南開大學
Nankai University

计算机学院、网络空间安全学院

大数据计算及应用

Recommendation System

姓名：刘修铭、李威远、申宗尚

学号：2112492、2112338、2213924

专业：信息安全、计算机科学与技术

2024 年 6 月 15 日

目录

1 实验要求与分析

1.1 实验要求

基于给定的数据集训练模型，使其能够预测 (u,i) 对的评价分数。

1.2 问题分析

基于给定的作业要求，分成如下几个部分进行实现。

- **数据集处理**：针对给定的几个数据集，进行基本的数据统计，以了解其组织形式，进而对数据进行预处理，删除无效数据，并将其转换为算法的输入格式。
- **算法实现**：基于给定的数据集的特征，选择合适的推荐算法，并予以实现。
- **模型训练**：按照划分的数据集，在训练集上基于实现的推荐算法进行模型训练，并保存相关数据。
- **模型评估与算法分析**：使用测试集对模型进行评估，包括 RMSE，training time 和 space consumption 等指标，并基于此，进一步对选择的推荐算法进行评价，以确定是否需要对算法进行改进。

2 实验原理

2.1 推荐系统

- **介绍**推荐系统是信息过滤系统的一种，旨在预测用户对物品的评价或偏好。这些系统在各种应用中非常普遍，例如在电子商务网站、在线影视服务以及社交媒体平台中。
- **推荐系统的类型**
  - **协同过滤推荐**：通过分析用户或物品之间的相似性来进行推荐。
  - **基于内容的推荐**：推荐与用户过去喜欢的物品在内容上相似的物品。
  - **混合推荐方法**：结合多种推荐技术，以弥补各自方法的不足。
  - **深度学习推荐**：近年来，以 NeuMF 作为起点的深度学习推荐逐渐发展，从 Transformer 到大模型，深度学习推荐正以主导者的姿态引领推荐算法。
- **关键挑战**：推荐系统面临的关键挑战包括冷启动问题、可扩展性问题以及如何提供多样性和新颖性的推荐。
- **未来发展**：随着技术的进步，未来的推荐系统将更加依赖于机器学习和人工智能技术，以提高推荐的准确性和用户满意度。

2.2 基于 user 的协同过滤算法

基于用户的协同过滤算法通过分析用户之间的相似性，推荐与目标用户兴趣相似的其他用户喜欢的物品。该算法的核心思想是“物以类聚，人以群分”，即具有相似兴趣的用户往往对相似的物品表现出类似的偏好。

- **算法步骤**

  1. **计算用户相似度**：使用某种相似度度量方法（如皮尔逊相关系数或余弦相似度）计算用户之间的相似度。
  2. **选择最近邻用户**：为目标用户选择若干个最相似的邻居用户。
  3. **预测评分**：根据最近邻用户的评分预测目标用户对未评分物品的评分。
  4. **推荐物品**：向目标用户推荐预测评分最高的物品。

- **用户相似度计算**

  用户相似度的计算是基于用户协同过滤的关键。常用的方法有余弦相似度和皮尔逊相关系数。

  - **余弦相似度**

    余弦相似度用于衡量两个用户评分向量之间的角度相似性，其公式为：

    $$\text{sim}(u,v) = \frac{\sum_{i \in I_{uv}} r_{u,i} \cdot r_{v,i}}{\sqrt{\sum_{i \in I_{uv}} r_{u,i}^2} \cdot \sqrt{\sum_{i \in I_{uv}} r_{v,i}^2}} \tag{1}$$

    其中，I_{uv} 表示用户 u 和 v 共同评分的物品集合，$r_{u,i}$ 表示用户 u 对物品 i 的评分。

  - **皮尔逊相关系数**

    皮尔逊相关系数考虑了评分的均值偏差，其公式为：

    $$\text{sim}(u,v) = \frac{\sum_{i \in I_{uv}} (r_{u,i} - \bar{r}_u) \cdot (r_{v,i} - \bar{r}_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{u,i} - \bar{r}_u)^2} \cdot \sqrt{\sum_{i \in I_{uv}} (r_{v,i} - \bar{r}_v)^2}} \tag{2}$$

    其中，$\bar{r}_u$ 和 $\bar{r}_v$ 分别表示用户 u 和 v 的平均评分。

- **评分预测**

  对于目标用户 u 对物品 i 的预测评分，通常使用加权平均的方法，其公式为：

  $$\hat{r}_{u,i} = \bar{r}_u + \frac{\sum_{v \in N(u)} \text{sim}(u,v) \cdot (r_{v,i} - \bar{r}_v)}{\sum_{v \in N(u)} |\text{sim}(u,v)|} \tag{3}$$

  其中，$\hat{r}_{u,i}$ 是预测评分，$N(u)$ 是与用户 u 最相似的 k 个邻居用户集合。

2.3 基于 item 的协同过滤算法

基于物品的协同过滤算法通过分析物品之间的相似性，推荐与目标用户喜欢的物品相似的其他物品。该算法的核心思想是“物以类聚”，即相似的物品往往会受到同一群用户的喜欢。

- **算法步骤**

  1. **计算物品相似度**：使用某种相似度度量方法（如余弦相似度或皮尔逊相关系数）计算物品之间的相似度。
  2. **选择最近邻物品**：为目标物品选择若干个最相似的邻居物品。
  3. **预测评分**：根据最近邻物品的评分预测目标用户对未评分物品的评分。

4. **推荐物品**：向目标用户推荐预测评分最高的物品。

- **物品相似度计算**

  物品相似度的计算是基于物品协同过滤的关键。常用的方法有余弦相似度和皮尔逊相关系数。

  - **余弦相似度**

    余弦相似度用于衡量两个物品评分向量之间的角度相似性，其公式为：

$$\text{sim}(i,j) = \frac{\sum_{u \in U_{ij}} r_{u,i} \cdot r_{u,j}}{\sqrt{\sum_{u \in U_{ij}} r_{u,i}^2} \cdot \sqrt{\sum_{u \in U_{ij}} r_{u,j}^2}} \tag{4}$$

    其中，U_{ij} 表示对物品 i 和 j 都评分的用户集合，$r_{u,i}$ 表示用户 u 对物品 i 的评分。

  - **皮尔逊相关系数**

    皮尔逊相关系数考虑了评分的均值偏差，其公式为：

$$\text{sim}(i,j) = \frac{\sum_{u \in U_{ij}} (r_{u,i} - \bar{r}_i) \cdot (r_{u,j} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{u,i} - \bar{r}_i)^2} \cdot \sqrt{\sum_{u \in U_{ij}} (r_{u,j} - \bar{r}_j)^2}} \tag{5}$$

    其中，$\bar{r}_i$ 和 $\bar{r}_j$ 分别表示物品 i 和 j 的平均评分。

- **评分预测**

  对于目标用户 u 对物品 i 的预测评分，通常使用加权平均的方法，其公式为：

$$\hat{r}_{u,i} = \frac{\sum_{j \in N(i)} \text{sim}(i,j) \cdot r_{u,j}}{\sum_{j \in N(i)} |\text{sim}(i,j)|} \tag{6}$$

  其中，$\hat{r}_{u,i}$ 是预测评分，$N(i)$ 是与物品 i 最相似的 k 个邻居物品集合。

2.4 FunkSVD（LFM）算法及其改进

2.4.1 SVD

奇异值分解 (SVD) 是一种矩阵分解技术，核心思想是将一个矩阵分解为三个部分：左奇异矩阵、奇异值矩阵和右奇异矩阵。

在推荐系统中，SVD 模型通常处理的是用户-物品评分矩阵 R，其中每一行代表一个用户，每一列代表一个物品，矩阵中的元素 R_{ij} 是用户 i 对物品 j 的评分。SVD 将矩阵 R 分解为三个矩阵的乘积：$R \approx U\Sigma V^T$。其中：

- U 是一个用户矩阵，它的每一行可以视为用户在潜在特征空间的表示。
- Σ 是对角矩阵，其对角线上的元素称为奇异值，代表了数据中每个潜在特征的强度或重要性。
- V^T 是物品矩阵的转置，其每一列代表物品在相同的潜在特征空间中的表示。

其主要应用如下：

- **缺失值预测**：通过计算 $\hat{R} = U\Sigma V^T$，SVD 能够预测矩阵 R 中的缺失值，即用户未评分的项。

- **降维**：在实际应用中，通常只选择最大的几个奇异值和相应的向量来近似原始矩阵，这样不仅可以大大减少数据的存储和计算量，还可以帮助去除噪声和冗余信息，提升模型的泛化能力。
- **个性化推荐**：利用分解后的矩阵，为每个用户生成个性化的推荐列表。通过比较不同用户和物品在潜在特征空间的相似度，可以识别出用户可能感兴趣的物品。

2.4.2 FunkSVD

FunkSVD 就是在 SVD 的技术上优化“数据稠密”+“计算复杂度高”+“只可以用来数据降维”难题的。一个矩阵做 SVD 分解成 3 个矩阵很耗时，同时还面临稀疏的问题，那么解决稀疏问题，同时只分解成两个矩阵呢？期望矩阵 M 这样进行分解成以下这个形式：

$$M_{m\times n} = P^T_{m\times k} Q_{k\times n}$$

对于如何将矩阵 M 分解两个特征维度为 P 和 Q 再继续优化计算，主要目标是让用户的评分和用矩阵乘积得到的评分残差尽可能的小，业界的主要思路就是用均方差作为损失函数，来学习出的 P 和 Q 的参数。线性回归思维理论的好处是优化策略完善，梯度下降，理论知识成熟，计算难度低。

对于某一个用户评分，采用 FunkSVD 进行矩阵分解，则拟合函数表示为 $q_i^T p_i$，采用均方差做为损失函数，则期望尽可能的小（损失函数极小值），如果考虑所有的物品和样本的组合，则期望最小化下式:

$$\sum_{i,j}(m_{i,j} - q_j^T p_i)^2$$

只要能够最小化损失上面的式子，并求出极值所对应的，就可以得到矩阵 P 和 Q，那么对于任意矩阵 M 任意一个空白评分的位置，可以通过计算 $q_j^T p_i$ 预测评分。在实际应用中，为了防止过拟合，会加入一个 L2 的正则化项，修改处理后的 FunkSVD 的优化目标函数 $J(p,q)$ 是这样的：

$$\text{argmin}_{p_i,q_j}\left[\sum_{i,j}(m_{i,j} - q_j^T p_i)^2 + \lambda(\|p_i\|_2^2 + \|q_j\|_2^2)\right]$$

其中，λ 为正则化系数。

使用梯度下降法进行优化时，其迭代公式为：

$$p_i = p_i + \alpha((m_{ij} - q_j^T p_i)q_j - \lambda p_i)$$

$$q_j = q_j + \alpha((m_{ij} - q_j^T p_i)p_i - \lambda q_j)$$

2.4.3 BiasSVD

针对于 FunkSVD 算法，而且多半是以这种社交媒体或者网站平台为主，久而久之会导致了很多商业问题和难点，很难做到合理化、公平化，因为有人的参与是最难的，经过了各种网站实时反馈，有下面几个问题：

- 人员问题（或者水军问题），就是参与人员的问题，比如有的人就是比较好说话，他对每一部看过的电影或者物品，都觉得很好，都给了很高的评分；也有一部分人艺术审美很高，比较严格，他对绝大部分都给了比大众水平低的分，当然这两部分人都有可能是水军，都是利益关联方，但是他们的数据是不准确的，不能反映真实大众的想法，怎么降低这部分误差成了关键。

- 作品问题，如说山寨电影，比如山寨物品，这部分本身就具有问题，而靠贴流量贴标签贴热点等状态，聚集了很多的评分，也反映不出真实大众的水平，所以降低这部分物品的误差也是关键。
- 全局问题，简单以电影为例，也许赶上了某一个时间段的大众审美热度，大家都去拍这个题材，而这个题材分数整体偏高了，如何降低这种全局偏差也成了关键。

针对于此，机器学习最常用的优化就是“正则化”+“偏置项”，正则化是为了防止整个模型的过分拟合，偏置项是既然知道参与的部分数据本身具有误差，那就人工手动提前去在损失函数里增加这部分偏差考虑，在计算求解中求出这个参数，然后应用在新的数据里就应该是更符合大众，更准确地数据了。

假设评分系统平均分为 u，第 i 个用户的用户偏置项为 b_i，而第 j 个物品的物品偏置项为 b_j，则加入了偏置项以后的优化目标函数 $J(p,q)$ 为：

$$\text{argmin}_{p_i,q_j}\left[\sum_{i,j}(m_{i,j}-\mu-b_i-b_j-q_j^Tp_i)^2+\lambda(\|p_i\|_2^2+\|q_j\|_2^2+\|b_i\|_2^2+\|b_j\|_2^2)\right]$$

2.5 （提高实现）GBDT+LR ——借助机器学习实现属性特征融合

那么，如何引入我们的属性特征呢？我们需要跳出协同过滤和矩阵分解的思想框架，寻找可以利用更多特征的模型，一个显然的思路是利用 LR 直接将属性的特征加权到结果中，但是加权的权重需要 LR 进一步的学习，隐特征之间的 LR 的处理显然比较困难，有没有自己完成特征交叉的处理的模型呢？

显然，借助集成学习的经典特征交叉方法 GBDT+LR 能够完成我们的任务。

GBDT 的思想使其具有天然优势，可以发现多种有区分性的特征及特征组合。决策树的路径可以直接作为 LR 输入特征使用，省去了人工寻找特征、特征组合的步骤。这种方法结合了 GBDT 在处理非线性关系方面的强大能力，以及 LR 在处理稀疏数据时的高效性。业界已有实践（Facebook、Kaggle 等）使用这种方式取得了不错的效果。

GBDT+LR 模型融合思想来源于 Facebook 公开的论文 Practical Lessons from Predicting Clicks on Ads at Facebook，其主要思想是 GBDT 每棵树的路径直接作为 LR 的输入特征使用，即用已有特征训练 GBDT 模型，然后利用 GBDT 模型学习到的树来构造新特征，最后把这些新特征加入原有特征一起训练模型。

- GBDT 首先对原始训练数据做训练，得到一个二分类器，当然这里也需要利用网格搜索寻找最佳参数组合。
- 与通常做法不同的是，当 GBDT 训练好做预测的时候，输出的并不是最终的二分类概率值，而是要把模型中的每棵树计算得到的预测概率值所属的叶子结点位置记为 1，这样，就构造出了新的训练数据。

