南開大學
Nankai University

计算机学院

大数据计算及应用

推荐系统

姓名：赵康明　李颖

学号：2110937　2110939

专业：计算机科学与技术

2024 年 6 月 13 日

目录

1 实验内容 **3**

2 问题重述 **3**

3 实验环境 **3**

4 数据集说明 **3**

5 实验参数设置 **4**

6 推荐系统 **5**
6.1 推荐系统常用算法 5
6.1.1 基于内容的推荐算法 5
6.1.2 协同过滤算法 5
6.1.3 SVD 5
6.2 评判指标 6

7 实验原理 **6**
7.1 基于用户的协同过滤算法 6
7.2 基于物品的协同过滤算法 7
7.3 基于 SVD 的隐语义模型 8
7.4 评测指标：RMSE 9

8 核心代码 **10**
8.1 划分数据集 10
8.2 基于用户的协同过滤算法 11
8.2.1 划分三元组及用户平均评分 11
8.2.2 计算用户相似度矩阵 12
8.2.3 预测评分 14
8.2.4 计算 RMSE 16
8.3 基于 SVD 的隐语义模型 17
8.3.1 训练过程 17
8.3.2 引入属性进行计算 18

9 实验结果 **18**
9.1 RMSE 18
9.2 训练时间 19
9.2.1 时间复杂度分析 19
9.2.2 训练时长 19
9.3 空间消耗 19
9.3.1 空间复杂度 19
9.3.2 实际空间占用 20

10 实验总结 **20**

1 实验内容

预测 Test.txt 文件中 (u, i) 的评分。
数据集：

- Train.txt，用于训练模型。
- Test.txt，用于测试。
- ItemAttribute.txt，用于训练模型（可选）。
- ResultForm.txt，结果文件的形式。

数据集的格式在 DataFormatExplanation.txt 中有解释。

请注意，如果能恰当地使用 ItemAttribute.txt 并改进算法的性能，最终成绩可以额外增加分数（最多 10 分）。

在这个项目中，需要报告 Test.txt 文件中未知对 (u, i) 的预测评分。可以使用从课程中学到的任何算法或从其他资源（如 MOOC）学到的算法。

2 问题重述

实现基于协同过滤或 SVD 的推荐系统算法，预测 Text.txt 中用户对物品的评分，并计算出 RMSE，评测推荐系统性能。

3 实验环境

- 系统：Windows 11
- 处理器：AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz
- 内存容量：16GB
- 开发环境：PyCharm Community Edition 2023.2
- Python 版本：3.11.4

4 数据集说明

本次的数据集有以下几类：

1. Train.txt，用于训练模型；
2. Test.txt，用于测试；
3. ItemAttribute.txt，用于训练模型（可选）；
4. ResultForm.txt，结果文件的形式。

数据集的格式在 DataFormatExplanation.txt 中显示。

```
train.txt
<user id>|<numbers of rating items>
<item id> <score>
test.txt
<user id>|<numbers of rating items>
<item id>
item.txt
<item id>|<attribute_1>|<attribute_2>('None' means this item is not belong to any
    of attribute_1/2)
```

对于训练集 Train.txt，统计得到的基本信息如下所示。

统计量	数值
用户数量	19835
物品数量	624961
有评分物品数	455691
评分总数	5001507
平均评分	49.506
最小用户 ID	0
最大用户 ID	19834
最小物品 ID	0
最大物品 ID	624960

由此可以分析出以下信息：

1. 由于最小用户 ID 为 0，最大用户 ID 为 19834，用户总数为 19835 名；
2. 由于最小物品 ID 为 0，最大物品 ID 为 624960，物品总数为 624961 件；
3. 有评分的物品数为 455691，说明有 169270 件物品未被任何一位用户评分；
4. 总共有 5001507 个评分，所有评分的平均值为 49.506。

由此可知，用户打分效用矩阵大小为 19835*624961=123696101435，而实际评分数只有 5001507 条，整个矩阵只填充了 0.004%，非常稀疏。因此，在存储该矩阵时，应使用存储稀疏矩阵的方法，只存储有评分的内容，避免存储整个矩阵。

5 实验参数设置

算法	参数	数值	含义
CF-user	thresh	0.1	用户相似度筛选阈值
SVD	lr	5e-3	学习率
	factor	50	隐藏因子个数

6 推荐系统

互联网的出现和普及给用户带来了大量的信息，满足了用户在信息时代对信息的需求，但随着网络的迅速发展而带来的网上信息量的大幅增长，使得用户在面对大量信息时无法从中获得对自己真正有用的那部分信息，对信息的使用效率反而降低了，这就是所谓的信息超载问题。

