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： **BD2201**

学　　号： **U202215566**

姓　　名： 刘师言

指导教师： 王蔚

报告日期： **2024 年 6 月 21 日**

计算机科学与技术学院

目录

1 Map-reduce 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 思想解决问题；
3、掌握并应用 combine 与 shuffle 过程。

1.2 实验内容

提供 9 个预处理过的文件夹（folder_1-9）模拟 9 个分布式节点中的数据，每个源文件夹中包含大约 6 千个文件，每个文件标题为维基百科条目标题，内容为对应的网页内容。提供 words.txt 文件作为待统计的词汇。

要求应用 map-reduce 思想，模拟 9 个 map 节点与 3 个 reduce 节点实现对维基百科条目词汇的词频的统计。

map 节点输出<((title1,key1),1),…,((titlem,keyn),1)>，其中 key 为文件 title.txt 中出现的且在 words.txt 中词。同时，要求最终的 reduce 节点输出出现次数最多的前 1000 个词汇，以及这些词汇的跳转关系。

输出对应的 map 文件和最终的 reduce 结果文件。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

Map-reduce 算法分为 map 和 reduce 两大过程，也可在这两大过程的基础上添加 combine 和 shuffle 过程来提高算法的效率。其中 combine 过程在 map 过程中对包含相同键的键值对进行分组，可提前汇总得到各词汇的出现次数，省去了 reduce 过程对出现次数的计算；shuffle 过程保证 map 节点卸载到每个 reduce 节点的工作量尽量相当，从而减少整体运行时间。

Map-reduce 算法的原理流程示意图如图 1.1 所示：

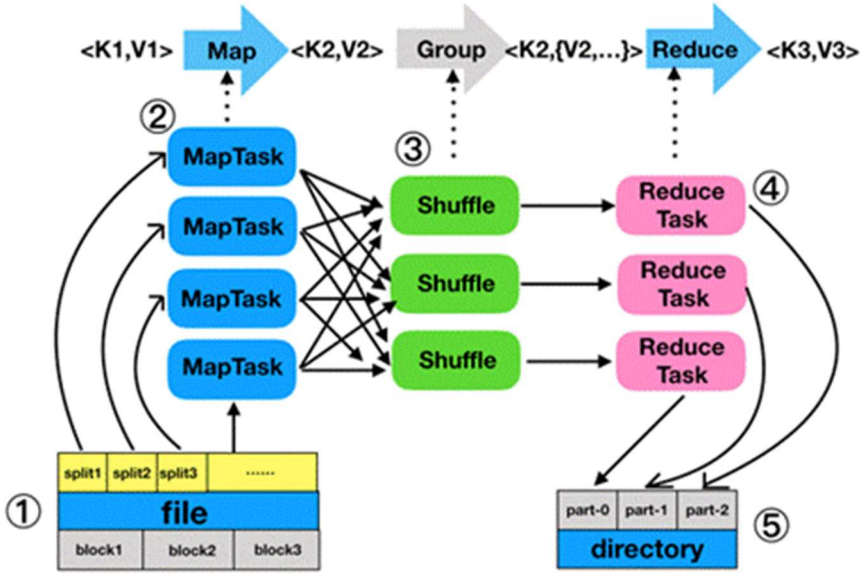

图 1.1 Map-reduce 算法的原理流程示意图

（1）数据集

数据集由 9 个文件夹组成，每个文件夹包含约 6 千个文件，每个文件标题为维基百科条目标题，内容为对应的网页内容。words.txt 文件为待统计的词汇，先读取该文件中的所有单词，并转化为列表存储在内存中。

（2）map 过程

map 过程使用多线程模拟 9 个分布式节点，每个节点对单个文件夹中的所有文件进行扫描。扫描过程中，每个文件的标题为 title，对文件的网页内容进行分词，并遍历分词后的词汇列表，将处于待统计词汇列表中的词汇 key 以键值对<<title, key>, 1>的形式加入到该 map 节点的输出中。

当加入 combine 过程时，用一个额外的字典来维护每组<title, key>键值对的出现次数，在遍历分词后的词汇列表时，若字典中未包含键<title, key>，则在字典中加入该键，并初始化出现次数为 0，然后对键<title, key>的出现次数加一，最后将字典转化为列表的形式作为 map 节点的输出，输出中的每项形如<<title, key>, n>，n 为该<title, key>键值对的出现次数。

由于 reduce 过程前可加入 shuffle 过程，保证每个 reduce 节点的工作量尽量相当，来减少整体运行时间，因此将每个 map 节点的输出平均分为 3 份，为后续的 shuffle 工作做准备。

（3）reduce 过程

reduce 过程使用多线程模拟 3 个分布式节点，节点上的任务由此前 map 节点的输出卸载而来。9 个 map 节点全部运行结束后，进行 shuffle 过程，将每个 map 节点特定位置的 3 分之一份输出合并作为一个 reduce 节点的任务，如 reduce 节点 1 的任务为每个 map 结点的前 3 分之一份输出之和，reduce 节点 2 的任务为每个 map 结点的中间 3 分之一份输出之和。

每个 reduce 节点内部维护一个 words 字典，用来记录各个词汇的出现次数及跳转关系。reduce 结点得到相应的输入后，依次遍历输入中的每一项<<title, key>, n>键值对，并解包为变量 title、key 和 n 分别表示标题词汇、网页内容中的词汇和词汇在该网页内容中的出现次数。若字典 words 中不包含 title 或 key 的键，则在字典中加入该键，并初始化词汇的出现次数为 0，跳转关系列表为空，然后对词汇 key 的出现次数加 n。若字典 words 中键 title 的跳转关系列表中不包含 key，则将 key 加入到 title 的跳转关系列表，表示由 title→key 的一个跳转关系。最后将 words 字典作为 reduce 节点的输出。

（4）合并统计过程

3 个 reduce 节点全部运行结束后，对每个 reduce 节点的输出进行合并，即将 3 个 words 字典合并成总的 words 字典，其中包含所有出现过的标题 title 以及关键词 key 所对应的出现次数和跳转关系列表。

最后将 words 字典转化为列表的形式，列表中的每个元素为单个词汇 word 的键值对，键为词汇 word 本身，值为记录 word 出现次数及跳转关系列表的字典。对转化得到的 words 列表按每个词汇的出现次数由大到小进行排序，取其前 1000 项保留，从而得到实验要求的出现次数最多的前 1000 个词汇及这些词汇的跳转关系。

为方便对实验结果进行查看及验证，通过遍历前 1000 个词汇的列表，将每个词汇的出现次数及跳转关系分别保存在两个不同的文本文件中。此外，为方便后续实验高效读取处理后的数据，还可将前 1000 个词汇的列表序列化到文件中，从而使后续实验可直接从文件中反序列化出数据集，简便算法编写流程。

1.3.2 遇到的问题及解决方式

（1）多线程协作

map 节点和 reduce 节点均采用多线程模拟分布式节点，而多个线程并发执行的耗时各不相同，需确保 map 过程在 9 个 map 节点均运行结束后再进入下一阶段，同样需确保 reduce 过程在 3 个 reduce 节点均运行结束后再进入下一阶段。

为解决上述问题，采取多线程的 join 方法，当 map 过程或 reduce 过程的所有线程开始执行后，依次调用各线程的 join 方法对其进行阻塞，以确保所有阻塞线程执行结束后主线程才会继续运行，即进入下一阶段。

此外，由于各个线程的运行时间各不相同，而程序的总运行时间往往由运行时间最长的线程决定。为确保程序有良好的运行效率，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。此时可对各 map 节点的输出进行 3 等分，并将每个 map 节点特定位置 1 等分进行合并并卸载到相应的 reduce 节点上，从而确保每个 reduce 节点的工作量尽可能相当。

（2）文本处理

遍历每个文本文件时，需要对文本文件的网页内容进行分词处理，将网页中的英文词汇提取并作为列表的元素保留，而切割其他非英文字符。因此，确定分词的策略十分关键，影响到算法中各词汇的出现次数及跳转关系。

本实验采用正则表达式[\W+]对网页进行分词，可精确匹配字符串中所有非英文字符进行切割，并将切割后保留下的英文词汇提取成列表。得到分词后的列表后，遍历列表中的每个词汇，将其统一转换成小写字母后再进行后续 map 过程的操作。

（3）运行效率

程序运行效率除受 shuffle 过程任务卸载策略的影响外，还受到多种因素的制约。比如在 map 过程遍历单个文件网页内容分词后的词汇列表时，为判断某词汇是否在给定的待统计词汇列表中，若直接用关键字 in 进行操作，其原理为遍历整个列表判断元素是否在列表中，时间复杂度为 O(n)，效率相对较低；而先将待统计词汇列表转化为以待统计词汇为键的字典，再判断该字典中是否包含某个词汇，时间复杂度为 O(1)，效率大幅提升。

1.3.3 实验测试与结果分析

程序事先对算法的各个阶段进行封装，接着在 main 函数中依次执行数据处理过程、map 过程、reduce 过程和合并统计过程，各过程执行结束后，控制台会打印相关提示信息，如图 1.2 所示：

```
C:\Users\LSY\AppData\Local\Programs\Python\Python39\python.exe
Map 1 starts!
Map 2 starts!
Map 3 starts!
Map 4 starts!
Map 5 starts!
Map 6 starts!
Map 7 starts!
Map 8 starts!
Map 9 starts!
Map 1 is done!
Map 5 is done!
Map 8 is done!
Map 7 is done!
Map 3 is done!
Map 9 is done!
Map 6 is done!
Map 4 is done!
Map 2 is done!
Reduce 1 starts!
Reduce 2 starts!
Reduce 3 starts!
Reduce 2 is done!Reduce 3 is done!

Reduce 1 is done!
Write words starts!
Write words done!

Process finished with exit code 0
```

图 1.2 控制台输出相关提示信息

由测试结果可以看到，map 过程的 9 个多线程节点全部执行完后才执行下一阶段，同样地，reduce 过程的 3 个多线程节点全部执行完后才执行下一阶段，验证了多线程模拟分布式操作的正确性。

当整个算法运行结束后，实验结果能界面友好地呈现在相应的文本文件中，包括记录词汇出现次数的 words_count.txt 文件和记录词汇跳转关系的 title_to_keys.txt 文件，分别如图 1.3 和图 1.4 所示。将文本文件中记录的词汇出现次数与词汇跳转关系数据分别与参考答案进行比对，发现各词汇的出现次数与跳转关系均准确无误，说明 map-reduce 算法的设计与实现正确。

words_count.txt × title_to_keys.txt

```
url : 70317
www : 65733
web : 64432
date : 62258
http : 60103
title : 56820
https : 47745
cite : 43409
archive : 42537
jpg : 31688
references : 24988
people : 24404
website : 23450
thumb : 21686
small : 19273
publisher : 18977
image : 18502
redirect : 17678
svg : 17677
br : 16610
```

图 1.3 词汇出现次数 words_count.txt 文件

words_count.txt title_to_keys.txt ×

```
url -> ['redirect']
www -> ['redirect', 'world', 'wide', 'web']
web -> ['wiktionary', 'web', 'mean', 'insect', 'common', 'short', 'world', 'wide',
date -> ['word', 'date', 'mean', 'day', 'fruit', 'wikt', 'people', 'person', 'go']
http -> ['redirect']
title -> ['title', 'social', 'status', 'royal', 'church', 'rank', 'names']
https -> ['redirect']
cite -> ['redirect', 'citation']
archive -> ['archive', 'term', 'location', 'cite', 'web', 'url', 'http', 'www', 'li
jpg -> ['redirect']
references -> ['redirect', 'citation', 'word']
people -> ['redirect']
website -> ['www', 'wikipedia', 'png', 'thumb', 'website', 'picture', 'http', 'set'
thumb -> ['redirect']
small -> ['redirect', 'size']
publisher -> ['redirect', 'publishing']
image -> ['redirect', 'picture']
redirect -> ['pages', 'wikipedia', 'see', 'help', 'redirect', 'line', 'law', 'url',
svg -> ['redirect']
br -> ['redirect']
```

