数据科学基础第二次实践

1. 考查内容

2. 实验说明

4. 代码设计与实现

   4.1 数据处理

   4.2 Dummy Apriori

   4.3 Apriori1

   4.4 Apriori2

   4.5 Apriori3

   4.6 FP_Growth算法

   4.7 calculate_association_rules

5. 实验结果与分析

   5.1 使用 apriori 算法在 Groceries数据集上挖掘频繁 3-项集(支持度 0.01)

   5.2 使用 apriori 算法在 Groceries数据集上挖掘关联规则(支持度 0.01，置信度 0.5)

   5.3 使用 FP-Growth 算法在 Groceries 数据集上挖掘的一些关联规则(置信度 0.5)

   5.4 对比 dummy apriori、仅使用第一种剪枝策略 advanced apriori、同时 使用第一种和第二种剪枝策略 a...

   5.5 使用命令行接收参数(数据集、支持率、挖掘频繁项集的大小

   5.6 寻找 2 个关联规则，比较强的关联规则，进行市场分析

   5.7 实现第四部分第三种剪枝策略，并将三种剪枝策略一起使用算法运行时间一起画在算法时 间对比图表中

   5.8

      apriori1 的内存使用情况

      apriori2 的内存使用情况

      apriori3 的内存使用情况

      fpgrowth 的内存使用情况

   6. 运行文件说明

1. 考查内容

频繁模式与关联规则挖掘，考察对 apriori&&fpgrowth 算法的掌握程度。

2. 实验说明

1. 在给定数据集上(10,000 量级的数据)上使用关联规则挖掘算法。
2. 通过改变不同等级的支持度和置信度，比较 apriori、fpgrowth 和 baseline 算法(说明见)的性能。
3. 试图根据 apriori 或 fpgrowth 算法揭露出的物品集相关性，发现一些有趣的关联规则。

4. 代码设计与实现

4.1 数据处理

将给定的csv文件处理成字典列表形式，每一个list代表是一次事务。

```
def load_data(path='./data/Groceries.csv'):
    r"""
    load info from origin data
    generate the itemset and c_1.
    """
    itemset = []
    goods = {}
    with open(path, 'r') as f:
        for i, line in tqdm(enumerate(f.readlines())):
            if i == 0:
                continue
            words = line.replace('{', '').replace('}', '').replac
e(
                '"', '').strip().split(',')[1:]
            item = {}
            if len(words) > 0:
                for word in words:
                    word = word.replace(' ', '').replace('/', '')
                    goods[word] = 1
                    item[word] = 1
                itemset.append(item)
    c_1 = []
    for word in list(goods.keys()):
        c_1.append(set([word]))
    return itemset, c_1
```

4.2 Dummy Apriori

没有进行剪枝，即用暴力搜索的方法根据 k 项频繁 项集生成候选 k +1 项集，对事务表不做任何处理。
下面对该算法进行一个简单的 `review`

Step1. 生成候选集

```
def generate_c_k_plus_1(l_ks):
    """ generate C_{k+1} 候选集
    l_ks: a set of c_k
    """
    c_k_plus_1s = {}
    for i in tqdm(range(0, len(l_ks)),
                  desc=f'Generate the c_k_plus_1 {len(l_ks[0])+1} dataset', mininterval=0.1):
        for j in range(i + 1, len(l_ks)):
            # 去重复
            item = sorted(l_ks[i].union(l_ks[j]))
            c_k_plus_1s[tuple(item)] = 1
    logger.info(f'get the c_k+1 {len(l_ks[0])+1}:{len(c_k_plus_1s)}')
    return [set(c_k_1) for c_k_1 in list(c_k_plus_1s.keys())]
```

Step2. 根据候选集过滤出符合支持度的频繁集

支持度的计算方法

```
def support_rate(itemset, c_k, num):
    """ support rate
    支持度
    itemset: 数据集
    c_k, the k-th 候选集
    """
    count = 0.0
    record_nums = [0 for _ in range(len(itemset))]
    for i, item in enumerate(itemset):
```

```
        hit = 0
        for c in c_k:
            if c in item:
                hit += 1
        if len(c_k) == hit:
            count += 1
            record_nums[i] = 1
    return count / num, record_nums
```