举个例子，下图是一个 GBDT+LR 模型结构，设 GBDT 有两个弱分类器，分别以蓝色和红色部分表示，其中蓝色弱分类器的叶子结点个数为 3，红色弱分类器的叶子结点个数为 2，并且蓝色弱分类器中对 0-1 的预测结果落到了第二个叶子结点上，红色弱分类器中对 0-1 的预测结果也落到了第二个叶子结点上。那么我们就记蓝色弱分类器的预测结果为 [0 1 0]，红色弱分类器的预测结果为 [0 1]，综合起来看，GBDT 的输出为这些弱分类器的组合 [0 1 0 0 1] ，或者一个稀疏向量（数组）。

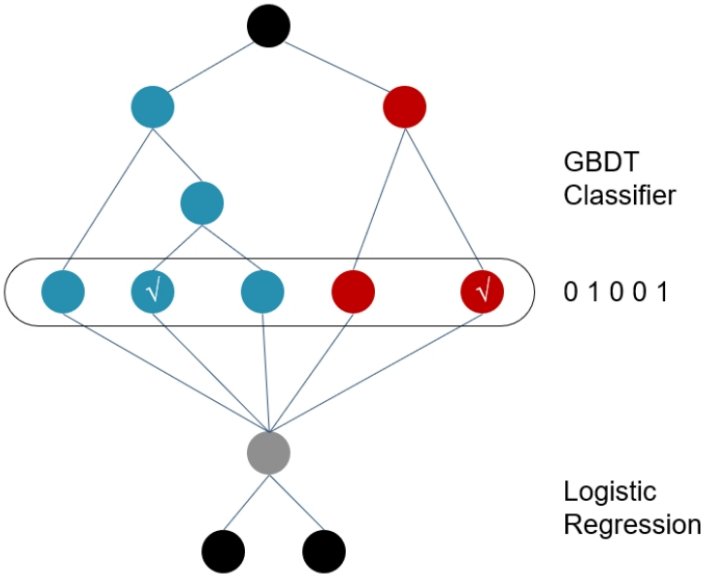

图 2.1: GBDT+LR example

这里的思想与 One-hot 独热编码类似，事实上，在用 GBDT 构造新的训练数据时，采用的也正是 One-hot 方法。并且由于每一弱分类器有且只有一个叶子节点输出预测结果，所以在一个具有 n 个弱分类器、共计 m 个叶子结点的 GBDT 中，每一条训练数据都会被转换为 1*m 维稀疏向量，且有 n 个元素为 1，其余 m-n 个元素全为 0。

- 新的训练数据构造完成后，下一步就要与原始的训练数据中的 label(输出) 数据一并输入到 Logistic Regression 分类器中进行最终分类器的训练。思考一下，在对原始数据进行 GBDT 提取为新的数据这一操作之后，数据不仅变得稀疏，而且由于弱分类器个数，叶子结点个数的影响，可能会导致新的训练数据特征维度过大的问题，因此，在 Logistic Regression 这一层中，可使用正则化来减少过拟合的风险，在 Facebook 的论文中采用的是 L1 正则化。

2.6 （提高实现）NeuralCF ——以神经网络的视角审视矩阵分解

这部分的内容参考了何向南老师团队最早的 NerualCF 的论文：《Neural collaborative filtering》。除了 CF 以外的代码实现都和这篇论文的思想相关，**这里的实验原理是我们组的核心思想，不论是引入 attribute 还是改进 RMSE，引入深度学习（包含图神经网络）是我们的思想核心**

重新思考我们的 SVD，即矩阵分解算法的实现，熟悉深度学习的我们其实能够发现，**SVD 本质是上一个点积输出的单层感知机**：

由于输入层是用户（项目）ID 中的一个 $one-hotencoding$ 编码，所获得的嵌入向量可以被看作是用户（项目）的潜在向量。

我们用 $P^T v_u^U$ 表示用户的潜在向量 p_u,$Q^T v_i^I$ 表示项目的潜在向量 q_i , 我们定义第一层神经 CF 层的映射函数为：

$$\phi(p_u, q_i) = p_u \odot q_i$$

其中 $\odot$ 表示向量的逐元素乘积。然后，我们将向量映射到输出层：

$$\hat{y}_{ui} = a_{out}(h^T(p_u \odot q_i))$$

其中 a_{out} 和 h 分别表示输出层的激活函数和连接权。

直观地讲，如果我们将 a_{out} 看做一个恒等函数，h 权重全为 1，显然这就是我们的 MF 模型。

更进一步的，如果我们允许从没有一致性约束（uniform constraint）的数据中学习 h，则会形成 MF 的变体，它允许潜在维度的不同重要性出现。如果我们用一个非线性函数 a_{out} 将进一步推广 MF 到非线性集合，使得模型比线性 MF 模型更具有表现力。在 NCF 下实现一个更一般化的 MF，它使用 Sigmoid 函数 $\sigma(x) = 1/1(1 + e^{-x})$ 作为激活函数，通过 $logloss$ 学习 h。称为 GMF（Generalized Matrix Factorization，广义矩阵分解）。

不难看出，将 MF 引申到深度学习领域后，我们能够允许神经网络对协同过滤进行更完整的处理，也就可以完整的处理我们的属性特征：

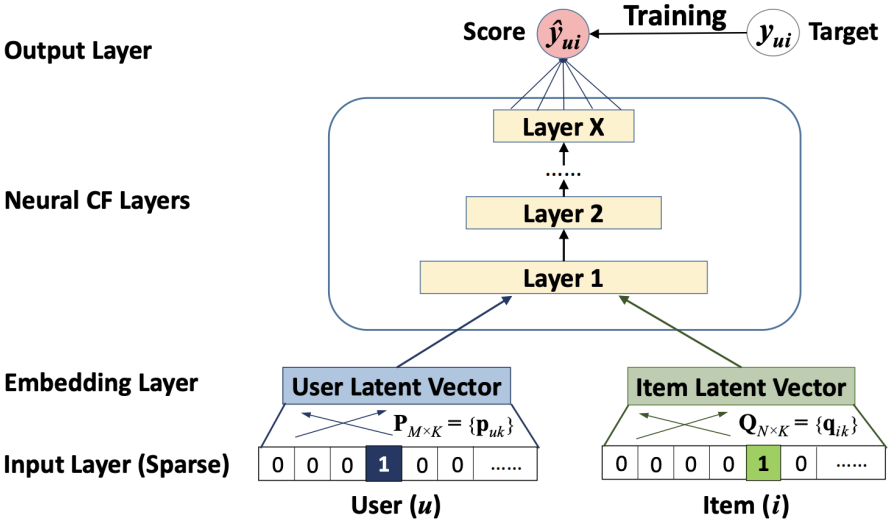

图 2.2: MF 的深度学习扩展

- **模型输入层：** 包括两个特征向量 v_u^U 和 v_i^I ，分别用来描述用户 u 和项目 i。他们可以进行定制，用以支持广泛的用户和项目的建模，例如上下文感知，基于内容（**这样我们就能更好地引入我们的属性特征了！**），和基于邻居的构建方式。

- **模型中间层：** 嵌入层 $EmbeddingLayer$；一个全连接层，用来将输入层的稀疏特征向量映射为一个稠密向量 $densevector$ 。所获得的用户（项目）的嵌入（就是一个稠密向量）可以被看作是在潜在因素模型的上下文中用于描述用户（项目）的潜在向量。

- **模型处理层：** 然后我们可以将用户嵌入和项目嵌入送入多层神经网络中，我们将它称为神经网络协同过滤层，它将潜在向量映射为预测分数。这里每一层都可以被定制，用以发现用户-项目交互的某些潜在结构。最后一个隐藏层 X 的维度尺寸决定了模型的能力。最终输出层是预测分数 $\hat{y}_{ui}$，通过最小化预测值 $\hat{y}_{ui}$ 和其目标值 y_{ui} 之间逐点损失进行训练。

基于此的思路，我们就可以给出论文中实现的最终目标：NeuralCF

NeuralCF 主要有 GMF、MLP 及 NeuMF 三种实现方式，区别在于 NeuralCF 通用框架中的 Neural CF Layers 不同。由于 NeuMF 模型的 Neural CF Layers 设计结合了 GMF 和 MLP，因此本次实验中主要使用 NeuMF 实现。

如图是 NeuMF 的模型结构。

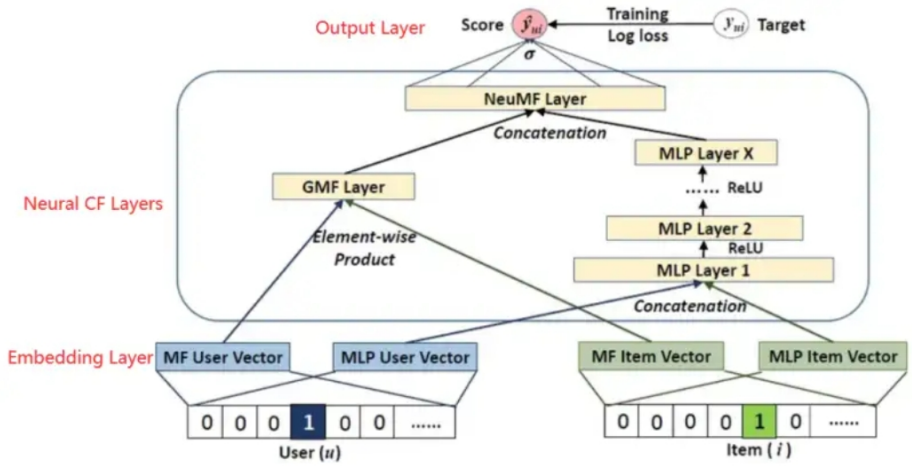

图 2.3: NeuMF

如上图所示，GMF，应用了一个线性内核来模拟潜在的特征交互；MLP，使用非线性内核从数据中学习交互函数。

接下来的问题是：我们如何能够在 NCF 框架下融合 GMF 和 MLP，使他们能够相互强化，以更好地对复杂的用户-项目交互建模？一个直接的解决方法是让 GMF 和 MLP 共享相同的嵌入层（Embedding Layer），然后再结合它们分别对相互作用的函数输出。

这种方式和著名的神经网络张量（NTN,Neural Tensor Network）有点相似。然而，共享 GMF 和 MLP 的嵌入层可能会限制融合模型的性能。例如，它意味着，GMF 和 MLP 必须使用的大小相同的嵌入；对于数据集，两个模型的最佳嵌入尺寸差异很大，使得这种解决方案可能无法获得最佳的组合。为了使得融合模型具有更大的灵活性，我们允许 GMF 和 MLP 学习独立的嵌入，并结合两种模型通过连接他们最后的隐层输出。

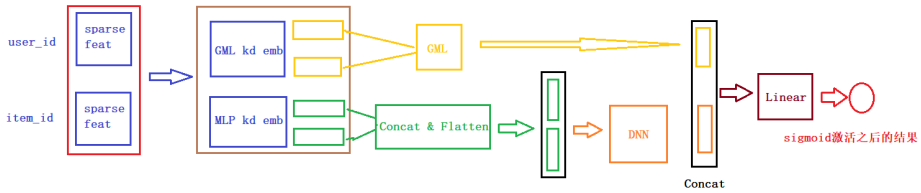

图 2.4: NeuMF 输入流程展示

此外，原文作者认为：在最优调参的情形中，GMF 和 MLP 的嵌入维数不一定相同，甚至相差很大。由此，在 NeuMF 模型训练的过程中，需要分别学习 GMF 和 MLP 的嵌入层参数。更进一步，GMF 通过逐元素相乘的方式，合并用户和物品的嵌入表示，要求用户和物品嵌入维数相同。MLP 则直接将用户和物品嵌入表示拼接起来，不要求用户和物品的嵌入维数相同。此后，将分别经过 GMF Layer 和 MLP 处理的不同输出向量拼接起来，投入到最后的 NeuMF layer 输出层中，即可完成预测。

NeuMF 模型不仅可以将表示某个特定用户或物品的独热编码作为原始输入，还可以通过重构某用户或物品原始输入向量的方式，额外考虑上下文，从而捕捉更丰富的特征信息和隐藏关系。比如，在表示某用户的原始输入向量尾部，额外拼接一个考虑用户年龄或偏好等属性的特征向量。

2.7 （提高实现）NGCF&LightGCN ——引入关联特征的图神经网络推荐

LightGCN 是将图卷积神经网络应用于推荐系统当中，是对神经图协同过滤 NGCF 算法（该算法也由先前提出 NCF 的团队后续提出）的优化和改进。

协同过滤的基本假设是相似的用户会对物品展现出相似的偏好，自从全面进入深度学习领域之后，一般主要是先在隐空间中学习关于 user 和 item 的 embedding，然后重建两者的交互即 interaction modeling，如 MF 做内积，NCF 模拟高阶交互等。但是他们并没有把 user 和 item 的交互信息本身编码进 embedding 中，这就是 NGCF 想解决的点：显式建模 User-Item 之间的高阶连接性来提升 embedding。不同于 NGCF，lightGCN 将 GCN 中最常见的两种设计：特征转换和非线性激活弃用，因为他们对模型并无实质性作用，另外 LightGCN 认为自信息的作用不大，也没有使用自信息链接。

其模型结构如图所示：

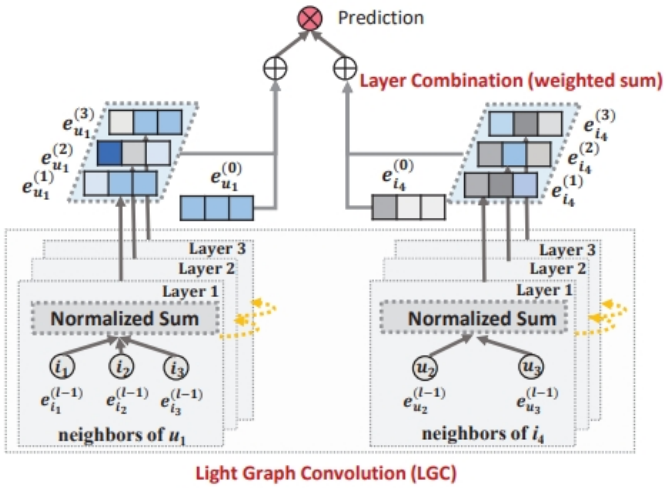

图 2.5: LightGCN 原理图

LightGCN 去除了特征转换和非线性激活函数，因为这些元素在提升性能方面没有显著效果，反而增加了训练的复杂性。LightGCN 通过在用户-项目交互图上进行线性传播，学习用户和项目在不同层次上的嵌入表征，最终通过加权求和这些嵌入表征来完成用户和项目的最终表示。

