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： 物联网工程 1801 班

学　　号： U201814500

姓　　名： 王英嘉

指导教师： 崔金华

报告日期： 2020.12.23

计算机科学与技术学院

目录

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 jarcard 方法计算相似度，需要对效用矩阵进行 01 处理，也即**将 0.5-2.5 的评分置为 0，3.0-5.0 的评分置为 1**。

基础版必做二：**基于内容的推荐算法**

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

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

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

选取相似度大于零的值进行计算，如果已打分电影与当前预测用户-电影相似度大于零，加入计算集合，否则丢弃。（相似度为负数的，强制设置为 0，表示无相关）假设计算集合中一共有 n 个电影，score 为我们预测的计算结果，score'(i)为计算集合中第 i 个电影的分数，sim(i)为第 i 个电影与当前用户-电影的相似度。如果 n 为零，则 score 为该用户所有已打分电影的平均值。

要求能够对指定的 userID 用户进行电影推荐，推荐电影为预测评分排名前 k 的电影。userID 与 k 值可以根据需求做更改。

推荐算法准确值的判断：**对给出的测试集中对应的用户-电影进行预测评分，输出每一条预测评分，并与真实评分进行对比，误差计算使用 SSE 误差平方和**。

选做部分提示：进阶版采用 minihash 算法对特征矩阵进行降维处理，从而得到相似度矩阵，注意 minihash 采用 jarcard 方法计算相似度，特征矩阵应为 01 矩阵。因此进阶版的特征矩阵选取采用方式为，**如果该电影存在某特征值，则特征值为 1，不存在则为 0，从而得到 01 特征矩阵**。

选做（进阶）部分：

本次大作业的进阶部分是在基础版本完成的基础上大家可以尝试做的部分。进阶部分的主要内容是使用**迷你哈希（MiniHash）算法对协同过滤算法和基于内容推荐算法的相似度计算进行降维**。同学可以把迷你哈希的模块作为一种近似度的计算方式。

协同过滤算法和基于内容推荐算法都会涉及到相似度的计算，迷你哈希算法在牺牲一定准确度的情况下对相似度进行计算，其能够有效的降低维数，尤其是对大规模稀疏 01 矩阵。同学们可以**使用哈希函数或者随机数映射来计算哈希签名**。哈希签名可以计算物品之间的相似度。

最终降维后的维数等于我们定义映射函数的数量，我们设置的映射函数越少，整体计算量就越少，但是准确率就越低。**大家可以分析不同映射函数数量下，最终结果的准确率有什么差别**。

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

1.3 实验过程

1.3.1 编程思路

本次实验主要完成了基于用户的协同过滤推荐和基于内容的推荐——两种推荐算法，之后分别通过 minhash 策略进行了优化。

User-User 协同过滤推荐算法主要流程说明如下：

读取训练集数据，并转化为透视表作为效用矩阵，大小为电影数量*用户数量，每一列代表某一用户对不同电影的评分（0 表示没看过），如图 1 所示。

```
# 生成utility matrix
matrix = data.pivot_table(index=['movieId'],columns=['userId'],values='rating').reset_index(drop=True)
matrix.fillna(0, inplace=True)
matrix.index = np.array(sorted(data['movieId'].unique()))

matrix
```

Out[3]:

userId	1	2	3	4	5	6	7	8	9	10	...	662	663	664	665	666	667	668	669	670	671
1	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	4.0	0.0	...	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	5.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
161944	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
162376	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
162542	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
162672	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
163949	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

9066 rows × 671 columns

图 1 生成 utility matrix

之后使用 pearson 相似度计算相似度矩阵，这里使用了 DataFrame 的 corr() 方法，计算后进行排序，取出最相似的 K 个用户进行后续预测，如图 2 所示。

```
# 不使用minhash优化
else:
    # 使用pearson系数计算用户之间的相似度
    users_num = len(matrix.iloc[0])
    sim_dict = {i : matrix[userid].corr(matrix[i], method='pearson') for i in range(1, users_num+1)}
    sorted_sim = sorted(sim_dict.items(), key=lambda d:d[1], reverse=True)

# 取K个最相似的用户
topK_id = [sorted_sim[i][0] for i in range(K)]
topK_matrix = matrix[topK_id]
```

