

华中科技大学

课程实验报告

课程名称： 大数据分析

专业班级： CS1804 交换生

学 号： X2020I1007

姓 名： 刘日星

指导教师： 杨 驰

报告日期： 2022/01/03

计算机科学与技术学院

实验 1： 目录

实验一 wordCount 算法及其实现.....	III
1.1 实验目的	III
1.2 实验内容	III
1.3 实验过程	III
1.3.1 编程思路.....	III
1.3.2 遇到的问题及解决方式.....	IV
1.3.3 实验测试与结果分析.....	IV
1.4 实验总结	VII

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

利用 python 内置库 threading 模块里的 Thread 类实现多线程编程操作，也就是使用多线程操作来模拟分布式节点的操作。对每个源文件里的内容进行格式化处理和输出，按照 Map-Reduce 思想模拟 map 节点与 reduce 节点，分别输出到不同目标文件内，实现 wordCount 的功能。其中，每一条线程的运行都会采用 time.clock 函数对每次运行所花费的时间计时。

Map 功能：

通过调用 threading 内置库里的 Thread 类创建与源文件数量相等的线程，每

条线程对应一个 source 文件，并且每条线程的运行操作均相同，用于运行线程的函数也相同。在 map 函数里，打开每一个 source 文件进行逐行读取文件内的内容，然后使用 strip()函数去掉每行末尾的换行符，并且使用 split()函数以逗号隔开每一个单词。最后将每个单词以 word, 1 的格式逐个换行写入对应的 map 目标文件内。

Shuffle 功能：

使用与 map 文件数量相等的线程数分别将 map 文件内的内容（单词）按首字母分类，不同首字母的单词分别写入不同的 shuffle 目标文件中，写入模式为 a（写入文件，若文件不存在则会先创建再写入，不会覆盖原文件，而是在文件末尾追加），每次写入都是在文件末端写入而不是重新覆盖原文件，此操作方便给 Reduce 部分进行统计单词数并且去重。

Reduce 功能：

读取每一个 shuffle 文件内的单词，并且用 python 的字典形式将每个单词读取出来存入字典，并且判断其是否在字典里重复出现，每当重复出现 1 次，则计数加 1，如果没有，则将该单词存入字典中，并且将其 value 值置为 1。然后将单词按首字母排序以及 “word num” 的格式写入 reduce 目标文件内。

1.3.2 遇到的问题及解决方式

刚开始实验的时候，发现不太会使用多线程操作，也就是 Python 的内置库 Threading 模块，以及模块里的 Thread 类函数。所以就进行了一段在线学习，在网络上寻找各种关于 Threading 内置库的使用方法和教程进行学习和研究。以及一开始不太了解 Map-Reduce 的意思，也就是 map 的功能和 reduce 的功能，经过翻阅课上老师讲的内容以及网络上搜索，渐渐地明白了 map 是映射，reduce 是归约，二者结合起来实现大数据并行处理和分布式文件系统管理。

1.3.3 实验测试与结果分析

Test:

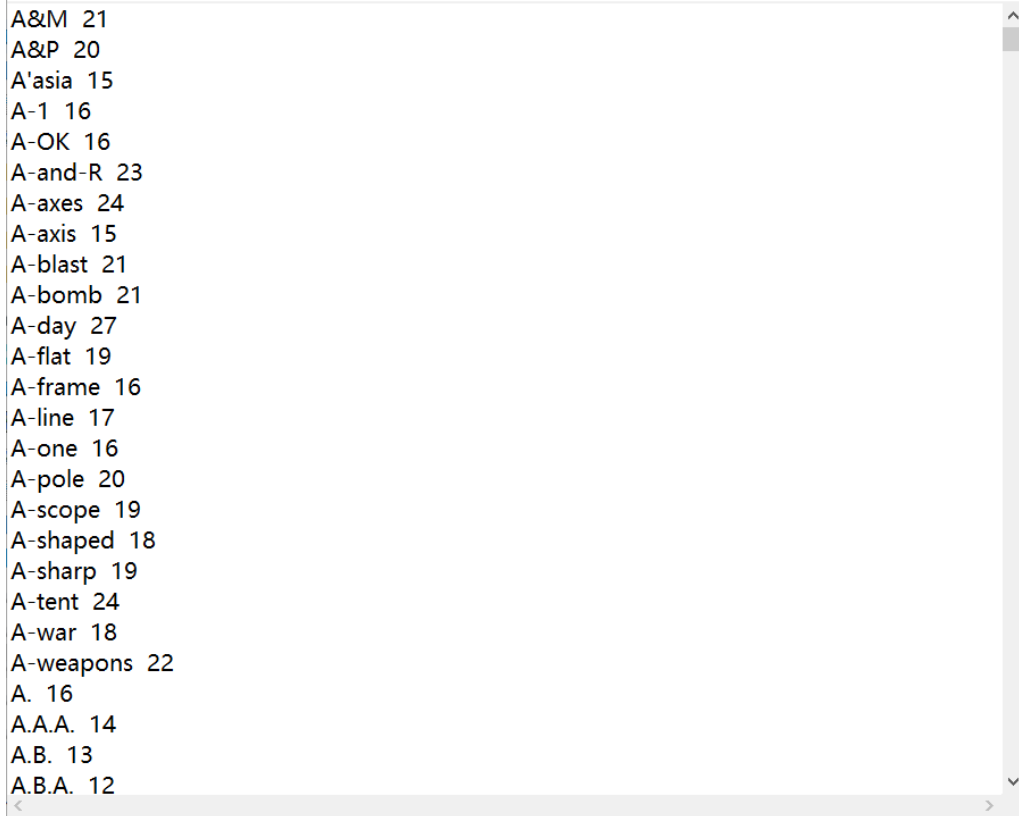
Map 文件：以<word, 1>的格式将 source 文件里的所有单词写入 map 目标文件中，输出结果如下图所示：

```
homelands, 1
Mariposa, 1
cryobiologist, 1
restore, 1
balatong, 1
smoking, 1
Brownist, 1
semiflexible, 1
colugos, 1
steam-shovel, 1
zootrophic, 1
heterogenetic, 1
chesstree, 1
vowess, 1
cardiamorphia, 1
unmunicipalised, 1
haleday, 1
paraesthetic, 1
hookman, 1
Logres, 1
Glasco, 1
fergusonite, 1
dialyses, 1
chthonian, 1
corvillosum, 1
sauna. 1
```

