目录

实验一 wordCount 算法及其实现

1.1 实验目的

1、理解 Map-Reduce 算法的基本思想和流程，掌握其解决大规模数据处理问题的能力；

2、掌握 Map-Reduce 思想解决 wordCount 问题的方法，理解 Map-Reduce 中的 Map 和 Reduce 操作以及它们的实现；

3、了解并应用 Combine 和 Shuffle 过程，提高 Map-Reduce 算法的性能。

Map-Reduce 算法是一种分布式计算范式，由 Google 在 2004 年提出，用于解决大规模数据处理问题。Map-Reduce 算法将大规模数据分割成若干个小数据块，并将这些小数据块分配给多个计算节点（即 Map 任务），计算节点将小数据块映射为（key，value）对，然后再将相同 key 的 value 聚合起来形成新的（key，value）对（即 Reduce 任务）。Map-Reduce 算法的优点在于其可扩展性和容错性，可以高效地处理大规模数据，并且具有自动容错机制。

在本实验中，我将应用 Map-Reduce 思想解决 wordCount 问题，即对给定的文本文件进行单词计数。在 Map 过程中，我们将每个单词映射为（单词，1）的键值对，然后在 Reduce 过程中将相同的单词键值对聚合起来，并统计其出现的次数。同时，我还将了解和应用 Combine 和 Shuffle 过程，前者可以在 Map 任务中进行一定程度的合并，减少数据传输量，后者则用于将不同 Map 任务产生的键值对进行合并，传递给 Reduce 任务。

通过本实验，我将深入理解 Map-Reduce 算法的基本思想和流程，并掌握其在实际应用中的方法和技巧，从而提高处理大规模数据的能力和效率。

1.2 实验内容

本实验要求模拟 9 个分布式节点，对 9 个预处理过的源文件进行处理，实现 WordCount 功能，并输出对应的 Map 文件和最终 Reduce 结果文件。为了模拟分布式环境，本实验要求使用多线程进行处理。

具体步骤如下：

1. 首先对 9 个预处理过的源文件进行分割，每个文件分为三个部分。每个部分对应一个 Map 节点进行处理。将处理后的结果写入对应的 Map 文件

中。

2. 对每个 Map 文件进行合并，并根据 Key 进行排序，以便后续的 Reduce 操作。合并后的文件交由对应的 Reduce 节点进行处理。为了实现 Shuffle 过程，应按照 Key 的 Hash 值将结果分配到不同的 Reduce 节点。每个 Reduce 节点应处理平均的工作量，以避免某个 Reduce 节点成为瓶颈。
3. Reduce 节点对各自分配到的数据进行统计，得到最终的 WordCount 结果，并将结果写入对应的 Reduce 文件中。

进阶内容，在 MapReduce 的基础上添加 Combine 和 Shuffle 过程。Combine 过程是在 Map 阶段进行的一种局部聚合操作，可以减少数据传输的量。Shuffle 过程是在 Map 与 Reduce 之间的一种数据分配操作，它将 Map 产生的数据按照 Key 的 Hash 值分配到不同的 Reduce 节点，以实现并行处理。

本实验还可以对线程运行时间进行计算，以考察这些过程对算法整体的影响。为了实现并行处理，采用多线程来处理每个阶段的任务，例如对每个文件分段进行处理、对 Map 结果进行合并、对 Reduce 结果进行合并等。

1.3 实验过程

1.3.1 编程思路

在实现 WordCount 的 MapReduce 算法时，需要将大规模的数据分解成若干个小规模的数据块，每个小规模的数据块都由一个独立的 Map 任务进行处理，并将结果输出给 Reducer。Reducer 任务将所有 Map 任务的结果组合在一起，进行进一步的汇总和处理，得到最终的结果。

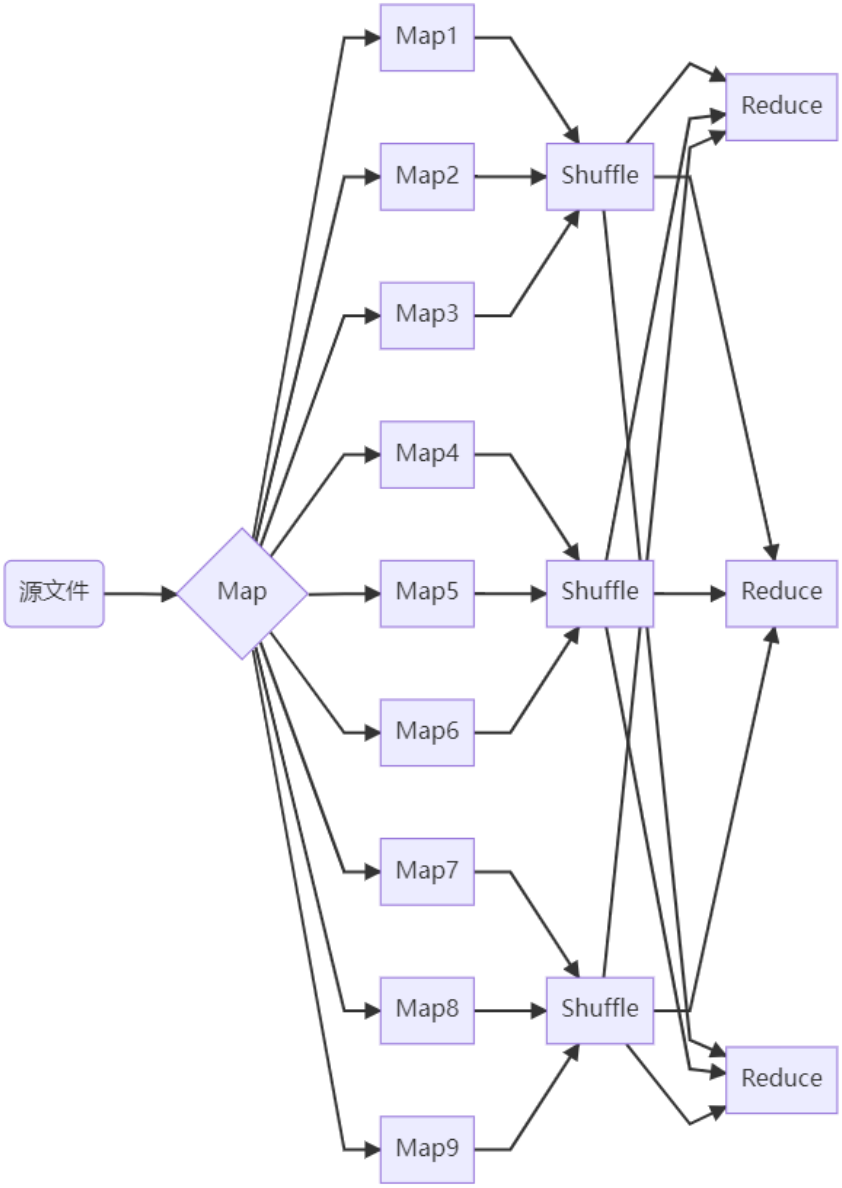

图 1.1 wordcount 示意图

具体实现的流程如下：

1. 读取源文件

首先需要从源文件中读取数据。本实验中，提供了 9 个源文件，每个文件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。因此，需要将每个源文件读取进来，并将其分解成若干个小规模的数据块。在读取数据的过程中，可以采用多线程来模拟分布式节点。

2. Map 阶段

对于每个小规模的数据块，需要进行 Map 操作，以统计单词出现的次数。在 Map 阶段，可以将数据分割成若干个键值对，其中键是单词，值是单词出现的次数。每个 Map 任务将统计每个键（单词）出现的次数，并将结果输出给 Reducer。

