利用 FP-Growth 算法对“电影推荐数据集”进行处理，给出处理结果，给出结论描述。

电影数据集关联度分析

胡成成——41724260——通信 1701

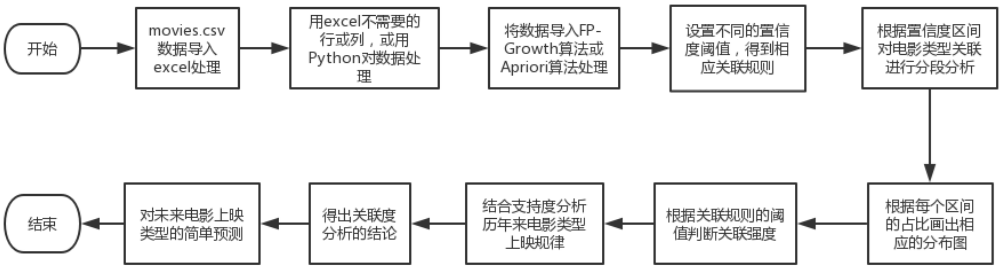
随着时代的变迁，科技的发展，看电影也成了人们生活的一部分。该篇报告对 2017 年之前的上映过的电影及其分类的数据集进行处理，旨在通过关联度分析对电影的一些类别联系进行研究，并对各类电影类别占比分析，对未来电影上映类别的预测。现在通过 FP-Growth 算法对这样的数据进行分析，通过支持度和置信度得到关联规则，通过对关联规则的分析处理得到我们要的结论。并在对电影数据集的处理中，通过实际操作处理，实践学习 FP-Growth 算法的关联度处理过程以及一般的流程。

1. 数据集引入分析：

第一列：电影的 id 编号，第二列，电影名称及其年份，第三列，电影所属的类别：包括 Adventure（冒险），Animation（动画），Children（儿童），Comedy（喜剧），Fantasy（幻想）等等类型。数据一共 9125 条。

	A	B	C
1	movieId	title	genres
2	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
3	2	Jumanji (1995)	Adventure|Children|Fantasy
4	3	Grumpier Old Men (1995)	Comedy|Romance
5	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
6	5	Father of the Bride Part II (1995)	Comedy
7	6	Heat (1995)	Action|Crime|Thriller
8	7	Sabrina (1995)	Comedy|Romance
9	8	Tom and Huck (1995)	Adventure|Children
10	9	Sudden Death (1995)	Action
11	10	GoldenEye (1995)	Action|Adventure|Thriller
12	11	American President, The (1995)	Comedy|Drama|Romance
13	12	Dracula: Dead and Loving It (1995)	Comedy|Horror
14	13	Balto (1995)	Adventure|Animation|Children
15	14	Nixon (1995)	Drama
16	15	Cutthroat Island (1995)	Action|Adventure|Romance
17	16	Casino (1995)	Crime|Drama
18	17	Sense and Sensibility (1995)	Drama|Romance
19	18	Four Rooms (1995)	Comedy
20	19	Ace Ventura: When Nature Calls (1995)	Comedy
21	20	Money Train (1995)	Action|Comedy|Crime|Drama|Thriller
22	21	Get Shorty (1995)	Comedy|Crime|Thriller
23	22	Copycat (1995)	Crime|Drama|Horror|Mystery|Thriller
24	23	Assassins (1995)	Action|Crime|Thriller
25	24	Powder (1995)	Drama|Sci-Fi
26	25	Leaving Las Vegas (1995)	Drama|Romance

2. 数据分析流程



3. 数据集预处理

	A
1	
2	Adventure|Animation|Children|Comedy|Fantasy
3	Adventure|Children|Fantasy
4	Comedy|Romance
5	Comedy|Drama|Romance
6	Comedy
7	Action|Crime|Thriller
8	Comedy|Romance
9	Adventure|Children
10	Action
11	Action|Adventure|Thriller
12	Comedy|Drama|Romance
13	Comedy|Horror
14	Adventure|Animation|Children
15	Drama
16	Action|Adventure|Romance
17	Crime|Drama
18	Drama|Romance
19	Comedy
20	Comedy
21	Action|Comedy|Crime|Drama|Thriller
22	Comedy|Crime|Thriller
23	Crime|Drama|Horror|Mystery|Thriller
24	Action|Crime|Thriller
25	Drama|Sci-Fi
26	Drama|Romance