根据候选集过滤出符合支持度的频繁集

```
def generate_l_k(itemset, c_ks, support, num, counter, trick=None
):
    """ Generate l_k from c_k
    trick: trick or not
    Return:
       候选集、支持度、新的事务表（可能会使用trick）
    """
    l_k = []
    sp_s = []
    k = len(c_ks[0])
    desc = f'Generate l_k k={k}'
    all_record_nums = [0 for _ in range(len(itemset))]
    for c_k in tqdm(c_ks, desc=desc, mininterval=0.1):
        sp, record_nums = support_rate(itemset, c_k, num)
        for i in range(len(record_nums)):
            all_record_nums[i] += record_nums[i]
        if sp > support:
            l_k.append(c_k)
            sp_s.append(sp)

    # Advanced aprorio 2 filter
    if trick is None:
        logger.info('dummy algorithm')
    elif trick == 'trick_2':
        itemset = trick_2(all_record_nums, itemset, k)
    elif trick == 'trick_3':
        itemset = trick_3(counter, k, itemset)
```

```
27
28      logger.info(f' get l_k len:{len(l_k)} ')
29      return l_k, sp_s, itemset
```

Step3. 找出所有的频繁集，计算其关联规则，找到符合要求的关联规则

```
1  args = opt()
2  all_lks, all_sps, interval = main(args)
3  k_set = get_k_set(all_lks, k=args.k)
4  logger.info(
5         f"len(all_lks):{len(all_lks)}, len(k_set):{len(k_set)}, k_
   set:{k_set}")
6  rules = get_association_rules(
7         all_lks, all_sps, args.confidence, model=args.model)
8  print(f"len(rules):{len(rules)}, rules:{rules}")
```

4.3 Apriori1

主要作用为: 减小生成候选项集规模。如果一个 k+1 候选项是频繁的，那么生成它的 口 频繁项集中必包含其 k+1 个子集。例 如，如果{t1, t2, t3, t5}是频繁的，那么它将由{t1, t2, t3}和{t1, t2, t5}生成，否则它就不会 被生成。这么做要求生成候选项集的频繁项集的项内是有序的，各项之间也是有序的。

```
1  def advanced_apriori1(itemset, c_ks, support, num):
2      all_lks = []
3      all_sps = []
4      counter = count_freq(itemset)
5
6      def apriori(itemset, c_k, support):
7          l_ks, sp_s, itemset = generate_l_k(itemset, c_k, support,
   num, counter)
8          all_lks.extend(l_ks)
9          all_sps.extend(sp_s)
10         if len(l_ks) > 1:
11             ck_plus_1s = generate_c_k_plus_1(l_ks)
12             # 类间排序
```

```
            ck_plus_1s = sorted(ck_plus_1s)
            apriori(itemset, ck_plus_1s, support)

    apriori(itemset, c_ks, support)

    return all_lks, all_sps
```

4.4 Apriori2

主要作用是：减小事务表(数据记录表)规模。
如果一个 k+1 项候选项能与事务表中一条数据记录匹配，那么其 k+1 个子集也必能与 事务表中该条记录匹配。因此在事务表中匹配 k 候选项集的时候，统计每条记录被匹配 到的次数，如果少于 k+1 次，那么将该条记录从事务表中移除，因为它绝不可能与下一 轮的任一 k+1 项候选项匹配。这样每次迭代都减小了事务表的规模，从而减小扫描事务 表的时间消耗。

```

def trick_2(all_record_nums, itemset, k):
    """
    trick_2: 减小事务表(数据记录表)规模。

    """
    dataset = []
    for record_num, item in zip(all_record_nums, itemset):
        if record_num > k:
            dataset.append(item)
    logger.info(f"before: {len(itemset)}, after: {len(dataset)}")
    return dataset
```

4.5 Apriori3

主要作用：减少事务表中元组的项
如果事务表中元组的某一项能包含在一个 k+1 频繁项中，那么该项必出现在这个 k+1 频 繁项的 k 个 k项子集中。所以在扫描事务表统计 k 项候选集的出现频次时，如果事务表 中任一元组的某一项未被匹配中 口次,那么该项将在筛选出 k 频繁项集后从元组中除去， 从而减少统计 k+1 项候选集频次时与事务表中元组的匹配次数

```python
# 首先统计一次全部元素的频率
def count_freq(itemset):
    """ Count freq of item in itemset
    itemset: dataset
    """
    counter = {}
    for item in tqdm(itemset):
        for record in item:
            if record in counter:
                counter[record] += 1
            else:
                counter[record] = 1

    return counter


def trick_3(counter, k, itemset):
    """
    trick_3 : 减少事务表中元组的项。

    Parameters
    ----------
    counter : [dict]
        每一个元素的词频
    k : [int]
        候选集元素的个数
    itemset : [list]
        事务表
    """
    dataset = []
    for item in itemset:
        elements = copy.deepcopy(item)
        for element in item.keys():
            if counter[element] <= k:
                del elements[element]
        if len(elements) > 0:
            dataset.append(elements)
    return dataset
```

