时空数据分析与挖掘

——关联规则相关实验与探究

学　院：遥感信息工程学院

班　级：2006 班（20F10）

姓　名：马文卓

学　号：2020302131249

老　师：田扬戈

时　间：2023 年 4 月 15 日

目录

一、实验目标与内容

1. 实验目标

根据学习的 Apriori 算法以及关联规则相关的知识对给定数据进行关联规则的挖掘，研究其中有哪些高频词、可以找到哪些关联规则、哪些是有效的。

2. 数据与要求

（1）数据

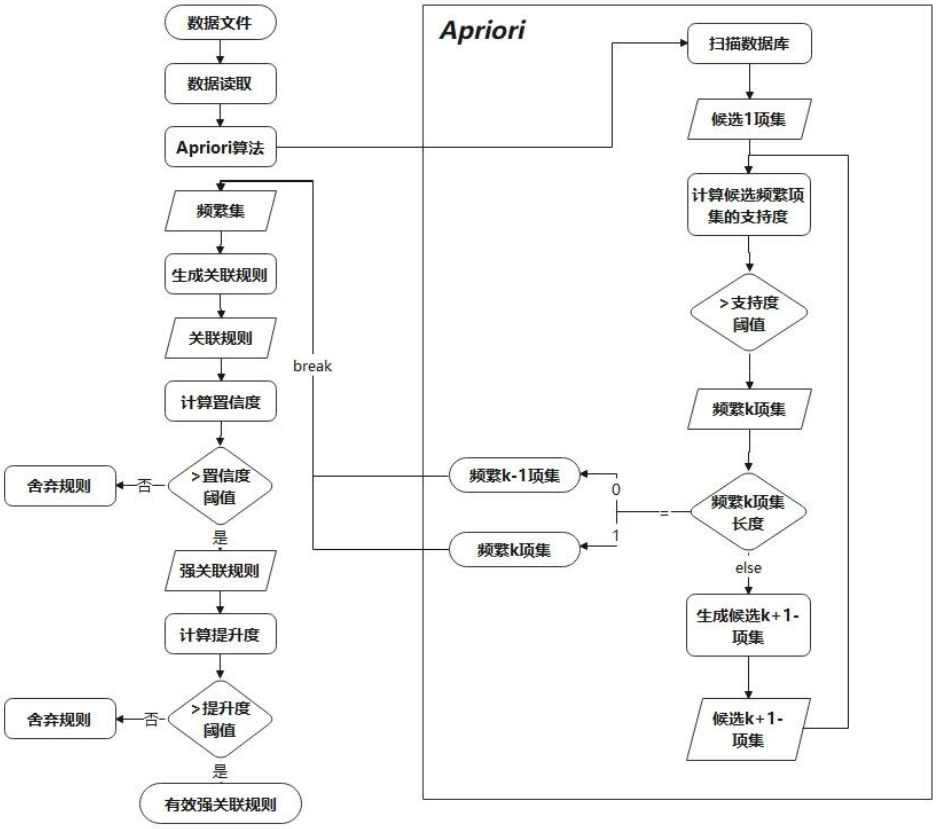
本次实验的数据为：SCI 数据库近 5 年所有有关 data mining 的论文的全部记录。数据一共 16071 篇论文记录，每个论文记录包含了 72 个字段。本次实验我们主要使用到其中的【Author Keywords】和【Keywords Plus】字段，对所有论文的关键词进行关联规则的挖掘。

（2）要求

①请大家根据 Author Keywords 和 Keywords Plus 这两个字段的信息，编程实现关联规则算法，回答数据挖掘研究中有哪些高频词，可以找到哪些关联规则；
②找到有用的关联规则指导研究（关联规则的评价）
③提供代码和简单报告
④报告里应有代码运行结果和分析