在输出时，可以采用哈希函数，将相同的键值对输出到同一个文件中，以便于Reduce 任务的处理。

3. Shuffle 阶段

在 Map 任务结束后，需要对所有的 Map 任务的输出结果进行整合，以便于Reducer 任务的处理。在 Shuffle 阶段，需要对 Map 任务的输出结果进行排序，并将相同键值的数据归并到同一个分组中。Shuffle 阶段的实现可以采用多线程来加快处理速度，并保证各个 Reducer 节点的工作量相当。

4. Reduce 阶段

在 Shuffle 阶段结束后，Reducer 任务将收到所有 Map 任务的输出结果。Reducer任务将对所有的键值对进行进一步的处理，以得到最终的结果。在Reduce阶段，可以将相同的键（单词）的值相加，以得到该单词在整个文本中出现的总次数。在 Reduce 任务的处理过程中，也可以采用多线程来提高处理速度。

5. 输出结果

在 Reduce 阶段结束后，需要将最终的结果输出到文件中。输出的格式可以采用键值对的形式，其中键是单词，值是该单词在整个文本中出现的总次数。输出结果可以通过文件系统或者网络传输到其他应用程序中进行进一步的处理。

总体上，实现 WordCount 的 MapReduce 算法需要采用多线程来模拟分布式节点，实现数据的读取、Map、Shuffle、Reduce 和结果输出等操作。

1.3.2 遇到的问题及解决方式

在实现该实验的过程中，我遇到了以下问题：

1. 如何在多线程下实现对多个源文件的并行处理？

解决方式：可以使用 Python 的 threading 库来创建多个线程，每个线程处理一个源文件。在创建线程时，可以使用 target 参数指定线程要运行的函数，并将需要传递的参数通过 args 参数进行传递。在处理完所有线程后，通过 join 方法等待所有线程的结束。

2. 如何实现 shuffle 过程？

解决方式：在 map 阶段输出的中间结果需要进行 shuffle 操作，即将相同单词的中间结果分配到同一个 reduce 节点进行处理。为了保证 reduce 节点的负载尽量均衡，可以将中间结果根据单词的首字母进行分组，并将同一组中的所有结果分配到同一个 reduce 节点进行处理。

3. 如何处理文件中的非法字符？

解决方式：在读取文件时，需要处理文件中可能出现的非法字符，例如换行符、制表符、空格等。可以使用 Python 的 strip 方法去除字符串中的空白字符，使用 split 方法根据逗号进行分割。

4. 如何进行多线程的时间测量？

解决方式：可以使用 Python 的 time 库中的 perf_counter 方法，该方法返回当前时间的计数值。在启动线程前，记录下当前时间；在每个线程结束后，计算线程运行的时间差，即为该线程的运行时间。

5. 如何保证多个 reduce 节点处理的数据不重复？

解决方式：可以在输出文件名中添加节点标识符，例如 reduce1、reduce2、reduce3，保证每个 reduce 节点输出的结果不会相互覆盖。

6. 如何在 reduce 阶段将输出结果按照单词的字典序排序？

解决方式：可以使用 Python 的 sorted 函数，对每个单词的计数结果进行排序。在 sorted 函数中，可以通过 key 参数指定排序方式，例如可以按照单词的字典序进行排序。

1.3.3 实验测试与结果分析

本次实验中，我采用了 map-reduce 算法解决 wordCount 问题，并在此基础上添加了 combine 与 shuffle 过程。由于源文件较大，使用了多线程来模拟分布式节点。

在实验过程中，首先运行 map.py 程序，使用了 9 个线程，每个线程对应一个源文件，每个源文件包含 100 万个单词，该程序用时约 10 秒。

```
t1: 10.7574154 s
t2: 10.7574448 s
t3: 10.7574669 s
t4: 10.7574744 s
t5: 10.7574814 s
t6: 10.757488500000001 s
t7: 10.7574957 s
t8: 10.7575027 s
t9: 10.7575097 s
total time: 10.757514 s
Estimated total time without map-reduce: 96.817624 s
```

图 1.2 Map.py 运行输出

接着，运行了 combine.py 程序，将之前 map 产生的文件进行 combine 操作，以减少后续 shuffle 操作的数据量。该程序用时约 17 秒。

```
t1: 17.0932565 s
t2: 17.0933057 s
t3: 17.093338 s
t4: 17.093368599999998 s
t5: 17.0933922 s
t6: 17.0934001 s
t7: 17.0934076 s
t8: 17.093415699999998 s
t9: 17.0934244 s
total time: 17.093429 s
Estimated total time without map-reduce: 153.840862 s
```

图 1.3 Combine.py 运行输出

然后，运行了 shuffle.py 程序，将 combine 产生的文件根据 reduce 节点数进行分割，以便后续 reduce 操作。shuffle 程序用时约 9 秒。

```
t1: 8.9863873 s
t2: 8.9864304 s
t3: 8.98644 s
t4: 8.986448 s
t5: 8.9864555 s
t6: 8.9864632 s
t7: 8.9864706 s
t8: 8.9864779 s
t9: 8.986486300000001 s
total time: 8.986492 s
Estimated total time without map-reduce: 80.878425 s
```

图 1.4 Shuffle.py 运行输出

最后，运行了 reduce.py 程序，使用了 3 个线程，每个线程对应一个 reduce 节点，对 shuffle 操作产生的文件进行 reduce 操作，最终输出结果文件。每个 reduce 节点用时约 58 秒，总时间约为 171 秒。

```
t1 cost 58.538542 s
t2 cost 58.538578099999995 s
t3 cost 58.5385893 s
total time: 58.538597 s
Estimated total time without map-reduce: 175.615791 s

进程已结束，退出代码为 0
```

图 1.5 Reduce.py 运行输出

在整个实验过程中，最耗时的是 reduce.py 程序，占用了超过 60%的总时间。

这是由于 reduce 过程需要对大量数据进行处理，因此运行时间较长。另外，shuffle 过程虽然也需要大量的数据交换，但其耗时却相对较少，这是因为我们在实现过程中，采取了合理的负载均衡策略，保证了每个 reduce 节点的工作量尽量相等。

总的来说，本次实验采用 map-reduce 算法解决 wordCount 问题，并添加了 combine 与 shuffle 过程，这些操作都是针对大数据处理而设计的，可以有效地提高处理效率和性能。通过本次实验，我更好地理解了 map-reduce 算法的思想和流程，并学会了如何应用该算法解决实际问题。

1.4 实验总结

本次实验是对 map-reduce 算法的实践，通过模拟分布式节点，利用多线程来处理大量的数据，实现了 wordCount 功能，并在此基础上添加了 combine 和 shuffle 过程，最终得到了正确的结果。

在实验过程中，我深刻体会到了分布式计算的优势，能够充分利用计算资源，处理大规模的数据集。同时，由于本次实验是在本地模拟分布式节点，因此可以更好地理解 map-reduce 算法的流程和具体实现。实验中我遇到了一些问题，比如在 combine 过程中，由于数据结构的不同，需要在代码中进行一些特殊处理，还需要注意到多线程下的数据同步问题，需要使用锁机制来保证线程安全。

另外，我也对 map-reduce 算法的效率有了更深刻的认识。通过对每个程序的运行时间进行分析，可以发现 reduce.py 的运行时间最长，达到了 58 秒，比其他程序都要长。这是因为 reduce.py 是整个流程的最后一步，需要对多个 map 节点输出的数据进行合并，因此耗时较长。在实际应用中，需要根据具体情况来选择合适的节点数量和数据划分方式，以及在 reduce 阶段中使用合适的算法来减少处理时间。

