華中科技大學

课 程 实 验 报 告

课程名称： 大数据分析

专业班级： **CSXJ1901**

学　　号： **U201911111**

姓　　名： 刘庆远

指导教师： 王蔚

报告日期： **2022/1/7**

计算机科学与技术学院

目录

实验一 wordCount 算法及其实现 1

1.1 实验目的 1

1.2 实验内容 1

1.3 实验过程 1

1.3.1 编程思路 1

1.3.2 遇到的问题及解决方式 2

1.3.3 实验测试与结果分析 2

1.4 实验总结 3

实验二 PageRank 算法及其实现 4

2.1 实验目的 4

2.2 实验内容 4

2.3 实验过程 4

2.3.1 编程思路 4

2.3.2 遇到的问题及解决方式 5

2.3.3 实验测试与结果分析 5

2.4 实验总结 6

实验三 关系挖掘实验 7

3.1 实验内容 7

3.2 实验过程 7

3.2.1 编程思路 7

3.2.2 遇到的问题及解决方式 11

3.2.3 实验测试与结果分析 11

3.3 实验总结 12

实验四 kmeans 算法及其实现 13

4.1 实验目的 13

4.2 实验内容 13

4.3 实验过程 14

4.3.1 编程思路 14

4.3.2 遇到的问题及解决方式 14

4.3.3 实验测试与结果分析 14

4.4 实验总结 16

大作业 推荐系统任务书 17

5.1 实验目的 17

5.2 实验内容 17

5.3 基础实验一实验过程.....19
5.3.1 编程思路.....19
5.3.2 遇到的问题以及解决方式.....24
5.3.3 实验测试结果与结果分析.....24
5.4 实验总结.....26

实验一 wordCount 算法及其实现

1.1 实验目的

1、理解 map-reduce 算法思想与流程；
2、应用 map-reduce 思想解决 wordCount 问题；
3、（可选）掌握并应用 combine 与 shuffle 过程。

1.2 实验内容

提供 9 个预处理过的源文件（source01-09）模拟 9 个分布式节点，每个源文件中包含一百万个由英文、数字和字符（不包括逗号）构成的单词，单词由逗号与换行符分割。

要求应用 map-reduce 思想，模拟 9 个 map 节点与 3 个 reduce 节点实现 wordCount 功能，输出对应的 map 文件和最终的 reduce 结果文件。由于源文件较大，要求使用多线程来模拟分布式节点。

学有余力的同学可以在 map-reduce 的基础上添加 combine 与 shuffle 过程，并可以计算线程运行时间来考察这些过程对算法整体的影响。

提示：实现 shuffle 过程时应保证每个 reduce 节点的工作量尽量相当，来减少整体运行时间。

1.3 实验过程

1.3.1 编程思路

本次试验虽然流程较为简单，但我认为还是有需要注意的点。首先是关于 map 的实现，map 的简单实现就是对于 dataset 的单词的一个记录，并没有统计。若使用 combine 的话，意味着在 map 阶段就首先要对 word 数进行一定程度的计算，在这里我使用的是有序字典 OrderedDict()实现的 combine；其次还应注意的是 python 对多线程操作，为了更准确地记录时间，个进程之间同步。我们还需要对于线程的同步进行设置:

```
for t in threads:
    t.join()  # 线程全部运行完程序才能结束
```

图 1.1 线程同步实现

在有了一个基本思路之后，就是对基本款与进阶版的实验具体实现了。

基本版：

- 指定 url 读取文件，使用 strip 函数进行分词，统计时仅用简单的列表保存，每一个 dataset 对应一个 map，在 map 后输出列表保存内容。（具体为单词+数量（数量为 1））
- 在 reduce 操作中，每一个 reduce 操作输入对应三个 map 操作结构，首先建立一个有序字典，每一次统计单词执行+1 操作即可。
- 最终 Merge 操作，对于三个 reduce 结果进行统计，实现方式与前两个一样，字典中 key 值相同的个数简单相加即可。

进阶版：

- 其中指定 url 读取文件，使用 strip 函数进行分词部分与基本款一样，但是在 map 过程首先要对每个 dataset 进行一个简单的统计，来减轻 reduce 的计算负担。最后输出 map 结果。（格式为单词+对应数量）
- reduce 过程与基本款不同处在于它需要加上 key 值对应的 value，也就是 map 预先计算的结果，而非简单加一。同时需要使用有序字典，对应相同的数据结构。
- 使用有序字典，对应相同的数据结构，其他与基本款一样。

1.3.2 遇到的问题及解决方式

在实验中，因果对线程并行计算的关系，三个操作 map，reduce 与 Merge 应该分为三个不同的函数，如果在同一个函数的话，可能由于进程之间不同步，导致 wordcount 缺词。

同时还应该注意最好不要讲 Merge 函数放在 reduce 实现内部，因为 reduce 之间相互并行，如果在 reduce 内部统计，正确的前提是 reduce 串行。理论上不正确。

1.3.3 实验测试与结果分析

基本版：

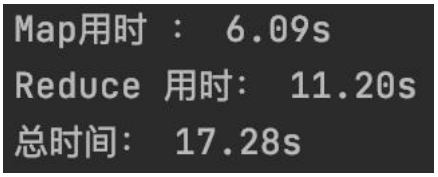

图 1.2 基本版时间统计

进阶版：

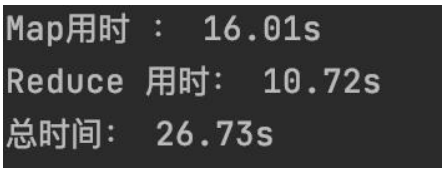

图 1.3 进阶版时间统计

结果分析：

如图所示，进阶版相较于基本版在 map 阶段耗时增加，这在代码中一方面是由于维护有序字典 OrderedDict()需要花费时间，另一个原因是 map 阶段提前分担了 reduce 的工作。对与 reduce 时间，发现进阶版会比基本版快一些，这里快的部分就是 map 阶段提前做的工作。我认为在数据集更大，以及对于用更高效的字典维护 map 结果的情况下，加上 combine 与 shuffle 的代码与基本版的差距会更大。

1.4 实验总结

虽然本次试验比较简单，不过也令我加深了对 mapreduce 原理的理解。同时也学习了关于 python 对线程处理的一些基本使用。对于 mapreduce 并行计算的优势有了一个清晰的认知，如果不对 word count 并行计算的话，可能时间花费要增加十倍以上。同时，关于每一个计算节点（对应本次试验中的线程）计算任务的平均分配我认为也是实际解决问题的重要一部分。如果任务过大，就会退变为单线程处理，任务过小则会在线程转化等方面开销过大。