二、具体实施步骤与源码

1. 步骤流程图



2. 具体实施步骤

（1）数据读取与预处理

使用 pandas 库函数 read_excel 读取文件，值得注意的是这里需要对空缺值做处理，我将所有空缺值设置为空字符串，以免影响后面的操作。最后转化为 list 返回。

```
def readFile(path,constraint=None,no_values=''):
    '''读取数据文件,返回list'''
    files=os.listdir(path)
    data_sum=[]
    for filename in files:
        file=os.path.join(path,filename)
        data=pd.read_excel(io=file,usecols=constraint) # 读取文件
        data.fillna(no_values,inplace=True) # 填写空缺值
        data=data.values.tolist() # 转为list
        data_sum+=data
    return data_sum
```

由于两个字段都是文章的 keyword，所以我们将两个字段的词合在一起分析。值得注意的是，原始数据中两个关键词之间是用“； ”分隔的，千万不能少了

空格。再者，有些关键词还有大小写不同的版本，所以我们在预处理步骤里面全部转化为小写。

```
    '''数据预处理:将两个字段中的值全部分割,组成database'''
    database=[]
    for i in range(len(data)):
        database.append([]) # 添加Article Title字段
        a=data[i][0].split('; ') # 分割Author Keywords字段
        b=data[i][1].split('; ') # 分割Keywords Plus字段
        for j in range(len(a)): # 全部转小写
            a[j]=a[j].lower()
        for j in range(len(b)):
            b[j]=b[j].lower()
        if a!=['']:
            database[i]+=a
        if b!=['']:
            database[i]+=b
    return database
```

然后在主函数中调用上述函数进行数据读取和预处理即可获得最终的事务数据库。

```
data=readFile(data_floder,object_col) # 读取文件
database=dataPrepoccess(data) # 数据预处理
```

（2）Apriori 算法得到频繁集

使用 Apriori 算法来获取满足支持度阈值的频繁项集，其主要的步骤都体现在上一节的流程图中，本节主要结合代码进行解释。

①首先初始化 k=1，开始进入 Apriori 循环

```
def Apriori(database,support_threshold):
    '''
    Aprioris算法
    params:
        database:事务数据库
        support_threshold:支持度阈值[0,1]
    returns:
        返回一个包含频繁集和其支持度的字典
    '''
    k=1
    prekeywords={}
    keywords={}
    while True:
```

②对候选 k 项集，计算每个候选 k 项集的支持度，支持度公式如下。如果支持度大于支持度阈值，则选为 k 项频繁集，否则舍弃。

$$\text{support(X)} = \text{P(X)} = \frac{\text{number(X)}}{\text{number(AllSamples)}}$$

```
print(f'==================================频繁{k}-项集挖掘==================================')
# 计算每个keyword组合的出现次数
for paper in tqdm(database, desc=f"Processing k={k}"): # 遍历database中的所有paper
    for keyword in get_subsets(paper,k): # 遍历每个paper中的所有keyword组合
        if keyword not in list(keywords.keys()): # keyword组合没出现过
            keywords[keyword]=1
        else: # keyword组合出现过
            keywords[keyword]+=1
# 计算每个组合的支持度
for key in list(keywords.keys()):
    support=keywords[key]/len(database) # 支持度=出现频次/事务总数
    keywords[key]=support
    if support < support_threshold:
        keywords.pop(key) # 如果小于支持度阈值就删除相应的组合
```

③算法结束判断：如果 k 项频繁集的个数为 0，则返回 k-1 项频繁集作为结果；若 k 项频繁集个数为 1，则将该 k 项频繁集作为结果；若长度大于 1，则继续生成候选 k+1 项集。

```
# 算法结束判断
if len(keywords)==1:
    print(f'Apriori算法结束,在{k}-项集终止\n频繁集为: {list(keywords.keys())[0]}, 支持度为: {list(keywords.values())[0]:5f}')
    print(f'================================================================================')
    return keywords
if len(keywords)==0:
    print(f'Apriori算法结束,在{k-1}-项集终止\n频繁集为:\n{list(prekeywords.keys())}\n支持度为:\n{list(prekeywords.values())}')
    print(f'================================================================================')
    return prekeywords
k+=1 # 增加频繁集层数
```