4.6 FP_Growth算法

这个通过调包实现，比较简单。使用时为了方便调用，做了一下封装。

```
def get_association_rules(all_lks, all_sps, confidence, model='fp'
):
    if model == 'fpgrowth':
        return generate_association_rules(all_lks, confidence)
    elif model == 'apriori':
        return calculate_association_rules(all_lks, all_sps, confi
dence)
    else:
        raise(
            f"Not support algorithm {model}, Please choose from [`
fpgrowth`,`apriori`]")
```

4.7 calculate_association_rules

计算其置信度，找出其关联规则. 这个是自己实现的一个查找关联规则的方法，但是在与pyfpgrowth包挖掘关联规则的结果对比中，发现结果存在不一致的问题。

```
def calculate_association_rules(all_lks, all_sps, confidence):
    counter = {}

    # Step1. 将 list to map
    for lk, sp in tqdm(zip(all_lks, all_sps)):
        counter[tuple(sorted(lk))] = sp
    rules = []
    filters = {}
    full_rules = []
    for lk in tqdm(all_lks):
        # print(type(lk))
        for i in range(1, len(lk)):
            combines = list(combinations(lk, i))
            for a in combines:
                a_s = tuple(sorted(a))
                b = list(set(lk).difference(set(a)))
                b_s = tuple(sorted(b))
```

```
                key = tuple(sorted(lk))
                max_info = None
                rule = None
                max_confidence = 0
                if a_s in counter and (counter[key] / counter[a_s]) > confidence and (counter[key] / counter[a_s]) > max_confidence:
                    info = f'{a_s}:{b_s}, {(counter[key] / counter[a_s])}'
                    max_confidence = (counter[key] / counter[a_s])
                    if info not in filters:
                        max_info = info
                        filters[info] = 1
                        rule = (a_s, b_s, max_confidence)
                if b_s in counter and (counter[key] / counter[b_s]) > confidence and (counter[key] / counter[b_s]) > max_confidence:
                    info = f'{b_s}:{a_s}, { (counter[key] / counter[b_s])}'
                    max_confidence = (counter[key] / counter[b_s])
                    if info not in filters:
                        max_info = info
                        filters[info] = 1
                        rule = (a_s, b_s, max_confidence)
                if max_info is not None:
                    rules.append(rule)
                    full_rules.append(lk)
    # print(full_rules)
    return rules
```

5. 实验结果与分析

下面要求均已完成

5.1 使用 apriori 算法在 Groceries数据集上挖掘频繁 3-项集(支持度 0.01)

使用 Apriori2算法得到的结果如下：

```
len(all_lks):333, len(k_set):32, k_set:[{'wholemilk', 'yogurt', 'citrusfruit'}, {'othervegetables', 'wholemilk', 'citrusfruit'}, {'othervegetables', 'rootvegetables', 'citrusfruit'}, {'wholemilk', 'yogurt', 'tropicalfruit'}, {'othervegetables', 'yogurt', 'tropicalfruit'}, {'othervegetables', 'wholemilk', 'tropicalfruit'}, {'wholemilk', 'tropicalfruit', 'rollsbuns'}, {'rootvegetables', 'wholemilk', 'tropicalfruit'}, {'othervegetables', 'rootvegetables', 'tropicalfruit'}, {'othervegetables', 'wholemilk', 'yogurt'}, {'wholemilk', 'yogurt', 'rollsbuns'}, {'curd', 'wholemilk', 'yogurt'}, {'yogurt', 'wholemilk', 'soda'}, {'rootvegetables', 'wholemilk', 'yogurt'}, {'whippedsourcream', 'yogurt', 'wholemilk'}, {'othervegetables', 'yogurt', 'rollsbuns'}, {'othervegetables', 'rootvegetables', 'yogurt'}, {'othervegetables', 'whippedsourcream', 'yogurt'}, {'othervegetables', 'wholemilk', 'pipfruit'}, {'othervegetables', 'butter', 'wholemilk'}, {'othervegetables', 'wholemilk', 'rollsbuns'}, {'othervegetables', 'wholemilk', 'bottledwater'}, {'othervegetables', 'wholemilk', 'soda'}, {'othervegetables', 'fruitvegetablejuice', 'wholemilk'}, {'othervegetables', 'wholemilk', 'pastry'}, {'othervegetables', 'rootvegetables', 'wholemilk'}, {'othervegetables', 'sausage', 'wholemilk'}, {'othervegetables', 'wholemilk', 'pork'}, {'othervegetables', 'whippedsourcream', 'wholemilk'}, {'othervegetables', 'wholemilk', 'domesticeggs'}, {'rootvegetables', 'wholemilk', 'rollsbuns'}, {'othervegetables', 'rootvegetables', 'rollsbuns'}]
```

