《数据仓库与数据挖掘》实验报告

姓名：	朱志儒	学号：	SA20225085	日期：	2020/11/23
上机题目：	Aprior 算法和 FP 算法的实现				

操作环境：
OS：Window 10
CPU：AMD Ryzen 5 3600X 6-Core Processor 4.25GHz
GPU：GeForce RTX 2070 super

一、基础知识：
Apriori 算法：

Input: DS 数据集; $support_{min}$: 最小支持度

Output: $L = \{L_1, L_2, \dots, L_k\}$: 频繁项集的集合

$k = 1, L \leftarrow \emptyset$, scan DS, 得到$L_k$

while $L_k \neq \emptyset$ *do*

$L = L \cup \{L_k\}$

generate C_{k+1} /* C_{k+1}是$k+1$项集，不一定是频繁的，等待“频繁”判定的候选集*/

compute support(C_{k+1}) /*support(C_{k+1})表示C_{k+1}中每个项集的支持度

$L_{k+1} = \{x | x \in C_{k+1}\ and\ support(x) > support_{min}\}$

$k \leftarrow k + 1$

return L

其中：

generate C_{k+1}:

Input: L_k每个项集有序的 k-项集集合

Output: C_{k+1}候选“频繁”的(k+1)-项集集合

满足前 k-1 项相同的 k-项集合并，产生(k+1)-项集

return support(C_{k+1})

以及

compute support(C_{k+1}):

Input: DS 数据集, C_{k+1}候选“频繁”的$(k+1)$ − 项集集合

Output: C_{k+1}中每个项集的支持度

扫描一遍 DS 的元组，检查它包含的哪些C_{k+1}中的项集，将被包含的项集的支持度+1

return support(C_{k+1})

FP-growth 算法：
输入：DS 数据集，$support_{min}$ 最小支持度
输出：频繁模式的完全集
方法：

1. 构造 FP 树：
   （a） 扫描数据集 DS 一次，收集频繁项的集合 F 和它的支持度计数，对 F 按支持度计数降序排序，结果为频繁项列表 L
   （b） 创建 FP 树的根节点，以“null”标记它，对于 DS 中每个事务 Trans，执行：
   选择 Trans 中的频繁项，并按 L 中的次序排序，设 Trans 排序后的频繁项列表为[p|P]，其中 p 是第一个元素，而 P 是剩余元素的列表。调用 insert_tree([p|P], T)。
   该过程执行情况如下：
   如果 T 有子女 N 使得 N.item-name = p.item-name，则 N 的计数增加 1；否则，创建一个新结点 N，将其计数设置为 1，链接到它的父结点 T，并通过结点链结构将其链接到具有相同 item-name 的结点。如果 P 非空，则递归调用 insert_tree(P, N)。
2. FP 树的挖掘通过调用 FP_growth(FP_tree, null)实现：
   Procedure FP_growth(Tree, α)
   (1) If Tree 包含单个路径 P then
   (2) for 路径 P 中结点的每个组合（记作 β）
   (3) 产生模式 β∪α，其支持度计数 support_count 等于 β 中结点的最小支持度
   (4) else for Tree 的头表中的每个 a_i
   (5) 产生一个模式 β= a_i∪α，其支持度计数 support_count = a_i.support_count
   (6) 构造 β 的条件模式基，然后构造 β 的条件 FP 树 $Tree_\beta$
   (7) if $Tree_\beta \neq \emptyset$ then
   (8) 调用 FP_growth($Tree_\beta$, β)

二、实验过程：

根据 Apriori 算法和 FP-growth 算法的伪代码编写程序，读入 data.txt 数据集，找出频繁项集，改变 $support_{min}$，查看频繁项集的变化，比较两个算法的性能。

三、结果分析：

当最小支持度为 0.4 时

Apriori 算法：

```
频繁项集: ['3']
频繁项集: ['2']
频繁项集: ['7']
频繁项集: ['8']
频繁项集: ['6']
频繁项集: ['1']
频繁项集: ['5']
频繁项集: ['4']
频繁项集数量: 8
用时: 0.20149850845336914 s
```