4. 数据处理

4.1 相关参数与标准设定

由于关联度分析需要对支持度和置信度分析处理，在这里我们对最小支持度根据数据集实际情况进行设定。对置信度进行分区段展示分析。

- 最小支持度：100
- 置信度 conf（P（A&B）/P(A)）的区间与 A 对 B 关联的等级划分：

置信度	conf<0.4	0.4<conf<0.55	0.55<conf<0.7	Conf>0.7
关联等级	很弱	较弱	较强	很强

- 支持度 support(sup)>100，这里用数量等价表示：

支持度	Sup<500	500<sup<1000	1000<sup<2000	sup>2000
数量等级	较低	中等	较高	很高

4.2 根据设定的等级对不同置信区间筛选

对程序（见附录）中的 min_conf 与 max_conf 进行调整得出四种情况下的存在关联的个数和规则。

置信度	conf<0.4	0.4<conf<0.55	0.55<conf<0.7	Conf>0.7
关联个数	167	28	13	2

- 相关程序块代码：

```
#关联规则和置信度
i = 0
min_conf=0.55
max_conf=0.7
association_rules = []
for item_set in frequent_all_list:
    for conclusion in frequent_all_list:
        if conclusion > item_set:
            confidence = float(frequent_all_key_value_set[str(list(conclusion))] / frequent_all_key_value_set[str(list(item_set))])
            if confidence > min_conf and confidence<max_conf:
                i += 1
                association_rules.append([[item_set, conclusion-item_set], confidence])
print(i)
print(association_rules)
```

- 对 conf>0.7 的数据关联规则做展示：

[[[{frozenset({'War'})},{frozenset({'Drama'})}],0.771117166212534],
[[{frozenset({'Adventure'}),frozenset({'Thriller'})},{frozenset({'Action'})}],0.7849462365591398]]

- 对 0.55<conf<0.7 部分关联规则展示：

[[[{frozenset({'Animation'})}, {frozenset({'Children'})}], 0.5525727069351231], [[{frozenset({'Animation'}), frozenset({'Comedy'})}, {frozenset({'Children'})}], 0.6256410256410256], [[{frozenset({'Animation'}), frozenset({'Adventure'})}, {frozenset({'Children'})}], 0.6291079812206573], [[{frozenset({'Mystery'})}, {frozenset({'Thriller'})}], 0.6058931860036832], [[{frozenset({'Mystery'}), frozenset({'Crime'})}, {frozenset({'Thriller'})}], 0.6826347305389222],……]]