5.2 使用 apriori 算法在 Groceries数据集上挖掘关联规则(支持度 0.01, 置信度 0.5)

使用 Apriori2算法得到的结果如下

```
en(rules):14, rules:[(('othervegetables',), ('citrusfruit', 'rootvegetables'), 0.5862068965517241), (('wholemilk',), ('tropicalfruit', 'yogurt'), 0.517361111111111), (('wholemilk',), ('rootvegetables', 'tropicalfruit'), 0.570048309178744), (('othervegetables',), ('rootvegetables', 'tropicalfruit'), 0.5845410628019324), (('wholemilk',), ('othervegetables', 'yogurt'), 0.5128805620608898), (('wh
```

```
olemilk',), ('curd', 'yogurt'), 0.5823529411764706), (('wholemilk'
,), ('rootvegetables', 'yogurt'), 0.562992125984252), (('wholemil
k',), ('whippedsourcream', 'yogurt'), 0.5245098039215685), (('whol
emilk',), ('othervegetables', 'pipfruit'), 0.5175097276264592), ((
'wholemilk',), ('butter', 'othervegetables'), 0.5736040609137055),
(('wholemilk',), ('othervegetables', 'whippedsourcream'), 0.507042
2535211268), (('wholemilk',), ('domesticeggs', 'othervegetables'),
0.5525114155251142), (('wholemilk',), ('rollsbuns', 'rootvegetable
s'), 0.5230125523012552), (('othervegetables',), ('rollsbuns', 'ro
otvegetables'), 0.502092050209205)]
```

5.3 使用 FP-Growth 算法在 Groceries 数据集上挖掘的一些关联规则(置信度 0.5)

```
len(rules):7, rules:{('curd', 'yogurt'): (('wholemilk',), 0.582352
9411764706), ('butter', 'othervegetables'): (('wholemilk',), 0.573
6040609137056), ('domesticeggs', 'othervegetables'): (('wholemilk'
,), 0.5525114155251142), ('othervegetables', 'whippedsourcream'):
(('wholemilk',), 0.5070422535211268), ('othervegetables', 'pipfrui
t'): (('wholemilk',), 0.5175097276264592), ('citrusfruit', 'rootve
getables'): (('othervegetables',), 0.5862068965517241), ('otherveg
etables', 'yogurt'): (('wholemilk',), 0.5128805620608899)}
```

5.4 对比 dummy apriori、仅使用第一种剪枝策略 advanced apriori、*同时 使用第一种和第二种剪枝策略* advanced apriori 的时间损耗

```
finish algorithm with 110.9324722290039s # dummy apriori
finish algorithm with 112.84710025787354s # apriori1
finish algorithm with 71.55795574188232s # apriori2
finish algorithm with 106.52765893936157s # apriori3
finish algorithm with 0.8581380844116211s # FP-Growth
```

5.5 使用命令行接收参数(数据集、支持率、挖掘频繁项集的大小

使用方法，在命令行中使用：

```
python main.py -support=0.01 -confidence=0.5 -k=3 -path='' -ty='dummy'
```

