

**课 程 实 验 报 告**

**课程名称： 大数据分析**

**专业班级： 校交1902**

**学 号： U201912633**

**姓 名： 张睿**

**指导教师： 王蔚**

**报告日期： 2022.1.4**

**计算机科学与技术学院**

**目录**

[实验一 wordCount算法及其实现 1](#_Toc92357376)

[**1.1实验目的** 1](#_Toc92357377)

[**1.2 实验内容** 1](#_Toc92357378)

[**1.3 实验过程** 1](#_Toc92357379)

[**1.3.1 编程思路** 1](#_Toc92357380)

[**1.3.2 遇到的问题及解决方式** 2](#_Toc92357381)

[**1.3.3 实验测试与结果分析** 3](#_Toc92357382)

[**1.4 实验总结** 3](#_Toc92357383)

[实验二 PageRank算法及其实现 4](#_Toc92357384)

[**2.1实验目的** 4](#_Toc92357385)

[**2.2 实验内容** 4](#_Toc92357386)

[**2.3 实验过程** 4](#_Toc92357387)

[**2.3.1 编程思路** 4](#_Toc92357388)

[**2.3.2 遇到的问题及解决方式** 5](#_Toc92357389)

[**2.3.3 实验测试与结果分析** 5](#_Toc92357390)

[**2.4 实验总结** 6](#_Toc92357391)

[实验三 关系挖掘实验 7](#_Toc92357392)

[**3.1 实验内容** 7](#_Toc92357393)

[**3.2 实验过程** 7](#_Toc92357394)

[**3.2.1 编程思路** 7](#_Toc92357395)

[**3.2.2 遇到的问题及解决方式** 8](#_Toc92357396)

[**3.2.3 实验测试与结果分析** 8](#_Toc92357397)

[**3.3 实验总结** 9](#_Toc92357398)

[实验四 kmeans算法及其实现 10](#_Toc92357399)

[**4.1实验目的** 10](#_Toc92357400)

[**4.2 实验内容** 10](#_Toc92357401)

[**4.3 实验过程** 11](#_Toc92357402)

[**4.3.1 编程思路** 11](#_Toc92357403)

[**4.3.2 实验测试与结果分析** 11](#_Toc92357404)

[**4.4 实验总结** 12](#_Toc92357405)

[实验五 推荐系统算法及其实现 13](#_Toc92357406)

[**5.1实验目的** 13](#_Toc92357407)

[**5.2 实验内容** 13](#_Toc92357408)

[**5.3 实验过程** 15](#_Toc92357409)

[**5.3.1 编程思路** 15](#_Toc92357410)

[**5.3.2 实验测试与结果分析** 21](#_Toc92357411)

[**5.4 实验总结** 24](#_Toc92357412)

# 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。

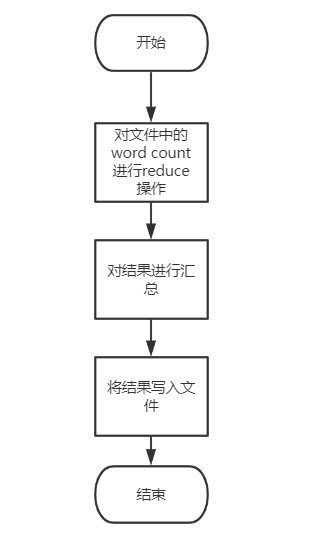
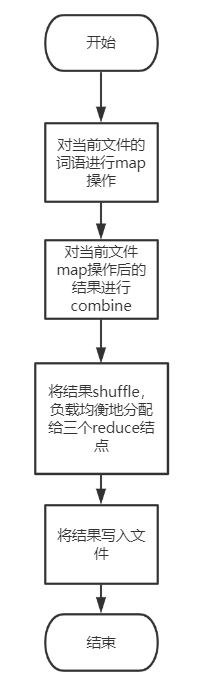
提示：实现shuffle过程时应保证每个reduce节点的工作量尽量相当，来减少整体运行时间。

**1.3 实验过程**

**1.3.1 编程思路**

根据MapReduce的思想，将流程分为Map和Reduce两个阶段，分别建立两个线程类MapThread和ReduceThread，重写run方法。

对于MapThread，其处理流程如下：



对于Map，即为一个简单地分词+建立字典的操作，combine即为将单个文件中的word\_count进行一个reduce的操作，对于shuffle，这里采取了两种策略：

1. 根据首字母分配给三个reduce结点

2. 首先根据word对字典排序，接着均匀分配给三个reduce结点

对于第一种策略，可以将三个reduce的结点的统计互不重叠，最大限度减少reduce的工作量，但是存在难以决定如何分配的问题，因为单词依据首字母的分布是不均匀的。