实验二 PageRank 算法及其实现

2.1 实验目的

1、学习 pagerank 算法并熟悉其推导过程；
2、实现 pagerank 算法，理解阻尼系数的作用；
3、将 pagerank 算法运用于实际，并对结果进行分析。

2.2 实验内容

提供的数据集包含邮件内容（emails.csv），人名与 id 映射（persons.csv），别名信息（aliases.csv），emails 文件中只考虑 MetadataTo 和 MetadataFrom 两列，分别表示收件人和寄件人姓名，但这些姓名包含许多别名，思考如何对邮件中人名进行统一并映射到唯一 id？（提供预处理代码 preprocess.py 以供参考）。

完成这些后，即可由寄件人和收件人为节点构造有向图，不考虑重复边，编写 pagerank 算法的代码，根据每个节点的入度计算其 pagerank 值，迭代直到误差小于 10^{-8}

实验进阶版考虑加入 teleport β，用以对概率转移矩阵进行修正，解决 dead ends 和 spider trap 的问题。

输出人名 id 及其对应的 pagerank 值。

2.3 实验过程

2.3.1 编程思路

本次试验主要就是对于基本 PageRank 的实现，所以在确定编程思路之前首先要清楚 PageRank 的实际数学推导。

本次试验提供了预处理代码 preprocess.csv 与处理文件，其内部包含邮件的收发方。提取次文件就可以构造邻接矩阵。提取到邻接矩阵之后，我们就可以确定实验的 node 数量（总人数）以及每个 node 度的数。

在 PageRank 中，首先需要转移矩阵 M，M 为一个 node_num * node_num 大小的矩阵，其中每个值对应节点之间右边的情况下，节点的出度分之一。具体如下：

$$M_{ij} = \frac{1}{d_i} \quad if \quad A_{ij} = 1 \quad else \quad M_{ij} = 0$$

在构造完成转移矩阵之后，还需构造一个节点初始值 r 矩阵，其初始值为 1/node_num。

完成转移矩阵 M 与节点初始值矩阵 r 之后，下一步需要对转移矩阵进行一个初始处理：

$$A = \beta M + (1 - \beta)[\frac{1}{N}]_{N \times N}$$

这里 beta 阻尼系数值取 0.85，用以修正转移矩阵。

在完成以上操作之后，就可以进行迭代更新操作，具体实现如下：

$$r^{(t+1)} = M \times r^{(t)}$$

迭代终止条件为：

$$|r^{(t+1)} - t^{(t)}|_1 < 1e - 8$$

2.3.2 遇到的问题及解决方式

在代码实现中，需要注意的一点事关于使用阻尼系数修正转移矩阵的操作，需要放在迭代循环之外。仅进行一次即可。

同时在转移矩阵初始化结束后，需要进行一步转置操作，这一步操作使得 r 与 M 相乘时节点位置对应。

还应注意的一点是，L1 范数的计算，可能比较细这一点，我在实验中因为 L1 范数的计算不准确导致卡了很久。最后发现是求和步骤出错了。

2.3.3 实验测试与结果分析

```
id,pagerank
4,2.0778915483499145e-10
5,1.3771187720540903e-10
7,1.3771187720540903e-10
9,6.754266258807845e-10
10,6.754266258807845e-10
11,1.3771187720540903e-10
12,6.754266258807845e-10
14,4.180209877237388e-10
15,6.754266258807845e-10
16,1.0690701337505675e-09
17,4.1802098772373873e-10
20,6.754266258807845e-10
21,6.754266258807845e-10
22,9.526106988609627e-10
23,2.9848662762989635e-10
24,1.3771187720540903e-10
25,1.3771187720540903e-10
29,1.3771187720540903e-10
30,6.754266258807845e-10
31,2.7786643246457384e-10
```

图 2.1 实验结果

实验结果经检查与助教的基本一致。

最开始结果与助教结果差别很大，把修正转移矩阵的步骤提前之后也有很大的差距。

$$A = \beta M + (1-\beta)[\frac{1}{N}]_{N\times N}$$

最后在迭代循环发现，关于迭代终止条件$|r^{(t+1)} - t^{(t)}|_1 < 1e-8$，这里是用的是 L1 范数，如下。计算过程为求和，而我一开始设置成了计算最大值作为差值。

$$|X|_1 = \sum_{1<i<N} X_i$$

2.4 实验总结

通过本次试验我对 PageRank 的计算过程有了一个初步的了解，同时在做实验过程中也学习到了很多实验中没有提到的知识点，比如关于网页之间随即跳转的问题，如何实现这种问题。还比如关于黑洞节点的处理问题。同时，我也加固了很多对于机器学习基本知识掌握不牢固的地方，比如 L1 范数的计算。

实验三 关系挖掘实验

3.1 实验内容

必做：

1. 实验内容

编程实现 Apriori 算法，要求使用给定的数据文件进行实验，获得频繁项集以及关联规则。

2. 实验要求

以 Groceries.csv 作为输入文件

输出 1~3 阶频繁项集与关联规则, 各个频繁项的支持度, 各个规则的置信度, 各阶频繁项集的数量以及关联规则的总数

固定参数以方便检查，频繁项集的最小支持度为 0.005，关联规则的最小置信度为 0.5

加分项：

1. 实验内容

在 Apriori 算法的基础上，要求使用 pcy 或 pcy 的几种变式 multiHash、multiStage 等算法对二阶频繁项集的计算阶段进行优化。

2. 实验要求

以 Groceries.csv 作为输入文件

输出 1~4 阶频繁项集与关联规则, 各个频繁项的支持度, 各个规则的置信度, 各阶频繁项集的数量以及关联规则的总数

输出 pcy 或 pcy 变式算法中的 vector 的值，以 bit 位的形式输出

- 参数不变，频繁项集的最小支持度为 0.005，关联规则的最小置信度为 0.5

3.2 实验过程

3.2.1 编程思路

完成本次试验，首先需要掌握一些前置知识。

支持度：(函数均为靠我自己理解写的，不严谨的话请见谅)

$$support(x) = \frac{x_{num}}{all_{num}}$$

置信度：

$$confidence(x|Y) = P(x|Y) = \frac{support(x)}{support(Y)}$$

本次试验所要求的是，给出顾客会购买同时购买的商品集合。为了找出这个集合，有位了避免低频率商品的干扰，所以实验大体上分为两步。

◆ 第一步是找出商品的 1～k 阶的频繁项集：

找频繁项集又可细分为两步：

- 找出第 k 阶频繁项集，具体含义为找出集合大小为 k 的商品集合，刺激和出现频率大于某个阈值。
- 根据第 k 阶频繁项集，选出 k+1 阶的候选项集，选出候选项集的作用是为 k+1 阶选频繁项集提供候选数据。