图 2 生成相似度矩阵并排序

之后如果是需要直接对评分进行预测，mode 为 1，对于要预测的某一部电影，通过取 K 个用户中看过这一部电影的所有评分的均值来作为预测值，否则返回 2.5，即中评。

如果要预测评分最高的 n 个电影，mode 为 0，相当于预测每一部电影的评分，

然后从高到低排序，按格式输出即可。

代码如图 3 所示。

```
# 直接预测评分模式
if mode == 1:
    x = topK_matrix.loc[movieid]
    pred_i = np.mean(x[x!=0])
    return 2.5 if np.isnan(pred_i) else pred_i

# topN推荐模式
else:
    pred_dict = {}
    for i in range(len(matrix)):
        x = topK_matrix.iloc[i]
        pred_i = np.mean(x[x!=0]) # 去掉里面的0项
        pred_dict[i] = 0 if np.isnan(pred_i) else pred_i

    # 取前n个电影推荐
    sorted_pred = sorted(pred_dict.items(), key=lambda d:d[1], reverse=True)
    pred = sorted_pred[:n]
    print('As for User %d, the top %d recommendations are shown below:' % (userid, n))
    print('-------------------------------------------------------------')
    for i in range(n):
        id, score = pred[i]
        print('%4d | %.4f' % (matrix.index[id], score))
```

图 3 根据不同 mode 进行推荐

minhash 算法可以对大规模训练数据进行降维处理，通过牺牲一定的准确率来提高效率。

首先需要将所有训练集中的评分 01 化，即将 0.5-2.5 的评分置为 0，将 3.0-5.0 的评分置为 1，代码如图 4 所示，生成 binary utility matrix 部分代码如图 5 所示。

```
# 如果是minhash模式，需要额外生成一个对rating进行01化的矩阵
binary_data = data.copy(deep=True)
if minhash:
    binary_data.loc[binary_data['rating']<2.6, 'rating'] = 0
    binary_data.loc[binary_data['rating']>2.9, 'rating'] = 1
```

图 4 将训练集评分 01 化

In [4]:
```
# 生成binary utility matrix
if minhash:
    binary_matrix = binary_data.pivot_table(index=['movieId'],columns=['userId'],values='rating').reset_index(drop=Tr
    binary_matrix.fillna(0, inplace=True)
    binary_matrix.index = np.array(sorted(binary_data['movieId'].unique()))

binary_matrix
```

Out[4]:

userId	1	2	3	4	5	6	7	8	9	10	...	662	663	664	665	666	667	668	669	670	671
1	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	...	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
161944	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
162376	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
162542	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
162672	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
163949	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

9066 rows × 671 columns

图 5 生成 binary utility matrix

如图 6 所示，生成 01 效用矩阵后，就可以通过 nfuncs 个 permutation π 进行降维，这里采用随机数映射的方式来生成。以图 6 中最左边红色的映射函数为例，通过找 input matrix 中每一列的第一个 1 是在乱序后的第几行出现的，结果作为右侧 signature matrix 的第一行，其他颜色映射函数同理，最终签名矩阵的大小为 nfuncs * users_num，从而实现了降维。

