1. Apriori 与FP-Growth算法流程图

1.1． 基本概念

- 支持度：P(A ∩ B)，既有A又有B的概率
- 置信度：P(B|A)，在A发生的事件中同时发生B的概率 p(AB)/P(A)
- 频繁k项集：如果事件A中包含k个元素，那么称这个事件A为k项集事件A满足最小支持度阈值的事件称为频繁k项集
- 强规则：同时满足最小支持度阈值和最小置信度阈值的规则称为强规则

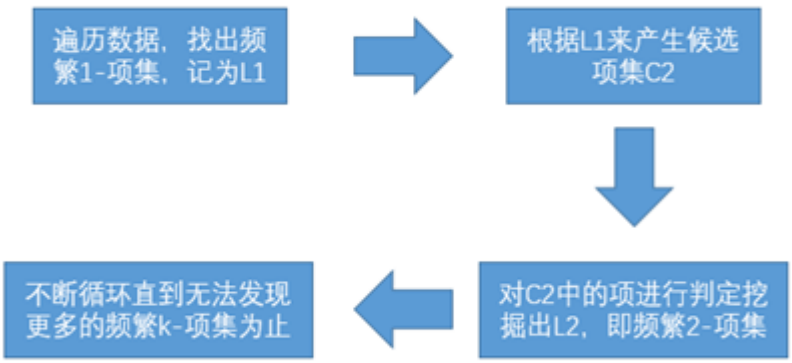
1.1． Apriori 算法过程

1.2.1 算法描述

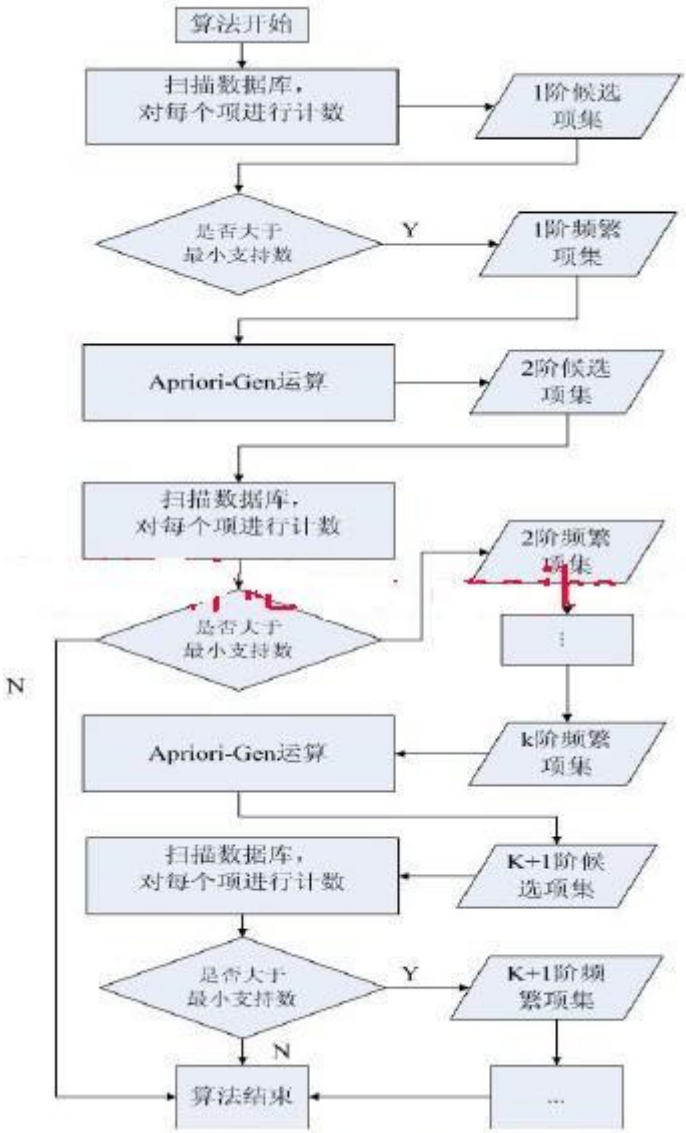
第一步通过迭代，检索出事务数据库中的所有频繁项集，即支持度不低于用户设定的阈值的项集；