对于第二种策略，可以使得三个reduce结点工作量相同，但是各个reduce结点间的词语会有所重叠，并没有显著减少总工作量。

对于Reduce，也是非常简单的遍历，然后对word对应的count值进行增加。

**1.3.2 遇到的问题及解决方式**

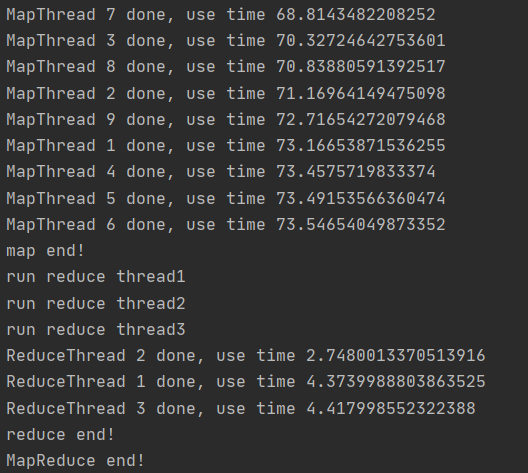
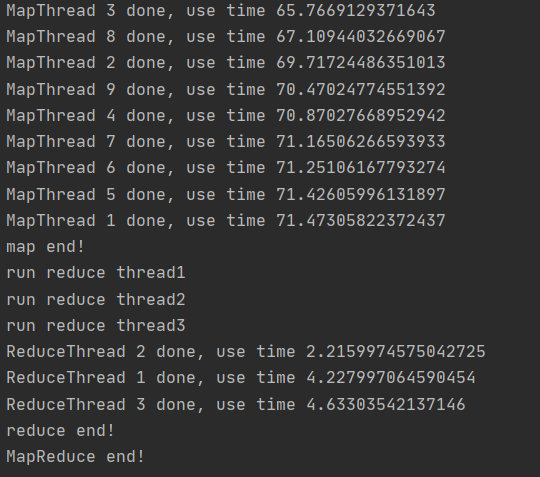
1. 结果错误

第一次跑出来的结果非常错误，属于是一看就错的一种，每一个词的count都有几十万。

解决：通过printf“调试”，以及仔细审阅代码，发现是从中间文件读出来count值没有转换成int类型，还是str类型，导致就结果非常之大

**1.3.3 实验测试与结果分析**

主要是测试了一下两种策略的时间。



可以发现，这两种策略在时间上并没有太大的区别，猜测是因为数据集中，每个文件的单词按照首字母的分布较为类似（注意，并不是说他们是均匀分布），这也是比较好理解的，因为每个文件都是词语库的一个子集，由于本身词语库很大，可以近似认为每个文件都是无偏采样，每个文件中的词语按照首字母的分布相近，所以当排完序后划分，会使得三个reduce结点之间重合度较小。

**1.4 实验总结**

本次实验有以下收获：

1. 深入理解并学会如何实现MapReduce

2. 学会使用python的多线程

# 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**

由于之前的相关经历，对networkx库较为熟悉，所以选择采用networkx库来完成实验。

1. 建图

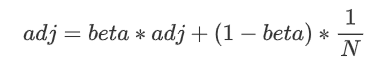
首先利用networkx建一个有向图，并从数据集读入数据。

2. 重命名边

由于原本的数据集person id并不是连续，所以需要简历一个映射关系，使得其映射成连续的id。

3. 获得转移矩阵

首先根据networkx自带的方法获得邻接矩阵，再根据beta值构造转移矩阵，公式为：



4. 利用公式进行迭代



利用该公式即可进行迭代，并利用两次迭代的误差，判断收敛，获得结果

**2.3.2 遇到的问题及解决方式**

1. 矩阵的转置问题

由包获得的邻接矩阵为Mij，是行对应的，需要进行转置获得列对应。

2. 转移矩阵的构造。

由于对公式的错误理解，以为每次迭代都需要进行beta的修正，但其实只需要对转移矩阵修饰一次即可。

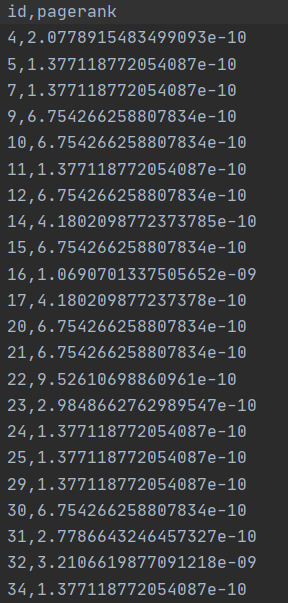
3. 误差的衡量

最开始以为是对误差进行L2范数的统计，结果是对无穷范数进行统计，即求每一维误差的最大值作为最终误差。

4. 归一化问题

自己写的初始版本每次迭代都会对rank值做一个归一化处理，为了方便助教检查，需要去掉归一化操作。

**2.3.3 实验测试与结果分析**



可以看到rank值是非常小的，一方面是因为没有做归一化操作，另一方面是因为边的权重默认为1，所以1在进行转移，自然会比较小。

**2.4 实验总结**

这次实验比较简单，主要是对pagerank算法的理解，包括转移矩阵的构建的迭代的方法，我个人觉得最重要的有以下几点：

1. 理解了矩阵乘法是如何实现信息（rank值）的传播，还好线代没有忘记，通过这次实验，让我明白了“转移”一词的真正含义。

2. 通过实验进一步加深了对利用teleport解决dead end 和 spider trap问题的认识。

**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**

关系挖掘算法主要有如下步骤：根据支持度找出各阶频繁项集，根据置信度从各阶频繁项集中找出关联规则。接下来分别叙述。

对于频繁项集的生成，一共需要生成4阶的频繁项集，根据其规则，由于二阶频繁项的每一个数据一定是一阶频繁项，以此类推，所以可以用1阶生成2阶，2阶生成3阶，如此类推来避免对某些项的不必要计算。这里又可以细分为两步，首先根据前一阶频繁项生成下一阶待选频繁项，再算待选频繁项的支持度并剔除小于支持度的项获得当前阶的频繁项。

对于PCY算法对于频繁项集生成的加速思路为，对于上一阶，将其排列组合成多个二阶待选频繁项，将其哈希进哈希桶中，并用bitmap来标识当前桶是否为频繁桶，采用如下策略对待选频繁项进行筛选：

非频繁桶中的项一定为非频繁项；

在频繁桶中可以进入下一轮筛选，即算其支持度；

其之所可以加速，是因为计算支持度是一个比较费时的操作，而且排列组合的结果非常多导致空间浪费，所以希望能尽可能减少需要进行支持度计算的项。

针对于编程来说，需要建立双方的映射，一方面既要将项映射到哈希桶中，又需要建立桶和原来项的映射关系，以便通过桶找回相应的项。

对于关联规则生成，则是需要针对所有频繁项，通过每次抽出部分元素来计算条件概率，即置信度，来判断当前规则是否成立，这里需要从取一个开始，这样可以方便进行剪枝，因为如果A,B,C🡪D,不成立的话A,B🡪C,D也一定不成立，通过这种trick，可以大大提升计算速度。