图 1.4 词汇跳转关系 title_to_keys.txt 文件

1.4 实验总结

本次实验我理解了 map-reduce 算法思想与流程，能应用 map-reduce 思想解决实际问题。在面对海量的数据时，传统算法往往开销极大，效率极低，导致耗时无法满足程序的需求；而 map-reduce 算法采用分布式的架构，可同时在多台设备上对大数据进行运算及处理，从而完美应对数据量之大带来的挑战。

此外，combine 过程和 shuffle 过程的加入使得 map-reduce 算法更加高效，本次实验对于这两个过程的应用能有效减少算法整体运行时间，提高算法效率。

尽管本次实验的数据集相比真实应用场景的海量数据集而言微不足道，map-reduce 算法、combine 过程和 shuffle 过程在性能上的优化可能无法立竿见影地体现出来，但本实验从 0 到 1 对 map-reduce 算法的设计与实现，也让我对算法的思想与流程有了更为深刻认识与见解，并最终通过 map-reduce 算法很好地实现了对小型场景实际问题的求解，收获满满的同时成就感倍增。

2 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

利用实验一得到的出现次数最多前 1000 个的 title 之间的引用关系<title,<title1,…,titlek>>，由 title 为节点构造有向图，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 title 及其对应的 pagerank 值。

2.3 实验过程

2.3.1 编程思路

pagerank 算法的主要流程包括由跳转关系构建有向图、构建概率转移矩阵 M、使用幂迭代法计算每个 title 对应网页的 pagerank 值和加入 teleport β 来对概率转移矩阵进行修正等。

pagerank 算法构建得到的概率转移示意图如图 2.1 所示：

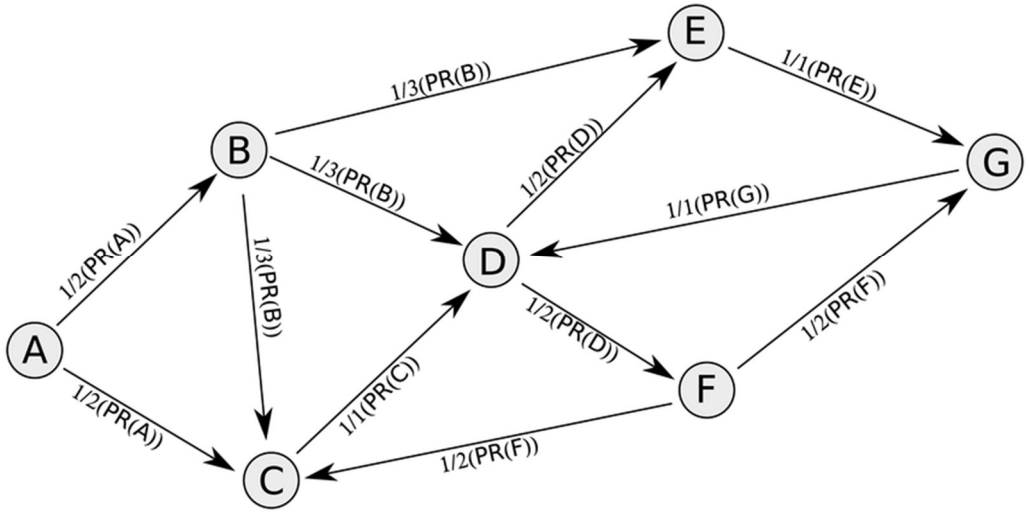

图 2.1 pagerank 算法概率转移示意图

（1）构建有向图

实验 1 中已将前 1000 个词汇的列表序列化到了文件，此时只需要对该文件进行反序列化即可得到前 1000 个词汇的列表对象。在得到该列表后，依据列表中各词汇的跳转关系构建有向图，具体方法如下：

先初始化大小为 1000×1000 的零矩阵表示有向图，以及列表中词汇到索引的字典 word_to_index。遍历列表中的每个词汇 title，再遍历 title 对应的跳转关系列表，遍历到的每

个 key 表明有向图中存在 title→key 的跳转关系，此时以 title 对应的索引作为列号，key 对应的索引作为行号，将该点在矩阵中的值设为 1。前 1000 个词汇的列表遍历结束时，有向图便构建完成。

（2）构建概率转移矩阵 M

概率转移矩阵 M 由（1）中构建好的有向图转化而来，即对有向图中每个 title 跳转到其跳转列表中各 key 的概率计算得来。由于每个 title 跳转到各 key 的概率均相同，因此概率转移矩阵在有向图的基础上，对矩阵中每一列上值为 1 的点除以该列值为 1 的点的个数，计算结果表示 title 跳转到各 key 的概率，从而得到最终的概率转移矩阵。

（3）使用幂迭代法计算 pagerank

在得到概率转移矩阵 M 后，根据 M 对各 title 对应网页的 pagerank 进行计算。首先初始化大小为 1000 的向量表示各 title 对应网页的 pagerank，所有 pagerank 的总和为 1，每个 pagerank 的起始值相同，即$1/1000$（或满足总和为 1 的其他值亦可，对最终结果基本无影响，但在一定程度上关系着幂迭代法的运行时间）。接下来使用幂迭代法计算哥 title 对应网页的 pagerank，公式如下：

$$v' = Mv$$

其中v'为一次迭代后的 pagerank 向量，M为概率转移矩阵，v为迭代前的 pagerank 向量。经过若干次迭代，当一次迭代前后v'和v之差的大小（即矩阵的二范数）小于阈值ε时，幂迭代法停止，此时 pagerank 向量即为最终各 title 对应网页的 pagerank 值。算法的结束条件如下公式所示：

$$\|v' - v\|^2 < \varepsilon$$

可对求解得到的 pagerank 向量各项累加求和，判断所有网页 pagerank 之和是否为 1，从而对算法的正确性进行初步验证。

最后，遍历计算出的 pagerank 向量，按前 1000 个词汇的列表顺序得到每个词汇对应网页的 pagerank，形成一个新的列表。接着按 pagerank 由大到小对该列表进行排序，并将排序后各词汇对应网页的 pagerank 依次写入到文本文件中，便于查看与验证结果。

（4）加入随机跳转

为避免 dead ends（一个结点只有入度而没有出度）和 spider trap（有向图中形成环）等不利现象的产生，在计算 pagerank 时可加入随机跳转β，即网页每次跳转时有$(1-\beta)$的概率随机跳转到任意一个网页，此时不会按照有向图中的既定路线进行跳转，从而有效化解了 dead ends 和 spider trap 的问题。

加入随机跳转β只需对（3）中幂迭代法的公式进行简单的修正，修正后的公式如下：

$$v' = \beta M v + [\frac{1-\beta}{N}]_N$$

其中β为阻尼系数（本实验设为 0.85），$1-\beta$表示随机跳转发生的概率，向量$[1/N]_N$表示随机跳转时，跳转到任一网页的概率均为$1/N$，其中 N 为所有网页的总个数，本实验中即 1000。修正后的公式能有效地对概率转移矩阵进行修正，使用该公式再次利用幂迭代法对 pagerank 进行计算，得到的 pagerank 与（3）中应有所差异。

最后，遍历计算出的修正后的 pagerank 向量，按前 1000 个词汇的列表顺序得到每个词汇对应网页的 pagerank，形成一个新的列表。接着按 pagerank 由大到小对该列表进行排序，并将排序后各词汇对应网页的 pagerank 依次写入到不同于（3）的文本文件中，便于查看并与（3）中 pagerank 的结果进行比较。

2.3.2 遇到的问题及解决方式

（1）跳转关系处理

由于只考虑前 1000 个词汇之间的跳转关系，因此对于每个词汇的跳转关系列表，应筛

除前 1000 个词汇以外的词汇。由于有向图中每个词汇出现且仅出现一次，因此对于每个词汇的跳转关系列表，还应去除其中可能包含的重复词汇，避免后续计算时出现错误。

此外，应注意跳转关系列表的跳转方向，为拥有该跳转关系列表的 title 跳转到跳转关系列表中的各个 key，即有向图中包含一个 title→key 的边，且在有向图的二维矩阵中表示为点(key 的索引, title 的索引)的值为 1。

（2）运行效率

在构建有向图的过程中，遍历前 1000 个词汇的列表时，得到 title 与其对应的跳转关系列表中各个 key 后，需在有向图的二维矩阵中对 title 与 key 所对应的点的值进行标记，因此需得到 title 与 key 在列表中的索引。如果每次用列表的 index 方法对索引进行获取，其原理为遍历整个列表查找元素在列表中的位置，时间复杂度为$O(n)$，效率相对较低；而事先构建词汇到索引的映射字典，每次使用该字典得到 title 与 key 在列表中的索引，时间复杂度为$O(1)$，效率大幅提升。

2.3.3 实验测试与结果分析

程序事先对 pagerank 算法的各过程进行封装，接着在 main 函数中依次执行各过程，控制台会打印相关提示信息，如图 2.2 所示：

```
C:\Users\LSY\AppData\Local\Programs\Python\P
Pagerank iteration starts!
Pagerank iteration done!
sum of Pagerank: 1.0

Process finished with exit code 0
```

图 2.2 控制台输出相关提示信息

在幂迭代法结束后，程序对求解出的 pagerank 向量中各项累加求和，得到所有网页 pagerank 的总和为 1.0，初步验证了算法的正确性。

此时，实验结果能界面友好地呈现在相应的文本文件中，其中各词汇按对应网页的 pagerank 由大到小进行排序。未使用随机跳转和使用随机跳转时，各词汇对应网页的 pagerank 求解结果分别如图 2.3、图 2.4 所示：

pagerank.txt × pagerank_with_beta.txt

```
1  redirect : 0.2211462526636635
2  help : 0.03846395706736062
3  page : 0.03382222797918228
4  pages : 0.024271674801697896
5  web : 0.02395351413971218
6  world : 0.022969574636392272
7  see : 0.0210904488768899
8  wide : 0.02085375852163335
9  url : 0.020718063565420504
10 wikipedia : 0.019688354225675435
11 line : 0.019628919275503424
12 law : 0.019513539531292538
13 width : 0.01200185468484331
14 word : 0.010059429370683808
15 number : 0.00971553957992874
16 mean : 0.00905950274302306
17 wiktionary : 0.007694195912675196
18 common : 0.006506089181355442
19 people : 0.00639153816983081
20 english : 0.006116787382480723
```

图 2.3 未使用随机跳转时的 pagerank

pagerank.txt pagerank_with_beta.txt ×

```
1  redirect : 0.1868098995372561
2  help : 0.024594036343878314
3  page : 0.02361670783849998
4  pages : 0.018208224065384627
5  web : 0.017579649767466943
6  world : 0.016957177416108346
7  see : 0.015558687350470949
8  url : 0.015471109041340473
9  wide : 0.01507845320971818
10 law : 0.014598762106554995
11 wikipedia : 0.014446058955791123
12 line : 0.014325103966989427
13 mean : 0.008550484984507285
14 number : 0.008364186181328724
15 word : 0.00830433883743388
16 width : 0.007577043024041581
17 wiktionary : 0.006970927372533482
18 people : 0.006479304791919125
19 english : 0.005841509359934936
20 one : 0.004937390935114978
```

图 2.4 使用随机跳转时的 pagerank

由使用随机跳转前后的 pagerank 结果对比可知，随机跳转使网页中 pagerank 的分布更加均衡，pagerank 较大的网页在使用随机跳转后 pagerank 普遍减小，反映 pagerank 较小的部分网页在使用随机跳转后 pagerank 得到了提升。说明随机跳转能很好应对 dead end 和

spider trap 等现象，使受该现象影响的网页的 pagerank 得到一定程度的补偿。

2.4 实验总结

本次实验我学习了 pagerank 算法并熟悉其推导过程，了解了由跳转关系构建有向图，再由有向图构建概率转移矩阵，最后通过幂迭代法求解 pagerank 的一系列流程。同时，自行实现了完整 pagerank 算法，将 pagerank 算法运用于实际问题的求解，并对实验结果进行分析，加深了自己对算法的认识与理解。

