華中科技大学

课 程 实 验 报 告

课程名称： 大数据分析

专业班级： **CS2011**

学　　号： **U202014774**

姓　　名： 王逸

指导教师： 崔金华

报告日期： **2022.12.31**

计算机科学与技术学院

目录

实验五 推荐系统算法及其实现

1.1 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 minhash 算法**对效用矩阵**进行降维处理**

1.2 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（ratings.csv）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。**

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 minhash 算法对效用矩阵进行降维处理，从而得到相似度矩阵，注意 minhash 采用 jaccard 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 minhash 算法对特征矩阵进行降维处理，从而得到相似度矩阵，注意 minhash 采用 jaccard 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**最小哈希（minhash）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。**同学可以把最小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，最小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。**

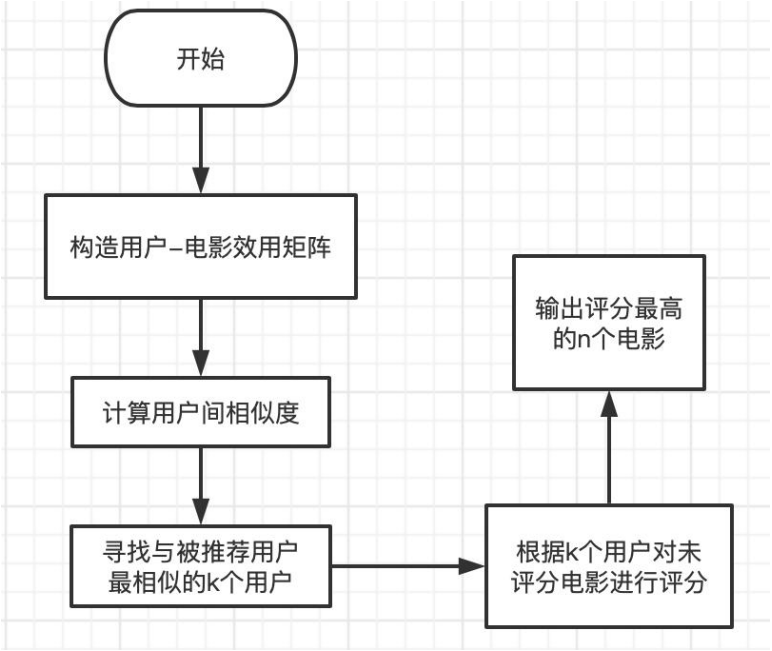
对基于用户的协同过滤推荐算法和基于内容的推荐算法进行推荐效果对比和分析，选做的完成后再进行一次对比分析。

1.3 实验过程

1.3.1 编程思路

1. 基于用户的协同过滤推荐算法

   该算法的总体实现流程如下：



   大体分为以下几个部分：

   A. 构造效用矩阵：

      a) read_movie_data（movie_file）：返回每个电影 id 的类别字典，每个电影的名字字典。

      b) reading_rating_data（rating_file）：返回用户对电影的评分列表，形如（user_id，movie_id，score）。

   B. 计算用户间相似度：基础版采用 pearson 相似度计算用户间相似度，进阶使用 mini-hash 计算。

      a) pearson 相似：用户 x 与用户 y 之间的相似度计算公式如下：xi 表示用户 x 对电影 i 的评分，yi 表示用户 y 对电影 i 的评分，n 为电影总数。

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

   代码实现部分截图如下：（为了加快运行计算速度，最终实现部分是调用了 np.corrcoef（）函数）

```python
def pearson(user1, user2):
    s1 = 0.0
    s2 = 0.0
    s_12 = 0.0
    avg1 = 0.0
    avg2 = 0.0

    for m_rate in user1:
        avg1 += m_rate[1]
    avg1 = avg1 / len(user1)
    for m_rate in user2:
        avg2 += m_rate[1]
    avg2 = avg2 / len(user2)

    for m_r1 in user1:
        for m_r2 in user2:
            # 评价相同电影
            if m_r1[0] == m_r2[0]:
                s_12 += (m_r1[1] - avg1) * (m_r2[1] - avg2)
                s2 += (m_r2[1] - avg2) * (m_r2[1] - avg2)
            s1 += (m_r1[1] - avg1) * (m_r1[1] - avg1)

    if s_12 == 0.0:
        return 0
    sx_sy = math.sqrt(s1 * s2)
    return s_12 / sx_sy
```