上面两部不断重复，直到（1）候选项集频率全部小于给定阈值；（2）k 大小等于商品总数。最终我们可以得到 1～k 阶的频繁项集

◆ 第二步是根据关联规则，找出“同类商品”：

找“同类商品”也可细分为两步：

- 在我们最终得到 1～k 阶的频繁项集中，选出 2 阶以上的频繁项集，对每一个集合进行重新的排列组合并将其分为两部分，注意要筛选出所有可能：比如给定一个商品 set{1,2,3}，在对其进行重组拆分后，返回：
  [(frozenset({1}), frozenset({2, 3})),
  (frozenset({2}), frozenset({1, 3})),
  (frozenset({3}), frozenset({1, 2})),
  (frozenset({1, 2}), frozenset({3})),
  (frozenset({1, 3}), frozenset({2})),
  (frozenset({2, 3}), frozenset({1}))]
  也就是找出所有可能的分组。
- 定义好从单个商品找出所有可能的分组函数之后，我们就可以依靠第一步求出的频繁项集，求出频繁项集内所有集合的所有可能的分组的关联置信度，具体计算如下：

$$confidence(x|Y) = P(x|Y) = \frac{support(x)}{support(Y)}$$

整体代码实现：

根据上面的梳理，有了具体的理论思路，我们就可以着手进行具体的代码实现。我在代码实现上，也分为如上述的两大部分，四小部分。在我的具

体代码实现中，也分别对应两大函数，四小函数。（四小函数包含两大函数）

在 main 函数中，分别对关联项集的求解就分为两大部分。具体为：

```
#计算所有频繁项集
L_all = utils.get_L_all(0.005,basket)
#生成关联规则
result = utils.rule_from_Lall(L_all,0.5)
```

图 2.2 main 函数

这里 result 就是一个字典列表，包含了最终所求的所有关联项集。结构如下：

[{'left': left, 'right': right, 'left|right': confidence}.....{...}]

第一部分 get_L_all 函数实现：

需要说明的是，我在计算频繁项集时，采用了循环求解策略。采用迭代的方式一次性把所有阶的频繁项集全部求解并统一放到一个字典中。具体使用了两个函数，第一个是候选项集想频繁项集的转化函数；第二个是将频繁项集转化为 k+1 阶候选项集的函数。

我认为在实现 **get_L_all** 函数有两点需要注意，第一点位于 **get_L_all** 函数本身，如下图所示，我采取了一个循环策略，通过 ck（候选项集）与 lk（频繁项集）的转化，每次迭代进行 update，最终得到全部 k 阶 lk。

```
def get_L_all(min_support,basket):
    """
    生成所有的频繁项集
    :param dataset:
    :param min_support:
    :return:
    """
    c1 = buildC1(basket)
    L1 = ck_to_lk(basket, c1, min_support)
    L_all = L1
    Lk = L1
    i = 1
    print(len(Lk))
    while len(Lk) > 1:
        lk_list = list(Lk.keys())

        ck = lk_to_ckset(lk_list)

        Lk = ck_to_lk(basket, ck, min_support)
        print(len(Lk))
        if len(Lk) > 0:
            L_all.update(Lk)
        else:
            break
        print(i)
        i += 1
    return L_all
```

图 2.3 模块一 **get_L_all** 函数实现

同时，**get_L_all** 函数还应注意其终止条件，分别对应（1）候选项集频率全部小于给定阈值；（2）k 大小等于商品总数。

第二点则是关于 lk_to_ck 函数也就是频繁项集转候选项集的函数，这里的实现需要注意每次只能找组合成 k+1 阶的候选项集，实现如下：

```
def lk_to_ckset(lk_list):
    """
    由频繁1项集组合形成频繁k+1项集
    :param lk_list:
    :return:
    """
    ckset = set()
    lk_size = len(lk_list)
    if lk_size > 1:  # 如果lk只有一个的话，说明无法再向上取候选集了
        k = len(lk_list[0])
        for i, j in itertools.combinations(range(lk_size), 2):
            t = lk_list[i] | lk_list[j]
            if len(t) == (k + 1):
                ckset.add(t)

    return ckset
```

图 2.4 lk_to_ck 实现

第二部分 rule_from_Lall 函数实现：

进过第一部分 **get_L_all** 函数对于数据的处理，我们可以得到 1～k 阶的频繁项集。有了这些频繁项集，寻找“相关”商品就相对简单一些。

首先我们要构造 rule_from_item 函数。

此函数作用就是选出 2 阶以上的频繁项集，对每一个集合进行重新的排列组合并将其分为两部分，注意要筛选出所有可能：比如给定一个商品 set{1,2,3}，在对其进行重组拆分后，返回：

[(frozenset({1}), frozenset({2, 3})),
(frozenset({2}), frozenset({1, 3})),
(frozenset({3}), frozenset({1, 2})),
(frozenset({1, 2}), frozenset({3})),
(frozenset({1, 3}), frozenset({2})),
(frozenset({2, 3}), frozenset({1}))]

也就是找出所有可能的分组。

- 有了 rule_from_item 函数，我们就可以依靠第一步求出的频繁项集，求出频繁项集内所有集合的所有可能的分组的关联置信度，具体计算如下：

$$confidence(x|Y) = P(x|Y) = \frac{support(x)}{support(Y)}$$

最终实现 rule_from_Lall 函数：返回 result 字典列表：

[{'left': left, 'right': right, 'left|right': confidence}.....{...}]

```python
# 生成关联规则
def rule_from_Lall(L_all, min_confidence):
    rules = []
    for Lk in L_all:
        if len(Lk) > 1:
            rules.extend(rule_from_item(Lk))

    result = []
    for left, right in rules:
        support = L_all[left | right]
        confidence = support / L_all[left]
        if confidence >= min_confidence:
            result.append({'left': left, 'right': right, 'left|right': confidence})

    return result
```