除了基础的 pagerank 算法外，还通过引入阻尼系数β，在网页中加入了随机跳转，从而有效地避免了 dead end 和 spider trap 等不利现象的产生，进一步对 pagerank 的求解进行了优化。最终将未使用随机跳转和使用随机跳转求解得到的 pagerank 分别呈现在不同的文本文件中，通过对两个实验结果的横纵对比，我对随机跳转在 pagerank 算法中的概念与作用有了更深刻的认知。

然而，在实现 pagerank 算法的过程也遇到了不少挫折，比如最初计算得到的 pagerank 值总是偏小，并且各网页的 pagerank 之和远小于 1。经过不断的调试和仔细地排查，终于发现了问题所在，即构建有向图时行坐标和列坐标混淆导致。当将这处关键问题修改正确后，程序的运行结果页随之正确，最终如愿得到与预期一致的 pagerank 结果。这也再次印证了算法的设计与实现过程中需要我们格外细心，在发现问题时也要保持镇定，一步一步使自己的程序得到完善和优化。

3 关系挖掘实验

3.1 实验目的

1、学习 Apriori 算法并熟悉其执行过程；
2、实现 Apriori 算法，理解频繁项集和关联规则的概念；
3、将 Apriori 算法运用于实际，并对结果进行分析。
4、使用 PCY 算法对二阶频繁项集的计算阶段进行优化。

3.2 实验内容

（1）Apriori 算法

编程实现 Apriori 算法，从实验一中得到的前 1000 个 title 及其引用关系数据为<<title,<title1,…,titlek>>,…>，将其处理为<<title,title1,…,titlek>,…>作为算法输入。

频繁项集的最小归一化支持度为 0.15（与 basket 总数的比值），关联规则的最小置信度为 0.3。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。

（2）PCY 算法

在 Apriori 算法的基础上，要求使用 PCY 算法对二阶频繁项集的计算阶段进行优化。

频繁项集的最小支持度为 0.15，关联规则的最小置信度为 0.3。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 算法中的 vector 的值，以 bit 位的形式输出。

3.3 实验过程

3.3.1 编程思路

APriori 算法是一个持续迭代的过程，核心步骤为生成候选项集和计算频繁项集与关联规则，二者不断交替，直到算法满足终止条件退出。PCY 算法在 APriori 算法的基础上，对二阶频繁项集的计算阶段进行优化。引入频繁桶的概念，预先对候选项中的非频繁项进行筛除，从而优化筛选频繁项集时的运行效率。

APriori 算法的原理流程示意图如图 3.1 所示：

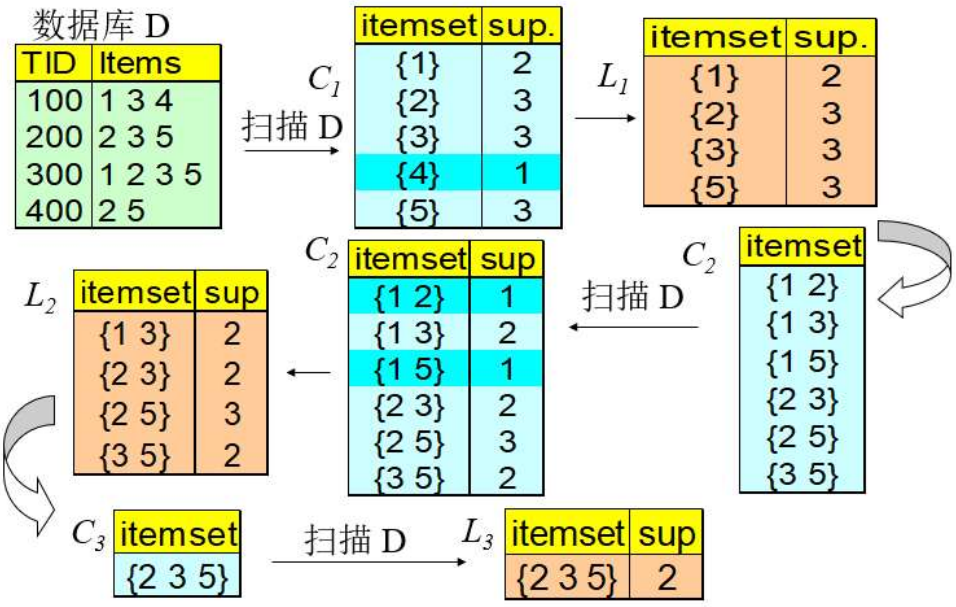

图 3.1 APriori 算法的原理流程示意图

（1）数据处理

数据集采用的是实验一中前 1000 个词汇的列表，列表中每项为<title, relations>的键值对，其中 relations 为 title 的跳转关系列表。首先将前 1000 个词汇的列表中每项的 title 与 relations 列表合并形成新的列表，然后将合并后的列表转换为集合<title, title1, …, titlek>，作为一个“篮子”。Apriori 算法的输入即为由所有 1000 个“篮子”组成的列表。

由于列表转换为集合时会自动去重，因此在将 title 与 relations 列表合并时只需简单地将 title 添加到 relations 列表中即可。

（2）生成一阶候选项集

一阶候选项集为 1000 个“篮子”中所有词汇自身包装而来的一个集合组成的列表，即每个一阶候选项为只包含一个词汇的集合。先初始化一个空集合，然后遍历所有“篮子”，依次与该集合取并集，最终得到所有不重复词汇组成的集合，再将该集合转换为列表，依次遍历列表中的每个词汇，将每个词汇包装成一个集合，加入到新的列表中。最终该列表即为生成的一阶候选项集。

（3）计算一阶频繁项集

对一阶候选项集进行遍历，判断遍历到的每个一阶候选项是否为频繁项。具体方法为对遍历到的每个一阶候选项，遍历所有“篮子”列表，统计该一阶候选项是其他“篮子”的子集的个数，如果最终该数目与总“篮子”个数的比值，即该候选项的归一化支持度，不少于实验规定的最小支持度 0.15，则将其以<itemset, support>键值对的形式加入到频繁项集中，否则舍弃。其中 itemset 为该候选项，support 为该候选项的支持度。

由于一阶频繁项集是得到高阶候选项集的基础，因此将一阶频繁项集保存在内存中，以便后续使用。为方便查看计算得到的频繁项集，将频繁项集按支持度由大到小对频繁项进行排序，并依次写入到一阶频繁项集对应的文本文件中。

（4）生成 k 阶候选项集（$k > 1$）

当阶数大于 1 时，通过对当前得到的最新频繁项集（即上一阶的频繁项集）与一阶频繁项集进行组合生成当前阶的候选项集。在组合时，只保留组合后长度与当前阶数相同的候选项，这是因为如果组合过程中候选项中出现相同元素会自动去重，候选项的长度仍不变，比当前阶数少 1。当所有遍历完所有的组合可能后，保留下的候选项组成的列表即为当前阶的候选项集。

（5）计算 k 阶频繁项集（$k > 1$）

计算 k 阶频繁项集的过程与计算一阶频繁项集的过程基本相似，即对 k 阶候选项集进行遍历，得到每个 k 阶候选项后，遍历所有“篮子”列表，统计该 k 阶候选项是其他“篮子”子集的个数，如果最终该候选项的归一化支持度不少于 0.15，则加入到 k 阶频繁项集中，否则舍弃。

为方便查看计算得到的频繁项集，将频繁项集按支持度由大到小对频繁项进行排序，并依次写入到 k 阶频繁项集对应的文本文件中。

（6）挖掘 k 阶关联规则（$k > 1$）

计算得到 k 阶频繁项集后，通过频繁项集对其中的关联规则进行挖掘。频繁项集的每个频繁项中，一个真子集蕴含其补集即为一个可能的关联规则，表达式如下所示，其中 I 是一个频繁项集，A 表示该频繁项集的一个真子集：

$$A \rightarrow I - A$$

通过排列组合生成所有可能的关联规则，根据频繁项的支持度与关联规则左部集合的支持度的比值对关联规则的置信度进行计算，如果不少于实验规定的最小置信度 0.3，则将其以<rule, confidence>键值对的形式加入到 k 阶关联规则列表中，否则舍弃。其中 rule 为该关联规则，confidence 为该关联规则的置信度。

由于计算置信度时需获取相关集合的支持度，因此在计算频繁项集的过程中，需额外维护一个频繁项到支持度的映射字典，将满足条件的候选项（即频繁项）及其支持度加入到该字典中，以便后续挖掘关联规则时取用。

为方便查看挖掘得到的关联规则，按置信度由大到小对关联规则进行排序，并依次写入到 k 阶关联规则对应的文本文件中。

（7）算法迭代

APriori 算法先执行（1）（2）（3）步操作，作为后续迭代的基础。当阶数大于 1 时，

循环执行（4）（5）（6）步操作，持续迭代，直到算法满足终止条件退出。

如果算法规定了最大的阶数，如本实验只要求输出 1~4 阶的频繁项集与关联规则，则算法迭代到 4 阶后便不再迭代，循环终止。

如果算法没有规定最大的阶数，则持续迭代直到某阶无法计算得到新的频繁项集，表明所有频繁项集及关联规则已经挖掘完毕，循环终止。

（8）PCY 算法

PCY 算法在 APriori 算法的基础上，对二阶频繁项集的计算阶段进行优化。在计算得到一阶频繁项集后，生成二阶频繁项集前，首先初始化若干个编号从 0 开始依次递增、计数为 0 的桶，将所有“篮子”中的物品两两组合，根据特定的哈希函数将物品组合哈希到对应的桶中，此时对应桶的计数加 1。初始化一个大小为桶的个数的零向量位图，所有物品组合哈希完毕后，遍历创建的所有桶，如果桶的计数与总“篮子”个数的比值不少于实验规定的最小支持度 0.15，则该桶为频繁桶，并在位图中将桶对应位置的值置为 1。

在生成二阶频繁项集时，遍历（4）中得到的候选项集，根据哈希函数将得到的每个二阶候选项哈希到对应的桶中，如果该桶在位图中的位置值为 1，即为频繁桶，则保留该二阶候选项，否则舍弃。这样可以对哈希值对应非频繁桶的二阶候选项进行初筛，因为非频繁桶中的候选项一定为非频繁项。而保留下来的候选项也并非一定为频繁项，还需进一步通过（5）的操作计算得到最终的二阶频繁项集，但 PCY 算法对非频繁项的初筛在一定程度上可以缓解（5）中遍历带来的大开销与长耗时，从而对二阶频繁项集的计算进行优化。

3.3.2 遇到的问题及解决方式

（1）频繁项作为字典的键

为便于算法挖掘关联规则，在得到频繁项后需将频繁项到支持度的映射保存在字典中，此时需用频繁项作为字典的键，而频繁项是集合类型，由于 Python 语言中集合类型的可变性，不支持以集合作为字典的键，而仅支持以不可变类型的变量作为字典的键。

对于这个问题，有两种解决方案：一是在将频繁项转换为字符串后作为键保存到字典中，此时需注意虽然集合是无序的，但是转换成的字符串是有序的，因此转换前要先对集合进行排序，而每次从字典中取值时，将频繁项转换成字符串前也需要先对集合进行排序；二是用 FrozenSet 类型替代 Set 类型来作为频繁项的数据结构，FrozenSet 是不可变的集合，因此可以直接用作字典的键。

对比上述两种解决方案，前者较为繁琐，且容易出错，而后者则明显更加简便快捷，因此最终选用 FrozenSet 类型替代 Set 类型作为频繁项的数据结构，从而实现以频繁项作为字典的键进行存取的功能。

（2）PCY 算法哈希函数设计

PCY 算法中哈希函数的设计关系到频繁桶的分布，进而影响算法的效率。如果一个哈希函数性能较差，即发生哈希碰撞的概率较高，会导致最终要么难以得到非频繁桶，要么频繁桶极其集中，从而使绝大多数候选项都哈希到频繁桶中，难以较好地发挥其筛选非频繁项的作用，故无法体现 PCY 算法的优化效果。

