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： **CS2008**

学　　号： **U202015533**

姓　　名： 徐瑞达

指导教师： 王蔚

报告日期： **2022 年 12 月 8 日**

计算机科学与技术学院

目 录

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。

1.3 实验过程

1.3.1 编程思路

（1）Map

由于共有 9 个源文件，因此需要利用多线程模拟 9 个 Map 节点，每个节点处理一个源文件。在每个节点中，利用 Scanner 和 split()函数从源文件中读取单词；使用 Map<String,Interger>对象存储每个单词出现的次数。

（2）Combine

在每个 Map 节点处理源文件时，可以使用 TreeMap 实现。当单词已存在于字典中时 put(word,get(word)+1)，否则调用 put(word,1)录入单词。

（3）Shuffle

在每个 Map 节点处理完成后，使用 shuffle 函数将统计结果分为三部分。将

首字母为 a~o 的单词划分至 shuffle2、首字母为 p-z 的单词划分至 shuffle3、其余单词划分至 shuffle3。最终输出结果时，每个节点将输出三个 map 文件。

（4）Reduce

本实验要求模拟 3 个 Reduce 节点，每个节点处理一个 shuffle 文件。以第 1 个 Reduce 节点为例，遍历每个 Map 节点输出的 shuffle1 文件，按照与 Combine 类似的逻辑统计单词个数，最终输出到 reduce1 文件中。

1.3.2 遇到的问题及解决方式

问题：在实现多线程时，Reduce 节点应在所有 Map 节点完成后运行，许因此需要统一管理 Map 节点。

解决方式：使用 Java 提供的 ExecutorService 类统一管理 Map 节点，通过轮询所有线程是否结束，实现 Map 节点均完成后运行 Map 节点。

1.3.3 实验测试与结果分析

由于 Map、Combine、Shuffle 的功能均在线程类 MapRunner 中实现，因此这里只展示 Shuffle 后的文件。文件结构如图 1.1 所示，可见每个 Map 节点共输出 3 份文件，分别是 part1、part2、part3 且各部分文件大小基本相同。文件 map01.part1 内容如图 1.2 所示，可见经过 Combine 处理后，得到每个单词出现的次数。

图 1.1 Map 处理后的文件结构

```
'sbodikins:2
'sdeath:2
'sfoot:4
'shun:3
'slid:3
'slife:2
'snails:3
't:3
'til:3
'tis:3
'twas:3
'tween:1
'tween-decks:2
'twere:6
```

图 1.2 Map 处理后的文件内容

Reduce 负责统计各个 part 中单词出现次数，最终合并 3 个 Reduce 节点的结果，文件内容如图 1.3 所示。

```
&c:19
'd:20
'em:17
'll:22
'm:18
'mid:17
'midst:24
'mongst:13
'prentice:21
're:14
's:21
'sblood:20
'sbodikins:22
```

图 1.3 Reduce 处理后的文件内容

统计各个 Map 节点和 Reduce 节点运行时间，结果如图 1.4 所示。

```
Start Mapper...
Finished Mapper:source06,Used Time:6700ms
Finished Mapper:source07,Used Time:6699ms
Finished Mapper:source03,Used Time:6733ms
Finished Mapper:source02,Used Time:6898ms
Finished Mapper:source05,Used Time:7200ms
Finished Mapper:source08,Used Time:7250ms
Finished Mapper:source01,Used Time:7312ms
Finished Mapper:source04,Used Time:7512ms
Finished Mapper:source09,Used Time:7718ms
Start Reducer...
Finished Reducer:./reduce/reduce02,Used Time:3183ms
Finished Reducer:./reduce/reduce03,Used Time:3384ms
Finished Reducer:./reduce/reduce01,Used Time:3415ms
Merge Reduce:Successful
```

图 1.4 各节点运行时间

1.4 实验总结

通过实现 MapReduce 算法，我认识到了分布式运算的强大之处——通过多线程处理提高运行效率。

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[1]；（可选进阶版）理解阻尼系数[2]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。

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。最后输出人名 id 及其对应的 pagerank 值。

2.3 实验内容

2.3.1 编程思路

1. 定义矩阵类 Matrix；
2. 读取文件 sent_receive.csv，统计节点集 nodes
3. 构建大小为 nodesLength×nodesLength 的邻接矩阵 M（当存在 i 发送给 j 的邮件时，M[i,j]=1）；
4. 标准化矩阵 M，使矩阵的每一列元素之和为 1；
5. 初始化 currentPr 矩阵；
6. 代入迭代公式计算 nextPr 并标准化，计算每次迭代前后的误差率