**3.2.2 遇到的问题及解决方式**

1. 占用空间过大，计算缓慢

这个问题是关联关系挖掘的核心问题，因为其实算法思想非常简单，无非是算一个条件概率，为解决这个问题，从以下方面进行改进：

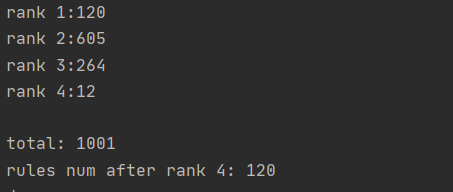
- 将物品数值化，不用物品本来的字符串，而是用一个id来进行接下来的计算

- 在计算过程中时刻进行排序，利用排序可以减少很多非必要的计算，比如123🡪4,其实和132🡪4是一样的

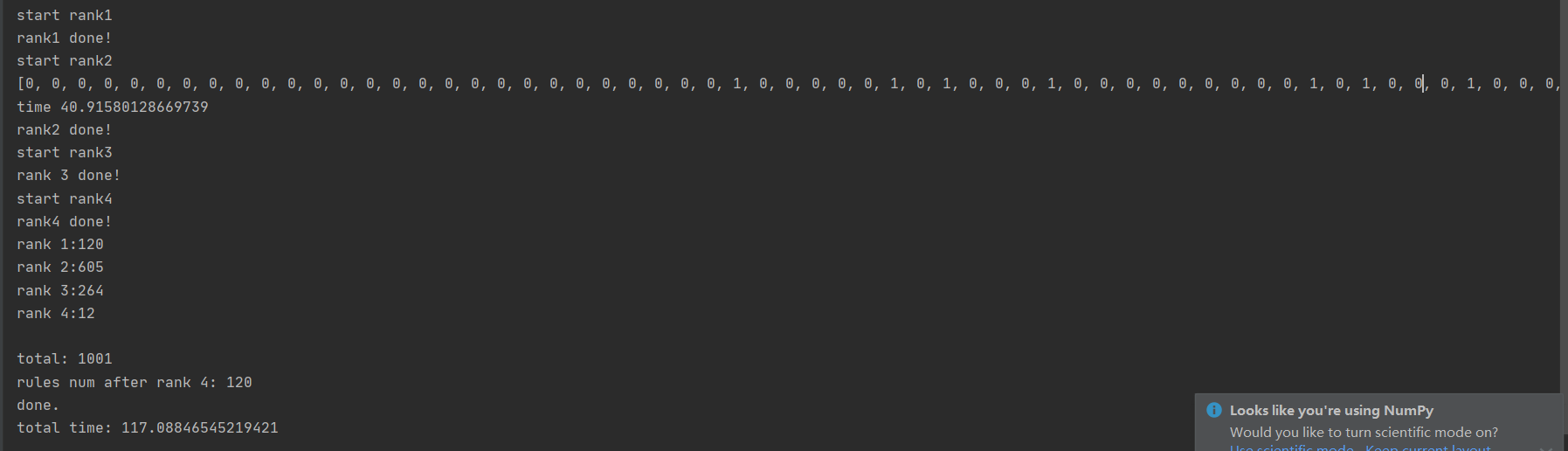
2. 桶的个数选择

桶的个数在这里是一个超参数，不同的个数对结果影响很大，这也很好理解，当桶的个数过小，会导致一个桶中有太多项，很容易成为频繁桶，也就没有了筛选功能，甚至还会因为多了这些操作拖慢时间；另一方面，当桶的个数过大，会导致计算开销上升。对此则是经过了一系列测试，最终将桶的个数定位1000。

**3.2.3 实验测试与结果分析**



通过和助教的答案进行对比，可知结果正确，相关规则在output.txt中已打印。



可以看到，总的时间用了117tick左右，算是比较快的，而对于PCY算法，其使用了约40tick左右。

值得注意的是，经过实验发现，在当前的数据规模下，使用PCY算法并没有很大地加速算法，这个跟数据内容和数据规模有一定地关系，也有可能是因为代码写地不好。

**3.3 实验总结**

这次实验感觉是四个实验中最难的一个了，花费了许多时间理解算法原理并查阅了很多相关的资料，以及python的使用上也遇到了一些难题，不仅如此，调bug也调了很久，性能也较差，希望自己继续努力。