第二步利用频繁项集构造出满足用户最小信任度的规则。具体做法就是：首先找出频繁1-项集，记为L1；然后利用L1来产生候选项集C2，对C2中的项进行判定挖掘出L2，即频繁2-项集；不断如此循环下去直到无法发现更多的频繁k-项集为止。每挖掘一层Lk就需要扫描整个数据库一遍。

1.2.2 逻辑流程图



1.2.3 算法流程图

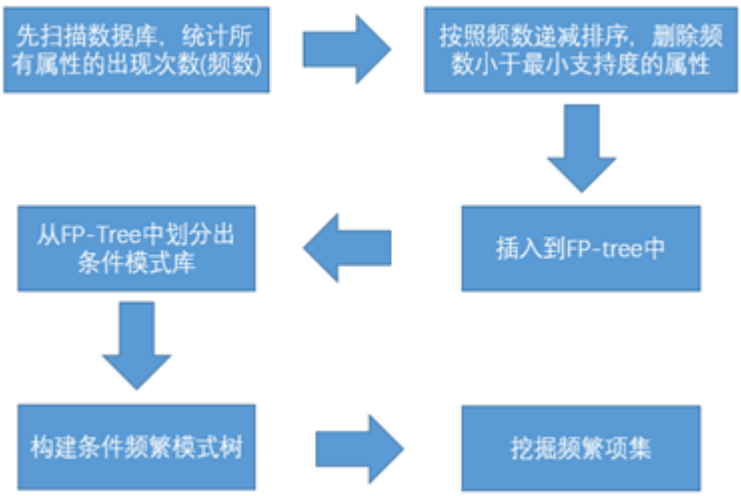


1.2. FP-Growth 算法过程

1.3.1 算法步骤

- 先扫描数据库，统计所有属性的出现次数(频数)，然后按照频数递减排序，删除频数小于min_suppt（最小支持度）的属性。
- 对每一条数据记录，重新排序（从大到小），并删除小于min_suppt的商品。并插入到FP-tree中。
- 从FP-Tree中划分出条件模式库。
- 构建条件频繁模式树。
- 挖掘频繁项集。

1.3.2 逻辑流程图



1.3.3 算法流程图

2. Apriori与FP-Growth算法效率对比

2.1． Apriori 算法实现

代码详见附录1

2.2． FP-Growth算法实现

代码详见附录2

2.3． 算法效率对比

- Apriori算法

```
运行时间:  0.005019664764404297
```

- FP-Growth算法

```
运行时间:  0.0030100345611572266
```

分析：从运行时间来看，FP-Growth算法的效率要比Apriori算法效率高。但由于我们所选数据集较小，遍历数据不多，所以运行速度相差效果不明显，随着数据的庞大，FP-Growth算法由于只遍历两次数据，其性能也才的得以凸显。

```
# 测试数据集
dataset = [
    ['牛奶', '鸡蛋', '面包','薯片'],
    [ '鸡蛋','爆米花', '薯片','啤酒'],
    ['牛奶','啤酒','面包'],
    ['牛奶','鸡蛋','面包','爆米花','啤酒','薯片'],
    ['鸡蛋','面包','薯片'],
    ['鸡蛋','面包','啤酒', ],
    ['牛奶','面包', '薯片'],
    ['牛奶','鸡蛋','面包','黄油','薯片'],
    ['牛奶','鸡蛋','黄油','薯片']
]
```

3. FP-Growth算法后加入关联规则

3.1 FP-Growth算法最终结果

FP-Growth算法生成的是频繁项集，而不是关联规则。 如果要进一步生成关联规则，首先需要根据 FP-Growth算法生成的频繁项集生成候选关联规则。然后通过预先设置的置信度阈值对关联规则进行进步过滤，得到强关联规则。