[1] 基本 pagerank 公式 r=Mr

[2] 进阶版 pagerank 公式：$r=\beta Mr+(1-\beta)\left[\frac{1}{N}\right]_{N\times N}$，其中$\beta$为阻尼系数，常见值为 0.85

errorRate；

7. 迭代至误差率小于 10^{-8} 时，终止迭代并输出最终结果。

2.3.2 遇到的问题及解决方式

问题：迭代次数过少，最终结果错误。

解决方式：邻接矩阵 M 构建错误，重新构建后结果正确。

2.3.3 实验测试与结果分析

迭代终止后，输出迭代次数与 ID 对应 PageRank 值，结果如图 2.1 所示。

```
迭代次数:51
ID  PageRank
 80 0.3177301970
 81 0.0281077590
 87 0.0271415768
 32 0.0232626315
 86 0.0101924246
153 0.0099642564
170 0.0100296133
116 0.0092869353
180 0.0092039920
 48 0.0087440822
145 0.0087240720
105 0.0080120619
196 0.0080120619
```

图 2.1 PageRank 输出结果

2.4 实验总结

通过本次实验，我掌握了 PageRank 算法，也了解了搜索引擎返回搜索结果的工作方式，深刻体会到了大数据算法在互联网中的重要作用。

实验三 关系挖掘实验

3.1 实验内容

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。

3.2 实验过程

3.2.1 编程思路

1. 从 Groceries.csv 中读取所有 basket 作为 C1；
2. 遍历 C1 中的 basket，统计每个元素出现的次数，同时统计所有可能的二项集出现的次数。遍历完成后，调用 getSupportedItems 方法排除出现频率小于于支持度 support 的元素以得到频繁一项集 L1。使用 hash 函数将二项集统计结果哈希到桶中，并压缩为 01 数组；
3. 组合 L1 中元素得到二项集，并根据 01 数组进行剪枝，得到候选集 C2。调用 getSupportedItems 方法得频繁二项集 L2；
4. 依照 C2、L2 的生成方式得到 C3、L3、C4、L4；
5. 遍历由 L2、L3、L4 生成的关联规则，根据置信度公式计算置信度，最后筛选出大于最小置信度的关联规则并输出。

3.2.2 遇到的问题及解决方式

问题：实现之前，没有正确理解 PCY 算法对于 Aprior 算法的改进方式，导致频繁二项集 L2 统计错误。

解决方式：PCY 算法的改进之处在于，在遍历 C1 中的 basket 时，统计所有

可能的二项集出现的次数，并哈希到桶中。

3.2.3 实验测试与结果分析

在使用 PCY 算法之前，运行结果如图 3.1 所示。

图 3.1 Aprior 算法运行结果

使用 PCY 算法之后，运行结果如图 3.2 所示，可见 C2 候选项集数缩减为一半左右，大大提高了效率。其中第一行是展示的部分 01 数组。

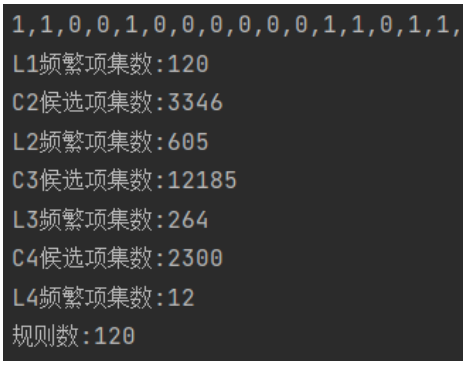

图 3.2 PCY 算法运行结果

输出的 L4 文件如图 3.3 所示。

	Set	Frequency Or Confidence
1	[citrus fruit,other vegetables,root vegetables,whole milk]	0.005795627859684799
2	[fruit/vegetable juice,other vegetables,whole milk,yogurt]	0.005083884087442806
3	[other vegetables,pip fruit,root vegetables,whole milk]	0.005490594814438231
4	[other vegetables,pip fruit,whole milk,yogurt]	0.005083884087442806
5	[other vegetables,rolls/buns,root vegetables,whole milk]	0.006202338586680223
6	[other vegetables,rolls/buns,whole milk,yogurt]	0.005998983223182512
7	[other vegetables,root vegetables,tropical fruit,whole milk]	0.007015760040671073
8	[other vegetables,root vegetables,whipped/sour cream,whole milk]	0.005185561769191663
9	[other vegetables,root vegetables,whole milk,yogurt]	0.007829181494661922
10	[other vegetables,tropical fruit,whole milk,yogurt]	0.007625826131164209
11	[other vegetables,whipped/sour cream,whole milk,yogurt]	0.005592272496187087
12	[root vegetables,tropical fruit,whole milk,yogurt]	0.005693950177935944