- 其他由于数据过多不做展示，只做后续统计分析。

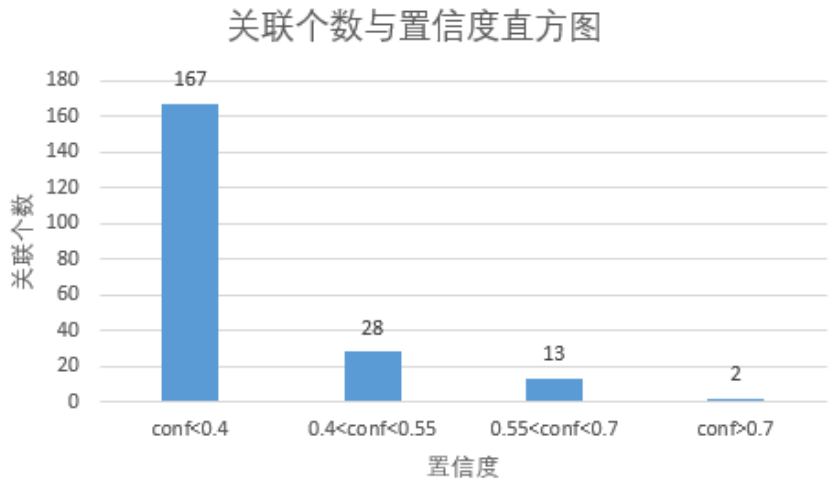
4.3 根据对置信度处理的结果初步分析

根据置信度定义可知在 2017 年之前的电影中，如果一部电影是战争片（War），有将近 77%的概率是戏剧片（Drama）。如果一部电影同时是冒险片（Adventure）和恐怖片（Thriller），有将近 78%的概率是动作片（Action）。也就是得知一部电影是战争片，它极大可能也就是部戏剧片。

同理，随着置信度区间的降低，这种关联关系也会减弱。

4.4 得出关联个数与置信度之间的关系直方图

可以看出大多数电影类型间没有必然的关联，少数电影类型之间存在着强烈的关联，即上述分析的战争片（War）大多是喜剧片（Drama）一样。而正是这些少数的有很强关联性的数据也正是我们需要的。



4.5 根据设定的等级对不同的支持度区间筛选

对程序（见附录）中的支持度区间判断哪些类型的电影上映最多，对数量做一个排序分析。

支持度	100<Sup<500	500<sup<1000	1000<sup<2000	sup>2000
数量个数	67	13	5	2

- 支持度相关代码块：

```
#支持度分析

set_min_support=100

set_max_support=500

j=0

for fre_key,fre_value in frequent_all_key_value_set.items():

    if fre_value>set_min_support and fre_value<set_max_support:

        print(fre_key,fre_value)

        j+=1

print(j)
```

- 对 sup>2000 的数量级进行分析输出结果：

```
 [frozenset({'Comedy'})] 3315

[frozenset({'Drama'})] 4365
```

➢ 对 1000<sup<2000 的数量级进行分析输出结果：

```
[frozenset({'Crime'})] 1100

[frozenset({'Adventure'})] 1117

[frozenset({'Romance'})] 1545

[frozenset({'Action'})] 1545

[frozenset({'Thriller'})] 1729
```

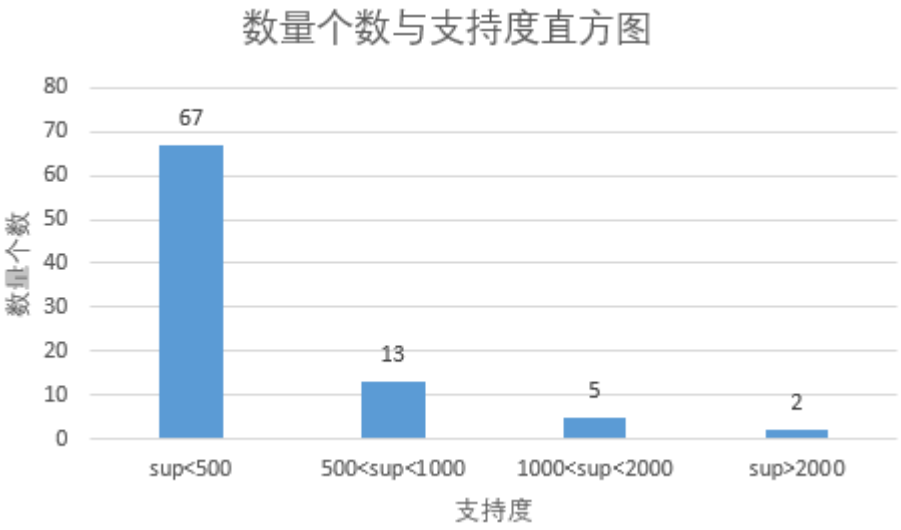
➢ 对 1000>sup 的数量级输出在这里不做展示。

4.6 根据对支持度处理的结果初步分析

根据支持度定义可知在 2017 年之前的电影中，有 4365 部，将近一半的电影（数据总量 9125）是戏剧（Drama）；有 3315 部电影是喜剧（Comedy）。同时，犯罪（Crime），冒险（Adventure），浪漫（Romance），动作（Action），恐怖（Thriller）电影也较多，都达到了一千多部。