④由 k 项频繁集进行组合生成 k+1 项候选集，然后进入新一轮的计算

```
k+=1 # 增加频繁集层数
prekeywords=keywords # 存储k-1层的结果
keywords=createKSet(list(keywords.keys()),k) # 创建下一层的候选集
```

```
def get_subsets(lst, k):
    """
    将一个列表转换为set,获取其所有长度为k的子集并转换为元组列表返回
    """
    s = set(lst)
    subsets = list(itertools.combinations(s, k))
    return [tuple(subset) for subset in subsets]

def createKSet(items,k):
    '''
    生成K项候选集
    params:
        items:k-1项集筛选过后的剩余项列表
        k:k项候选集看k>1
    '''
    keywords=[]
    for item in items:
        for i in list(item):
            if i not in keywords:
                keywords.append(i)
    Ck=get_subsets(keywords,k)
    return Ck
```

（3）由频繁项集生成关联规则

由 Apriori 算法得到频繁集 I 之后，我们求出频繁集 I 所有的子集 X 和其对应的 I-X，将每一对（X=>I-X）作为一个关联规则。

```
keywords=Apriori(database,threshold_support) # Apriori算法计算频繁项集
frequent_sets=list(keywords.keys()) # 找出的所有频繁项集
print('频繁项集: ',frequent_sets)
# 找关联规则
rules=[]
for frequent_set in frequent_sets: # 遍历每一个频繁项集
    frequent_set=set(frequent_set)
    for i in range(1, len(frequent_set)):
        for itemset in itertools.combinations(frequent_set, i): # 寻找每一个位数的子频繁项集
            itemset = set(itemset)
            complement = frequent_set - itemset
            for j in range(1, len(complement) + 1):
                for consequent in itertools.combinations(complement, j):
                    rule = (tuple(itemset), tuple(consequent))
                    rules.append(rule)
print('关联规则: ',rules)
```

（4）计算置信度筛选强关联规则

对于每一个关联规则计算其置信度，如果其置信度达到阈值要求，则认为这个规则是一个强关联规则。置信度计算公式如下：

$$\text{confidence}(Y => X) = P(X|Y) = \frac{P(XY)}{P(Y)}$$

```
# 计算每个关联规则的置信度
strong_rules=[] # 筛选后置信度达标的规则
confidences={} # 计算的强关联规则的置信度
for antecedent, consequent in rules: # 遍历所有关联规则
    antecedent = set(antecedent) # 起因
    consequent = set(consequent) # 后果
    support_antecedent = 0
    support_rule = 0
    for transaction in database: # 遍历所有事务
        if antecedent.issubset(set(transaction)): # 如果包含起因
            support_antecedent += 1
            if consequent.issubset(set(transaction)): # 包含起因的同时做了结果
                support_rule += 1
    confidence = support_rule / support_antecedent # 计算置信度
    print(f"规则 {antecedent} => {consequent} 的置信度为: {confidence:5f}")
    if confidence > threshold_confidence:  # 大于置信度阈值的才是有意义的规则
        strong_rules.append((antecedent,consequent))
        confidences[(list(antecedent)[0],list(consequent)[0])]=confidence
print('强关联规则: ',strong_rules)
```