图 3.3 输出的 L4 文件

输出的关联规则文件如图 3.4 所示。

	Set	Frequency Or Confidence
1	[baking powder]->whole milk	0.522988505747126400
2	[beef,rolls/buns]->whole milk	0.500000000000000000
3	[beef,yogurt]->whole milk	0.521739130434782600
4	[bottled beer,yogurt]->whole milk	0.560439560439560400
5	[bottled water,butter]->whole milk	0.602272727272727300
6	[brown bread,other vegetables]->whole milk	0.500000000000000000
7	[brown bread,root vegetables]->whole milk	0.560000000000000000
8	[brown bread,tropical fruit]->whole milk	0.533333333333333300
9	[butter,citrus fruit]->whole milk	0.555555555555555500
10	[butter,domestic eggs]->whole milk	0.621052631578947400
11	[butter,other vegetables]->whole milk	0.573604060913705500
12	[butter,root vegetables]->other vegetables	0.511811023622047200
13	[butter,root vegetables]->whole milk	0.637795275590551200
14	[butter,tropical fruit]->other vegetables	0.551020408163265400
15	[butter,tropical fruit]->whole milk	0.622448979591836800
16	[butter,whipped/sour cream]->other vegetables	0.570000000000000100
17	[butter,whipped/sour cream]->whole milk	0.660000000000000000

图 3.4 输出的关联规则文件

3.3 实验总结

在本次实验中，我实现了关联规则挖掘 Aprior 算法，并使用 PCY 算法显著提高了算法的运行效率，这让我认识到通过剪枝策略在运行时剔除无效数据对提高大数据运算性能的重要作用。

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(WineData.csv)，数据集已经被归一化(normalizedwinedata.csv)。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。

4.3 实验过程

4.3.1 编程思路

1. 从归一化数据文件中读取各项数据，每项数据包含 13 维特征，存储在 Point 对象中；

2. 随机选取三个点作为初始中心点，计算每个点到这三个中心点的距离，选取距离最小对应的中心点并根据每个点距离最小的中心点划分为三个聚类 Cluster；

3. 根据欧式距离公式，计算每个聚类的新质心，重复 2、3 步；

4. 迭代直至迭代前后中心点的坐标不再变化，即得最终结果；
5. 分别统计标准结果、实际结果的 SSE，并计算准确度，准确度公式如下

$$\text{acc} = \frac{\text{正确分类的项目数}}{\text{项目总数}}$$

4.3.2 遇到的问题及解决方式

问题：误以为准确度计算公式为实际结果 SSE/标准结果 SSE。

解决方式：按照正确的准确度计算公式，统计正确分类的项目数。

4.3.3 实验测试与结果分析

程序运行结果如图 4.1 所示。其中准确度约为 0.9428，程序结果符合预期。

```
迭代总次数:6
标准距离和:48.28976523849757
实际距离和:48.97029115513917
算法准确率:0.942857142857143
各聚类的质心与距离和:
Centroid:[0.5446890215311005 0.47844055156306153 0.5601361740398638 0.5383317764761013 0.31146245167984193 0.244
SSE:15.50611410757559
Centroid:[0.7077193245614037 0.24189725296442685 0.5863636898395723 0.34424399140893475 0.4115942039855073 0.643
SSE:12.59183009348725
Centroid:[0.31566418671679225 0.24330261089152394 0.4733045267804091 0.4976272347406316 0.24948240274327113 0.45
SSE:20.872346954076328
```

图 4.1 Kmeans 聚类算法输出结果

4.4 实验总结

本次实验中，我实现了 Kmeans 聚类算法，掌握了通过迭代修正逼近标准结果的算法思想。同时，由于我采用 Java 语言编写的程序，不方便绘制准确度图表，让我认识到了 Python 语言在大数据处理、机器学习等领域的重要性。

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 minhash 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