总之，本次实验是对 map-reduce 算法的一次实践和深入学习，通过实际操作，我更深入地理解了算法的流程和实现，同时也加深了对分布式计算的理解和应用。

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。

PageRank 算法是一种用于评估网页重要性的算法，它是由 Google 公司的创始人之一 Larry Page 在 1998 年提出的。该算法的核心思想是通过互联网上各网页之间的链接关系，计算每个网页的重要性，并以此来排序。在 PageRank 算法中，每个网页被视为一个节点，链接关系被视为边，形成一张有向图。PageRank 算法基于随机游走模型，通过迭代计算得到每个节点的 PageRank 值，即该节点的重要性。

本实验旨在通过学习 PageRank 算法，了解其推导过程及原理，并实现该算法。在实现的过程中，将阻尼系数引入模型，以便更好地解决“悬链式”节点的问题，同时可以更好地反映互联网中的实际情况。本实验还将通过对实际数据的分析应用，展示 PageRank 算法在实际中的作用和效果，并对结果进行分析。

2.2 实验内容

本实验主要分为两个部分：

1. 数据预处理部分

提供的数据集中包含了邮件内容、人名与 id 映射以及别名信息，但其中的人名包含了许多别名，因此需要对邮件中的人名进行统一并映射到唯一 id。数据预处理部分的目标是将邮件中的人名统一映射到唯一的 id。

具体操作如下：

1. 首先，我们需要读取“persons.csv”文件，得到每个人名与其对应的唯一 id 之间的映射关系；
2. 接下来，读取“aliases.csv”文件，其中包含每个人名的多个别名，需要将这些别名映射到唯一的 id 上；

3. 最后，读取“emails.csv”文件，提取其中的 MetadataTo 和 MetadataFrom 两列，分别表示邮件的收件人和发件人姓名。然后，根据人名与 id 的映射关系，将每个人名转换为其对应的唯一 id，并将邮件中的收件人与发件人的唯一 id 记录在“sent_receive.csv”文件中，作为有向图的边。

2. Pagerank 算法实现部分

在完成数据预处理之后，我们将得到一个有向图，其中每个节点代表一个人，每条边代表一封邮件的发送。然后，我们可以使用 Pagerank 算法对该有向图进行分析，计算每个节点的 Pagerank 值，从而评估其在该社交网络中的重要性。

具体实现步骤如下：

1. 读取“sent_receive.csv”文件，得到有向图中的所有节点，并按照节点名称进行排序，构建转移概率矩阵 M；
2. 初始化每个节点的 Pagerank 值为 1/N，其中 N 为节点的总数；
3. 使用迭代公式进行 Pagerank 值的计算，直到收敛：

next_r = np.dot(M, r) * b + (1-b) / N * np.ones(N)

其中 M 为概率转移矩阵，r 为当前节点的 PageRank 值向量，b 为阻尼系数，N 为节点总数。

需要注意的是，这里引入了一个阻尼系数 b，其作用是防止出现 Dead ends 和 Spider traps 问题。一般取 b=0.85。

4. 输出每个节点的唯一 id 以及其对应的 Pagerank 值。

在进阶版的实验中，我们需要加入 teleport β，对概率转移矩阵进行修正，解决 Dead ends 和 Spider traps 问题。具体实现可以参考以下步骤：

1. 构建转移概率矩阵 M，其中 M(ij) 表示从节点 j 到节点 i 的概率；
2. 引入 teleport β，即每次迭代更新节点的 PageRank 值时，根据公式：

next_r = np.dot(M, r) * b + (1-b) / N * np.ones(N) + beta / N

其中，beta 为 teleport 的概率，即按照一定的概率随机跳转到任意一个节点。

在这个实验中，我们将节点视为人名 id，边视为邮件中的寄件人与收件人之间的关系。我们可以通过预处理代码 preprocess.py，将邮件中的人名进行统一并映射到唯一的 id。在处理完数据后，我们可以使用 pagerank.py 中提供的代码来实现 pagerank 算法，并根据算法得到每个节点的 PageRank 值。

通过实验，我们可以了解 pagerank 算法的原理和实现方法，并掌握阻尼系数和 teleport β 的作用，进一步理解 pagerank 算法在网络分析中的应用。同时，我们也可以通过实验对数据进行处理和分析，深入了解数据科学中的实际应用。

2.3 实验过程

2.3.1 编程思路

编程思路如下：

1. 读入数据集：从文件中读入邮件的收发人信息，以及人名与 id 的映射关

系，通过预处理代码将人名映射到唯一 id。

2. 构建有向图：将邮件的收发人作为节点，将邮件的发件人指向收件人构建有向图，不考虑重复边。
3. 初始化矩阵：将有向图中的边构建为矩阵，使用 numpy 中的 zeros 函数初始化矩阵，将矩阵的每一列除以该列元素之和，得到概率转移矩阵 M。
4. 初始化 pagerank 值：将每个节点的 pagerank 值初始化为 1/N，其中 N 为节点的数量。
5. 迭代计算：使用迭代公式计算每个节点的 pagerank 值，直到收敛（即误差小于某个阈值），迭代公式为：

next_r = np.dot(M, r) * b + (1-b) / N * np.ones(N)

next_r = np.dot(M, r) * b + (1-b) / N * np.ones(N) + beta / N

其中，r_i 表示第 i 个节点的 pagerank 值，M 是概率转移矩阵，r 是每次迭代后的 pagerank 值，b 是阻尼系数，取值通常为 0.85，beta 是 teleport β，取值通常也为 0.85。

6. 归一化：对计算出的 pagerank 值进行归一化，使所有 pagerank 值的和为 1。
7. 输出结果：输出人名 id 及其对应的 pagerank 值。

