数据挖掘作业二报告

关联规则挖掘

姓名：赵赫　　学号：**2120171103**

一、 问题描述

本次作业中，我选择将对 San Francisco Building Permits 数据集进行关联规则挖掘。

本次关联规则挖掘任务包含以下四个子任务：

- 对数据集进行处理，转换成适合关联规则挖掘的形式。
- 找出频繁项集。
- 导出关联规则，计算其支持度和置信度。
- 对规则进行评价，可使用 Lift，也可以使用教材中所提及的其他指标。

二、 数据描述

- **数据集 2：San Francisco Building Permits**
  该数据集共包含 43 个属性，198900 条数据记录。

三、 关联规则挖掘过程

3.1 将数据集转换成适合关联规则挖掘的形式

考虑到问题的复杂度和运算速度，且数据集中存在大量的数据缺失情况，因此我选择了 Building Permits 数据集中的 11 个标称属性进行关联规则挖掘。

标称属性名称存放在 “./data/关联挖掘属性列名.txt” 目录下，11 个属性列名分别为：

Permit Type
Current Status
Existing Construction Type
Proposed Construction Type
Neighborhoods Analysis Boundaries
Zipcode
Existing Construction Type
Existing Use

Proposed Use

Fire Only Permit

Site Permit

由于我们需要对多个属性进行关联规则挖掘，此问题属于多维关联分析问题，不同于“啤酒-尿布”的加购物车问题，单维规则分析问题仅针对于单个属性，因此我们需要先将来自于不同属性的值转化为可生成频繁项集的形式。首先我将每一个属性名：属性值的组合用一个二元组的形式表示，既（属性名，属性值），作为一个单项。并使用 python 中的 frozenset 类型表示项集。与 set 类型不同的是，frozenset 类型一经创建便不可修改，因此可以将可迭代对象转化为字典的键进行处理。对数据集进行处理的代码如下：

```
# 遍历全部属性
columns = []
for feature_name in feature_list:
    print("Dealing with feature: {}".format(feature_name))
    feature_col = self.get_feature_values(feature_name, table_name)
    columns.append(feature_col)
rows = list(zip(*columns))
# 将数据转为数据字典存储
dataset = []
feature_names = rows[0]
for data_line in rows[1:]:
    data_set = []
    for i , value in enumerate(data_line):
        if not value:
            data_set.append((feature_names[i], 'NA'))
        else:
            data_set.append((feature_names[i], value))
    dataset.append(data_set)
```

对数据集进行处理的代码位于“./src/data.py”。

3.2 找出频繁项集

在经过处理的数据集的基础上，采用 Apriori 算法构建频繁项集。在此任务中，频繁项集（frequent item sets）是指经常出现在一起的属性项的集合，而一个项集的支持度（support）定义为数据集中包含该项集的记录所占的比例。首先，规定最小支持度（min-support）为 0.01（最小支持度较小是因为数据集中有几个属性取值较多，若支持度设定过大会忽略这些属性），最小置信度（min-confidence）为 0.5：

```
def __init__(self, min_support = 0.01, min_confidence = 0.5):
    self.min_support = min_support          # 最小支持度
    self.min_confidence = min_confidence    #  最小置信度
```

Apriori 算法首先会生成所有单个（属性名，属性值）的项集列表，然后扫描全部数据集来查看哪些项集满足最小支持度要求，其中不满足最小支持度的集合会被去掉，接着对剩下的集合进行组合（组合中，要求属性名相同的项仅有一个）以生成包含两个项的项集，接着重新扫描交易记录，去掉不满足最小支持度的项集，该过程重复进行直到所有项集都被滤掉。

Apriori 算法的实现位于“./src/associasion.py”，程序代码如下：

- create_C1()函数用于生成初始的单个项项集集合：

```
def create_C1(self, dataset):
    # 扫描dataset，构建全部可能的单元素候选项集合(list)
    # 每个单元素候选项：（属性名，属性取值）
    C1 = []
    for data in dataset:
        for item in data:
            if [item] not in C1:
                C1.append([item])
    C1.sort()
    return [frozenset(item) for item in C1]
```

- Scan_D()函数用于扫描项集集合，并过滤掉小于最小支持度的项集：

```
def scan_D(self, dataset, Ck):
    # 过滤函数
    # 根据待选项集Ck的情况，判断数据集D中Ck元素的出现频率
    # 过滤掉低于最小支持度的项集
    Ck_count = dict()
    for data in dataset:
        for cand in Ck:
            if cand.issubset(data):
                if cand not in Ck_count:
                    Ck_count[cand] = 1
                else:
                    Ck_count[cand] += 1

    num_items = float(len(dataset))
    return_list = []
    support_data = dict()
    # 过滤非频繁项集
    for key in Ck_count:
        support  = Ck_count[key] / num_items
        if support >= self.min_support:
            return_list.insert(0, key)
        support_data[key] = support
    return return_list, support_data
```

- Apriori_gen()函数用于非重复地合并两个项集：

```
def apriori_gen(self, Lk, k):
    # 当待选项集不是单个元素时，  如k>=2的情况下，合并元素时容易出现重复
    # 因此针对包含k个元素的频繁项集，对比每个频繁项集第k-2位是否一致
    return_list = []
    len_Lk = len(Lk)

    for i in range(len_Lk):
        for j in range(i+1, len_Lk):
            # 第k-2个项相同时，将两个集合合并
            L1 = list(Lk[i])[:k-2]
            L2 = list(Lk[j])[:k-2]
            L1.sort()
            L2.sort()
            if L1 == L2:
                return_list.append(Lk[i] | Lk[j])
    return return_list
```