（5）计算提升度筛选有效强关联规则

对于每一个强关联规则计算其提升度，如果其提升度达到阈值（=1）要求，则认为这个强关联规则是一个有效的强关联规则。提升度计算公式如下：

$$\text{lift}(Y => X) = \frac{P(X|Y)}{P(X)} = \frac{\text{confidence}(Y => X)}{P(X)}$$

```
# 计算每个强关联规则的提升度
useful_rules=[] # 有效的强关联规则
lifts={} # 有效的强关联规则的提升度
for antecedent,consequent in strong_rules: # 遍历所有强关联规则
    confidence=confidences[(list(antecedent)[0],list(consequent)[0])] # 获取置信度
    support_consequent=0
    for transaction in database: # 遍历所有事务
        if consequent.issubset(set(transaction)): # 如果包含结果
            support_consequent += 1
    support_consequent/=len(database) # 计算P（结果）
    lift=confidence/support_consequent # 计算提升度
    print(f"规则 {antecedent} => {consequent} 的提升度为: {lift:5f}")
    if lift > 1: # 是有效的强关联规则
        useful_rules.append((antecedent,consequent))
        lifts[(list(antecedent)[0],list(consequent)[0])]=lift
print('有用的强关联规则: ',useful_rules)
```

3. 代码优化

上述内容虽然已经完成挖掘关联规则的功能，但是实际运行起来效率非常低下，我们可以通过【剪枝】的方法来改进代码。主要有两种剪枝方法，具体如下：

（1）Prune-1

根据频繁项集的性质：任何频繁项集的非空子集都是频繁项集（它是基于这样的观察：如果项集 I 不是频繁的，那么 I 的超集必然不可能频繁）。根据这样的性质，我们在生成 k+1 项候选集时，可以剪掉那些 k 项子集不属于 k 项频繁集的 k+1 项候选集。

```
# Prune: 任何非频繁项集都不可能是频繁项集的子集
# 剪枝1: 减掉k-项候选集中的子集不在频繁k-1项集中的候选组合
pruned_candidates = []
for candidate in keywords: # 遍历生成的k-项集关键字组合
    subsets = get_subsets(candidate, k-1) # 取其k-1项子集
    if all(subset in prekeywords for subset in subsets): # 如果其所有的k-1项子集都被k-1项频繁集所包含
        pruned_candidates.append(candidate) # 则被纳入剪枝后的k-项候选集
keywords = dict(zip(pruned_candidates,[0]*len(pruned_candidates)))
```

（2）Prune-2

还是根据频繁项集的性质：任何频繁项集的非空子集都是频繁项集。我们可以思考，只要是没在频繁 k 项集中出现过的 keyword，必然不会出现在 k+1 项频繁集当中。因此在计算 k+1 项频繁集之前，我们可以删除事务数据库当中所有没在频繁 k 项集中出现过的 keyword，这样大大缩减了需要扫描的 keyword，提高了效率。

```
# 剪枝2: 由于k-1项频繁集中所有的1项子集都是频繁集，所以删掉每个事务中的非频繁1项集
preset=[]
for key in list(prekeywords.keys()):
    for a in key:
        if a not in preset:
            preset.append(a)
for i in range(len(database)):
    j=0
    while j<len(database[i]):
        if database[i][j] not in preset:
            database[i].remove(database[i][j])
        else:
            j+=1
```

三、实验分析

1. 结果分析

结果如下图所示（详见 result 目录下 associate_rules_prune_s0.025_c0.85.log）。该结果的支持度阈值为 0.025，置信度阈值为 0.85.最终得到两条有效的强关联规则：feature extraction=>data mining 和 task analysis=>data mining。这两项规则的置信度都非常高（>0.9），提升度也是远大于 1，所以说这两条规则是有效且紧密联系的。