解决信息超载问题一个非常有潜力的办法是推荐系统，它是根据用户的信息需求、兴趣等，将用户感兴趣的信息、产品等推荐给用户的个性化信息推荐系统。和搜索引擎相比推荐系统通过研究用户的兴趣偏好，进行个性化计算，由系统发现用户的兴趣点，从而引导用户发现自己的信息需求。一个好的推荐系统不仅能为用户提供个性化的服务，还能和用户之间建立密切关系，让用户对推荐产生依赖。

推荐系统算法是一种通过分析用户的历史行为和偏好，为用户提供个性化推荐的技术。推荐系统的目的是帮助用户发现他们可能感兴趣的物品，例如商品、电影、音乐或文章，从而提高用户的满意度和网站的转化率。主要的推荐系统算法包括但不限于基于内容的推荐算法、协同过滤算法、基于矩阵分解、基于模型的推荐算法等。不同的推荐系统算法适用于不同的场景和数据结构，开发者需要根据实际情况选择合适的算法来构建有效的推荐系统。

6.1 推荐系统常用算法

6.1.1 基于内容的推荐算法

在基于内容的推荐系统中，项目或对象是通过相关特征的属性来定义的，系统基于用户评价对象的特征、学习用户的兴趣，考察用户资料与待预测项目的匹配程度。用户的资料模型取决于所用的学习方法，常用的有决策树、神经网络和基于向量的表示方法等。基于内容的用户资料需要有用户的历史数据，用户资料模型可能随着用户的偏好改变而发生变化。基于内容的推荐与基于人口统计学的推荐有类似的地方，只不过系统评估的中心转到了物品本身，使用物品本身的相似度而不是用户的相似度来进行推荐。

6.1.2 协同过滤算法

基于协同过滤的推荐算法技术是推荐系统中应用最早和最为成功的技术之一。它一般采用最近邻技术，利用用户的历史喜好信息计算用户之间的距离，然后利用目标用户的最近邻居用户对商品评价的加权评价值来预测目标用户对特定商品的喜好程度，从而根据这一喜好程度来对目标用户进行推荐。

协同算法可以分为以下几类：

1. **基于用户的协同过滤**：基于用户的协同过滤算法通过比较用户对物品的评分或行为来发现相似的用户，并向目标用户推荐那些与相似用户喜欢的物品相同的物品。

2. **基于物品的协同过滤**：基于物品的协同过滤算法则是通过比较物品之间的相似性来向用户推荐相似的物品。此方法通常比基于用户的协同过滤更高效，因此人的习惯爱好容易发生改变，但物品特性不容易改变，因此，得到的结果更具参考性。

6.1.3 SVD

奇异值分解（SVD），是一种线性代数中矩阵分解的方法，任何矩阵都可以通过 SVD 分解成几个矩阵相乘的形式。在推荐系统中，SVD 的主要作用是降维。具体来说，SVD 在推荐系统中的应用可以帮助解决稀疏矩阵的问题，提高推荐系统的效率和效果。以下是对 SVD 在推荐系统中的作用和过程的详细介绍：

1. 构建效用矩阵

   首先，将用户对物品的评分记录整理成一个矩阵。矩阵中的行表示用户，列表示物品，矩阵中的元素表示用户对物品的评分。由于用户通常只对少量的物品进行评分，这会形成一个非常大的稀疏矩阵。

2. 矩阵分解使用 SVD 技术对效用矩阵进行分解，将其分解为三个矩阵的乘积：用户特征矩阵 U、奇异值矩阵 Σ 和物品特征矩阵 V^T。具体的分解形式为：

   $$M = U\Sigma V^T$$

   这里，U 和 V 分别表示用户和物品在特征空间中的表示，Σ 是对角矩阵，包含奇异值。

3. 降维 SVD 的一个关键优势在于降维。通过选择前 k 个最大的奇异值及其对应的特征向量，可以将原始高维矩阵近似为一个低维矩阵。这种降维不仅减小了数据的存储和计算复杂度，还保留了矩阵的大部分重要信息。研究表明，一个矩阵的前 10% 的大的特征值可以代表 90% 的矩阵信息。

4. 预测评分根据降维后的用户特征矩阵和物品特征矩阵的乘积，可以预测用户对未评分物品的评分。通过计算用户特征向量与物品特征向量之间的相似度，可以确定推荐给用户的物品。