3.2 FP-Growth算法增加关联规则方法

3.2.1 关联规则生成过程

- 筛选出满足最小支持度的频繁项集
- 选出满足最小支持度的频繁项集

3.2.2 关联规则生成实现

```
    # 已筛选的频繁项集置信度计算
    for i in range(0, len(L)):
        for freq_set in L[i]:
            for sub_set in sub_set_list:
                if sub_set.issubset(freq_set):
                    conf = support_data[freq_set] / support_data[freq_set - sub_set]
                    big_rule = (freq_set - sub_set, sub_set, conf)
                    if conf >= min_conf and big_rule not in big_rule_list:
                        # print freq_set-sub_set, " => ", sub_set, "conf: ", conf
                        big_rule_list.append(big_rule)
            sub_set_list.append(freq_set)
```

4. 总结

- 与枚举所有的项集相比， Apriori算法利用频繁项集的单性，大大减少了候选集的数量，从而提高了关联规则挖掘的效率。然而这种方法仍然可能构造大量无用的候选项集。
- P-Growth算法首先通过遍历两次原始数据集，将原始数据集表示成一个压缩的树形数据结构 FP-tree。后续的频繁项集挖掘直接利用FP-tree, 而不再依赖于原始数据集。
- FP-tree通常比原始数据集更小, 因此与需要多次遍历原始数据集的 Apriori算法相比, FP-Growth往往能够获得更高的性能。同时，与 Apriori进行相比, FP-Growth没有生成无用的候选项集，运行相对快一个数量级。

5. 附录

5.1 附录1：Apriori算法实现Python代码

```
import time

def load_data_set():
    #载入数据
    # 数据集
    data_set = [
        ['牛奶', '鸡蛋', '面包', '薯片'],
        ['鸡蛋', '爆米花', '薯片', '啤酒'],
        ['牛奶', '啤酒', '面包'],
        ['牛奶', '鸡蛋', '面包', '爆米花', '啤酒', '薯片'],
        ['鸡蛋', '面包', '薯片'],
        ['鸡蛋', '面包', '啤酒', ],
        ['牛奶', '面包', '薯片'],
        ['牛奶', '鸡蛋', '面包', '黄油', '薯片'],
        ['牛奶', '鸡蛋', '黄油', '薯片']
    ]
    return data_set


def create_C1(data_set):
    #频繁一项集创建
    C1 = set()
    for t in data_set:
        for item in t:
            item_set = frozenset([item])
            C1.add(item_set)
    return C1
```

```
def is_apriori(Ck_item, Lksub1):
    for item in Ck_item:
        sub_Ck = Ck_item - frozenset([item])
        if sub_Ck not in Lksub1:
            return False
    return True


def create_Ck(Lksub1, k):
    #频繁k项集创建
    Ck = set()
    len_Lksub1 = len(Lksub1)
    list_Lksub1 = list(Lksub1)
    for i in range(len_Lksub1):
        for j in range(1, len_Lksub1):
            l1 = list(list_Lksub1[i])
            l2 = list(list_Lksub1[j])
            l1.sort()
            l2.sort()
            if l1[0:k-2] == l2[0:k-2]:
                Ck_item = list_Lksub1[i] | list_Lksub1[j]
                # pruning
                if is_apriori(Ck_item, Lksub1):
                    Ck.add(Ck_item)
    return Ck


def generate_Lk_by_Ck(data_set, Ck, min_support, support_data):
    Lk = set()
    item_count = {}
    for t in data_set:
        for item in Ck:
            if item.issubset(t):
                if item not in item_count:
                    item_count[item] = 1
                else:
                    item_count[item] += 1
    t_num = float(len(data_set))
    for item in item_count:
        if (item_count[item] / t_num) >= min_support:
            Lk.add(item)
            support_data[item] = item_count[item] / t_num
    return Lk


def generate_L(data_set, k, min_support):
    support_data = {}
    C1 = create_C1(data_set)
    L1 = generate_Lk_by_Ck(data_set, C1, min_support, support_data)
    Lksub1 = L1.copy()
    L = []
    L.append(Lksub1)
```

```
    for i in range(2, k+1):
        Ci = create_Ck(Lksub1, i)
        Li = generate_Lk_by_Ck(data_set, Ci, min_support, support_data)
        Lksub1 = Li.copy()
        L.append(Lksub1)
    return L, support_data


def generate_big_rules(L, support_data, min_conf):
    big_rule_list = []
    sub_set_list = []
    for i in range(0, len(L)):
        for freq_set in L[i]:
            for sub_set in sub_set_list:
                if sub_set.issubset(freq_set):
                    conf = support_data[freq_set] / support_data[freq_set - sub_set]
                    big_rule = (freq_set - sub_set, sub_set, conf)
                    if conf >= min_conf and big_rule not in big_rule_list:
                        # print freq_set-sub_set, " => ", sub_set, "conf: ", conf
                        big_rule_list.append(big_rule)
            sub_set_list.append(freq_set)
    return big_rule_list

#主函数
if __name__ == "__main__":
    start = time.time()
    data_set = load_data_set()
    L, support_data = generate_L(data_set, k=3, min_support=0.2)
    big_rules_list = generate_big_rules(L, support_data, min_conf=0.7)
    for Lk in L:
        print ("="*50)
        print ("frequent " + str(len(list(Lk)[0])) + "-itemsets (频繁项集) \t\tsupport (支持
度) ")
        print ("="*50)
        for freq_set in Lk:
            print (freq_set, support_data[freq_set])
    print
    print ("强关联")
    for item in big_rules_list:
        print (item[0], "=>", item[1], "conf(置信度): ", item[2])

    end=time.time()
    print('运行时间: ',str(end-start))
```