其核心部分如下：

- **邻域聚合**：LightGCN 在每个图卷积层中使用加权求和聚合器，完成用户和项目节点表示的邻域聚合。
- **组合嵌入**：通过加权和不同图卷积层生成的嵌入表示，获得用户和项目的最终嵌入表示。
- **预测输出**：使用用户和项目的最终嵌入向量的内积来预测用户和项目间的相关性分数。

LightGCN 使用 BPR 作为损失函数，并采用 Adam 优化器进行迭代训练。模型的可训练参数仅包括用户和项目的初始嵌入表示，使得其复杂度与矩阵分解相当。此外，LightGCN 不需要引入 dropout，因为嵌入的 L2 正则化处理已足以应对过拟合问题。

3 数据集基本情况

3.1 数据集介绍

参考作业要求文档，给出四个数据集的介绍。

- **Train.txt**：包含了用户对物品的评分数据，通常用于训练推荐系统模型。每条数据包括用户 ID、物品 ID 和相应的评分。这些数据反映了用户的偏好，是推荐系统学习如何预测未来用户行为的基础。
- **Test.txt**：用于评估训练好的模型的性能。这个数据集类似于训练集，但不包含用户的评分，目标是预测这些未知的评分。通过比较预测和实际用户行为，可以评估模型的准确性和泛化能力。
- **ItemAttribute.txt**：包含了额外的关于物品的信息，比如描述、类别、制造商等属性。这些属性可以用来增强推荐系统，尤其是在处理冷启动问题（即如何推荐新用户或新物品）时非常有用。利用这些属性，可以通过基于内容的推荐方法，或者作为混合推荐系统中的一部分来改进推荐的相关性和准确性。
- **ResultForm.txt**：定义了需要提交的预测结果的格式。

3.2 数据集统计信息

统计分析可以帮助了解数据集的基本结构，例如用户数量、物品数量、评分数量等，能帮助选择合适的推荐算法和评估指标。同时，统计分析能揭示数据中可能存在的问题，如缺失值、异常值或数据不一致性等，对于清洗数据、进行预处理非常关键，这可以显著影响推荐系统的效果。

```python
import statistics
from collections import defaultdict

def read_data(file_path):
    user_ratings = defaultdict(list)
    item_ratings = defaultdict(list)
    max_user_id = 0
    max_item_id = 0

    try:
        with open(file_path, 'r') as f:
            while True:
                user_line = f.readline()
                if not user_line:
                    break
                user_id, num_ratings = map(int, user_line.strip().split('|'))
                max_user_id = max(max_user_id, user_id)
                for _ in range(num_ratings):
                    item_line = f.readline()
                    item_id, rating = map(float, item_line.strip().split())
                    max_item_id = max(max_item_id, item_id)
                    user_ratings[user_id].append(rating)
                    item_ratings[item_id].append(user_id)
```

```
24
25          return user_ratings, item_ratings, max_user_id, max_item_id
26      except FileNotFoundError:
27          print("File not found. Please check the file path.")
28          return {}, {}, 0, 0
29      except Exception as e:
30          print(f"An error occurred: {e}")
31          return {}, {}, 0, 0
32
33  def calculate_statistics(user_ratings, item_ratings):
34      num_users = len(user_ratings)
35      num_items = len(item_ratings)
36      total_ratings = sum(len(ratings) for ratings in user_ratings.values())
37      average_rating = statistics.mean(rating for ratings in user_ratings.values() for
            rating in ratings)
38
39      return num_users, num_items, total_ratings, average_rating
```

基于上述代码，得到如下的统计信息。

- Number of users: 19835
- Number of items rated: 455691
- Total number of ratings: 5001507
- Average rating: 49.50627100991761
- Max user ID: 19834
- Max item ID: 624960

由上述数据，可以分析得到下面的结论。

- 总共有 5001507 个评分，说明每个用户平均对大约 252 个物品进行了评分，表明用户通常对多个物品有评价，有利于通过用户的评分行为学习用户的偏好。
- 最大物品 ID 为 624960，远大于物品的实际数量 455691，表明物品 ID 不是连续的，有些物品 ID 没有对应的评分记录或者中间有一些物品被删除，需要考虑冷启动问题。

而对于 itemAttribute 中的内容，使用下面代码进行统计性分析：

```
1  # 读取文件并处理数据
2  def read_attribute_data(filepath):
3      data = []
4      with open(filepath, 'r') as f:
5          for line in f:
6              parts = line.strip().split('|')
7              item_id = int(parts[0])
8              attr1 = int(parts[1]) if parts[1] != 'None' else -1
9              attr2 = int(parts[2]) if parts[2] != 'None' else -1
```

```
            data.append((item_id, attr1, attr2))
    return data

# 统计属性类型出现次数
def count_attributes(data):
    attribute_count = defaultdict(int)
    for item_id, attr1, attr2 in data:
        if attr1 is not None:
            attribute_count[attr1] += 1
        if attr2 is not None:
            attribute_count[attr2] += 1
    return attribute_count

# 计算总共出现多少个属性类型、平均每个属性出现多少次、最少出现和最多出现的属性类型次数
def analyze_attributes(attribute_count):
    total_attributes = len(attribute_count)
    total_occurrences = sum(attribute_count.values())
    if total_attributes > 0:
        avg_occurrences = total_occurrences / total_attributes
    else:
        avg_occurrences = 0
    min_occurrences = min(attribute_count.values(), default=0)
    max_occurrences = max(attribute_count.values(), default=0)
    return total_attributes, avg_occurrences, min_occurrences, max_occurrences
```

得到如下的数据:

- There are a total of 71879 attribute types
- n average, each attribute appears 14.11 times
- The least frequent attribute type appeared 1 times
- The most frequent attribute type appeared 105725 times

可以看到，其中共计有 71879 种不同的属性，但是其平均出现次数为 14.11 次，表明此部分数据存在较高的稀疏性。少数属性出现次数极高，甚至达到 105725 次。在训练时，模型可能会高度依赖此部分数据，而忽视出现频率较少的属性，后续训练时需要注意。

4 推荐算法实现

4.1 基于用户的协同过滤

基于用户的协同过滤算法通过分析用户之间的相似性，推荐与目标用户兴趣相似的其他用户喜欢的物品。该算法的核心思想是“人以群分”，即具有相似兴趣的用户往往对相似的物品表现出类似的偏好。

算法步骤

1. **计算用户相似度**：使用余弦相似度计算用户之间的相似度。
2. **选择最近邻用户**：为目标用户选择若干个最相似的邻居用户。
3. **预测评分**：根据最近邻用户的评分预测目标用户对未评分物品的评分。
4. **推荐物品**：向目标用户推荐预测评分最高的物品。

数据读取与预处理 为了进行推荐，首先需要读取并预处理数据。训练数据和测试数据分别存储在指定的文件路径中。训练数据文件包含用户的评分信息，而物品数据文件则包含物品的属性信息。训练数据被读取后，为了防止划分训练集及测试集时，有的用户信息全部被划入测试集，阻碍训练效果，这里我们在数据集中每一位用户中随机选择一条物品数据加入测试集，其他作为训练集，用训练集训练评分矩阵，随后用测试集计算 RMSE。

```
def read_and_split_train_data(filepath):
    train_data = defaultdict(list)
    with open(filepath, 'r') as f:
        for line in f:
            user_id, num_ratings = line.strip().split('|')
            ratings = []
            for _ in range(int(num_ratings)):
                item_id, score = f.readline().strip().split()
                ratings.append((item_id, float(score)))
            train_data[int(user_id)] = ratings

    train_split = defaultdict(list)
    test_split = defaultdict(list)

    for user, ratings in train_data.items():
        if len(ratings) > 1:
            random.shuffle(ratings)
            test_split[user].append(ratings.pop())  # 随机选择一条数据作为测试集
            train_split[user] = ratings  # 剩余的数据作为训练集
        else:
            train_split[user] = ratings  # 如果用户只有一条数据，则全部作为训练集

    return train_data, train_split, test_split
```

构建评分矩阵 在读取和预处理数据之后，下一步是构建用户-物品评分矩阵。该矩阵的每一行代表一个用户，每一列代表一个物品，矩阵中的值表示用户对物品的评分。为了构建这个矩阵，我们首先需要对用户和物品进行编号映射。然后，利用训练数据中的评分信息填充矩阵。

```
def build_rating_matrix(train_data):
    users = list(train_data.keys())
    items = list({item for ratings in train_data.values() for item, _ in ratings})
    user_map = {user: i for i, user in enumerate(users)}
    item_map = {item: i for i, item in enumerate(items)}
    # 构建稀疏矩阵
```

```
    rating_matrix = csr_matrix((data, (row, col)), shape=(len(users), len(items)))
    return rating_matrix, user_map, item_map
```

计算用户相似度 评分矩阵构建完成后，需要计算用户之间的相似度。此处采用余弦相似度进行评估，用于衡量两个向量（在这里是两个用户的评分向量）之间的相似程度。同时，为了提高计算效率，采用批处理的方式进行相似度计算。

```
def calculate_user_similarity(rating_matrix, batch_size=5000):
    num_users = rating_matrix.shape[0]
    similarity_matrix = np.zeros((num_users, num_users))
    for start in tqdm(range(0, num_users, batch_size), desc="Calculating similarity"):
        end = min(start + batch_size, num_users)
        batch_similarity = cosine_similarity(rating_matrix[start:end], rating_matrix)
        similarity_matrix[start:end] = batch_similarity
    return similarity_matrix
```

预测用户评分 对于目标用户，找到与其相似度最高的若干个用户，然后使用这些用户的评分通过加权平均的方法预测目标用户的评分。

```
def predict_user_ratings(similarity_matrix, rating_matrix, user_map, item_map,
    test_data, top_k=10):
    predictions = defaultdict(list)
    for user, items in test_data.items():
        if user not in user_map:
            continue
        user_idx = user_map[user]
        similar_users = np.argsort(similarity_matrix[user_idx])[::-1][1:top_k + 1]
        for item in items:
            if item not in item_map:
                predictions[user].append((item, 0))
                continue
            item_idx = item_map[item]
            sim_scores = rating_matrix[similar_users, item_idx].toarray().flatten()
            weights = similarity_matrix[user_idx, similar_users]
            valid_mask = (sim_scores != 0) & (weights != 0)
            if np.any(valid_mask):
                predicted_score = np.dot(sim_scores[valid_mask], weights[valid_mask])
                    / np.sum(weights[valid_mask])
            else:
                predicted_score = 0
            predicted_score = round(predicted_score, 4)
            predictions[user].append((item, predicted_score))
    return predictions
```

在这一部分，原本的算法需要 10 个小时的长时间预测，我们进行了大量优化，使其时间缩短到一分钟内，主要进行了矢量化操作和预测的剪枝。**矢量化操作**：在我们的代码中，直接利用 NumPy 的矢量化操作一次性获取相似用户对某个物品的所有评分，并进行加权计算。矢量化操作可以充分利用底层的

高效计算库，大大减少循环的开销。**剪枝**: 同时，在优化后的代码中，对于不在 user_map 和 item_map 中的用户和物品，提前跳过了这些计算。从而减少了不必要的计算开销，大大减少了运行时间：

推荐物品 基于预测的评分，可以向目标用户推荐物品。推荐的物品是那些预测评分最高的物品。为了保证推荐结果的准确性和多样性，在此只推荐那些目标用户未评分的物品。

```
def write_predictions(predictions, filepath):
    with open(filepath, 'w') as f:
        for user, items in predictions.items():
            f.write(f"{user}|{len(items)}\n")
            for item, score in items:
                f.write(f"{item} {score}\n")
```

模型的保存与加载 为了便于后续的使用和验证，将计算好的相似度矩阵、评分矩阵、用户映射和物品映射等模型数据保存到文件中。在需要时，可以通过加载这些文件来快速恢复模型，而不需要重新进行计算。

```
def save_model(model, matrix, user_map, item_map, filepath):
    with open(filepath, 'wb') as f:
        pickle.dump((model, matrix, user_map, item_map), f)

def load_model(filepath):
    with open(filepath, 'rb') as f:
        model, rating_matrix, user_map, item_map = pickle.load(f)
    return model, rating_matrix, user_map, item_map
```

验证与测试 在模型训练完成后，对其进行验证和测试。验证集用于评估模型的性能并进行必要的调整，而测试集则用于最终评估模型的效果。通过对验证集和测试集中的用户进行评分预测，并与实际评分进行对比，从而计算模型的预测精度。

```
train_data, train_split, val_data = read_and_split_train_data(train_data_path)
rating_matrix, user_map, item_map = build_rating_matrix(train_data)
similarity_matrix = calculate_user_similarity(rating_matrix)
model_path = '../model/model_userCF.pkl'
save_model(similarity_matrix, rating_matrix, user_map, item_map, model_path)
model, rating_matrix, user_map, item_map = load_model(model_path)
val_data, val_real = val_data_change(val_data)
predictions = predict_user_ratings(model, rating_matrix, user_map, item_map,
    val_data, top_k=500)
val_result_path = './result/validation_userCF.txt'
write_validation_predictions(predictions, val_real, val_result_path)
test_data_path = '../data/test.txt'
result_path = './result/result_userCF.txt'
test_data = read_test_data(test_data_path)
predictions = predict_user_ratings(model, rating_matrix, user_map, item_map,
    test_data, top_k=500)
write_predictions(predictions, result_path)
```

定义了一个函数用于计算推荐系统中预测评分与实际评分之间的均方根误差（RMSE）。

```python
def calculate_rmse(predictions, targets):
    if len(predictions) != len(targets):
        raise ValueError("Length of predictions and targets must be the same.")

    n = len(predictions)
    rmse = math.sqrt(sum((predictions[i] - targets[i]) ** 2 for i in range(n)) / n)
    return rmse
```

4.2 基于物品的协同过滤

算法步骤

1. **计算物品相似度**：使用余弦相似度计算目标项目与用户已评分项目的相似度。
2. **选择最近邻物品**：为目标物品选择若干个最相似的邻居物品。
3. **预测评分**：根据最近邻物品的评分预测目标用户对未评分物品的评分。
4. **推荐物品**：向目标用户推荐预测评分最高的物品。