# 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。

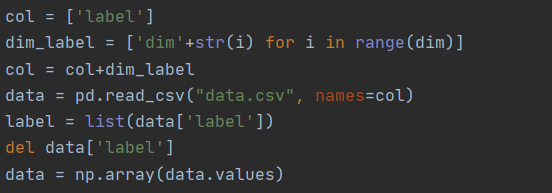


图4.1 葡萄酒数据集在黄酮和总酚维度下聚类图像（SSE为距离平方和，Acc为准确率）

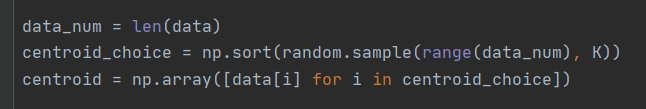
**4.3 实验过程**

**4.3.1 编程思路**

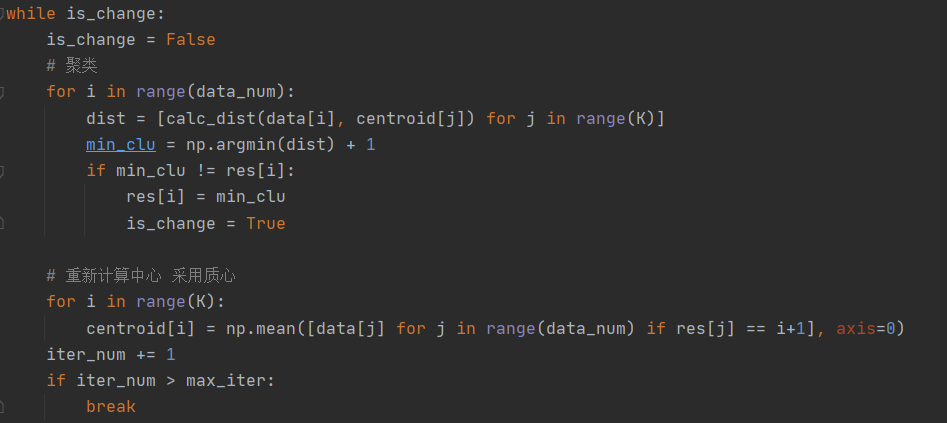
K-means思路很简单，首先读入原始数据并将其转换为矩阵方便操作。



接着利用random模块随机选择三个初始点。

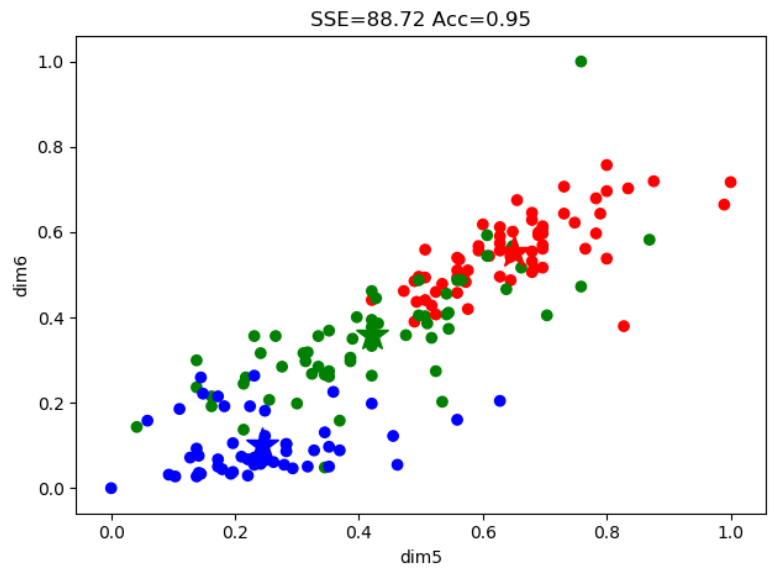


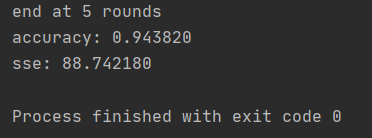
随后开始迭代，每一次迭代都重新计算每个点到中心点的距离并重新归到距其距离最小的中心点的类，并重新计算中心点，计算中心点的方式采用质心，及当前类中所有的点对应维度的平均值作为中心点当前维度的值，迭代结束的条件一方面是每个点不再变化类别以及不超过最大迭代次数。



最后利用matplotlib进行可视化分析

**4.3.2 实验测试与结果分析**





这里利用第五个维度和第六个维度进行平面的分析，星型的点代表中心点，可以看到，基本上分的非常好SSE为88.72，准确度有95%，需要注意的是，聚类算法有随机性，有一定可能结果非常差，这是因为最初的三个点是随机选的，并且是随机分配类的，这有两个影响因素，一方面初始点是随机的，会导致结果偏差，因为有可能选到三个特别近的点，另一方面，初始分配的类也是随机的，所以有可能点虽然聚对了，但是这个类的label不对，这也会使准确率很低，但是sse并不会受此影响。除此之外，由于数据集处理地非常好，可以看到5轮左右迭代就会停止。

**4.4 实验总结**

K-means的实验较为简单，主要就是理解算法的过程，由于数据集处理地非常好，效果也很喜人，通过这次实验，学到的更多的是可视化的使用，这是之前接触比较少的知识。

# 实验五 推荐系统算法及其实现

## **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：**

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：

**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**。

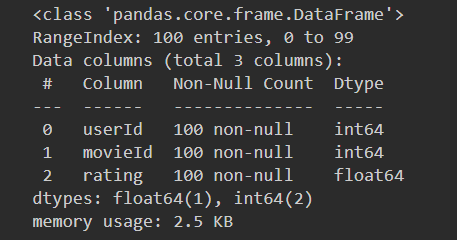
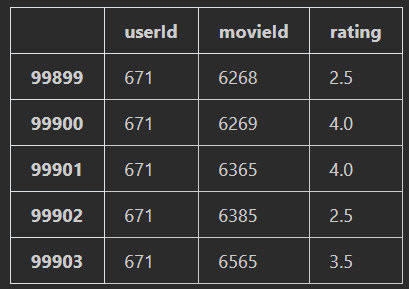
**对基于用户的协同过滤推荐算法和基于内容的推荐算法进行推荐效果对比和分析，选做的完成后再进行一次对比分析。**