b) mini-hash：首先需要将传进来的 user_movie 矩阵化为 01 矩阵，以用户 id 为行，电影 id 为列，若用户 i 对电影 j 评分在[0.5, 2.5]则对应为 0，3.0-5.0 对应为 1.再调用 minihash 函数获得签名矩阵。

i. mini_hash 函数：因为最小哈希需要对行号进行哈希处理，所以需要多个哈希函数进行处理，这里采用随机产生哈希函数系数。首先初始化一个以哈希函数数目为行数，用户数为列数的签名矩阵 sig，并将每一个值初始化为一个较大值 inf。然后开始按行遍历 movie_user 数组，按照如下步骤处理，最后返回签名矩阵 sig。

- 根据哈希算出每行的h1(i),...,hn(i).(i=1...N)
- For i = 1:N
- For c = 1:M
- 如果S[i,c]为0；跳过
- 否则；对于每个排列r=1...n, K(r,c) = min (K(r,c), h_r(i))

ii. calculate_jaccard(user1, user2)：计算签名矩阵元素（列）间的相似度。对于向量 user1 和 user2，计算两个向量对应元素相等的个

数，并除以总长度得到相似度。

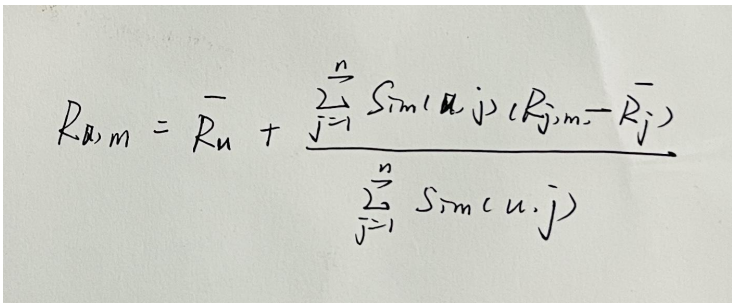
C. 寻找最相似 k 个用户

a) get_near_k （user_id, user_rate, movie_user, user_user, k)：寻找与给定用户 user_id 最为相似的 k 个用户。首先获取该用户的评分过的电影，对于每个给该电影评分过的用户，加入邻居列表中，计算该用户与 user_id 的相似度，以[user, dist]的形式加入 nei_dist 列表中。最后对 nei_dist 列表进行按 dist 从高到低排序，返回前 k 个用户与相似度。

```
def get_near_k(user_id, user_rate, movie_user, sig, k):
    neighbors = []
    nei_dist = []
    for m_rate in user_rate[user_id]:
        # 对于每个给该电影评分的用户
        for m_user in movie_user[m_rate[0]]:
            if m_user != user_id and m_user not in neighbors:
                neighbors.append(m_user)
                dist = calculate_jaccard(sig[:, user_id-1], sig[:, m_user-1])
                # dist = user_user[user_id-1, m_user-1]
                nei_dist.append([m_user, dist])
    nei_dist.sort(reverse=True)
    return nei_dist[:k]
```

D. 预测评分：预测 user_id 未评分电影的评分。

a) Predict_score()：预测评分由两部分组成：该用户所有已评分电影的平均打分；基于 k 个相似用户的补充打分，补充打分是该相似用户集中每个用户对于当前电影的补充打分的一个加权平均，权重依赖于用户的相似度。计算公式如下：

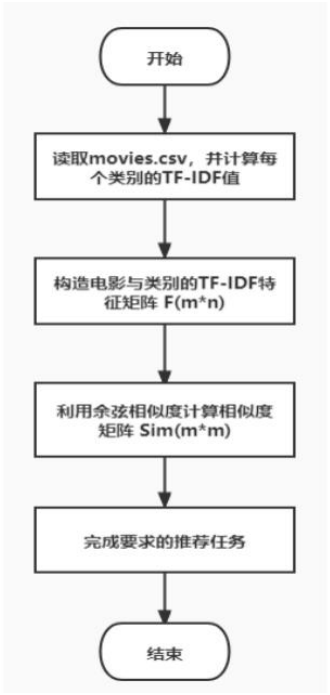


E. 输出推荐电影：

a) Recommend()：步骤与 predict_score （）类似，不过该函数传入参数并没有 movie_id，需要遍历用户所有未评分电影并进行预测打分，返回一个给该用户的推荐列表。