4.7 得出支持度与数量个数之间的关系直方图

可以看出大多数电影都是喜剧（Comedy）和戏剧（Drama），其次是犯罪（Crime），冒险（Adventure），浪漫（Romance），动作（Action），恐怖（Thriller）。



5. 分析数据

5.1 支持度分析

➢ 对于数据支持度，首先我们知道了大多数电影类型是喜剧（Comedy）和

戏剧（Drama）。但我们知道这两种类型是很大的范畴，所以他的支持度很高是可以理解的。

- 其次，是犯罪（Crime），冒险（Adventure），浪漫（Romance），动作（Action），恐怖（Thriller）这几种类型的电影，这些类型在我们日常生活确实很多见，其中恐怖片 1729 部，占据这一阶段的最高值。
- 再往下看，我们根据下面程序输出结果，在 500 到 1000 这一个区段，有神秘（Mystery），儿童（Children），幻想（Fantasy），科幻（Sci-Fi）等等类型的也都有，且数量也不少。

```
[frozenset({'Mystery'})] 543
[frozenset({'Children'})] 583
[frozenset({'Fantasy'})] 654
[frozenset({'Sci-Fi'})] 792
[frozenset({'Horror'})] 877
[frozenset({'Crime'}), frozenset({'Thriller'})] 528
[frozenset({'Drama'}), frozenset({'Crime'})] 617
[frozenset({'Adventure'}), frozenset({'Action'})] 532
[frozenset({'Romance'}), frozenset({'Comedy'})] 822
[frozenset({'Romance'}), frozenset({'Drama'})] 924
[frozenset({'Action'}), frozenset({'Thriller'})] 574
[frozenset({'Drama'}), frozenset({'Thriller'})] 795
[frozenset({'Comedy'}), frozenset({'Drama'})] 949
```

- 最后其他的数量小于 500，相对于 9000+的数据已经很小了，这里就不详细分析。

5.2 关联规则分析。

- 首先我们可以看到战争片（War）有将近 77%的概率是戏剧片（Drama）。

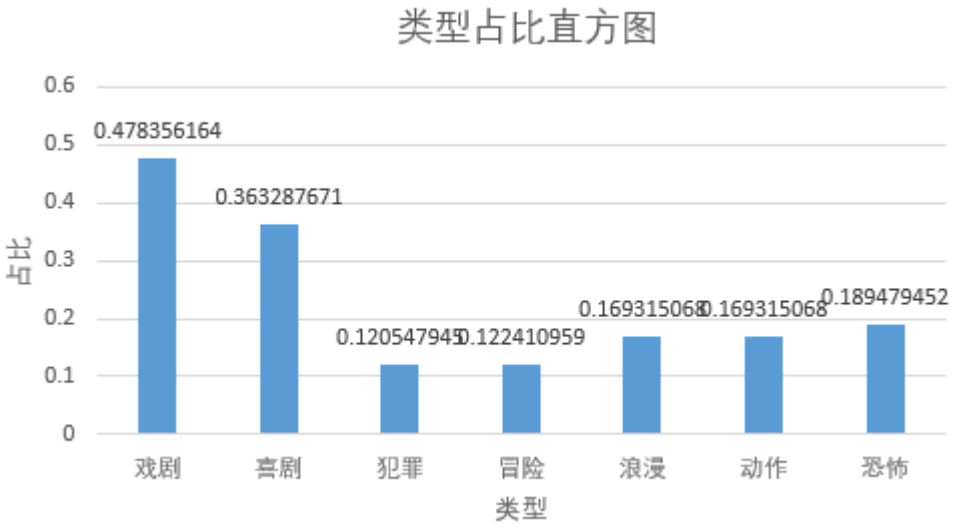
冒险片（Adventure）和恐怖片（Thriller）有将近 78%的概率是动作片（Action）。也就是当我们看到一部电影是战争片，它很大可能是戏剧。