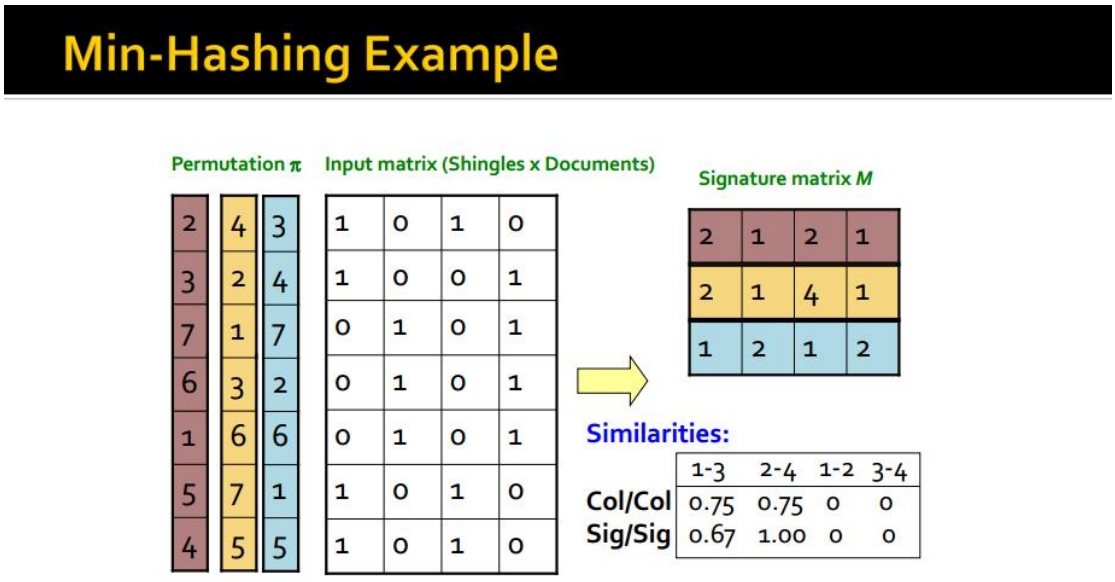

图 6 minhash 示例

这里 signature matrix 的计算效率比较低，目前的优化方案是：利用 DataFrame 的 reindex 方法进行按行打乱，比手动调整顺序要快，打乱后生成的新 DataFrame

之间从上到下遍历，集合 s 用来判断对应某一种 permutation 来说 matrix 中的每一列是否找到了第一个“1”，一开始里面有 users_num 个元素，如果都找到了集合 s 变为空，就提前结束遍历。

度量方式仍然使用 jaccard 近似度，如图 7 所示。

```
# minhash优化
if minhash:
    # 根据随机生成的nfuncs个映射函数生成哈希签名矩阵
    users_num = len(binary_matrix.columns)
    movies_num = len(binary_matrix[1])

    sig_matrix = np.zeros((nfuncs, users_num))
    for i in range(nfuncs):
        shuffled_matrix = binary_matrix.reindex(np.random.shuffle(list(range(1, movies_num+1))))
        s = set(range(users_num))# 记录对于每个func，user是否找到第一个1的集合，当user找到了则从集合中弹出

        sig_i = np.zeros(users_num)
        for j in range(movies_num):
            row = np.array(shuffled_matrix.iloc[j])
            for r in range(users_num):
                if row[r] and r in s:
                    s.remove(r)
                    sig_i[r] = j
            if not s:
                break

        sig_matrix[i] = sig_i # 更新签名矩阵的第i行


    sig_matrix = pd.DataFrame(sig_matrix)
    sig_matrix.columns = list(range(1, users_num+1))

    # 使用jaccard系数计算用户之间的相似度
    sim_dict = {i : np.sum(sig_matrix[userid] == sig_matrix[i]) / nfuncs for i in range(1, users_num+1)}
    sorted_sim = sorted(sim_dict.items(), key=lambda d:d[1], reverse=True)
```

图 7 生成 signature matrix

基于内容的推荐算法主要流程说明如下：

首先读取 movies.csv 中的数据，以电影类别作为特征值，计算 tf-idf 特征矩阵，如图 8 所示。

```
# 不使用minhash生成tf-idf矩阵
else:
    tfidf = TfidfVectorizer()
    matrix = tfidf.fit_transform(genres).toarray()
    matrix = pd.DataFrame(matrix)

matrix.shape
```

Out[5]: (9125, 24)

In [6]:
```
matrix
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	...	14	15	16	17	18	19	20	21	22	23
0	0.000000	0.410433	0.531527	0.496423	0.266469	0.0	0.0	0.000000	0.481230	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0
1	0.000000	0.510466	0.000000	0.617414	0.000000	0.0	0.0	0.000000	0.598519	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0
2	0.000000	0.000000	0.000000	0.000000	0.586999	0.0	0.0	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.809588	0.000000	0.0	0.0	0.0
3	0.000000	0.000000	0.000000	0.000000	0.523605	0.0	0.0	0.452029	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.722155	0.000000	0.0	0.0	0.0
4	0.000000	0.000000	0.000000	0.000000	1.000000	0.0	0.0	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9120	0.000000	0.687462	0.000000	0.000000	0.000000	0.0	0.0	0.385314	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.615573	0.000000	0.0	0.0	0.0
9121	0.376898	0.420914	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.493519	0.467556	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.467556	0.0	0.0	0.0
9122	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	1.0	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0
9123	0.000000	0.000000	0.000000	0.000000	1.000000	0.0	0.0	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0
9124	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	1.0	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0

9125 rows × 24 columns

图 8 生成 tf-idf 特征矩阵

再根据 tf-idf 特征矩阵用余弦相似度计算相似度矩阵，如图 9 所示。