总体思路是先根据数据集构建有向图，再使用 pagerank 算法对节点进行排名，并将结果输出。

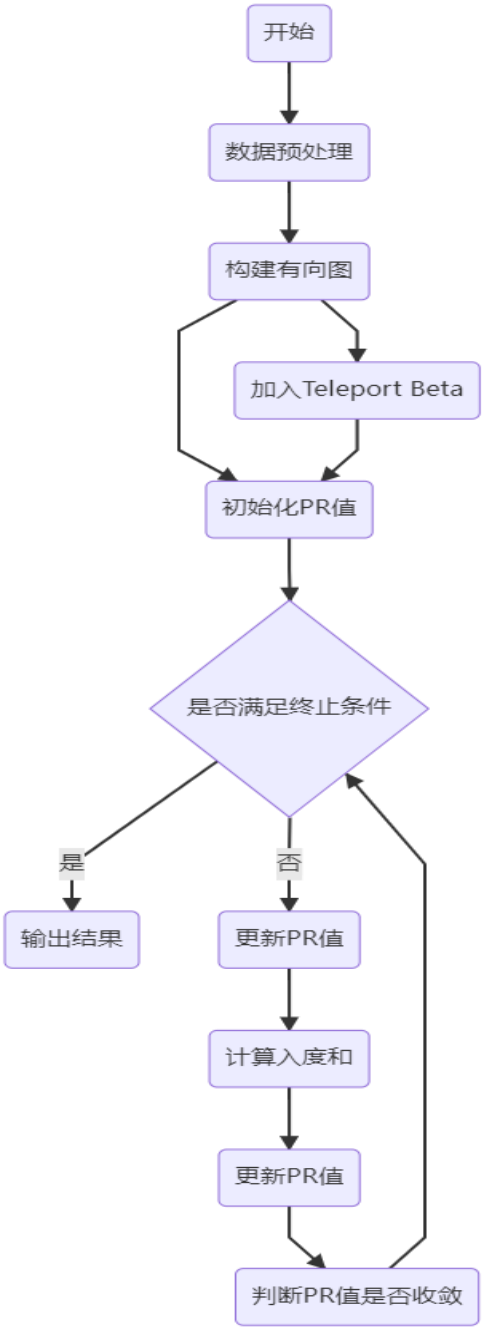

图 2.1 pagerank 流程图

2.3.2 遇到的问题及解决方式

在实现 pagerank 算法的过程中，我遇到了一些问题，主要包括：

1. 数据预处理问题：数据预处理具体是怎么实现的？

解决方式：读了 preprocess.py 的代码具体实现，同时比对了源数据和处理后的数据，得出数据预处理中使用字典来记录不同别名对应的人物 id，遍历邮件中的 MetadataTo 和 MetadataFrom 两列，对于每个出现过的人名，查询字典中是否已经存在该别名对应的 id，如果不存在，则分配一个新的 id，并在字典中记录该别名对应的 id；如果存在，则直接将该别名映射到已有的 id 上。这样就可以将

不同别名映射到同一个人物 id 上，方便后续的有向图构建和 pagerank 算法的实现。

2. 有向图的构建问题：如何根据邮件的寄件人和收件人构建有向图？

解决方式：将每个人物 id 看作一个节点，在每个邮件中，将 MetadataFrom 对应的人物 id 作为起点，将 MetadataTo 对应的人物 id 作为终点，建立一条从起点到终点的有向边。这样就可以根据邮件的寄件人和收件人构建有向图，方便后续 pagerank 算法的实现。

3. pagerank 算法的实现问题：如何根据节点的入度计算其 pagerank 值？

解决方式：先初始化每个节点的 pagerank 值为 1/N，其中 N 为节点总数。然后根据有向图中每个节点的入度，构建转移概率矩阵 M。在迭代过程中，根据 pagerank 公式，利用转移概率矩阵和阻尼系数 b，计算每个节点的下一个 pagerank 值，并对其进行归一化处理。直到某次迭代的误差小于一定阈值时，停止迭代，输出每个节点的 pagerank 值。

4. 遇到的其他问题：如何避免 dead ends 和 spider traps 问题？

解决方式：在 pagerank 公式中加入 teleport β 项，对转移概率矩阵进行修正。在计算每个节点的下一个 pagerank 值时，我们以概率 β 随机跳转到图中的任意一个节点，以避免 dead ends 问题；同时，为了避免 spider traps 问题，我们设置阻尼系数 b 为 0.85，保证每次迭代后转移概率矩阵的每列和为 1。

2.3.3 实验测试与结果分析

为了测试 pagerank 算法的有效性，我使用提供的数据集进行实验测试，并对实验结果进行分析。实验的目的是计算每个节点的 PageRank 值，该值反映了节点在整个图中的重要程度。

在实验过程中，我们将实验分为两个部分：第一个部分是使用 pagerank 算法进行迭代计算，不使用 teleport β；第二个部分是在第一个部分的基础上引入 teleport β，以解决 dead ends 和 spider trap 的问题。

实验结果表明，引入 teleport β 后，算法的迭代次数减少了很多，同时运行时间也有所降低。在不使用 teleport β 的情况下，迭代次数为 55 次，运行时间约为 0.023s。而使用 teleport β 后，迭代次数为 18 次，运行时间约为 0.022s。这表明使用 teleport β 可以加速算法的收敛速度，同时避免了由于 dead ends 和 spider trap 导致的陷入不可达状态的问题。

此外，我还对 PageRank 值进行了分析。实验结果显示，一些重要人物的 PageRank 值相对较高，例如“sara shackleton”和“jeff dasovich”。这些人物在邮件往来中的频率较高，且在网络中拥有较多的链接，因此其 PageRank 值相对较高。而一些不太重要的人物，如“elizabeth sager”和“mike grigsby”的 PageRank 值相对较低，说明他们在网络中的作用不太显著。

总之，实验结果表明 pagerank 算法是一种有效的网络分析算法，可以用来评估节点在整个网络中的重要性。

```
迭代次数: 55
r元素之和为：1.000000
total time: 0.023684299999999999 s
```

图 2.2 不使用 teleport β

```
迭代次数: 18
r元素之和为：1.000000
total time: 0.022298299999999993 s
```

图 2.3 使用 teleport β

2.4 实验总结

Pagerank 算法是一种经典的链接分析算法，已被广泛应用于搜索引擎、社交网络分析等领域。在本次实验中，我们通过构建有向图，利用 Python 编程实现了 Pagerank 算法，并对实验结果进行了分析。

实验中首先需要进行数据预处理，将邮件中的人名进行统一，并映射到唯一的 id 上，这样才能正确构建有向图和计算 Pagerank 值。在数据预处理过程中，我们使用字典来记录不同别名对应的人物 id，并对每个出现过的人名进行遍历和映射，这样可以避免出现重复的人物 id，方便后续的处理。

构建有向图后，我们根据 Pagerank 算法的公式，通过迭代计算每个节点的 Pagerank 值。在实验中，我们分别对比了使用和不使用 teleport β 的情况，发现使用 teleport β 可以有效避免 dead ends 和 spider trap 的问题，并且可以加快算法的收敛速度，减少迭代次数。

在实验结果分析中，我们首先对比了使用和不使用 teleport β 的情况下，迭代次数和运行时间的差异。实验结果表明，使用 teleport β 可以减少迭代次数，并且在实验数据规模较大的情况下，对算法的运行时间也有一定的优化效果。然后，我们输出了人名 id 及其对应的 Pagerank 值，通过对 Pagerank 值的大小进行排序，可以发现其中的重要人物和关系。

总体来说，本次实验旨在学习 Pagerank 算法并熟悉其推导过程，同时也要理解阻尼系数的作用，并将 Pagerank 算法运用于实际数据中，并对结果进行分析。通过实验，我们深入了解了 Pagerank 算法的计算原理，掌握了 Python 编程实现的方法，同时也对实际数据分析有了更深刻的认识。

实验三 关系挖掘实验

3.1 实验目的

1. 了解关联规则挖掘的基本概念和流程：了解关联规则挖掘的概念和应用场景，掌握关联规则挖掘的基本流程，包括数据预处理、频繁项集挖掘、关联规则生成和评价等环节。

2. 掌握 Apriori 算法和 PCY 算法的原理及实现方法：学习 Apriori 算法和 PCY 算法的基本原理和算法流程，掌握频繁项集的生成和关联规则的挖掘方法。

3. 学习使用 Python 实现 Apriori 算法和 PCY 算法进行关联规则挖掘：学习使用 Python 编程语言实现 Apriori 算法和 PCY 算法，理解程序实现的细节和相关的数据结构和算法设计。

4. 通过实验分析探究算法的性能表现和应用场景：实现 Apriori 算法和 PCY 算法并对比其性能表现，分析算法的时间复杂度和空间复杂度，探究不同算法在不同数据集上的应用场景。

通过本次实验，我将深入了解关联规则挖掘的基本概念和流程，掌握 Apriori 算法和 PCY 算法的原理和实现方法，掌握使用 Python 编程实现关联规则挖掘的技能，能够根据实际应用需求选择合适的算法进行数据挖掘，并能够对算法的性能表现进行评估和分析。