数据读取与预处理 将每个用户的评分记录读取并存储在 defaultdict 中，以用户 ID 为键，值为项目 ID 和评分的列表。测试集数据读取同训练集，在此不再赘述。

```python
def read_train_data(filepath):
    train_data = defaultdict(list)
    with open(filepath, 'r') as f:
        for line in f:
            user_id, num_ratings = line.strip().split('|')
            num_ratings = int(num_ratings)
            for _ in range(num_ratings):
                item_id, score = next(f).strip().split()
                train_data[int(user_id)].append((item_id, float(score)))
    return train_data
```

构建评分矩阵 从训练数据中构建一个评分矩阵，每一行对应一个用户，每一列对应一个物品。由于很多用户可能只对很少的项目进行评分，这导致评分矩阵中会有很多零值。使用稀疏矩阵（如 CSC 格式）可以有效地只存储非零元素，从而大幅节约内存和提高数据处理速度。

```python
def build_rating_matrix(train_data):
    users = list(train_data.keys())
    items = list(set(item for ratings in train_data.values() for item, _ in ratings))
    user_map = {user: i for i, user in enumerate(users)}
    item_map = {item: i for i, item in enumerate(items)}

    data, row, col = [], [], []
```

```
    for user, ratings in train_data.items():
        for item, score in ratings:
            data.append(score)
            row.append(user_map[user])
            col.append(item_map[item])

    rating_matrix = csc_matrix((data, (row, col)), shape=(len(users), len(items)))
    return rating_matrix, user_map, item_map
```

相似度计算 在基于项目的协同过滤推荐系统中，选择直接在推荐过程中计算目标项目与用户已评分项目的相似度，而不是预先计算并存储所有项目之间的相似度，有如下几个考虑：

1. 如果事先计算所有项目之间的相似度，对于一个包含大量项目的数据集来说，相似度矩阵可能会非常巨大，需要消耗大量的存储空间。
2. 在实际应用中，很多项目之间可能没有足够的共同评分者，导致相似度计算基础薄弱或相似度为零，使得预存的相似度矩阵中会有大量的零值，这种存储效率低下。
3. 在动态变化的数据集中，如新项目和新评分持续加入，相似度矩阵需要频繁更新以反映最新数据。但是新数据只影响部分计算，不需要重新计算整个相似度矩阵。

```
def calculate_similarity(target_item_id, user_id, user_purchases, rating_matrix,
    item_map, user_map):
    if target_item_id not in item_map:
        return 0.0

    target_index = item_map[target_item_id]
    similarities = []

    target_vector = rating_matrix[:, target_index].toarray().T
    for purchase in user_purchases:
        if purchase in item_map:
            purchase_index = item_map[purchase]
            purchase_vector = rating_matrix[:, purchase_index].toarray().T
            sim = cosine_similarity(purchase_vector, target_vector)[0, 0]
            similarities.append((purchase, sim))

    # 取前十高相似度的物品
    top_similar_items = heapq.nlargest(10, similarities, key=lambda x: x[1])
    weighted_sum = sum(
        score * rating_matrix[user_map[int(user_id)], item_map[item]] for item, score
            in top_similar_items)
    sim_sum = sum(score for _, score in top_similar_items)
    predicted_score = weighted_sum / sim_sum if sim_sum != 0 else 0.0
    return round(predicted_score, 4)
```

评分预测 结合训练数据和测试数据，使用用户的历史评分数据（相似度矩阵）来预测未知项目的评分，从而向用户推荐可能感兴趣的新项目。

```python
def predict_ratings(test_data, train_data, rating_matrix, user_map, item_map):
    predictions = defaultdict(list)
    for user_id, item_ids in tqdm(test_data.items(), desc="predicting"):
        user_ratings = train_data[int(user_id)]
        user_purchases = [item for item, _ in user_ratings]
        for item_id in item_ids:
            if item_id not in user_purchases:
                score = calculate_similarity(item_id, user_id, user_purchases,
                    rating_matrix, item_map, user_map)
                predictions[user_id].append((item_id, score))
    return predictions
```

其他处理 由于存储物品之间的相似度矩阵需要大量的存储空间，加之 itemCF 矩阵的稀疏性，在此暂不考虑对其模型进行保存等问题。

4.3 FunkSVD（LFM）算法及其改进

1、算法步骤

对于 FunkSVD 的代码实现，我们首先需要思考，以什么形式组织代码实现呢?

考虑到我们先前在**（提高实现）NeuralCF ——以神经网络的视角审视矩阵分解**章节的分析，我们计划使用 Pytorch 深度学习框架完成 SVD 的搭建，以便于一步步向 NeuCF 改进。

值得说明的是，SVD 采用 pytorch 和不采用 pytorch 构建，唯一的工作量差别在于梯度下降法不需要实现 (本身梯度下降就不是我们研究的重点)。

使用 pytorch，我们还进行了分批度、gpu 并行等等优化。而借助 pytorch 的最大成果是，我们还有了额外众多深度学习模型的实现工作，我们进一步实现了：引入 attr 的 SVD 深度学习模型（自行设计）、NeuCF（论文复现）、LightGCN（论文复现）

综上，我们的实现的步骤如下：

1. **数据准备**：导入并预处理数据集，包括分割训练集和测试集，标准化处理等。
2. **模型构建**：使用 Pytorch 定义 FunkSVD 模型。该模型包含用户和物品的嵌入层，利用这些嵌入层计算预测评分。
3. **损失函数和优化器**：选择合适的损失函数（如均方误差）和优化器（如 Adam）来训练模型。
4. **训练循环**：实现训练循环，包括前向传播、计算损失、反向传播和参数更新。每个 epoch 结束后可以计算模型在验证集上的性能。
5. **模型评估**：在测试集上评估模型性能，计算相关指标如 RMSE 或 MAE。
6. **模型改进**：根据模型性能和实验现象，设计对模型的改进

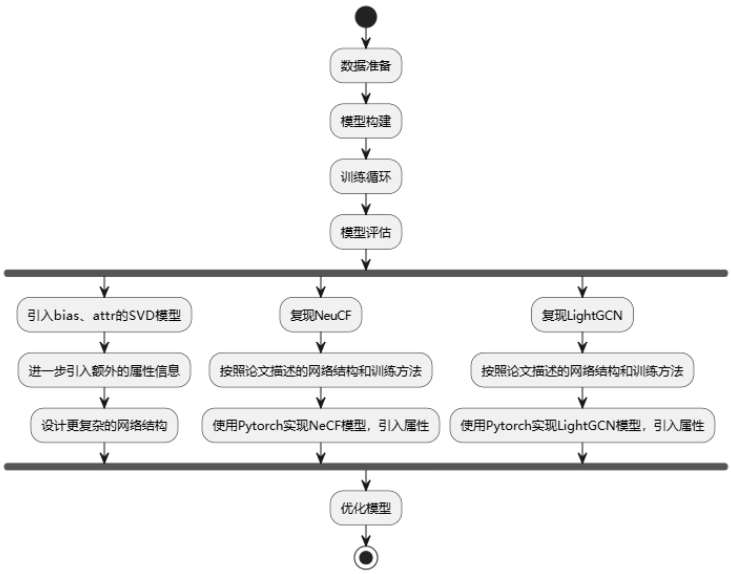

图 4.6: SVD 及相关改进实现思路图

2、模型建立

根据我们先前对 **MF 模型实际上是一个点积实现的单层感知机**的论断，我们首先可以先进行模型的建立，再根据输入层的要求，进行数据处理：

```python
class SVDModel(nn.Module):
    def __init__(self, num_users, num_items, latent_dim):
        super(SVDModel, self).__init__()
        self.user_factors = nn.Embedding(num_users, latent_dim)
        self.item_factors = nn.Embedding(num_items, latent_dim)
        self.user_factors.weight.data.uniform_(0, 0.05)
        self.item_factors.weight.data.uniform_(0, 0.05)

    def forward(self, user_idx, item_idx):
        user_factors = self.user_factors(user_idx)
        item_factors = self.item_factors(item_idx)
        dot = (user_factors * item_factors).sum(1)
        return dot

    def save_model(self, epoch=0, save_dir='model/SVD/model_SVD.pt'):
        if epoch != 0:
            save_dir = f'model/SVD/model_SVD_{epoch}_epochs.pt'
        torch.save(self.state_dict(), save_dir)
        print(f'Model saved at {save_dir}\n')

    def load_model(self, load_dir='model/SVD/model_SVD.pt'):
        self.load_state_dict(torch.load(load_dir))
        print(f'Model loaded from {load_dir}\n')
```

实现思路如下：

1. **定义模型类**：创建一个名为 `SVDModel` 的类，继承自 `nn.Module`。
2. **初始化参数**：
   - 在初始化方法 `__init__` 中，定义用户和物品的嵌入层。
   - 使用 `nn.Embedding` 为用户和物品创建嵌入矩阵，嵌入维度为 `latent_dim`。
   - 初始化嵌入矩阵的权重，使用均匀分布在 `[0, 0.05]` 范围内。
3. **前向传播**：
   - 在 `forward` 方法中，接收用户和物品的索引。
   - 使用嵌入层提取用户和物品的潜在因子。
   - 计算用户和物品因子的点积，并返回作为预测评分。

3、数据预处理

接下来，我们需要思考对于如上实现的模型，怎么进行数据的输入：

我们需要注意的是，嵌入层需要针对出现的物品、用户总数进行输入参数大小的设置，根据先前的数据统计，我们知道部分物品、用户是没有利用到的，因此，我们需要对用户、物品 id 建立映射，以进行嵌入层的输入。

为此，我们定义了 DatasetMapper 类，用于在推荐系统中管理用户、物品及属性之间的映射。这个类功能包括从训练和测试数据构建映射，并使用 pickle 保存和加载这些映射，以实现高效的数据检索。还包括从文件读取训练数据、测试数据和属性数据的函数，并进行必要处理。其主要代码如下：

```
class DatasetMapper:
    #存储和读取映射……

    def build_mappings(self, train_data, test_data):
        """从训练集和测试集数据中构建user_map和item_map"""
        train_users = {user for user, _, _ in train_data}
        test_users = {user for user, _ in test_data}
        all_users = train_users.union(test_users)

        train_items = {item for _, item, _ in train_data}
        test_items = {item for _, item in test_data}
        all_items = train_items.union(test_items)

        self.user_map = {user: idx for idx, user in enumerate(all_users)}
        self.item_map = {item: idx for idx, item in enumerate(all_items)}

    #存储和读取映射……
```

在这个函数中，我们完成了如下的功能：

1. **初始化映射字典**：初始化用户映射 (`user_map`)、物品映射 (`item_map`) 和属性映射 (`attr1_map` 和 `attr2_map`) 字典。
2. **构建映射**：

- 从训练集和测试集数据中提取用户和物品。
- 使用集合操作获取所有用户和物品。
- 为每个用户和物品分配一个唯一的索引，并存储在映射字典中。

3. **加载映射**：

- 从指定的 pickle 文件中加载用户映射和物品映射。

4. **保存映射**：

- 将用户映射和物品映射保存到指定的 pickle 文件中，以便高效读取。

在完成了映射的处理后，我们要将输入到 SVD 模型中的数据进行转换为 tensor 格式并映射，为此，我们提供了数据准备的接口：

```
def prepare_train_data(user_map, item_map, data):
    users = torch.tensor([user_map[u] for u, _, _ in data], dtype=torch.long)
    items = torch.tensor([item_map[i] for _, i, _ in data], dtype=torch.long)
    ratings = torch.tensor([r for _, _, r in data], dtype=torch.float32)
    return TensorDataset(users, items, ratings)
```

这里，我们将结果用 TensorDataset(users, items, ratings) 返回，是为了分批次的加速处理。

4、训练集、测试集划分

这是一个值得讨论的话题，为此我们单独分个章节来讨论一下：
我们给出了如下两种训练集、测试集的划分方式：
全局随机划分：

```
def split_train_test(data, train_ratio=0.9):
    train_size = int(len(data) * train_ratio)
    random.shuffle(data)
    return data[:train_size], data[train_size:]
```

- **描述**：
  - 将整个数据集打乱，然后根据给定的训练比例 (`train_ratio`) 划分成训练集和测试集。
  - 例如，如果 `train_ratio` 为 0.9，则 90% 的数据将用于训练，10% 的数据将用于测试。
- **适用场景**：
  - 适用于数据集较小且用户行为数据相对均匀的情况。
  - 当需要整体上评估模型的性能，而不关注特定用户的表现时，此方法更为适用。

按用户划分：

```
def split_train_test2(data):
    user_ratings = {}
    for user_id, item_id, score in data:
        if user_id not in user_ratings:
```

```
            user_ratings[user_id] = []
        user_ratings[user_id].append((item_id, score))
    train_data = []
    test_data = []
    for user_id, ratings in user_ratings.items():
        if ratings:
            random_rating = random.choice(ratings)
            test_data.append((user_id,) + random_rating)
            ratings.remove(random_rating)
            train_data.extend([(user_id, item_id, score) for item_id, score in
                ratings])

    return train_data, test_data
```

- **描述**:
  - 针对每个用户，随机选择一条评价记录作为测试集，其余的评价记录作为训练集。
  - 确保每个用户在测试集中至少有一条记录。
- **适用场景**:
  - 适用于用户行为数据较为丰富且希望确保每个用户在测试集中都有代表时。
  - 当需要评估模型对单个用户的个性化推荐效果时，此方法更为适用。

在我们的实验中，能够明确感受到第二种划分方式性能更为优秀，这也有我们数据集很大的原因(500 万条数据)。

5、模型训练

基础训练 为了加速实现，我们使用 cuda、分批次训练完成基础的训练代码，并用 tdqm 库完成进程可视化:

```
def train(model, train_data, test_data, num_epochs=20, lr=0.01, weight_decay=1e-6,
    batch_size=512, device='cuda'):
    model.to(device)
    criterion = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=lr, weight_decay=weight_decay)
    scheduler = StepLR(optimizer, step_size=5, gamma=0.9)

    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True,
        num_workers=12, pin_memory=True)
    test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False,
        num_workers=12, pin_memory=True)

    for epoch in range(num_epochs):
        model.train()
        with tqdm(train_loader, desc=f'Epoch {epoch + 1}/{num_epochs}', unit='batch')
            as train_iterator:
            for user, item, rating in train_iterator:
```

```
                user = user.to(device, non_blocking=True)
                item = item.to(device, non_blocking=True)
                rating = rating.to(device, non_blocking=True)

                optimizer.zero_grad()
                prediction = model(user, item)
                loss = criterion(prediction, rating)
                loss.backward()
                optimizer.step()

        scheduler.step()
```