- 其次，一部动画（Animation）有较高概率是儿童片（Children）；以及同时是动画（Animation）和戏剧（Comedy）有较高概率是儿童片（Children）；神秘（Mystery）有较高概率是恐怖片（Thriller）等等。
- 对于置信度小于 0.4 的，像战争片（War）就不太可能是动作片（Action）等等，如下摘取了部分数据结果。

```
[[[{frozenset({'War'})},      {frozenset({'Action'})}],      0.3242506811989101],
[[{frozenset({'Musical'})},   {frozenset({'Romance'})}],   0.2868020304568528],
[[{frozenset({'Musical'})},{frozenset({'Drama'})}],0.3223350253807107],……
…]]
```

5.3 根据 2017 年以前的电影类型来预测之后的电影大概会那些类型较多。

- 根据支持的再分析，可以得到前几的电影类型占比直方图：



根据直方图的分布，可大致预测之后的电影也可能按着样的比例上映。

6. 结论

- 在整个电影数据集共 9125 条电影类别数据进行处理，从中得出喜剧和喜剧的支持度最高，大多数电影都是这两个类型。
- 同时，一部电影是战争片很强的概率是戏剧片。一部同时是恐怖和冒险

的电影极大可能是动作片。

7. 附录：程序源码

```
import pandas as pd
import numpy as np
data1 = pd.read_csv('movies.csv', header=None, error_bad_lines=False)
data1.head()
data = []
with open('movies.csv') as f:
    for line in f.readlines():
        line = line.strip()
        if line:
            row_list = line.split('|')
            data.append(row_list)

len(data)

class Node:

    def __init__(self, name, count, parent):
        self.name = name
        self.count = count
        self.link = None
        self.parent = parent
        self.children = {}
    def increment(self, num):
        self.count += num
    def display(self, lens=1):
        print('    ' * lens, self.name, ' ', self.count)
        for child in self.children.values():
            child.display(lens + 1)
class FPTree:
    def __init__(self, transactions, min_support, root_value, count, flag):

        self.flag = flag
        self.root = Node(root_value, count, None)
        self.data = transactions
        self.data_type = type(self.data)
        self.min_support = min_support
        self.frequent = self.find_frequent_items()
        self.headerTable = {v: [self.frequent[v], None] for v in self.frequent}
    def find_frequent_items(self):
        from collections import defaultdict
        freq1 = defaultdict(int)
```

```
        # 统计列表中各个元素的出现频次
        if self.data_type == list and self.flag == 'fptree':
            flatten_list = [element for item in self.data for element in item]
            for value in flatten_list:
                freq1[frozenset([value])] += 1
        # 条件 FP 树
        elif self.data_type == dict and self.flag == 'cfptree':
            for item in self.data:
                for element in item:
                    if type(element) == frozenset:
                        freq1[element] += self.data[item]
                    else:
                        freq1[frozenset([element])] += self.data[item]
        # 过滤掉非频繁项
        return {v: freq1[v] for v in freq1 if freq1[v] >= self.min_support}
    def build_tree(self):
        root = Node('null', 1, None)
        sorted_headertable = [v[0] for v in sorted(self.frequent.items(), key=lambda kv: (-kv[1],
list(kv[0])[0]))]
        # 将业务数据库中的每条记录插入树中，更新频繁项表
        for record in self.data:
            # 找到业务数据纪录中的频繁项
            sorted_items = [item for item in sorted_headertable if list(item)[0] in record]
            # print u'将排序后的频繁项集放入 FP 树中: '
            # print sorted_items
            # print ''
            node = self.root
            while len(sorted_items) > 0:
                first_value = sorted_items[0]
                # print u'  开始插入节点  ', first_value
                if first_value in node.children:
                    # print u'       当前节点的子节点中包含节点  ', first_value
                    if self.data_type == list:
                        node.children[first_value].increment(1)
                    elif self.data_type == dict:
                        node.children[first_value].increment(self.data[record])
                else:
                    # 创建新的子节点
                    # print u'       创建新节点  ', first_value
                    if self.data_type == list:
                        node.children[first_value] = Node(list(first_value)[0], 1, node)
                    elif self.data_type == dict:
                        node.children[first_value] = Node(first_value, self.data[record], node)