各个参数的含义如下：

```
parser = argparse.ArgumentParser(
        description='The association rule mining')
# 支持率
parser.add_argument('-support', type=float,
                        default=0.01, help='The support rate')
# 置信度
parser.add_argument('-confidence', type=float,
                        default=0.5, help='The confidence rate')
# 挖掘频繁项集的大小
parser.add_argument('-k', type=int, default=3,
                        help='The size of k_item_freq')
# 数据集路径
parser.add_argument('-path', type=str,
                        default='./data/Groceries.csv',
                        help='The path of dataset')
# 算法选择
parser.add_argument('-ty', type=str,
                        default='dummy',
                        help='The type of algorithm, must be one of [`dummy`,`apriori1`,`apriori2`,`apriori3`,`fpgrowth`]')
```

5.6 寻找 2 个关联规则，比较强的关联规则，进行市场分析

('citrusfruit', 'rootvegetables'): (('othervegetables',), 0.5862068965517241)

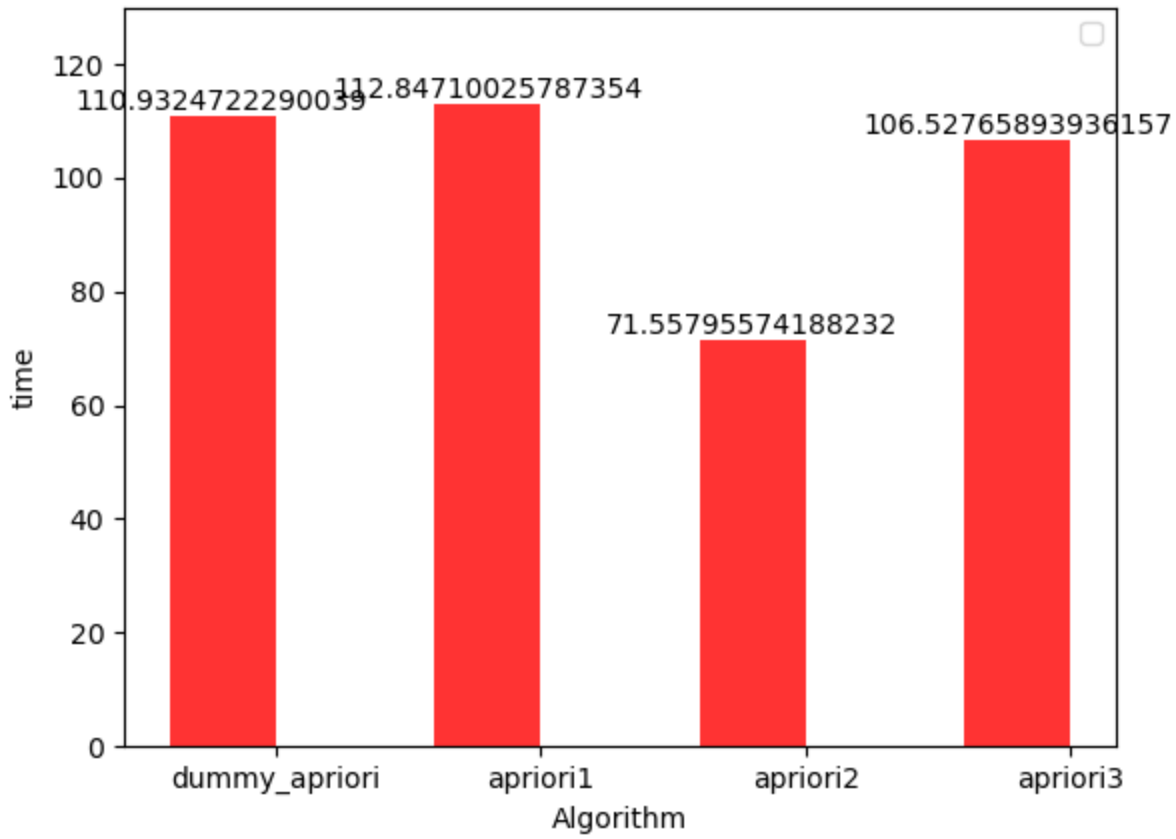
对于素食主义者或者节食的人，可能在看水果和蔬菜时，也会考虑其他品种的蔬菜

('curd', 'yogurt'): (('wholemilk',), 0.5823529411764706)