图 2.5 生成关联规则代码实现

字典列表，就是我们最终所求的结果。

第三部分进阶部分 PCY 实现：

PCY 算法的基本原理为：将不同的频繁项组成的有序对通过 hash 函数映射不同的桶里并计数，如果桶的支持度低于最小支持度，那么桶里的数据一定都不频繁，从而实现备选频繁项集的初次筛选，之后思想与普通算法一致，对筛选后的备选频繁项集计算支持度并舍弃低于 min_support 的部分。

值得注意的是，桶的数目 nBuckets 是一个超参，不同值对结果影响较大，生成的 bitmap 可以反映桶的数目是否合适。考虑极端情况，桶太满或太空效果都不好，如果所有桶的 bit 都为 1，相当于没有 item 被筛选出去，如果只有一个桶的 bit 为 1，也说明几乎所有 item 都映射到了这个桶。当 nBuckets=1000 时 bit vector ，此时 1 和 0 相对均衡，效果也比较好。 另外经过一些测试发现，PCY 算法效果还和数据规模有很大关系，对大规模数据来说 PCY 算法效果更好一些。

3.2.2 遇到的问题及解决方式

3.2.3 实验测试与结果分析

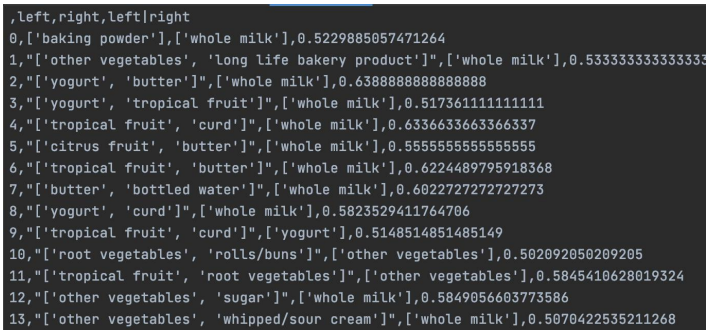

图 2.6 部分实验结果

做实验的过程中，我们知道一般集合是不允许重复的，所以一般会用到 set（int）操作，但是因为 set 类型的数据是可变的，所以无法作为字典的 key 值，而我们仍需要一个可以方便使用的 key 值 int 型 id，所以我就选用的 frozenset 类型的数据作为 id。frozenset 是冻结的集合，它是不可变的，存在哈希值，好处是它可以作为字典的 key，也可以作为其它集合的元素。缺点是一旦创建便不能更改，没有 add，remove 方法。

同时在实验的过程中经多次测试发现我的最后关联集合会比别人少四个，经过多次检查，发现最终提取商品集合时，比较置信度时需要包含等于 0.5 的集合。

```
        if confidence >= min_confidence:
            result.append({'left': left, 'right': right, 'left|right': confidence})
```

图 2.7 置信度比较

3.3 实验总结

本次试验我认为非常有趣，同时也有一定的难度，在课堂上曾经做过一次一样的关系挖掘的题，对于我在本次试验中的初步理解减轻了难度。我认为本次试验我学到的东西非常多，不仅仅是包含关系挖掘实验本身，也包含一些关于数据结构 frozenset，字典 key 值约束，PCY 算法等等。同时也锻炼了我的做项目的提前规划的能力。实验前首先要将大问题转换为小问题，分模块求解。这次试验令我获益良多。

实验四 kmeans 算法及其实现

4.1 实验目的

1、加深对聚类算法的理解,进一步认识聚类算法的实现；
2、分析 kmeans 流程,探究聚类算法院里；
3、掌握 kmeans 算法核心要点；
4、将 kmeans 算法运用于实际，并掌握其度量好坏方式。

4.2 实验内容

提供葡萄酒识别数据集，数据集已经被归一化。同学可以思考数据集为什么被归一化，如果没有被归一化，实验结果是怎么样的，以及为什么这样。

同时葡萄酒数据集中已经按照类别给出了 1、2、3 种葡萄酒数据，在 cvs 文件中的第一列标注了出来，大家可以将聚类好的数据与标的数据做对比。

编写 kmeans 算法，算法的输入是葡萄酒数据集，葡萄酒数据集一共 13 维数据，代表着葡萄酒的 13 维特征，请在欧式距离下对葡萄酒的所有数据进行聚类，聚类的数量 K 值为 3。

在本次实验中，最终评价 kmean 算法的精准度有两种，第一是葡萄酒数据集已经给出的三个聚类，和自己运行的三个聚类做准确度判断。第二个是计算所有数据点到各自质心距离的平方和。请各位同学在实验中计算出这两个值。

实验进阶部分：在聚类之后，任选两个维度，以三种不同的颜色对自己聚类的结果进行标注，最终以二维平面中点图的形式来展示三个质心和所有的样本点。效果展示图可如图 1.1 所示。

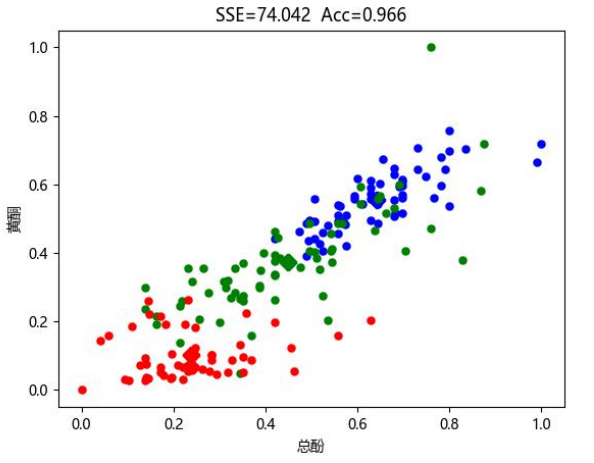

图 4.1 葡萄酒数据集在黄酮和总酚维度下聚类图像（SSE 为距离平方和，Acc 为准确率）

4.3 实验过程

4.3.1 编程思路

我认为该算法相对来说较为简单，根据老师所提供的归一化归一化数据，每个数据都有 13 个唯独作为物品的特征。

首先随机选择 n 个初始簇心，由于葡萄酒一共有 1、2、3 三个种类，所以这里 n 的值取 3，然后在各个种类中选取一个作为簇心，然后进行循环迭代。循环迭代过程中如下：