1. **模型准备**：将模型加载到指定设备 (`device`)。
2. **定义损失函数和优化器**：
   - 使用均方误差 (`MSELoss`) 作为损失函数。
   - 使用随机梯度下降 (`SGD`) 作为优化器，并设置学习率和权重衰减。
   - 使用学习率调度器 (`StepLR`) 每隔一定的 epoch 调整学习率。
3. **数据加载**：
   - 使用 `DataLoader` 加载训练集和测试集，设置批量大小和其他参数。
4. **训练循环**：
   - 对每个 epoch 进行迭代，将模型设置为训练模式。
   - 对每个批次的数据进行前向传播、计算损失、反向传播和参数更新。
   - 每个 epoch 结束后，更新学习率。

如上是一个最基础的实现，但是，更进一步的，我们还需要增加别的处理：

训练时测试 为了更好的体现模型训练的变化，我们借助划分的测试集，记录模型训练中每个 epoch 下模型的 MSE 变化，为了更好反映情况，我们还引入了 AUC 指标：

```
all_predictions = []
all_ratings = []
with torch.no_grad():
    model.eval()
    for user, item, rating in test_loader:
        user = user.to(device, non_blocking=True)
        item = item.to(device, non_blocking=True)
        rating = rating.to(device, non_blocking=True)

        prediction = model(user, item)
        test_loss += criterion(prediction, rating).item() * len(user)

        all_predictions.extend(prediction.cpu().numpy())
```

```
        all_ratings.extend(rating.cpu().numpy())

test_loss /= len(test_loader.dataset)
 # Calculate AUC
if len(set(all_ratings)) > 1:
    auc = roc_auc_score((torch.tensor(all_ratings) >
        torch.tensor(all_ratings).mean()).numpy(), all_predictions)
else:
    auc = float('nan')

print(f'\n======Epoch {epoch + 1}/{num_epochs}======\nMSE Loss: {test_loss} AUC:
    {auc} \n')
```

1. **评估测试数据**：
   - 遍历 `test_loader` 中的每个批次。
   - 将用户、物品和评分数据移动到指定设备 (`device`)。使用模型进行预测。
   - 计算损失并累积。将预测值和实际评分转换为 numpy 数组并存储在相应的列表中。
   - **计算测试损失**：将累积的测试损失除以测试数据集的总长度。
2. **计算 AUC**：
   - 如果实际评分的取值大于 1 种，计算 AUC。
   - 否则，将 AUC 设置为 `nan`。
3. **打印结果**：打印当前 epoch 的 MSE 损失和 AUC。

注意这里我们还给出了模型的保存，每执行一个 epoch 自动保存文件

早停操作 为了方便挂机训练，我们还实现了早停操作：

```
# 早停
best_auc = 0
early_stopping_counter = 0
early_stopping_patience = 4

……
    if auc > best_auc:
        best_auc = auc
        early_stopping_counter = 0
    else:
        early_stopping_counter += 1
        if early_stopping_counter >= early_stopping_patience:
            print(f'Early stopping at epoch {epoch + 1}')
            break
```

完整训练过程 综上，我们给出完整的训练文件实现，其中我们根据模型不同可以切换，这里先只展示第一种：

```
# 读取训练数据
train_data = read_train_data('../data/train.txt')
atrr_data = read_attribute_data('../data/itemAttribute.txt')


#参数
latent_dim = 10
num_epochs= 50
lr=0.05
weight_decay=1e-6
model_type = "SVDattr"
data_divide_method = 1
gamma=0.75 #衰减率
batch_size=512
device='cuda'  #可以改成cpu

# 划分训练集和测试集
if data_divide_method == 1:
    train_data, test_data = split_train_test(train_data, train_ratio=0.9)
elif data_divide_method == 2:
    train_data, test_data = split_train_test2(train_data)


# 更改SVD/SVDbias来选择模型
if model_type == "SVD":
    # 获取映射、准备数据
    mapper = DatasetMapper()
    user_map, item_map = mapper.load_mappings()
    train_data = SVD.prepare_train_data(user_map, item_map, train_data)
    test_data = SVD.prepare_train_data(user_map, item_map, test_data)

    num_users = len(user_map)
    num_items = len(item_map)

    model = SVD.SVDModel(num_users, num_items, latent_dim)
    SVD.train(model, train_data, test_data, num_epochs, lr, weight_decay)
elif model_type == "SVDbias":
    ……
elif model_type == "SVDattr":
    ……
```

6、模型测试

类似的，我们需要准备测试数据，这里，为了还原原本的测试文件结果，还需要增加反向 map：

```
def prepare_test_data(user_map, item_map, data, test =False):
```

```
    ……

    user_inverse_map = {v: k for k, v in user_map.items()}
    item_inverse_map = {v: k for k, v in item_map.items()}
    return TensorDataset(users, items), user_inverse_map, item_inverse_map
```

基于此，我们可以效仿 train 函数的实现，给出如下的测试实现：

```
# 进行预测并输出结果
def test(model, user_map, item_map, test_data, output_filepath):

    test_dataset, user_inverse_map, item_inverse_map = prepare_test_data(user_map,
        item_map, test_data)
    test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)

    predictions = defaultdict(list)

    # 将张量传递给模型进行预测
    with torch.no_grad():
        for users, items in test_loader:
            preds = model(users, items)
            for i in range(len(users)):
                user_id = user_inverse_map[users[i].item()]
                item_id = item_inverse_map[items[i].item()]
                predictions[user_id].append((item_id, preds[i].item()))

    # 组织预测结果并写入文件
    with open(output_filepath, 'w') as f:
        for user_id, preds in predictions.items():
            num_ratings = len(preds)
            f.write(f"{user_id}|{num_ratings}\n")
            for item_id, score in preds:
                f.write(f"{item_id}\t{score:.6f}\n")

    print(f"预测结果已经写入到 {output_filepath}")
```

综上，就是我们的 SVD 的完整实现，难点主要在数据的处理上。模型评估我们放到最后的结果评估中展示，接下来，我们来进一步看看 SVD 的几个改进实现。

7、模型改进：bias、attr

引入 bias 在 pytorch 框架中的实现非常简单，我们只需要修改模型的框架即可。而引入 attr 模型就比较复杂，我们需要从头到尾的对模型进行修改，因此我们单独放到下个章节来叙述

```
class SVDbiasModel(nn.Module):
    def __init__(self, num_users, num_items, latent_dim):
        super(SVDbiasModel, self).__init__()
        # …… 与之前一致
        # 用户偏置
```

```
        self.item_bias = nn.Embedding(num_items, 1)  # 物品偏置
        self.global_bias = nn.Parameter(torch.zeros(1))  # 全局偏置

        # …… 与之前一致
        self.user_bias.weight.data.uniform_(0, 0.05)  # 初始化用户偏置
        self.item_bias.weight.data.uniform_(0, 0.05)  # 初始化物品偏置

    def forward(self, user_idx, item_idx):
        # …… 与之前一致
        user_bias = self.user_bias(user_idx).squeeze()
        item_bias = self.item_bias(item_idx).squeeze()
        dot = (user_factors * item_factors).sum(1)
        return dot + user_bias + item_bias + self.global_bias
```

1. **用户偏置 (`user_bias`)**:
   - 在模型中增加一个用户偏置向量，使用 `nn.Embedding(num_users, 1)` 创建。
   - 该偏置向量捕捉每个用户的整体偏好，比如某些用户可能普遍打高分或低分。
2. **物品偏置 (`item_bias`)**:
   - 在模型中增加一个物品偏置向量，使用 `nn.Embedding(num_items, 1)` 创建。
   - 该偏置向量捕捉每个物品的整体吸引力，比如某些物品可能普遍收到较高或较低的评分。
3. **全局偏置 (`global_bias`)**:
   - 在模型中增加一个全局偏置，使用 `nn.Parameter(torch.zeros(1))` 创建。
   - 该偏置表示所有评分的整体平均值。
4. **前向传播**:
   - 在计算用户和物品的潜在因子的点积 (`dot`) 后，加上用户偏置、物品偏置和全局偏置。
   - 最终的预测评分由点积结果和所有偏置项的和组成。

4.4 （提高实现）引入 attr 的 SVD

我们先假装不知道 NeMF 这个东西存在，试想，什么方法是最高效的引入 attr 属性的方法?

一个最简单的思路就是，我们分别进行三次 SVD 模型的计算，针对用户-物品、用户-属性 1、用户-属性 2 分别测试，然后按照固定加权把结果输出即可。

但是这样未免有些太丑陋了些，我们都用 pytorch 实现了 SVD，多少也应该优美一点吧。为此，我们可以把刚刚的思路改一改，固定加权对应的当然是 LR 的处理，我们把输出层更改为三种结果的求和，神经网络自己会在梯度下降中为我们解决权重适应的问题:

1、模型建立

于是，我们给出如下的完整代码实现:

```python
class SVDattrModel(nn.Module):
    def __init__(self, num_users, num_items, latent_dim, num_attr1, num_attr2):
        super(SVDattrModel, self).__init__()
        #…… 和之前一致

        # 新增两个物品属性的embedding
        self.item_attributes = nn.ModuleList([
            nn.Embedding(num_attr1, latent_dim),  # 假设第一个属性的类别数量为num_item_attributes[0]
            nn.Embedding(num_attr2, latent_dim)   # 假设第二个属性的类别数量为num_item_attributes[1]
        ])


        #…… 和之前一致
        for attr_embedding in self.item_attributes:
            attr_embedding.weight.data.uniform_(0, 0.05)

    def forward(self, user_idx, item_idx, item_attr1_idx, item_attr2_idx):
        #…… 和之前一致

        # 获取物品属性的embedding
        item_attr1_factors = self.item_attributes[0](item_attr1_idx)
        item_attr2_factors = self.item_attributes[1](item_attr2_idx)

        # 计算预测评分
        dot = (user_factors * item_factors).sum(1)
        attr1_term = (item_attr1_factors * user_factors).sum(1)
        attr2_term = (item_attr2_factors * user_factors).sum(1)

        prediction = dot +  attr1_term + attr2_term +user_bias + item_bias + self.global_bias

        return prediction
```

如下是借助 pytorch 的 torchviz 可视化的 SVDattrModel 模型的结构:

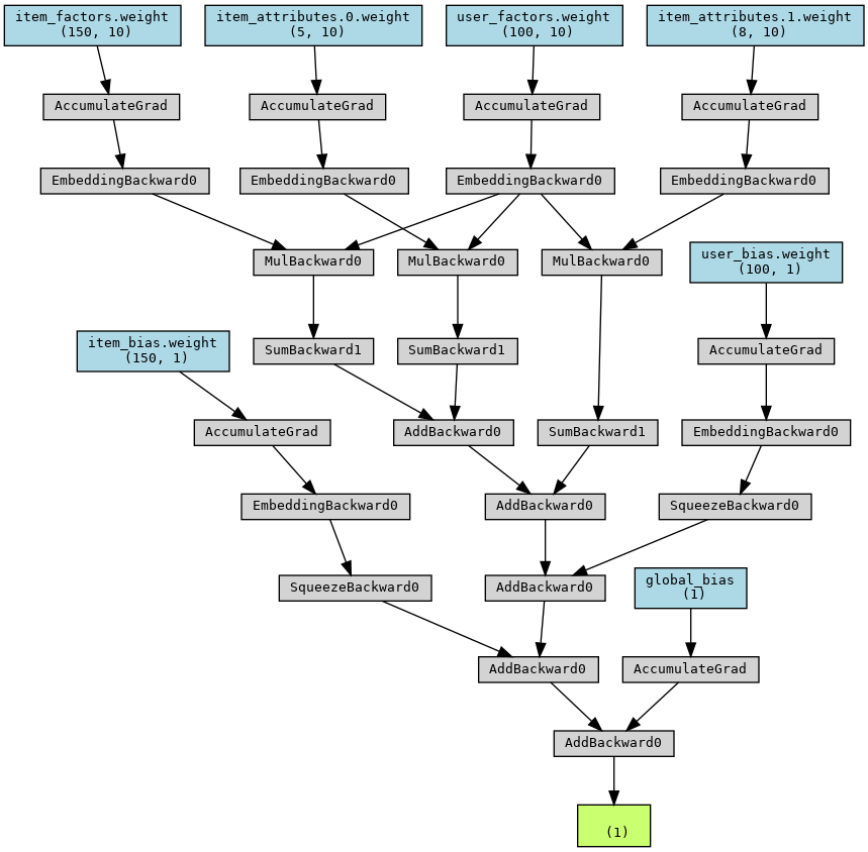

图 4.7: SVD+attr 模型结构

对于这个模型实现，我们需要在许多细节上进行修改。

2、数据预处理

首先，我们也需要对属性进行映射的处理，以传入 embedding 层。我们补充如下的实现

```
def build_attr_mappings(self, attr_data):
    """构建属性映射"""
    attr1_set = set()
    attr2_set = set()

    # 遍历attr_data，收集所有的属性值
    for attrs in attr_data.items():
        attr1_set.add(attrs[1][0])
        attr2_set.add(attrs[1][1])

    # 构建属性映射
    self.attr1_map = {attr: idx for idx, attr in enumerate(attr1_set)}
    self.attr2_map = {attr: idx for idx, attr in enumerate(attr2_set)}
```

此外，对于属性的处理，需要更复杂的实现，这是由于在我们的数据集合中，有很多物品是没有属性实现的:

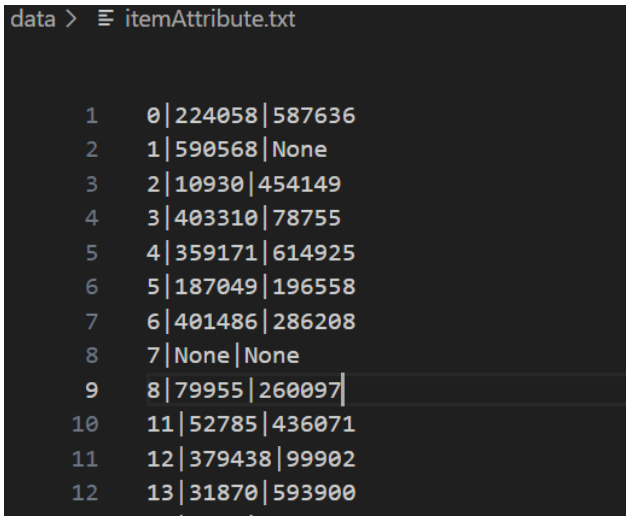

图 4.8: 部分物品缺失属性信息

为此，我们需要打个补丁，用下面的处理进一步完善我们的属性 map：

```
def map_attr_data(attr_data, attr1_map, attr2_map, item_map):
    mapped_data = {}
    for item_id, item_idx in item_map.items():
        if item_id in attr_data.keys():
            attr1 = attr_data[item_id][0]
            attr2 = attr_data[item_id][1]
            if attr1 in attr1_map and attr2 in attr2_map and item_id in item_map:
                attr1_idx = attr1_map[attr1]
                attr2_idx = attr2_map[attr2]
                mapped_data[item_idx] = (attr1_idx, attr2_idx)
        else :
            attr1 = -1
            attr2 = -1
            attr1_idx = attr1_map[attr1]
            attr2_idx = attr2_map[attr2]
            mapped_data[item_idx] = (attr1_idx, attr2_idx)

    return mapped_data
```

