南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

大数据计算及应用期末实验报告

实验名称：推荐系统实现

年级：2020 级

小组成员：许克婧 2013211
管昀玫 2013750
黄丹禹 2012030

指导教师：杨征路

2023 年 6 月 6 日

目录

一、 实验要求

任务：预测 Test.txt 文件中对 (u,i) 的评分。

数据集：

(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，这是你的结果文件的形式。

数据集的格式在 DataFormatExplanation.txt 中解释。

你可以使用你从课程或其他资源 (如 MOOC) 中学到的任何算法。一个小组 (最多由三个学生组成) 需要写一份关于这个项目的报告。报告应包括但不限于以下内容:

（1）数据集的基本统计信息 (如用户数量、评分数量、产品数量等);

（2）算法细节;

（3）推荐算法的实验结果 (RMSE、训练时间、空间消耗);

（4）对算法进行理论分析或实验分析。

二、 推荐系统简介

随着互联网和数字化技术的发展，用户面临的信息量不断增加，从购物、阅读、观影到旅游和餐饮等各个领域，都存在大量的选择和内容供用户挑选。然而，用户往往没有足够的时间和精力去逐个筛选和评估这些选项，这就需要一个智能化的系统来为用户提供个性化的推荐。因此，推荐系统应运而生，它被认为是解决数据信息过载的一个有效途径。推荐系统的核心目标是根据用户的兴趣和偏好，预测和推荐可能对用户有价值的物品。它基于用户的历史行为数据、个人信息以及其他辅助信息，使用机器学习、数据挖掘和推荐算法等技术进行数据分析和模型构建，以生成个性化的推荐结果。

目前，推荐系统广泛应用于电子商务、社交媒体、音乐、电影、新闻、搜索引擎等领域。它可以提高用户体验，增加用户粘性，帮助用户发现新的内容和产品，同时也为商家和平台提供了精准的营销和推广渠道。推荐系统在满足用户需求和优化用户体验方面发挥着重要作用，它为用户和商家之间搭建了桥梁，提供了个性化的服务和推荐，推动了信息和资源的有效利用和共享。

(一) 推荐系统数据来源

在推荐系统应用当中，存在两类元素，一类称为用户 (user)，另一类称为项 (item)。用户会偏爱某些项，这些偏好信息必须要从数据中梳理出来。数据本身会表示成一个效用矩阵 (utility matrix), 该矩阵中每个用户-项对所对应的元素值代表的是当前用户对当前项的喜好程度（评分)。这些喜好程度值来自一个有序集合，比如 1 5 的整数集合，这些整数代表用户对项的评级 (比如分别代表评论的星级)。实际生活中该矩阵是非常稀疏的，即意味着大部分元素都未知。一个未知的评级也暗示着我们对当前用户对当前项的喜好信息还不清楚。（参见大数据互联网大规模数据挖掘与分布式处理第 2 版第 239 页)

效用矩阵中的数据是实施推荐系统算法的根本。然而，效用矩阵的数据获取往往十分困难。目前有两种一般的方法可以用于发现用户对项的评级结果。

1. 直接获取

直接邀请用户对项评级。然而这种方法的效果十分有限，因为通常而言用户都不愿意提供反馈，因此最终得到的信息也会由于它们主要来自那些愿意反馈的用户而带有偏向性。

2. 间接获取

即根据用户的行为推理。最明显地，比如用户购买了某个商品，或者观看了某部电影、阅读了一篇新闻报道，那么就有理由认为用户“喜欢”这些项。更一般地，系统还可以从购物之外的其他行为中推出用户的兴趣。例如，如果某个客浏览了某件商品的信息，我们就可以认为他对该商品感兴趣，即使他最终没有购买这件商品。

本次实验中我们实施推荐系统算法的数据来源为老师提供的数据集。

(二) 常用的推荐算法

常见的推荐算法包括基于内容的推荐、协同过滤推荐、深度学习推荐、图网络推荐等。这些算法根据不同的数据特点和问题背景，以不同的方式对用户和物品进行建模和匹配，以实现个性化推荐的目标。下面简介几种常用的推荐算法。

1. 基于内容的推荐

基于内容的推荐算法算是最早应用于工程实践的推荐算法。所谓基于内容的推荐算法 (Content-Based Recommendations) 是基于标的物相关信息、用户相关信息及用户对标的物的操作行为来构建推荐算法模型，为用户提供推荐服务。这里的标的物相关信息可以是对标的物文字描述的 metadata 信息、标签、用户评论、人工标注的信息等。用户相关信息是指人口统计学信息 (如年龄、性别、偏好、地域、收入等等)。用户对标的物的操作行为可以是评论、收藏、点赞、观看、浏览、点击、加购物车、购买等。基于内容的推荐算法一般只依赖于用户自身的行为为用户提供推荐，不涉及到其他用户的行为。

广义的标的物相关信息不限于文本信息，图片、语音、视频等都可以作为内容推荐的信息来源，只不过这类信息处理成本较大，不光是算法难度大、处理的时间及存储成本也相对更高。

2. 协同过滤

协同过滤 (Collaborative Filtering, 简写 CF) 是推荐系统最重要得思想之一，其思想是根据用户之前得喜好以及其他兴趣相近得用户得选择来给用户推荐物品 (基于对用户历史行为数据的挖掘发现用户的喜好偏向，并预测用户可能喜好的产品进行推荐)，一般仅仅基于用户的行为数据（评价，购买，下载等），而不依赖于物品的任何附加信息（物品自身特征）或者用户的任何附加信息（年龄，性别等）。其思想总的来说就是：人以类聚，物以群分。目前应用比较广泛的协同过滤算法是基于邻域的方法，而这种方法主要有两种算法：基于用户的协同过滤算法（给用户推荐和他兴趣相似的其他用户喜欢的产品）和基于物品的协同过滤算法（给用户推荐和他之前喜欢的物品相似的物品)。还有一种更为复杂的是基于模型的协同过滤算法，这里我们也给予了实现。

用户协同过滤（UserCF）：相似的用户可能喜欢相同物品。如加了好友的两个用户，或者点击行为类似的用户被视为相似用户。如我兄弟和她的太太互加了抖音好友，他们两人各自喜欢的视频，可能会产生互相推荐。

物品协同过滤（ItemCF）：相似的物品可能被同个用户喜欢。这个就是著名的世界杯期间沃尔玛尿布和啤酒的故事了。这里因为世界杯期间，奶爸要喝啤酒看球，又要带娃，啤酒和尿布同时被奶爸所需要，也就是相似商品，可以放在一起销售。

模型协同过滤：使用矩阵分解模型来学习用户和物品的协同过滤信息。一般这种协同过滤模型有：SVD，SVD++ 等。

3. 基于 SVD 的推荐算法

SVD(singular value decomposition)，中文名字奇异值分解，顾名思义，是一种线性代数当中矩阵分解的方法，任何矩阵，都可以通过 SVD 的方法分解成几个矩阵相乘的形式。那么 SVD 在推荐系统中有什么作用呢？此处简单介绍一下其最直接的作用：降维。

在推荐系统中，用户数常常远大于物品数，每个用户只会对少量的物品进行打分，如果把用户和物品得分构成一个矩阵，这会是一个非常大的稀疏矩阵，因此在进行推荐之前，需要用到 SVD 进行矩阵的简化 (即降维)，从而大大提升推荐系统的效率和效果。降维不是简单的维度变小，而是维度变小后，矩阵的原始信息没有发生很大改变。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，所以可以借助该思想进行矩阵降维。**(参见** SVD 奇异值分解)

降维技术具有重要的实际意义。主要包括以下几个方面：

(1) 降噪：通过降维可以去除原始数据中的噪声和冗余信息，提高数据的质量和可靠性。

(2) 展示数据分布：降维可以将高维数据映射到低维空间，使得数据的分布更加清晰可见，有助于发现数据中的模式和结构。

(3) 存储和处理效率高：低维数据相比高维数据更加紧凑，占用的存储空间更小，处理速度更快，可以节省计算资源和时间成本。

对于 SVD 的优势，除了上述提及的降维的优点之外，还包括算法稳定、准确率高效果、适用范围广等等。

具体而言，基于 SVD 矩阵分解的推荐算法主要包括以下步骤：

(1) 构建效用矩阵：将用户对物品的评分记录整理成一个矩阵，其中行表示用户，列表示物品，矩阵中的元素表示用户对物品的评分。

(2) 进行矩阵分解：使用 SVD 技术对效用矩阵进行分解，将其分解为用户特征矩阵和物品特征矩阵的乘积。用户特征矩阵包含用户在特征空间中的表示，物品特征矩阵包含物品在特征空间中的表示。

(3) 预测评分：根据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。

(4) 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。

尽管基于 SVD 矩阵分解的推荐算法在推荐系统中取得了一定的成功，但也存在一些挑战，例如处理大规模数据时的计算复杂性和冷启动问题。因此，研究者们也提出了许多基于 SVD 的改进算法和技术，以进一步提升推荐系统的性能和效果。基于 SVD 分解的算法变种很多，比如：FunkSVD、BiasSVD、SVD++ 等**(参见** 推荐系统-矩阵分解 (SVD) 原理和实战)，后文将详细介绍本次实验用到的基于 SVD 的隐语义模型。

三、 算法设计与实验原理

(一) 基于用户的协同过滤

1. 算法思想

当一个用户 A 需要个性化推荐的时候，我们可以先找到和他有相似兴趣的其他用户，然后把那些用户喜欢的，而用户 A 没有听说过的物品推荐给 A，如图1。

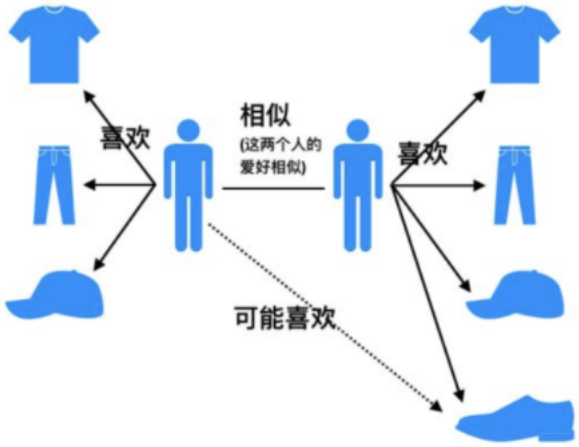


图 1: user-based collaborative filtering

基于用户的协同过滤算法步骤为:

1. 找到和目标用户兴趣相似的其他用户集合
2. 找到这个集合中的用户喜欢的，且目标用户没有听说过的物品推荐给目标用户

该算法的主题部分是计算用户的相似矩阵，以及最终根据与目标用户最为相似的 N 个对目标物品打过分的用户对目标物品的打分进行加权计算最终结果。用户相似度的计算公式如下所示:

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

其中，r_{xs} 为用户 x 对 s 的评分。预测评分的公式有两种:

1. 利用用户相似度和相似用户的平均，加权平均获得用户的平均预测

$$R_{u,p} = \frac{\sum_{s\in S}(w_{u,s}\cdot R_{s,p})}{\sum_{s\in S} w_{u,s}}$$

其中权重 $w_{u,s}$ 是用户 u 和用户 s 的相似度，$R_{s,p}$ 是用户 s 对 p 的评分。

2. 由于有的用户内心的评分标准不一样，有的用户喜欢打高分，有的用户喜欢打低分，所以用该用户的评分与此用户的所有评分的差值进行加权:

$$P_{i,j} = \bar{R_i} + \frac{\sum_{k=1}^{n}(S_{i,k}(R_{k,j}-\bar{R_k}))}{\sum_{k=1}^{n} S_{i,k}}$$

其中，$S_{i,k}$ 是相似度，与第一个式子的 w 意义相同。

在这里，我们采用第二种评分。

2. 算法缺陷

1. **数据稀疏性**：一个大型的电子商务推荐系统一般有非常多的物品，用户可能买的其中不到1%的物品，不同用户之间购买的物品重叠性较低，导致算法无法找到一个用户的偏好相似的用户。这导致 UserCF 不适用与那些正反馈获取较困难的应用场景 (如酒店预订，大件商品购买等低频应用)

2. **用户相似度矩阵维护度大**：UserCF 需要维护用户相似度矩阵以便快速的找出 Topn 相似用户，该矩阵的存储开销非常大，存储空间随着用户数量的增加而增加，不适合用户数据量大的情况使用。在互联网应用场景中，绝大多数产品的用户数都要远大于物品数，因此维护用户相似度矩阵的难度要大很多。其适用于用户少，物品多，时效性较强的场合如新闻推荐场景。

(二) 基于物品的协同过滤（含物品属性）

1. 算法思想

由于 UserCF 的缺陷，导致很多电商平台并没有采用这种算法，而是采用了 ItemCF 算法实现推荐系统。基于物品的协同过滤基本思想是预先根据所以用户的历史偏好数据计算物品之间的相似度，然后把与用户喜欢的物品相类似的物品推荐给用户。ItemCF 算法并不利用物品的内容属性计算物品间的相似度，主要通过分析用户的行为记录计算物品之间的相似度。

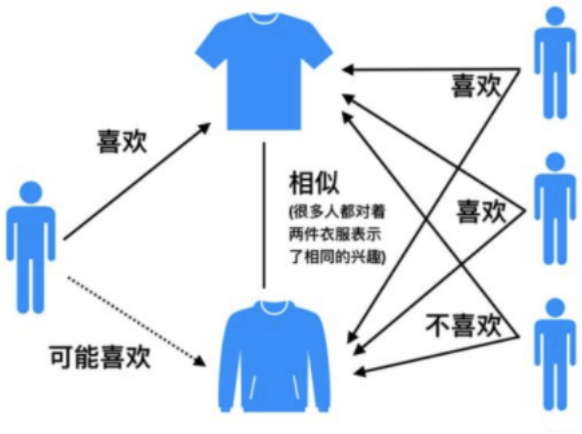


图 2: item-based collaborative filtering

基于物品的算法与基于用户的算法很像，只不过相似度计算是计算物品相似度而已。其算法步骤为：

1. 计算物品之间的相似度
2. 根据物品的相似度和用户的历史行为给用户生成推荐列表（购买了该商品的用户也经常购买的其他商品）

我们在这里使用了一些线性回归的思想。在这里，我们将一个物品的打分分解成以下三个部分：

$$\hat{r_{u,i}} = \mu + b_u + b_i + e$$

其中，各个参数的含义如下：

- $\hat{r_{u,i}}$: 一个商品的评分
- μ: 训练集中所有物品的所有评分的平均值
- b_u: 用户偏置项，表示一个用户评分的平均值偏离全局平均值的程度

- b_i: 物品偏置项，表示一个物品的评分平均值偏离全局平均值的程度
- e: 该个物品相对于全局平均 + 用户偏置 + 物品偏置的偏离分数。

因此，相似度计算公式也需要作相应的更改，为:

$$similarity_{i,j} = (bias\_similarity_{i,j} + attribute\_similarity_{i,j})/2,$$

其中,$bias\_similarity_{i,j}$ 与之前的相似度计算公式相同,即物品的皮尔森系数。而 $attribute\_similarity_{i,j}$ 使用余弦相似度，即:

$$attribute\_similarity_{i,j} = \frac{attr_1 * attr_2}{\sqrt{attr_1^2}\sqrt{attr_2^2}}$$

因此，最终的打分也应做相应的更改，公式与基于用户的协同过滤差值加权评分相同（如1），但是 $\bar{R}_i$ 和 $\bar{R}_k$ 变为了该物品的全局平均 + 用户偏置 + 物品偏置，即 $\mu + b_{u,i} + bi, i$。

该算法的伪代码如下所示:

Algorithm 1 基于物品的协同过滤算法

Input: *train.txt*, 存储的 μ, b_u, b_i

Output: *result.txt*

```
function CollaborativeFiltering(self)
    for each item_i do
        similar_items ← a list of similar items and their similarity
        bias_i ← μ + b_u + b_i
        rating ← 0
        norm ← 0
        for each similar item_j do
            bias_j ← μ + b_u + b_j
            rating += similarity × (rate_j − bias_j)
            norm += similarity

            rating ← bias_i + rating/norm
            predict_rate append rating

        return predict_rate
        =0
```

注：上述伪代码有误，计算最终 rating 是在外层 for 循环，而非内层。

(三) 基于 SVD 的隐语义模型

本实验实现了基于 SVD 矩阵分解的隐语义模型。现介绍其原理。

由于用户和物品之间存在隐藏关联，比如用户买书，那么书的类型、书的作者等因素都可能影响用户的选择，这里提到的书的类型、作者等因素就是隐藏因子，而隐语义模型的核心思想是通过隐含特征 (Latent Factor) 计算用户和物品的相似性。

在大型推荐系统中，用户和物品的数量都在百万数量级甚至更多，如果仅使用一个矩阵，则该矩阵的求解过程极复杂，故本方法借用了 SVD 的思想——矩阵分解的思想，通过 LFM (Latent Factor Model) 隐因子模型，将评分矩阵 $R_{m,n}$（m 为用户数量，n 为系统中物品数量）分解为

$R_{m,n} = P_{m,F} \cdot Q_{F,n}$ ，其中维度 F 是该推荐系统中设置的隐藏因子个数，从而实现评分矩阵的维护。

其中需要训练的参数为 P 和 Q 两个矩阵，收敛对象为用户 u 对物品 i 的预测评分 $\hat{r}_{ui}$ 与真实评分 r_{ui} 之间的差值，其中预测评分 $\hat{r}_{ui}$ 的表达式为：$\hat{r}_{ui} = \sum_{f=1}^{F} P_{uf} Q_{fi}$，损失函数为：$\sum_{r_{ui} \neq 0} (r_{u,i} - \hat{r}_{ui})^2$。训练过程中为了防止过拟合，加入正则项：

$$\text{min: CostFunctionJ } = \sum_{r_u \neq 0} (r_{u,i} - \hat{r}_{ui})^2 + \lambda \left(\sum P_{uf}^2 + \sum Q_{fi}^2 \right)$$

通过随机梯度下降进行迭代收敛：

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

由于每个用户具有个人的整体倾向，比如标准严苛的用户打分整体偏低，标准宽容的用户打分整体偏高，生活用品类物品打分整体偏低、娱乐设施类整体偏高等，所以为每个用户和物品添加固有属性值，代表该用户对于所有物品以及物品收到的评价的整体水平，于是添加偏置项后如下：

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

带偏置项的 LFM 隐因子模型称为 SVD，同样通过梯度下降进行迭代收敛：

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

得到偏置项：

$$b_u^{(t+1)} := b_u^{(t)} + \alpha * \left(r_{u,i} - \hat{r}_{ui} - \lambda * b_u^{(t)} \right)$$
$$b_i^{(t+1)} := b_i^{(t)} + \alpha * \left(r_{u,i} - \hat{r}_{ui} - \lambda * b_i^{(t)} \right)$$

（四） Metrics

均方根误差 RMSE 用于评价推荐算法的实现效果，公式如下：

$$RMSE = \sqrt{\sum_{(i,x) \in R} \frac{(\hat{\mathrm{r}}_{xi} - r_{xi})^2}{|R|}}$$

公式中 $\hat{r}_{ui}$ 表示评分预测值，r_{xi} 表示评分真实值，R 为验证集，|R| 表示验证集大小。

RMSE 表示了预测值与真实值之间的差异，用在验证集上对算法效果进行评估，其值越小，表示预测值越接近真实值，推荐算法实现效果则越好。

四、 数据集分析

本实验所使用的三个数据集为：训练集 train.txt，测试集 test.txt 以及商品属性集 itemAttribute.txt。数据格式如下：

data format explanation

```
1 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)
```