基础内容一：**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**。

基础内容二：**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**。

5.3 实验过程

5.3.1 编程思路

1. 基于用户的协同过滤算法

该算法的核心在于计算用户之间的相似度，以根据邻居评分情况推荐电影。

（1）读取数据

- 从 test_set.csv 文件中读取所有评分记录：读取用户 ID、电影 ID、评分存储在 Record 对象中，获得原始评分数据集；
- 从 movies.csv 文件中读取所有电影：读取电影 ID、电影名称、电影类型存储在 Movie 对象中，获得电影信息集以便于计算电影之间的相似度；
- 根据评分记录，统计出每个用户的评分列表、看过的电影列表，同时统计出看过某部电影的用户列表。

（2）计算用户之间的相似度

- 构建 User-Movie 矩阵：设用户数为 M，电影数为 N，构建 M×N 的矩阵，Matrix[i,j]表示用户 i 对电影 j 的评分；
- 计算 Pearson 相似度：根据 User-Movie 矩阵，计算两个用户之间的相似度，构建 M×M 的相似度矩阵，计算公式如下：

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

（3）推荐电影

- 选取邻居：对于某个用户，根据相似度矩阵获取与该用户相似度最高的 K 个用户；
- 计算预期评分：计算得到的 K 个邻居看过但是该用户未看过的电影的预期评分。预期评分的计算公式如下，其中一部分为该用户对已看过电影的评分平均值，一部分为 K 个邻居对当前电影的评分与该邻居平均评分之差的加权值（加

权系数为相似度）；

$$\widehat{R}_{i,m} = \overline{R_i} + \frac{\sum_{j=1}^{k} Sim(i,j)\left(R_{j,m} - \overline{R_j}\right)}{\sum_{j=1}^{k} Sim(i,j)}$$

- 对所有预期评分结果排序，选取评分最高的 N 个结果推荐给该用户。

（4） 运行测试集

读取测试集，计算每部电影的预期评分，与实际评分比较，计算得到 SSE 值。

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

2. 基于内容的推荐算法

该算法的核心在于计算电影之间的相似度，以根据用户评分情况推荐电影。

（1） 读取数据

- 从 test_set.csv 文件中读取所有评分记录：读取用户 ID、电影 ID、评分存储在 Record 对象中，获得原始评分数据集；
- 从 movies.csv 文件中读取所有电影：读取电影 ID、电影名称、电影类型存储在 Movie 对象中，获得电影信息集以便于计算电影之间的相似度；
- 根据评分记录，统计出每个用户的评分列表、看过的电影列表。

（2） 计算电影之间的相似度

- 计算 TF-IDF 矩阵：根据电影信息集，统计电影总数 M 和类型总数 N，构建 M×N 的 TF-IDF 矩阵。首先分别计算每个电影的每个类型标签的 TF 值和 IDF 值；TF 表示词频，即单词在文件中出现的次数，当电影具有某个标签时，TF 为 1/N，否则为 0；IDF 表示反文档频率，其公式如下。TF 矩阵和 IDF 矩阵对应相乘即为 TF-IDF 矩阵。

$$IDF = \log\left(\frac{\text{文档总数}}{\text{包含该单词的文档总数} + 1}\right)$$

- 计算余弦相似度：构建 M×M 的相似度矩阵。遍历 TF-IDF 矩阵中的某两行，根据余弦公式（如下）计算得到这两行对应电影的相似度，写入相似度矩阵中。Matrix[i,j]表示电影 i 和电影 j 之间的相似度。

$$CosineSimilarity(x,y) = \frac{\vec{x} \cdot \vec{y}}{|\vec{x}| \cdot |\vec{y}|}$$

（3） 推荐电影

- 对于某一用户，计算其未看过电影的预期评分：根据用户的评分记录和电影间的相似度计算预期评分，电影相似度为负数时不参与计算。计算公式如下：

$$\hat{S} = \frac{\sum_{i=1}^{n} S_i \cdot sim(i)}{\sum_{i=1}^{n} sim(i)}$$

- 对所有预期评分结果排序，选取评分最高的 N 个结果推荐给该用户

（4） 运行测试集

读取测试集，计算每部电影的预期评分，与实际评分比较，计算得到 SSE 值。

5.3.2 遇到的问题及解决方式

问题：在基于用户的协同过滤算法中，采用原始公式计算时，由于在某用户的邻居中，可能存在很多人未看过电影 A，因此预期评分会偏向于看过电影 A 的少部分邻居的评分，存在较大误差。

解决措施：将预期评分分为两部分，其中一部分为该用户对已看过电影的评分平均值，一部分为 K 个邻居对当前电影的评分与该邻居平均评分之差的加权值。这样可以充分利用各邻居的评分，不会导致太大的误差。