基于此，我们再在准备函数中稍作处理，就可以处理得到我们需要的数据，具体处理这里不再展示。

3、模型训练

在模型训练的过程中，我们只需要增加 attr 属性特征的输入即可：

```
user = user.to(device, non_blocking=True)
rating = rating.to(device, non_blocking=True)
item_raw_id = item_raw_id.to(device, non_blocking=True)
attr1 = attr1.to(device, non_blocking =True)
attr2 = attr2.to(device, non_blocking =True)
```

```
6
7  optimizer.zero_grad()
8  prediction = model(user, item_raw_id, attr1, attr2)
9  loss = criterion(prediction, rating)
10 loss.backward()
11 optimizer.step()
```

测试的处理同理，更多的细节处理这里不再展示，可以在代码中查看，最后给出训练的全部调用：

```
1  elif model_type == "SVDattr":
2      # 获取映射、准备数据
3      mapper = DatasetMapper()
4      user_map, item_map, attr1_map, attr2_map = mapper.load_attr_mappings()
5
6      num_users = len(user_map)
7      num_items = len(item_map)
8      num_attr1 = len(attr1_map)
9      num_attr2 = len(attr2_map)
10
11     # 这里更换为SVDattr特有的
12     attrmap = SVDattr.map_attr_data(atrr_data, attr1_map, attr2_map, item_map)
13     train_data = SVDattr.prepare_train_data(user_map, item_map, train_data, attrmap)
14     test_data = SVDattr.prepare_train_data(user_map, item_map, test_data, attrmap)
15
16     model = SVDattr.SVDattrModel(num_users, num_items, latent_dim, num_attr1,
           num_attr2)
17     SVDattr.train(model, train_data, test_data, num_epochs, lr, weight_decay)
```

4.5 （提高实现）GBDT+LR 方案实现

在先前的 SVD+attr 的基础上，一个稍显异想天开的想法是，对于我们的四个特征输入：用户、物品、属性 1、属性 2，能否进一步捕捉它们中的隐藏交叉特征呢?

GBDT+LR 方案可以解决这个问题。具体的思路是利用 SVD 得到用户、物品特征，加上属性特征进行 GBDT+LR 的训练完成特征交叉构建，得到一个更好地预测结果：

- **设计思路**：
  - 使用 SVD 模型提取用户和物品的隐含特征向量及偏置，并结合物品的属性特征。
  - 将这些特征作为输入，训练一个 GBDT 模型进行回归预测物品的评分。
  - 使用 GBDT 的叶子节点输出作为 LR 模型的输入特征，进一步提升预测效果。
- **代码实现**：
  - `prepare_gbdt_lr_data` 函数准备训练数据，包括用户和物品的 SVD 特征，以及物品的属性特征。
  - `train_gbdt_lr` 函数训练 GBDT 模型和 LR 模型，GBDT 模型输出叶子节点作为 LR 模型的输入特征。
  - `test_gbdt_lr` 函数使用训练好的模型在测试集上进行预测，并计算 RMSE 评估模型性能。

1、数据处理

```
elif model_type == "SVDattr":
# 准备用于 GBDT+LR 模型的训练数据
def prepare_gbdt_lr_data(user_map, item_map, data, item_attributes):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    users = torch.tensor([user_map[u] for u, _, _ in data],
        dtype=torch.long).to(device)
    items = torch.tensor([item_map[i] for _, i, _ in data],
        dtype=torch.long).to(device)

    # 加载训练好的 SVD 模型
    svd_model = SVD.SVDbiasModel(num_users=len(user_map), num_items=len(item_map),
        latent_dim=10)
    svd_model.load_model('../trainSVD_cuda/model/SVDbias/model_SVDbias_method2_best.pt')
    svd_model.eval().to(device)

    # 获取用户和物品的 embedding 以及bias的
    with torch.no_grad():
        user_embeddings = svd_model.user_factors(users).detach().cpu().numpy()
        item_embeddings = svd_model.item_factors(items).detach().cpu().numpy()
        ubs = svd_model.user_bias(users).detach().cpu().numpy()
        ibs = svd_model.item_bias(items).detach().cpu().numpy()

    # 构建物品属性特征
    item_features = []
    for item_id, attr1, attr2 in item_attributes:
        item_features.append([attr1, attr2])

    # 合并 embedding 和物品属性特征
    X = []
    y = []
    for i, (u, item, r) in enumerate(data):
        user_idx = user_map[u]
        item_idx = item_map[item]
        user_emb = user_embeddings[i]
        item_emb = item_embeddings[i]
        ub = ubs[i]
        ib = ibs[i]
        item_attr = item_features[item_idx]
        combined_features = list(user_emb) + list(item_emb) + item_attr + list(ub) +
            list(ib)
        X.append(combined_features)
        y.append(r)

    return X, y
```

实现思路如下:

- 函数 `prepare_gbdt_lr_data` 接受四个参数：`user_map`、`item_map`、`data` 和 `item_attributes`。
- 首先，根据当前环境选择设备（CPU 或 CUDA）。
- 然后，将用户和物品从数据 `data` 中提取为张量，并将它们移到选定的设备上。
- 加载预先训练好的 SVD 模型（`SVDbiasModel`），并将其设置为评估模式。
- 使用 SVD 模型预测得到的用户和物品的 embedding 向量以及偏置项（user bias 和 item bias）。
- 构建物品的属性特征，这些特征通常是预先定义好的属性，例如类别编号或者其他特定属性。
- 将用户 embedding、物品 embedding、物品属性特征以及偏置项组合成特征向量，并将这些特征向量作为 X。
- 将每条数据的评分作为 y。
- 返回特征向量 X 和对应的评分 y，用于训练 GBDT+LR 模型。

2、模型训练

```
# 训练 GBDT+LR 模型的函数
def train_gbdt_lr(X_train, y_train, model_save_path):
    # 初始化 LightGBM GBDT 模型
    gbdt_params = {
        'boosting_type': 'gbdt',
        'objective': 'regression',
        'metric': 'rmse',
        'num_leaves': 31,
        'learning_rate': 0.1,
        'n_estimators': 100,
        'feature_fraction': 0.9,
        'verbose': 2, # 设置 verbose 参数
        'device': 'cpu'
    }
    print("开始训练 GBDT 模型...")
    gbdt_model = lgb.LGBMRegressor(**gbdt_params)
    gbdt_model.fit(X_train, y_train)
    print("GBDT 模型训练完成")

    # 使用 GBDT 输出作为 LR 的特征
    X_gbdt = gbdt_model.predict(X_train, pred_leaf=True)
    lr_model = LinearRegression()
    print("开始训练 LR 模型...")
    lr_model.fit(X_gbdt, y_train)
    print("LR 模型训练完成")

    # 保存模型
    torch.save({
        'gbdt_model': gbdt_model,
        'lr_model': lr_model
```

```
    }, model_save_path)
    print(f"模型已保存到 {model_save_path}")
    return gbdt_model,lr_model
```

- 函数 `train_gbdt_lr` 接受三个参数：`X_train`、`y_train` 和 `model_save_path`。
- 首先，定义了用于训练 GBDT 模型的参数 `gbdt_params`，包括了树的数量、学习率等参数。
- 初始化 LightGBM GBDT 模型 `gbdt_model`，并使用 `X_train` 和 `y_train` 进行训练。
- 输出训练过程信息，如开始训练和训练完成。
- 使用训练好的 GBDT 模型 `gbdt_model` 预测 `X_train` 的叶子节点索引作为 LR 模型的输入特征。
- 初始化线性回归模型 `lr_model`，并使用 GBDT 输出的特征 `X_gbdt` 和 `y_train` 进行训练。
- 输出训练过程信息，如开始训练和训练完成。
- 将训练好的 GBDT 和 LR 模型保存到指定路径 `model_save_path`。
- 返回训练好的 GBDT 模型和 LR 模型。

4.6 （提高实现）NeuralCF 方案实现

终于来了！这里是我们实现的核心，具体思路我们已经在原理部分详细介绍了，这里我们直接来看代码！

我们给出如下的完整模型实现：

```
class NeuMF(nn.Module):
    def __init__(self, num_users, num_items, num_attr1, num_attr2, gmf_dim = 64,
        mlp_dim = 128, layers_dim=[256,128,64,32]):
        super(NeuMF, self).__init__()

        # GMF部分
        self.user_gmf = nn.Embedding(num_users, gmf_dim)
        self.item_gmf = nn.Embedding(num_items, gmf_dim)
        self.gmf_out = nn.Linear(gmf_dim, 1)

        # MLP部分
        self.user_mlp = nn.Embedding(num_users, mlp_dim)
        self.item_mlp = nn.Embedding(num_items, mlp_dim)
        self.mlp_layers = nn.ModuleList()
        now_num = 4 * mlp_dim
        for layer_dim in layers_dim:
            self.mlp_layers.append(nn.Linear(now_num, layer_dim)) # Concatenate user
                and item embeddings
            now_num = layer_dim

        self.mlp_out = nn.Linear(layers_dim[-1], 1)
        # Fusion layer
```

```
        self.fusion_layer = nn.Linear(2, 1)  # Fusion layer to combine GMF and MLP
            outputs

        # Item属性部分
        self.item_attr1 = nn.Embedding(num_attr1, mlp_dim)
        self.item_attr2 = nn.Embedding(num_attr2, mlp_dim)

        self.init_weights()

    def init_weights(self):
        # 初始化所有 embedding 和偏置
        # 这里不再展示啦，太多了

    def forward(self, user_idx, item_idx, item_attr1_idx, item_attr2_idx):
        # GMF部分
        user_gmf = self.user_gmf(user_idx)
        item_gmf = self.item_gmf(item_idx)
        gmf = user_gmf * item_gmf
        gmf_out = self.gmf_out(gmf)

        # MLP部分
        user_mlp = self.user_mlp(user_idx)
        item_mlp = self.item_mlp(item_idx)
        attr1 = self.item_attr1(item_attr1_idx)
        attr2 = self.item_attr2(item_attr2_idx)
        mlp_input = torch.cat([user_mlp, item_mlp, attr1, attr2], dim=1)
        for layer in self.mlp_layers:
            mlp_input = torch.relu(layer(mlp_input))

        mlp_out = self.mlp_out(mlp_input)

        fusion_input = torch.cat([gmf_out, mlp_out], dim=1)
        prediction = torch.sigmoid(self.fusion_layer(fusion_input)).squeeze() * 100

        return prediction
```

具体的实现上，我们采用如下方式完成：

- 构造函数 `__init__` 初始化了一个基于 PyTorch 框架的 NeuMF 模型，该模型是通过将 GMF 和 MLP 两个部分融合来预测用户对物品的评分。
- GMF 部分：
  - 使用 `nn.Embedding` 定义了用户和物品的 GMF 部分的 embedding 层 `user_gmf` 和 `item_gmf`，维度为 `gmf_dim`。
  - 使用 `nn.Linear` 定义了 GMF 部分的输出层 `gmf_out`，将 GMF 部分的结果映射到单个评分。
- MLP 部分：

  - 使用 `nn.Embedding` 定义了用户和物品的 MLP 部分的 embedding 层 `user_mlp` 和 `item_mlp`，维度为 `mlp_dim`。
  - 使用 `nn.ModuleList` 和 `nn.Linear` 定义了 MLP 部分的多层网络 `mlp_layers`，结构为全连接层，层数和每层神经元数由 `layers_dim` 控制。
  - 使用 `nn.Linear` 定义了 MLP 部分的输出层 `mlp_out`，将 MLP 部分的结果映射到单个评分。
- 融合层：
  - 使用 `nn.Linear` 定义了融合层 `fusion_layer`，将 GMF 和 MLP 部分的输出进行融合，并映射到最终的评分。
- 物品属性部分：
  - 使用 `nn.Embedding` 定义了物品的两个属性的 embedding 层 `item_attr1` 和 `item_attr2`，维度与 MLP 部分相同。
- 初始化权重：
  - 使用不同的初始化方法对所有的 embedding 层和线性层进行权重初始化，如均匀分布和 Xavier 初始化。
- `forward` 函数定义了模型的前向传播过程：
  - 计算 GMF 部分的输出 `gmf_out`。
  - 计算 MLP 部分的输出 `mlp_out`。
  - 将 GMF 和 MLP 部分的输出连接起来，输入到融合层 `fusion_layer`，并使用 sigmoid 函数将输出映射到预测评分。
  - 返回预测评分。

其他部分的数据处理和实现，和我们的 SVD+attr 模型是类似的，这里就不再过多展示，具体可能存在一些小细节差异，但是缺乏讨论的价值。

4.7 （提高实现）NGCF&LightGCN 方案尝试

迫于各方面时间原因，LightGCN 的最终实现在领域聚合的 Conv 传递中出现了一些问题，并不能很好地完成推荐任务，还需要进一步改进（但是没有时间了）。这里我们简要介绍我们实现的思路：

```
class LightGCN(nn.Module):
    def __init__(self, num_users, num_items, num_attr1, num_attr2, embedding_dim=32,
        num_layers=3):
        super(LightGCN, self).__init__()
        self.num_layers = num_layers

        # 初始化用户和项目嵌入
        self.user_embedding = nn.Embedding(num_users, embedding_dim)
        self.item_embedding = nn.Embedding(num_items, embedding_dim)