5. 个性化推荐根据预测的评分结果，为每个用户推荐最高评分的物品或根据一定的阈值筛选出符合用户兴趣的物品，实现个性化推荐。

6.2 评判指标

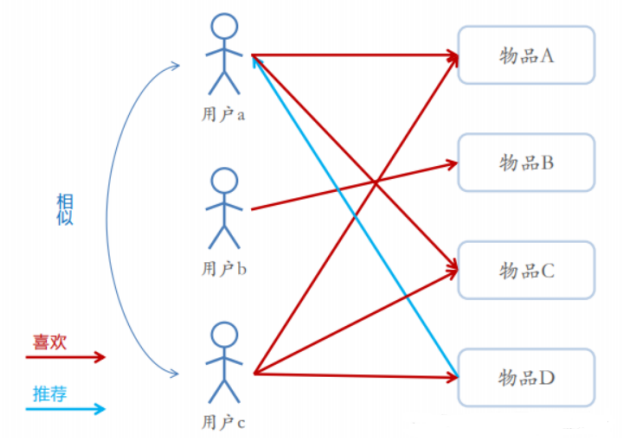
评测指标用于评测推荐系统的性能，有些可以定量计算，如预测准确度、覆盖率、多样性、实时性；有些只能定性描述，如用户满意度、新颖性、惊喜度、信任度、健壮性和商业目标。有了这些指标，就可以根据需要对推荐系统进行优化，通常的优化目标是在给定覆盖率、多样性、新颖性等限制条件下，尽量提高预测准确度。

其中，预测准确度是度量一个推荐系统或其中推荐算法预测用户行为的能力。是推荐系统最重要的离线评测指标。在本次实验中，使用均方根误差（RMSE）来评测所实现的推荐系统的性能。

7 实验原理

7.1 基于用户的协同过滤算法

基于用户的协同过滤算法的基本思想为，当一个用户需要被推荐时，找到与他兴趣相似的用户，根据另一用户的喜好向该用户推荐相应的物品，如下图所示。



因此，首先需要计算用户之间的相似度，即算出相似度矩阵。假设共有 u 个用户，则矩阵大小为 u*u，计算公式如下所示。

$$cor(x,y) = \frac{\sum_{s\ S_{xy}}(r_{xs}-\bar{r_x})(r_{ys}-\bar{r_y})}{\sqrt{\sum_{s\ S_{xy}}(r_{xs}-\bar{r_x})^2}\sqrt{\sum_{s\ S_{xy}}(r_{ys}-\bar{r_y})^2}} \tag{1}$$

其中，r_{xy} 为用户 x 为物品 y 的评分。

之后，就可以根据用户之间的相似度，预测用户对其他物品的评分。由于用户内心评分标准不一，因此可以用该用户的平均评分与其他用户的所有评分的差值进行加权计算，公式如下所示。

$$P_{i,j} = \hat{R}_i + \frac{\sum_{k=1}^{n}(Cor_{i,k} * (R_{k,j} - \hat{R}_k))}{\sum_{k=1}^{n} Cor_{i,k}} \tag{2}$$

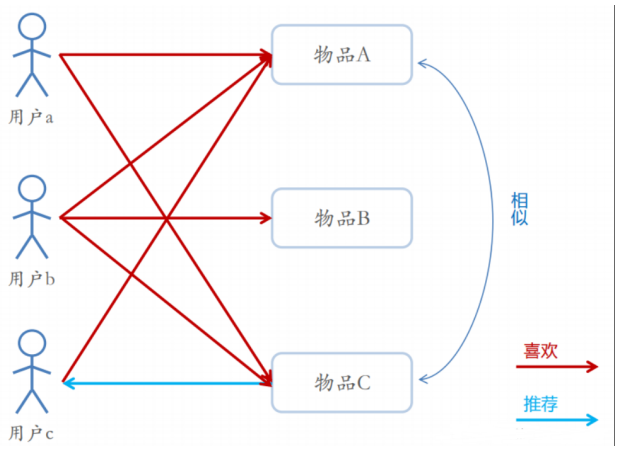
其中，$P_{i,j}$ 为预测的用户 i 对物品 j 的评分；$\hat{R}_i$ 为用户曾经打过的平均评分；$cor_{i,k}$ 为用户 i 与用户 k 的相似度；$\hat{R}_k$ 为用户 k 曾经的平均评分。

综上所述，假设共有 u 个用户和 i 件物品，则在具体的代码实现中，需要准备以下矩阵：

1. 用户评分矩阵，大小为 u*i；
2. 用户相似度矩阵，大小为 u*u；
3. 用户平均评分矩阵，大小为 u*2。

7.2 基于物品的协同过滤算法

基于用户的协同过滤算法的基本思想为，根据物品之间的相似性以及用户的历史行为，为用户生成推荐列表。



在这里同样需要计算物品之间的相似度矩阵，公式同上。

$$cor(i,j) = \frac{\sum_{s\ S_{ij}}(r_{is}-\bar{r_i})(r_{js}-\bar{r_j})}{\sqrt{\sum_{s\ S_{ij}}(r_{is}-\bar{r_i})^2}\sqrt{\sum_{s\ S_{ij}}(r_{js}-\bar{r_j})^2}} \tag{3}$$

其中，r_{xy} 为用户 x 为物品 y 的评分。

预测评分公式也与用户协同过滤算法相似，只是将用户平均评分更改为物品平均评分。

$$P_{i,j} = \hat{R_i} + \frac{\sum_{k=1}^{n}(Cor_{i,k} * (R_{k,j} - \hat{R_k}))}{\sum_{k=1}^{n} Cor_{i,k}} \tag{4}$$