2. 基于内容的推荐算法

该算法的总体实现流程如下：



大体分为以下几个部分：

A. 获得电影信息：

a) read_movie_info（）：读取 movie_file 获得电影信息字典 movie_info（[id, title, movie_genre])和电影 id 列表 movie_ids.

b) read_rating_data（）：获取用户-电影评分信息。

B. 计算 tf_idf 特征矩阵：tf、idf、tf_idf 矩阵均为电影 id-种类矩阵。

a) Calculate_TF_IDF()：若某电影 i 拥有种类 g 的标签，则将 tf 和 idf 对应的值标为 1.tf 为词频矩阵，idf 为反文档频率，计算公式如下：

```
# tf 词频
for i in range(movie_num):
    sum_r = sum(tf[i, :])
    for j in range(genre_num):
        if tf[i, j]:
            tf[i, j] /= sum_r

# idf 反文档频率
for i in range(genre_num):
    sum_c = sum(idf[:, i])
    for j in range(movie_num):
        if idf[j, i]:
            idf[j, i] = math.log(movie_num / (sum_c + 1))

for i in range(movie_num):
    for j in range(genre_num):
        tf_idf[i, j] = tf[i, j] * idf[i, j]
```

最后返回对应元素相乘得到的 tf_idf 矩阵。

C. 相似度计算：

a) 余弦相似度：按照余弦相似度的计算公式实现的代码部分如下：

```
def calculate_cos_sim(tf_idf):
    n = len(tf_idf)
    for i in range(n):
        a = np.dot(tf_idf[i], tf_idf[i])
        if a!= 0:
            tf_idf[i] = tf_idf[i] / math.sqrt(a)
    sim = np.dot(tf_idf, tf_idf.T)
    return sim
```

b) Minihash：总体步骤与基于用户的协同过滤算法部分的最小哈希算法相同，其中本部分的 01 矩阵是电影 id-种类 genre 矩阵，若电影 i 含有标签 j，则 movie_genre[i, j] = 1。再调用最小哈希得到签名矩阵。使用 jaccard 方法计算相似度。

D. 预测评分：计算公式如下：

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

实现代码如下：

```
def predict_score(user_rate, user_id, movie_id, sig, m_id):
    sig = sig.T
    rated = user_rate[user_id]
    u_rated_num = len(rated)
    col = movie_id.index(m_id)

    s1 = 0
    s2 = 0
    s3 = 0
    for movie in rated:
        row = movie_id.index(movie[0])
        sim = calculate_jaccard(sig[row], sig[col])
        if sim > 0:
            if movie_id[col] in rated:
                continue
            else:
                s1 += sim * movie[1]
                s2 += sim
                s3 += movie[1]
        else:
            continue
    if s2 == 0:
        pre_score = s3 / u_rated_num
    else:
        pre_score = s1 / s2
    return pre_score
```

(1) 推荐任务：该部分实现逻辑与预测评分部分相同，区别在于预测评分给

定电影 id，而推荐部分只给出了用户 id，需要遍历完成所有电影的评分。

1.3.2 遇到的问题及解决方式

在实验过程中遇到以下几个问题：

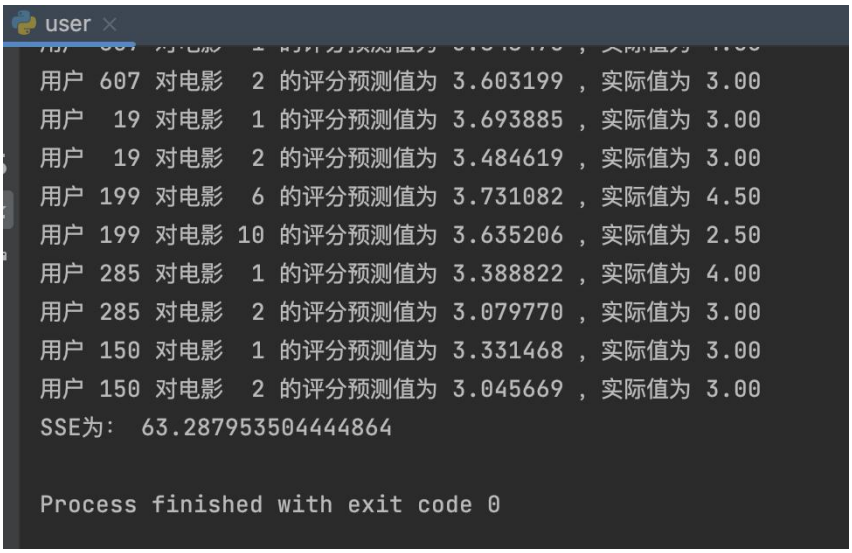
1. 使用自己定义的 pearson 与 cosine_sim 函数计算相似度时间较长，后采用调用库函数加快运行速度。
2. 基于用户的协同过滤算大最小哈希实现，由于电影-用户数组较大处理起来需要较长时间，后面询问助教后发现是正常现象。

