大数据分析与知识发现实验报告

西安交通大學

Apriori 算法设计和实现

自动化 94--胡欣盈--2194323176

目　录

一、实验目的

1、编写 Apriori 算法实现对指定数据集关联规则的挖掘；

2、通过编程实现过程，加深对频繁项集、支持度、置信度等概念的理解；

3、通过对比实验分析不同支持度、置信度情况下的实验结果，探究其对关联规则挖掘的影响。

二、算法原理

（1）概念

1.关联规则的一般形式

关联规则的支持度(相对支持度)：项集 A、B 同时发生的概率称为关联规则的支持度(相对支持度)。

$$\text{support}(A \Rightarrow B) = P(A \cup B) = \frac{number(AB)}{number(AllSamples)}$$

关联规则的置信度：项集 A 发生，则项集 B 发生的概率为关联规则的置信度。

$$Confidence(A \Rightarrow B) = P(B \mid A) = \frac{P(AB)}{P(A)}$$

2.最小支持度和最小置信度

最小支持度是衡量支持度的一个阈值，表示项目集在统计意义上的最低重要性。最小置信度是衡量置信度的一个阈值，表示关联规则的最低可靠性。强规则是同时满足最小支持度阈值和最小置信度阈值的规则。

3.支持度计数

项集 A 的支持度计数是事务数据集中包含项集 A 的事务个数，简称项集的频率或计数

一旦得到项集 A、B 和 A∪BA、B 和 A∪BA、B 和 A∪B 的支持度计数以及所有事务个数，就可以导出对应的关联规则，并可以检查该规则是否为强规则。

$$\text{Support}(A \Rightarrow B) = \frac{A,B\text{同时发生的事务个数}}{\text{所有事务个数}} = \frac{\sigma(A \cup B)}{N}$$

$$\text{Confidence}(A \Rightarrow B) = P(B \mid A) = \frac{\sigma(A \cup B)}{\sigma(A)}$$

4.频繁项集

满足最小支持度的所有项集，称作频繁项集。其具有以下性质：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

5.先验原理

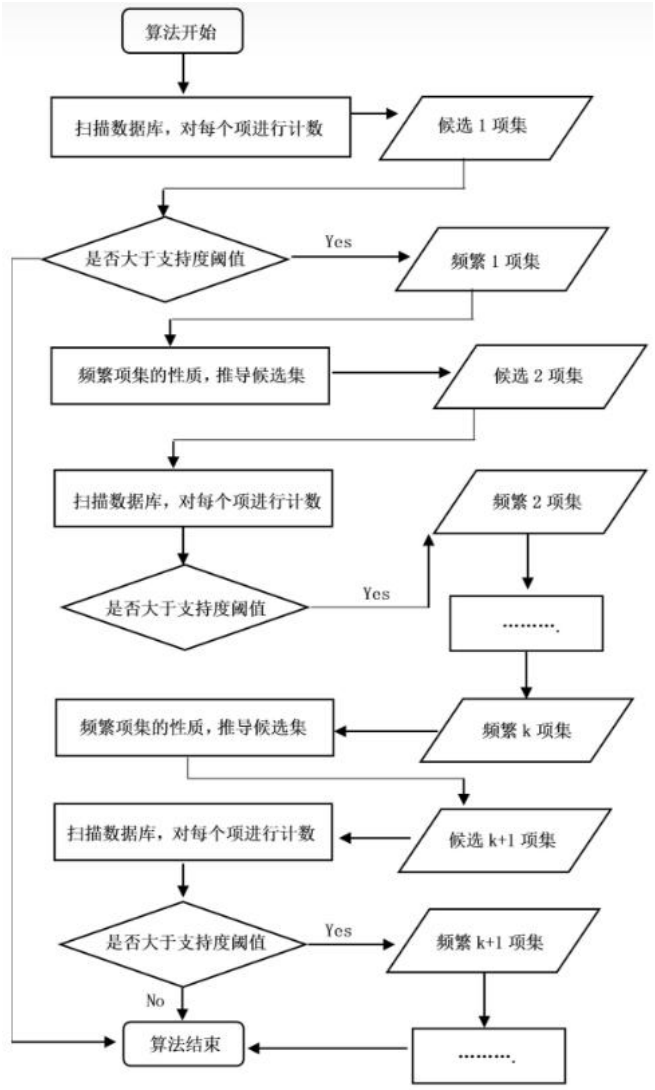
要想获得频繁项集，最简单直接的方法就是暴力搜索法，但是这种方法计算量过于庞大，因此我们要利用支持度对数据集进行剪枝。根据上述频繁项集的两条性质我们可以完成较快的搜索。