- 每次遍历所有节点，选取离其最近的簇心 label 作为其自身的 label，最后簇心要依据其簇内节点的个维度平均值重新调整。
- 根据新簇心重复动作 1
- 知道所有节点的 label 不再发生变化或者循环达到最大次数。

4.3.2 遇到的问题及解决方式

在实验的过程中，对于簇心的计算一开始总是出错，调试后发现，我申请的是一维列表，不能完成原先使用的 tmp[min_index] += data.loc[j]操作。这样子计算会将各个点到三个簇心的距离和算成一样的，最后导致新的簇心出错。结局方法就是简单的再在 tmp 上加个[]使之成为二维数组。

4.3.3 实验测试与结果分析

为了更清晰的观察各种纬度下的 label 分布，我随机选取了一些纬度绘制散点图进行观察。

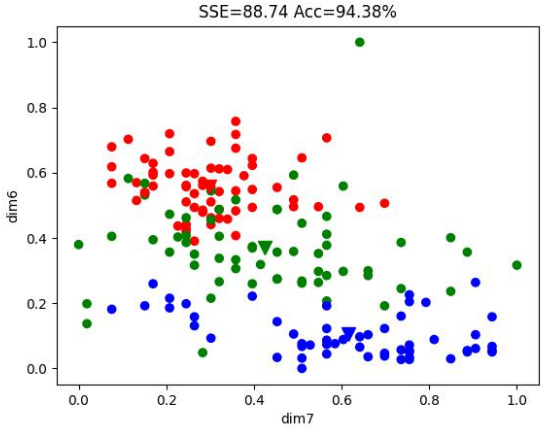

图 2.8 dim6 与 dim7

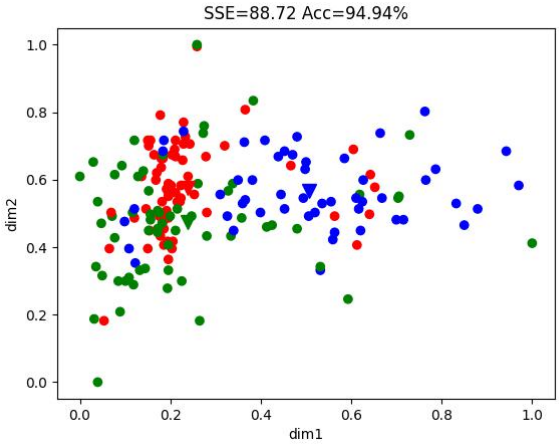

图 2.9 dim1 与 dim2

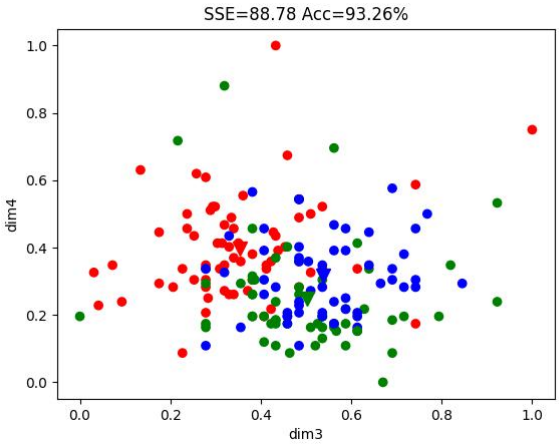

图 2.10 dim3 与 dim4

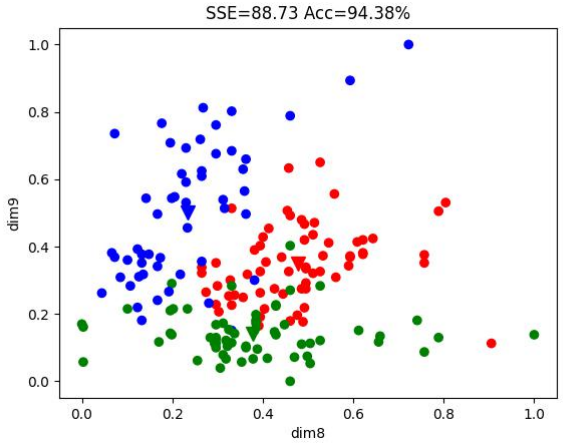

图 2.11 dim8 与 dim9

经过多次调试发现，dim6 与 dim7 对于节点的划分较为准确。

4.4 实验总结

我认为本次试验虽然较为简单，不过也让我学到了关于 k-means 算法的具体实现与思想，同时在调试代码的过程中，我也学习到了一些函数的使用约束。也意识到该算法在很多策略上都需要优化才能发挥出更好的分类效果，在此基础上进一步了解了 kmeans 衍生的很多优化版本，如 kmeans++、mini batch kmeans 等等，在聚类算法方面收获很大。

大作业 推荐系统任务书

5.1 实验目的

1、 了解推荐系统的多种推荐算法并理解其原理。
2、 实现 **User-User 的协同过滤算法**并对用户进行推荐。
3、 实现**基于内容的推荐算法**并对用户进行推荐。
4、 对两个算法进行电影预测评分对比
5、 在学有余力的情况下，**加入 minihash 算法**对效用矩阵**进行降维处理**

5.2 实验内容

给定 MovieLens 数据集，包含电影评分，电影标签等文件，其中电影评分文件分为训练集 train_set 和测试集 test_set 两部分

基础版必做一：**基于用户的协同过滤推荐算法**

对训练集中的评分数据构造用户-电影效用矩阵，使用 **pearson 相似度计算方法计算用户之间的相似度**，也即相似度矩阵。**对单个用户进行推荐时，找到与其最相似的 k 个用户，用这 k 个用户的评分情况对当前用户的所有未评分电影进行评分预测，选取评分最高的 n 个电影进行推荐。预测评分按照以下方式计算：**

$$\text{predict\_rating} = \frac{\sum_{i=1}^{k} rating(i) * sim(i)}{\sum_{i=1}^{k} sim(i)}$$

在测试集中包含 100 条用户-电影评分记录，用于计算推荐算法中预测评分的准确性，对测试集中的每个用户-电影需要**计算其预测评分，再和真实评分进行对比，误差计算使用 SSE 误差平方和**。

选做部分提示：此算法的进阶版采用 minihash 算法对效用矩阵进行降维处理，从而得到相似度矩阵，注意 minihash 采用 jarcard 方法计算相似度，需要对效用矩阵进行 01 处理，也即**将 0.5-2.5 的评分置为 0，3.0-5.0 的评分置为 1**。