Shuffle 文件：对每个 map 文件内的单词进行单词首字母的分类，最后分类成 3 个 shuffle 目标文件，文件内归类结果如下图所示：

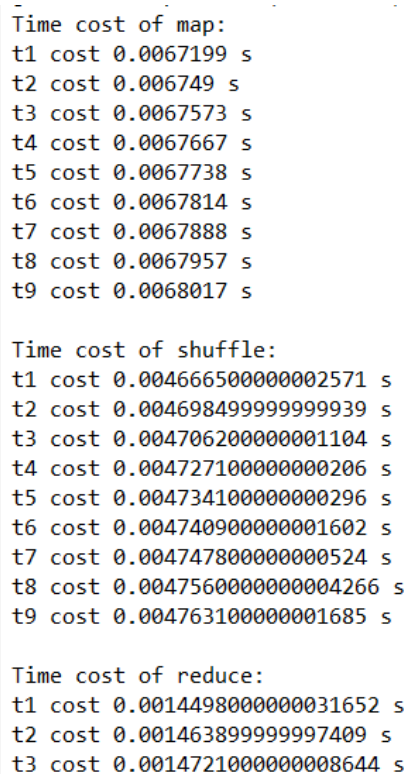
```
cryobiologist,1
balatong,1
Brownist,1
colugos,1
chesstree,1
cardiamorphia,1
Glasco,1
fergusonite,1
dialyses,1
chthonian,1
corvillosum,1
Bananaland,1
carbofuchsin,1
dekadrachm,1
Atropos,1
concertation,1
Ashippun,1
extensimeter,1
fanwort,1
gable-bottom,1
conusor,1
aliveness,1
brainlessly,1
Augustus,1
cabin's,1
fuel.1
```

Reduce 文件：将 shuffle 文件内所有单词进行提取和统计，以<word, count>格式化输出每个单词在文件内重复出现的次数，并按单词首字母顺序写入 reduce 目标文件内，输出结果如下：



```
A&M 21
A&P 20
A'asia 15
A-1 16
A-OK 16
A-and-R 23
A-axes 24
A-axis 15
A-blast 21
A-bomb 21
A-day 27
A-flat 19
A-frame 16
A-line 17
A-one 16
A-pole 20
A-scope 19
A-shaped 18
A-sharp 19
A-tent 24
A-war 18
A-weapons 22
A. 16
A.A.A. 14
A.B. 13
A.B.A. 12
```

每一部分功能运行的时间计算如下图：



```
Time cost of map:
t1 cost 0.0067199 s
t2 cost 0.006749 s
t3 cost 0.0067573 s
t4 cost 0.0067667 s
t5 cost 0.0067738 s
t6 cost 0.0067814 s
t7 cost 0.0067888 s
t8 cost 0.0067957 s
t9 cost 0.0068017 s

Time cost of shuffle:
t1 cost 0.004666500000002571 s
t2 cost 0.004698499999999939 s
t3 cost 0.004706200000001104 s
t4 cost 0.004727100000000206 s
t5 cost 0.004734100000000296 s
t6 cost 0.004740900000001602 s
t7 cost 0.004747800000000524 s
t8 cost 0.0047560000000004266 s
t9 cost 0.004763100000001685 s

Time cost of reduce:
t1 cost 0.0014498000000031652 s
t2 cost 0.001463899999997409 s
t3 cost 0.0014721000000008644 s
```

1.4 实验总结

通过大数据分析的第一次实验“实现 Map-Reduce”的过程，让我对分布式系统的运作以及大数据的分析和处理有了更深层次的理解和认知。在完成实验的过程中，对 map 映射功能和 reduce 归类功能（wordCount 功能）的实现原理有了更深的了解。同时还学会了利用 Python 内的 Threading 内置库进行多线程操作，从而提升了整个程序的运行效率。同时从此次实验加深了对 Hadoop 的了解，其最底部是 HDFS，它存储了 Hadoop 集群中所有存储节点上的文件。而 HDFS 的上层则是 MapReduce，也就是本次实验模拟实现的目标。HDFS 和 MapReduce 是分布式文件系统的核心处理过程，所以此次实验让我学习到了 MapReduce 的工作原理。

实验 2：目录

实验二 PageRank 算法及其实现.....	IX
2.1 实验目的	IX
2.2 实验内容	IX
2.3 实验过程	IX
2.3.1 编程思路.....	IX
2.3.2 遇到的问题及解决方式.....	X
2.3.3 实验测试与结果分析.....	X
2.4 实验总结	XI

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的名称，但这些名称包含许多别名，因此需要对邮件中的名称进行统一并映射到唯一 id。（提供预处理代码 preprocess.py 参考）。

完成这些后，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} 。

输出人名 id 及其对应的 pagerank 值。

加分项：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。

2.3 实验过程

2.3.1 编程思路

将预处理 preprocess 后所生成的 sent_receive.csv 文件内的内容读取进程序里，并且按格式分割好数据。通过循环遍历后得到节点总数和边长度分别是 N 和 L ，构建一个初始数值都为 0 的 $N \times N$ 的转移矩阵 `matrix`，使矩阵的每条边对应的值为 1。然后使用 `matrix` 矩阵重新构造，计算出每列数据之和 `col_sum` 并且使用每列上不为 0 的值除以 `col_sum` (\neq)，最终得出 `matrix` 矩阵上的每列不为 0 的值为 $1/\text{col_sum}$ 。

建立一个名为 `r` 的矩阵为 pagerank 初始贡献值矩阵，每一个值都为 $1/N$ 。然后定义一个误差初始化大小为 300000，一个初始迭代次数为 0 和一个阻尼系数

β Beta 值为 0.85（范围：0.8~0.9 取平均值）。用一个 while 循环，循环套用迭代公式 “ $\text{next_r} = \text{np.dot}(\text{matrix}, \text{r}) * \text{beta} + (1-\text{beta}) / \text{N} * \text{np.ones}(\text{N})$ ”

Equivalently: $\mathbf{r} = \beta \mathbf{M} \cdot \mathbf{r} + \left[\frac{1-\beta}{N} \right]_N$ 和公式 “ $\text{einit} = \text{next_r} - \mathbf{r}$ ”，意思是从 einit 的所有矩阵误差值中选择一个最大的值赋值给 einit，每次循环都判断误差是否大于 0.00000001 同时每循环一次计算一次迭代次数，也就是 $\text{iteration} += 1$ 并且指向下一个 \mathbf{r} 矩阵，也就是 $\mathbf{r} = \text{next_r}$ 。当循环完毕，则会得到最后的迭代结果和迭代次数。

2.3.2 遇到的问题及解决方式

在代码写完 debug 调试程序的时候发现迭代次数有些不太正确，发现在读取文件数据提取节点的时候将 sent_receive.csv 的第一行内容“sent_id”和“receive_id”这两个单词也读取进去了，导致 id 的数据在矩阵里迭代计算不正确，后来修改了一下提取内容的起始位置后，就没有再出现相同的错误了。