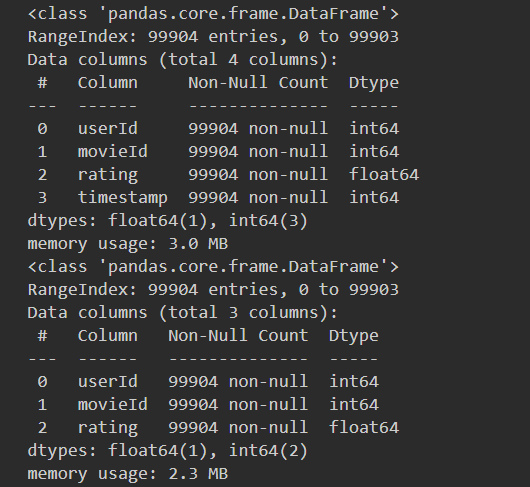
## **5.3 实验过程**

### 5.3.1 编程思路

1. User-User Collaborative Filtering

这个部分是利用用户-用户之间的协同滤波来实现推荐，这一部分将首先从数据集的观察入手，然后设计推荐算法。

 首先观察数据集。

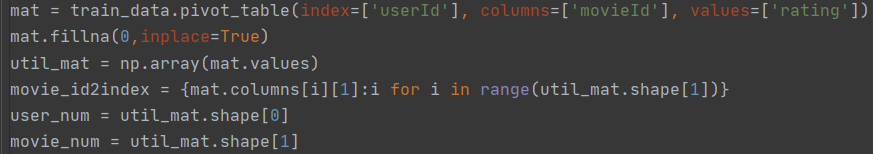


可以看到，训练集中有99904条数据，其中有四个属性，userId，movieId，rating和timestamp，由于本次实验中不需要利用到时间戳，所以在这里去掉时间戳这个属性。而测试集则是有100条记录，和训练集具有同样的属性。还可以看到，movieId并不是连续的，存在某些movie没有任何记录的情况，这需要我们对movieId进行映射。

接下来就是考虑算法的实现。这里主要有三个步骤：生成效用矩阵utility-matrix，生成相似度矩阵similarity-matrix，以及最后实现预测。对于预测，根据任务，需要有两种预测的模式：topN预测以及评分预测，其中topN预测是在评分预测的基础上对评分进行排序后选前N个进行推荐。

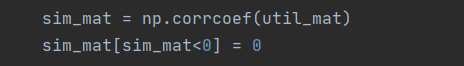
对于minihash优化来说，minihash主要是影响相似度矩阵的生成，通过hash达到一个降维的效果，所以只用改相似度矩阵的生成部分即可。

- 效用矩阵的获取。

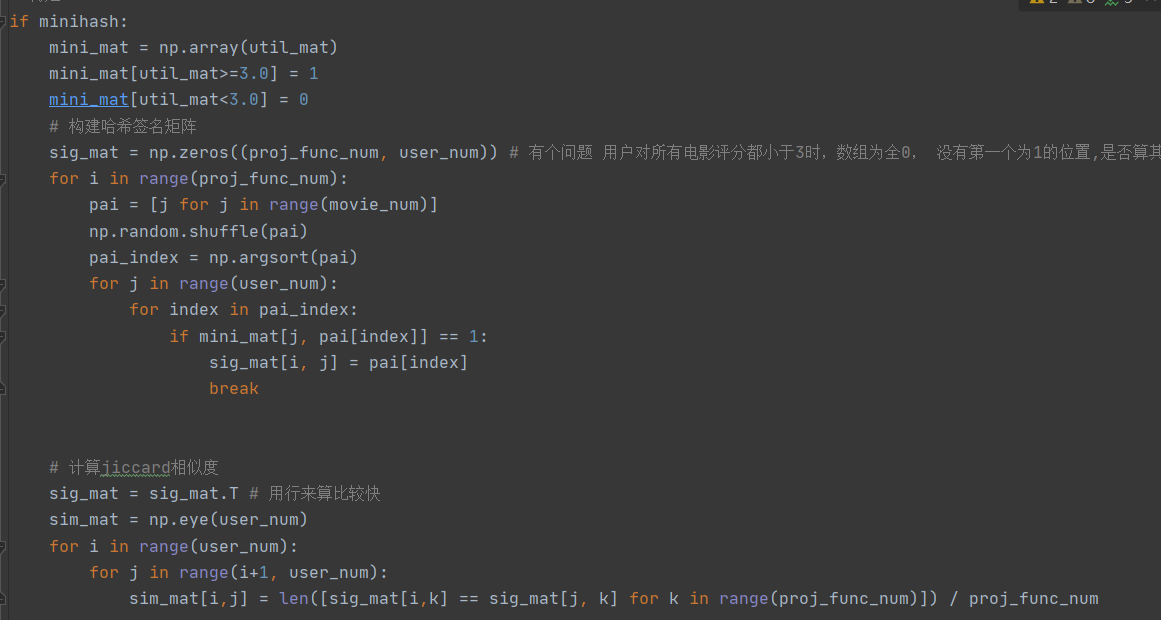


这里简单的使用pandas中的pivot\_table函数获得效用矩阵，并将其转换为numpy矩阵方便后续的操作，由于之前观察到数据集movieId并不连续，所以这里建立了一个字典存储从movieId到矩阵中的列的映射关系，这里需要注意的是，我在具体实现时，效用矩阵和理论课讲的不太一样，我是将每一行当成一个user，每一列当成一个movie，这种更符合能够利用程序的局部性，让cache多命中，加快读取速度，而且许多库函数都是基于行来进行计算的。