基础版必做二：**基于内容的推荐算法**

将数据集 movies.csv 中的电影类别作为特征值，**计算这些特征值的 tf-idf 值**，得到关于电影与特征值的 n（电影个数）*m（特征值个数）的 **tf-idf 特征矩阵**。根据得到的 tf-idf 特征矩阵，**用余弦相似度的计算方法，得到电影之间的相似度矩阵**。

对某个用户-电影进行预测评分时，获取当前用户的已经完成的所有电影的

打分，通过电影相似度矩阵获得已打分电影与当前预测电影的相似度，按照下列方式进行打分计算：

$$\text{score} = \frac{\sum_{i=1}^{n} score'(i) * sim(i)}{\sum_{i=1}^{n} sim(i)}$$

选取相似度大于零的值进行计算，如果已打分电影与当前预测用户-电影相似度大于零，加入计算集合，否则丢弃。（相似度为负数的，强制设置为 0，表示无相关）假设计算集合中一共有 n 个电影，score 为我们预测的计算结果，score'(i)为计算集合中第 i 个电影的分数，sim(i)为第 i 个电影与当前用户-电影的相似度。如果 n 为零，则 score 为该用户所有已打分电影的平均值。

要求能够对指定的 userID 用户进行电影推荐，推荐电影为预测评分排名前 k 的电影。userID 与 k 值可以根据需求做更改。

推荐算法准确值的判断：**对给出的测试集中对应的用户-电影进行预测评分，输出每一条预测评分，并与真实评分进行对比，误差计算使用 SSE 误差平方和。**

选做部分提示：进阶版采用 minihash 算法对特征矩阵进行降维处理，从而得到相似度矩阵，注意 minihash 采用 jarcard 方法计算相似度，特征矩阵应为 01 矩阵。因此进阶版的特征矩阵选取采用方式为，**如果该电影存在某特征值，则特征值为 1，不存在则为 0，从而得到 01 特征矩阵。**

选做（进阶）部分：

本次大作业的进阶部分是在基础版本完成的基础上大家可以尝试做的部分。进阶部分的主要内容是使用**迷你哈希（MiniHash）算法对协同过滤算法和基于内容推荐算法的相似度计算进行降维。**同学可以把迷你哈希的模块作为一种近似度的计算方式。

协同过滤算法和基于内容推荐算法都会涉及到相似度的计算，迷你哈希算法在牺牲一定准确度的情况下对相似度进行计算，其能够有效的降低维数，尤其是对大规模稀疏 01 矩阵。同学们可以**使用哈希函数或者随机数映射来计算哈希签名。**哈希签名可以计算物品之间的相似度。

最终降维后的维数等于我们定义映射函数的数量，我们设置的映射函数越少，整体计算量就越少，但是准确率就越低。**大家可以分析不同映射函数数量下，最终结果的准确率有什么差别。**

对基于用户的协同过滤推荐算法和基于内容的推荐算法进行推荐效果对比和分析，选做的完成后再进行一次对比分析。

5.3 基础实验一实验过程

5.3.1 编程思路

1.在梳理编程思路之前，首先需要明确几个概念：

- **pearson 相似度：**

反应了两个变量之间的线性相关程度，它的取值在[-1, 1]之间。做完标化处理的 pearson 相似度计算如下：

$$P_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

- **jarcard 相似度：**jarcard 相似度同时也是使用 minihash 算法的入口，只有使用 jacard 相似度，理论上才能用 minihash 算法降维。其计算如下：

$$J(A,B) = |\frac{A \cap B}{A \cup B}|$$

- **迷你哈希 (MiniHash) 思想：**minihash 降维后在计算相似度必须要使用 jarcard 相似度。其将为过程为：

（1）将原先每个数据特征进行打乱，取第一个非零值的位置作为降维后的数据特征。

（2）根据所要降的维度重复对数据进行打乱，重复动作（1），最终得到降维后的数据。

因为 jarcard 相似度计算方式的原因，可以在 minihash 降维后的数据上得到一样的结果。

- **基于用户的协同过滤推荐算法：**

基于用户的含义是首先构造用户的特征，然后计算用户的相似度，根据相似度较高的其他用户的特征（这里的特征也就是看过的电影）来对本用户的某一个特征进行预测。计算如下：

$$\text{predict\_rating} = \frac{\sum_{i=1}^{k} rating(i) * sim(i)}{\sum_{i=1}^{k} sim(i)}$$

当使用 jarcard 相似度计算时因为涉及到交集与并集之间的计算, 所有首先要

对原先特征进行一个 0，1 转换，转变方式为：将 0.5-2.5 的评分置为 0，3.0-5.0 的评分置为 1。

然后根据 jarcard 相似度矩阵重新计算用户之间的相似度，根据相似度最高的 k 个用户推算本用户其他特征可能取值。

至于推荐电影系统，其实就是推算特征的一个延伸，将推算好的特征进行排序，选择数值最高的前 n 个最为可以向用户推荐的电影。（数值高代表着本用户可能的评分就越高）

- **基于内容的推荐算法：**

基于内容的推荐算法，就是要去比较电影之间的相似度而非用户之间的。其比较方式依旧是用的是 pearson 相似度与 jarcard 相似度。

但在推荐单影方面会与基于用的算法不同，向用户推荐电影的候选集打分方式如下：

$$\text{score} = \frac{\sum_{i=1}^{n} score'(i) * sim(i)}{\sum_{i=1}^{n} sim(i)}$$

这里的含义是根据用户已经打过分的电影，根据这些电影与其他所有未知电影的相似度与用户打得分进行加权求和。最终选取前 k 个电影进行推荐。

2.具体实现：

经过上面对概念的梳理，我们会发现大作业的实现起来其实并没有想象中的那么难，只不过非常的绕，来来回回有点让人晕。不过可以实现做好函数功能的构思，模块化实现。

- **实现基于用户的协同过滤推荐算法：**

  - 计算 pearson 相似度

```
19
20      sim_dict = {i: feature[userid].corr(feature[i], method='pearson') for i in range(1, 671 + 1)}
```

图 3.1 pearson 相似度实现

这里我使用的是 pandas 自带的 pearson 相似度计算库，根据 main 函数传过来的 UserID 来计算其他用户与他的 pearson 相似度。

  - 选取相似度最高的 topk 个用户

```
sorted_sim = sorted(sim_dict.items(), key = lambda kv:(kv[1], kv[0]), reverse=True)
topK_id = [sorted_sim[i][0] for i in range(K)]
```