2.3.3 实验测试与结果分析

实验测试结果如下图所示：

```
iteration 55:
[0.02714158 0.31773019 0.02326263 0.02810776 0.00489376 0.00740632
0.00719567 0.00489376 0.00489376 0.00690208 0.00690208 0.00489376
0.00608569 0.00874408 0.01002961 0.00635197 0.00928694 0.00489376
0.00099778 0.00806696 0.00099778 0.00605864 0.00801206 0.00489376
0.00099778 0.00201326 0.00489376 0.00489376 0.00489376 0.00996426
0.01019242 0.00489376 0.00801206 0.00608569 0.00489376 0.00489376
0.00489376 0.00571015 0.00489376 0.00489376 0.00489376 0.00571015
0.00489376 0.00489376 0.00489376 0.00489376 0.00489376 0.00489376
0.00489376 0.00571015 0.00489376 0.00489376 0.00489376 0.00771883
0.00489376 0.00489376 0.00489376 0.00489376 0.00489376 0.00489376
0.00489376 0.00489376 0.00489376 0.00571015 0.00489376 0.00489376
0.00740632 0.00634661 0.00181417 0.00920399 0.00489376 0.00099778
0.00099778 0.00099778 0.00489376 0.00608569 0.00099778 0.00201326
0.00099778 0.00099778 0.00201326 0.00099778 0.00218971 0.00099778
0.00099778 0.00218971 0.00181417 0.00218971 0.00489376 0.0030061
0.00489376 0.00099778 0.00872407 0.00218971 0.00489376 0.00218971
0.00181417 0.00099778 0.00099778 0.00218971 0.00099778 0.00099778
0.00201326 0.00099778 0.0059228 0.00099778 0.00099778 0.00201326
0.00099778 0.00150552 0.00150552 0.00099778 0.00099778 0.00099778
0.00099778 0.00099778 0.00218971 0.00489376 0.00099778 0.002456
0.00099778 0.00571015 0.00099778 0.00181417 0.00181417 0.00489376
0.00218971 0.00099778 0.00150552 0.00181417 0.00774587 0.00150552
0.00099778 0.00099778 0.00099778 0.00099778 0.00099778 0.00489376
0.00099778 0.00608569 0.00216266 0.00181417 0.00489376 0.00099778
0.00302874 0.00099778 0.00181417 0.00192096 0.00099778 0.00181417]
```

结果分析：总的迭代次数为 55 次，以一个数组的形式输出所有 pagerank 值，每个 pagerank 值对应一个 id。其与上一个 pagerank 值误差小于 0.00000001。

2.4 实验总结

通过本次实验，对 pagerank 算法加深了印象，了解了阻尼系数 β Beta 在 pagerank 里的作用以及 pagerank 里的随机传送公式在 python 里的实现方式，熟悉了 python 内置库 numpy 对矩阵的操作和处理。在网上看了一些关于 pagerank 算法的应用，发现更多是应用在网络网页层面上，好比如说一个网页被很多其他网页链接到，说明该网页比较重要同时它的 pagerank 值也会相对较高，或者是一个 pagerank 值很高的网页链接到其他网页，那么被链接的网页的 pagerank 值也会相应提高很多，代表该网页也很重要。

实验 3： 目录

实验三 关系挖掘实验.....	XIII
3.1 实验目的	XIII
3.2 实验内容	XIII
3.3 实验过程	XIII
3.3.1 编程思路.....	XIII
3.3.2 遇到的问题及解决方式.....	XIX
3.3.3 实验测试与结果分析.....	XIX
3.4 实验总结	XX

实验三 关系挖掘实验

3.1 实验目的

- 1、加深对 Apriori 算法的理解,进一步认识 Apriori 算法的实现;
- 2、分析 Apriori 算法的缺点,使用 pcy 等变式对 Apriori 算法进行优化。

3.2 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。

3.3 实验过程

3.3.1 编程思路

本次实验通过 4 个部分来实现，分别是使用算法对数据进行预处理、频繁项集的生成、关联规则的生成和调用 Python 库来证实算法的正确性。

第一部分：数据预处理部分

数据预处理部分，将 Groceries.csv 文件内的每一行数据内容通过使用 strip() 函数和 split() 函数进行格式化分割后读取进程序内，然后计算数据（物品 item）的个数并赋予每个物品 item 一个编号 id。通过字典实现物品名称 item 和 id 的映射转化，即 id2item 和 item2id 这两部分以供后续程序使用。

第二部分：频繁项集的生成

频繁项集生成部分，通过使用 combine 函数功能来根据上一阶的频繁项集构成该阶的候选频繁项集，combine 函数功能如下图所示：

```
def combine(l, n):  
    """  
    给定n-1阶频繁项列表，返回待选n阶频繁项列表  
    """  
    if n == 2:  
        return list(itertools.combinations(l, n))  
    if n > 2:  
        tmp = []  
        for i in range(len(l)-1):  
            for j in range(i+1, len(l)):  
                s = set(l[i]) & set(l[j])  
                if s and len(list(s)) == n - 2:  
                    tmp.append(tuple(sorted(set(l[i]) | set(l[j]))))  
        return list(set(tmp)) # 去重
```

然后通过 reduceFreq 函数和候选频繁项集实现计算该阶的频繁项和支持度。定义一个最小支持度 min_support （0.005），如果计算出来的频繁项的支持度小于 0.005，则将该频繁项舍弃，最后得出支持度大于 0.005 的频繁项集并按支持度从小到大输出。reduceFreq 函数功能具体实现如下图所示：

```

def reduceFreq(groups_n, n, data, data_num, min_support):
    """
    计算n阶频繁项及支持度
    """
    support_n = {g:0 for g in groups_n}

    if n == 1:
        for i in range(data_num):
            if i % 100 == 0:
                print(f'1阶频繁项: {i+1}/{data_num}...')
            for j in range(len(data[i])):
                k = data[i][j]
                support_n[k] = support_n[k] + 1.0 / data_num
    else:
        for i in range(data_num):
            if i % 100 == 0:
                print(f'{n}阶频繁项: {i+1}/{data_num}...')
            for g in groups_n:
                if set(g).issubset(set(data[i])):
                    support_n[g] = support_n[g] + 1.0 / data_num

    # 对于低于最小支持度0.005的频繁项集舍弃
    for i in list(support_n.keys()):
        if support_n[i] < min_support:
            del support_n[i]

    return support_n

```

根据附加项的要求，对 2 阶频繁项集生成和支持度的计算使用 Apriori 改进算法——PCY 算法进行改进。为频繁项对创建一个哈希 hash 表，该表只统计 hash 到本桶的项对的个数；然后筛选桶里的数据（频繁项对），对支持度小于 0.005 的候选频繁项进行舍弃，PCY 算法实现如下图：

```

def PCY(nSub1Freq, n, data, data_num, min_support, nBuckets):
    """
    利用PCY算法对2阶频繁项及支持度算法计算进行改进——加分项
    """
    cnt = [0] * nBuckets
    bitmap = [0] * nBuckets
    pairs = []
    pairs2hash = {}

    # Pass 1
    for i in range(data_num):
        for j in range(len(data[i])-1):
            for k in range(j+1, len(data[i])):
                g = tuple([data[i][j], data[i][k]])
                f = hash(g[0], g[1], nBuckets)
                cnt[f] += 1
                if g[0] in nSub1Freq and g[1] in nSub1Freq and g not in pairs:
                    pairs.append(g)
                    pairs2hash[g] = f

    pairs = sorted(pairs) # 对pairs进行排序