（2）算法流程

简述：挖掘频繁项集在逻辑上就是搜索一颗枚举树并计算结点支持率的过程。Apriori 算法利用向下闭包性，在逻辑上对枚举树进行了剪枝操作，当 n 很大时极大的减少了搜索的子集个数，提高了效率。

算法过程：首先，生成长度为 k 的候选项集(k 从 1 开始)，计算它们的支持率；然后，根据支持率生成长度为 k 的频繁项集，这个时

候非频繁的 k 项集被抛弃了；之后，用 k 频繁项集生成长度为（k+1）的候选项集，再计算它们的支持率。如此重复上述过程直到不存在更大的频繁项集为止。其流程图如下图所示。



三、实验内容与结果

（1）数据集介绍与预处理

实验中使用的是 groceries 数据集，该数据集是某个杂货店一个月真实的交易记录，共有 9835 条消费记录，169 个商品。

在此定义了 load_Groceries(filename)函数完成读入数据的操作，得到嵌套列表形式的数据。

```python
def load_Groceries(filename):
    dataset = []
    #使用open函数打开文件
    with open(filename)as f:
        #读出文件内容
        f_csv = csv.reader(f)
        #enumerate枚举函数,返回标签index和元素value
        for index, value in enumerate(f_csv):
            if index != 0:
                #按照','分割
                goodsList = str(value[1]).split(',')
                dataset.append(list(goodsList))
    return dataset
```

数据预处理：因为我编写的函数中是对数字形式的数据进行判断的机制，字符串不适用，所以按照商品列表顺序将数据集中的商品名字符串替换成数字，如下图所示。

```python
good = Groceries[0]
for i in range(1,len(Groceries)):
    for j in range(len(Groceries[i])):
        good.append(Groceries[i][j])
#用集合的形式将重复元素去掉
goodlist = set(good)
goodlist

#按顺序得到商品编号
goodnum = [k for k in range(len(goodlist))]

#将Groceries数据集中的商品名称换成商品标号
dataset = []
for i in range(len(Groceries)):
    #list形式不能使用replace函数,Series和str才行
    data = pd.Series(Groceries[i])
    data.replace(list(goodlist),goodnum,inplace=True)
    dataset.append(list(data))
```

（2）算法编写

3.2.1 定义给候选集计算支持数 sup 的函数

```
def C_sup(dataset,itemset_Ck):
    C_sup = []
    for c in itemset_Ck:
        count = 0  #计数
        for i in range(len(dataset)):
            if set(c) <= set(dataset[i]):  #选择不重复的元素
                count += 1
        C_sup.append([c,count])
    return C_sup
```

3.2.2 定义生成集合子集的生成器对象

```
'''
二进制法枚举所有子集
输入：列表items
输出：需要调用之后循环输出,列表形式
4 >> 1 = 2  ,4 >> 2 = 1(因为4=100，1向右移动1位得到010=2，向右移动2位得到001=1)
'''
def PowerSetsBinary(items):
    #生成n个元素items的所有组合
    n = len(items)
    #枚举2**n个可能的组合(n个元素的集合的子集有2**n个)
    for i in range(2**n):
        subset = []  #存放子集
        #j控制要选取的元素当前的下标对应的值
        for j in range(n):
            #i >> j 指i在二进制的形式将1向右移动j位
            #向右移动j次，判断结果除以2 余1 来得到是否要取当前的下标对应的值
            if(i >> j) % 2 == 1:
                subset.append(items[j])
        #呼叫subset
        yield subset
```