```

```
        # 初始化属性嵌入
        self.item_attr1 = nn.Embedding(num_attr1, embedding_dim)
        self.item_attr2 = nn.Embedding(num_attr2, embedding_dim)

        # 初始化权重
        nn.init.xavier_uniform_(self.user_embedding.weight)
        nn.init.xavier_uniform_(self.item_embedding.weight)
        nn.init.xavier_uniform_(self.item_attr1.weight)
        nn.init.xavier_uniform_(self.item_attr2.weight)

    def forward(self, user_idx, item_idx, item_attr1_idx, item_attr2_idx, edge_index):
        user_emb = self.user_embedding(user_idx)
        item_emb = self.item_embedding(item_idx)

        # 获取属性嵌入
        attr1_emb = self.item_attr1(item_attr1_idx)
        attr2_emb = self.item_attr2(item_attr2_idx)

        # 初始嵌入
        all_emb = torch.cat([user_emb, item_emb, attr1_emb, attr2_emb], dim=0)
        embs = [all_emb]
        # 邻域聚合
        for layer in range(self.num_layers):
            # 使用torch_geometric的GCNConv进行邻域聚合
            conv = GCNConv(all_emb.size(1), all_emb.size(1))
            all_emb = conv(all_emb, edge_index)
            embs.append(all_emb)

        # 组合嵌入
        embs = torch.stack(embs, dim=1)
        final_emb = torch.mean(embs, dim=1)

        # 获取最终的用户和项目嵌入
        user_final_emb = final_emb[user_idx]
        item_final_emb = final_emb[item_idx + len(user_idx)]

        # 预测输出
        score = (user_final_emb * item_final_emb).sum(dim=1)
        return score
```

- 构造函数 `__init__` 初始化了一个基于 PyTorch 框架的 LightGCN 模型，该模型利用 GCN 进行用户和物品嵌入的学习。
- 初始化部分：
  - 使用 `nn.Embedding` 分别定义了用户和物品的 embedding 层 `user_embedding` 和 `item_embedding`，维度为 `embedding_dim`。
  - 使用 `nn.Embedding` 分别定义了物品的两个属性的 embedding 层 `item_attr1` 和 `item_attr2`，

维度也为 `embedding_dim`。

- 使用 `nn.init.xavier_uniform_` 对所有的 embedding 层进行了 Xavier 初始化。

- `forward` 函数定义了模型的前向传播过程：
  - 获取输入的用户和物品的 embedding `user_emb` 和 `item_emb`。
  - 获取输入的物品属性的 embedding `attr1_emb` 和 `attr2_emb`。
  - 将所有的 embedding 连接起来形成初始的嵌入 `all_emb`。
  - 使用 `GCNConv` 进行多层的邻域聚合，每一层的输出嵌入 `all_emb` 被添加到列表 `embs` 中。
  - 将所有层的嵌入拼接在一起 `embs`，并取平均作为最终的嵌入 `final_emb`。
  - 根据输入的用户和物品索引获取最终的用户和物品嵌入 `user_final_emb` 和 `item_final_emb`。
  - 计算用户和物品的预测评分，采用点积方式计算。
  - 返回预测评分。

5 结果评估

5.1 算法复杂度分析

5.1.1 基于用户的协同过滤

该算法主要包括读取数据、构建评分矩阵、计算用户相似度、进行评分预测几个部分。我们对于各个部分的时间复杂度和空间复杂度进行了一定分析：

- **读取数据**：
  时间复杂度
  - `read_and_split_train_data(filepath)`: 读取和处理训练数据，时间复杂度为 $O(N+R)$，其中 N 是用户数，R 是评分总数。
  - `read_item_data(filepath)`: 读取物品数据，时间复杂度为 $O(M)$，其中 M 是物品数。
  - `read_test_data(filepath)`: 读取测试数据，时间复杂度为 $O(T)$，其中 T 是测试用户数。

  空间复杂度
  - 存储数据的字典结构，空间复杂度为 $O(N+R)$ 和 $O(M)$。

- **构建评分矩阵**：
  时间复杂度
  - `build_rating_matrix(train_data)`: 构建评分矩阵，时间复杂度为 $O(R)$，其中 R 是评分总数。

  空间复杂度
  - 评分矩阵的空间复杂度为 $O(N \times M)$，但由于使用稀疏矩阵表示，实际空间复杂度为 $O(R)$。

- **计算用户相似度**：
  时间复杂度

- `calculate_user_similarity(rating_matrix, batch_size=5000)`: 计算用户相似度，时间复杂度为 $O(U^2 \cdot I)$，其中 U 是用户数，I 是物品数。实际计算中使用分批计算，时间复杂度接近 $O(U \cdot I)$。

空间复杂度

- 存储相似度矩阵，空间复杂度为 $O(U^2)$。

- **进行评分预测**:
  时间复杂度

  - `predict_user_ratings`: 进行评分预测，时间复杂度为 $O(U \cdot I \cdot k)$，其中 k 是考虑的相似用户数。

  空间复杂度

  - 存储预测结果，空间复杂度为 $O(T \cdot I)$，其中 T 是测试用户数。

- **综合分析**:
  总时间复杂度：

  - 读取数据：$O(N + R + M + T)$
  - 构建评分矩阵：$O(R)$
  - 计算用户相似度：$O(U^2 \cdot I)$
  - 进行评分预测：$O(U \cdot I \cdot k)$

  总时间复杂度约为 $O(U^2 \cdot I + U \cdot I \cdot k)$，其中 U 通常远大于 I 和 k。

  总空间复杂度：

  - 存储数据和矩阵：$O(N + R + M + U^2 + T \cdot I)$

  总空间复杂度约为 $O(U^2 + R + M + T \cdot I)$，其中 U^2 是相似度矩阵的存储空间。

备注和改进：在实际应用中，如果用户数 U 非常大（例如百万级别），则计算用户相似度和存储相似度矩阵可能会成为瓶颈。可以采用优化方法，如用户降维、局部敏感哈希等技术来降低计算复杂度和存储需求。

5.1.2 基于物品的协同过滤

该算法主要包括读取数据、构建评分矩阵、计算物品相似度、进行评分预测几个部分。我们对于各个部分的时间复杂度和空间复杂度进行了一定分析：

- **读取数据**:
  时间复杂度

  - `read_train_data(filepath)`: 读取和处理训练数据，时间复杂度为 $O(N + R)$，其中 N 是用户数，R 是评分总数。
  - `read_item_data(filepath)`: 读取物品数据，时间复杂度为 $O(M)$，其中 M 是物品数。

- `read_test_data(filepath)`: 读取测试数据，时间复杂度为 $O(T)$，其中 T 是测试用户数。

空间复杂度

- 存储数据的字典结构，空间复杂度为 $O(N+R)$ 和 $O(M)$

与基于用户的协同过滤算法相似，读取数据部分的时间复杂度和空间复杂度都是基于读取的数据量来计算的。

- **构建评分矩阵**:
  时间复杂度
  - `build_rating_matrix(train_data)`: 构建评分矩阵，时间复杂度为 $O(R)$，其中 R 是评分总数。

  空间复杂度
  - 评分矩阵的空间复杂度为 $O(N \cdot M)$，但由于使用稀疏矩阵表示，实际空间复杂度为 $O(R)$
- **计算物品相似度**:
  时间复杂度
  - `calculate_similarity(target_item_id, user_id, user_purchases, rating_matrix, item_map, user_map)`: 计算物品相似度，时间复杂度为 $O(I \cdot P)$，其中 I 是物品数，P 是用户购买的物品数。

  空间复杂度
  - 存储相似度计算结果的临时变量，空间复杂度为 $O(P)$
- **进行评分预测**:
  时间复杂度
  - `predict_ratings(test_data, train_data, rating_matrix, user_map, item_map)`: 进行评分预测，时间复杂度为 $O(T \cdot I \cdot P)$，其中 T 是测试用户数, I 是物品数，P 是用户购买的物品数。

  空间复杂度
  - 存储预测结果，空间复杂度为 $O(T \cdot I)$
- **综合分析**:
  - **总时间复杂度**:
    * 读取数据：$O(N+R+M+T)$
    * 构建评分矩阵：$O(R)$
    * 计算物品相似度：$O(I \cdot P)$
    * 进行评分预测：$O(T \cdot I \cdot P)$

    总时间复杂度约为 $O(T \cdot I \cdot P$，其中 T 通常远大于 I 和 P。

- **总空间复杂度**:
  - 存储数据和矩阵：$O(N + R + M + T \cdot I)$

  总空间复杂度约为 $O(N + R + M + T \cdot I)$。

备注和改进：在实际应用中，如果物品数 I 非常大，则计算物品相似度和存储预测结果可能会成为瓶颈。可以采用优化方法，如物品降维、局部敏感哈希等技术来降低计算复杂度和存储需求。

5.1.3 SVD

算法涵盖了三个模型：基础的 SVD 模型、带偏置项的 SVDbias 模型以及 SVDattr 模型，这三个模型都是在协同过滤的基础上，通过矩阵分解来进行用户对物品的评分预测。

- SVDModel: 此模型通过两个嵌入层（用户和物品）来实现矩阵分解，其中的点积用于预测评分。
- SVDbiasModel: 在 SVDModel 的基础上增加了用户偏置、物品偏置和全局偏置，以提高预测的准确性。
- SVDattrModel: 此模型考虑了物品属性，通过对属性进行嵌入处理，并将其融入到评分预测中。

时间复杂度

- **模型训练**: 主要时间开销来自于前向和反向传播，计算复杂度主要依赖于用户数、物品数以及潜在特征维度。整体的时间复杂度大致为 $O(num\_epochs \times (N \cdot M \cdot latent\_dim))$，其中 N 是用户数，M 是物品数，$latent\_dim$ 是潜在因子的维数。
- **数据准备**: 包括用户和物品的映射以及训练数据的准备，这部分的复杂度为 $O(N + M)$。

空间复杂度

- **模型参数**: 主要包括用户和物品的嵌入矩阵，其空间复杂度为 $O((N + M) \cdot latent\_dim)$。对于带偏置的模型，还需要考虑偏置项的存储，这部分相对较小。
- **训练数据存储**: 依赖于训练数据集的大小，如果使用稀疏表示，可以显著减少存储需求。

5.2 算法性能分析

各类算法性能总览

如下结果中，SVD 算法使用第二种训练集划分方式。

Model	RMSE	Time Overhead	Space Overhead
ItemCF	28.463	2.22h	无意义/7.16g
UserCF	26.799	62.88s	7.02g
SVD	24.14	27s/epoch	7.71g
SVDbias	23.021	35s/epoch	7.9g
SVDattr	21.771	45s/epoch	8.13g
GBDT+LR	22.243	3096.41s	7.93g
NeuMF	20.347	220s/epoch	8.37g

表 5.1: Comparison of Recommendation Models

注意：需要注意的是，这里 ItemCF 一开始进行实现的时候，我们采用和 UserCF 类似的思路去计算相似度矩阵，但是发生了内存爆炸的现象，这是由于 ItemCF 的物品相似度矩阵的复杂度达到 50w*50w*size(float)，本机的内存直接崩了，后续我们改成了牺牲时间来换时间的方式 (不计算相似度矩阵，每次重新计算相似度)，把内存降下来了，能够正常运行 ItemCF，但是时间复杂度上升。

由于 SVD 系列算法和协同过滤实现差异大，接下来我们分开来展示

5.2.1 协同过滤算法

这里我们给个更详细的协同过滤算法的性能展示

表 5.2: **Result Valuation**

Model	Training Time	Predicting Time	Total Time	RMSE
UserCF	20.75s	42.13s	62.88s	26.7996
ItemCF	22.35s	2.22h	2.22h	28.463