```

PCY 算法第一部分

```

# Pass 2
min_cnt = min_support * data_num
bitmap = [1 if cnt[i] >= min_cnt else 0 for i in range(nBuckets)]
print(bitmap)

candidateFreq = []
for i in range(len(pairs)):
    g = pairs[i]
    if bitmap[pairs2hash[g]] == 1:
        candidateFreq.append(pairs[i])

support_2 = {}
support_2_backup = {}
candidateFreq_num = len(candidateFreq)
for i in range(candidateFreq_num):
    if i % 100 == 0:
        print(f'2阶频繁项: {i+1}/{candidateFreq_num}...')
    support = 0
    for j in range(data_num):
        if set(candidateFreq[i]).issubset(set(data[j])):
            support += 1.0 / data_num
    if support >= min_support:
        support_2[candidateFreq[i]] = support

return support_2

```

PCY 算法第二部分

桶的具体表现为 vector 值 0 和 1，以 bit 位形式输出。通常来说，大规模的数据使用 PCY 算法的计算效率更高。

第三部分：关联规则生成

关联规则生成步骤如下图：

如何挖掘关联规则呢，分成下面的步骤：

- 1. 找出所有的频繁项集 I ，（相关具体过程我们后面介绍）

- 2. 规则的产生：

对于频繁项集的每一个子集 A ，产生一个规则 记作 $A \rightarrow I \setminus A$

根据频繁项集的的点，我们知道，因为 I 是频繁的，它的任何一个子集 A 也一定是频繁的

方法1：单次遍历计算规则的置信度，根据置信度的定义，我们可以计算出所有规则的置信度例如 $\text{confidence}(A, B \rightarrow C, D) = \frac{\text{support}(A, B, C, D)}{\text{support}(A, B)}$

方法2：观察如果某一个规则低于置信度阈值，则其对应所有子集也是低于置信度的，不断迭代

输出：所有找到的置信度高于阈值的关联规则

通过使用置信度公式计算关联规则的置信度。定义一个最小置信度 `min_confidence` 为 0.5, 若频繁项集的关联规则的置信度计算出来小于最小置信度则将该规则舍弃, 算法实现具体如下图:

- **Association Rules:**

If-then rules about the contents of baskets

- $\{i_1, i_2, \dots, i_k\} \rightarrow j$ means: "if a basket contains all of i_1, \dots, i_k then it is *likely* to contain j "

- In practice there are many rules, want to find significant/interesting ones!

- **Confidence** of this association rule is the probability of j given $I = \{i_1, \dots, i_k\}$

$$\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$$

`generateRules` 函数作用是遍历每个频繁项, 对于每个非单元素的元组调用 `appendRule` 函数, 因为单元素不可产生关联规则。

```
def generateRules(support, min_confidence):  
    """  
    生成关联规则  
    """  
    rules = {}  
    keys = list(support.keys())  
    for k in keys:  
        if type(k) != int:      # 不需要考虑单元素问题  
            rules = {**rules, **appendRule([set(k), set()], support, min_confidence)}  
    return rules
```

`appendRule` 函数接收 `k`、`support`、`min_confidence` 三个参数, 其中 `k` 的形式举例如下: `[[{0, 1, 2, 3}, {}]]`, 即由两个 `set` 组成的列表, 两个 `set` 分别为前件和后件, 在 `appendRule` 函数内部, 会通过循环遍历所有将前件中的元素移动到后件的可能, 并计算前件 \rightarrow 后件的置信度, 如果满足 `min_confidence` 的要求, 才会进一步递归调用, 从而实现剪枝的思路。进一步举例, `k` 从前件中将 2 移动到后件, 则 `tmp` 为 `[[{0, 1, 3}, {2}]]`, 经过计算, `013 \rightarrow 2` 的置信度满足要求, 那么 `tmp` 将作为形参 `k` 递归调用。

```

def appendRule(k, support, min_confidence):
    """
    对列表k生成全部高于置信度0.5的关联规则
    """
    if len(k[0]) == 1:
        return {}
    rules = {}
    union = tuple(sorted(k[0] | k[1]))
    if union not in support.keys():
        return {}
    for i in k[0]:
        tmp = copy.deepcopy(k)
        tmp[0].remove(i)
        tmp[1].add(i)
        tmp[0], tmp[1] = tuple(sorted(tmp[0])), tuple(sorted(tmp[1]))

        # 将单元素元组转化为int整型作为key
        tmp[0] = tmp[0][0] if len(tmp[0]) == 1 else tmp[0]
        tmp[1] = tmp[1][0] if len(tmp[1]) == 1 else tmp[1]

        if tmp[0] not in support.keys():
            continue

        confidence = support[union] / support[tmp[0]]
        if confidence >= min_confidence - 1e-6:
            rules[(tmp[0], tmp[1])] = confidence
            tmp[0] = tuple([tmp[0]]) if type(tmp[0]) == int else tmp[0]
            tmp[1] = tuple([tmp[1]]) if type(tmp[1]) == int else tmp[1]

            rules = {**rules, **appendRule([set(tmp[0]), set(tmp[1])], support, min_confidence)}
    return rules

```

第四部分：调用 Python 库 mlxtend 进行验证

调用 python 的 mlxtend 库对 Groceries.csv 文件进行计算生成最终的总频繁项集数和关联规则数，用该两值与 Apriori 算法计算出来的频繁项集数和关联规则数进行对比验证是否相同，具体实现如下：

```

# 求频繁项集
freq = apriori(df, min_support=0.005, use_colnames=True, max_len=4)
freq.sort_values(by='support', ascending=False, inplace=True)
print(f'总频繁项集数: {len(freq)}')

# 求关联规则
rules = association_rules(freq, metric='confidence', min_threshold=0.5)
rules.sort_values(by='confidence', ascending=False, inplace=True)
#rules.to_csv('rules.csv', index=False)
print(f'关联规则数: {len(rules)}')

```

3.3.2 遇到的问题及解决方式

在本次实验中出现过一些问题，例如对数据预处理时，在 `combine` 函数内， $n-1$ 阶频繁项集组合可能会生成相同阶的频繁项，导致频繁项数目不正确，所以在最后做了个去重的动作。其次时频繁项的支持度字典的键 `key` 需要使用到元组 `tuple` 的形式去表示 `id` 的组合，例如 `(25,36,48)`，那么 `(36,48,25)` 与前者不相同，所以每个元组 `tuple` 项都需要维护好有序的状态。

