FP-growth

作者: 刘伟杰 日期:2015-12-13
参考: [1] 《机器学习实战》 Peter

1. 理论

1. 概述:

   用项集的集合构造出FP树，再从FP树中挖掘出频繁项集。该方法会比Apriori更高效

2. 步骤:

   - 第一步: 创建FP树
   - 第二步: 从FP树中挖掘出频繁项集

3. 示例:

   参考Peter写出的实例，具体代码见附录，调用过程如下:

```
# 获取数据
data_set =  [
    ['r', 'z', 'h', 'j', 'p'],
    ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
    ['z'],
    ['r', 'x', 'n', 'o', 's'],
    ['y', 'r', 'x', 'z', 'q', 't', 'p'],
    ['y', 'z', 'x', 'e', 'q', 's', 't', 'm']
]
fiq = fp_growth(data_set, 0.5)  # 0.5为最小支持度
print fiq
```

附录

FP-growth算法python实现:

```
# -*- coding:utf-8 -*-
'''
    using FP-growth to find frequent items set
    @author: Liu Weijie
'''
from numpy import *

def load_data():
    simp_data = [
        ['r', 'z', 'h', 'j', 'p'],
        ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
        ['z'],
        ['r', 'x', 'n', 'o', 's'],
        ['y', 'r', 'x', 'z', 'q', 't', 'p'],
        ['y', 'z', 'x', 'e', 'q', 's', 't', 'm']
    ]
    return simp_data


def create_init_data(data_set):
    ret_dict = {}
    for one_set in data_set:
        ret_dict[frozenset(one_set)] = 1
    return ret_dict


class TreeNode:
    def __init__(self, name_value, num_occur, parent_node):
        self.name = name_value
        self.count = num_occur
        self.node_link = None
        self.parent = parent_node
        self.children = {}

    def inc(self, num_occur):
        self.count += num_occur

    def disp(self, ind=1):
        print '->' * ind, self.name, ' ', self.count
        for child in self.children.values():
            child.disp(ind + 1)


def create_fp_tree(data_set, min_sup=1):
    # 筛选出支持率大于min_sup的header_table
    header_table = {}
    for one_set in data_set.keys():
        for item in one_set:
            header_table[item] = header_table.get(item, 0) + data_set[one_set]
    for item in header_table.keys():
        if header_table[item] < min_sup:
```

```
            del(header_table[item])

    freq_item_set = set(header_table.keys())
    if len(freq_item_set) == 0:
        return None, None

    for k in header_table:
        header_table[k] = [header_table[k], None]

    ret_tree = TreeNode('Null_Set', 1, None)
    # 用一个个项集trans_set来更新fp－tree
    for trans_set, count in data_set.items():
        # 求该项集各元素在header_table中的频数
        local_D = {}
        for item in trans_set:
            if item in freq_item_set:
                local_D[item] = header_table[item][0]
        if len(local_D) > 0:
            order_item_list = [v[0] for v in sorted(local_D.items(), key=lambda p: p[1
            # 更新tree
            update_tree(order_item_list, ret_tree, header_table, count)
    return ret_tree, header_table


def update_tree(items, in_tree, header_table, count):
    # 创建叶子节点
    if items[0] in in_tree.children:
        in_tree.children[items[0]].inc(count)
    else:
        in_tree.children[items[0]] = TreeNode(items[0], count, in_tree)
        # 更新header_table
        if header_table[items[0]][1] is None:
            header_table[items[0]][1] = in_tree.children[items[0]]
        else:
            update_header(header_table[items[0]][1], in_tree.children[items[0]])
    # 递归创建节点
    if len(items) > 1:
        update_tree(items[1::], in_tree.children[items[0]], header_table, count)


def update_header(node_test, target_node):
    # 用最末尾的那个节点指向新节点
    while node_test.node_link is not None:
        node_test = node_test.node_link
    node_test.node_link = target_node


# 上朔整课树，找到该结点的前缀路径，prefix_path should be a list []
def ascend_tree(leaf_node, prefix_path):
    if leaf_node.parent is not None:
        prefix_path.append(leaf_node.name)
        ascend_tree(leaf_node.parent, prefix_path)
```

```
# 返回该项集的条件模式基
def find_prefix_path(base_pat, tree_node):
    cond_pats = {}
    while tree_node is not None:
        prefix_path = []
        ascend_tree(tree_node, prefix_path)
        if len(prefix_path) > 1:
            cond_pats[frozenset(prefix_path[1:])] = tree_node.count
        tree_node = tree_node.node_link
    return cond_pats


def mine_tree(inTree, header_table, min_sup, prefix, freq_item_list):
    big_l = [v[0] for v in sorted(header_table.items(), key=lambda p: p[1])]  # (sort
    for base_pat in big_l:  # start from bottom of header table
        new_freq_set = prefix.copy()
        new_freq_set.add(base_pat)
        # print 'finalFrequent Item: ',new_freq_set    # append to set
        freq_item_list.append(new_freq_set)
        cond_patt_bases = find_prefix_path(base_pat, header_table[base_pat][1])
        # print 'cond_patt_bases :',base_pat, cond_patt_bases
        # 2. construct cond FP-tree from cond. pattern base
        my_cond_tree, my_head = create_fp_tree(cond_patt_bases, min_sup)
        # print 'head from conditional tree: ', my_head
        if my_head is not None:  # 3. mine cond. FP-tree
            # print 'conditional tree for: ',new_freq_set
            # my_cond_tree.disp(1)
            mine_tree(my_cond_tree, my_head, min_sup, new_freq_set, freq_item_list)


def fp_growth(data_set, min_sup=0.6):
    init_data_set = create_init_data(data_set)  # 把数据转成后续支持的格式
    min_sup = min_sup * len(data_set)  # 设置最小支持度
    fp_tree, header_table = create_fp_tree(init_data_set, min_sup)  # 构建FP树
    my_freq_list = []
    mine_tree(fp_tree, header_table, min_sup, set([]), my_freq_list)  # 从FP树中挖掘出频
    return my_freq_list

# def example():
#     # 获取数据
#     data_set = load_data()
#     init_data_set = create_init_data(data_set)
#     min_sup = 3  # 设置最小支持度
#     fp_tree, header_table = create_fp_tree(init_data_set, min_sup)  # 构建FP树
#     my_freq_list = []
#     mine_tree(fp_tree, header_table, min_sup, set([]), my_freq_list)  # 从FP树中挖掘出
#     print my_freq_list


if __name__ == '__main__':
```

```
data_set = load_data()
fiq = fp_growth(data_set, 0.5)
print fiq
```