```
else:
    cosine_matrix = cosine_similarity(matrix)
    print(cosine_matrix.shape)
```

```
(9125, 9125)
```

图 9 生成余弦相似度矩阵

接下来读取训练集，对于每个用户，获得训练集中该用户看过的电影和评分数据，若要预测该用户对某一部电影的打分，先从电影相似度矩阵中提取出该电影与其他电影的相似度向量，然后对该用户看过的所有电影按图 10 中的公式进行加权，求得最终的评分，其中 sim(i)为第 i 部电影与当前电影的相似度，score’(i)为该用户对该电影的评分。

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

图 10 评分预测公式

值得注意的是，如果没有相似度大于 0 的电影被该用户打过分，即计算集合为空，则返回用户所有打过的评分的平均值。

整个预测评分部分代码如图 11 所示。

```python
def cal_score(rated_movies, rating, method, movieid):
    '''
    计算当前用户对index为movieid的电影的打分
    @params:
        rated_movies: 当前用户评价过的movie_id(numpy.array)
        rating: 当前用户评价过的电影评分(numpy.array)
        method: 距离度量方式(cosine或jaccard)
        movieid: 电影的index，与id不同，注意区分
    '''
    # cosine
    if method == 'cosine':

        distances = cosine_matrix[movieid] # 从movieid出发的距离向量
        computed_dict = {} # 计算集合
        for i in range(len(rated_movies)):
            rated_movie = rated_movies[i]
            cosine = distances[Id2index[rated_movie]]
            if cosine > 1e-6:
                computed_dict[i] = cosine

        # 计算集合不为空
        if len(computed_dict.keys()):
            score = 0
            sum_v0 = 0
            for k, v in computed_dict.items():
                score += rating[k] * v
                sum_v0 += v

            return score / sum_v0

        # 计算集合为空
        else:
            return np.mean(rating)
```

图 11 cosine 近似度预测电影评分

minhash 算法在基于内容的推荐中的改进体现在：不计算特征的 tf-idf 值，而是特征存在则为 1，特征不存在则为 0，仍然生成 01 矩阵，这里通过 CountVectorizer 并将 binary 参数修改为 True 来实现，如图 12 所示。

```python
# 如果使用minhash生成01矩阵
if minhash:
    cnt = CountVectorizer(binary=True)
    matrix = cnt.fit_transform(genres).toarray().T # 此处转置获得(features_num * movies_num)大小的矩阵，便于后续优化
    matrix = pd.DataFrame(matrix)
    matrix.columns = list(index2Id.values())
```

图 12 生成 01 相似度矩阵

度量方式仍然使用 jaccard 度量，如图 13 所示。

```
    # jaccard
    elif method == 'jaccard':

        computed_dict = {} # 计算集合
        for i in range(len(rated_movies)):
            rated_movie = rated_movies[i]
            sim = np.sum(sig_matrix[movieid] == sig_matrix[Id2index[rated_movie]]) / nfuncs
            if sim > 1e-6:
                computed_dict[i] = sim

        # 计算集合不为空
        if len(computed_dict.keys()):
            score = 0
            sum_v0 = 0
            for k, v in computed_dict.items():
                score += rating[k] * v
                sum_v0 += v

            return score / sum_v0

        # 计算集合为空
        else:
            return np.mean(rating)

    # error
    else:
        raise Exception("Only Cosine and Jaccard are accepted.")
```

图 13 jaccard 近似度预测电影评分

1.3.2 遇到的问题及解决方式

整个实验没有出现太多问题，指导书中对算法流程讲的也很清晰，相对来说主要是数据的处理比较繁琐。

值得一提的是，minhash 中生成签名矩阵这一部分虽然已经尽量优化效率，但性能还是较差，进一步可以考虑利用多线程同时处理多个 permutation 输出到不同行来加速，或者将多个 permutation 组成的矩阵对 input matrix 进行矩阵运算来提高并行性。

1.3.3 实验测试与结果分析

1.3.3.1 top 电影推荐

先测试基于用户的协同过滤推荐，参数配置如图 14 所示，生成对 userid=671 的用户的前 20 部推荐的电影，生成过程中取最相似的 50 名用户来预测。

结果如图 15 所示。