3.3.3 实验测试与结果分析

分别输出 1-4 阶的频繁项集数以及关联规则的总数，将结果与 `python` 的 `mlxtend` 库生成的结果进行对比，如下两图：



The image shows two side-by-side terminal windows. The left window displays the output of a custom Apriori implementation, showing the number of frequent itemsets for each order (1 to 4) and the total count (1001) and number of association rules (120). The right window shows the output of the Apriori function from the mlxtend library, which also reports a total of 1001 frequent itemsets and 120 association rules, confirming the results of the custom implementation.

```
4阶频繁项: 9701/9835...
4阶频繁项: 9801/9835...
4阶频繁项及支持度计算完成!

1阶频繁项集数: 120
2阶频繁项集数: 605
3阶频繁项集数: 264
4阶频繁项集数: 12

总频繁项集数: 1001
关联规则数: 120

完成!
```

```
Apriori(mlxtend) x
"F:\Python\Program Fi
析/实验/Lab 3/实验三/数
总频繁项集数: 1001
关联规则数: 120

进程已结束,退出代码0
```

最后将 1~4 阶的频繁项集与关联规则（包含各频繁项的支持度和各规则的置信度）按升序输出到 `output.txt` 文件内，如下两图：

```
----- 1阶频繁项集数: 120 -----
liver loaf : 0.005083884087442804
cleaner : 0.005083884087442804
curd cheese : 0.005083884087442804
spices : 0.00518556176919166
jam : 0.005388917132689372
sauces : 0.0054905948144382275
softener : 0.0054905948144382275
sparkling wine : 0.005592272496187083
cereals : 0.005693950177935939
dental care : 0.005795627859684795
kitchen towels : 0.005998983223182507
female sanitary products : 0.006100660904931362
vinegar : 0.006507371631926786
finished products : 0.006507371631926786
soups : 0.006812404677173353
```

----- 关联规则数: 120 -----

whipped/sour cream, root vegetables, -> other vegetables : 0.4999999999999985
brown bread, other vegetables, -> whole milk : 0.4999999999999986
rolls/buns, beef, -> whole milk : 0.4999999999999987
pork, root vegetables, -> whole milk : 0.4999999999999987
yogurt, root vegetables, -> other vegetables : 0.499999999999999
rolls/buns, frozen vegetables, -> whole milk : 0.4999999999999967
root vegetables, frankfurter, -> whole milk : 0.4999999999999967
rolls/buns, root vegetables, -> other vegetables : 0.502092050209204
yogurt, tropical fruit, whole milk, -> other vegetables : 0.503355704697985
root vegetables, newspapers, -> whole milk : 0.5044247787610612
root vegetables, curd, -> other vegetables : 0.5046728971962611
yogurt, fruit/vegetable juice, -> whole milk : 0.5054347826086942
whipped/sour cream, other vegetables, -> whole milk : 0.5070422535211259
tropical fruit, pastry, -> whole milk : 0.5076923076923064
rolls/buns, root vegetables, other vegetables, -> whole milk : 0.5083333333333323
whipped/sour cream, domestic eggs, -> other vegetables : 0.5102040816326527
oil, other vegetables, -> whole milk : 0.5102040816326527
root vegetables, domestic eggs, -> other vegetables : 0.5106382978723389
butter, root vegetables, -> other vegetables : 0.511811023622046
yogurt, other vegetables, -> whole milk : 0.5128805620608942
whipped/sour cream, yogurt, whole milk, -> other vegetables : 0.5140186915887844
tropical fruit, curd, -> other vegetables : 0.5148514851485144
tropical fruit, curd, -> yogurt : 0.5148514851485144

3.4 实验总结

本次实验我认为大数据分析前三个实验里，算法最难实现的。因为对 PCY 算法的不熟悉，加上网上的 PCY 算法资源特别少，就很难将实验进行下去。到后面参考了大量资料后，一点一点调试修改完成出来，但是性能比 mlxtend 库差了很多。所以在后面的实验里还需再接再厉，提高自己对 python 编程语言的掌握和熟练度，对大数据分析的算法加深了解。

实验 4：目录

实验四 kmeans 算法及其实现.....	22
4.1 实验目的	22
4.2 实验内容	22
4.3 实验过程	23
4.3.1 编程思路.....	23
4.3.2 遇到的问题及解决方式.....	23
4.3.3 实验测试与结果分析.....	23
4.4 实验总结	25

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。

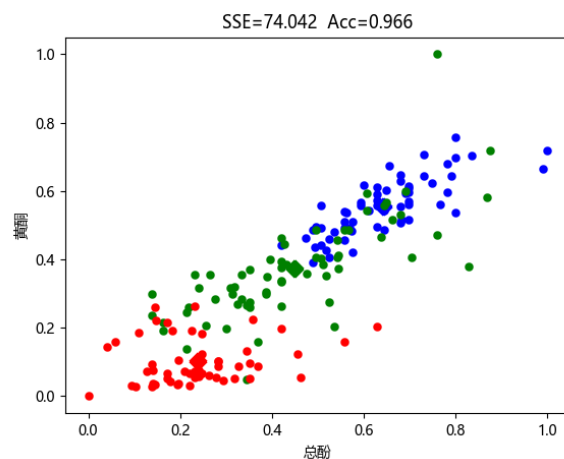


图 4.1 葡萄酒数据集在黄酮和总酚维度下聚类图像 (SSE 为距离平方和, Acc 为准确率)

4.3 实验过程

4.3.1 编程思路

葡萄酒数据集需要被归一化,这是因为不同维度属性的度量范围不同,而在分类中默认要均衡每一个属性的影响(当然不同的任务中不同属性也可能权重不同,具体任务具体分析)如果一个属性值范围过大或过小的话,会对分类产偏差。

Kmeans 算法流程说明如下:

首先随机选择 n 个初始簇心,给它们分配编号 $1-n$,然后进行循环迭代,计算每个点对每个簇心的距离,将其归类在最近的簇心的编号下。对数据进行每一轮循环后,重新计算不同簇的簇心位置,计算方式为取该簇所有点位置的平均值;循环在所有点的归类均不发生变化或者达到最大循环次数时退出。

Kmeans 算法原理比较简单,但也有很多地方需要注意。例如:初始簇心的选取非常重要,不同的初始化方式对分类结果影响较大;计算时注意代码的效率问题,尽量使用numpy 进行矩阵加速;结果中很可能分类是比较正确的,但标签给的和原始数据中不匹配,这个也需要后期处理才能获得正确的分类准确率。