由此知训练集 train.txt 包含了每个用户对不同商品打出的评分，为本次推荐系统实现的主要数据来源。商品属性集 itemAttribute.txt 描述了某商品是否具有具有属性 1 和属性 2，如果不具有其中任何一个属性，则对应属性值为 None；如果具有该属性，则对应属性值数值越大，表示商品对该属性的关联程度越密切。商品属性集 itemAttribute.txt 可作为额外数据优化推荐算法。测试集 test.txt 则要求我们在实现的推荐系统上预测指定用户对指定商品的评分，而并无真实评分作为对照。这要求我们实现推荐系统算法时进一步将 train.txt 按照一定的比例划分为新的训练集和验证集（亦可叫测试集），用新的训练集进行模型训练并在验证集上计算 RMSE 评估训练结果。

接下来对这三个数据集进行分析以便后续能从宏观上对推荐算法预测结果做出评价。

(一) 数据集整体情况统计

在数据预处理中，我们对数据进行了较为完善的统计分析并生成了相关统计信息文件。

数据集统计信息

```
数据集中最小用户id: 0
数据集中最大的用户id: 19834
数据集中用户总数: 19835
数据集中最小商品id: 0
数据集中最大商品id: 624960
数据集中商品总数: 602305
train.txt评分总数: 5001507
train.txt所有得分的平均值: 49.504580
属性集数据条数: 507172
属性1的平均值: 288394.873765
属性2的平均值: 272492.800332
没有被提供属性的商品数: 117789
```

可得出如下结论：

（1）用户 id 最小为 0，最大为 19834，实际用户总数为 19835 名。

（2）商品 id 最小为 0，最大为 624960，实际商品总数为 624961 件。但是有些商品没有被任何用户做过评价，根据统计可以看出所有数据集中总共有 602305 商品曾经被用户打过分，说明绝大部分商品都至少被打过 1 次分。

（3）train.txt 总共包含了 5001507 条评分，所有评分的平均分值为 49.504580。

(4) 商品属性集共提供了 507172 条商品属性，共 117789 个商品没有被提供属性。其中属性 1 的平均值约为 288394.873765，属性 2 的平均值约为 272492.800332。

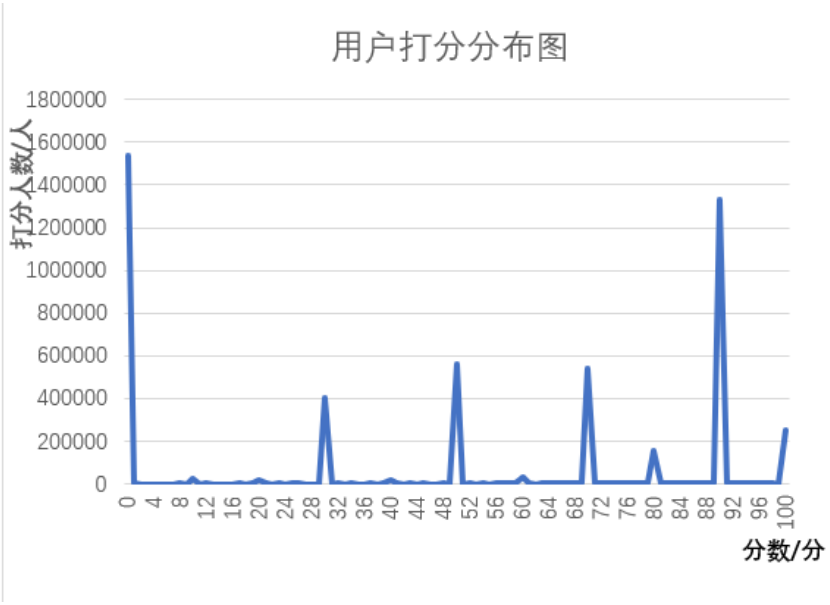
通过上面的分析,我们可以推断出用户打分效用矩阵的大小应该是 19835*624961=12396101435,但 train.txt 中评分总数为 5001507，只占整个效用矩阵的约 0.0403%，由此可见效用矩阵非常稀疏，因此在保存效用矩阵时也应该用稀疏的方式进行保存，以节省计算资源。

(二) 具体统计信息

我们将统计信息绘制成图表以便做出更好的分析。

1. 用户打分分布情况

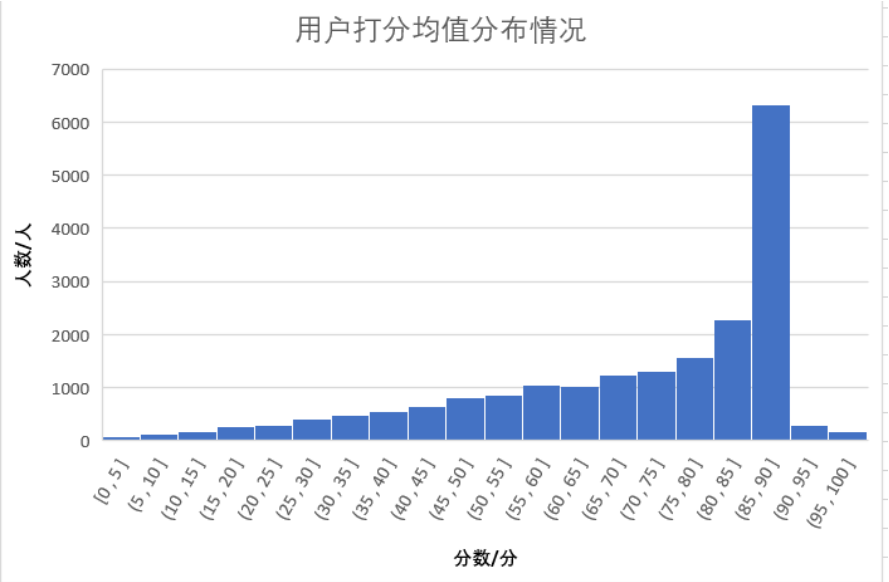
横轴为不同分值，纵轴为打出该分值的人次。



由上图知用户打分主要集中在 0、88 92、48 52、68 72、28 32、96 100、80 84 分。大部分用户打分较为稳定，主要集中在 88 92 分之间，但很大一部分用户对商品打出了 0 分的分值，也有一部分用户打出了 100 分的分值。经统计 0 分这样的极端分值在整个训练集中的占比高达 30%，说明打分出现 0 不是偶然现象，本次实验中 0 分的存在亦是反应商品被评分真实情况的重要参考，故进行算法实现时不将 0 分排除在训练数据之外。

2. 用户打分均值分布

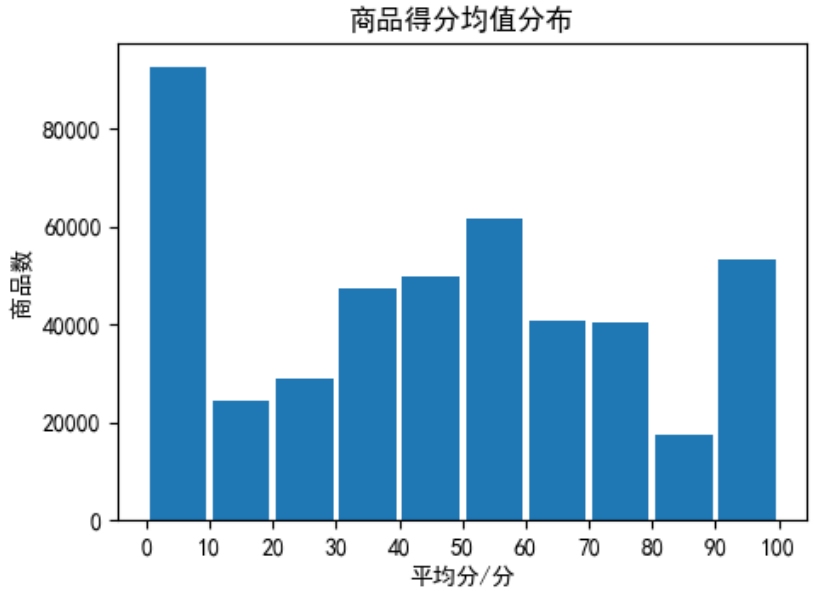
横轴为每个用户给出的评分均分，范围为 0 到 100 分；纵轴为该均分对应的人次。



用户打分平均值主要集中在 85 90、80 85、75 80 这三个分数段上。如果设定 60 分为合格分，可以看到大部分用户给出了合格以上的分数，且给出 85 90 分的用户比重很大。整体来说，对于单个用户而言，其打分较为宽容。