- 相似度矩阵的获取

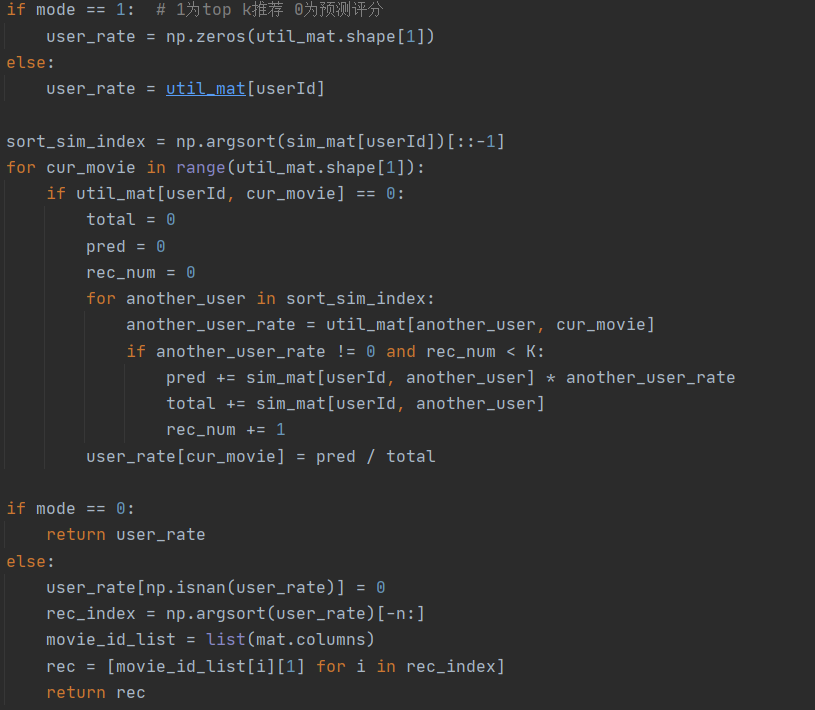


相似度的计算利用pearson相似度，在这里直接利用numpy提供的库函数进行生成，这个库函数是算每一个行向量之间的pearson相似度，这也是之前提到生成的效用矩阵为每一行是一个user的原因之一。需要注意的是，这里计算相似度有可能出现负值，这里的负值应被置为0，以免在后续计算评分时出现无法归一化的情况。



然后是考虑利用minihash生成相似度矩阵，具体而言，首先需要将效用矩阵变为01矩阵，把评分3.0以上的归为1，以下的归为0，接着就是生成签名矩阵，首先shuffle一个序列Π，然后找原矩阵根据shuffle后的序列的结果Π找第一个为1的位置，注意，这里并不是真的需要将原矩阵进行shuffle，只需要找到第一个为1的位置即可。接着再利用jaccard计算签名矩阵中user和user的相似度，由此构造了相似度矩阵。

- 评分计算及topN推荐



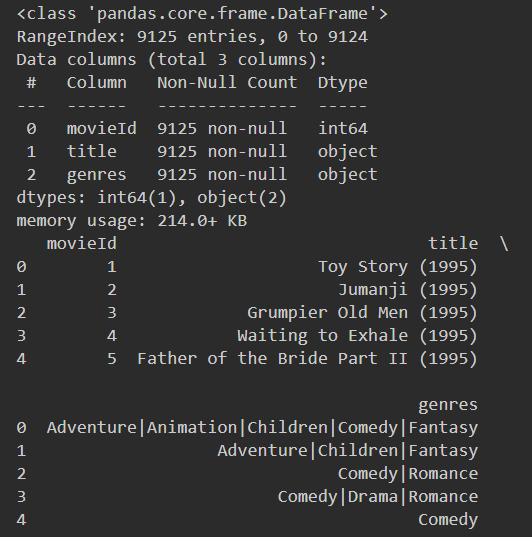
这里计算评分，主要就是利用公式：

对于topN推荐推荐则是选取前n个未被用户评过分中的评分最高的前n个进行返回。这里要注意的是，由于有浮点运算，所以需要将结果为nan的置为0。

2. Content-based Recommendation

这个任务主要是利用movie-movie的协同滤波来进行推荐，与之前不同的是，这里的相似度矩阵不是通过效用矩阵得到的，而是通过对电影进行特征化后的特征矩阵求出，即利用电影的“文本”，这里主要是电影的种类文本，来对电影进行特征化，利用特征矩阵算相似度。

同理，首先观察一下数据集。



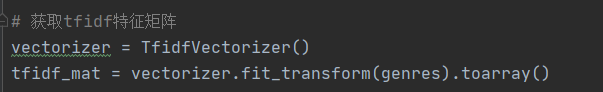
可以看到数据集中有9125条记录，对应为有9125个movie，需要注意的是，movieId并不是连续的，需要建立映射，一条数据有三个属性，分别是movieId，title和genres，其中genres是我们进行特征化的对象。而训练集和数据集则与上个任务相同。