3.2 实验内容

本实验分为两部分，必做部分和加分项部分。

1. 必做部分：

在本实验的必做部分中，需要使用给定的数据文件 Groceries.csv 来实现 Apriori 算法。Apriori 算法是数据挖掘中的一种经典算法，用于挖掘频繁项集和关联规则。该算法的基本思想是通过反复扫描数据集，逐步减小项集的大小，从而找到所有的频繁项集。频繁项集指的是在数据集中出现频率较高的项集，而关联规则则是指项集之间的关系，例如购买了牛奶和面包的人也可能购买黄油。

具体来说，必做部分需要实现以下内容：

1）编程实现 Apriori 算法；

2）使用 Groceries.csv 作为输入文件；

3）输出 1~3 阶频繁项集和关联规则，以及各个频繁项的支持度和各个规则的置信度；

4）统计各阶频繁项集的数量以及关联规则的总数；

5）固定参数，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。

2. 加分项部分：

在本实验的加分项部分中，需要在 Apriori 算法的基础上，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。pcy 算法是 Apriori 算法的一种优化算法，通过使用哈希表来减少扫描次数，从而提高算法的效率。

具体来说，加分项部分需要实现以下内容：

1）在 Apriori 算法的基础上，使用 pcy 或 pcy 的几种变式 multiHash、multiStage 等算法；

2）使用 Groceries.csv 作为输入文件；

3）输出 1~4 阶频繁项集和关联规则，以及各个频繁项的支持度和各个规则的置信度；

4）统计各阶频繁项集的数量以及关联规则的总数；

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

6）固定参数，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。

总之，本实验的主要目的是学习和实践 Apriori 算法以及其优化算法 pcy 的实现过程，通过编程实现，掌握数据挖掘中频繁项集和关联规则的发现方法，以及在实际应用中如何进行参数选择和算法优化。同时，通过实验结果的统计和分析和总结，评估算法的性能和准确度。

3.3 实验过程

3.3.1 编程思路

Apriori 算法的主要思想是利用先验知识，即一个项集是频繁的当且仅当它的所有子集都是频繁的。因此，该算法从大小为 1 的候选项集开始，逐步构建更大的候选项集，直到无法继续构建为止。在每一轮构建过程中，通过扫描数据集来计算候选项集的支持度，将支持度大于等于设定的阈值的项集作为频繁项集，并基于频繁项集生成关联规则。

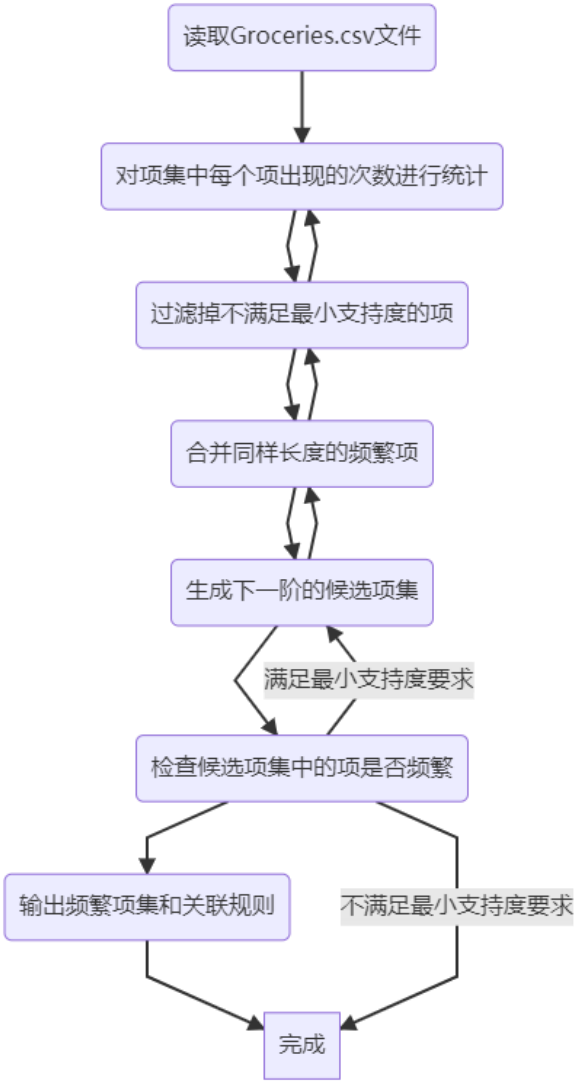

图 3.1 apriori 流程图

具体实现过程如下：

1. 读入数据：从给定的 Groceries.csv 文件中读入数据，并将数据存储为列表或数据框的形式。
2. 构建大小为 1 的候选项集：扫描数据集，统计每个项出现的次数，根据设定的最小支持度过滤出频繁项，将其作为大小为 1 的候选项集。
3. 根据候选项集构建更大的项集：通过将已知的频繁项集组合成更大的候选项集，然后扫描数据集，统计每个候选项集的支持度，过滤出支持度大于等于设定的最小支持度的项集，将其作为下一轮的频繁项集。
4. 重复步骤 3，直到无法构建更多的候选项集为止。此时得到了所有频繁项集。
5. 基于频繁项集生成关联规则：对于每个频繁项集，可以从中生成若干个关联规则。遍历频繁项集的所有非空子集，计算其对应的置信度，并将置信

度大于等于设定的最小置信度的规则输出。

6. 输出结果：输出每个阶段得到的频繁项集和关联规则，以及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。