因此，在设计哈希函数时，需要多次尝试不同的哈希函数，使不同候选项在哈希函数上的分布尽可能随机。每当尝试一个哈希函数时，在控制台打印根据该哈希函数生成的桶的位图，观察位图上频繁桶和非频繁桶的分布，从而对哈希函数的性能进行评估，最终挑选性能最好的哈希函数作为 PCY 算法的哈希函数。

3.3.3 实验测试与结果分析

程序具有交互界面，可以选择要执行的算法（APriori 或 PCY）。当选择 APriori 算法或 PCY 算法后，程序开始执行，控制台会分别打印相关提示信息，如图 3.2、图 3.3 所示：

```
C:\Users\LSY\AppData\Local\Programs\Python\Python39\python.exe
Please choose an algorithm to run:
1. APriori      2. PCY
1
APriori starts!
Write 1-itemset starts!
Write 1-itemset done!
Write 2-itemset starts!
Write 2-itemset done!
Write 3-itemset starts!
Write 3-itemset done!
Write 4-itemset starts!
Write 4-itemset done!
APriori is done with the maximum of k is 4!
Run time: 1.18s

Process finished with exit code 0
```

图 3.2 执行 APriori 算法控制台输出相关提示信息

```
C:\Users\LSY\AppData\Local\Programs\Python\Python39\python.exe
Please choose an algorithm to run:
1. APriori      2. PCY
2
PCY starts!
Write 1-itemset starts!
Buckets bitmap: [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
Write 1-itemset done!
Write 2-itemset starts!
Write 2-itemset done!
Write 3-itemset starts!
Write 3-itemset done!
Write 4-itemset starts!
Write 4-itemset done!
PCY is done with the maximum of k is 4!
Run time: 1.47s

Process finished with exit code 0
```

图 3.3 执行 PCY 算法控制台输出相关提示信息

比较两个算法的运行时间，发现 APriori 算法的运行速度比 PCY 算法略快，而理论上 PCY 算法在二阶频繁项集的计算阶段进行了优化，理应有更高的效率。出现这种现象的原因是本次实验的数据集规模较小，且哈希函数设计性能有限，PCY 算法对于非频繁项的筛选程度较小；而算法在生成频繁桶与非频繁桶的过程中，又进行了大量的排列组合操作，造成了更多时间损耗，算法在本次实验带来的优化还不足以弥补这些损耗，最终导致运行时间比改进前的 APriori 算法略长。

此外，观察到执行 PCY 算法时控制台输出了桶的位图，由位图可以清晰直观地看到频繁桶与非频繁桶的分布，本实验所采用的哈希函数计算得到的频繁桶分布较为均匀，说明

PCY 算法中哈希函数的设计较为成功。

当程序执行完毕后，每一阶计算得到的频繁项集和关联规则会保存到相应的文本文件中。执行 APriori 算法后，观察相应文本文件，得到的一到四阶的频繁项集如图 3.4~图 3.7 所示：

1-itemset.txt

```
total: 35
{'redirect'}: 0.37
{'people'}: 0.337
{'one'}: 0.307
{'thumb'}: 0.287
{'pages'}: 0.27
{'references'}: 0.263
{'jpg'}: 0.253
{'title'}: 0.248
{'english'}: 0.248
{'date'}: 0.245
{'url'}: 0.235
{'https'}: 0.231
{'web'}: 0.231
{'www'}: 0.231
{'different'}: 0.228
{'like'}: 0.224
{'cite'}: 0.22
{'example'}: 0.209
{'http'}: 0.202
```

图 3.4 一阶频繁项集

2-rules.txt

```
total: 204
{'thumb'} -> {'jpg'}: 0.81
{'jpg'} -> {'thumb'}: 0.92
{'title'} -> {'cite'}: 0.88
{'cite'} -> {'title'}: 1.00
{'title'} -> {'url'}: 0.88
{'url'} -> {'title'}: 0.92
{'https'} -> {'url'}: 0.93
{'url'} -> {'https'}: 0.91
{'url'} -> {'cite'}: 0.91
{'cite'} -> {'url'}: 0.98
{'title'} -> {'references'}: 0.86
{'references'} -> {'title'}: 0.81
{'web'} -> {'url'}: 0.92
{'url'} -> {'web'}: 0.91
{'url'} -> {'references'}: 0.90
{'references'} -> {'url'}: 0.81
{'date'} -> {'url'}: 0.86
{'url'} -> {'date'}: 0.90
{'https'} -> {'references'}: 0.91
```

图 3.5 二阶频繁项集

3-itemset.txt

```
total: 136
{'title', 'url', 'cite'}: 0.215
{'title', 'references', 'cite'}: 0.206
{'title', 'references', 'url'}: 0.205
{'title', 'https', 'url'}: 0.204
{'title', 'https', 'cite'}: 0.203
{'references', 'cite', 'url'}: 0.203
{'https', 'date', 'url'}: 0.202
{'https', 'cite', 'url'}: 0.202
{'web', 'https', 'url'}: 0.2
{'title', 'web', 'url'}: 0.199
{'www', 'https', 'url'}: 0.199
{'title', 'date', 'cite'}: 0.198
{'title', 'web', 'cite'}: 0.198
{'title', 'date', 'url'}: 0.198
{'references', 'https', 'url'}: 0.198
{'web', 'cite', 'url'}: 0.198
{'title', 'www', 'url'}: 0.197
{'web', 'date', 'url'}: 0.197
{'title', 'www', 'cite'}: 0.196
```

图 3.6 三阶频繁项集

4-itemset.txt

```
total: 124
{'title', 'url', 'references', 'cite'}: 0.203
{'title', 'url', 'https', 'cite'}: 0.202
{'title', 'url', 'web', 'cite'}: 0.198
{'title', 'url', 'date', 'cite'}: 0.196
{'title', 'url', 'www', 'cite'}: 0.195
{'https', 'title', 'references', 'url'}: 0.193
{'https', 'title', 'references', 'cite'}: 0.192
{'web', 'https', 'date', 'url'}: 0.192
{'title', 'https', 'date', 'url'}: 0.191
{'https', 'references', 'cite', 'url'}: 0.191
{'title', 'references', 'date', 'url'}: 0.19
{'title', 'www', 'https', 'url'}: 0.19
{'title', 'references', 'date', 'cite'}: 0.189
{'web', 'title', 'https', 'url'}: 0.189
{'title', 'https', 'date', 'cite'}: 0.189
{'title', 'www', 'https', 'cite'}: 0.189
{'references', 'https', 'date', 'url'}: 0.189
{'www', 'https', 'date', 'url'}: 0.189
{'cite', 'https', 'date', 'url'}: 0.189
```

图 3.7 四阶频繁项集

将四个频繁项集的结果与参考答案进行比较，发现结果准确无误，说明 APriori 算法的设计与实现正确。而随着阶数增大，各阶频繁项集中频繁项的个数呈现先增多后减少的变化趋势，二阶频繁项的个数最多，说明二阶物品组合的可能较多，二阶频繁项在“篮子”中的出现频率也较高。

接下来再观察执行 APriori 算法得到的二到四阶关联规则，如图 3.8~3.10 所示：

2-rules.txt

```
total: 204
{'thumb'} -> {'jpg'}: 0.81
{'jpg'} -> {'thumb'}: 0.92
{'title'} -> {'cite'}: 0.88
{'cite'} -> {'title'}: 1.00
{'title'} -> {'url'}: 0.88
{'url'} -> {'title'}: 0.92
{'https'} -> {'url'}: 0.93
{'url'} -> {'https'}: 0.91
{'url'} -> {'cite'}: 0.91
{'cite'} -> {'url'}: 0.98
{'title'} -> {'references'}: 0.86
{'references'} -> {'title'}: 0.81
{'web'} -> {'url'}: 0.92
{'url'} -> {'web'}: 0.91
{'url'} -> {'references'}: 0.90
{'references'} -> {'url'}: 0.81
{'date'} -> {'url'}: 0.86
{'url'} -> {'date'}: 0.90
{'https'} -> {'references'}: 0.91
```

图 3.8 二阶关联规则

3-rules.txt

```
total: 816
{'title'} -> {'cite', 'url'}: 0.87
{'url'} -> {'title', 'cite'}: 0.91
{'cite'} -> {'title', 'url'}: 0.98
{'title', 'url'} -> {'cite'}: 0.99
{'title', 'cite'} -> {'url'}: 0.98
{'url', 'cite'} -> {'title'}: 1.00
{'title'} -> {'cite', 'references'}: 0.83
{'references'} -> {'title', 'cite'}: 0.78
{'cite'} -> {'title', 'references'}: 0.94
{'title', 'references'} -> {'cite'}: 0.97
{'title', 'cite'} -> {'references'}: 0.94
{'references', 'cite'} -> {'title'}: 1.00
{'title'} -> {'references', 'url'}: 0.83
{'url'} -> {'title', 'references'}: 0.87
{'references'} -> {'title', 'url'}: 0.78
{'title', 'url'} -> {'references'}: 0.94
{'title', 'references'} -> {'url'}: 0.96
{'url', 'references'} -> {'title'}: 0.97
{'title'} -> {'https', 'url'}: 0.82
```

图 3.9 三阶关联规则

4-rules.txt

```
total: 1240
{'title'} -> {'references', 'cite', 'url'}: 0.82
{'url'} -> {'title', 'cite', 'references'}: 0.86
{'references'} -> {'title', 'cite', 'url'}: 0.77
{'cite'} -> {'title', 'references', 'url'}: 0.92
{'title', 'url'} -> {'cite', 'references'}: 0.94
{'title', 'references'} -> {'cite', 'url'}: 0.95
{'title', 'cite'} -> {'references', 'url'}: 0.93
{'url', 'references'} -> {'title', 'cite'}: 0.96
{'url', 'cite'} -> {'title', 'references'}: 0.94
{'references', 'cite'} -> {'title', 'url'}: 0.99
{'title'} -> {'https', 'cite', 'url'}: 0.81
{'url'} -> {'title', 'https', 'cite'}: 0.86
{'https'} -> {'title', 'cite', 'url'}: 0.87
{'cite'} -> {'title', 'https', 'url'}: 0.92
{'title', 'url'} -> {'https', 'cite'}: 0.93
{'title', 'https'} -> {'cite', 'url'}: 0.98
{'title', 'cite'} -> {'https', 'url'}: 0.92
{'https', 'url'} -> {'title', 'cite'}: 0.94
{'url', 'cite'} -> {'title', 'https'}: 0.94
```

图 3.10 四阶关联规则

将三个关联规则与参考答案进行比较，发现结果准确无误，说明关联规则挖掘算法的设计与实现正确。由测试结果可以看出，随着阶数的增大，各阶关联规则的数量也不断增大，说明随着频繁项中物品个数的增加，关联规则的组合更多，关联规则的挖掘也更容易，从而可以挖掘出更多置信度在实验允许范围内的关联规则。

执行 PCY 算法后分别查看记录得到的一到四阶频繁项集与二到四阶关联规则，发现结果与 APriori 算法得到的结果完全一致，说明 PCY 算法的设计与实现正确，PCY 对 APriori 的改动只体现在算法执行效率上，而不影响算法的最终执行结果。

3.4 实验总结

本次实验我学习了 APriori 算法和 PCY 算法的思想及流程，比较并了解了二者在算法实现上的差异与优劣，并将算法运用于实际问题的解决中，在独立设计与实现算法的过程中，逐渐对这两个算法的精妙之处有了深刻的理解与感悟，最终圆满完成了实验任务。

在实现 APriori 算法的过程中，由于数据集是由 1000 个“篮子”组成的列表，每个“篮子”又是由若干词汇组成的集合，数据类型较为多样，数据间的关联也较为复杂，因此算法的编写需要有极高的细心与耐心。尤其是生成候选项集的步骤，用到了组合计数的策略，这就涉及到对不同数据结构的存取，同时又要求所有组合情况的完备，既不能遗漏，也不能重复，因此极容易出现纰漏。

在实现 PCY 算法的过程中，哈希函数的设计尤为重要，而哈希函数的具体实现需要先在脑中构思，再在程序中实现，最后通过测试检验其性能。因此，为得到一个性能良好的哈希函数，我也设想了大量的哈希函数方案，并逐个在程序中验证其性能的好坏，耗费了不少时间。在大量努力尝试下，一个性能较好的哈希函数最终被我选定。