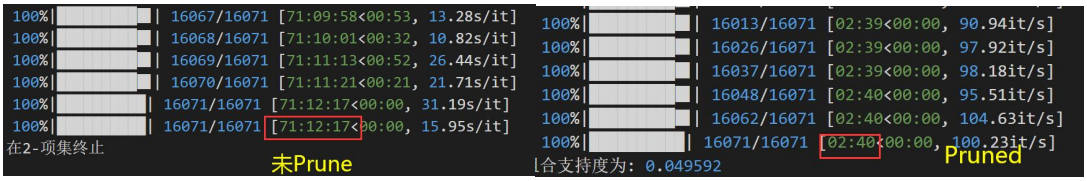
```
==========
频繁项集:  [('machine learning', 'data mining'), ('feature extraction', 'data mining'), ('data mining', 'classification'), ('task analysis', 'data mining')]
关联规则:  [(('machine learning',), ('data mining',)), (('data mining',), ('machine learning',)), (('feature extraction',), ('data mining',)), (('data mining',
规则 {'machine learning'} => {'data mining'} 的置信度为: 0.535183
规则 {'data mining'} => {'machine learning'} 的置信度为: 0.121692
规则 {'feature extraction'} => {'data mining'} 的置信度为: 0.936441
规则 {'data mining'} => {'feature extraction'} 的置信度为: 0.204000
规则 {'data mining'} => {'classification'} 的置信度为: 0.098462
规则 {'classification'} => {'data mining'} 的置信度为: 0.462428
规则 {'task analysis'} => {'data mining'} 的置信度为: 0.903509
规则 {'data mining'} => {'task analysis'} 的置信度为: 0.079231
强关联规则:  [({'feature extraction'}, {'data mining'}), ({'task analysis'}, {'data mining'})]
规则 {'feature extraction'} => {'data mining'} 的提升度为: 2.315314
规则 {'task analysis'} => {'data mining'} 的提升度为: 2.233891
有用的强关联规则:  [({'feature extraction'}, {'data mining'}), ({'task analysis'}, {'data mining'})]
+-------+--------------------+-------------+--------------------+--------------------+
| index |    antecedence     | consequence |     confidence     |        lift        |
+-------+--------------------+-------------+--------------------+--------------------+
|   1   | feature extraction | data mining | 0.9364406779661016 | 2.3153135593220338 |
|   2   |   task analysis    | data mining | 0.9035087719298246 | 2.2338906882591094 |
+-------+--------------------+-------------+--------------------+--------------------+
```

对于这两条规则的共同之处在于，他们的 consequence 全部是 data mining，也就是说无论是 feature extraction 还是 task analysis，都会有很大可能导致 data

mining 的使用。其实，在现在海量数据的时代，对于特征提取和任务分析，数据挖掘是一种不可获取的信息获取手段、解决方法。所以从这五年来大数据时代的发展和数据挖掘手段的飞速进步来看，这两条关联规则也是合理的。

2. 效率分析

对于剪枝的作用，我做了如下消融实验：没有剪枝策略的方法和有剪枝策略的方法对相同的数据集进行关联规则的挖掘，最后在得到相同结果的条件下，对比其耗费的时间成本如下。可以看到没有剪枝的花费了 71 小时，远超剪枝过后的 3 分钟左右的时间成本。可以得出结论，我们的两个剪枝策略是有效的。