5.2 附录2：FP-Growth算法实现Python代码

```
from collections import defaultdict, namedtuple
import time

def find_frequent_itemsets(transactions, minimum_support, include_support=False):

    items = defaultdict(lambda: 0)
```

```
    for transaction in transactions:
        for item in transaction:
            items[item] += 1

    # 频繁1项集筛选
    items = dict((item, support) for item, support in items.items()
        if support >= minimum_support)

    def clean_transaction(transaction):
        transaction = filter(lambda v: v in items, transaction)
        transaction_list = list(transaction)   # 为了防止变量在其他部分调用，这里引入临时变量
transaction_list
        transaction_list.sort(key=lambda v: items[v], reverse=True)
        return transaction_list

    #FPtree构建
    master = FPTree()
    for transaction in map(clean_transaction, transactions):
        master.add(transaction)

    def find_with_suffix(tree, suffix):
        for item, nodes in tree.items():
            support = sum(n.count for n in nodes)
            if support >= minimum_support and item not in suffix:

                found_set = [item] + suffix
                yield (found_set, support) if include_support else found_set

                cond_tree = conditional_tree_from_paths(tree.prefix_paths(item))
                for s in find_with_suffix(cond_tree, found_set):
                    yield s

    for itemset in find_with_suffix(master, []):
        yield itemset

class FPTree(object):

    Route = namedtuple('Route', 'head tail')

    def __init__(self):
        # 初始化根节点和分支
        self._root = FPNode(self, None, None)

        self._routes = {}

    @property
    def root(self):
        # 创建根节点
        return self._root

    def add(self, transaction):
        # 添加节点
        point = self._root
```

```
        for item in transaction:
            next_point = point.search(item)
            if next_point:
                #当前已存在节点
                next_point.increment()
            else:
                # 创建节点
                next_point = FPNode(self, item)
                point.add(next_point)

                # 更新链路
                self._update_route(next_point)

            point = next_point

    def _update_route(self, point):
        assert self is point.tree

        try:
            route = self._routes[point.item]
            route[1].neighbor = point # route[1] is the tail
            self._routes[point.item] = self.Route(route[0], point)
        except KeyError:
            # 开启新节点
            self._routes[point.item] = self.Route(point, point)

    def items(self):

        for item in self._routes.keys():
            yield (item, self.nodes(item))

    def nodes(self, item):

        try:
            node = self._routes[item][0]
        except KeyError:
            return

        while node:
            yield node
            node = node.neighbor

    def prefix_paths(self, item):

        def collect_path(node):
            path = []
            while node and not node.root:
                path.append(node)
                node = node.parent
            path.reverse()
            return path
```

```
        return (collect_path(node) for node in self.nodes(item))

    def inspect(self):
        print('Tree:')
        self.root.inspect(1)

        print
        print('Routes:')
        for item, nodes in self.items():
            print('  %r' % item)
            for node in nodes:
                print('    %r' % node)

def conditional_tree_from_paths(paths):
    tree = FPTree()
    condition_item = None
    items = set()

    for path in paths:
        if condition_item is None:
            condition_item = path[-1].item

        point = tree.root
        for node in path:
            next_point = point.search(node.item)
            if not next_point:
                # Add a new node to the tree.
                items.add(node.item)
                count = node.count if node.item == condition_item else 0
