说明文档

Apriori算法

Apriori算法：

1）扫描整个数据集，得到所有出现过的不重复的数据，作为候选频繁1项集。k=1，频繁0项集为空集。

2）挖掘频繁k项集

a) 扫描数据计算候选频繁k项集的支持度

b) 去除候选频繁k项集中支持度低于阈值的数据集,得到频繁k项集。如果得到的频繁k项集为空，则直接返回频繁k-1项集的集合作为算法结果，算法结束。如果得到的频繁k项集只有一项，则直接返回频繁k项集的集合作为算法结果，算法结束。

c) 基于频繁k项集，连接生成候选频繁k+1项集。

3） 令k=k+1，转入步骤2。

产生大量候选项集，每轮迭代都要根据上一级频繁项集来产生新的候选集，因此在数据集很大，数据种类很多的时候，算法效率低。

```
def generate_candidates(frequent_candidates_k1):
    """
    根据频繁项集 L(k-1) 生成候选项集 C(k)。

    参数:
        frequent_itemsets_k1 (list): 频繁项集 L(k-1) 的列表，每个项集用元组表示。

    返回:
        list: 候选项集 C(k) 的列表，每个项集用元组表示。
    """
def calculate_support(dataset1, candidate):
    """
    计算数据集中候选项集的支持度。“出现个数”
    candidate 是 元组
    """
def Apriori(data_set, min_sup_):
    """
    频繁项集挖掘的Apriori算法。
    """
    # 初始化频繁项集列表
    frequent_item_sets = []  # 这是结果
```

```
22
23      # k = 1
24      # 包含所有数据集中所有的项，不重复
25      unique_items = set(item for transaction in data_set for item in
    transaction)
26
27      # 生成长度为1的频繁项集
28      # 每个候选项都是一个包含单个项的元
29      k1_candidates = [item for item in unique_items]
30      # 对于每个候选项集 candidate，如果其支持度不低于最小支持度阈值 min_sup_
31      for candidate in k1_candidates:
32          # 计算候选项集 candidate 在数据集中的支持度
33          support = calculate_support(data_set, (candidate,))
34          # 如果支持度满足最小支持度阈值，则将该候选项集添加到频繁项集列表中
35          if support >= min_sup_:
36              # 将候选项集转换为列表形式，并添加到频繁项集列表中
37              frequent_item_sets.append([candidate])
38
39      # 生成长度> 1的频繁项集
40      k = 2
41      frequent_candidates = sorted(frequent_item_sets)
42      frequent_candidates = [tuple(sublist) for sublist in frequent_candidates]
43      while True:
44          # 生成新候选集
45          new_candidates = generate_candidates(frequent_candidates)
46          # 计算每个候选项集的支持度，并筛选出频繁项集
47          frequent_candidates = []
48          for candidate in new_candidates:
49              # 计算候选项集的支持度
50              support = calculate_support(data_set, candidate)
51              # 如果支持度满足最小支持度阈值，则将其添加到频繁项集列表中
52              if support >= min_sup_:
53                  frequent_candidates.append(candidate)
54
55          if not frequent_candidates:
56              break
57          frequent_item_sets.extend(frequent_candidates)
58          k += 1
59
60      sorted_output = sorted(frequent_item_sets, key=lambda x: (len(x), x))
61      return sorted_output
```

Apriori算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['DonaldF.Towsley'], ['FathiE.AbdElSamie'], ['FlorentinSmarandache'], ['FranciscoHerrera'], ['GeyongMin'], ['H.VincentPoor'], ['HsiaoHwaChen'], ['JeanFran'],