3.2.3 定义输出最大频繁项集的函数

```
'''
最大频繁项集的元素，不为别的频繁项集的子集
输入：频繁项目集L
输出：最大频繁项集Lmax
'''
def max_frequentSet(L):
    L = L[1:]  # 因为要考虑的是关联规则，频繁1项集生成的规则不是就关联规则，故不考虑
    Lmax = []  # 存放最大频繁项集
    # 对于每组频繁k-1项集，频繁k项集不考虑，因为其肯定为最大频繁项集
    for i in range(len(L) - 1):
        L_compared = []  # 存放要被拿来比较是否为子集的元素
        for j in range(i + 1, len(L)):
            for k in range(len(L[j])):
                L_compared.append(L[j][k])
        # 开始比较
        for j in range(len(L[i])):
            count = 0  # 统计L_compared中元素不包含L[i][j]的个数
            for k in range(len(L_compared)):
                if set(L[i][j]) < set(L_compared[k]):  # 如果是子集，直接结束在L_compared中的比较
                    break
                else:
                    count += 1  # 不是子集，则次数加一
            # 如果L_compared中元素不包含L[i][j]的个数和L_compared的个数相等，则可以认为L[i][j]不是L_compared的子集
            if count == len(L_compared):
                Lmax.append(L[i][j])

    # 将频繁k项集加入最大频繁项集
    for i in range(len(L[-1])):
        Lmax.append(L[-1][i])

    return Lmax
```

3.2.4 定义判断候选集的元素

```
'''
输入：一个候选k项集c，格式为[2, 3, 5]
      频繁k-1项集，格式为[[1,3],[2,3],[2,5],[3,5]]
输出：c是否从候选集中删除的布尔判断
      返回True删除
      返回False不删除
'''
def has_infrequent_subset(c,Lkn):
    #调用函数PowerSetsBinary(c)
    for subset in PowerSetsBinary(c):
        #只考虑k-1个元素的子集
        if len(subset) == len(Lkn[0]):
            if subset not in Lkn:
                return True
    return False
```

3.2.5 定义候选集产生的函数

```
'''
输入：频繁(k-1)项集Lkn，格式为[[1],[2],[3],[5]]
输出：候选k项集Ck，格式为
'''
def apriori_gen(Lkn):
    Ck = []  # 存放候选k项集
    # 遍历频繁k-1项集的每一个元素
    for i in range(len(Lkn)):
        p = Lkn[i]
        # 遍历频繁k-1项集在p之后的每个元素
        for j in range(i + 1, len(Lkn)):
            q = Lkn[j]

            count = 0  # 计算相等元素的个数
            for k in range(len(p) - 1):
                if p[k] == q[k]:
                    count += 1

            c = []  # 存放可以合并的元素
            if count == len(p) - 1:
                if p[count] < q[count]:  # 如果p最后一个元素<q最后一个元素
                    c = p.copy()  # c.append(p)
                    c.append(q[count])  # 将q的第k-1个(最后一个)元素加到p中并赋给c（c=p∞q）

            # 若c非空，则判断c是否为频繁项目集的候选元素
            if c != []:
                if has_infrequent_subset(c, Lkn) == False:
                    Ck.append(c)
    return Ck
```