其中，$P_{i,j}$ 为预测的用户 i 对物品 j 的评分；$\hat{R_i}$ 为物品 i 的平均评分；$cor_{i,k}$ 为用户 i 与用户 k 的相似度；$\hat{R_k}$ 为物品 k 的平均评分。

综上所述，需要准备以下数据：

1. 用户评分矩阵，大小为 u*i；
2. 物品相似度矩阵，大小为 i*i；
3. 物品平均评分矩阵，大小为 i*2。

7.3 基于 SVD 的隐语义模型

在大型推荐系统中，用户和物品的数量常常达到百万级甚至更多，单独使用一个矩阵进行求解会变得极为复杂。因此，我们借用了 SVD 的思想，通过隐因子模型（Latent Factor Model，LFM），对评分矩阵进行分解以实现高效的评分矩阵维护。

1. 评分矩阵的分解将评分矩阵 $R_{m,n}$（其中 m 为用户数量，n 为系统中物品数量）分解为两个矩阵的乘积：

$$R_{m,n} = P_{m,F} \cdot Q_{F,n} \tag{5}$$

其中，F 是该推荐系统中设置的隐藏因子个数。通过这种分解，我们可以将原始的评分矩阵表示为用户矩阵 P 和物品矩阵 Q 的乘积。

2. 预测评分需要训练的参数为矩阵 P 和 Q。我们预测用户 u 对物品 i 的评分 $\hat{r}_{ui}$ 的表达式为：

$$\hat{r}_{ui} = \sum_{f=1}^{F} P_{uf} Q_{fi} \tag{6}$$

损失函数为：

$$J = \sum_{rui \neq 0} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(\sum P_{uf}^2 + \sum Q_{fi}^2 \right) \tag{7}$$

其中，λ 是正则化参数，用于防止过拟合。

3. 通过随机梯度下降法进行参数优化和迭代收敛

更新规则如下：

$$\frac{\partial J}{\partial P_{uf}} = \sum_{i, rui \neq 0} -2(r_{ui} - \hat{r}_{ui}) Q_{fi} + 2\lambda P_{uf} \tag{8}$$

$$\frac{\partial J}{\partial Q_{fi}} = \sum_{u, rui \neq 0} -2(r_{ui} - \hat{r}_{ui}) P_{uf} + 2\lambda Q_{fi} \tag{9}$$

4. 添加偏置项

每个用户和物品都有其固有的评分倾向。为了考虑这些个体差异，我们为每个用户和物品添加了偏置项。预测评分公式变为：

$$\hat{r}_{ui} = \sum_{f=1}^{F} P_{uf} Q_{fi} + \mu + b_u + b_i \tag{10}$$

其中，μ 是全局平均评分，b_u 和 b_i 分别是用户和物品的偏置项。

加入偏置项后的损失函数为：

$$J = \sum_{rui \neq 0} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(\sum P_{uf}^2 + \sum Q_{fi}^2 + \sum b_u^2 + \sum b_i^2 \right) \tag{11}$$

偏置项的更新规则如下：

$$b_u^{(t+1)} := b_u^{(t)} + \alpha (r_{ui} - \hat{r}_{ui} - \lambda b_u^{(t)}) \tag{12}$$

$$b_i^{(t+1)} := b_i^{(t)} + \alpha (r_{ui} - \hat{r}_{ui} - \lambda b_i^{(t)}) \tag{13}$$

7.4 评测指标：RMSE

设测试集 T 中有用户 u 和物品 i，用 r_{ui} 表示用户 u 对物品 i 的实际评分；$\hat{r}_{ui}$ 表示推荐算法给出的预测评分，则 RMSE 的公式为：

$$RMSE = \frac{\sqrt{\sum_{(u,i)\in T}(\hat{r_{ui}} - r_{ui})^2}}{|T|} \tag{14}$$

8 核心代码

8.1 划分数据集

由于计算 RMSE 时，需要知道预测分数与实际分数，因此需要划分原始训练集，得到带有分数的验证集，并将其转化为新的测试集。用该测试集预测出相应的分数后，再与验证集进行比较，计算 RMSE。