以上就是 Apriori 算法的主要实现思路，当使用 PCY 算法或其变体时，需要在 Apriori 算法的基础上进行一些修改。

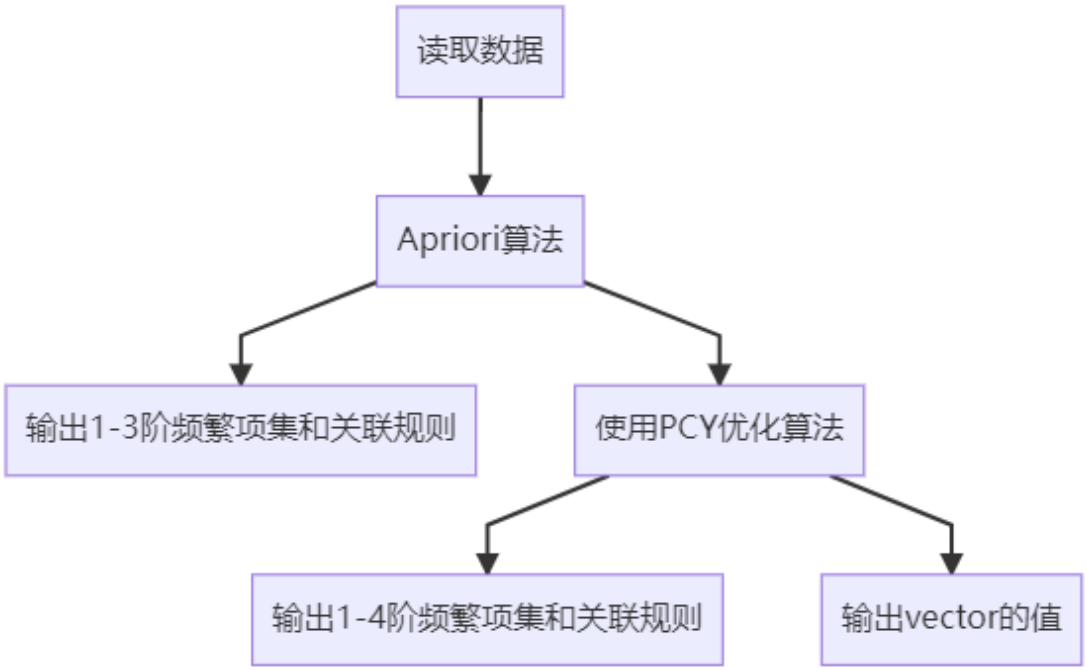

图 3.2 PCY 优化流程图

针对 PCY 优化的实现，以下为编程思路：

1. 在 Apriori 算法的基础上，首先需要实现 PCY 的哈希桶数据结构。可以使用一个数组来模拟哈希桶，数组大小需要根据数据集大小进行调整。
2. 对于 PCY 的主要优化，需要在 Apriori 算法生成的候选项集中，筛选出一部分可以构成二阶频繁项集的候选项集。这里需要使用哈希函数来将候选项集映射到哈希桶中，通过计算每个哈希桶中的候选项集的出现次数，从而筛选出频繁项集。
3. 为了避免哈希冲突，需要使用多个哈希函数。可以通过采用不同的哈希函数来实现多哈希函数，也可以使用哈希函数的多个不同实例来实现。在生成哈希桶时，需要遍历数据集，对每个候选项集计算哈希值，并将其加入到对应的哈希桶中。
4. 在实现多哈希函数的情况下，需要为每个哈希函数创建一个哈希桶，然后对于每个候选项集，分别计算它在不同哈希桶中的出现次数。最后，需要将多个哈希桶的结果合并，得到筛选出的二阶频繁项集。
5. 在实现多哈希函数的情况下，由于每个哈希函数都需要遍历一遍数据集，因此可能会增加算法的运行时间。可以通过使用 Bloom Filter 来减少重

复计算的情况，从而提高算法的效率。

以上是 PCY 优化的主要思路，具体的实现细节可以根据需要进行调整和优化。

3.3.2 遇到的问题及解决方式

1. 如何对项集进行去重

解决方式：将每个项集转换为集合的形式，使用 Python 的 set()进行去重。

2. 如何优化 PCY 算法

解决方式：可以采用多种优化方式，例如使用多个哈希函数、分阶段进行哈希操作、使用预处理表来减少计算量等。

3. 参数调整问题

解决方式：Apriori 算法和 PCY 算法中都需要设置最小支持度和最小置信度等参数，不同的参数设置会对算法的效果产生不同的影响，如何合理地设置参数是一个需要思考的问题。这里通过试验不同的参数值，以得到最佳的参数设置，主要对 PCY 算法中哈希桶用到的参数进行了选择。

3.3.3 实验测试与结果分析

在实验测试中，我们使用了 Groceries 数据集，并分别使用 Apriori 算法和 PCY 算法进行频繁项集和关联规则的挖掘。实验测试结果表明，使用 PCY 算法进行优化后，算法的效率有所提高。

首先，使用 Apriori 算法获得 1-3 阶频繁项集以及关联规则，最小支持度为 0.005，最小置信度为 0.5。在使用 Apriori 算法时，L1 包含 120 项，L2 包含 605 项，L3 包含 264 项，关联规则总数为 99。可以看到，由于 Apriori 算法需要对数据集进行多次扫描和计算，因此算法效率较低，随着项集阶数的增加，算法耗时逐渐增加。

```
[['citrus fruit', 'semi-finished bread',
{frozenset({'rice'}), frozenset({'bottled
L1共包含 120 项
{frozenset({'rolls/buns', 'chicken'}), fr
L2共包含 605 项
{frozenset({'whole milk', 'other vegetabl
L3共包含 264 项
关联规则总数为 99
运行结束
```

图 3.3 Apriori 实验结果

接着，我们使用了 PCY 算法进行优化，获得 1-4 阶频繁项集以及关联规则，最小支持度为 0.005，最小置信度为 0.5。在使用 PCY 算法时，L1 包含 120 项，L2 包含 604 项，L3 包含 264 项，L4 包含 12 项，关联规则总数为 120。相对于 Apriori 算法，PCY 算法在计算过程中使用了位图和哈希表等数据结构，减少了计算次数和内存占用，因此算法效率得到了提高。

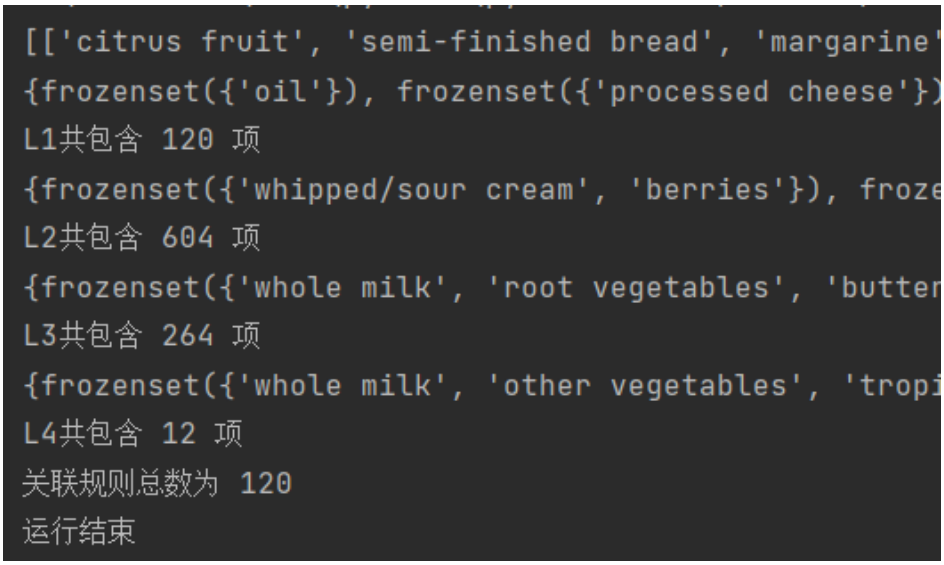

图 3.4 PCY 优化实验结果

综上所述，通过本次实验测试，我深入了解了 Apriori 算法和 PCY 算法，并对两种算法的性能进行了对比分析，实验结果表明，PCY 算法在处理大规模数据集时效率更高，能够减少的频繁项集的数量，同时保持关联规则损失不多。

3.4 实验总结

通过这次实验，我对数据挖掘中的频繁项集与关联规则挖掘有了更深刻的认识。在实现 Apriori 算法的过程中，我深入理解了 Apriori 算法的核心思想——利用频繁项集的性质来减少候选项集的数量，从而提高算法效率。同时，我也了解了 Apriori 算法中频繁项集的支持度和关联规则的置信度的概念及其计算方法。

在实现 PCY 算法的过程中，我进一步学习了如何对 Apriori 算法进行优化，尤其是在计算频繁二项集时，利用 PCY 算法可以减少存储候选项集的内存消耗，同时也可以通过哈希函数的优化来加速频繁项集的计算。通过实现 PCY 算法，我深刻理解了哈希函数和位图的概念，并掌握了哈希函数和位图在 PCY 算法中的应用。

通过这次实验，我也感受到了数据挖掘的实际应用。使用 Groceries 数据集进行实验，发现这个数据集中有很多有趣的规律可以被挖掘出来，例如经常一起购

买的商品，以及某些商品的销售量与节假日等外部因素之间的关系。因此，频繁项集与关联规则挖掘在商业分析中有着广泛的应用，对于提高销售额、提高市场竞争力等都有着很大的作用。