3.2.6 定义发现频繁项集的函数

```
def Apriori(dataset, minsupport):
    # 计算最小支持数
    minsup_count = len(dataset) * minsupport

    # 候选1项集C1
    C1 = []
    C1_set = set()  # 因为候选集中元素不能重复，用集合是最方便的
    for i in range(len(dataset)):
        data = set(dataset[i])
        C1_set.update(data)  # 取两个集合的并集
    # 将集合中元素依次转换成列表形式，加入到大列表C1中
    # e.g.C1 = [[1], [2], [3], [4], [5]]
    for i in range(len(list(C1_set))):
        C1.append([list(C1_set)[i]])
    # print(C1)

    # 计算候选1项集的sup，调用自编函数C_sup(dataset,itemset_Ck)
    C1_sup = C_sup(dataset, C1)
    # print(C1_sup)

    # 频繁1项集L1
    L1 = []
    for item in C1_sup:
        if item[1] >= minsup_count:
            L1.append(item[0])
    # print(L1)
```

```
    L = []  # 存放频繁项目集
    L.append(L1)

    # 求接下来的频繁k项集
    Lkn = L1.copy()  # 频繁k-1项集
    while True:
        # 调用自编函数apriori_gen(Lkn)获得k个元素的候选集
        Ck = apriori_gen(Lkn)
        if Ck != []:
            # 计算候选k项集的sup，调用自编函数C_sup(dataset,itemset_Ck)
            Ck_sup = C_sup(dataset, Ck)
            # print(Ck_sup)

            # 得到所有支持数不小于minsup_count的频繁k项集Lkn
            Lkn = []  # 存放频繁k项集
            for item in Ck_sup:
                if item[1] >= minsup_count:
                    Lkn.append(item[0])

            L.append(Lkn)
        else:  # 候选k项集Ck为空则频繁k项集为空，故结束循环
            break

    print('频繁项集为：', L)

    if L != [[]]:
        # 调用自编函数max_frequentSet(L)求出最大频繁项集
        Lmax = max_frequentSet(L)
        return Lmax
    else:
        return []
```

3.2.7 定义生成关联规则的函数（深度优先搜索）

```
'''
输入：最大频繁项集Lmax，格式为[[1, 3],[2, 3, 5]]
      最小置信度minconf，格式为小数(e.g.0.8)
输出：关联规则
'''



def Rule_generate(dataset, Lmax, minconf):
    rules = []
    # 遍历所有最大频繁项目集
    for l_k in Lmax:
        # 递归调用genrules函数
        rule = genrules(dataset, l_k, l_k)
        rules.append(rule)

    finalRules = []
    # 去除重复项
    for rule in rules:
        finalRules.append(list(set(rule)))

    return finalRules
```

```
'''
输入：l_k频繁k项集，格式为[2,3,5]
      x_m频繁m项集
输出：关联规则
'''
def genrules(dataset, l_k, x_m):
    # 求出x_m的含有m-1项的子集
    m = len(x_m)

    rules = []
    # 调用自编函数PowerSetsBinary得到x_m的所有子集s
    for s in PowerSetsBinary(x_m):
        # 找出含有m-1项的子集
        if len(s) == m - 1:
            # 调用自编函数C_sup计算支持数
            l_k_sup = C_sup(dataset, [l_k])
            s_sup = C_sup(dataset, [s])
            # 计算置信度（因为 C_sup函数输出格式的原因，sup值在a[0][1]）
            conf = l_k_sup[0][1] / s_sup[0][1]
            # l_k - x_(m-1)得到的元素
            other = [l_k[i] for i in range(len(l_k)) if l_k[i] not in s]
            # 计算l_k的支持率
            support = l_k_sup[0][1] / len(dataset)
            print('l_k:{0}, x_m-1:{1}'.format(l_k, s))

            # 判断规则是否是强关联规则
            if conf >= minconf:  # 是强关联规则                                    round(a,2)是保留a小数点后两位有效数字
                rule = '规则 {0} => {1}, support={2}, confidence={3},({4})强关联规则'.format(s, other, round(support, 2),
                                                                                round(conf, 2), '是')
            else:  # 不是强关联规则
                rule = '规则 {0} => {1}, support={2}, confidence={3},({4})强关联规则'.format(s, other, round(support, 2),
                                                                                round(conf, 2), '不是')
            rules.append(rule)
```

```
            print(rule)
            print('\n')

            # 若x_(m-1)中元素个数大于1（因为小等于1个元素就无法生成关联规则）
            if m - 1 > 1:
                # 递归调用genrules函数
                rule = genrules(dataset, l_k, s)
                # 拉平了放入rules中
                for i in range(len(rule)):
                    rules.append(rule[i])

    return rules
```

（3）调用与结果

考虑到实际情况，顾客购买的商品不会出现很多次，所以在这里判断了一下出现在 0.1*9835=983.5 次交易中的商品个数。

```
data = pd.Series(Groceries[0])
#计算数据中不同元素以及其出现的次数
data_count = data.value_counts()
for k,v in dict(data_count).items():
    #找出所有support>=0.1的商品
    if v >= len(Groceries)*0.1:
        print(k)
```