最后，经过一点点地调试与优化，两个算法均能正常运行，并有着一致且正确的输出。通过对实验结果的比较与分析，我总结得到了两个算法的优劣，也对关系挖掘算法的思想有了更为全面的认知与见解，收获颇丰。

4 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,探究聚类算法原理；
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

提供动漫得分数据集（anime.csv）,包含用户对动漫评分(Score 2~Score 10)、动漫的欢迎程度(Popularity)等数据。

在对数据集进行处理时，按照 Popularity 列进行降序排序，在其中选择 K 类（eg. 选择 Popularity 高、中、低三类），每类选择一定数量的数据（eg. 每类选择 60 个数据），将选出的 K 类数据的 K 作为标签与 Popularity 和 Score2~Score10 组合成一个 11 维的数据，对除 K 以外的数据进行归一化处理。

编写 kmeans 算法，算法的输入是归一化后的数据集，动漫数据集一共 11 维数据，代表着动漫的 11 维特征，请在欧式距离下对动漫的所有数据进行聚类，聚类的数量为 K。

以处理后的 anime.csv 作为输入文件。

在本次实验中，最终评价 kmeans 算法的精准度有两种，第一是处理后的动漫数据集已经给出的 K 个聚类，和自己运行的 K 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。

进阶任务：在聚类之后，任选两个维度（为了效果良好建议选择 Score 10 和 Score 2 列数据进行展示），以 K 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所有的样本点。效果展示图如图 4.1 所示：

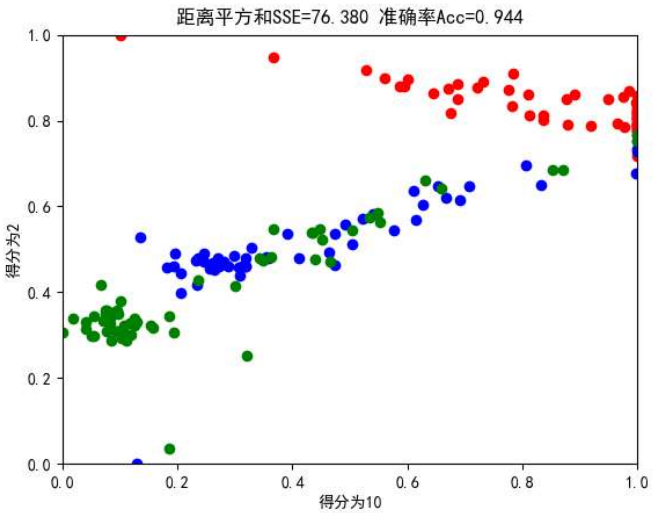

图 4.1 聚类效果展示图

4.3 实验过程

4.3.1 编程思路

kmeans 算法是聚类算法的一种，其核心步骤为：

a）先初始化 K 个簇中心；

b）将数据集中的每个点根据点到 K 个簇中心的欧氏距离计算距离其最近的簇，并将点纳入该簇中；

c）根据得到的 K 个簇重新计算每个簇的簇中心；

d）循环执行（a）（b）操作，直到算法满足终止条件退出。终止条件即执行（3）后每个簇的簇中心相比原来均没有改变，说明聚类结束。

kmeans 算法的原理流程示意图如图 4.2 所示：

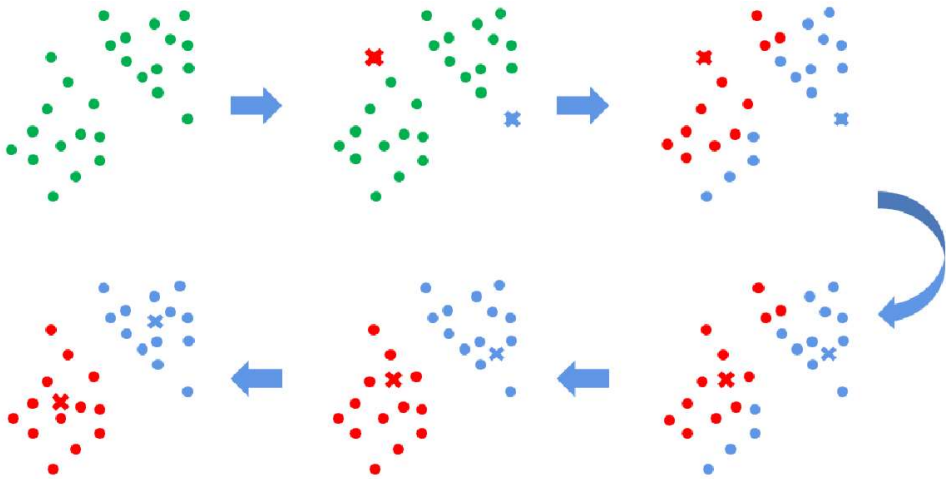

图 4.2 kmeans 算法原理流程示意图

（1）数据集

数据集是若干个经归一化处理后的动漫得分，每条动漫为一个 11 维向量，可以视为 11 维空间中的一个点。动漫欢迎程度分为 3 类，每条动漫的第 1 维为该动漫的欢迎程度，用作供测试时比对的标签，后 10 维为该动漫不同得分的数目，用作聚类的训练集。

聚类算法迭代结束后，数据集将被分为 3 类，表示不同的欢迎程度，将每条动漫的预测欢迎程度与真实欢迎程度比较即可对聚类算法的性能进行评估。

（2）初始化 K 个簇中心

初始化 K 个簇中心主要有两种方法：一是直接从数据集中随机抽样得到 K 个点，分别作为 K 个簇的簇中心；二是先从数据集中随机选取一个点作为第一个簇中心，然后再从数据集中选取距离第一个簇中心最远的点作为第二个簇中心，接着从数据集中选取距离前两个簇中心最小距离最远的点作为第三个簇中心，以此类推，直到得到 K 个簇中心。

本实验对两种初始化方法均有实现，在测试时可根据聚类算法的性能比较二者的优劣。

（3）更新数据集中的点到最近的簇

初始化得到 K 个簇中心后，遍历数据集中的所有点，根据每个点到各个簇中心的欧氏距离计算每个点到各个簇的距离，并将点纳入距离最近的簇中。

（4）更新 K 个簇中心

更新数据集中的点到最近的簇后，得到分别包含若干点的 K 个簇，接下来对 K 个簇的簇中心进行更新，以每个簇中所有数据点的均值作为该簇新的簇中心。

（5）算法迭代

为得到最终的 K 个簇，接下来 kmeans 算法进入模型拟合过程，循环执行（3）（4）两步操作，直到算法满足终止条件退出。算法的终止条件为当一次更新 K 个簇中心前后，K 个簇中心相比原来均没有改变，则聚类算法结束。此时模型拟合完毕，得到的 K 个簇即为最终数据集划分成的 K 个不同类。

（6）模型评估

kmeans 算法执行完毕后，需对构建好的模型，即数据集划分成的 K 个不同类进行评估来得知模型性能的好坏。评价指标包括准确率和 SSE（误差平方和）两种。其中准确率指数据集中预测正确（即预测类别与真实类别相同）的点占数据集所有点的百分比，SSE 指数据集中所有数据点到各自所在簇的簇中心的距离的平方和。因此，准确率越高，SSE 越小，则预测效果越好，模型性能越出色。

（7）结果展示

kmeans 算法执行完毕，且模型评估结束后，任选两个维度（为效果较好本实验选择 Score10 和 Score2 两个维度，维度跨度较大，数据点对比更强烈），以 K 种不同的颜色对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所有的样本点，使模型更清晰直观。同时，在点图上方标注聚类算法的准确率及 SSE 两种平均指标，作为最终的结果展示。

4.3.2 遇到的问题及解决方式

（1）准确率计算

计算准确率时，由于聚类算法仅将所有数据点分成 K 个不同类，而每一类在程度上没有任何差异，如本实验得到的 K 个类只能得知每个类中所有数据点欢迎程度相同，而无法确定这些数据点具体的欢迎程度，因此准确率的计算较为困难。

此时，需要通过排列组合的方式手动为各个类的欢迎程度进行赋值。本实验共有 3 种不同的欢迎程度，因此得到的 3 个聚类共有$A_3^3 = 6$种不同的赋值组合。依次按每个赋值组合对三个类中各数据点的预测欢迎程度赋值，再将该值与其真实欢迎程度比较，计算出聚类算法的准确率。由于有 6 种不同的赋值组合，因此总共可计算出 6 个不同的准确率，取其中最高的准确率作为聚类算法最终的准确率，这样准确率计算困难的问题便迎刃而解。

（2）运行效率

由于算法涉及到的对数据集中所有点的运算较多，如计算所有数据点到簇中心的欧氏距离、计算簇中所有数据点的均值等，如果采用一般的列表和 for 循环遍历方式对上述问题进行求解，则效率必然较慢，算法运行时间过长。

而采用 numpy 库的矩阵运算，通过对数据点整体构成的多维矩阵操作，可以方便快捷地完成上述计算，从而有效提高算法效率，大幅降低运行时间。同时，矩阵运算省去了复杂的多重嵌套循环，可减少代码冗余，使代码简洁美观、井然有序。

4.3.3 实验测试与结果分析

程序先采用随机抽样的方式初始化簇中心，算法执行的过程中在控制台会打印出相关提示信息。算法执行完毕后，程序对聚类模型的性能进行评估，评估结果会输出在控制台上，如图 4.3 所示：

```
C:\Users\LSY\AppData\Local\Programs\Python\Python39\python.exe
KMeans starts!
KMeans ends!
Accuracy: 84.36%, SSE: 24.91
Run time: 0.01s
```

图 4.3 控制台输出相关提示信息及模型评估结果

接下来采用尽可能相互远离的方式初始化簇中心，算法的执行结果如图 4.4 所示：

```
C:\Users\LSY\AppData\Local\Programs\Python\Python39\python.exe
KMeans starts!
KMeans ends!
Accuracy: 83.80%, SSE: 24.92
Run time: 0.02s
```

图 4.4 控制台输出相关提示信息及模型评估结果

从两者的算法执行结果对比可以看出，尽可能相互远离方式初始化比随机抽样方式初始化的算法执行时间略长，这是因为前者初始化时需要遍历所有数据点，找到与其他簇中心距离最远的点作为簇中心，从而耗时更长。

而尽可能相互远离方式初始化比随机抽样方式初始化的模型性能略优，二者的 SSE 相当，而前者的准确率略高于后者，说明尽可能相互远离方式初始化使算法构建出的聚类模型效果更好，原因是尽可能相互远离的点更有可能属于不同的聚类，因此最终构建好的聚类模型也与真实情况更加贴近。

从测试结果可以看到，尽管不同初始化算法间有细微差异，但二者的运行时间均较小，且模型性能均较为出色，准确率和 SSE 均在可接受范围内，说明 kmeans 算法整体的设计与实现正确，圆满完成了实验任务。

最后，对两者生成的二维平面点图，即聚类效果展示图进行查看，分别如图 4.5、图 4.6 所示。通过聚类效果展示图能够清晰直观地看到模型对各数据点的分类情况，以及聚类的整体分布情况等。从测试结果可以看出，两个模型得到的 3 个聚类基本准确无误，各数据点都被正确地划分在相应的聚类中，说明本实验聚类算法的性能较为出色。

