

```
'''
    关联规则搜索的Apriori算法及蛮力算法实现
    Apriori算法使用demo:
        L, sup_collection = gen_L(dataset, sup_min=0.3)
        rule_list = gen_rules(L, sup_collection, conf_min=0.7)
    蛮力算法使用demo:
        L, sup_collection = gen_L_naive(dataset, sup_min=0.3)
        rule_list = gen_rules(L, sup_collection, conf_min=0.7)
'''

def gen_C1(data):
    '''生成第一个频繁项集候选集合'''
    C1 = set()
    for transaction in data:
        for item in transaction:
            itemset = frozenset([item])
            C1.add(itemset)
    return C1


def gen_Ckplus1(Lk, k):
    '''根据Lk, 生成下一个候选集合'''
    Ckplus1 = set()
    list_Lk = list(Lk)
    for i in range(len(Lk)):
        for j in range(len(Lk)):
            l1 = list(list_Lk[i])
            l2 = list(list_Lk[j])
            l1.sort()
            l2.sort()
```

```python
            if l1[:k-1] != l2[:k-1]:
                continue
            Ckplus1_item = list_Lk[i].union(list_Lk[j])
            if is_candidate(Ckplus1_item, Lk):
                Ckplus1.add(Ckplus1_item)
    return Ckplus1


def is_candidate(Ck_item, Lksub1):
    '''通过子集判断某个候选项集是不是频繁项集'''
    for item in Ck_item:
        tmp_Cksub1 = Ck_item - frozenset([item])
        if tmp_Cksub1 not in Lksub1:
            return False
    return True

def gen_Lkplus1(Ckplus1, data, sup_min, sup_collection):
    '''根据候选集合生成频繁(k+1)项集的集合'''
    Lkplus1 = set()
    item_cnt = {}
    for transaction in data:
        for item in Ckplus1:
            if item.issubset(transaction):
                if item not in item_cnt:
                    item_cnt[item] = 1
                else:
                    item_cnt[item] += 1
    num_transaction = len(data)
    for item in item_cnt:
        tmp_sup = item_cnt[item] / float(num_transaction)
        if tmp_sup >= sup_min:
            Lkplus1.add(item)
            sup_collection[item] = tmp_sup
    return Lkplus1


def gen_L(data, sup_min, k=None):
    '''运用Apriori算法生成全部频繁项集'''
    sup_collection = {}
    C1 = gen_C1(data)
    L1 = gen_Lkplus1(C1, data, sup_min, sup_collection)
    Li = L1.copy()
    L = []
    L.append(Li)
    i = 1
    while True:
        Ciplus1 = gen_Ckplus1(Li, i)
        Liplus1 = gen_Lkplus1(Ciplus1, data, sup_min, sup_collection)
```

```
        if len(Liplus1) == 0:
            break
        L.append(Liplus1)
        i += 1
        Li = Liplus1
        if k is not None and i > k:
            break
    return L, sup_collection


def gen_rules(L, sup_collection, conf_min):
    '''根据找出的频繁项集, 生成相应的关联规则'''
    rule_list = []
    subset_list = []
    for Li in L:
        for freq_i_set in Li:
            for subset in subset_list:
                if subset.issubset(freq_i_set):
                    conf = sup_collection[freq_i_set] / float(sup_collection[
                                                        freq_i_set-subset])
                    rule = (freq_i_set - subset, subset, conf)
                    if conf >= conf_min and rule not in rule_list:
                        rule_list.append(rule)
            subset_list.append(freq_i_set)
    return rule_list


def gen_L_naive(data, sup_min):
    '''生成频繁项集的蛮力算法实现'''
    items = []
    max_num_item = 0
    for t in data:
        for i in t:
            if i not in items:
                items.append(i)
    num_transaction = len(data)
    num_item = len(items)
    sup_collection = {}
    L = []
    L_collection = []
    for i in range(1,1<<num_item):
        tmp_set = set()
        for j in range(num_item):
            if i%2==1:
                tmp_set.add(items[j])
            i = i/2
        tmp_sup = 0
        tmp_set = frozenset(tmp_set)
```

```
        for transaction in data:
            if tmp_set.issubset(transaction):
                tmp_sup += 1
        tmp_sup = tmp_sup/float(num_transaction)
        if tmp_sup >= sup_min:
            sup_collection[tmp_set] = tmp_sup
            L_collection.append(tmp_set)
            if max_num_item < len(tmp_set):
                max_num_item = len(tmp_set)

    for i in range(1,1+max_num_item):
        tmp_Li = set()
        for tmp_set in L_collection:
            if len(tmp_set) == i:
                tmp_Li.add(tmp_set)
        L.append(tmp_Li)

    return L, sup_collection
```

B 自建数据集参数设置

表3中展示了生成数据集时用的后验概率分布表，这是一个稀疏矩阵，大部分值为0。

表 3: 商品后验概率分布

	milk	cake	shampoo	soap	egg	notebook	pencil	calculator	thermometer	cup
milk	0	0.6	0	0	0.3	0	0	0	0	0.1
cake	0.4	0	0	0	0.4	0	0	0	0	0.2
shampoo	0	0	0	0.5	0	0	0	0	0.2	0.3
soap	0	0	0.6	0	0	0	0	0	0.4	0
egg	0.5	0.2	0.3	0	0	0	0	0	0	0
notebook	0	0	0	0	0	0	0.6	0.2	0	0.2
pencil	0	0	0	0	0	0.5	0	0.5	0	0
calculator	0	0	0	0	0.6	0.4	0	0	0	0
thermometer	0	0	0	0.2	0	0	0	0	0	0.8
cup	0	0	0.3	0.2	0	0.4	0	0	0.1	0