'凝乳'，'酸奶'，'全脂牛奶'，这三者都是乳制品，在购买纯牛奶时可以页会考虑酸奶或者凝乳。

5.7 实现第四部分第三种剪枝策略，并将三种剪枝策略一起使用算法运行时间一起画在算法时 间对比图表中

`apriori3` 算法即是第三种剪枝方案。这四种方案（baseline与三种剪枝）的算法运行时间对比图如下：



从数据集的表现来看，advanced_2 apriori 性能是最好的，advanced_1 apriori性能与advanced_3 apriori性能相近。

5.8

dummy apriori 的内存使用情况

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
   130   39.012 MiB   39.012 MiB           1   @profile
```

apriori1 的内存使用情况

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
   130   39.062 MiB   39.062 MiB           1   @profile
   131                                         def opt():
   132   39.062 MiB    0.000 MiB           1       parser = argpar
```

apriori2 的内存使用情况

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
   130   39.027 MiB   39.027 MiB           1   @profile
   131                                         def opt():
```

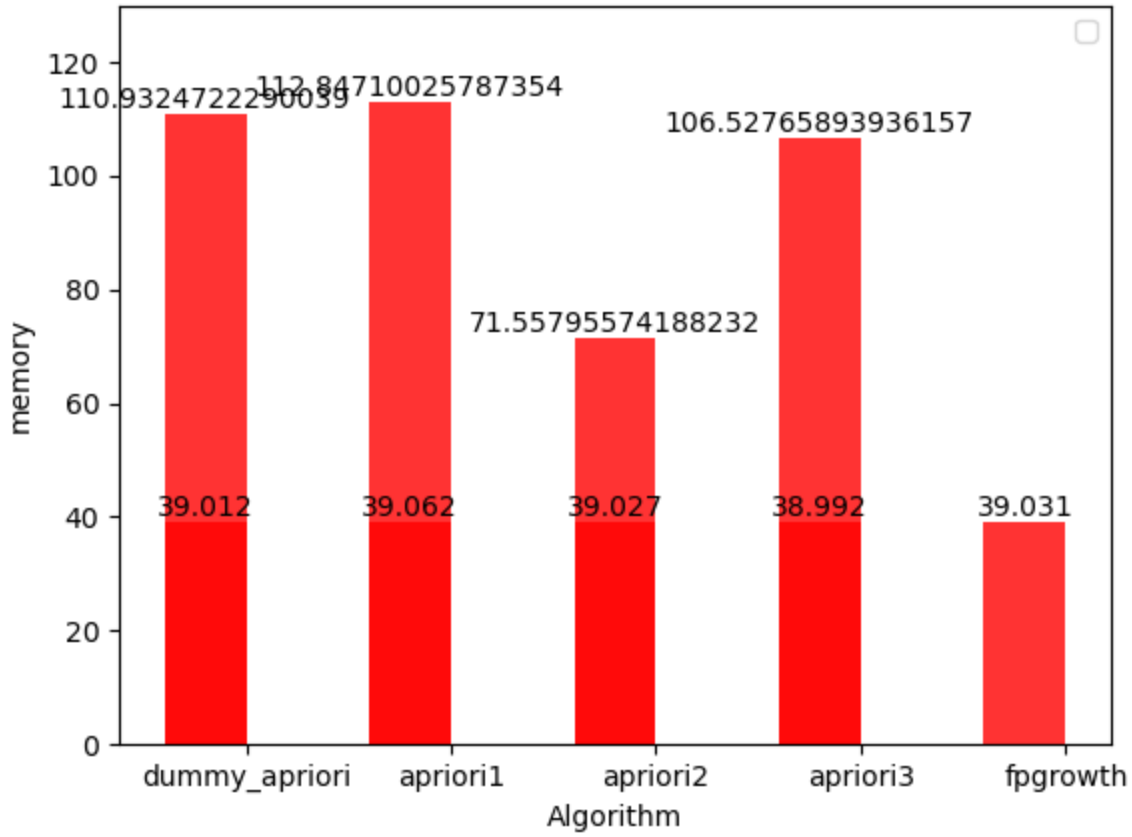
apriori3 的内存使用情况

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
   130   38.992 MiB   38.992 MiB           1   @profile
```

fpgrowth 的内存使用情况

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
   130   39.031 MiB   39.031 MiB           1   @profile
   131                                         def opt():
```

好像没看出啥差别，可能我实现的方法不对



6. 运行文件说明

main.py 为主函数，运行该文件是挖掘频繁集和关联规则，使用方法见5.6说明

draw.py 是对一些运行数据的图形绘制

out.log 是运行时的一些日志