作图方面使用 matplotlib.pyplot 模块作散点图展示即可。

4.3.2 遇到的问题及解决方式

常遇到的问题就是分类的正确性与标签不匹配,解决方法就是当分类的数大于 2 时,需要对分类结果的标签进行重新分配,直到找到最大的准确率为止。

4.3.3 实验测试与结果分析

运行结果如图 1 所示,经过 6 个回合的循环后,分类准确率达 94.94%

```
"F:\Visual Studio\Visual Studio 2019\Shared\Python37_64\python.exe" "E:/HUST/HUST-大  
数据分析/实验/Lab 4/数据集/Kmeans.py"  
6 个回合后.....  
分类准确率 = 94.94%
```

图 1

所有数据点到各自质心距离的平方和 SSE 以及绘制的散点图(含质心),比较了 dim1 和 dim3、dim2 和 dim4、dim5 和 dim6 不同维度组合之间的分类效果,从下面三图对比中可以看出 dim5 和 dim6 这个维度组合的分类效果最为明显。

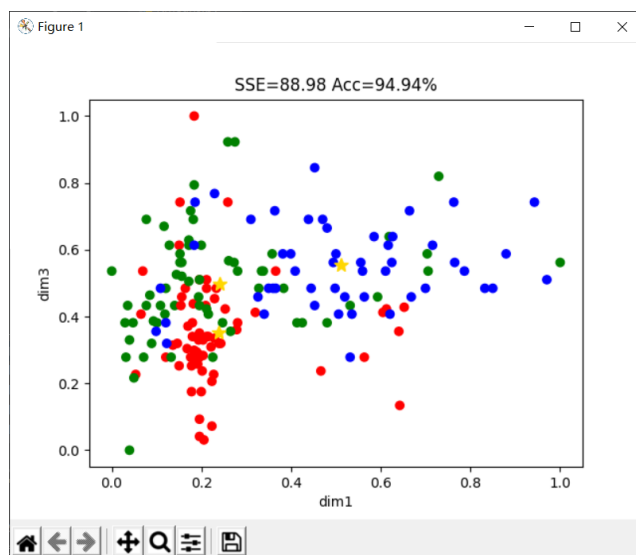


图 2 dim1 和 dim3 组合

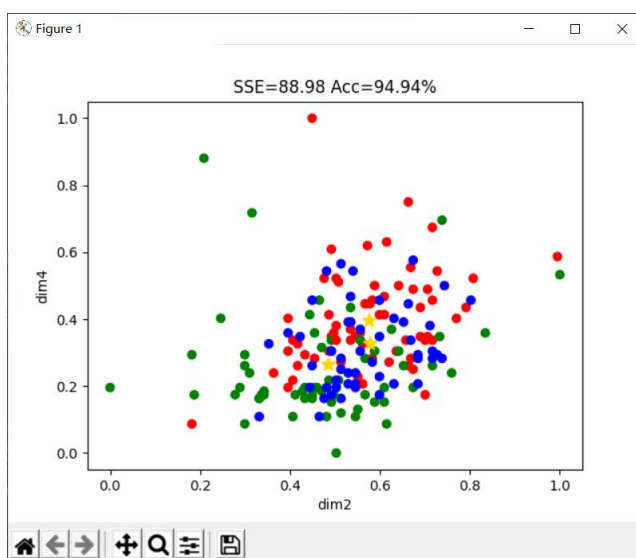


图 3 dim2 和 dim4 组合

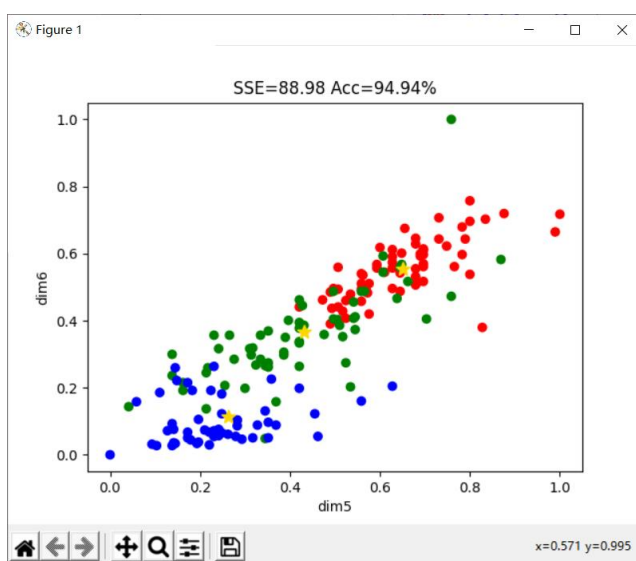


图 4 dim5 和 dim6

4.4 实验总结

本次实验采用的是 Kmeans 算法——K 均值聚类算法，该算法是一种迭代求解的聚类算法，其实现步骤是预将数据分为 3 组，则随机选取 3 个对象作为初始的聚类中心（质心），然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。尽管本次实验过程中会遇到一些问题，但在大量查询网络资源后，还是逐步解决问题。最后还实现了一个功能，用户可以自由选择不同维度组合进行对比分类效果。总的来说，这次实验让我对 Kmeans 算法的实现有了更深的了解，不仅结合了课堂上老师讲的算法原理，以及实验上对算法的实践。

实验 5：大作业目录

实验五 推荐系统算法及其实现.....	27
1.1 实验目的	27
1.2 实验内容	27
1.3 实验过程	28
1.3.1 编程思路.....	28
1.3.2 遇到的问题及解决方式.....	30
1.3.3 实验测试与结果分析.....	31
1.4 实验总结	32

实验五 推荐系统算法及其实现

1.1 实验目的

- 1、了解推荐系统的多种推荐算法并理解其原理。
- 2、实现 **User-User** 的协同过滤算法并对用户进行推荐。
- 3、实现**基于内容的推荐算法**并对用户进行推荐。
- 4、对两个算法进行电影预测评分对比
- 5、在学有余力的情况下，加入 **minihash** 算法对效用矩阵进行降维处理

1.2 实验内容

给定 MovieLens 数据集，包含电影评分，电影标签等文件，其中电影评分文件分为训练集 `train_set` 和测试集 `test_set` 两部分

基础版必做一：**基于用户的协同过滤推荐算法**

对训练集中的评分数据构造用户-电影效用矩阵，使用 **pearson** 相似度计算方法计算用户之间的相似度，也即相似度矩阵。对单个用户进行推荐时，找到与其最相似的 **k** 个用户，用这 **k** 个用户的评分情况对当前用户的所有未评分电影进行评分预测，选取评分最高的 **n** 个电影进行推荐。

在测试集中包含 100 条用户-电影评分记录，用于计算推荐算法中预测评分的准确性，对测试集中的每个用户-电影需要计算其预测评分，再和真实评分进行对比，误差计算使用 **SSE** 误差平方和。