```
:  # 基于用户的协同过滤推荐
   userid = 671
   K = 50
   n = 20
```

图 14 参数配置 1

```
# 进行topK推荐
recommender(0, minhash, userid, K, n)
```

```
As for User 671, the top 20 recommendations are shown below:
-------------------------------------------------------------
  30 | 5.0000
  89 | 5.0000
 140 | 5.0000
 178 | 5.0000
 247 | 5.0000
 257 | 5.0000
 279 | 5.0000
 280 | 5.0000
 281 | 5.0000
 290 | 5.0000
 299 | 5.0000
 306 | 5.0000
 307 | 5.0000
 308 | 5.0000
 340 | 5.0000
 412 | 5.0000
 452 | 5.0000
 501 | 5.0000
 633 | 5.0000
 667 | 5.0000
```

图 15 推荐结果 1

再测试基于内容的推荐，参数配置如图 16 所示，同样生成前 20 部推荐的电影，结果如图 17 所示。

```
# 基于内容推荐
userid = 671
K = 20
```

图 16 参数配置 2

```
recommender(0, minhash, userid, K)
```

```
As for User 671, the top 20 recommendations are shown below:
-------------------------------------------------------------
 25771 |                          Andalusian Dog, An (Chien andalou, Un) (1929) | 4.3702
 95113 |                                      Eaux d'artifice (1953) | 4.3702
 95115 |                 Inauguration of the Pleasure Dome (1954) | 4.3702
  1583 |                                   Simple Wish, A (1997) | 4.3604
  7045 |                                     Witches, The (1990) | 4.3604
 47721 |                    Red Balloon, The (Ballon rouge, Le) (1956) | 4.3604
   343 |                              Baby-Sitters Club, The (1995) | 4.3483
  3820 |                   Thomas and the Magic Railroad (2000) | 4.3483
   606 |                  Candyman: Farewell to the Flesh (1995) | 4.3305
  5128 |                              Queen of the Damned (2002) | 4.3305
  7951 |                                      Nightbreed (1990) | 4.3305
 25737 |  Golem, The (Golem, wie er in die Welt kam, Der) (1920) | 4.3305
 27778 |                 Ginger Snaps Back: The Beginning (2004) | 4.3305
 70946 |                                           Troll 2 (1990) | 4.3305
 85412 |                    Troll Hunter, The (Trolljegeren) (2010) | 4.3305
  2017 |                                 Babes in Toyland (1961) | 4.3302
  4294 |                      5,000 Fingers of Dr. T, The (1953) | 4.3302
 74089 |                                       Peter Pan (1960) | 4.3302
 74630 |                                       Tom Thumb (1958) | 4.3302
  2876 |                                     Thumbelina (1994) | 4.3217
```

图 17 推荐结果 2

对比两种推荐算法的生成结果，即图 15 和图 17，可以看出基于内容的推荐生成的结果似乎更可靠一些，而基于用户的协同过滤推荐前 20 部电影都是五星

推荐。

仔细思考 User-User 协同过滤的算法原理，可以想到它的决策策略是以最相似的 k 个用户的平均预测来当作该用户的预测，那么如果这 k 个用户只有极少数人看了某一部电影，还都给了五星好评，那这样的推荐显然是不合理的，因为没有考虑到样本数的影响。

而基于内容的推荐相对来说效果好一些，一方面因为它通过加权均衡了一些极高评分的影响（因为大多数电影都和该电影或多或少有相似），另一方面训练集的数据给的也比较好，没有新用户出现。

为了验证猜想，如图 18 所示，可以简单修改一下代码，去掉样本数小于等于 10 的电影，结果如图 19 所示。