可以将原始训练集分成训练集、验证集、测试集三类，且训练集与验证集的比例通过参数 ratio 调节。代码如下所示。

```
def split_data(in_file=TRAIN_FILE, ratio=0.01, data_random=True):
    with open(in_file, 'r') as file:
        line = file.readline().strip()
        while line:
            user_id, num_items = line.split('|')
            user_id = int(user_id)
            num_items = int(num_items)
            items = []
            for _ in range(num_items):
                item_id, score = file.readline().strip().split()
                items.append((item_id, score))

            #打乱顺序
            if data_random:
                random.shuffle(items)

            #分割
            index = int(len(items) * ratio)
            train_items = items[index:]
            validation_items = items[:index]

            with open('tmp/train_data.txt', 'a') as train_file:
                #if len(train_items) > 0:
                train_file.write(f"{user_id}|{len(train_items)}\n")
                for item in train_items:
                    train_file.write(f"{item[0]} {item[1]}\n")

            with open('tmp/validation_data.txt', 'a') as validation_file:
                validation_file.write(f"{user_id}|{len(validation_items)}\n")
```

```
            for item in validation_items:
                validation_file.write(f"{item[0]} {item[1]}\n")

        with open('tmp/test_data.txt', 'a') as validation_file:
            validation_file.write(f"{user_id}|{len(validation_items)}\n")
            for item in validation_items:
                validation_file.write(f"{item[0]}\n")

        line = file.readline().strip()
```

8.2 基于用户的协同过滤算法

8.2.1 划分三元组及用户平均评分

首先,为了方便统计以及方便后续计算相似度,先将训练数据切割成 <user_id> <item_id> <score> 三元组，写入到 sparse_matrix.txt 文件中。

由于在预测时需要用到用户平均评分数据，因此，在这一步中可以顺便统计每个用户对物品打出的平均评分。

```
def sparse_matrix(in_file=TRAIN_FILE, out_file1=SPARSE_MATRIX_FILE,
    out_file2=USER_AVERANGE_FILE, out_file3 = ITEM_LIST_FILE):
    '''
    将训练集转换为稀疏矩阵形式，并计算每个用户的平均评分
    :param in_file:
    <user id>|<numbers of rating items>
    <item id> <score>
    :param out_file1: <user id> <item id> <score>
    :param out_file2: <user id> <average score>
    :return:
    '''
    count = 1
    item_list = set()

    with open(in_file, 'r') as file:
        line = file.readline().strip() # 读取第一行并去除首尾空格
        while line:
            user_id, num_items = line.split('|')
            score_sum = 0
            num_items = int(num_items)
            # 读取 num_items 行数据并进行相关操作
            for _ in range(num_items):
                item_id, score = file.readline().strip().split()
                score_sum += int(score)
```

```
            if count%10000 == 0:
                print("进度", (count/5001507)*100, "%")
            count += 1
            with open(out_file1, 'a') as file2:
                file2.write(f"{user_id} {item_id} {score}\n")
            item_list.add(item_id)

        with open(out_file2, 'a') as file3:
            aver = float(score_sum / num_items)
            file3.write(f"{user_id} {aver}\n")

        line = file.readline().strip()

with open(out_file3, 'w') as file:
    # 将集合中的元素逐行写入文件
    for element in item_list:
        file.write(str(element) + '\n')
print("Convert to a sparse matrix!\n")
```

8.2.2 计算用户相似度矩阵

之后，需要进行用户之间的相似度的计算，结果存在 u*u 的矩阵中。其中，矩阵中 (x,y) 的值即为用户 x 与用户 y 之间的相似度。

由于用户评分矩阵太大（矩阵大小约为 2 万 *60 万），遍历计算用户相似度需要花费的时间太长，因此，导入了 coo_matrix 库与 cosine_similarity 作为辅助。前者的作用为将三元组数据转化为稀疏矩阵 (即用户评分矩阵)，后者的作用为计算该稀疏矩阵的相似度，并将相似度矩阵保存在内存中。

```
def calculate_similarity(aver_file = USER_AVERANGE_FILE, sparse_file =
    SPARSE_MATRIX_FILE):
    '''
    计算用户之间的相似度
    :return: correlation matrix 相似度矩阵
    '''
    user_ids = []
    item_ids = []
    scores = []

    with open(sparse_file, 'r') as file:
        for line in file:
            user_id, item_id, score = line.strip().split()
            user_ids.append(int(user_id))
            item_ids.append(int(item_id))
```

```
            scores.append(float(score))


    matrix = coo_matrix((scores, (user_ids, item_ids)), shape=(user_num,
        item_num+1))

    correlation_matrix = cosine_similarity(matrix)
    print(correlation_matrix.shape)
    return correlation_matrix
```