选做部分提示：此算法的进阶版采用 **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 **jaccard** 方法计算相似度，需要对效用矩阵进行 01 处理，也即将 **0.5-2.5** 的评分置为 **0**，**3.0-5.0** 的评分置为 **1**。

基础版必做二：**基于内容的推荐算法**

将数据集 `movies.csv` 中的电影类别作为特征值，计算这些特征值的 **tf-idf** 值，得到关于电影与特征值的 **n**（电影个数）***m**（特征值个数）的 **tf-idf** 特征矩阵。根据得到的 **tf-idf** 特征矩阵，用余弦相似度的计算方法，得到电影之间的相似度矩阵。

对某个用户-电影进行预测评分时，获取当前用户的已经完成的所有电影的打分，通过电影相似度矩阵获得已打分电影与当前预测电影的相似度，按照下列方式进行打分计算：

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

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， $\text{score}'(i)$ 为计算集合中第 i 个电影的分数， $\text{sim}(i)$ 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 **jaccard** 方法计算相似度，特征矩阵应为 01 矩阵。因此进阶版的特征矩阵选取采用方式为，如果该电影存在某特征值，则特征值为 **1**，不存在则为 **0**，从而得到 **01** 特征矩阵。

选做（进阶）部分：

本次大作业的进阶部分是在基础版本完成的基础上大家可以尝试做的部分。进阶部分的主要内容是使用**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。

1.3 实验过程

1.3.1 编程思路

1.本次实验旨在两种推荐算法来完成实验。

第一种为基于用户的协同过滤推荐算法。

协同过滤是利用集体智慧的一个典型方法。要理解什么是协同过滤 (Collaborative Filtering, 简称 CF), 首先想一个简单的问题, 如果你现在想看部电影, 但你不知道具体看哪部, 你会怎么做? 大部分的人会问问周围的朋友, 看看最近有什么好看的电影推荐, 而我们一般更倾向于从口味比较类似的朋友那里得到推荐。这就是协同过滤的核心思想。本次实验我们采用的是基于用户的协同过滤算法。

第一步: 收集用户偏好 (建立用户模型) 用训练集中的评分数据构造用户-电影效用矩阵。

第二步: 找到相似的用户或物品。对每个用户进行推荐时, 分别调用函数计算出该用户和其他用户的相似度, 取相似度高的前 K 个用户。计算公式为:

$$sim(x, y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \bar{r}_y)^2}}$$

第三步: 基于用户的 CF (user CF)。基于用户对物品的偏好找到相邻邻居用户, 然后将邻居用户喜欢的推荐给当前用户。计算上, 就是将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度, 找到 K 邻居后, 根据邻居的相似度权重以及他们对物品的偏好, 预测当前用户没有偏好的未涉及物品, 计算得到一个排序的物品列表作为推荐。

prediction functions as in user-user model

$$r_{xi} = \frac{\sum_{j \in N(i; x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i; x)} s_{ij}}$$

s_{ij} ... similarity of items i and j
 r_{xj} ... rating of user u on item j
 $N(i; x)$... set items rated by x similar to i

通过预测评分公式, 取前 k 个与该用户相似度最高的用户, 对于这 k 个用户中与该用户相似度大于 0 的用户套用该公式, 小于 0 的舍弃。

选取前 n 个电影进行推荐时, 这些电影都是与其相似的 k 个用户看过的, 但是该用户还未看过。

误差计算: 对测试集中的 100 条用户-电影记录预测评分 \hat{y}_i , 与真实评分 y_i 一起带入下列公式计算。

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

第二种为基于内容的推荐算法

基于内容的协同过滤算法与基于用户的协同过滤算法很像，将商品和用户互换。通过计算不同用户对不同物品的评分获得物品间的关系。基于物品间的关系对用户进行相似物品的推荐。这里的评分代表用户对商品的态度和偏好。简单来说就是如果用户 A 同时购买了商品 1 和商品 2，那么说明商品 1 和商品 2 的相关度较高。当用户 B 也购买了商品 1 时，可以推断他也有购买商品 2 的需求。再了解了相关算法后，首先对数据集 movies.csv 中的电影类别进行分词，获取 m 个特征值。先生成电影与类别的矩阵。利用该矩阵生成 tf-idf 矩阵。

$$\text{公式: } tf_{ij} = \frac{n_{i,j}}{\sum_k n_{k,j}} \quad \text{即: } TF_w = \frac{\text{在某一类中词条 } w \text{ 出现的次数}}{\text{该类中所有的词条数目}}$$

此处 tf 取特征值在该电影类别中的出现频率，也就是该电影对应一行中相加的和的倒数，idf 为总电影数 n 与包含某特征值的电影数的商，再取以 10 为底的对数，也就是对应矩阵[,类别]这一列的和除以总行数，再取以 10 为底的对数就是 idf 值。

对每个电影，调用余弦相似度函数根据 tf-idf 矩阵计算出该电影于其他电影的相似度，生成电影相似度矩阵。

■ Cosine similarity measure

$$\text{■ } \text{sim}(x, y) = \cos(r_x, r_y) = \frac{r_x \cdot r_y}{\|r_x\| \cdot \|r_y\|}$$

对测试集中的每条用户-电影记录，根据下列公式预测评分：

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

其中，sim(i)为第 i 个电影与当前用户-电影的相似度，score'(i)为计算集合中第 i 个电影的分数。计算集合不一定为 K，因为对于相似度为负数的电影不纳入计算。当 n 为 0 时，score 为该用户所有已打分电影的平均值。

为用户推荐电影时，利用该公式通过用户已经打分的电影来依次计算电影相似度矩阵中每一个电影的预测打分(除去用户已看过的电影)，对其进行排序，选出其中前 k 个电影进行推荐。

1.3.2 遇到的问题及解决方式

1. 基于用户的协同过滤算法

(1) 计算用户之间相似度的时候，矩阵的计算量比较大，在用上述公式计算的时候，程序的开销有些大，需要很长的时间才可以运行出来，后来调用了

python 中的库函数，这个问题得到了解决。运行就比较快了。

(2) 对于用户 i ，假如与其相近的 k 个用户都未看过电影 m ，但是测试集里要求用户 i 对电影 m 进行评分，这就会造成无法评分。这与数据稀疏性有关系，不同用户之间重叠性较低，导致算法无法找到一个用户的邻居，即偏好相似的用户。因此不得不将 K 变的比较大。