然后设置了具体参数并生成关联规则

```
#设置minsupport = 0.03
minsupport =  0.03
Lmax = Apriori(dataset,minsupport)

#设置最小置信度
minconf = 0.25

rules = Rule_generate(dataset,Lmax,minconf)
```

为了可视化效果，这里定义将规则中数字替换为原本的商品名的函数

```
def Replace_goodsname(rules):
    finalRules = []
    # 调用re函数库中的complie函数设置一个查找的模式
    # \d为0~9的数字
    # +表示无论多少位
    pattern = re.compile(r'\d+')

    # 遍历规则rules准备替换
    for rule in rules:
        # 调用re函数库中的findall函数，可以按照自己定义的模式寻找所有符合条件的字符串，返回一个列表
        ruleNum = findall
        # 序列解包，按照','分割，rest是除了前面以外剩下的
        ruleList, *rest = rule[0].split(',')
        # 得到新的规则
        ruleNew = '规则 [%s] => [%s]' % (list(goodlist)[int(ruleNum[0])], list(goodlist)[int(ruleNum[1])])
        print(ruleNew)
        for other in rest:
            ruleNew = ruleNew + ',' + other
        finalRules.append(ruleNew)
    return finalRules
```

最终结果如下图所示，可以看出，购买 tropical fruit(热带水果)的顾客有 40%的可能会同时购买 whole milk。

```
['规则 [tropical fruit] => [whole milk], support=0.04, confidence=0.4,(是)强关联规则',
 '规则 [other vegetables] => [tropical fruit], support=0.04, confidence=0.19,(不是)强关联规则',
 '规则 [root vegetables] => [whole milk], support=0.05, confidence=0.45,(是)强关联规则',
 '规则 [root vegetables] => [other vegetables], support=0.05, confidence=0.44,(是)强关联规则',
 '规则 [whole milk] => [rolls/buns], support=0.06, confidence=0.22,(不是)强关联规则',
 '规则 [other vegetables] => [rolls/buns], support=0.04, confidence=0.22,(不是)强关联规则',
 '规则 [yogurt] => [rolls/buns], support=0.03, confidence=0.25,(不是)强关联规则',
 '规则 [rolls/buns] => [soda], support=0.04, confidence=0.21,(不是)强关联规则',
 '规则 [rolls/buns] => [sausage], support=0.03, confidence=0.17,(不是)强关联规则',
 '规则 [pip fruit] => [whole milk], support=0.03, confidence=0.4,(是)强关联规则',
 '规则 [other vegetables] => [whole milk], support=0.07, confidence=0.39,(是)强关联规则',
 '规则 [whole milk] => [yogurt], support=0.06, confidence=0.22,(不是)强关联规则',
 '规则 [whole milk] => [bottled water], support=0.03, confidence=0.13,(不是)强关联规则',
 '规则 [whipped/sour cream] => [whole milk], support=0.03, confidence=0.45,(是)强关联规则',
 '规则 [domestic eggs] => [whole milk], support=0.03, confidence=0.47,(是)强关联规则',
 '规则 [soda] => [whole milk], support=0.04, confidence=0.23,(不是)强关联规则',
 '规则 [whole milk] => [citrus fruit], support=0.03, confidence=0.12,(不是)强关联规则',
 '规则 [whole milk] => [pastry], support=0.03, confidence=0.13,(不是)强关联规则',
 '规则 [yogurt] => [other vegetables], support=0.04, confidence=0.31,(是)强关联规则',
```

四、实验结果分析

通过修改最小支持度与最小置信度，我们可以发现 MinSupport 越大，得到的最大频繁项目集就越少，生成的关联规则也就越少；反之生成的关联规则就越多

MinConfidence 越大，生成的强关联规则就越少；反之生成的强关联规则就越多

五、实验总结

通过本次实验，我对关联规则有了更深的理解，对 MinSupport、MinConfidence 等参数的作用也有了新的体会。