5.3.3 实验测试与结果分析

1. 基于用户的协同过滤算法

对于用户 12，选择 10 个邻居，为其推荐 5 部电影。运行结果如图 5.1 所示。

```
输入推荐算法类型(输入0表示UserCF, 输入1表示ContentBased):0
[UserCF]输入用户ID(输入-1退出程序,输入0运行测试集):12
------------------------------为用户12推荐的电影列表------------------------------
 电影ID  预测评分      电影类型              电影名称
  1173  5.072280    Comedy|Drama        Cook the Thief His Wife & Her Lover The (1989)
  1244  5.072280    Comedy|Drama|Romance    Manhattan (1979)
  1625  5.072280    Drama|Mystery|Thriller  Game The (1997)
  2167  5.072280    Action|Horror|Thriller  Blade (1998)
  2318  5.072280    Comedy|Drama        Happiness (1998)
```

图 5.1 基于用户的协同过滤算法推荐结果

运行测试集的结果如图 5.2 所示。

```
 607       2    3.638747    3.000000
  19       1    3.593027    3.000000
  19       2    3.472668    3.000000
 199       6    3.691536    4.500000
 199      10    3.568844    2.500000
 285       1    3.298484    4.000000
 285       2    3.258902    3.000000
 150       1    3.237399    3.000000
 150       2    3.021726    3.000000
SSE为66.722671
```

图 5.2 基于用户的协同过滤算法运行测试集结果

2. 基于内容的推荐算法

对于用户 12，为其推荐 5 部电影。运行结果如图 5.3 所示。

```
输入推荐算法类型(输入0表示UserCF, 输入1表示ContentBased):1
[ContentBased]输入用户ID(输入-1退出程序,输入0运行测试集):12
------------------------------为用户12推荐的电影列表------------------------------
 电影ID  预测评分      电影类型              电影名称
   37  5.000000    Documentary|IMAX    Across the Sea of Time (1995)
   77  5.000000    Documentary         Nico Icon (1995)
   99  5.000000    Documentary         Heidi Fleiss: Hollywood Madam (1995)
  108  5.000000    Documentary         Catwalk (1996)
  116  5.000000    Documentary         Anne Frank Remembered (1995)
```

图 5.3 基于内容的推荐算法推荐结果

运行测试集的结果如图 5.4 所示。

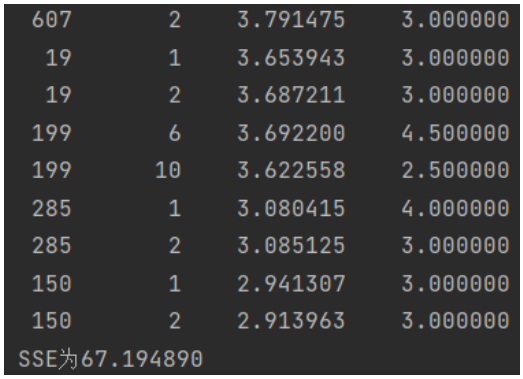

图 5.4 基于内容的推荐算法运行测试集结果

5.4 实验总结

本次实验中，我掌握了基于用户的协同滤波和基于内容两种推荐算法。这两种算法是实际推荐系统中的重要算法，也各有其优缺点。

基于用户的协同滤波算法基于相似用户推荐电影，能够涵盖不同的喜好，适用于个性化需求不高的场景，也不会导致始终给用户推荐某种内容的极端情况。但是其又有冷启动问题，即当某推荐系统积累数据量过少时将无法向新用户进行个性化推荐。在实际应用中，可以通过根据用户个人信息进行粗粒度的个性化推荐、要求用户填写个人喜好、先基于内容推荐然后根据基于用户推荐等方式解决该问题。

基于内容的推荐算法基于相似电影推荐电影，能够符合用户的个人喜好，适用于个性化需求强烈的场景。该算法不存在冷启动问题，当某个用户产生对某个物品的新行为时，即可根据该行为进行个性化推荐。然而，使用这种推荐算法将会导致推荐内容过于专一，用户只能获得与其自身画像中的项目类似的推荐结果。同时，如何精准刻画内容之间的相似度也是十分重要的。在向用户推荐之前，推荐系统需要存储丰富的内容描述，如推荐电影时，不仅需要考虑电影类型，还需要考虑导演、演员、情节、台词等相关指标。

可见，不同的推荐算法各有优缺点，在实际应用中，需要通过结合不同的推荐算法，进行不断地测试调整优化相关参数来达到更理想的推荐效果。