1.3.3 实验测试与结果分析

1. 基于用户的协同过滤算法：

(1) Pearson：

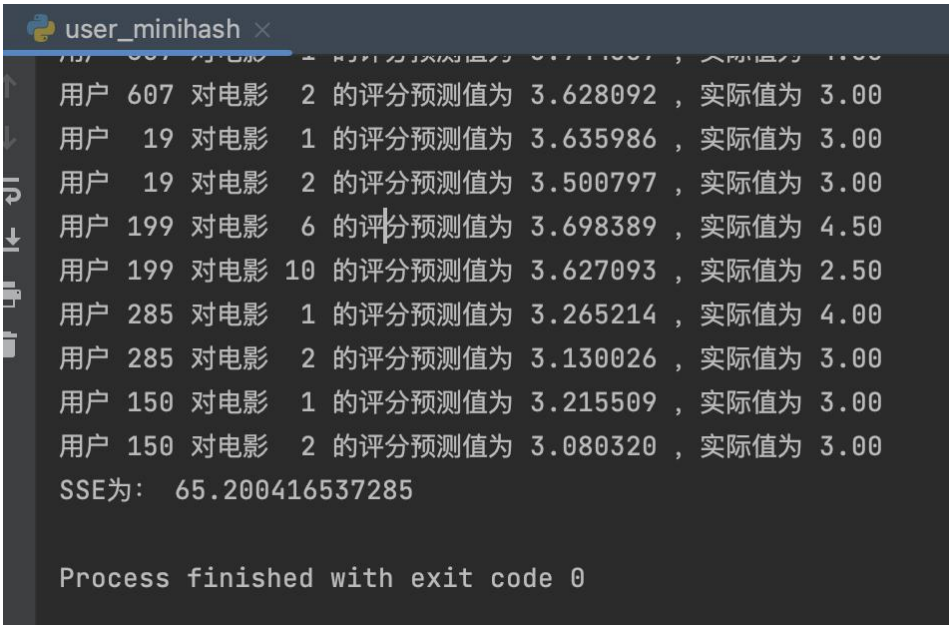
① sse 计算输出结果：



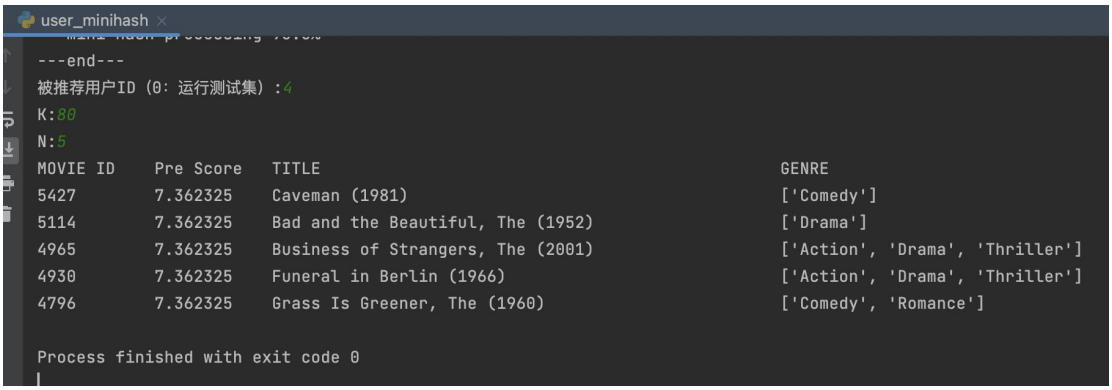
② 推荐结果：

(2) 最小哈希：

① sse 计算结果：



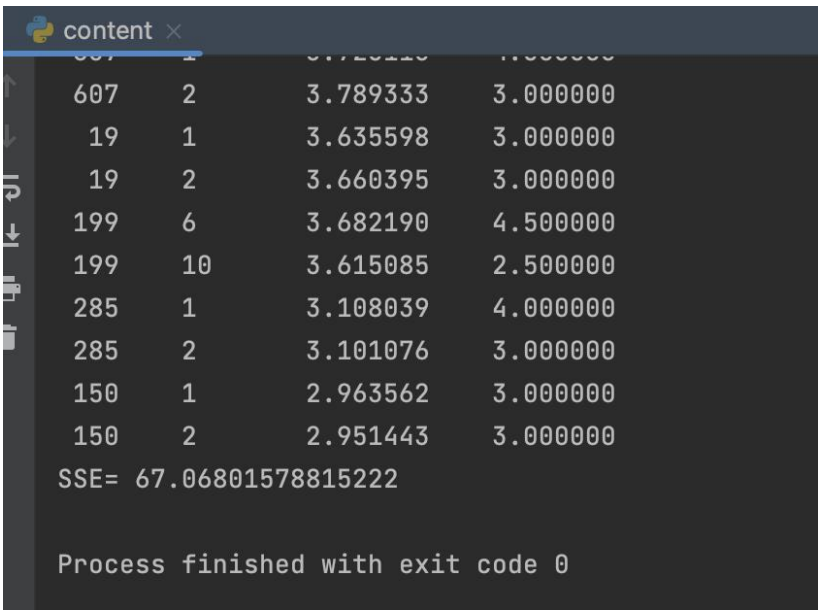
② 推荐结果:



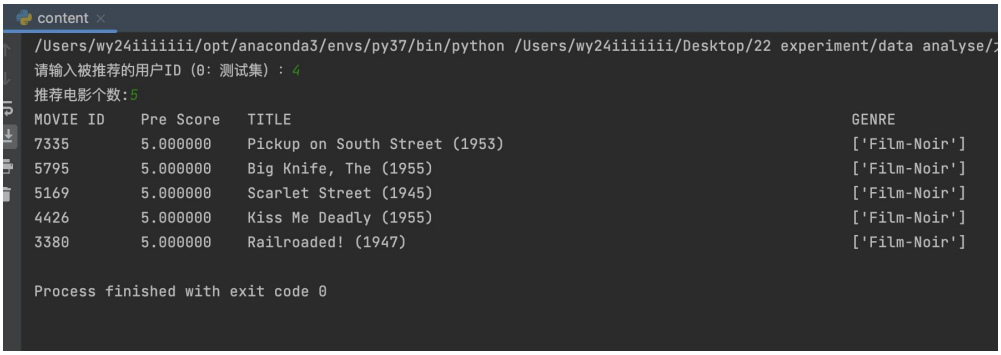