                next_point = FPNode(tree, node.item, count)
                point.add(next_point)
                tree._update_route(next_point)
            point = next_point

    assert condition_item is not None

    # 计算节点
    for path in tree.prefix_paths(condition_item):
        count = path[-1].count
        for node in reversed(path[:-1]):
            node._count += count

    return tree

class FPNode(object):

    def __init__(self, tree, item, count=1):
        self._tree = tree
        self._item = item
        self._count = count
        self._parent = None
        self._children = {}
        self._neighbor = None
```

```
    def add(self, child):

        if not isinstance(child, FPNode):
            raise TypeError("Can only add other FPNodes as children")

        if not child.item in self._children:
            self._children[child.item] = child
            child.parent = self

    def search(self, item):
        try:
            return self._children[item]
        except KeyError:
            return None

    def __contains__(self, item):
        return item in self._children

    @property
    def tree(self):
        return self._tree

    @property
    def item(self):
        return self._item

    @property
    def count(self):
        return self._count

    def increment(self):
        if self._count is None:
            raise ValueError("Root nodes have no associated count.")
        self._count += 1

    @property
    def root(self):
        return self._item is None and self._count is None

    @property
    def leaf(self):
        return len(self._children) == 0

    @property
    def parent(self):
        return self._parent

    @parent.setter
    def parent(self, value):
        if value is not None and not isinstance(value, FPNode):
            raise TypeError("A node must have an FPNode as a parent.")
        if value and value.tree is not self.tree:
```

```
            raise ValueError("Cannot have a parent from another tree.")
        self._parent = value

    @property
    def neighbor(self):
        return self._neighbor

    @neighbor.setter
    def neighbor(self, value):
        if value is not None and not isinstance(value, FPNode):
            raise TypeError("A node must have an FPNode as a neighbor.")
        if value and value.tree is not self.tree:
            raise ValueError("Cannot have a neighbor from another tree.")
        self._neighbor = value

    @property
    def children(self):
        return tuple(self._children.itervalues())

    def inspect(self, depth=0):
        print(('  ' * depth) + repr(self))
        for child in self.children:
            child.inspect(depth + 1)

    def __repr__(self):
        if self.root:
            return "<%s (root)>" % type(self).__name__
        return "<%s %r (%r)>" % (type(self).__name__, self.item, self.count)



# 数据集
dataset = [
    ['牛奶', '鸡蛋', '面包','薯片'],
    [ '鸡蛋','爆米花', '薯片','啤酒'],
    ['牛奶','啤酒','面包'],
    ['牛奶','鸡蛋','面包','爆米花','啤酒','薯片'],
    ['鸡蛋','面包','薯片'],
    ['鸡蛋','面包','啤酒', ],
    ['牛奶','面包', '薯片'],
    ['牛奶','鸡蛋','面包','黄油','薯片'],
    ['牛奶','鸡蛋','黄油','薯片']
]

#主函数
if __name__ == '__main__':

    '''
    调用find_frequent_itemsets()生成频繁项
    @:param minimum_support表示设置的最小支持度，即若支持度大于等于inimum_support，保存此频繁项，否则删除
    @:param include_support表示返回结果是否包含支持度，若include_support=True，返回结果中包含itemset和support，否则只返回itemset
```

```
    '''
    start = time.time()

    frequent_itemsets = find_frequent_itemsets(dataset, minimum_support=1,
include_support=True)
    print(type(frequent_itemsets))   # print type

    result = []
    for itemset, support in frequent_itemsets:    # 将generator结果存入list
        result.append((itemset, support))

    result = sorted(result, key=lambda i: i[0])   # 排序后输出
    for itemset, support in result:
        print(str(itemset) + ' ' + str(support))

    end = time.time()
    print('运行时间: ', str(end - start))
```