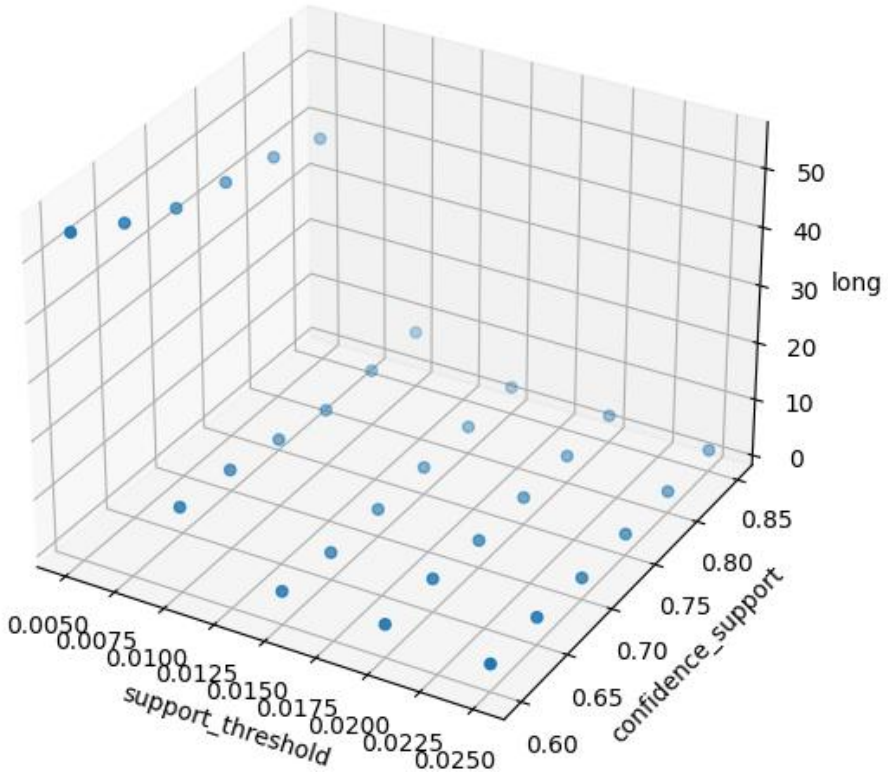
3. 趋势分析

为了得到数据的整体情况，我做了这样的实验：让支持度阈值分别取 0.005 到 0.025（每次增加 0.005），置信度阈值分别取 0.6 到 0.85（每次增加 0.05），分别执行关联规则的挖掘，然后汇总每次挖掘出的有效的强关联规则数目成表格，并且绘图可视化。（代码见 code 目录下的 analysis.py）

```
record=[]
index=0
support_threhold=0.005
while support_threhold<0.03:
    confidence_threshlod=0.6
    while confidence_threshlod<0.9:
        index+=1
        print(f'==============================【support_threhold:{support_threhold},confidence_threshold:{confidence_threshlod}】==============================')
        useful_rules,tb=ar.get_associate_rules('/home/ubuntu/mwz/Spatio-temporal_data_mining_and_analysis/data',['Author Keywords','Keywords Plus'],support_threhold,confidence_thre
        record.append([index,support_threhold,confidence_threshlod,len(useful_rules)])
        confidence_threshlod+=0.05
    support_threhold+=0.005
tb=pt.PrettyTable()
tb.field_names=['index','support_threshold','confidence_threshold','number of rules']
for item in record:
    tb.add_row(item)
print(f'==================================【All In All】==================================')
```

```
# 绘图
x = [r[1] for r in record]
y = [r[2] for r in record]
z = [r[3] for r in record]
# 创建三维图形对象
fig = plt.figure()
ax = Axes3D(fig)
# 绘制三维图形
ax.scatter(x, y, z)
# 设置坐标轴标签
ax.set_xlabel('support_threshold')
ax.set_ylabel('confidence_support')
ax.set_zlabel('long')
# 保存图形
plt.savefig('Spatio-temporal_data_mining_and_analysis/work1/analysis.png')
```

最终结果如下，（详见 result 下的 analysis.log 和 analysis.png）。我们可以看到是，显而易见，随着支持度阈值的升高和置信度阈值的升高，筛选出的关联规则越来越少。因此，对于关联规则的挖掘，合适的支持度阈值和置信度阈值是最后结果是否为好的关联规则的重要影响因素。



==========【All In All】==========

index	support_threshold	confidence_threshold	number of rules
1	0.005	0.6	54
2	0.005	0.65	49
3	0.005	0.7	45
4	0.005	0.75	43
5	0.005	0.8	41
6	0.005	0.85	38
7	0.01	0.6	13
8	0.01	0.65	12
9	0.01	0.7	10
10	0.01	0.75	8
11	0.01	0.8	8
12	0.01	0.85	8
13	0.015	0.6	4
14	0.015	0.65	3
15	0.015	0.7	3
16	0.015	0.75	3
17	0.015	0.8	3
18	0.015	0.85	3
19	0.02	0.6	4
20	0.02	0.65	4
21	0.02	0.70	3
22	0.02	0.75	3
23	0.02	0.8	3
24	0.02	0.85	3
25	0.025	0.6	3
26	0.025	0.65	3
27	0.025	0.7	2
28	0.025	0.75	2
29	0.025	0.8	2
30	0.025	0.85	2