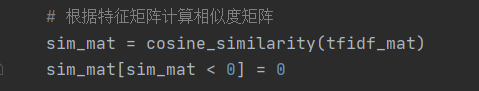
接着便是设计算法，同理，也是三个步骤：生成效用矩阵，生成相似度矩阵，进行推荐。

- 生成效用矩阵

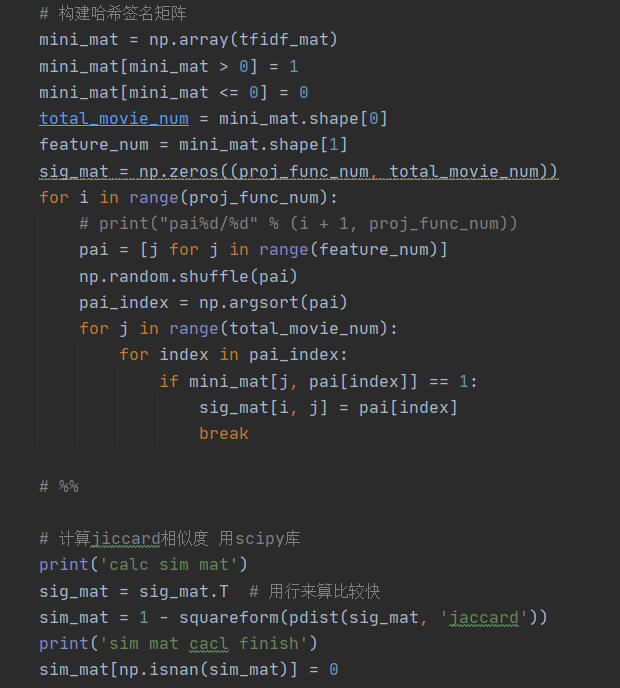
这一部分和上一个任务一样，不多叙述。

- 生成相似度矩阵



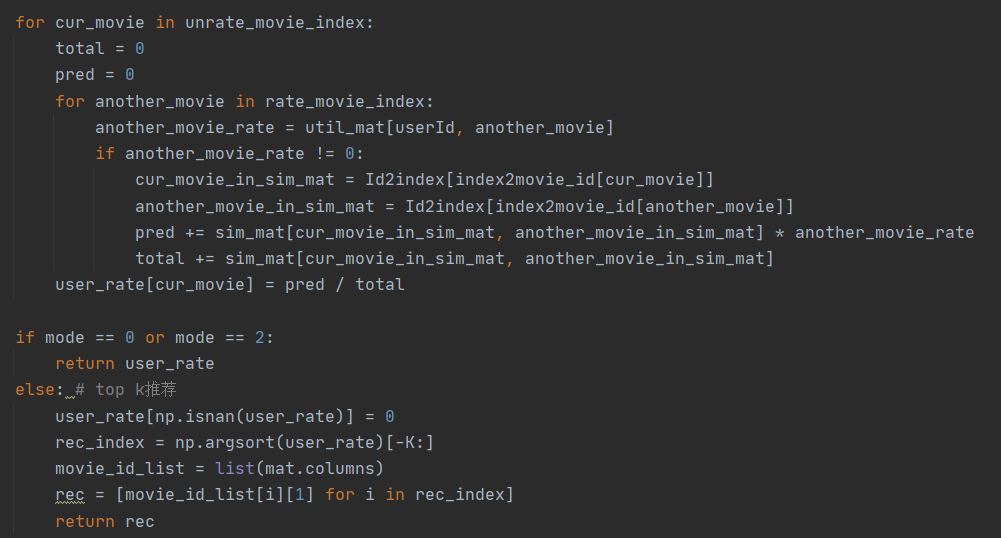
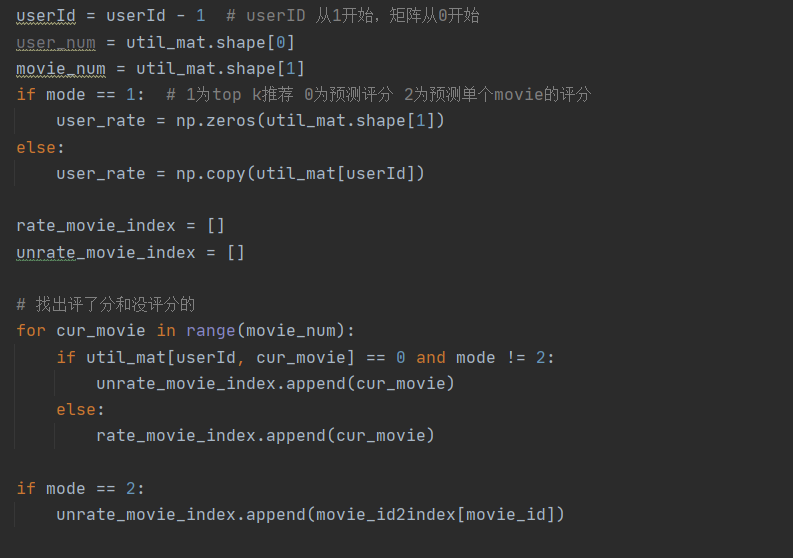


这个和上一个任务不同的是，这里利用了电影的种类特征来进行相似度的计算。具体而言，首先需要对文本进行特征化，这里利用sklearn中的库函数，获得文本的特征矩阵。再利用特征矩阵去计算相似度矩阵，其中cosine\_similarity是sklearn中的库函数，用于计算行与行之间的余弦相似度。



对于minihash优化，首先需要生成01矩阵，这里将特征矩阵中大于0的置为1，小于0的置为0，接着利用同样的方法生成签名矩阵，并计算jaccard相似度，这里和之前不同，利用了库函数而不是自己实现，主要是想对比一下两个的速度差别。

- 评分计算及topN推荐



这里和之前有一点点区别，一方面进行了效率的提升，在第一个任务中，是利用了两重循环去算评分，由于是user-user，user只有671个，即671\*671的次循环，尚可接受，但对于movie-movie来说，有9000多个movie，即9000\*9000次循环，开销很大，于是采取了另一种策略，首先用一次循环把当前user评了分的和没评分的取出来，然后再进入两重循环，这样可以大大减少循环的开销。剩下的基本就和上个任务相同，这里运用公式

进行计算，需要注意的是，这里并没有取最相似的K个进行评分预测，而是算算所有其他的电影和当前电影的加权，然后选评分前K个的电影进行推荐，即这个任务的K相当于上个任务的N。

### 5.3.2 实验测试与结果分析

1. User-User Collaborative Filtering

主要设计四个测试：

- TopN推荐测试

- 测试集SSE测试

- minihash效果测试

- minihash的超参--映射函数数量敏感测试

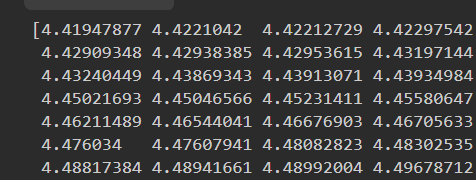
需要注意的是，实验中的其它超参数K和N为任务指定，这里不对其进行敏感测试，其中K为10，N为5

a. TopN测试实验





上处为userID为1，20，600，三个user的推荐结果，经过比对，可知结果正确，比较值得在意的是可以观察到，他们的评分都是5.0的满分，于是我将N值设大观察了一下。