图 3.2 topk 用户选择

  - 根据前 topk 个用户对于 User 没有看过的电影进行推算

```
up_sum = 0
down_sum = 0
mean = 0
for i in range(K):
    up_sum+=sim_dict[topK_id[i]]*feature[topK_id[i]][movieid]
    down_sum+=sim_dict[topK_id[i]]
    mean+=feature[topK_id[i]][movieid]
return mean/K#up_sum/down_sum"""
```

图 3.3 score 计算

这里计算依据：

$$\text{predict\_rating} = \frac{\sum_{i=1}^{k} rating(i) * sim(i)}{\sum_{i=1}^{k} sim(i)}$$

经过上述三步的计算就可以求出大多数电影用户评分的估值了。

- 选取前 n 个电影进行推荐

推荐电影则是根据上面一个计算出来的 score，选取前 n 个进行推荐。实现较为简单。

- jarcard 相似度实现

如代码所示，首先根据评分生成一个 0，1 矩阵。

```
nfuncs = 1000  # 映射函数数量
# 是minhash模式，需要额外生成一个对rating进行01化的矩阵
binary_data = df_train.copy(deep=True)
binary_data.loc[binary_data['rating'] < 2.6, 'rating'] = 0
binary_data.loc[binary_data['rating'] > 2.9, 'rating'] = 1
```

图 3.4 jarcard 相似度计算

- minihash 实现

如下图所示，实际上就是重复将原先每个数据特征进行打乱，取第一个非零值的位置作为降维后的数据特征。我这里使用利用 DataFrame 的 reindex 方法进行按行打乱，比手动调整顺序要快，打乱后生成的新 DataFrame 之间从上到下遍历，集合 s 用来判断对应某一种 permutation 来说 matrix 中的每一列是否找到了第一个“1”，一开始里面有 users_num 个元素，如果都找到了集合 s 变为空，就提前结束遍历。

```
for i in range(nfuncs):
    func = list(range(1, movies_num + 1))
    np.random.shuffle(func)  # permutation π
    shuffled_matrix = feature.reindex(func)
    s = set(range(users_num))  # 记录对于每个func，user是否找到第一个

    sig_i = np.zeros(users_num)
    for j in range(movies_num):
        row = np.array(shuffled_matrix.iloc[j])
        for r in range(users_num):
            if row[r] and r in s:
                s.remove(r)
                sig_i[r] = j + 1
        if not s:
            break

    sig_matrix[i] = sig_i  # 更新签名矩阵的第i行

sig_matrix = pd.DataFrame(sig_matrix)
sig_matrix.columns = list(range(1, users_num + 1))
```

图 3.5 minihash 实现

- **实现基于内容的推荐算法**
  - pearson 相似度计算

    首先使用 TfidfVectorizer, CountVectorizer 两个函数生成 tf-idf 特征矩阵

```
cosine_matrix = cosine_similarity(matrix)
return cosine_matrix
```

图 3.6 Pearson 相似度矩阵

  - 计算得分

    实现公式:

$$\text{score} = \frac{\sum_{i=1}^{n} score'(i) * sim(i)}{\sum_{i=1}^{n} sim(i)}$$

```
distances = cosine_matrix[movieid]  # 从movieid出发的距离向量
computed_dict = {}  # 计算集合
for i in range(len(rated_movies)):#对已经评过分的电影中找与movieID相似的
    rated_movie = rated_movies[i]
    cosine = distances[Id_index[rated_movie]]
    if cosine > 1e-6:#收集与电影movieid相似度高的电影
        computed_dict[i] = cosine#某一部评过分的电影k，与movieID相似度cosine

# 计算集合不为空
if len(computed_dict.keys()):
    score = 0
    sum_v0 = 0
    for k, v in computed_dict.items():
        score += rating[k] * v#根据与movieID相似的用户评分过的电影，加权求和，这里的k与v，都是用户真实的
        sum_v0 += v

    return score / sum_v0#求相似度高的电影的评分加权和

# 计算集合为空
else:
    return np.mean(rating)
```

图 3.7 score 计算

因为事先已经得出相似度矩阵，所以我们只需要取得 movieID 所对应的那一行，然后根据用户已经评过分的电影，根据相似度，去筛选用户评分的电影中与 moiveID 相似的电影，保存于 computed_dict[i]中。

```
for i in range(len(rated_movies)):#对已经评过分的电影中找与movieID相似的
    rated_movie = rated_movies[i]
    cosine = distances[Id_index[rated_movie]]
    if cosine > 1e-6:#收集与电影movieid相似度高的电影
        computed_dict[i] = cosine#某一部评过分的电影k，与movieID相似度cosine
```

图 3.8 筛选相似电影

之后计算 computed_dict 的加权和即可。

```
# 计算集合不为空
if len(computed_dict.keys()):
    score = 0
    sum_v0 = 0
    for k, v in computed_dict.items():
        score += rating[k] * v#根据与movieID相似的用
        sum_v0 += v

    return score / sum_v0#求相似度高的电影的评分加权和
```

图 3.9 score 的计算，用到已评分电影与其对 moiveID 相似度

若使用 minihash 做降维处理，首先则将重新制作一份 jarcard 相似度的相似矩阵，在进行数据将为，处理过程与基本实验一类似。最终也是通过相似度去计算得分。

5.3.2 遇到的问题以及解决方式

本次试验没有遇到较大问题，但我认为我还有一些地方做的不是很好。比如说基本实验一的 moiveID 与 User 特征维度（movie 类别）之间的关系。为曾计算过 movieID 的最大值远高于 movie 类别数量，但我为了实验简单仍然使用了 movieID 的最大值作为用户特征维度。我认为可以加上一个映射函数，降低用户维度减少计算时间。

在基本实验二中我也面临着同样的问题，就是计算量过大，主要在 minhash 中生成签名矩阵这一部分，我认为可以不用将整个签名矩阵全部计算出来，可以在传入 moiveID 时仅计算与之相关的相似度。