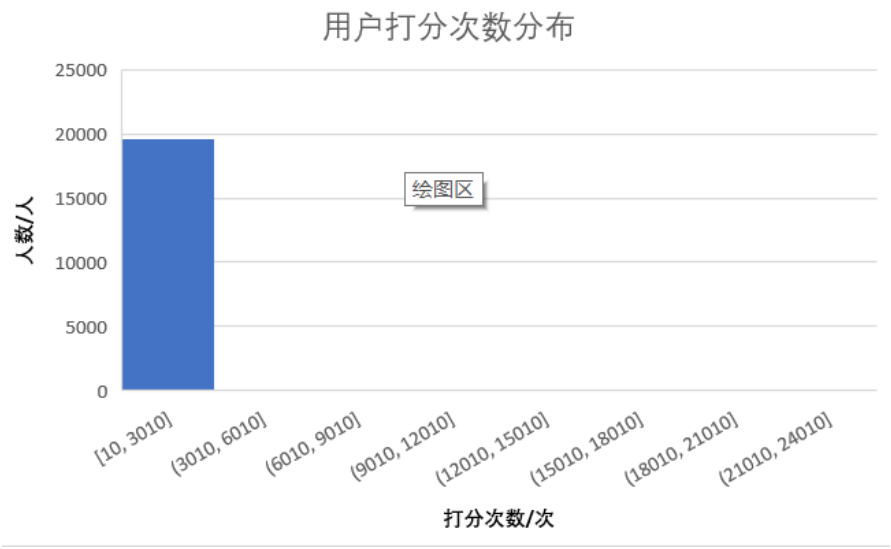
3. 商品得分均值分布

横轴为商品获得的打分平均分，纵轴为具有该平均分的商品数量。



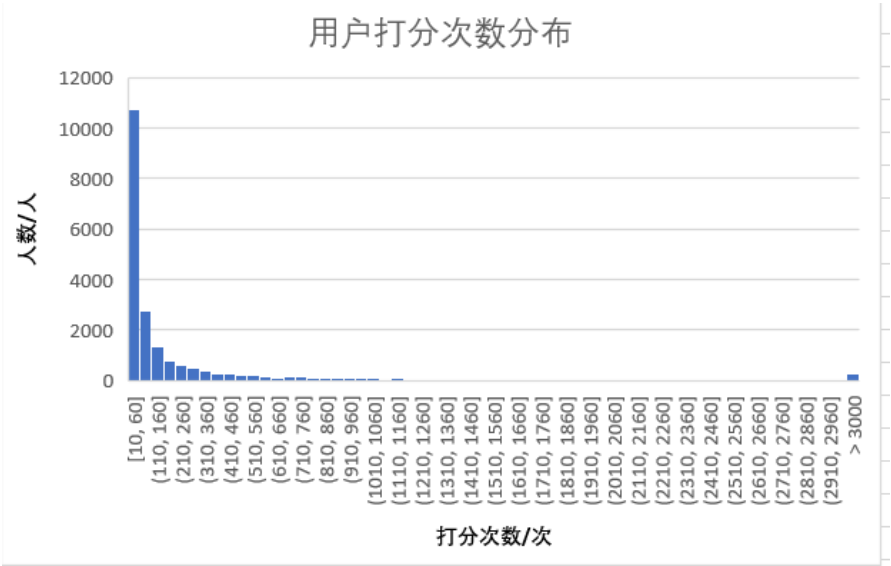
不难发现商品平均得分和前述分析有内在的一致性：有相当一部分数量的商品评分均分在低分段，即 0 10 分；大部分商品获得的评分均分在 50 分以上。

4. 用户打分次数分布



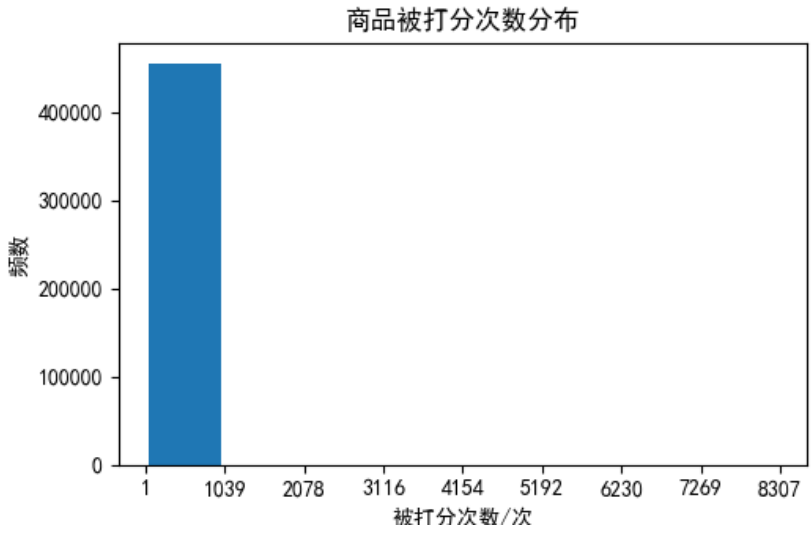
据统计每个用户至少进行过 10 次评分，评分次数最高的用户进行过 21398 次评分。

根据直方图可以看出用户打分次数基本在 3000 次以内，为了更直观地观察分布情况，现将打分次数范围限定在 3000 次以内，绘制出下图：

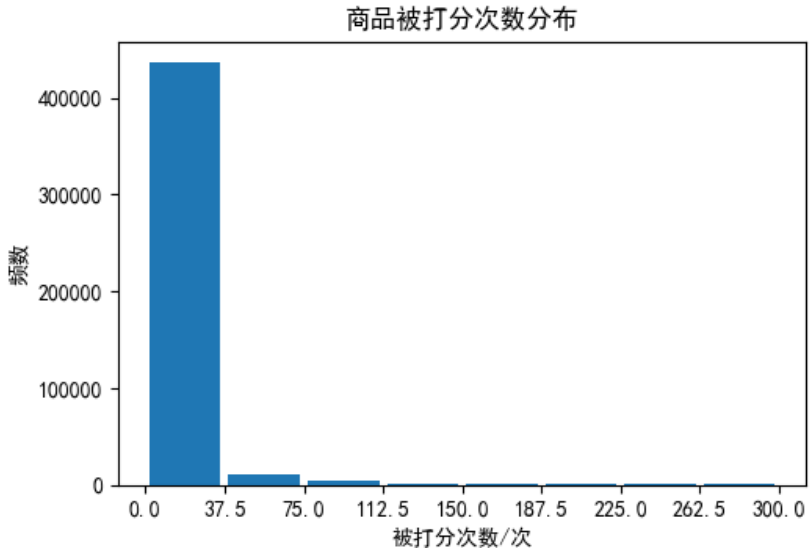


据上图：用户打分次数集中在 10 60 次，说明用户整体评分意愿不高。

5. 商品被打分次数分布



商品被打分次数集中在 1 到 1039 次，因此限制范围重新制图如下：



可以发现大多数商品被评分次数仅在 38 次以内，虽然绝大部分商品至少被评分过 1 次（前述分析已说明），但是被评分次数总体很少，侧面反映出用户总体打分的意愿确实不强。

五、 关键代码细节

(一) 共用函数

1. 划分数据集

我们使用 train_test_split 函数来划分数据集，以进行后续的 RMSE 计算工作。我们使用随机数来划分：如果随机数小于 0.1，那么划分进测试集，否则划分进训练集。需要注意的是，我们需要保证划分后 train 中包含所有 item，在后续计算相似度时我们不会对某个 item 是茫然无知的状态。

train_test_split()

```python
def train_test_split():
    # 按照 split_size 定义的比例划分成train与validate数据集，同时保证划分后train中包含所有item
    # 目的是后续需要计算item与item之间相似度，如果train中不存在则无法计算，影响效果
    item_user_train_data = {}
    user_item_train_data = {}
    train_train_data = []
    train_test_data = []
    train_data = file_read('./Save/train_data.pickle')
    for item, rates in train_data.items():
        for index, (user, score) in enumerate(rates.items()):
            if index == 0: # 每个item的第一个评分放入train中
                train_train_data.append([user, item, score])
                if item not in item_user_train_data:
                    item_user_train_data[item] = {}
                item_user_train_data[item][user] = score
                if user not in user_item_train_data:
                    user_item_train_data[user] = {}
                user_item_train_data[user][item] = score
                continue
            if np.random.rand() < split_size: # 如果随机数小于split_size，放入test中
                train_test_data.append([user, item, score])
            else: # 放入train
                train_train_data.append([user, item, score])
                if item not in item_user_train_data:
                    item_user_train_data[item] = {}
                item_user_train_data[item][user] = score
                if user not in user_item_train_data:
                    user_item_train_data[user] = {}
                user_item_train_data[user][item] = score
    del train_data

    train_test_data = pd.DataFrame(data=train_test_data, columns=['user', 'item', 'score'])
    train_test_data.to_csv('./Save/train_test.csv')
    train_train_data = pd.DataFrame(data=train_train_data, columns=['user', 'item', 'score'])
    train_train_data.to_csv('./Save/train_train.csv')
    file_save(item_user_train_data, "./Save/item_user_train.pickle")
    file_save(user_item_train_data, "./Save/user_item_train.pickle")
    print("train test data")
    static_analyse(len(train_test_data.index.drop_duplicates()),
                   len(train_test_data['item'].drop_duplicates()),
                   len(train_test_data))
    print("train train data")
```

```
    static_analyse(len(train_train_data.index.drop_duplicates()),
                           len(train_train_data['item'].drop_duplicates()),
                           len(train_train_data))
```

2. 修正评分

由于在计算时，我们使用的是 float 数据，因此在最终输出的时候我们还需要进行一些修正。其中，rate_modify 是在正式输出 test 的文件时用的，保证所有数据是整数；而 valid_rate 是在测试 RMSE 时用的，保留了小数，但对小于 0 和大于 100 的数据进行的了修正。

rate_modify()

```
def rate_modify(rate):
    if rate < 0:
        return 0
    elif rate > 100:
        return 100
    else:
        return round(rate)

def valid_rate(rate):
    if rate < 0:
        return 0
    elif rate > 100:
        return 100
    else:
        return rate
```

3. csv2txt

我们在划分数据集时，几个模型总共就划分一次，将结果保存为 csv 文件。对于其他模型来说，训练的输入仍需要 txt 文件，因此这个函数用于将 csv 文件转化为符合格式要求的 txt 文件。

csv2txt()

```
def csv2txt(df, path):
    # 检测目录是否存在
    # if not os.path.exists(path):
    #     # 不存在则创建
    #     os.mkdir(path)

    # 读取CSV文件数据并转换为字典
    data = read_csv(df)

    with open(f'Data/{path}.txt', 'w', encoding='utf-8') as f:
        for user, user_data in data.items():
            user_df_len = len(user_data)

            f.write(f'{user}|{user_df_len}\n')
            for item_data in user_data:
```

```
                item = item_data['item']
                score = item_data['score']
                f.write(f'{item} {score}\n')
```

(二) 基于用户的协同过滤

1. 成员变量与函数

在 CF_user 类中，重要的成员变量与函数如下所示:

成员变量与函数

```
class CF_user():
    def __init__(self, train_p, test_p):
        self.if_build = False
        self.if_train = False
        self.if_test = False
        self.rated_num = 0  # 总评分数
        self.user_matrix = []  # 存储用户对物品的评分 [{itemid: score,...},...]
        self.user_ave = []  # 用户对物品的评分准则(对物品评分的平均数)[u1,u2,...]
        self.sim_matrix_user = None  # user 的相似矩阵（稀疏）lil_matrix
        self.item_list = set() # 物品列表，使用set是为了去重
        self.inverted_item_list = dict() # 物品列表的反向索引
        self.r = []  # predicted matirx
        self.train_p = train_p # 训练集路径
        self.test_p = test_p # 测试集路径
        self.total_sim = 0 # 总相似度数，即相似度矩阵的一半
        self.sim1 = 'sim1.user' # 文件名
        self.sim2 = 'sim2.user' # 文件名
        self.mid = 0 # 相似度矩阵总大小的一半
        self.sec_half = False # 由于将相似度矩阵分成两个文件来存储，因此此变量用于标识是前一半还是后一半

    def static_analyse(self) # 用于打印数据集的基本信息

    def build(self, path) # 读取文件，构建user_matrix，并计算user_ave

    def train(self) # 进行模型训练，计算相似度，并存储在sim_matrix_user中

    def predict(self, user, item_j) # 进行预测商品打分

    def test(self, path) # 读取test文件，调用predict函数进行预测商品打分，并输出
```

2. 重要函数

CF_user : *build*() 这个函数首先遍历 *user_item*。这里使用一个小技巧，即评分总数量比用户 id 总数量多得多，因此使用 *is not None* 语句来增加 hit 的概率，实现加速。使用一个 *item_list* 来存储所有已经出现的 item，并在 user_matrix 中记录商品与评分，其格式为: ['user':'item':score]。每当记录完用户，就计算该用户的平均打分，记录在 user_avg 中。最后，使用 inverted_item_list 来反转 item_list 的索引和 item id。函数实现如下所示:

CF_user: build()

```
def build(self, path):
        print("Building Rating Matrix...")
        user_item = file_read(path)
        temp_count = 0
        score_count = 0
        user_id = None
        user_item_num = None
        for i in user_item:
            # 大部分行为评分，因此使用is not None来增加hit概率
            if user_id is not None:
                now_item, now_score = [int(k) for k in i.split()]
                self.item_list.add(now_item)
                score_count += now_score
                self.user_matrix[user_id][now_item] = now_score
                temp_count += 1
                if temp_count == user_item_num:
                    self.user_avg[user_id] = score_count / temp_count
                    user_id = None
            else:
                score_count = 0
                user_id, user_item_num = [int(j) for j in i.split('|')[:2]]
                self.rated_num += user_item_num
                while len(self.user_matrix) < user_id + 1:
                    self.user_matrix.append({})
                    self.user_avg.append(0)
                temp_count = 0

        user_item.close()
        self.item_list = list(self.item_list)
        self.item_list.sort()
        for x in range(len(self.item_list)):
            self.inverted_item_list[self.item_list[x]] = x
        with open("./Save/user_avg.txt", 'w') as f:
            for i, r in enumerate(self.user_avg):
                f.write(str(i) + " "+ str(r) +'\n')
        self.if_build = True
```

在 CF_user: train() 函数中,我们主要进行计算用户的相似度,并将相似度存储在 sim_matrix_user 中。关于相似度的计算公式前面已给出。需要注意的是，我们使用 math 库，意在加速计算，因为 math 的计算速度比普通的运算能节省大约 18% 的时间。

在存储 sim_matrix_user 时,我们有意将其区分为两个部分存储,因为整个 sim_matrix_user.pickle 文件足有 4.4G 之多。而且，在 test 文件中，用户的排序是正序的，完全可以先读完一部分，然后清空该变量内存，再读取另一个部分。由此想到，分更多的块也是可行的，只是没必要太过麻烦。

CF_user: train()

```
    def train(self):
        start = time.time()
        print(f"Start train at {time.asctime(time.localtime(start))}")
        self.sim_matrix_user = [{} for _ in range(len(self.user_ave))]
        self.now_size = 0
        self.mid = int(len(self.user_ave) / 2)
        count = 0

        def calculate_similarity(i, j, item, rirj):
            m1 = self.user_matrix[i][item] - self.user_ave[i]
            m2 = self.user_matrix[j][item] - self.user_ave[j]
            rirj += m1 * m2
            return rirj

        # 遍历所有用户对，计算相似度
        for i in range(len(self.user_ave)):
            for j in range(i + 1, len(self.user_ave)):
                rirj = 0
                # 由于用户对物品的评分矩阵是稀疏的，因此使用len来判断哪个用户的评分矩阵更稀疏
                # 而且能够减少计算量
                ri2 = np.sum([math.pow(self.user_matrix[i][item] - self.user_ave[i], 2) for item in self.user_matrix[i]])
                rj2 = np.sum([math.pow(self.user_matrix[j][item] - self.user_ave[j], 2) for item in self.user_matrix[j]])
                if len(self.user_matrix[i]) <= len(self.user_matrix[j]):
                    for item in self.user_matrix[i]:
                        if self.user_matrix[j].get(item) is not None:
                            rirj += calculate_similarity(i, j, item, rirj)
                else:
                    for item in self.user_matrix[j]:
                        if self.user_matrix[i].get(item) is not None:
                            rirj += calculate_similarity(i, j, item, rirj)

                if ri2 == 0 or rj2 == 0:
                    self.sim_matrix_user[i][j] = 0
                    self.sim_matrix_user[j][i] = 0
                else:
                    self.sim_matrix_user[i][j] = (rirj / (math.sqrt(ri2) * math.sqrt(rj2)))
                    self.sim_matrix_user[j][i] = self.sim_matrix_user[i][j]

```

```
                count += 1
                if count % pow(10, print_per) == 0:
                    now_time = time.time()
                    print(f"Now time: {time.asctime(time.localtime(now_time))
                        }, batch time: {now_time - start}, {count}/{self.
                        total_sim}")

        for i in range(len(self.user_ave)):
            # 对相似度进行排序，按照相似度从大到小排序
            self.sim_matrix_user[i] = dict(sorted(self.sim_matrix_user[i].
                items(), key=lambda x: x[1], reverse=True))

        now_time = time.time()
        print(f"Begin save at {time.asctime(time.localtime(now_time))}")
        save_class(self.sim_matrix_user, Save_path, os.path.join(Save_path, '
            sim.user'))
        # 由于sim_matrix_user有4G，因此使用两个sim_matrix_user分别保存
        temp = []
        for i in range(self.mid):
            temp.append(self.sim_matrix_user[i])
            self.sim_matrix_user[i] = {}
        save_class(temp, Save_path, os.path.join(Save_path, self.sim1))
        temp = []
        for i in range(self.mid, len(self.user_ave)):
            temp.append(self.sim_matrix_user[i])
            self.sim_matrix_user[i] = {}
        save_class(temp, Save_path, os.path.join(Save_path, self.sim2))
        self.if_train = True
        end = time.time()
        print(f"Now is: {time.asctime(time.localtime(end))}, train time cost
            is {end - start}.")
```

在 CF_user: train() 函数中，它计算了一个物品的预期打分并返回。打分的公式如前文所属，此处不再赘述。如果用户相似矩阵中该商品不存在，那么这个商品的分则应该是该用户所打出的平均分。

CF_user: train()

```
    def predict(self, user, item_j):
        x = 0
        y = 0
        count = 0
        item_j = self.inverted_item_list[item_j]
        if self.if_2:
            user = user - self.mid
        for u in self.sim_matrix_user[user]:
            if self.user_matrix[u].get(item_j) is not None and self.
                sim_matrix_user[user][u] >= Thresh:
                count += 1
```

```
                y += self.sim_matrix_user[user][u]
                x += self.sim_matrix_user[user][u] * (self.user_matrix[u][
                    item_j] - self.user_ave[u])
            if count == topn:
                break

        if y == 0:
            return self.user_ave[user]
        else:
            return x / y + self.user_ave[user]
```

(三) 基于物品的协同过滤（含物品属性）

1. 成员变量与函数

以下为基于物品的协同过滤类的重要成员变量与函数:

CollaborativeFiltering

```
class CollaborativeFiltering:
    def __init__(self, train_path, test_path, attribute_path, is_processed=
        True):
        self.is_processed = is_processed
        self.similarity_map = {} #相似度矩阵
        self.attribute_similarity = {} # 物品相似度
        self.train_path = train_path # 训练集路径
        self.test_path = test_path # 测试集路径
        self.attribute_path = attribute_path # 物品属性路径
        self.split_size = 0.1 # train test split比例
        self.train_data = {} # 训练集
        self.test_data = {} # 测试集
        self.user_item_train_data = {} # {'user':{'item':rate}}
        self.item_user_train_data = {} # {'item':{'user':rate}}
        self.item_attributes = [] # {'item': attr1, attr2}
        self.train_train_data = [] # 自己划分的训练集
        self.train_test_data = [] # 自己划分的测试集
        self.bias = {} # 三部分的bias
        self.num_of_user = 0 # 用户数量
        self.num_of_item = 0 # 物品数量
        self.num_of_rate = 0 # 评分数量

    def static_analyse(self, u, i, r) # 打印数据集信息

    def load_train_data(self) # 载入训练集数据, 存储进train_data, 以 {item:{
        user: rate}} 形式存储, 保证在划分数据集的时候, 训练集能包含所有item项

    def load_test_data(self) # 载入测试集数据, 存储进test_data, 格式为格式为
        {user:{item:score}}

```

```python
    def process_item_attribute(self) # 预处理物品属性信息

    def calculate_data_bias(self) # 计算三个偏置项

    def fetch_similarity(self, item_i, item_j) # 获取相似度

    def calc_similar_item(self, user, item_i) # 计算相似度矩阵

    def train(self) # 训练协同过滤算法

    def predict(self) # 预测打分

    def exec(self) # 执行pipeline
```