若不使用库，直接手工计算相似度，则需要嵌套三个 for 循环计算每两个用户之间的相似度：前两个循环遍历两个用户，第三个循环遍历物品，找出这两个用户之间都评了分的物品，并进行相应的计算。

```
def calculate_similarity_2(aver_file = USER_AVERANGE_FILE, sparse_file =
    SPARSE_MATRIX_FILE):
    '''
    计算用户之间的相似度（手工计算）
    :return: correlation matrix 相似度矩阵
    '''

    correlation_matrix = np.zeros((user_num, user_num))
    count = 0

    #遍历user_num次
    for i in range(user_num):
        i_item_count = len(user_matrix[i])
        #用户i对其他物品的评分与均值之差
        tmp1 = np.sum([math.pow(user_matrix[i][item] - user_averange[i], 2) for item
            in user_matrix[i]])
        for j in range(i+1, user_num):
            j_item_count = len(user_matrix[j])
            x = 0 #分子
            # 用户j对其他物品的评分与均值之差
            tmp2 = np.sum([math.pow(user_matrix[j][item] - user_averange[j], 2) for
                item in user_matrix[j]])

            #从更少评分物品的列表开始遍历，会更节省时间
            if i_item_count<= j_item_count:
                for item in user_matrix[i]:
                    #有共同评分的物品
                    if user_matrix[j].get(item) is not None:
                        x1 = user_matrix[i][item] - user_averange[i]
```

```
                    x2 = user_matrix[j][item] - user_averange[j]
                    x += x1 * x2
                else:
                    continue
        else:
            for item in user_matrix[j]:
                # 有共同评分的物品
                if user_matrix[i].get(item) is not None:
                    x1 = user_matrix[i][item] - user_averange[i]
                    x2 = user_matrix[j][item] - user_averange[j]
                    x += x1 * x2
                else:
                    continue

        if tmp1== 0 or tmp2 == 0:
            correlation_matrix[i][j] = 0
            correlation_matrix[j][i] = 0
        else:
            correlation_matrix[i][j] = x / (math.sqrt(tmp1 * tmp2))
            correlation_matrix[j][i] = correlation_matrix[i][j]
    count +=1
    if count % 100 == 0:
        print("已经计算了",count, "轮")
return correlation_matrix
```

该算法耗时较长，在 2 万个用户的情况下，需要消耗 8 个小时的时间去遍历；而调用库函数直接计算相似度只需要消耗 1 分钟。

8.2.3 预测评分

由实验原理中给出的预测公式可知，在这一阶段需要准备好用户评分矩阵、用户历史评分均值向量、用户相似度矩阵。因此，需要准备一个 load_data() 函数导入提前计算好的数据。由于该函数较简单，这里不再展示。

读取 test.txt 文件，为文件中的物品预测评分。若用户评分矩阵中未出现该物品，则该物品的评分为用户曾打出的历史平均分；否则，按照 predict() 函数中的计算规则评分。

test() 函数如下所示。

```
def test(correlation_matrix, test_path=TEST_FILE, result_file = RESULT_FILE):
    '''
    测试
    :param correlation_matrix: 相似度矩阵
    '''
    predict_matrix = []
```

```
    count = 0

    with open(test_path, 'r') as file:
        line = file.readline().strip() # 读取第一行并去除首尾空格
        while line:
            if count%1000==0:
                print("test:", count)
            user_id, num_items = line.split('|')
            user_id = int(user_id)
            num_items = int(num_items)
            #拓展predic_matrix, 防止数组越界
            while len(predict_matrix)< user_id + 1 :
                predict_matrix.append([])
            count += 1
            # 读取 num_items 行数据并进行相关操作
            for _ in range(num_items):
                item_id = int(file.readline().strip())
                if item_id not in item_list:
                    p = user_averange[user_id]
                else :
                    p = predict(user_id, item_id, correlation_matrix)
                predict_matrix[user_id].append((item_id, p))


            line = file.readline().strip()



    with open(result_file, 'w') as file:
        for i in range (len(predict_matrix)):
            file.write(str(i) + "|" + str(len(predict_matrix[i]))+ "\n")
            for j in range(len(predict_matrix[i])):
                score = 0
                if predict_matrix[i][j][1] < 0 :
                    score = 0
                elif predict_matrix[i][j][1] > 100 :
                    score = 100
                else:
                    score = round(predict_matrix[i][j][1])
                file.write(str(predict_matrix[i][j][0]) + " " + str(score) + "\n")

    print("Finish test!")
```

其中，对于需要预测评分的物品，计算公式已在前面给出。predict() 函数如下所示。

```
def predict(user, item, correlation_matrix):
    x = 0
    y = 0 #相似度之和
    count = 0
    #遍历每一位用户
    for u in range (user_num):
        if item in user_matrix[int(u)] and correlation_matrix[user][int(u)] >= 0.1:
            x += correlation_matrix[user][int(u)] * (user_matrix[int(u)][item] -
                user_averange[int(u)])
            y += correlation_matrix[user][int(u)]
    if y == 0:
        return user_averange[user]
    else:

        return (x/y + user_averange[user])
```