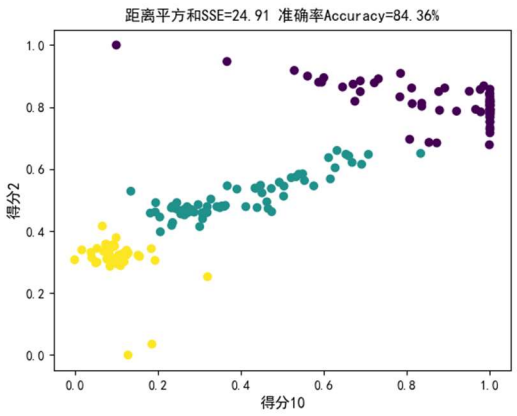

图 4.5 随机抽样初始化的聚类效果

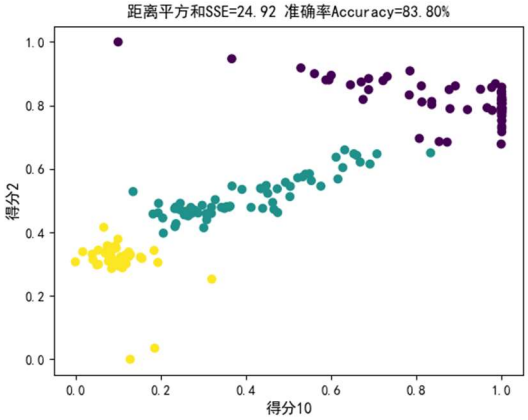

图 4.6 尽可能相互远离初始化的聚类效果

4.4 实验总结

本次实验通过对聚类算法之一 kmeans 算法的认识与实践，我对聚类算法有了更深刻的理解，进一步了解了聚类算法的实现流程。通过对 kmeans 算法的设计与编写，我从对原理的初步认识发展到对算法的灵活运用，并最终运用于实际问题的求解当中，做到学以致用。在聚类算法结束后，还通过准确率、SSE 等多维度对聚类模型进行了全方位地评估，并以二维平面点图的方式直观呈现，使我对聚类算法性能好坏的度量更加客观全面、清晰明了。

在完成本次实验之前，仅通过理论学习还无法真正体会到 kmeans 等聚类算法对于实现数据分类任务的强大；而当我完成本次实验，看到实验结果中极短的运行时间与较高的准确率等指标后，不由得感慨 kmeans 算法的便捷、高效，同时又兼具出色的性能，是解决有关实际问题的一大法宝。

kmeans 算法初始化的方式有多种，而聚类算法除 kmeans 外也有层次聚类、BFR、CURE 等等。通过实验，对 kmeans 算法进行横纵对比，可以分析总结得到每一种方法的特点、应用场景与孰优孰劣，从而有利于我们更好地对聚类算法形成全面而深刻的认识，与大数据分析的理论学习相得益彰。

5 推荐系统

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、对两个算法进行动漫预测评分对比
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 Anime 数据集，包含用户对动漫评分、动漫标签等文件，其中动漫评分文件分为训练集 train_set 和测试集 test_set 两部分

（1）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-动漫效用矩阵，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动漫进行评分预测，选取评分最高的 n 个动漫进行推荐。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 120 条用户-动漫评分记录，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-动漫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。

（2）基础版必做二：基于内容的推荐算法

将数据集 anime.csv 中的动漫类别作为特征值，计算这些特征值的 tf-idf 值，得到关于动漫与特征值的 n（动漫个数）*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，得到动漫之间的相似度矩阵。

对某个用户-动漫进行预测评分时，获取当前用户的已经完成的所有动漫的打分，通过动漫相似度矩阵获得已打分动漫与当前预测动漫的相似度，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，如果已打分动漫与当前预测用户-动漫相似度大于零，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 个动漫，score 为我们预测的计算结果，score’(i)为计算集合中第 i 个动漫的分数，sim(i)为第 i 个动漫与当前用户-动漫的相似度。如果 n 为零，则 score 为该用户所有已打分动漫的平均值。

要求能够对指定的 userID 用户进行动漫推荐，推荐动漫为预测评分排名前 k 的动漫。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-动漫进行预测评分，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。

（3）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。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

由于本次实验需要实现基于用户的协同过滤推荐算法和基于内容的推荐算法，并通过迷你哈希对两个算法的相似度计算进行降维，因此不同算法在设计与实现上有较大差异，但仍有部分功能为这些算法的公共部分，如数据集分析、构建效用矩阵等，因此先对公共部分的功能进行设计与实现。

（1）数据集

数据集包含动漫评分、动漫标签两部分，前者用于两个算法构建用户-动漫效用矩阵，后者用于基于内容的推荐算法构建 tf-idf 特征矩阵。

动漫评分包含训练集和测试集两部分，训练集中每条数据为一个用户对一条动漫的评分，两个算法将依据用户对动漫的评分构建用户-动漫效用矩阵；测试集中每条数据为一个用户与一条动漫，算法将对该用户为该动漫的评分进行预测。

动漫标签中每条数据为一条动漫的相关信息，包括动漫 id、动漫名称、动漫分类、动漫排名、动漫每种评分的具体数目等等，但本次实验只需关注动漫 id 及动漫分类即可，基于内容的推荐算法将依据动漫 id 和动漫分类构建 TF.IDF 特征矩阵。

（2）构建效用矩阵

效用矩阵反映了所有用户对动漫的评分，构建效用矩阵前，需要先读取动漫评分数据集中的训练集，获取用户数目与动漫数目。先初始化一个大小为最大动漫 id×最大用户 id 的矩阵，矩阵中每一项初始值为 NaN（Not a Number），表示非数值型数据，即矩阵上该点对应的用户为动漫的评分不存在。采用 NaN 而不用 0 对效用矩阵初始化的原因是，0 在程度上有着数值含义，可视作矩阵上该点对应用户为动漫的评分为 0，但实际上用户为动漫的评分由于不存在而无法确定。而在后续矩阵运算时值为 NaN 的点将不参与运算，因此 NaN 相比 0 更适合用于效用矩阵的初始化。

初始化效用矩阵完成后，遍历训练集中的每条数据，将<用户 id, 动漫 id, 评分>的三元组解包，并以用户 id 为列号，动漫 id 为行号，将矩阵中对应点的值置为该条评分。当训练集中的所有数据遍历完毕后，效用矩阵便构建完成。

5.3.1.1 基于用户的协同过滤推荐算法

在完成效用矩阵的构建后，执行基于用户的协同过滤推荐算法。其主要步骤是基于效用矩阵，使用 pearson 相似度计算方法计算用户之间的相似度，构建相似度矩阵；基于构建好的相似度矩阵，对指定用户为指定动漫的评分进行预测，或对指定用户进行动漫推荐等。

基于用户的协同过滤推荐算法的原理图如图 5.1 所示：

（1）构建相似度矩阵

基于用户的协同过滤算法的关键是得到与各个用户最相似的 k 个用户，因此需要构建相似度矩阵来表示每个用户到其他各用户的相似度，并基于相似度矩阵完成评分预测及动漫推荐等任务。

相似度矩阵由效用矩阵构建而来，相似度矩阵中两个用户之间的相似度取决于效用矩阵中这两个用户的动漫评分向量的相似度，通过 pearson 相似度计算方法计算得到。因此，在构建相似度矩阵前，需要先对效用矩阵进行修正，具体方法是将效用矩阵中每个用户的动漫评分减去该用户所有动漫评分的均值，并注意值为 NaN 的点不参与均值的计算。

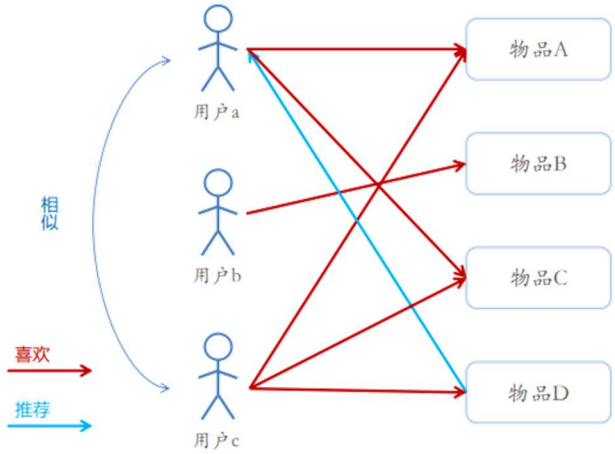

图 5.1 基于用户的协同过滤推荐算法原理图

先初始化一个行数和列数均为用户数的零矩阵作为相似度矩阵。得到修正后的效用矩阵后，依次遍历矩阵的每一列，得到各用户的动漫评分向量后，再次遍历矩阵的每一列，得到其他用户的动漫评分向量，即进行一次嵌套循环。由于相似度矩阵是对称矩阵，因此对得到的两个用户之间的相似度进行计算前，先通过相似度矩阵判断这两个用户之间的相似度是否已经计算，如果是则跳过本次计算，否则通过 pearson 相似度计算方法进行计算。pearson 相似度方法的具体实现为对两个用户评分向量的交集（即在两个向量中同时有评分的位置各自组成的向量）计算余弦相似度，作为这两个用户之间的相似度。嵌套循环结束后，得到的相似度矩阵即为最终的相似度矩阵。将该相似度矩阵保存在内存中，便于后续任务使用。

相似度矩阵除通过嵌套循环手动运算得到，还可以调用 pandas 库中 DataFrame 对象的 curr 方法来计算。本次实验采用前者进行计算。

（2）预测用户对动漫的评分

预测指定用户为指定动漫的评分时，先基于构建好的相似度矩阵，找到与该用户最相似的 k 个用户，然后使用这 k 个用户对该动漫的平均评分进行预测。

具体做法是先从相似度矩阵中得到该用户所在行，表示该用户与其他各用户的相似度，接着找到该向量中值最大的 k 个位置对应的索引，表示与该用户最相似的 k 个用户。然后在原始效用矩阵中依次遍历这 k 个用户对该动漫的评分，同时维护总评分和评分人数两个变量。如果某个用户对该动漫没有评分，即评分为 NaN，则跳过该用户；否则将评分加入到总评分中，并将评分人数加一。

遍历结束后，如果评分人数不为 0，将总评分与评分人数的比值作为平均评分，也就是对指定用户为指定动漫的评分的预测值；否则将 5 作为评分的预测值，即评分的中位数，因为此时在最相似的 k 个用户中没有任何用户对该动漫进行评分，无法获得参考数据，所以只能用评分的中位数作为评分的预测值，具有猜测含义，因此此时的预测值可能与真实值偏差较大。

（3）对用户进行动漫推荐

对指定用户进行动漫推荐是在预测用户对动漫评分的基础上进行的，总体流程为遍历得到用户没有评分的动漫，一一对其进行预测，在所有没有评分的动漫中取评分预测值排在前 N 的动漫推荐给用户。

具体做法是先初始化一个记录动漫到其评分预测值的字典。在效用矩阵中遍历该用户所在列，即该用户对动漫的评分向量，当遍历到的点的值为 NaN 时，表示该用户对该动漫没有评分，因此该动漫是潜在的推荐对象之一，此时调用（2）中的预测函数对该用户为该动漫的评分进行预测，并将该动漫和评分预测值保存在字典中。需要注意的是，由于预测动漫

时，选取的最相似的 k 个用户中对该动漫评分过的用户人数可能较少，此时将这些用户对该动漫评分的均值作为预测值具有较大的偶然性，预测评分不完全合理，因此对于这些动漫应考虑舍弃，而不加入到字典中。在对潜在的推荐对象进行评分预测时，先最相似的 k 个用户重对该动漫评分过的用户人数，该人数可以通过（2）中的预测函数与预测值一并返回，如果小于某个设定好的阈值则考虑舍弃该动漫。

遍历结束后，按评分预测值从大到小对记录有动漫到其评分预测值的字典进行排序，得到评分预测值最大的前 N 个动漫，并将这些动漫推荐给用户。最终，在控制台将这些动漫的编号及其评分预测值格式化打印出来，便于查看和比对实验结果。至此，基于用户的协同过滤推荐算法中预测和推荐两大功能就已完成。

5.3.1.2 基于用户的协同过滤推荐算法（迷你哈希降维）

迷你哈希算法可以对推荐算法的计算过程进行降维，从而减少程序的整体运行时间，但维度减少带来的是准确度的少许损失，然而这些损失也在我们可接受的范围之内。因此，迷你哈希算法可以在对结果准确性影响较小的情况下，有效减少程序运行时间，提高运行效率。

迷你哈希算法在推荐算法中体现在相似度矩阵的构建上。对于基于用户的协同过滤算法，迷你哈希算法在构建相似度矩阵时共经历了构建特征矩阵、生成签名矩阵、计算相似度矩阵三个步骤，其中构建签名矩阵一步完成了对原效用矩阵的降维。采用迷你哈希算法的推荐算法在除构建相似度矩阵的其他步骤上都与基础版相同。