['JieLi0002'], ['JieZhang'], ['JingChen'], ['LaurenceT.Yang'], ['LichengJiao'], ['LiuqingYang0001'], ['MohsenGuizani'], ['NanCheng'], ['PengShi0001'], ['PengWang'], ['QianWang'], ['RobertSchober'], ['SajalK.Das0001'], ['Sebasti'], ['ShigekiSugano'], ['VictorC.M.Leung'], ['VinceD.Calhoun'], ['XiaojiangDu'], ['XuelongLi0001'], ['XueminShen'], ['YanLiu'], ['YingLi'], ['lvarez'], ('MohsenGuizani', 'XiaojiangDu'), ('NanCheng', 'XueminShen')]

```
D:\anaconda\python.exe "D:\program\data_mining\frequent pattern mining\AprioriTest.py"
A:
Apriori算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['DonaldF.Towsley'],
```

FPGrowth算法

代码详情请见.py文件

```python
class FPNode:
    def __init__(self, item, count, parent):
        self.item = item
        self.count = count
        self.parent = parent
        self.children = {}
        self.next = None
def build_FP_tree(dataset, min_sup):
    """
    构造FP树
    :param dataset: 数据集
    :param min_sup:
    :return:
    """
def insert_tree(items, node, header_table):
    """
    执行 插入事务的一个项到树里
    :param items:  事务，一条记录
    :param node: 事务插到node下
    :param header_table:  项头表
    :return:
    """
def mine_FP_tree(header_table, min_sup, L):
...
def FPGrowth(dataset, min_sup):
    # Count item occurrences in dataset
    item_counts = {}
    for transaction in dataset:
        for item in transaction:
            item_counts[item] = item_counts.get(item, 0) + 1

```

```
    # 过滤出不符合要求的项，按频率逆序排列，得到L
    L = {item: count for item, count in item_counts.items()}
    L = sorted(L.items(), key=lambda x: (-x[1], x[0]))

    root, header_table = build_FP_tree(dataset, min_sup)
    frequent_patterns = mine_FP_tree(header_table, min_sup, L)

    frequent_patterns = sorted(frequent_patterns, key=lambda x: (len(x),x ))
    # 将元组转换为列表，并放入一个列表中
    frequent_patterns = [[item for item in pattern] for pattern in
frequent_patterns]
    return frequent_patterns
```

B:

FPGrowth算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['DonaldF.Towsley'], ['FathiE.AbdElSamie'], ['FlorentinSmarandache'], ['FranciscoHerrera'], ['GeyongMin'], ['H.VincentPoor'], ['HsiaoHwaChen'], ['JeanFran'], ['JieLi0002'], ['JieZhang'], ['JingChen'], ['LaurenceT.Yang'], ['LichengJiao'], ['LiuqingYang0001'], ['MohsenGuizani'], ['NanCheng'], ['PengShi0001'], ['PengWang'], ['QianWang'], ['RobertSchober'], ['SajalK.Das0001'], ['Sebasti'], ['ShigekiSugano'], ['VictorC.M.Leung'], ['VinceD.Calhoun'], ['XiaojiangDu'], ['XuelongLi0001'], ['XueminShen'], ['YanLiu'], ['YingLi'], ['lvarez'], ['MohsenGuizani', 'XiaojiangDu'], ['NanCheng', 'XueminShen']]

```
D:\anaconda\python.exe "D:\program\data_mining\frequent pattern mining\FPMining.py"
A:
Apriori算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['DonaldF.Towsley'], ['FathiE.AbdElSamie'],
B:
FPGrowth算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['DonaldF.Towsley'], ['FathiE.AbdElSamie']

进程已结束，退出代码为 0
```

Eclat算法

过程如下：

1. 通过扫描一次数据集，把水平格式的数据转换成垂直格式；
2. 项集的支持度计数简单地等于项集的TID集的长度；
3. 从k=1开始，可以根据先验性质，使用频繁k项集来构造候选（k+1）项集；
4. 通过取频繁k项集的TID集的交，计算对应的（k+1）项集的TID集。
5. 重复该过程，每次k增加1，直到不能再找到频繁项集或候选项集。