8.2.4 计算 RMSE

最后，需要将得到的 result.txt 文件和验证集文件 validation_data.txt 作对比，按照实验原理中给出的公式计算，代码如下所示。

```
def evaluate(val_file=VALIDATION_FILE, res_file=RESULT_FILE):
    """
        比较validate_data.txt的结果与result.txt中的结果，计算RMSE
    """
    with open(val_file, "r") as val, open(res_file, "r") as res:
        RMSE = 0
        count = 0

        for line_val, line_res in zip(val, res):
            user_id, num_items = line_val.strip().split('|')
            user_id_r, num_items_r = line_res.strip().split('|')
            assert int(user_id) == int(user_id_r) and int(num_items) ==
                int(num_items_r)
            count += int(num_items)

            for _ in range(int(num_items)):
                item_id, score = val.readline().strip().split()
                item_id_r, score_r = res.readline().strip().split()
                print(f"score_id: {score} score_r:{score_r}")
                assert item_id == item_id_r
```

```
            RMSE += (float(score) - float(score_r)) ** 2

    RMSE = math.sqrt(RMSE / count)
    print(f"最终测试结果: test数量: {count}, RMSE = {RMSE}")
    return RMSE
```

8.3 基于 SVD 的隐语义模型

8.3.1 训练过程

使用 SVD 模型进行训练的流程为:

1. 计算全局平均评分 global_mean。
2. 初始化用户和物品的隐因子矩阵 P 和 Q。
3. 进行多个训练轮次，每一轮训练一个 epoch。
4. 每个 epoch 计算训练损失和验证损失，以及训练和验证的 RMSE。
5. 输出每个 epoch 的损失和 RMSE。
6. 返回训练好的 P 和 Q 矩阵。

```
# 训练模型
def train(bx, bi, train_data, valid_data, factor=50, lr=5e-3, lambda1=1e-2,
    lambda2=1e-2, lambda3=1e-2, lambda4=1e-2, epochs=10):
    global_mean = calculate_global_mean(train_data)
    Q = np.random.normal(0, 0.1, (factor, len(bi)))
    P = np.random.normal(0, 0.1, (factor, len(bx)))

    for epoch in range(epochs):
        train_loss = train_one_epoch(bx, bi, train_data, P, Q, global_mean, lr,
            lambda1, lambda2, lambda3, lambda4)
        #valid_loss = calculate_loss(valid_data, bx, bi, P, Q, global_mean)
        train_rmse = calculate_rmse(train_data, bx, bi, P, Q, global_mean)
        valid_rmse = calculate_rmse(valid_data, bx, bi, P, Q, global_mean)
        print(f'Epoch {epoch + 1} train loss: {train_loss:.6f} train RMSE:
            {train_rmse:.6f} valid RMSE: {valid_rmse:.6f}')
    return P, Q
```

而每一轮次的训练，我们都会对训练集中每条数据进行遍历，根据当前参数计算出预测分值，再通过计算真实值和预测值之间的差值后对全部参数进行梯度下降优化。具体公式见实验原理部分。

```
# 单个训练周期
def train_one_epoch(bx, bi, train_data, P, Q, global_mean, lr, lambda1,
    lambda2, lambda3, lambda4):
```

```
        train_loss, count = 0.0, 0
        for user_id, items in tqdm(train_data.items()):
            for item_id, true_score in items:
                pred_score = predict(user_id, item_id, bx, bi, P, Q, global_mean)
                error = true_score - pred_score

                # 更新参数
                bx[user_id] += lr * (error - lambda3 * bx[user_id])
                bi[item_id] += lr * (error - lambda4 * bi[item_id])
                P[:, user_id] += lr * (error * Q[:, item_id] - lambda1 * P[:,
                    user_id])
                Q[:, item_id] += lr * (error * P[:, user_id] - lambda2 * Q[:,
                    item_id])

                train_loss += error ** 2
                count += 1
        return train_loss / count if count > 0 else 0
```

8.3.2 引入属性进行计算

在 SVD 模型中，我们尝试引入物品属性减小评分误差。主要思路如下：

1. 在每次训练中，拟合真实值与预测评分值之间误差与物品属性值之间的线性关系
2. 更新偏置参数 bi
3. 训练完毕后，用返回的参数用于 test 预测。

在实际视线中，由于属性值过大，尽管做了归一化处理和映射，仍出现了梯度爆炸等问题，导致精度溢出，未能顺利得出结果。经过反复尝试后放弃。

9 实验结果

假设有 U 个用户和 I 件物品，特征向量长度为 K，K 是隐藏因子的数量，E 是迭代次数。最终的 RMSE 与用时如下所示。

算法	RMSE	用时	时间复杂度	占用空间	空间复杂度
CF-user	28.7618	8h(或 7min)	O(IU^2)	3GB	O(UI)
SVD	15.0698	16min	O(E×(U×I×K))	126MB	O(K(U+I))

9.1 RMSE

由表格可知，SVD 算法能够达到最低的 RMSE 值，说明该算法预测的用户评分最接近真实评分。这是因为 SVD 算法基于潜在因子模型，通过将用户-物品评分矩阵分解为用户特征矩阵和物品特征矩