（1）构建特征矩阵

迷你哈希算法中，特征矩阵是由效用矩阵舍入得到的。对于基于用户的协同过滤算法，本次实验要求将效用矩阵中评分大于 5 的点设为 1，其余点设为 0，完成数据的舍入操作，从而构建得到迷你哈希算法生成签名矩阵时所需的特征矩阵。

（2）生成签名矩阵

签名矩阵由构建好的特征矩阵计算而来，签名矩阵的生成也是迷你哈希算法的核心步骤，完成了对原效用矩阵的降维。首先给定哈希函数的数目，当哈希函数数目较多时，为了避免设计大量的哈希函数，此时采用对特征矩阵行随机排列的办法来模拟不同哈希函数，可以不用完成各哈希函数的具体实现即可对签名矩阵进行计算。该办法中一次随机排列的作用与一个哈希函数相同，对随机排列后得到的特征矩阵的列扫描，在向量中分别记录每列中第一个出现 1 的位置，从而计算得到签名矩阵中的一行。

当给定 N 个哈希函数时，总共需要对特征矩阵的所有行随机排列 N 次。为确保每次随机排列的随机性，先生成 N 个随机数作为后续 N 次随机排列的种子。每次随机排列时，从种子列表中取相应位置上的数，并将其设置为随机数生成器的种子。先初始化一个行数为 N、列数为用户数的零矩阵，作为空签名矩阵。接下来通过 numpy 库中 ndarray 对象的 shuffle 方法来对特征矩阵进行随机排列，然后遍历随机排列后矩阵的每一列，从上到下找到该列中第一个值为 1 的位置，并将该位置对应行的索引加入到空列表中。遍历结束后，得到的列表即为一行签名后的向量，按顺序将其赋给签名矩阵的相应行。当执行完全部 N 个随机排列后，得到的签名矩阵即为最终迷你哈希算法的签名矩阵。

（3）计算相似度矩阵

迷你哈希算法中的相似度矩阵由生成好的签名矩阵计算而来，由于签名矩阵中每一列向量对应一个用户处理后的动漫评分，因此计算签名矩阵中列与列之间的相似度即可得到用户之间的相似度矩阵，并且该相似度矩阵和不采用迷你哈希所计算得到的相似度矩阵相比，除各用户之间的相似度在两个矩阵中略有差异外，两个矩阵的大小、形状完全一致，因此迷你哈希算法得到的相似度矩阵可直接用于后续推荐算法的功能实现。

计算相似度矩阵时，可对签名矩阵整体进行矩阵运算，从而加快算法的执行效率。具体做法是先对签名矩阵进行归一化处理，然后将处理后的矩阵与其倒置矩阵点乘，得到的矩阵

即为算法最终计算得到的相似度矩阵。将该相似度矩阵保存在内存中，便于后续任务使用。至此，迷你哈希算法在构建相似度矩阵中的作用便体现完毕，接下来只需基于此相似度矩阵，按照基础版的基于用户的协同过滤推荐算法中的预测和推荐功能进行实现即可。

5.3.1.3 基于内容的推荐算法

基于内容的推荐算法与基于用户的协同过滤推荐算法在相似度矩阵的构建上有所差异，由于该推荐算法是基于动漫内容的，因此构建相似度矩阵时借助的不是用户-动漫效用矩阵，而是 TF.IDF 特征矩阵，该矩阵反映的是各派别在每个动漫中的重要程度。通过计算得到的 TF.IDF 矩阵，构建动漫与动漫之间的相似度矩阵，再基于构建好的相似度矩阵，对指定用户为指定动漫的评分进行预测，或对指定用户进行动漫推荐等。

基于内容的推荐算法的原理图如图 5.2 所示：

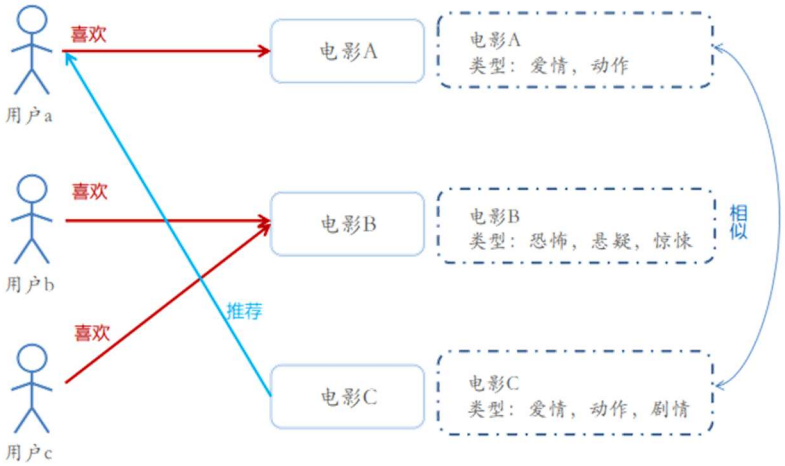

图 5.2 基于内容的推荐算法原理图

（1）构建 TF.IDF 矩阵

TF.IDF 指词项频率（TF）与逆文档频率（IDF）的乘积。其中词项频率表示词项 i 在文档 j 中出现频率的归一化结果，即词项 i 在文档 j 中出现的频率f_{ij}与同一文档中出现最多的词项 k 的频率f_{kj}的比值，计算公式如下：

$$TF_{ij} = \frac{f_{ij}}{max_k f_{kj}}$$

逆文档频率表示总文档数目与出现词项 i 的文档的数目的比值，并对比值取对数后的结果，计算公式如下：

$$IDF_i = \log_2 \frac{N}{n_i}$$

在本次实验中，词项表示动漫的派别，文档表示动漫，根据上述公式对动漫标签数据集中的信息进行提取和计算，可以构建得到最终的 TF.IDF 矩阵。矩阵中每条动漫对应的向量为不同派别与该动漫之间的 TF.IDF 得分，表示派别在动漫中的重要程度，也是后续计算两个动漫之间的相似度，构建相似度矩阵的重要依据。

（2）构建相似度矩阵

基于内容的推荐算法构建根据（1）中构建好的 TF.IDF 矩阵构建动漫-动漫的相似度矩阵，相似度矩阵中两个动漫之间的相似度为对应的两个向量的余弦相似度。相似度矩阵可以通过嵌套循环的方式得到，如 5.3.1.1 中（1）的方法；也可以直接对 TF.IDF 矩阵整体进行矩阵运算得到，如 5.3.1.2 中（3）的方法，故此处不再赘述。

（3）预测用户对动漫的评分

基于内容的推荐算法与先前基于用户的协同过滤算法在预测用户对动漫的评分的实现

上略有不同，主要体现在本算法是基于相似的动漫，而先前算法是基于相似的用户。

预测用户对动漫的评分时，先根据公共部分初始化得到的效用矩阵，取出该用户对所有动漫评分的向量，同时初始化总评分及总相似度两个变量用于对相似动漫进行记录。然后依次遍历该评分向量，如果值为 NaN，表示该用户对遍历到的动漫没有评分，则跳过该动漫；否则通过相似度矩阵得到向量该位置对应动漫与要预测动漫之间的相似度，如果相似度大于 0，表明两个动漫之间具有一定的正相关性，则将该用户对该动漫的评分与两个动漫之间的相似度的乘积，即根据相似度加权后的评分加入到总评分中，将两个动漫之间的相似度加入到总相似度中。

当用户对所有动漫评分的向量遍历结束后，得到的总评分与总相似度的比值即为最终对要预测动漫的加权平均评分，并将该评分作为预测值返回。

（4）对用户进行动漫推荐

与基于用户的协同过滤算法相同，基于内容的推荐算法在对用户进行动漫推荐时同样是在预测用户对动漫评分的基础上进行的，且实现基本上完全一致。

具体做法仍是先初始化一个记录动漫到其评分预测值的字典。在效用矩阵中遍历该用户所在列，即该用户对动漫的评分向量，当遍历到的点的值为 NaN 时，表示该用户对该动漫没有评分，因此该动漫是潜在的推荐对象之一，此时调用（3）中的预测函数对该用户为该动漫的评分进行预测，并将该动漫和评分预测值保存在字典中。

遍历结束后，按评分预测值从大到小对记录有动漫到其评分预测值的字典进行排序，得到评分预测值最大的前 N 个动漫，并将这些动漫推荐给用户。最终，在控制台将这些动漫的编号及其评分预测值格式化打印出来，便于查看和比对实验结果。至此，基于内容的协同推荐算法中预测和推荐两大功能就已完成。

5.3.1.4 基于内容的推荐算法（迷你哈希降维）

使用迷你哈希算法对基于内容的推荐算法进行降维时，主要流程与先前使用迷你哈希算法对基于用户的协同过滤推荐算法进行降维一致，即构建特征矩阵、生成签名矩阵和计算相似度矩阵，且具体实现也基本相同，唯一差别是本算法在构建特征矩阵时基于的是 TF.IDF 矩阵，而不是先前的效用矩阵。

构建特征矩阵时，将 TF.IDF 矩阵中值不为 0 的点的值设为 1，其他所有点的值设为 0，完成数据的舍入操作，从而构建得到迷你哈希算法生成签名矩阵时所需的特征矩阵。舍入的目的是只关注动漫中涉及到的派别，而忽略派别在动漫中的重要程度，因此会造成少许准确度上的牺牲，但是能有效减少算法运行时间，加快运行效率。

在构建好特征矩阵后，由于迷你哈希算法对基于内容的推荐算法的后续操作，即生成签名矩阵、计算相似度矩阵等，与 5.3.1.2 中的实现完全相同，故此处不再赘述。

5.3.2 遇到的问题及解决方式

（1）空值处理

在两个推荐算法构建效用矩阵的过程中，如果一个用户对一个动漫没有评分，即该评分不存在，那么矩阵中对应点的值应该为 NaN 而不是 0。这是因为 0 在程度上有着数值含义，可视作矩阵上该点对应用户为动漫的评分为 0，但实际上用户为动漫的评分由于不存在而无法确定。

NaN 值的存在使得矩阵进行相关运算（如点乘、求平均值等）时，对值为 NaN 的点要做特殊处理，否则将无法得到正确的计算结果，如 numpy 库中对于包含值为 NaN 的点的矩阵的一般运算得到的运算结果全为 NaN。因此，需要通过对值为 NaN 的点进行特殊处理来避免错误情况的产生，包括手动处理和调用 numpy 库中 NaN 相关的运算函数等。

其中，手动处理指在需要对矩阵进行运算（如计算相似度矩阵）时，对矩阵进行遍历，

判断其中包含值为 NaN 的点的行或列，对这些行或列作特殊处理，如舍弃该行或该列、舍弃该值为 NaN 的点等，然后再进行相应的计算。这样的手动处理可有效避免 NaN 对计算带来的不方便影响，但整体上效率较慢。

调用 numpy 库中 NaN 相关的运算函数（如 nanmean、nansum 等）可以高效地化解 NaN 带来的不便之处，其原理为将矩阵中值为 NaN 的点排除在外后对矩阵进行相应运算。如 nanmean 函数可以计算向量中除值为 NaN 的点以外的所有点的平均值。这种方法既能有效解决 NaN 带来的运算问题，又能确保程序运行的简洁性与高效性，将重点聚焦于推荐算法实现本身，因此本次实验选用该方法在计算时对 NaN 值进行处理。

（2）内存有限

由于动漫的总数很大，且动漫编号的最大值超过动漫总数两倍（由于动漫编号是非连续的），因此直接用动漫编号作为效用矩阵或相似度矩阵的索引会导致内存无法容纳如此庞大的数据（约需 17GB 空间），且这样的矩阵是极其稀疏的。

因此，在遍历数据集得到用户动漫评分的同时，维护一个动漫编号到矩阵索引的字典，这样可以对矩阵大小进行大幅压缩，使得矩阵的行数与动漫总数完全相同，从而使矩阵在内存中占用的空间减小了约四分之三。尽管对动漫编号到矩阵索引进行映射后，能将矩阵容纳在内存中，但此时矩阵占用的空间仍是一个不小的数字（约需 4GB 空间），因此考虑对数据的存储方法进行进一步优化。