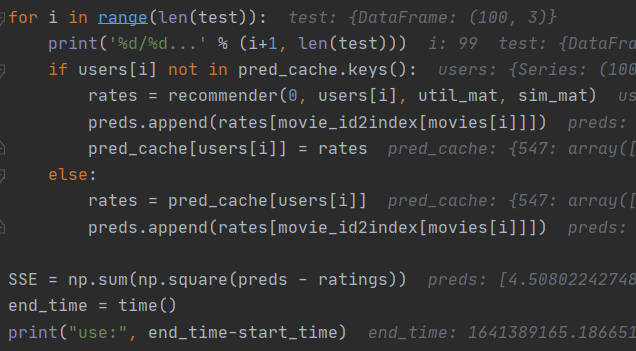
可以看到，并不全是5.0的，所以计算过程应该无误。

b. 测试集SSE





经过测试，SSE在85.90925左右，符合预期，并且时间方面用了约153秒，之所以会这么慢，是因为每测试一条记录，都会把当前user所有的movie评分都预测出来，而不是只预测测试集中特定一个的movie，当然为了加快进度，我也进行一些改进。



可以看到，主要是利用了一个缓存计数，由于一次是需要把所有评分都计算出来，所以如果计算过该user的评分，只需要找到相应movie即可，不需要重复进行计算，但也可以发现，还是计算了很多不需要的评分，这一部分将在下个任务中进行改进。

c. minihash效果测试





可以看到，利用minihash后，时间成本并没有显著上升，主要时间集中在利用minihash计算相似矩阵上，就最后的结果来看，比不用minihash稍差一点，这也比较符合预期，一方面minihash采用的是近似算法，另一方面是minihash采用的jaccard相似度。

d. 映射函数数量敏感测试

这里设计了3组，分别为10，100，1000对原来的特征进行降维处理，获得相似度矩阵，并分别衡量时间和SSE效果

|  |  |  |
| --- | --- | --- |
| 映射函数数量 | 计算时间 | SSE |
| 10 | 1.7459897994995117 | 99.69321180555556 |
| 100 | 15.414995908737183 | 90.69321180555556 |
| 1000 | 151.75450205802917 | 80.69321180555556 |

可以看到，随着映射函数数量的提升，效果也有所提升，这页式比较符合常理的，因为降维实际上是在损失信息。

2. Content-based Recommendation

同理，这里也设置如下四个实验：

- TopN推荐测试

- 测试集SSE测试

- minihash效果测试

- minihash的超参--映射函数数量敏感测试

A. TopN推荐测试







这是userId为1，20，600的三个user的Top5的推荐结果，可以发现，这个和之前那个任务所推荐的结果完全不一样，这是比较值得思考的，可能原因是由于相似度变为计算movie和movie之间的相似度来计算评分，算法变了，从这方面来看，两种算法之间的差距可谓非常大。

B. 测试集SSE测试

这里需要注意的是，在具体实现时，为了加快测试的速度，还增加了一种模式，即只预测某一个user对某一个movie的评分，而不用预测出某一个user对所有movie的评分，这样可以大大加快测试的速度，是之前提到一个改进的手段。



上图是SSE的结果，可以看到效果比User-User的任务要好很多，这也和之前推荐的结果不一样互相印证。

C. minihash效果测试



可以看到，加了minihash后，效果反而还变好了，这是比较反直觉的，因为minihash是降维，会损失信息，猜测可能原因是minihash的降维达到了一定的去噪效果。

D. minihash的超参--映射函数数量敏感测试

|  |  |  |
| --- | --- | --- |
| 映射函数数量 | 计算时间 | SSE |
| 2 | 3.4139909744262695 | 72.71847575457196 |
| 5 | 3.7710354328155518 | 62.73455729371988 |
| 10 | 4.88656759262085 | 52.29328575238883 |
| 15 | 4.585342884063721 | 51.23988374837842 |
| 20 | 6.487042188644409 | 50.87398792384827 |

首先可以看到映射函数的数量减少了很多，这是因为经过tfidf向量化后，向量的维度就只有24维，所以再进行降维只能往更小的走，另一方面，可以看到随这映射函数数量增加，计算时间增加，这比较符合常理，还有可以看到，SSE在经过大幅下降后，趋于平稳，这可能是因为已经接近原先的维度。

## **5.4 实验总结**

通过本次实验，掌握了基于内容推荐以及基于用户的协同过滤的推荐算法，以及利用minihash进行降维处理的手段。

对于基于用户的协同过滤，其不需要电影的特征，只需要用户的历史评分，即可对未评分的电影进行评分预测，但是其也存在着一些问题，比如不能很好的对从来没有过评分的用户进行预测。

而对于基于内容的推荐，从结果上来看，其推荐效果会更加好一点，但也存在着极为费时的弊端，一方面其需要对文本进行特征化处理，另一方面，电影的数量很庞大，计算开销较大。

整个实验做下来，印象最深刻的是优化部分，因为最初的代码很快就写好了，但是存在着跑的过慢的问题，为了进行优化主要进行过以下的操作：

- 通过模块化代码并分别测时，找到拖慢时间的地方

- 对于测试设置“cache”，记录user评过的电影

- 增加一种模式，单独实现某个user针对某个movie的评分

- 将自己实现的相似度计算更换为numpy，scipy库函数

- 优化代码结构，减少循环的次数

总的来说，这次实验令我受益匪浅，可以改进的地方可能是希望数据量能再大一点，比如基于内容的推荐，降维的幅度很小，虽然有效果，但是并不明显。最后，希望大数据分析课程越来越好。