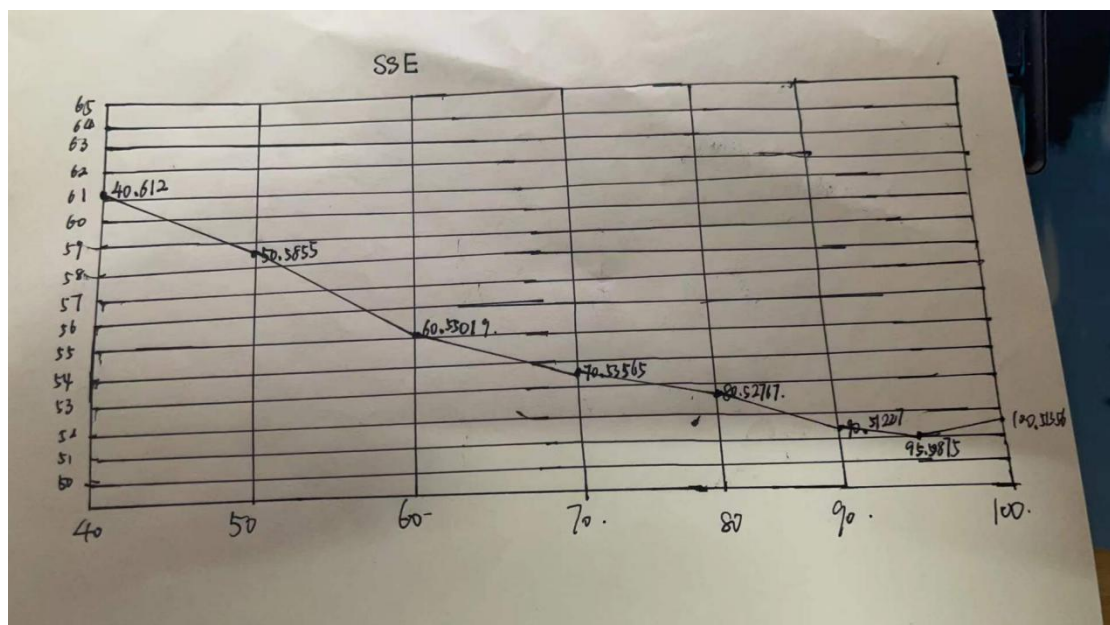
2. 基于内容的推荐算法

(1) 余弦相似度矩阵的计算时间过长，导致程序运行是占用了太多 CPU，降低了运行效率。因此我查了一些资料，调用了库函数 `cosine_similarity()` 计算余弦相似度矩阵。这与基于内容的推荐算法的缺点也有关。物品多的时候，计算物品相似度矩阵代价很大。

(2) 在得到一系列数据后，发现不知道怎么给电影排名，后来我利用用户已经观看的电影所打出的分数以及给出的预测分数公式，对所有电影进行预测打分，选出排名前 k 个预测打分高的电影，输出电影名字进行推荐。

1.3.3 实验测试与结果分析

1. 基于用户的协同过滤算法



改变 K 值得到不同的 SSE 值，如图所示, K 值为 95 时，SSE 最小为 50.87。

对于用户 2，根据最相似前 50 个用户进行推荐，为该用户推荐出的 5 个电影如下图所示，包括电影 id、推荐指数、电影名称、以及电影派系（种类）。

```
请输入用户id: 2
请输入与用户id'2'最相似的前k位用户: 50
请输入为这些用户推荐的电影数: 5
2      1      3.8781158455825793

电影id      推荐指数      电影名称      电影派系
255      5.299926217412691      Jerky Boys, The (1995)      ['Comedy']
534      5.238941533152815      Shadowlands (1993)      ['Drama', 'Romance']
162      5.203691828829079      Crumb (1994)      ['Documentary']
778      5.097183342039127      Trainspotting (1996)      ['Comedy', 'Crime', 'Drama']
219      5.090290381125227      Cure, The (1995)      ['Drama']
```

对用户 2 的推荐结果

2. 基于内容的推荐算法

测试集预测打分结果：

```
['Adventure', 'Animation', 'Children', 'Comedy', 'Fantasy', 'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror']
预测分值: {547: [(1, 3.5), (6, 2.5)], 564: [(1, 4.0), (2, 4.0)], 624: [(1, 5.0), (2, 3.0)], 15: [(1, 2.0), (2, 2.0)], 73: [(1, 3.0), (2, 3.0)]}
实际分值: {547: [(1, 3.252042375634131), (6, 3.385064560575199)], 564: [(1, 3.408251103150588), (2, 3.276935909638767)], 624: [(1, 3.500000000000000), (2, 3.500000000000000)], 15: [(1, 2.000000000000000), (2, 2.000000000000000)], 73: [(1, 3.000000000000000), (2, 3.000000000000000)]}
SSE误差平方和: 67.06801578815222
```

结果分析：由图可知预测打分结果中有的分数预测和实际分数接近，有的预测分数与实际分数相差一定值，最终 SSE 误差平方和计算为 67.068。

测试为用户 ID 为 3 的推荐前 10 部电影：

```
请输入用户id: 3
请输入为该用户推荐的电影数: 10

电影名称      推荐指数
Prisoner of the Mountains (Kavkazsky plennik) (1996)      4.151463
Pork Chop Hill (1959)      4.151463
Run Silent Run Deep (1958)      4.151463
Cross of Iron (1977)      4.151463
Duel at Diablo (1966)      4.151463
Murphy's War (1971)      4.151463
Richard III (1995)      4.115199
Misérables, Les (1995)      4.115199
Before the Rain (Pred dozhdot) (1994)      4.115199
Walking Dead, The (1995)      4.115199
```

1.4 实验总结

李延波：

本次实验是大数据分析的最后一个实验，也是相对来说比较重要的一部分，推荐系统。本次实验是基于用户的协同过滤推荐系统和基于内容的协同过滤的推荐系统。两种推荐系统各有各的优势。通过本次实验让我对推荐系统有了一些了解。尤其是基于内容的协同过滤推荐系统。利用余弦相似度矩阵运行，耗费了很长的时间，后来调用库函数，则比较快速的完成了实验。而且基于内容的协同

过滤推荐系统也很好的弥补了基于用户的一些不足之处。因为物品直接的相似性相对比较固定，所以可以预先在线下计算好不同物品之间的相似度，把结果存在表中，当推荐时进行查表，计算用户可能的打分值。这样就解决了数据稀疏性和算法扩展性的问题。另外，对于测试集计算 SSE 是发现了该算法的一些问题，然我对该算法有了更深入的理解。本次实验让我收获颇丰。

刘日星：

这次实验对比于前几次的实验工程量要大一些，因此在和朋友一起完成代码时，首先要一起确定一些思路，然后通过确定好的思路进行编写。我主要是负责基于用户协同过滤算法的代码编写。同时在写的时候也出现了几点问题。在用户对电影做评价环节。由于数据的稀疏性，导致很多电影没有评分，这最开始让我很头疼。后来通过商量决定，将 K 值变大。另外在 SSE 上我也发现了一些算法上的问题。这次试验让我对推荐系统有了进一步的了解。也让我对于基于用户和基于内容的两个不同的算法进行了比较之后，有了更多收获。这次实验让我明白，推荐系统取自于生活，也用之于生活。