5.2.2 SVD

(1) 模型训练过程展示

我们使用 tdqm 等可视化方式，可视化了模型的训练过程如下，并记录了部分数据：

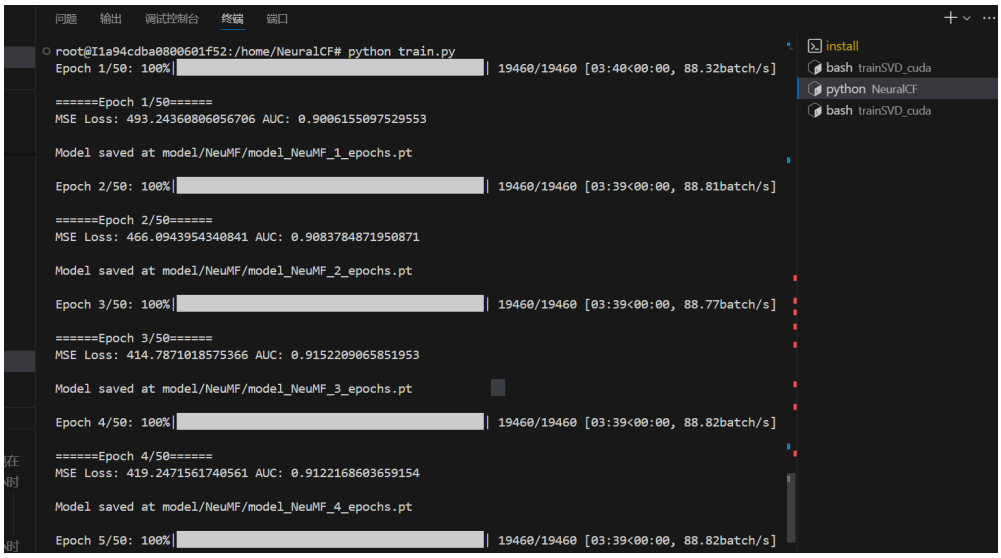

图 5.9: 训练过程信息展示

(2) 模型更新过程展示

在划分方式 1 下，对于实现的三种 SVD 算法，在此给出其训练中 MSE Loss 和 AUC 的变化图像。

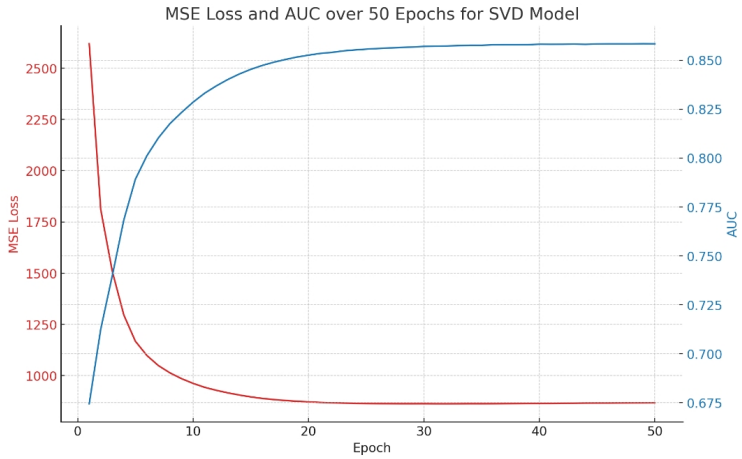

图 5.10: SVD 变化图-数据集划分方式 1

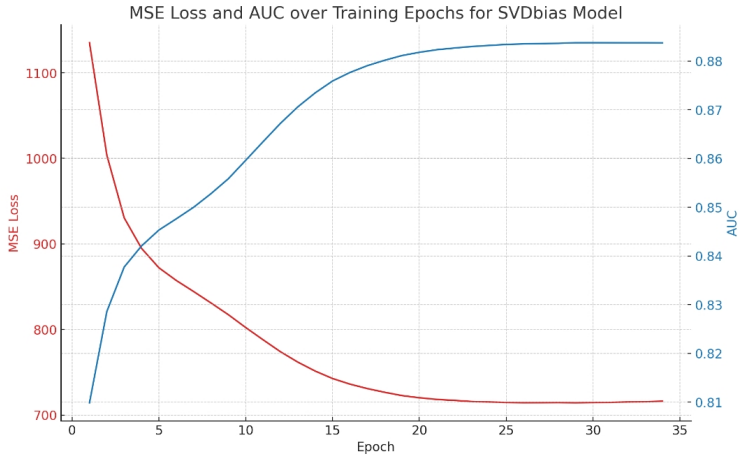

图 5.11: SVD_bias 变化图-数据集划分方式 1

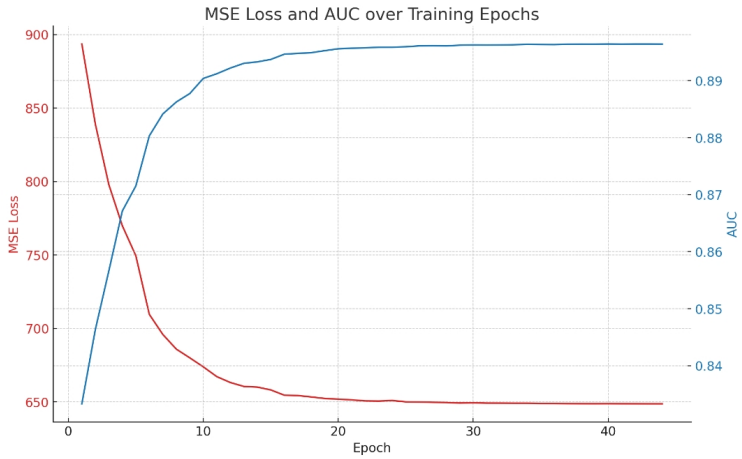

图 5.12: SVD_attr 变化图-数据集划分方式 1

在划分方式 2 下，对于实现的三种 SVD 算法，在此给出其训练中 MSE Loss 和 AUC 的变化图像：

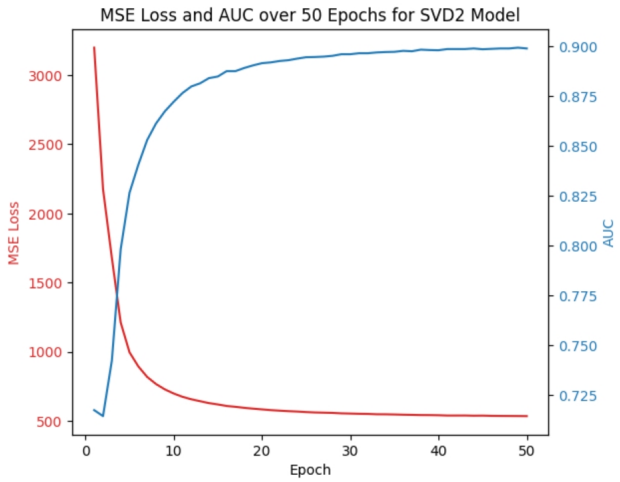

图 5.13: SVD 变化图-数据集划分方式 2

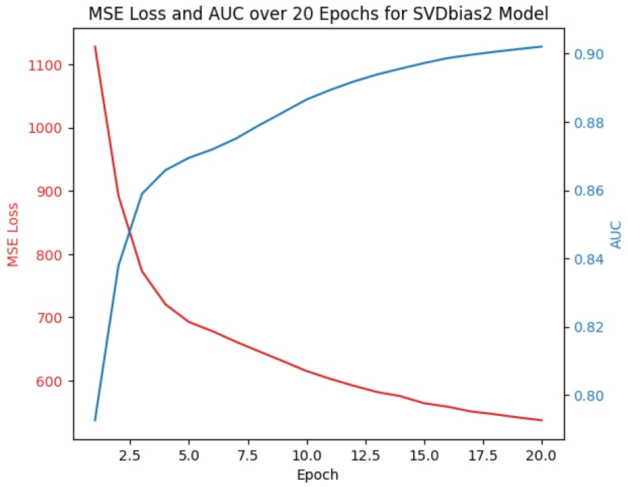

图 5.14: SVD_bias 变化图-数据集划分方式 2

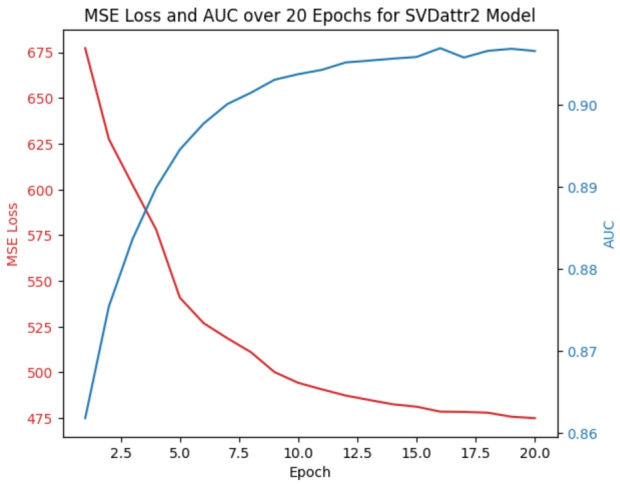

图 5.15: SVD_attr 变化图-数据集划分方式 2

此外，NeuMF 算法在两种不同划分方式下的性能变化如下：

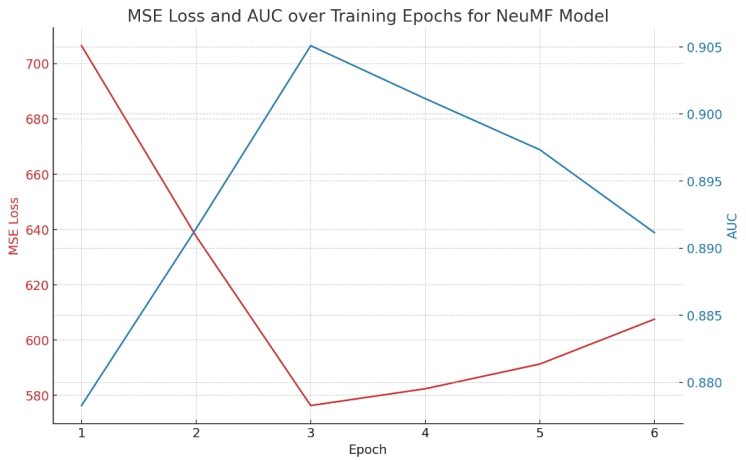

图 5.16: NeuMF 变化图-数据集划分方式 1

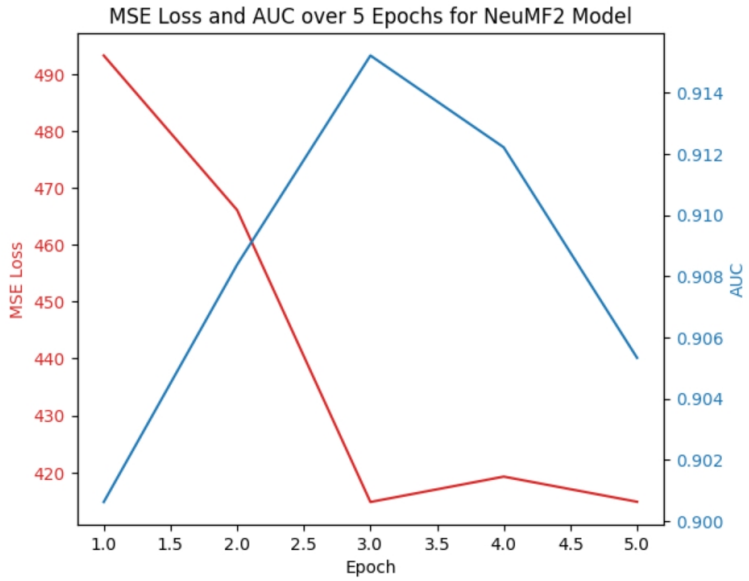

图 5.17: NeuMF 变化图-数据集划分方式 2

(3) 模型内存占用分析

我们通过直接查看服务器后端数据来查看：

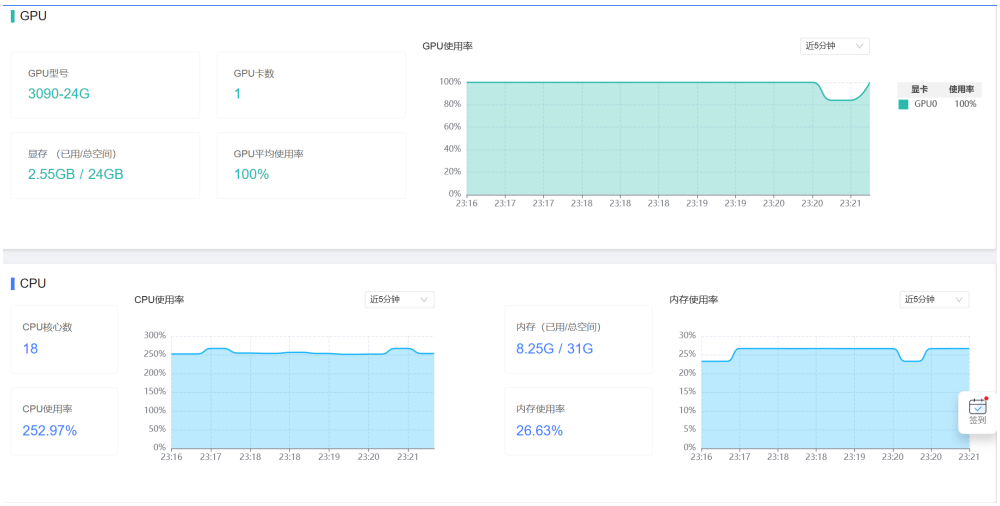

图 5.18: NeuCF 训练后台数据显示

可以看到，NeuCF 训练过程中，gpu 是主要性能瓶颈，能够占用率达到 100%，但是 cpu 占有率只有 330%(8 核处理器)。

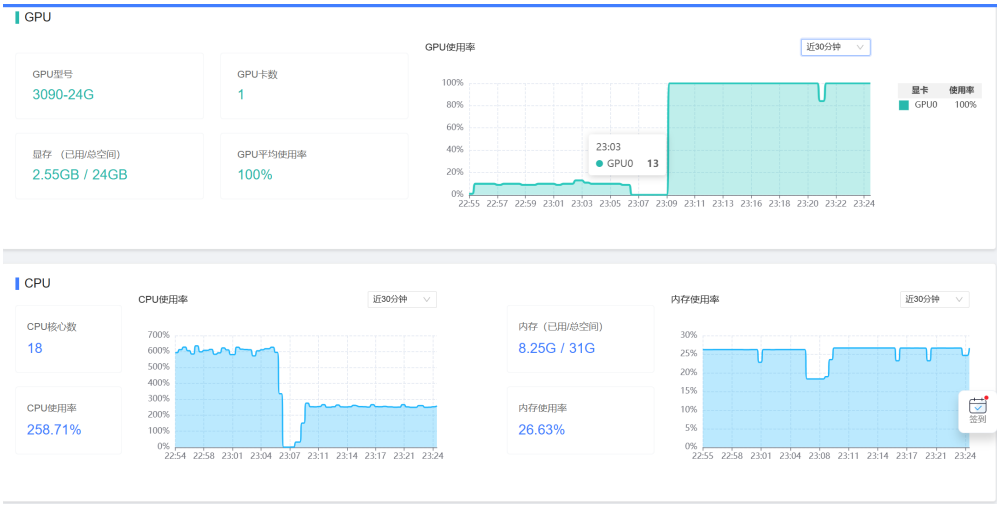

图 5.19: SVD 系列训练后台数据显示

相对的，次优的 SVD+attr 等 SVD 系列算法训练过程中，cpu 则为主要性能瓶颈，占有率达到 610%(8 核处理器)，但是 gpu 占有率只有 10%。

6 遇到的问题及解决方案

遇到的问题及解决方案已在前文中阐述，再此不再赘述。

7 实验总结

本次实验顺利完成了推荐算法的实现，并对多种实现算法进行了尝试，对于大数据处理有了基本的了解。

感谢杨老师本学期的讲授。

8 文件说明

在源码文件中，主要包括 CF、data、GBDT+LR、LightGCN、NeuralMF、SVD_cuda 几个文件夹。

- data 文件夹主要是存储相关数据
- CF 文件夹主要是对两种协同过滤算法进行了实现，其中包括 itemCF.py、userCF.py 和 RMSE.py 文件。相关结果会输出在./CF/result 文件夹中。其中，result_itemCF_light.txt 文件是对 itemCF 进行的小范围测试。
- SVD_cuda 文件夹是对 SVD 相关算法的实现，包括 SVD、SVD_bias 和 SVD_attr。需首先运行目录下的 data_process.py 进行索引构建，接着运行 train.py，选择模型即可进行训练。训练完成后，需运行 test.py，并选择对应的模型进行测试。
- GBDT+LR 文件夹是对 GBDT+LR 算法的实现。需首先运行目录下的 data_process.py 进行索引构建，接着运行 train.py 进行训练。训练完成后，需运行 test.py 进行测试。
- NeuralMF 文件夹是对 NeuralMF 算法的实现。需首先运行目录下的 data_process.py 进行索引构建，接着运行 train.py 进行训练。训练完成后，需运行 test.py，并选择对应的模型进行测试。
- LightGCN 文件夹是对 LightGCN 算法的实现。需首先运行目录下的 data_process.py 进行索引构建，接着运行 train.py 即可进行训练。训练完成后，需运行 test.py，并选择对应的模型进行测试。

在可执行文件中，对于前面实现的各个源码文件进行了打包，需要使用命令行在工作区目录下（即在 CF、GBDT+LR、LightGCN、NeuralCF 和 SVD_cuda 几个文件夹目录下）运行，运行顺序参考上述内容，相关内容会对应输出到响应的文件夹中，否则会出现路径错误导致运行失败。