2. 重要函数

在 process_item_attribute 这个函数中，首先读入所有的物品属性数据，使用一个 list，即 item_attributes 来存储物品属性。为了后续操作便利，我们将 item_attribute 转换为 dataframe 存储，并以 item 作为索引值，其余的 Nan 值补 0. 我们提前计算模长 morn，以减少后续的计算时间，而且使用 math 库计算来提高效率。最后输出两个属性的基本信息，并存储回了 dict(list) 的形式。这里将其转化回 dict 是因为底层 hash 能使复杂度降为 $O(1)$。

process_item_attribute

```python
    def process_item_attribute(self):
        attr = file_read(self.attribute_path)
        num_of_item = 0
        for i in attr:
            i = i.strip()
            item, attr1, attr2 = i.split('|')
            item = int(item)
            if item > num_of_item:
                for i in range(num_of_item, item):
                    self.item_attributes.append([i, None, None])
            num_of_item = item
            attr1 = None if attr1 == "None" else int(attr1)
            attr2 = None if attr2 == "None" else int(attr2)
            num_of_item += 1
            self.item_attributes.append([item, attr1, attr2])
        del attr
        # 使用DataFrame数据结构，后续处理更加方便
        self.item_attributes = pd.DataFrame(data = self.item_attributes,
            columns=['item', 'attribute1', 'attribute2'])
        self.train_train_data.set_index('item', inplace = True)   # 将item列
            设置为索引
        # 使用 0 对空值进行填充，后续遇到属性全零项，属性相似度为0
        self.item_attributes["attribute1"].fillna(0, inplace = True)
        self.item_attributes["attribute2"].fillna(0, inplace = True)
        # 提前计算模长，减少训练时的计算时间
```

```
        # 使用math可以提高计算效率
        self.item_attributes["norm"] = self.item_attributes.apply(lambda x: \
            math.sqrt(math.pow(x["attribute1"], 2) + math.pow(x["attribute2"
                ], 2)), axis=1)

        print(f"number of items: {num_of_item}")
        print("items information: \nAttribute1:")
        print(self.item_attributes["attribute1"].describe())
        print("Attribute2:")
        print(self.item_attributes["attribute2"].describe())
        # 将DataFrame转为dict, 提高训练时的查询效率
        item_attributes = {}
        for item, row in self.item_attributes.iterrows():
            item_attributes[item] = [int(row['attribute1']), int(row['
                attribute2']), row['norm']]
        self.item_attributes = item_attributes
        file_save(self.item_attributes, "./CF_item/Save/item_attributes.
            pickle")
```

在 calculate_data_bias() 这个函数中，我们将之前处理好的 train_train_data（数据结构为 dataframe，有三个列：item, user, score）进行进一步处理。我们首先使用 mean() 函数计算出全局 μ，然后使用两个 groupby() 语句分别计算用户偏置项 b_u 和物品偏置项 b_i。最后，将计算好的信息存储到 bias 中，并保存中间过程文件。

calculate_data_bias

```
    def calculate_data_bias(self):
        # 计算全局bias信息
        mean = self.train_train_data['score'].mean()
        # 使用groupby计算每个用户和每个商品的bias
        user_bias = self.train_train_data['score'].groupby(
            self.train_train_data['user']).mean() - mean
        item_bias = self.train_train_data['score'].groupby(
            self.train_train_data.index).mean() - mean
        self.bias["mean"] = mean
        self.bias["user_bias"] = dict(user_bias)
        self.bias["item_bias"] = dict(item_bias)
        file_save(self.bias, "./CF_item/Save/bias.pickle")
```

fetch_similarity() 这个函数较为简单，主要用途即从已计算好的 similar_map 中取相似度。如果不存在，则返回 None，后续将要重新进行计算。

fetch_similarity

```
    def fetch_similarity(self, item_i, item_j):
        similar_item = None
        if item_i in self.similarity_map and item_j in self.similarity_map[
            item_i]:
            similar_item = self.similarity_map[item_i][item_j]
        elif item_j in self.similarity_map and item_i in self.similarity_map[
            item_j]:
```

```
            similar_item = self.similarity_map[item_j][item_i]
        else:
            similar_item = None
        return similar_item
```

calc_similar_item() 这个函数主要用户计算物品的相似度。

首先取出 bias_i，$bias_i = \mu + b_{ii}$，然后遍历该用户的打分列表，计算 $bias_j$，并计算二者属性值的余弦距离。之后再用一个循环遍历 $item_i$ 的用户列表和 $item_j$ 的用户列表，计算二者的相似度，并使用属性预先距离对相似度进行调整。最后将计算结果存储在 similarity_map 中，并返回 similar_item，其格式为'item':similarity，即与物品 i 所有的相关物品 j 的相似度。

calc_similar_item

```
    def calc_similar_item(self, user, item_i):
        bias_i = self.bias["item_bias"][item_i] + self.bias["mean"]
        similar_item = {}
        # 计算物品相似度
        for item_j in self.user_item_train_data[user].keys():
            # 如果有在similarity_map中，直接取出
            similar_item[item_j] = self.fetch_similarity(item_i, item_j)
            # 如果不在similarity_map中，计算相似度
            if similar_item[item_j] is None:
                if item_j in self.bias["item_bias"]:
                    bias_j = self.bias["item_bias"][item_j] + self.bias["mean
                        "]
                else:
                    bias_j = self.bias["mean"]
                if self.item_attributes[item_i][2] == 0 or self.
                    item_attributes[item_j][2] == 0:
                    attribute_similarity = 0
                else:
                    # attribute 1, attribute 2 相乘后除以范式，以计算属性相似
                        度
                    attribute_similarity = (self.item_attributes[item_i][0] *
                         self.item_attributes[item_j][0]
                                            + self.item_attributes[item_i][1]
                                                * self.item_attributes[
                                                item_j][1]) \
                                            / (self.item_attributes[item_i
                                                ][2] * self.item_attributes[
                                                item_j][2])
                norm_i = 0
                norm_j = 0
                sim_ = 0
                count = 0
                # 想办法提升性能，不然三个循环太慢了
                # norm_i = np.sum([math.pow(self.item_user_train_data[item_i
                    ][user] - bias_i, 2) for user, item in self.
                    item_user_train_data[item_i].items()])
```

```
                norm_j = np.sum([math.pow(self.item_user_train_data[item_j][
                    user] - bias_j, 2) for user, item in self.
                    item_user_train_data[item_j].items()])
                if item_i in self.item_user_train_data:
                    for same_user, score in self.item_user_train_data[item_i
                        ].items():
                        norm_i += math.pow(self.item_user_train_data[item_i][
                            same_user] - bias_i, 2)
                        if same_user not in self.item_user_train_data[item_j
                            ]:
                            continue
                        count += 1
                        sim_ += (self.item_user_train_data[item_i][same_user]
                             - bias_i) \
                            * (self.item_user_train_data[item_j][same_user] -
                                bias_j)
                    if count < 20:
                        sim_ = 0
                    if sim_ != 0:
                        sim_ /= math.sqrt(norm_i * norm_j)
                similarity = (sim_ + attribute_similarity) / 2
                # 对称填充item_x
                if item_i not in self.similarity_map:
                    self.similarity_map[item_i] = {}
                self.similarity_map[item_i][item_j] = similarity
                similar_item[item_j] = similarity
        return similar_item
```

由于训练函数与 predict() 函数大致相似，这里就只说明 predict 函数。我们使用 pred_dict 来保存预测值。首先二重循环遍历 test 文件具体到某个具体的 item_i, 并计算与其相似的 item_j 和相似度。更正 bais_i 和 bias_j，使用得分公式1来计算最终的物品得分，并存储在 pred_dict 中。

predict

```
    def predict(self):
        index = 0
        pred_dict = ""
        pred_dict = defaultdict(dict)
        for user, items in self.test_data.items():
            pred_dict[user] = {}
            for item_i in items:
                pred_dict[user][item_i] = 0
                rating = 0
                if item_i in self.bias["item_bias"]:
                    similar_item = self.calc_similar_item(user, item_i)
                    similar_item = sorted(similar_item.items(), key=lambda
                        item: item[1], reverse=True)
                    bias_i = self.bias['mean'] + self.bias['item_bias'][
```

```
                    item_i] + self.bias['user_bias'][user]
                norm = 0
                for i, (item_j, similarity) in enumerate(similar_item):
                    if i > topn:
                        break
                    if item_j in self.bias["item_bias"]:
                        bias_j = self.bias['mean'] + self.bias['item_bias
                            '][item_j] + self.bias['user_bias'][user]
                    else:
                        bias_j = self.bias["mean"] + self.bias['user_bias
                            '][user]
                    rating += similarity * (self.item_user_train_data[
                        item_j][user] - bias_j)
                    norm += similarity
                if norm == 0:
                    rating = 0
                else:
                    rating /= norm
                rating += bias_i
            else:
                rating = self.bias['mean'] + self.bias['user_bias'][user]
            index += 1
            pred_dict[user][item_i] = rate_modify(rating)
            if index % 1000 == 0 and index != 0:
                print("predicted", index)
    with open('./CF_item/Save/result_CF_bias.txt', 'wb') as f:
        for user, _ in pred_dict.items():
            f.write(str(user)+'|6\n')
            for item, score in pred_dict[user]:
                f.write(str(item)+' '+str(score)+'\n')
    file_save(pred_dict, "./CF_item/Save/result_CF_bias.pickle")
```

(四) 基于 SVD 的隐语义模型

1. 训练过程

从已经划分好的训练集读入数据后，首先将本次训练所用到的一些参数打印出来，方便实验中进行统计。由于需要训练的参数（P、Q 矩阵，用户、物品偏置值）已经在构造函数中初始化完成，故可以直接开启训练。