```
    if len(x[x!=0]) > 10:
        pred_i = np.mean(x[x!=0]) # 去掉里面的0项
        pred_dict[i] = 0 if np.isnan(pred_i) else pred_i
    else:
        pred_dict[i] = 0
```

图 18 不考虑样本数小于等于 10 的电影

```
# 进行topK推荐
recommender(0, minhash, userid, K, n)
```

```
As for User 671, the top 20 recommendations are shown below:
-------------------------------------------------------------
 858 | 4.5377
 318 | 4.5370
1204 | 4.5000
1221 | 4.5000
1252 | 4.5000
4235 | 4.5000
 953 | 4.4412
58559 | 4.4000
1196 | 4.3803
1197 | 4.3729
 912 | 4.3654
 527 | 4.3636
1228 | 4.3636
1136 | 4.3571
7153 | 4.3472
5618 | 4.3409
2571 | 4.3395
  50 | 4.3393
4993 | 4.3333
 296 | 4.3228
```

图 19 推荐结果 3

也可以修改样本数下限为 50，进一步观察，结果如图 20 所示。

```
# 进行topK推荐
recommender(0, minhash, userid, K, n)
```

```
As for User 671, the top 20 recommendations are shown below:
-------------------------------------------------------------
 858 | 4.5377
 318 | 4.5370
1196 | 4.3803
1197 | 4.3729
 527 | 4.3636
7153 | 4.3472
2571 | 4.3395
  50 | 4.3393
4993 | 4.3333
 296 | 4.3228
4226 | 4.3000
2918 | 4.2642
5952 | 4.2470
1198 | 4.2297
2858 | 4.2254
 260 | 4.2222
1210 | 4.2119
1704 | 4.2037
2959 | 4.1736
1291 | 4.1053
```

图 20 推荐结果 4

分析得知，猜想是正确的，并且样本数下限越高，得到的预测评分数据将越可靠。

1.3.3.2 测试集 test_set.csv

仍然先测试基于用户的协同过滤推荐，结果如图 21 所示，SSE=73.902。

```
Out[7]: 73.90166180588304
```

图 21 测试结果 1

观察预测值和实际值的差，结果如图 22 所示，可以看到有一些有一些数据偏差很大，100 条数据中有 8 条数据预测值和真实值差的平方大于 2。

```
In [10]: np.array(preds - ratings)
Out[10]: array([ 0.5       ,  1.25      ,  0.15      , -0.47368421, -0.925     ,
                 0.29411765,  1.9375    ,  1.36666667, -1.05681818,  0.80952381,
                 0.51428571,  1.75      ,  0.11538462,  1.41176471, -0.09      ,
                -0.175     ,  1.22368421, -0.10416667,  0.14473684,  1.23076923,
                 0.17346939, -0.28571429,  1.        ,  0.75      , -0.04761905,
                -0.23809524,  0.87931034,  0.43103448,  0.96875   ,  0.26086957,
                -0.92857143,  0.72413793, -1.19354839,  1.26190476,  0.9625    ,
                 0.35483871,  2.51282051,  1.61111111, -0.71794872, -0.22222222,
                 0.625     ,  1.        ,  0.96153846, -0.390625  , -0.45454545,
                -0.44285714,  1.45348837,  1.26470588, -0.6372549 ,  0.10526316,
                 1.08333333, -0.45454545,  1.96428571,  0.53846154, -0.91860465,
                 0.36363636, -0.91860465,  2.125     ,  0.02941176, -0.51470588,
                 1.14102564, -0.57142857,  0.9375    , -0.22093023, -0.03333333,
                -0.75      ,  0.35      ,  1.26190476,  0.31111111, -0.025     ,
                -0.16666667,  0.04347826, -0.28787879,  0.29761905,  1.70408163,
                 0.57692308,  0.39583333, -0.07142857, -0.36      ,  0.45588235,
                -0.13157895, -0.60606061,  1.02380952, -1.09375   ,  0.96428571,
                 0.67105263, -0.14102564,  0.02083333,  0.46      ,  0.15384615,
                -0.35106383,  0.56818182,  1.16216216,  0.53846154, -0.47222222,
                 0.79310345,  0.08333333,  0.43181818,  0.93      ,  0.44736842])