FP-growth 算法：

```
频繁项集: {'7'}
频繁项集: {'8'}
频繁项集: {'6'}
频繁项集: {'4'}
频繁项集: {'5'}
频繁项集: {'3'}
频繁项集: {'1'}
频繁项集: {'2'}
频繁项集数量: 8
用时: 0.001993417739868164 s
```

最小支持度为 0.3 时

Apriori 算法：

```
频繁项集: ['1']
频繁项集: ['8']
频繁项集: ['14']
频繁项集: ['11']
频繁项集: ['5']
频繁项集: ['12']
频繁项集: ['6']
频繁项集: ['7']
频繁项集: ['2']
频繁项集: ['13']
频繁项集: ['3']
频繁项集: ['10']
频繁项集: ['4']
频繁项集: ['9']
频繁项集数量: 14
用时: 0.3015632629394531 s
```

FP-growth 算法：

```
频繁项集: {'14'}
频繁项集: {'13'}
频繁项集: {'12'}
频繁项集: {'11'}
频繁项集: {'10'}
频繁项集: {'9'}
频繁项集: {'7'}
频繁项集: {'8'}
频繁项集: {'6'}
频繁项集: {'4'}
频繁项集: {'5'}
频繁项集: {'3'}
频繁项集: {'1'}
频繁项集: {'2'}
频繁项集数量: 14
用时: 0.0219419002532959 s
```

以上结果显示 FP-growth 算法的时间复杂度比 Apriori 算法的时间复杂度要低

附录

算法源代码（C/C++/JAVA 描述）：

FP-growth 算法：

```
1. import time
2.
3. def load_data():
4.     data = []
5.     file = open("data.txt")
```

```
    lines = file.readlines()
    for line in lines:
        data.append([x for x in line.split()])
    data_dict = {}
    for line in data:
        data_dict[frozenset(line)] = 1
    return data_dict

class Node:
    def __init__(self, value, count, parent):
        self.value = value
        self.count = count
        self.link = None
        self.parent = parent
        self.children = {}

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

    def __lt__(self, other):
        return self.count < other.count

def create_tree(data_dict, min_support):
    head_table = {}
    for line in data_dict:
        for item in line:
            head_table[item] = head_table.get(item, 0) + da
ta_dict[line]
    for k in list(head_table.keys()):
        if head_table[k] / 2000.0 < min_support:
            head_table.pop(k)
    item_set = set(head_table.keys())
    if not item_set:
        return None, None
    for key in head_table.keys():
        head_table[key] = [head_table[key], None]
    tree = Node("null set", 1, None)
    for tran_set, count in data_dict.items():
        local = {}
        for item in tran_set:
            if item in item_set:
                local[item] = head_table[item][0]
        if len(local) > 0:
            order_items = [x[0] for x in sorted(local.items
```

```
    (), key=lambda p: p[1], reverse=True)]
49.             update_tree(order_items, tree, head_table, coun
    t)
50.     return tree, head_table
51.
52. def update_tree(items, tree, head_table, count):
53.     if items[0] in tree.children:
54.         tree.children[items[0]].inc(count)
55.     else:
56.         tree.children[items[0]] = Node(items[0], count, tre
    e)
57.         if head_table[items[0]][1] == None:
58.             head_table[items[0]][1] = tree.children[items[0
    ]]
59.         else:
60.             update_head(head_table[items[0]][1], tree.child
    ren[items[0]])
61.     if len(items) > 1:
62.         update_tree(items[1::], tree.children[items[0]], he
    ad_table, count)
63.
64. def update_head(test, target):
65.     while (test.link != None):
66.         test = test.link
67.     test.link = target
68.
69. def ascend_tree(leaf, prefix_path):
70.     if leaf.parent != None:
71.         prefix_path.append(leaf.value)
72.         ascend_tree(leaf.parent, prefix_path)
73.
74. def find_prefix_path(base, tree_node):
75.     cond_pats = {}
76.     while tree_node != None:
77.         prefix_path = []
78.         ascend_tree(tree_node, prefix_path)
79.         if len(prefix_path) > 1:
80.             cond_pats[frozenset(prefix_path[1:])] = tree_no
    de.count
81.         tree_node = tree_node.link
82.     return cond_pats
83.
84. def mine_tree(tree, head_table, min_support, pre_fix, freq_
    item_list):
```

```
    big = [k for k, v in sorted(head_table.items(), key=lambda p: p[1][0])]
    for base in big:
        new_freq_set = pre_fix.copy()
        new_freq_set.add(base)
        freq_item_list.append(new_freq_set)
        cond_pat_base = find_prefix_path(base, head_table[base][1])
        cond_tree, head = create_tree(cond_pat_base, min_support)
        if head != None:
            mine_tree(cond_tree, head, min_support, new_freq_set, freq_item_list)

if __name__ == "__main__":
    min_support = 0.03
    data_dict = load_data()
    fp_tree, head_table = create_tree(data_dict, min_support)
    freq_list = []
     start_time = time.time()
    mine_tree(fp_tree, head_table, min_support, set([]), freq_list)
    end_time = time.time()
    for item in freq_list:
        print("频繁项集:", item)
    print("用时:", end_time - start_time, 's')
```