阵，隐含了物品之间的关系和用户对这些隐含特征的偏好。这种模型更能捕捉数据背后的隐藏关系，提高推荐准确性。

9.2 训练时间

9.2.1 时间复杂度分析

CF-user 对于 CF-user 算法，这里只考虑计算用户相似度的时间复杂度。由于嵌套了三个循环，分别遍历用户 i、用户 j 和物品 item，因此算法时间复杂度为 O(IU^2)。

SVD 对于 SVD 算法，模型时间复杂度主要取决于训练过程中的迭代次数和每次迭代的计算量计算预测评分和损失需要遍历训练数据集，因此时间复杂度与数据集的大小成正比。更新模型参数的时间复杂度取决于参数的维度。因此时间复杂度为：

$$O(E \times (U \times I \times K)$$

9.2.2 训练时长

对于训练时间消耗，当 CF-user 算法使用 for 循环手工计算相似度时，耗费的时间为 8 小时；如果调用库中的 cosine_similarity() 直接计算，则训练全过程只需要消耗 7 分钟。在这里讨论的主要是前一种耗时 8 小时的算法。

从表格记录中可以发现 SVD 消耗时间最短。这是因为用户协同过滤算法的大部分时间消耗在计算用户之间相似度上，而用户数量为 2 万，因此遍历耗时较久。虽然 CF-user 耗时较长，但由于用户数量比物品数量少，仍然会比物品协同过滤算法快。SVD 算法计算分解矩阵，需要进行迭代训练，所用时间随参数量和迭代次数相关。

总的来说，SVD 算法在消耗时间和准确度上都要比 CF-user 算法好，为它能够更准确地利用物品之间的相似性进行预测。

9.3 空间消耗

9.3.1 空间复杂度

CF-User 对于 CF-user 算法，需要在内存中维持用户相似度二维数组 correlation_matrix[i][j]，存储用户 i 与用户 j 的相似度，因此空间复杂度为 O(U^2)；此外，还需要存储用户评分矩阵 user_matrix[i][j]，即用户 i 对物品 j 的评分，因此空间复杂度为 O(U*I)。由于物品数量 I 远大于用户数量 U，因此最终的空间复杂度为 O(U^2)+O(U*I)=O(U*I)。

SVD 对于 SVD 算法，空间复杂度主要取决于模型参数的存储需求，以及其他数据结构的存储需求。在这个模型中，主要的空间复杂度来源包括：

1. 用户偏置 bx 和物品偏置 bi 的存储，它们的长度分别是 U 和 I，因此空间复杂度为 $O(U + I)$。
2. 用户和物品的潜在因子向量 P 和 Q 的存储，它们的维度分别是 $K \times U$ 和 $K \times I$，因此空间复杂度为 $O(K \times U + K \times I)$。
3. 全局平均分的存储，空间复杂度为 $O(1)$。

综合起来，总的空间复杂度为：

$$O(U + I + K \times U + K \times I)$$

这是在给定 U 个用户和 I 件物品以及 K 个潜在因子的情况下，模型的空间复杂度。

9.3.2 实际空间占用

实际有 19835 名用户、624961 件物品、5001507 个评分。

CF-user CF-user 算法中，根据空间复杂度，可以计算出实际占用的内存空间大小。其中，用户相似度矩阵存储的数值为双精度浮点数，用户评分矩阵存储的数值为整型；用户平均评分矩阵大小可忽略不计。实际上，在实现用户评分存储时，使用的是 dict 而不是 matrix，因此存储的元素数量为评分数量。

- 用户相似度矩阵：19835 * 19835 * 8 bytes = 3,147,417,800 bytes 2.93 GB
- 用户评分矩阵：5001507 * 4 bytes = 20,006,028 bytes 19.08MB

因此，CF-user 总共需要使用大约 3GB 的内存。

SVD 在给定 19835 名用户和 624961 件物品的情况下，假设潜在因子的数量 $K = 50$，那么空间复杂度大约为：

$$O(19835 + 624961 + 50 \times 19835 + 50 \times 624961)$$

$$= O(19835 + 624961 + 991750 + 31248050)$$

$$= O(32219596)$$

每个浮点数占据 4 字节（32 位），那么空间复杂度将会是：

$$32.22\,\text{million} \times 4\,\text{bytes}$$

$$= 128.88\,\text{million bytes}$$

换算成兆字节（MB）：

$$\frac{128.88}{1024} \approx 125.93\,\text{MB}$$

所以空间复杂度大约是 126MB。

10 实验总结

本次实验我们实现了推荐系统的二主流算法：基于用户的协同过滤算法以及基于 SVD 的隐语义模型，通过分析比对不同算法的实现效果，进一步加深了对推荐系统的理解和掌握。在这过程中，遇

到了因为数值过大而导致梯度爆炸的问题，因此对数值进行恒等映射处理，但由于属性值过大，简单的映射处理和归一化并未有效。囿于时间的原因，并未成功完成实现在 SVD 模型中加入属性进行更精确的预测，但这也为之后的工作留下了开展方向。