```

```
                        # 更新频繁项表
                        if self.headerTable[first_value][1] == None:
                            # print u'      在频繁项表中，添加对应链表的第一个节点', first_value
                            self.headerTable[first_value][1] = node.children[first_value]
                        else:
                            # print u'      添加到频繁项表对应链表的末尾'
                            currentNode = self.headerTable[first_value][1]
                            # 遍历连表，确保在表的尾部插入新的节点指针
                            while currentNode.link != None:
                                currentNode = currentNode.link
                            currentNode.link = node.children[first_value]
                    sorted_items.pop(0)
                    node = node.children[first_value]
    def get_tree(self):
        return self.root
    def get_frequent_items(self):
        return self.frequent

    def get_headertable(self):
        return self.headerTable
    def get_data(self):
        return self.data
    def show(self, depth=1):
        print(' ' * depth, self.root.name, ' ', self.root.count)
        for node in self.root.children.values():
            node.display(depth + 1)
min_support = 100
fptree = FPTree(data, min_support, 'null', 1, 'fptree')
freq1 = fptree.get_frequent_items()
def find_prefix(node):
    cpb = {}
    suffix_list = []
    while node != None:
        suffix_list.append(node)
        node = node.link
    for item in suffix_list:
        path = []
        num = item.count
        while item.parent != None:
            path.append(item.name)
            item = item.parent
        cpb[frozenset(path[1:])] = num
    return cpb
frequent_all_key_value_set = {}
```

```
def mine_tree(frequent_items, headerTable, min_support, frequent, item_list):
    candidates = [v[0] for v in sorted(frequent_items.items(), key=lambda kv: (-kv[1], list(kv[0])[0]))]
    global frequent_all_key_value_set
    for item in candidates[::-1]:
        freq_set = frequent.copy()
        freq_set.add(item)
        freq_set_new = set()
        for i in freq_set:
            freq_set_new.add(i)
        fre_newlist = list(freq_set_new)
        frequent_all_list.append(freq_set_new)
        b = str(fre_newlist)
        frequent_all_key_value_set[b] = frequent_items[item]
        cpbs = find_prefix(headerTable[item][1])
        # print( 'its cpbs: ', cpbs)
        #  创建条件 FP 树
        cTree = FPTree(cpbs, min_support, 'root', 1, 'cfptree')
        cTree.build_tree()
        if len(cTree.get_headertable()) != 0:
            # print ('condtional tree for: ', freq_set)
            # cTree.show(1)
            mine_tree(cTree.get_frequent_items(), cTree.get_headertable(), min_support, freq_set,
item_list)
frequent_item = []
frequent = set()
root_node = fptree.get_tree()
headerable = fptree.get_headertable()
freq_items = fptree.get_frequent_items()
mine_tree(freq_items, headerable, min_support, frequent, frequent_item)

#支持度分析
set_min_support=100
set_max_support=500

j=0
for fre_key,fre_value in frequent_all_key_value_set.items():
    if fre_value>set_min_support and fre_value<set_max_support:
        print(fre_key,fre_value)
        j+=1
print(j)
#关联规则和置信度
i = 0
#最小置信
min_conf=0.0
```

```
max_conf=0.4
association_rules = []
for item_set in frequent_all_list:
    #print(item_set)
    for conclusion in frequent_all_list:
        #print(conclusion)
        if conclusion > item_set:
            confidence = float(frequent_all_key_value_set[str(list(conclusion))] /
frequent_all_key_value_set[str(list(item_set))])
            #print(confidence)
            #print(conclusion)
            if confidence > min_conf and confidence<max_conf:
                i += 1
                association_rules.append([[item_set, conclusion-item_set], confidence])
                #print('conclusion:', conclusion-item_set, 'condition:', item_set, 'confidence:', confidence)
print(i)
print(association_rules)
```