In [12]: np.sum(np.square(preds-ratings) > 2)
Out[12]: 8
```

图 22 测试结果 2

由之前的猜想，如图 23 所示修改代码，将样本数小于等于 15 的直接给中评，结果如图 24 所示，SSE=69.506，进一步验证了样本的数量对推荐正确性和可靠性的影响。

```
# 直接预测评分模式
if mode == 1:
    x = topK_matrix.loc[movieid]
    if len(x[x!=0]) > 15:
        pred_i = np.mean(x[x!=0])
        return pred_i
    else:
        return 2.5
```

图 23 不考虑样本数小于等于 15 的电影

```
Out[7]: 69.50618852762733

In [8]: np.array(preds - ratings)

Out[8]: array([ 0.5       ,  1.25      ,  0.15      , -0.47368421, -0.925     ,
                0.29411765,  1.9375    ,  0.5       , -1.05681818,  0.80952381,
                0.51428571,  1.5       ,  0.11538462,  1.41176471, -0.09      ,
               -0.175     ,  1.22368421, -0.10416667,  0.14473684,  0.5       ,
                0.17346939, -0.28571429,  1.        ,  0.5       , -0.04761905,
               -0.23809524,  0.87931034,  0.43103448,  0.96875   ,  0.26086957,
               -0.92857143,  0.72413793, -1.19354839,  1.26190476,  0.9625    ,
                0.35483871,  2.51282051,  1.        , -0.71794872, -0.22222222,
                0.625     ,  1.        ,  0.96153846, -0.390625  , -1.5       ,
               -0.44285714,  1.45348837,  1.26470588, -0.6372549 ,  0.10526316,
                1.08333333, -0.45454545,  1.96428571, -0.5       , -0.91860465,
                0.36363636, -0.91860465,  1.5       ,  0.02941176, -0.51470588,
                1.14102564, -1.5       ,  0.9375    , -0.22093023, -0.03333333,
               -0.75      ,  0.35      ,  1.26190476,  0.31111111, -0.025     ,
               -0.16666667,  0.04347826, -0.28787879,  0.29761905,  1.70408163,
                0.57692308,  0.39583333, -0.07142857, -0.36      ,  0.45588235,
               -0.13157895, -0.60606061,  1.02380952, -1.09375   ,  0.5       ,
                0.67105263, -0.14102564,  0.02083333,  0.46      , -0.5       ,
               -0.35106383,  0.56818182,  1.16216216,  0.53846154, -0.47222222,
                0.79310345,  0.08333333,  0.43181818,  0.93      ,  0.44736842])

In [9]: np.sum(np.square(preds-ratings) > 2)

Out[9]: 9
```

图 24 测试结果 3

之后使用 minhash 优化的版本，分别测试 nfuncs=500、nfuncs=1000、nfunc=2000、nfuncs=4000 和 nfuncs=8000 时的 SSE，结果如图 25 所示。

```
Out[7]: 98.6921240327096

Out[7]: 94.08364489684323

Out[7]: 92.52965431930402

Out[7]: 84.17246591023654

Out[7]: 77.29453094129771
```

图 25 测试结果 4

再测试基于内容的推荐，结果如图 26 所示，SSE=67.170。

```
Out[11]: 67.16953413803502

In [12]: np.array(preds - ratings)