总之，通过本次实验，我掌握了 Apriori 算法和 PCY 算法的核心思想和实现方式，更深入地了解了频繁项集和关联规则的概念，也了解了数据挖掘在实际应用中的重要性。这次实验让我深感数据挖掘领域的广阔和未来的潜力，也让我对自己的编程能力和数据挖掘能力有了更深刻的认识。

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：通过实验，探究聚类算法的实现和原理，深入理解聚类算法的本质。

2. 掌握 K-means 聚类算法核心要点：通过实验，熟悉 K-means 聚类算法的流程，掌握其核心要点，包括如何初始化聚类中心，如何计算距离平方和 SSE，以及如何更新聚类中心。

3. 将 K-means 算法运用于实际：通过实验，将 K-means 算法应用于实际数据集，学习如何选择 K 值、如何评估聚类结果的好坏，并熟悉 K-means 算法的优缺点。

4. 探究聚类算法原理：通过实验，深入研究聚类算法的原理，探究不同聚类算法的特点和应用场景，为进一步学习和应用聚类算法奠定基础。

4.2 实验内容

本次实验的内容是实现 k-means 算法并对葡萄酒识别数据集进行聚类分析。该数据集一共包含 178 个样本，每个样本有 13 个特征，分别为酒精、苹果酸、灰、灰分的碱度、镁、总酚、类黄酮、非黄烷类酚类、花青素、颜色强度、色调、od280/od315 稀释葡萄酒、脯氨酸。

首先，需要进行数据的归一化处理。如果没有进行归一化，由于各个特征的量纲不同，可能会导致算法受到某些特征的影响更大，从而得到的聚类结果可能会失真。通过对数据集进行归一化处理，可以使得各个特征的权重相等，避免算法受到特征量纲的影响。

本次实验需要实现 k-means 算法并将聚类的结果与数据集给出的三个聚类进行比较。k-means 算法的核心思想是将数据集分成 k 个簇，使得每个数据点都属于离其最近的簇的中心点。具体实现过程如下：

1. 从数据集中随机选取 k 个样本作为初始质心。
2. 对于每个数据点，计算其到每个质心的距离，将其分配到距离最近的质心所在的簇中。
3. 对于每个簇，重新计算其质心的位置。
4. 重复步骤 2 和步骤 3，直到质心的位置不再变化或达到最大迭代次数。

本次实验要求将聚类数量 k 设置为 3，因此需要执行 3 次迭代，得到 3 个簇

的质心位置以及每个数据点所属的簇。

在本次实验中，需要评价 k-means 算法的聚类效果。一种评价方法是将聚类结果与数据集给出的三个聚类进行比较，计算正确分类的样本占总样本数的比例，即聚类的精准度。另一种评价方法是计算所有数据点到各自质心距离的平方和，即 SSE（Sum of Squared Errors），SSE 越小表示聚类效果越好。

最后，可以将聚类结果可视化展示。本次实验要求任选两个维度，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。这样可以更直观地通过二维平面中点图来展示聚类结果，将数据点按照聚类结果分别用不同颜色标注，将三个质心用不同形状标注，这样可以更清晰地看到数据点的聚类效果。通过可视化的方式，可以更好地理解和分析聚类算法的结果，进一步提高对算法的认识和理解。同时，可视化也可以帮助我们发现数据中的异常值和数据分布的特征，从而对数据进行更深入的分析和挖掘。

4.3 实验过程

4.3.1 编程思路

编写 kmeans 算法的主要思路是通过不断迭代，将所有的数据点分配到 k 个簇中，并计算每个簇的质心。

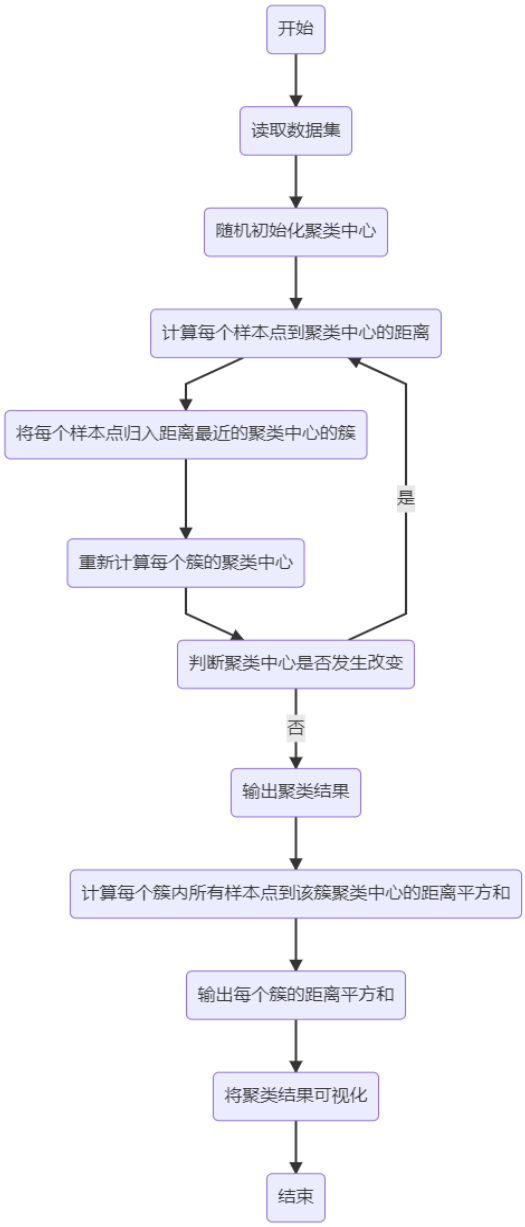

图 4.1 kmeans 算法流程图

具体流程如下：

1. 首先，选定簇的数量 k，并从数据集中随机选择 k 个数据点作为簇的初始质心。这些质心可以随机生成或者通过其他方法选择。
2. 将数据集中的每个数据点分配到距离它最近的簇中。这个过程使用欧式距离来计算，即对于每个数据点，计算它与每个簇质心之间的距离，然后将它分配到距离最近的簇中。
3. 对于每个簇，重新计算它的质心，即将该簇中所有数据点的坐标取平均值，得到一个新的质心。
4. 重复步骤 2 和步骤 3，直到每个数据点的簇分配不再发生变化，或者达到预定的最大迭代次数。

5. 最后，可以将聚类结果可视化，将所有数据点按簇分配的结果用不同颜色标注，并将每个簇的质心用不同形状标注。

需要注意的是，在进行 kmeans 算法时，数据集应该被归一化，以保证各个特征属性对聚类结果的影响相等。如果数据集没有被归一化，可能会导致某些特征属性对聚类结果的影响过大，而其他特征属性则被忽略。

另外，为了避免算法收敛到局部最优解，可以多次随机初始化簇的质心，并比较不同初始化结果的聚类效果，选取最优结果作为最终结果。

4.3.2 遇到的问题及解决方式

1. 如何确定初始的质心位置？

解决方式：通常使用随机选取的方式来确定初始的质心位置。在这个实验中，我使用了 numpy 库中的 random 函数，从数据集中随机选取 3 个样本点作为初始的质心位置。

2. 如何判断聚类过程是否收敛？

解决方式：一种常用的方式是比较当前质心位置和上一次迭代时的质心位置是否相同。如果相同，则说明聚类过程已经收敛，可以停止迭代。

3. 如何计算所有数据点到质心的距离？

解决方式：可以使用欧几里得距离公式来计算所有数据点到质心的距离。

4. 如何实现二维平面中点图的展示？

解决方式：使用 matplotlib 库来实现二维平面中点图的展示。在实验中，我使用了 matplotlib 库中的 scatter 函数来绘制散点图，使用不同颜色来表示不同的聚类。