梯度下降 训练过程中的核心步骤是对四个参数进行梯度下降优化，通过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，具体公式见三，代码如下：

```
    for epoch in range(0, self.epochs):
        rmse = 0.0
        self.lil_matrix_diff = [] #打分预测值与真实值之间的差距

```

```
        for u, i, r in self.lil_matrix:
            # 计算预测分值
            rp = self.overall_train_mean_rating + self.user_bias[u] + self.
                item_bias[i] + self.dot(u, i)
            rp = self.correctRank(rp)

            #计算真实值和预测值之间的差值
            diff = r - rp
            # 修正偏置参数: 使用四个Lambda值LambdaUB、LambdaIB、LambdaP、
                LambdaQ
            self.user_bias[u] += self.learning_rate * (diff - self.LambdaUB *
                 self.user_bias[u])
            self.item_bias[i] += self.learning_rate * (diff - self.LambdaIB *
                 self.item_bias[i])
            for k in range(0, self.factors):
                # 每次迭代计算矩阵之前需要保存原值, 否则越迭代值越大, 最终导
                    致超过浮点数范围
                q_i_k = self.Q[i][k]
                p_u_k = self.P[u][k]

                self.P[u][k] += self.learning_rate * (diff * q_i_k - self.
                    LambdaP * p_u_k)
                self.Q[i][k] += self.learning_rate* (diff * p_u_k - self.
                    LambdaQ * q_i_k)
            rmse += diff ** 2
            self.lil_matrix_diff.append((u, i, r, diff))

        rmse /= self.num_trainingData
        rmse = math.sqrt(rmse)
```

在迭代中记录训练集中每条数据的损失值、差值平方，最后在每个 epoch 后获得本次训练的总损失值和 RMSE 值。

动态学习率 在梯度下降过程中使用到学习率参数，根据深度学习等课程中学习到的知识和方法，我们知道当学习率较大时，算法会以更快的速度朝着梯度的方向更新参数值，从而加速算法的收敛速度。而当梯度变化很小时，学习率变小可以使得算法更加稳定，避免出现过度更新的情况。而动态调整的学习率能够减少梯度下降算法迭代的次数，因此可以快速找到合适的最优解。故在本次实验中，我们采取动态调整学习率的方法对训练过程进行优化。在每个 epoch 结束后，在当前学习率基础上乘 decay_factor（0-1 之间），以提高算法的收敛速度和准确性，具体代码如下：

```
    # 每次迭代之后降低学习率, 提高正确率, 防止因为学习率过高错过正确的值
    self.learning_rate *= decay_factor
    print('RMSE in epoch %d: %f' % (epoch, rmse))
```

注：**decay_factor** 值的选取见**4**部分。

参数保存 经过 epoches 轮迭代，最终得到训练好的参数 b_i，b_u，P，Q，将这些参数和实验中关键的数据结构分别存储到本地。在 evaluate 阶段可以对参数等直接读取，减少所用时间，具体

保存代码如下（以保存 b_u 为例）：

```
    f1 = open(USER_BIAS_VEC, 'wb')
    f1.write(struct.pack('i', self.user_num))
    for i in self.user_bias:
        f1.write(struct.pack('d', i))
    f1.close()
```

2. itemAttribute.txt 的使用

由于不同物品具有不同的属性，用户在对物品进行打分时会因为这些属性而产生评分上的影响，所以在进行分数预测时，我们需要增加一个属性偏置项，代表来自物品本身的属性对打分的影响值。

Itemattribute.txt 文件主要表达了各个物品在属性 1、2 上的特征值，不同物品对于同一特征的特征值越接近，代表二者受到来自属性的影响越相近，则二者的属性偏置值越接近。为了利用商品属性集，我们对其进行如下处理：将属性值 None 改为 0，并将缺失属性的商品的两个属性都用属性平均值代替（这是一种简单而直观的填补缺失商品属性的方法）。本次实验中我们通过多元线性回归的方法建立预测得分偏差值与两个属性之间的线性关系，并将此关系应用在打分预测中得到属性偏置值，以获得更小偏差、更低 RMSE 值。

该关系的建立的核心通过 linear 函数实现，函数首先读取之前保存的已经训练好的参数和变量，读入相对应的矩阵中，后调用 basic_linear 函数通过最小二乘法拟合真实值和预测评分之间的差值，利用公式 $Residuals = a \times attr1 + b \times attr2 + c$ 计算得到当前物品的固有属性对预测评分的影响差值，并以此作为偏置项加入最终评分结果，具体实现如下：

```
def linear(self):
    #1.从已经存储的.dat文件中读取参数
    print("正在读取保存的参数...")
    self.user_bias = []
    self.item_bias = []
    self.P = []
    self.Q = []
    #按照存储写出的方式读入参数

    #1.1 读取UB_VECTOR.dat
    with open(USER_BIAS_VEC, 'rb') as f:
        byte_str = f.read(4)
        user_len = struct.unpack('i', byte_str)[0]
        for i in range(user_len):
            byte_str = f.read(8)
            x = struct.unpack('d', byte_str)[0]
            self.user_bias.append(x)
    f.close()

    #1.2 读取IB_VECTOR.dat
    with open(ITEM_BIAS_VEC, 'rb') as f:
        byte_str = f.read(4)
        item_len = struct.unpack('i', byte_str)[0]
        for i in range(item_len):
```

```
            byte_str = f.read(8)
            l = struct.unpack('d', byte_str)[0]
            self.item_bias.append(l)
    f.close()

    #1.3 读取P_MATRIX.dat
    with open(P_MATRIX, 'rb') as f:
        byte_str = f.read(4)
        user_len = struct.unpack('i', byte_str)[0]
        byte_str = f.read(4)
        factor_len = struct.unpack('i', byte_str)[0]
        for i in range(user_len):
            new_list = []
            for j in range(factor_len):
                byte_str = f.read(8)
                l = struct.unpack('d', byte_str)[0]
                new_list.append(l)
            self.P.append(new_list)
    f.close()

    #1.4 读取Q_MATRIX.dat
    with open(Q_MATRIX, 'rb') as f:
        byte_str = f.read(4)
        item_len = struct.unpack('i', byte_str)[0]
        byte_str = f.read(4)
        factor_len = struct.unpack('i', byte_str)[0]
        for i in range(item_len):
            new_list = []
            for j in range(factor_len):
                byte_str = f.read(8)
                l = struct.unpack('d', byte_str)[0]
                new_list.append(l)
            self.Q.append(new_list)
    f.close()

    #1.5 读取LIL_MATRIX.dat
    with open(LIL_MATRIX, 'rb') as f:
        byte_str = f.read(8)
        self.overall_train_mean_rating = struct.unpack('d', byte_str)[0]
        byte_str = f.read(4)
        self.num_trainingData = struct.unpack('i', byte_str)[0]
        for ii in range(self.num_trainingData):
            byte_str = f.read(4)
            u = struct.unpack('i', byte_str)[0]
            byte_str = f.read(4)
            i = struct.unpack('i', byte_str)[0]
            byte_str = f.read(8)
            r = struct.unpack('d', byte_str)[0]
```

```
            byte_str = f.read(8)
            err = struct.unpack('d', byte_str)[0]
            self.lil_matrix_diff.append((u, i, r, err))
    f.close()

    #2.读取itemAttribute.csv的数据
    self.loadItemAttributeDataset()

    #3.对所有用户，用最小二乘法拟合残差值
    self.user_para={}
    # self.user_para = defaultdict(list)
    for k, v in self.user_item_attrs.items():
        self.user_para[k] = self.basic_linear(k)
```

```
def basic_linear(self, user):
    # equation: Residuals = a * attr1 + b * attr2 + c
    def regression(x, y, p):  # 回归函数
        a, b, c = p
        return a * x + b * y + c
    # 残差函数
    def residuals(p, z, x, y):
        return z - regression(x, y, p)

    l = len(self.user_item_attrs[user])
    # 存当前用户打分的所有商品的attr1
    x = np.array([self.user_item_attrs[user][i][2] for i in range(0, l)])
    # 存当前用户打分的所有商品的attr2
    y = np.array([self.user_item_attrs[user][i][3] for i in range(0, l)])
    # 存商品真实值和预测值之间的残差
    z = np.array([self.user_item_attrs[user][i][1] for i in range(0, l)])
    # 最小二乘法拟合
    plsq = optimize.leastsq(residuals, [0, 0, 0], args=(z, x, y))
    # 获得拟合结果
    a, b, c = plsq[0]

    return a, b, c
```

3. 测试过程

直接测试（evaluateWithTrain 函数） 通过对划分好的验证集中的数据逐条读取，预测出每条数据的评分，并与真实评分进行对比得到差值，计算得到 RMSE:

```
def evaluateWithTrain(self):
    print('通过验证集计算RMSE...',end='')
    num_test_data = 0
    rmse = 0.0
    with open(test_set, 'r') as f:
        for line in f.readlines():
```

```
            if line == "":
                    continue
            user_id, item_id, rate_str = line.split(',')
            u = self.user_dict[int(user_id)]
            i = self.item_dict[int(item_id)]
            r = float(rate_str)
            rp1 = self.overall_train_mean_rating + self.user_bias[u] + self.
                item_bias[i] + self.dot(u, i)
            rp1 = self.correctRank(rp1)
            err1 = r - rp1
            rmse += err1 ** 2
            num_test_data += 1
    rmse = math.sqrt(rmse / num_test_data)
    print(f'验证集上的RMSE为: {rmse}')
```

该函数使用公式

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

进行预测，多用于在训练过程中对于每轮次训练的结果的测试，用来判断参数是否过拟合。

使用 itemattribute.txt 进行测试(evaluate 函数) 首先通过调用 linear 函数通过最小二乘法对真实值和预测值的差值进行拟合，得到该用户对该物品针对不同属性的影响权重值 user_para，之后通过与 evaluateWithTrain 函数相同的原理得到初步预测结果 rp1，再通过 linear_predict 函数得到根据属性得出的对评分的影响差值，将此值与原 rp1 相加，得到最终预测结果 rp2。