5.3.3 实验测试结果与结果分析

- topk 推荐与预估评分

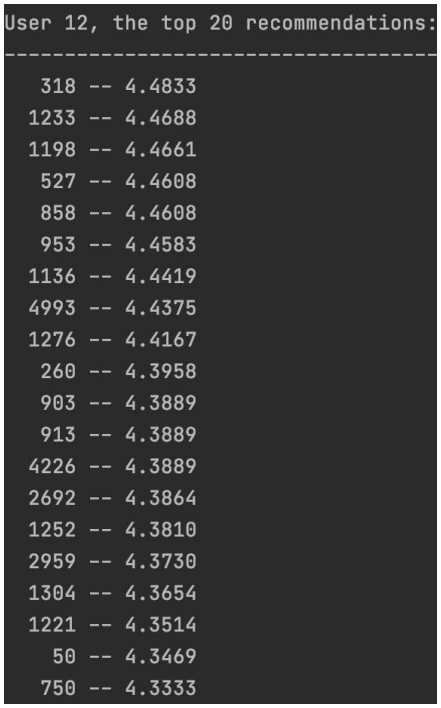

图 3.10 随机选取 ID12 用户推荐前 20 个电影

根据与同学交流，我发现在实验过程中，考虑到最终评分是以最相似的 k 个用户的平均预测来当作该用户的预测，那么如果这 k 个用户只有极少数人看了某一部电影，还都给了五星好评，那这样的推荐显然是不合理的，因为没有考虑到样本数的影响。

而基于内容的推荐相对来说效果好一些，一方面因为它通过加权均衡了一些极高评分的影响（因为大多数电影都和该电影或多或少有相似），另一方面训练

集的数据给的也比较好，没有新用户出现。

所以我在实验中加入了一些新的处理：

```
if len(x[x != 0]) > 10:#大于10人才计算此电影
    pred_i = np.mean(x[x != 0])
    pred_dict[i] = 0 if np.isnan(pred_i) else pred_i#给出此电影评分/推荐分数
```

图 3.11 去掉少数与极端的评分

虽说最后验证结果并不太稳定，不过我认为可能是我们数据集过少的缘故，如果给予足够大的数据集，我认为这种筛除操作还是有益的。

- minihash 降低维度选择

在实验完成后，我仍对 minihash 的取值较感兴趣，我对 minihash 的取值做了一些小实验。

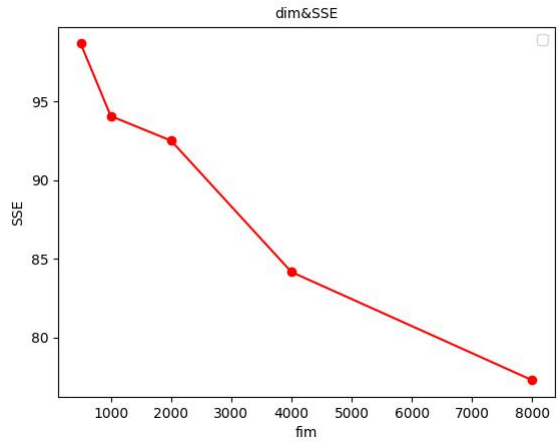

图 3.12 基于用户推荐 SSE 与降低维度的关系

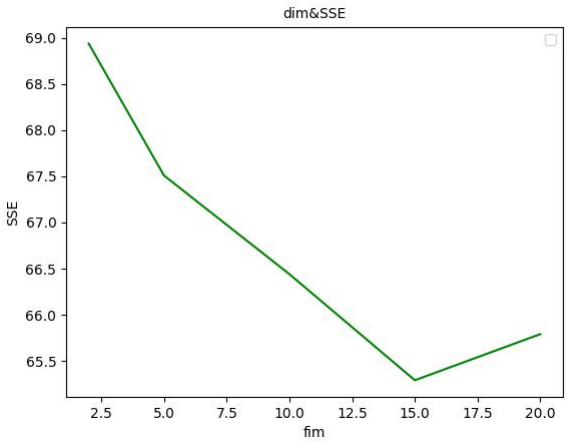

图 3.13 基于内容 SSE 与降低维度关系

1、经试验观察得知，在该测试集上，Content-based 推荐效果普遍优于 User-User 协同过滤推荐。

2、对于基于内容推荐算法来说，数据维度很大，加入 minhash 后效果明显变差，说明了 minhash 降维通过牺牲准确率提高效率的特点；而对于基于内容推荐算

法来说，出现拐点的原因一方面可能是数据随机性导致，另一方面可能同为 jarcard 相似度使用 minihash 将弱特征转化为强特征导致。

在完成基本实验一与基本实验二之后，我也对我之前的疑惑有了一个较为清晰的认知，即基于内容和基于用户算法谁更好，本次试验看似是基于内容更加精确，但这种精确是建立在电影数量远大于用户数量的基础上实现的，甚至说我认为在这种情况下基于用户之间的算法实现根本没有意义，因为差距太大了。即便基于内容算法经过降维处理，精确度仍比基于用的算法高。

5.4 实验总结

首先非常感谢老师提供这样一个具有挑战性的实验，我认为在本次试验中我收获良多，一方面在实验过程中充分了解了基于内容的推荐算法与基于用户的推荐算法。另一方面在完成进阶实验中也掌握了很多概念比如 jarcard 相似度与 minihash 算法之间的联系与证明，SSE 评估指标，pearson 相似度标化处理前后的关系与差别。虽然说本次试验主要内容依旧是硬性的计算，不过却有很多东西值得思考，

比如说 topk 用户与 score 前 n 个电影的选择，考虑到计算公式，k 的大小究竟和什么有关系。再比如说进阶实验中 minihash 降维处理，酒精降低到一个什么样的维度才能较好的满足题目需要，我们知道 minihash 算是一个牺牲准确性减少计算量的（或者说降维处理都是这样）步骤，那么我们实验究竟需要多少准确度，能接受多少计算时间呢。

同时在完成基本实验一与基本实验二之后，我也对我之前的疑惑有了一个较为清晰的认知，即基于内容和基于用户算法谁更好，本次试验看似是基于内容更加精确，但这种精确是建立在电影数量远大于用户数量的基础上实现的，甚至说我认为在这种情况下基于用户之间的算法实现根本没有意义，因为差距太大了。即便基于内容算法经过降维处理，精确度仍比基于用的算法高。

同时本实验还面临着计算量过大，计算时间过长的问题，我认为可以适当的减少计算量（对于基于内容算法而言）比如仅计算传入 moiveID 与其他 moives 的相似度，或者使用实验一的 mapreduce 方法并行处理，因为相似度数据之间也没有相互制约，可以并行运行。