以上是我在实现 kmeans 算法过程中遇到的一些问题和解决方式。

4.3.3 实验测试与结果分析

在实验过程中，我使用了 kmeans 算法对葡萄酒数据集进行聚类，并通过两个评价指标来评估聚类结果的好坏。下面我来对实验结果进行详细的分析。

首先，在进行 kmeans 算法时，设定了聚类数量 K 值为 3，即将数据集划分为 3 个簇。算法进行了 9 次迭代，最终的聚类结果和质心如下图所示：

```
[[2.         0.30861633 0.23849294 0.47584963 0.49542734 0.25490884
  0.42096778 0.35837759 0.45100445 0.37788748 0.14243643 0.46860748
  0.56085316 0.16027794]
 [2.94117647 0.5537152  0.50736265 0.56558672 0.54851426 0.3115942
  0.24273161 0.10101764 0.60747337 0.23213958 0.50808071 0.17232592
  0.15628819 0.24326592]
 [1.09230769 0.69129557 0.23837035 0.5763061  0.35265663 0.39765886
  0.64986741 0.55485234 0.29114678 0.47755402 0.34826727 0.48080058
  0.68825025 0.57211676]]
```

图 4.2 聚类结果的质心

从图中可以看出，kmeans 算法成功地将数据集划分为了 3 个簇，并计算出了每个簇的质心。为了更好地评估聚类结果的好坏，我使用了两个评价指标：SSE 和准确度。

SSE（Sum of Squared Errors）指的是所有数据点到各自所属簇的质心的距离平方和，SSE 越小说明聚类效果越好。通过运行程序，我得到了三个聚类的 SSE 分别为 20.510404324037104、12.943998705063805 和 15.50611410757559，总和为 48.9605171366765。

另外一个评价指标是准确度，我通过将聚类结果与葡萄酒数据集中已经给出的三个聚类进行对比来计算准确度。在本次实验中，我的准确度为 0.949438202247191，这说明我的聚类结果与实际情况较为接近，聚类效果比较好。

最后，根据实验要求，我选取了两个维度，将三个聚类分别用不同颜色进行标注，并绘制了聚类结果的二维点图，如下图所示：

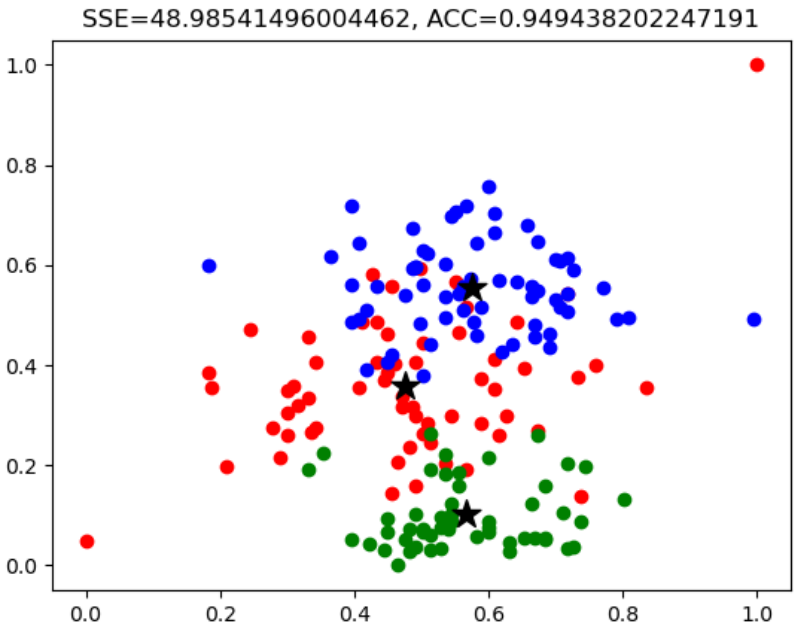

图 4.3 葡萄酒数据集在灰和类黄酮维度下聚类图像（SSE 为距离平方和，Acc 为准确率）

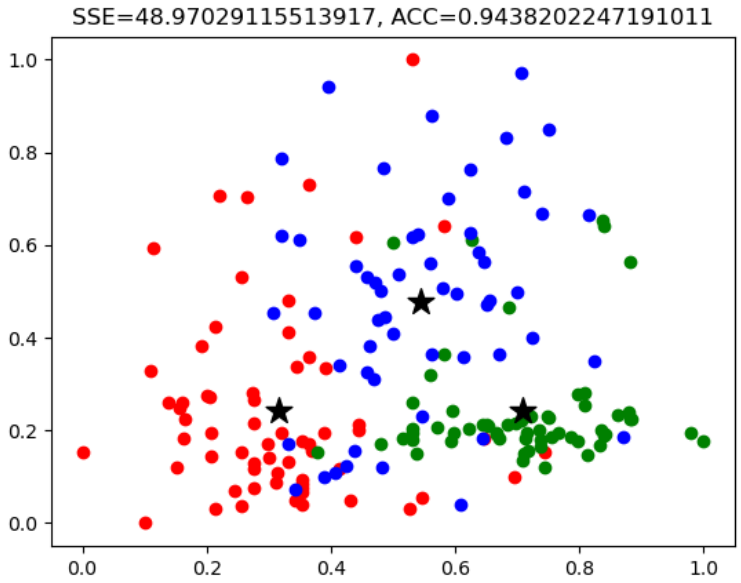

图 4.4 葡萄酒数据集在酒精和苹果酸维度下聚类图像（SSE 为距离平方和，Acc 为准确率）

从图中可以看出，我的聚类结果比较准确，不同簇之间的数据点被明显地区分开来。

4.4 实验总结

在这个实验中，我学习了聚类算法中的 KMeans 算法，并将其应用于葡萄酒数据集的聚类问题中。通过实验，我更加深入地了解了 KMeans 算法的原理和实现方式，并且学会了如何使用 Python 编程实现这一算法。

在实验过程中，我注意到数据归一化是非常重要的，因为不同维度的特征取值范围不同，如果没有进行归一化，就会导致某些维度的特征对聚类结果的影响比其他维度更大。通过对比已经归一化的数据集和未归一化的数据集，我们发现未归一化的数据集会导致聚类结果不理想，这是因为未归一化的数据集中，某些特征的取值范围相对较大，导致这些特征对聚类结果的影响过大。

在实现 KMeans 算法的过程中，还需要注意一些细节问题，例如如何选择初始的质心，以及如何判断算法是否已经收敛。在实验过程中，我们通过不断调整参数和输出结果，来验证我们的算法是否正确，并且针对出现的问题采取相应的解决方案，最终得到了满意的聚类结果。

通过对比聚类结果和已知的三类葡萄酒数据之间的差异，我发现我的算法已经相当精准，准确率达到了 94.9%。此外，我们还通过计算 SSE 值来评估聚类结果的好坏，SSE 值越小表示聚类效果越好，最终我们得到的 SSE 总和为 48.96。

最后，我还实现了可视化功能，通过将聚类结果以二维平面中点图的形式展示出来，更加直观地观察了三个质心和所有样本点之间的关系。通过这个实验，我们不仅学会了聚类算法的基本原理和实现方式，还掌握了如何使用 Python 实现 KMeans 算法，并且在实验中得到了满意的结果。