2. 基于内容的推荐算法:

(1) 余弦相似度:

① Sse:

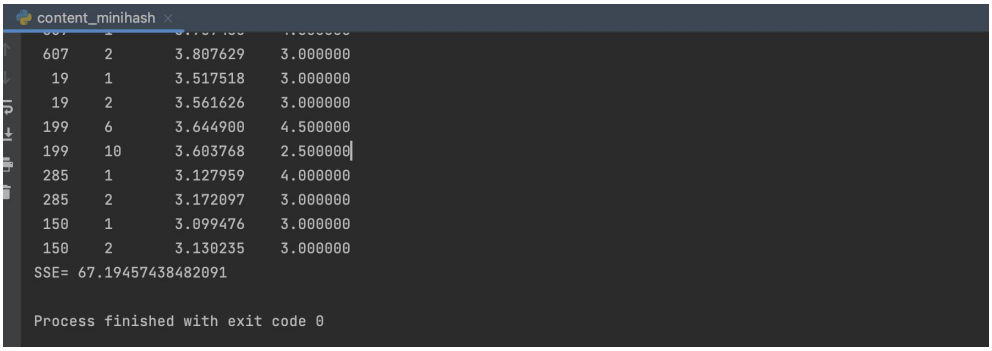


② 推荐结果：



(2) 最小哈希：

① Sse：



② 推荐结果：

1.4 实验总结

本次实验中除了最小哈希的实现以外遇到的最大问题还是选择使用什么类型存储数据，选择合适的数据类型可以加快查找处理速度，所以一开始在这上面耗了一段时间，后面也由于类型转换在编译过程中除了一点小问题。总体来说，本次综合实验难度适中，把各模块划分好之后写起来还是比较顺利的。