- Apriori 主函数：

```
def apriori(self, dataset):
    """
    Apriori算法实现
    :param dataset: 数据集，类型为一个list，
           list中每个元素是一个dict，key为属性名，value为对应属性的取值
    :return: 生成频繁项集
    """
    C1 = self.create_C1(dataset)
    dataset = [set(data) for data in dataset]
    L1, support_data = self.scan_D(dataset, C1)
    L = [L1]
    k = 2
    while len(L[k-2]) > 0:
        Ck = self.apriori_gen(L[k-2], k)
        Lk, support_k = self.scan_D(dataset, Ck)
        print(Lk)
        support_data.update(support_k)
        L.append(Lk)
        k += 1
    return L, support_data
```

3.3 导出关联规则并计算支持度、置信度、Lift 指标

基于使用 Apriori 算法产生的频繁项集，产生强关联规则的算法过程为：首先从一个频繁项集开始，创建一个规则列表，其中规则右部只包含一个元素，然后对这些规则计算是否满足最小置信度要求。接下来合并所有的剩余规则列表来创建一个新的规则列表，其中规则右部包含两个元素。最后，对于产生的每个规则，我们分别计算其支持度(support)、置信度(confidence)以及提升度(Lift)指标。计算公式如下：

- 支持度（support）：

$$Sup(X) = \frac{Sum(X)}{N}$$

- 置信度（confidence）：

$$Conf(X \Rightarrow Y) = \frac{Sup(X \cup Y)}{Sup(X)}$$

- 提升度（Lift）：

用来判断规则 X ⇒ Y 中的 X 和 Y 是否独立，如果独立，那么这个规则是无效的。如果该值等于 1 ,说明两个条件没有任何关联。如果小于 1 ,说明 X 与 Y 是负相关的关系，意味着一个出现可能导致另外一个不出现。大于 1 才表示具有正相关的关系。一般在数据挖掘中当提升度大于 3 时,我们才承认挖掘出的关联规则是有价值的。

$$Lift(X \Rightarrow Y) = \frac{Sup(X \cup Y)}{Sup(X) \times Sup(Y)} = \frac{Conf(X \cup Y)}{Sup(Y)}$$

产生强关联规则的算法同样位于“./src/associasion.py”的 Associasion 类中，程序代码如下：

- Rules_from_conseq()函数，用于递归地产生规则右部的结果项集：

```
def rules_from_conseq(self, freq_set, H, support_data, big_rules_list):
    # H->出现在规则右部的元素列表
    m = len(H[0])
    if len(freq_set) > (m+1):
        Hmp1 = self.apriori_gen(H, m+1)
        Hmp1 = self.cal_conf(freq_set, Hmp1, support_data, big_rules_list)
        if len(Hmp1) > 1:
            self.rules_from_conseq(freq_set, Hmp1, support_data, big_rules_list)
```

- Generate_rules()函数，用于产生强关联规则：

```
def generate_rules(self, L, support_data):
    """
    产生强关联规则算法实现
    基于Apriori算法，首先从一个频繁项集开始，接着创建一个规则列表，
    其中规则右部只包含一个元素，然后对这些规则进行测试。
    接下来合并所有的剩余规则列表来创建一个新的规则列表，
    其中规则右部包含两个元素。这种方法称作分级法。
    :param L: 频繁项集
    :param support_data: 频繁项集对应的支持度
    :return: 强关联规则列表
    """
    big_rules_list = []
    for i in range(1, len(L)):
        for freq_set in L[i]:
            H1 = [frozenset([item]) for item in freq_set]
            # 只获取有两个或更多元素的集合
            if i > 1:
                self.rules_from_conseq(freq_set, H1, support_data, big_rules_list)
            else:
                self.cal_conf(freq_set, H1, support_data, big_rules_list)
    return big_rules_list
```

- Cal_conf()函数，用于评价生成的规则，并计算支持度、置信度、lift 指标：

```
def cal_conf(self, freq_set, H, support_data, big_rules_list):
    # 评估生成的规则
    prunedH = []
    for conseq in H:
        sup = support_data[freq_set]
        conf = sup / support_data[freq_set - conseq]
        lift = conf / support_data[freq_set - conseq]
        if conf >= self.min_confidence:
            big_rules_list.append((freq_set-conseq, conseq, sup, conf, lift))
            prunedH.append(conseq)
    return prunedH
```

3.4 挖掘结果及分析

频繁项集结果位于“./results/freq_set.json”文件下，文件每行描述了一个频繁项集，且所有项集按照支持度由大到小进行排列，结果文件格式如下（以一行为例）：

```
{
"sup": 0.48662644544997485,
"set": [["Permit_Type", "8"], ["Existing_Construction_Type", "5"],
["Fire_Only_Permit", "NA"], ["Site_Permit", "NA"]]
}
```

关联规则结果位于“./results/rules.json”文件下，文件每行描述了一个关联规则，且所有关联规则按照置信度由大到小排列，结果文件如下（以一行为例）：


```
{
"X_set": [["Fire_Only_Permit", "NA"], ["Proposed_Use", "office"],
"Y_set": [["Neighborhoods_Analysis_Boundaries", "Financial District/South
Beach"], ["Site_Permit", "NA"]],
"conf": 1.0,
"sup": 0.01147812971342383,
 ["Zipcode", "94105"], ["Current_Status", "complete"]],
"lift": 87.1222076215506}
```


对最终挖掘得到的关联规则进行分析我们可以得知，“Exiting Use”属性与“Proposed Use”属性关联度极高，且“Exiting Use –> Proposed Use”规则的置信度极高，这说明大部分建筑的现用途与预期用途是一致的。