Apriori 算法：

```
import copy
import time

def load_data():
    data = []
    file = open("data.txt")
    lines = file.readlines()
    for line in lines:
        data.append([x for x in line.split()])
    return data

def satisfy_apriori(Ck_item, Lk_1):
    for item in Ck_item:
        sub_Ck = Ck_item - frozenset([item])
```

```
        if sub_Ck not in Lk_1:
            return False
    return True

def generate_Ck(Lk_1, k):
    Ck = set()
    Lk_1_len = len(Lk_1)
    Lk_1_list = list(Lk_1)
    for i in range(Lk_1_len):
        for j in range(1, Lk_1_len):
            li = list(Lk_1_list[i])
            lj = list(Lk_1_list[j])
            li.sort()
            lj.sort()
            if li[0:k-2] == lj[0:k-2]:
                Ck_item = Lk_1_list[i] | Lk_1_list[j]
                if satisfy_apriori(Ck_item, Lk_1):
                    Ck.add(Ck_item)
    return Ck

def generate_Lk_from_Ck(data, Ck, min_support, support_dict
):
    Lk = set()
    item_count_dict = {}
    for line in data:
        for item in Ck:
            if item.issubset(line):
                if item not in item_count_dict.keys():
                    item_count_dict[item] = 1
                else:
                    item_count_dict[item] += 1
    data_len = float(len(data))
    for item in item_count_dict.keys():
        if item_count_dict[item] / data_len >= min_support:

            Lk.add(item)
            support_dict[item] = item_count_dict[item] / da
ta_len
    return Lk

def generate_L_and_support_dict(data, min_support):
    support_dict = {}
    C1 = set()
    for line in data:
```

```
56.         for item in line:
57.             C1.add(frozenset([item]))
58.     Li = generate_Lk_from_Ck(data, C1, min_support, support
    _dict)
59.     Lk_1 = copy.deepcopy(Li)
60.     L = []
61.     k = 2
62.     while Li:
63.         L.append(Lk_1)
64.         Ci = generate_Ck(Lk_1, k)
65.         Li = generate_Lk_from_Ck(data, Ci, min_support, sup
    port_dict)
66.         Lk_1 = copy.deepcopy(Li)
67.         k += 1
68.     return L, support_dict
69.
70. if __name__ == "__main__":
71.     min_support = 0.03
72.     data = load_data()
73.     start_time = time.time()
74.     L, support_dict = generate_L_and_support_dict(data, min
    _support)
75.     end_time = time.time()
76.     for Li in L:
77.         for item in Li:
78.             print("频繁项集:", list(item), "支持
    度:", support_dict[item])
79.     print("用时:", end_time - start_time, 's')
```