其中的核心函数 linear_predict 将现有不同属性的权重与其对应属性值相乘，得到该物品的属性偏置值，其具体实现如下:

```
def linear_predict(self, u, i):
    item_attr = self.item_attr.get(i)
    user_para = self.user_para.get(u)

    if item_attr is None or user_para is None:
        return 0.0

    attr1, attr2 = item_attr
    a, b, c = user_para
    return a * attr1 + b * attr2 + c
```

4. 调整参数

训练集 & 验证集的划分 实验中考虑了两种基本划分，在控制其他参数全部相同的情况下进行训练和实验。其中通过在每轮训练结束时对验证集结果进行预测，并计算当前轮数下验证集的 RMSE，在确保没有过拟合的情况下，选用 20 个 epoch 进行实验，实验结果如下:

train_set : test_set	0.8 : 0.2	0.9 : 0.1
20epoch-RMSE	26.86294135	26.64038626

故在实验中将原有 train.txt 中的全部数据按照 9:1 的比例进行划分，具体划分函数见1。

分析：由于本次实验所给数据集并不是非常大，所以给予模型更多的训练样本有助于提高模型的训练效果。

粗粒度调整梯度下降的 lambda 在不同维度，以不同粒度选取了不同的 lambda 值进行实验，初步结果如下：

decay=0.5	**0.01*2+0.04*2**	**0.04*4**	**0.01*4**
10epoch-RMSE	26.57652531	26.61374422	26.60332592

故在此阶段，选取 LAMBDAUB、LAMBDAIB、LAMBDAP、LAMBDAQ 都为 0.01 进行继续探索。

注其中 **0.01*2+0.04*2** 代表 **LAMBDAUB**、**LAMBDAIB** 为 **0.01**，**LAMBDAP**、**LAMBDAQ** 为 **0.04**。

探索细节 lambda 的调整方法 观察两个偏置的参数 LAMBDAUB、LAMBDAIB 还是两个矩阵的参数 LAMBDAP、LAMBDAQ 需要相对更低的 lambda，由于两个偏置和矩阵的梯度下降公式类似，故不将不同的偏置值即矩阵分开讨论。

decay=0.5	**0.001*2+0.01*2**	**0.01*2+0.001*2**
10epoch-RMSE	26.60681295	26.60257222

通过上表可以得出，对两个矩阵用更小的 lambda 更有助于提升预测准确率，所以在下一组实验中采用两个偏置的 lambda 值（LAMBDAUB、LAMBDAIB）用 0.01，两个矩阵的 lambda 值（LAMBDAP、LAMBDAQ）用 0.001。

细粒度调整梯度下降的 lambda 将上一探索中得到的结论进行应用并进一步实验。

decay=0.5	**0.01*2+0.001*2**	**0.01*2+0.003*2**	**0.01*2+0.005*2**	**0.01*2+0.007*2**	**0.01*2+0.009*2**
10epoch-RMSE	26.60257222	26.60812455	26.63498686	26.59485623	26.62035724

结合探索出的结论加以实验，最终在实验中选用 LAMBDAUB、LAMBDAIB 值为 0.01，LAMBDAP、LAMBDAQ 值为 0.007 的参数组进行实验。

调整学习率 在控制参数不变的情况下，对学习率进行单一改变，实验结果如下：

lr	**0.001**	**0.002**	**0.003**	**0.01**
10epoch-RMSE	27.16287361	26.59485623	26.65270012	27.16017435

故在后续实验中采用学习率为 0.002。

调整学习率迭代值 decay 实验中使用动态学习率，在每个 epoch 中学习率逐渐变小，通过 self.learning_rate *= decay_factor 进行控制，所以需要对此参数进行调整。

0.01*4-decay	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
10epoch-RMSE	27.00184488	26.78626485	26.66640113	26.68515863	26.60332592	26.61887989	26.70609483	26.80634503	27.042709

最后选用 decay_fatctor 为 0.5 作为最后的参数进行实验。

0.001*4-decay	0.5	0.6	0.7	0.8	0.9
10epoch-RMSE	26.81007512	26.7994592	26.69373412	26.80971558	27.05584463

调整 factor 值 对不同的 factor 值进行实现，其中 factor 值为矩阵分解中的推荐系统打分过程中的隐因子个数，隐因子个数越多，应该对于预测更准确，但是由于分解的 P、Q 矩阵维度的增加，会给训练过程的计算量带来指数级上升，所以实验中最多选用的 factor 值为 100，具体结果如下：

factor	5	10	15	20	30	50	100
10epoch-RMSE	26.59485623	26.10428933	25.97415088	25.89201254	25.78323989	25.6763966	25.55287891

注意，其中当 factor 值逐渐增大时，训练过程中（10 个 epoch）会出现过拟合现象，需要在训练的每个 epoch 后进行验证集测试，保证 RMSE 值处于下降趋势，避免过拟合导致错误的结果。例如当 factor 为 15 时，第 7 轮训练时验证集 RMSE 值最低，当 factor 为 30 时，第 6 轮训练时验证集 RMSE 值最低，说明 10 轮已经过拟合。

5. 小结

最后经过对模型及参数的不断优化，SVD 模型可以达到最低为 25.55287891 的 RMSE 值，此时的参数组如下：

参数名称	选用值
lr	0.002
decay	0.5
LAMBDAUB	0.01
LAMBDAIB	0.01
LAMBDAP	0.007
LAMBDAQ	0.007
epoch	10
factor	100

六、 实验结果与分析

最终结果如下：

选用模型	最低 RMSE 值	所用时间
CF-user	46.6321	3.8h
CF-item(using item$_a ttr$)	23.2418	12h
LFM/SVD	25.5532	1h
LFM/SVD(using item$_a ttr$)	25.5218	1h

由表格可以知：CF-item 模型可以达到最低的 RMSE 值，这是因为基于物品的协同过滤方法能够更准确地捕捉到物品之间的相似性，且基于物品的协同过滤方法剔除了用户的主观不稳定因素，从而更精确地预测用户的评分。

从时间消耗的角度来看，LFM/SVD 方法用时最短，为 1 小时，其次是 CF-user 方法，用时为 3.8 小时，而 CF-item 方法用时最长，为 12 小时。基于用户的协同过滤方法计算用户之间的相似性比较快速，因为用户数量通常比较少。而基于物品的协同过滤方法需要计算物品之间的相似性，通常物品数量较多，所以时间消耗较高。LFM/SVD 方法计算分解矩阵，需要进行迭代训练，所用时间随参数量和迭代次数相关。LFM/SVD 方法利用物品属性对评分预测值进行修正，在训练时间上与基本 LFM/SVD 是一样的，且获使得 RMSE 下降了 0.0314，在预测准确率上有一定的优化。

总的来说，CF-item 方法在 RMSE 值和时间消耗上都表现较好，因为它能够更准确地利用物品之间的相似性进行预测。LFM/SVD 方法在 RMSE 值上略高于 CF-item 方法，但计算时间较短，它能够通过矩阵分解捕捉用户和物品之间的潜在关系。

CF 空间复杂度分析 数据预处理的空间消耗与 train.txt、itemAttribute.txt 大小一致，并未额外维护数据结构处理数据，故不进行理论分析。下面分析训练协同过滤模型的空间复杂度：

- **基于用户的协同过滤**：这里仅讨论 train() 时的复杂度。由于有三个 for 循环，分别遍历用户 i、用户 j 和相同物品，因此它的时间复杂度为 $O(n^3)$。由于算法在 train 的过程中始终需要维持 sim_matrix_user，它的结构为 list[dict]，sim_matrix_user[i][j] 存储的是用户 i 和用户 j 的相似度。假设共有 N 个用户，则二维数组的大小为 N*N，则该数组的空间复杂度为 $O(N^2)$。而 user_matrix 存储用户对物品的评分，索引为用户 id，里面存储 dict，形式为：[{itemid: score,...},...]。假设有 M 个物品，则该数组的空间复杂度为 $O(N*M)$。
- **基于物品的协同过滤**：同上，也是 3 个 for 循环，分别遍历物品 i、物品 j 和相同用户，因此它的时间复杂度也为 $O(n^3)$。存储训练数据时，其形式为 {user:{item:score}}，其空间复杂度为 $O(M^N)$。在这里，我们存储相似度的数组只是一个空间变量，因此不做考虑。最后使用一个数组 pred_dict，存储用户的打分，因此空间复杂度为 $O(N)$。

LFM/SVD 空间复杂度分析 数据预处理的空间消耗与 train.txt、itemAttribute.txt 大小一致，并未额外维护数据结构处理数据，故不进行理论分析。下面分析训练 LFM/SVD 模型的空间复杂度：

通过 factor 参数构建的两个矩阵大小分析模型的空间复杂度。两个矩阵维度分别为 userNum*factor 和 itemNum*factor，由于实验中使用 dict 存储稀疏矩阵，所以空间复杂度上应该占用所有用户的所有打分对象数量（设为 N）个基本单位，经测试每个 dict 的元素占用 24 字节，所以空间复杂度约为 O(N*24)。其次，分析其所用的数据类型，在实验中需要使用到 user_item_attrs 存储用户物品属性及残差，故总空间复杂度约为 O(N)=O(N*4*24)。

七、 总结

本次实验我们从简单到复杂，较为全面地实现了推荐系统的几个主流算法：基于用户的协同过滤算法、基于物品的协同过滤算法、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，锻炼了小组成员的团结能力、动手实践的工程和科研能力。我们逐步了解了数据预处理和分析在处理大数据问题中的重要作用，以及通过保存必要的参数，可以加速程序运行，节约算力。本学期的学习中我们初步具备了利用常用的大数据处理技术解决一些实际问题的能力，这为我们今后进一步的学习提供了一种重要的思维方法和工具，受益良多。