由于效用矩阵是一个稀疏矩阵，其中绝大部分位置上的值都为空，因此将带来不小的资源浪费。可以考虑用三元组的方式替代矩阵对每条用户为动漫的评分进行存储，这样虽在数据查询上会增加一定的开销与时间复杂度，但能大幅减少数据在内存中的占用，带来数量级上的优化。此外，对于海量数据，还可以考虑运用分布式的方式对数据进行处理，如实验 1 中的 Map-Reduce 算法等，由于本实验篇幅有限，此处不作讨论。

5.3.3 实验测试与结果分析

为方便实验测试和用户使用，程序编写了用户友好的选择菜单界面（如图 5.3 所示），程序开始后，可以选择要选用的算法（基于用户的协同过滤推荐算法、基于内容的推荐算法），并选择是否使用迷你哈希对推荐算法计算过程进行降维（普通版、迷你哈希版）。

```
C:\Users\LSY\AppData\Local\Programs\Python\Python39\python.exe
请选择推荐系统模型：
1. 基于用户-用户协同过滤的推荐
2. 基于内容的推荐
1
请选择模型算法：
1. 普通版
2. MinHash版
1
```

图 5.3 程序选择菜单界面

接下来分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，及这两个算法对应的迷你哈希版本，总共四个不同版本的推荐系统模型的性能进行测试，测试指标包括算法总运行时间、算法在测试集上的 SSE（误差平方和）、算法输出的推荐结果等。在得到所有模型的测试结果后，对四个模型进行全方面的比较，分析得出各模型的特点及优缺点，并将结果进行汇总。

运行各个版本的推荐系统模型后，该模型在各项指标上的性能会输出在控制台上。先对基于用户的协同过滤推荐算法普通版和迷你哈希版进行测试和比较，测试结果分别如图 5.4、图 5.5 所示：

CollaborativeFiltering 基础版对用户629推荐如下动漫：

Anime	Score
4181	9.333
11061	9.286
7311	8.967
12365	8.964
11741	8.917
23273	8.898
17074	8.867
5081	8.850
10030	8.800
245	8.750
457	8.739
1698	8.720
1	8.692
28851	8.674
2251	8.636
28025	8.593
7674	8.571
9756	8.545
9617	8.542
36296	8.533

CF 基础版SSE: 251.20397262060328
总时间：4.83s

图 5.4 基于用户的协同过滤普通版

CollaborativeFiltering MinHash版对用户629推荐如下动漫：

Anime	Score
918	9.450
11061	9.320
245	9.125
4181	9.000
7311	8.931
15417	8.905
28851	8.881
10030	8.880
457	8.818
12355	8.767
17074	8.758
1	8.742
11741	8.729
38329	8.667
16894	8.641
31181	8.640
23273	8.636
25835	8.625
3784	8.609
30654	8.575

CF MinHash版SSE: 270.3618539448793
总时间：2.92s

图 5.5 基于用户的协同过滤迷你哈希版

从测试结果可以看出，基于用户的协同过滤推荐算法普通版相比迷你哈希版具有更小的SSE，但是算法总运行时间更长。具有更小的 SSE 表明普通版的模型准确率更高，而更长的运行时间表明模型的构建和预测过程耗时较长。

两个模型测试结果的差异也正反映出迷你哈希算法通过降维在相似度矩阵的计算上进行的优化，即在牺牲少许准确度的同时有效地提高算法效率，减少程序整体运行时间。由测试结果看到，迷你哈希版的 SSE 相比普通版仅增加了不到 20，比值约 7%，而运行时间却缩短了近 40%，说明迷你哈希算法对算法效率的提高作用是相当大的，且准确度依旧能达到较高的水准，在可接受范围内，表明了算法的稳定性俱佳。

总的来说，迷你哈希算法的设计和编写是相当成功的。而比较两个版本的推荐结果，发现其中有大多动漫的排序发生了变化，但整体上在普通版排名靠前的动漫在迷你哈希版也排名靠前，表明两个算法最后的结果是相近且基本正确的。

接下来对基于内容的推荐算法普通版和迷你哈希版进行测试和比较，测试结果分别如图5.6、图 5.7 所示。

从测试结果可以看出，基于内容的推荐算法普通版相比迷你哈希版具有更大的 SSE，且算法总运行时间更长。迷你哈希算法在算法准确度方面略有提升，这可能是因为基于内容推荐时，普通版中动漫内容维度过多导致分类困难、过拟合等不利现象产生，而迷你哈希算法对模型构建的过程进行了降维，有效避免了上述情况的发生，所以准确度能略胜一筹。除此之外，迷你哈希版的运行时间缩短了高达 74%，可见迷你哈希算法在基于内容的推荐算法中的设计与实现是极其成功的，对程序整体运行效率的优化效果十分明显。

比较两个版本的推荐结果，发现其中两个算法推荐大多数动漫并不相同，这可能是因为迷你哈希算法进行降维后对基于内容的推荐影响更大，导致最后推荐结果差异较大，但两个算法在测试集上的 SSE 均较小，表明算法的准确度仍然是较高的，说明两个算法的设计与编写是基本正确的。

对两个算法的普通版和迷你哈希版进行纵向对比后，再对两个不同的推荐算法本身进行横向对比。这里均以两个推荐算法的普通版为例进行比较，分析两个模型在不同指标上的性能差异及其原因。

```
Content-Based 基础版对用户629推荐如下动漫：
Anime       Score
--------------------------
 5140       8.111
 7296       8.111
 1991       8.071
 2349       8.057
 3411       8.057
 1299       8.036
 3800       7.980
  782       7.962
  989       7.952
 3326       7.951
  437       7.949
  934       7.943
  658       7.942
 3002       7.942
 1561       7.933
 3685       7.933
  376       7.933
  323       7.920
 3465       7.920
 1983       7.919
Content-Based  基础版SSE: 175.7253596826293
总时间：40.25s
```

图 5.6 基于内容的推荐算法普通版

```
Content-Based MinHash版对用户629推荐如下动漫：
Anime       Score
--------------------------
 4103       10.000
30766       10.000
37695       8.360
37530       8.105
 2349       8.090
 5273       8.060
 5826       8.060
22111       8.060
24237       8.060
31177       8.060
37468       8.054
 5622       8.041
37775       8.034
22197       8.026
37549       7.977
37528       7.950
  989       7.934
 1561       7.924
 2728       7.924
22059       7.924
Content-Based  MinHash版SSE: 164.3462561692136
总时间：10.56s
```

图 5.7 基于内容的推荐算法迷你哈希版

经测试结果对比可以得到，基于用户的协同过滤推荐算法相比基于内容的推荐算法具有更大的 SSE，但是算法总运行时间更短。这表明基于用户的推荐算法的运行效率更高，而基于内容的推荐算法的准确度更高。产生这种差异的可能原因是相比庞大的动漫数据集，用户数据集中供模型训练的用户数据较少，导致基于用户时构建的模型性能略差，而基于内容时动漫数目、动漫类别等数据较为丰富，因此构建出来的模型性能略优。

而也正是由于庞大的动漫数据集和丰富的动漫类别等数据，导致基于内容时模型构建的时间比基于用户长。且基于内容时算法需要额外构建 TF.IDF 矩阵，且基于内容时的相似度矩阵的大小约为 20000×20000，几乎是基于用户时的 600 倍（800×800），庞大的计算量和更为复杂的构建过程也是使得基于内容的推荐算法总运行时间要多于基于用户的推荐算法的主要因素之一。

实验测试部分的最后，将四个不同版本的推荐系统模型在各指标上的性能以折线图的形式呈现，从而能清晰直观地对各模型的差异进行对比总结，如图 5.8 所示：

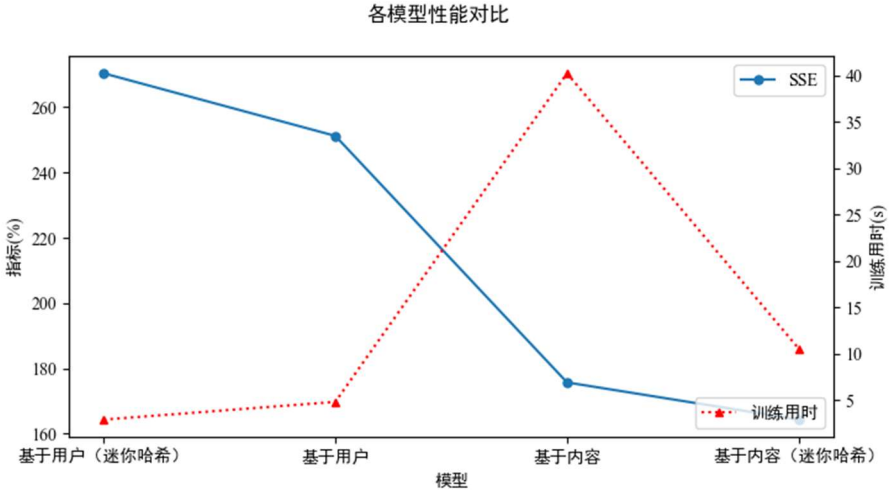

图 5.8 各推荐算法模型性能对比

5.4 实验总结

本次实验是《大数据分析实验》系列实验中的最后一个实验，也是难度最高的一个实验，涉及到两套采取不同策略的推荐算法的设计与实现，并使用了迷你哈希算法分别对其进行降维。尽管本次实验是五个实验中最为复杂、耗时最久的一个实验，但在对这些推荐算法设计和实现的过程中，我也将理论课的知识与实践相结合，在实践中对推荐系统的流程与实现有了更为深刻的理解和掌握，同时还乐在其中。

根据理论课上所学的知识，结合实验指导书中介绍推荐算法的流程图和助教老师实验课上的讲解，我对基于用户的协同过滤推荐算法和基于内容的推荐算法有了更为清晰的认识，并能独自完成两个算法全流程的编写，并在测试部分证明了算法的正确性与高效性，令我感到十分欣慰与满足。在完成对普通版的实现后，我还尝试了迷你哈希算法的编写，尽管起初对迷你哈希算法的认知不够深入，以至于几乎无从下手，但通过上网查阅有关资料，并与老师同学讨论学习后，我对该算法有了全新的认识，也顺利在实验中完成了迷你哈希算法的编写，并与先前普通版的推荐算法进行了横纵对比，分析和总结了各模型之间的特点和差异，并以图表的形式呈现，回看本次实验一路走来的成果，可谓收获颇丰。

《大数据分析实验》是一门趣味与挑战并存的实验课，涵盖范围之广，将理论课上 Map-Reduce 算法、PageRank 算法、关联规则挖掘、KMeans 算法和推荐系统等知识点应用到实际问题的处理和解决中，不仅让我们通过实践对理论课本中枯燥乏味、晦涩难懂的知识有了更为深刻、更为新颖、更为全面的体会，又很好地培养了我们的程序设计能力、系统架构能力、独立思考能力和问题解决能力，在我看来本次实验对个人的帮助不仅体现在《大数据分析》一门课本身，更体现在个人能力塑造和提高的方方面面，是多样的、价值极高的。

《大数据分析实验》虽然课时较少，但是任务量繁重，富有较大的挑战性，这使我不得不利用一些课余时间以争取在规定时间内圆满完成各项实验任务。尽管如此，我认为这数周以来为本次实验付出的心血都是值得且帮助极大的，一路上也认识了许多优秀的助教学长，在与他们验收交流的过程中思维能力得到了很强的锻炼；也遇到了循循善诱、耐心负责的王蔚老师，在她的指引下我对大数据分析与数据挖掘领域有了初步且全面的了解。

尽管大数据分析学习的课程任务已经收官，但大数据学习和计算机学习的道路仍在不断延伸，一路上还会遇到新的坎坷，也能收获惊喜与成长。希望在今后的学习生活中永葆持之以恒的探索精神，遇到陌生的问题与挑战时能临危不乱，耐心且细心地学习新的知识、新的方法，积少成多，久久为功。