Out[12]: array([-0.27314866,  0.87999514, -0.57543419, -0.71773035, -2.02738345,
                -0.02310746,  0.42043843,  0.31309611, -1.72995898,  0.76031178,
                -0.5228546 ,  1.97705611, -1.15738643,  0.79155263, -0.65927611,
                -0.61839317, -0.10558032, -0.95647171, -0.43703254,  1.53990839,
                -0.40675427,  0.11085225,  0.1954691 ,  1.18441499, -0.75739634,
                -0.29491523, -0.4351187 , -0.36959005, -0.00211768,  0.06908615,
                -1.26999597,  0.78074644, -1.44820919,  1.56723336,  0.54186088,
                 0.09281789,  0.68345173,  0.93573039, -1.39289445, -0.33785183,
                -0.56353213,  1.4103754 ,  0.94427234, -0.06159044, -0.0123309 ,
                -0.01196377,  0.00493659,  0.55349768, -2.01803525, -0.82185938,
                 0.396917  , -0.59835902,  1.42571044,  0.32473435, -0.90647382,
                 1.12341898, -1.95642419,  2.0542806 , -0.49441867, -0.93319555,
                 0.44607463, -0.58474935,  0.40103957, -0.20942683, -0.34660722,
                -0.56250634, -1.02397911,  0.49285009,  0.28481653, -0.1890036 ,
                -0.47803792,  0.02144235,  0.04930605,  0.55310766,  0.31736872,
                -0.17836182, -0.0733601 , -0.34573752, -1.300695  , -0.01211967,
                -0.25422438,  0.14792478,  0.3164994 , -0.64217767,  0.983975  ,
                -0.02132623,  0.00422374,  0.32436638,  0.45178873,  0.19142987,
                -0.79548593,  0.79678661,  0.60690252,  0.65517057, -0.8164531 ,
                 1.12168163, -0.90955907,  0.10313219, -0.03295746, -0.0424357 ])
```

图 26 测试结果 5

之后使用 minhash 优化的版本，分别测试 nfuncs=2、nfuncs=5、nfuncs=10、nfunc=15、nfuncs=20 时的 SSE，结果如图 27 所示。

```
Out[11]: 68.94247161560186

Out[11]: 67.51770514554069

Out[11]: 66.44569437454912

Out[11]: 65.29721203600708

Out[11]: 65.79431017813054
```

图 27 测试结果 6

整理两种推荐在不同条件下的 SSE，分别如表 1 和表 2 所示，其中 User-User 协同过滤中输入矩阵的维度为(9066, 671)，Content-based 推荐中输入矩阵的维度为(24, 9125)。

表 1 User-User 不同条件下的 SSE

	common	minhash nfuncs=500	minhash nfuncs=1000	minhash nfuncs=2000	minhash nfuncs=4000	minhash nfuncs=8000
User-User	73.902	98.692	94.084	92.530	84.172	77.295

表 2 Content-based 不同条件下的 SSE

	common	minhash nfuncs=2	minhash nfuncs=5	minhash nfuncs=10	minhash nfuncs=15	minhash nfuncs=20
Content-based	67.170	68.942	67.518	66.446	65.297	65.794

由表 1 和表 2 分析可知：

1、在该测试集上，无论是否加入 minhash 算法，Content-based 推荐效果都优于 User-User 协同过滤推荐。

2、对于 User-User 协同过滤推荐来说，数据维度很大，加入 minhash 后效果较差，说明了 minhash 降维通过牺牲准确率提高效率的特点；对于 Content-based 推荐来说，加入 minhash 后 nfunc<=5 时效果稍差，nfuncs>=10 时效果更好，猜测一方面因为结果相差不大且有随机数影响，另一方面也在于对于 minhash 可能将一些弱特征进一步转化为了强特征，从而提高了准确率。

3、在两种推荐中，nfuncs 即哈希函数的数量反映了是否充分的对原输入矩阵的特征进行刻画，只不过这里区别于传统特征工程而采用了随机数映射的方式。相比之下，nfuncs 较大时特征充分，但降维效果差，nfuncs 较小时特征少，但推荐系统效果不好，所以在 nfuncs 值的设置上需要权衡。

1.4 实验总结

通过本次实验，掌握了基于内容推荐和基于用户的协同过滤两种推荐算法的原理，以及 minhash 在推荐中的降维应用。

基于内容的推荐系统虽然可以较好地根据用户历史数据做类似的推荐，也能避免冷启动的问题，但正因为它需要历史数据，所以无法为新用户进行推荐，也不能尝试拓展用户的兴趣爱好，并且特征的设计好坏对推荐效果影响很大。

相比之下协同过滤不需要筛选特征，在本实验中即不同电影的类型，它尝试去根据相似用户推荐相对来说公认评价比较高的电影，但是存在冷启动的问题，也不能推荐一个没被评价过的电影。

因此，在实际应用中，一般都是使用混合类型的推荐系统进行推荐，例如可以将不同的推荐按照具体问题分配不同的权重对结果进行组合作为最终结果，也可以采取多层推荐机制——将一个推荐机制的结果作为另一个的输入，还可以采用分区推荐机制，将不同的推荐结果分不同区显示给用户。