Eclat算法产生候选项集的理论基础是：频繁K-项集可以通过或运算生成候选的K+1-项集，频繁K-项集中的项是按照字典序排列，并且进行或运算的频繁K-项集的前K-1个项是完全相同的。

Eclat算法除了在产生候选（k+1）项集时利用先验性质外，另一个优点是不需要扫描数据库来确定（k+1）项集的支持度（k>=1），这是因为每个k项集的TID集携带了计算支持度的完整信息。然而，TID集可能很长，需要大量的内存空间，长集合的交运算还需要大量的计算时间。

Eclat整体上只扫描了一遍数据库，但是在频繁项较多时的交集运算会比较花费时间

```
# ECLAT
def eclat(prefix, items, min_support, freq_items):
    '''
    递归方式遍历数据集，找出频繁项集
    :param prefix:当前的前缀
    :param items:数据集中的项集
    :param min_support:
    :param freq_items:频繁项集的字典
    :return:
    '''
    while items:
        # 初始遍历单个的元素是否是频繁
        key, item = items.pop()
        key_support = len(item)
        if key_support >= min_support:
            # print frozenset(sorted(prefix+[key]))
            freq_items[frozenset(sorted(prefix+[key]))] = key_support
            suffix = []  # 存储当前长度的项集
            for other_key, other_item in items:
                new_item = item & other_item  # 求和其他集合求交集
                if len(new_item) >= min_support:
                    suffix.append((other_key, new_item))
            eclat(prefix+[key], sorted(suffix, key=lambda item: len(item[1]),
reverse=True),
             min_support, freq_items)
    return freq_items
def ECLAT(data_set, min_sup):
    """
    Eclat方法:将数据集进行倒排，即将每个项映射到包含该项的事务编号的集合中。然后调用
eclat 函数来
    找出频繁项集
    :param data_set:
    :param min_sup:
    :return:
    """
    # 将数据倒排
    data = {}
    trans_num = 0
    for trans in data_set:
        trans_num += 1
```

```
        for item in trans:
            if item not in data:
                data[item] = set()
            data[item].add(trans_num)
    freq_items = {}
    freq_items = eclat([], sorted(data.items(), key=lambda item: len(item[1]),
     reverse=True), min_sup, freq_items)
    freq_itemsets = [list(item) for item in freq_items.keys()]
    freq_itemsets =  sorted(freq_itemsets, key=lambda x: (len(x), x))
    return freq_itemsets
```

C:

ECLAT算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['DonaldF.Towsley'], ['FathiE.AbdElSamie'], ['FlorentinSmarandache'], ['FranciscoHerrera'], ['GeyongMin'], ['H.VincentPoor'], ['HsiaoHwaChen'], ['JeanFran'], ['JieLi0002'], ['JieZhang'], ['JingChen'], ['LaurenceT.Yang'], ['LichengJiao'], ['LiuqingYang0001'], ['MohsenGuizani'], ['NanCheng'], ['PengShi0001'], ['PengWang'], ['QianWang'], ['RobertSchober'], ['SajalK.Das0001'], ['Sebasti'], ['ShigekiSugano'], ['VictorC.M.Leung'], ['VinceD.Calhoun'], ['XiaojiangDu'], ['XuelongLi0001'], ['XueminShen'], ['YanLiu'], ['YingLi'], ['lvarez'], ['XiaojiangDu', 'MohsenGuizani'], ['XueminShen', 'NanCheng']]

```
B:
FPGrowth算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['Don
C:
ECLAT算法挖掘的频繁项集结果： [['BernardDeBaets'], ['C.L.PhilipChen'], ['ChinChenChang0001'], ['Donald
```

不同算法之间的性能差异：

Apriori算法在DBLPdata-10k.txt数据集下运行时间远远大于FPGrowth和ECLAT算法处理数据集产生结果的时间

可能是因为迭代产生新候选集在数据集很大（k1_candidates>30000+）的情况下用的时间较多