南開大學
Nankai University

计算机学院

大数据计算与应用

推荐系统编程

张瑜 2212040 延嵩 2211555 刘沛鑫 2213109

年级：2022 级

指导教师：杨征路

2025 年 6 月 8 日

目录

一、 实验要求

任务：预测 Test.txt 文件中用户与物品对（u，i）的评分。
数据集：

- Train.txt，用于训练模型。
- Test.txt，用于测试。
- ResultForm.txt，是结果文件的格式。
- 数据集的格式已在 DataFormatExplanation.txt 中进行了说明。

在这个项目中，你需要预测 Test.txt 文件中未知用户与物品对（u，i）的评分。你可以使用课程中学习到的任何算法，或者从其他资源（例如在线课程）中获取的算法。

一个小组（最多由三名学生组成）需要撰写一份关于该项目的报告。报告应包括但不限于以下内容：

- 数据集的基本统计信息（例如，用户数量、评分数量、物品数量等）；
- 算法的详细信息；
- 推荐算法的实验结果（均方根误差（RMSE）、训练时间、空间消耗）；
- 算法的理论分析和/或实验分析。

二、 数据集描述

```python
def parse_train(filename):
    user_count = 0
    item_set = set()
    rating_count = 0
    rating_values = []
    user_rating_count = []
    item_rating_counter = Counter()
    with open(filename, 'r', encoding='utf-8') as f:
        while True:
            head = f.readline()
            if not head:
                break
            user_count += 1
            user_id, num = head.strip().split('|')
            num = int(num)
            user_rating_count.append(num)
            for _ in range(num):
                line = f.readline()
                if not line:
                    break
                item_id, score = line.strip().split()
                item_set.add(item_id)
```

```
                rating_count += 1
                rating_values.append(int(score))
                item_rating_counter[item_id] += 1
    return {
        'user_count': user_count,
        'item_count': len(item_set),
        'rating_count': rating_count,
        'rating_values': rating_values,
        'user_rating_count': user_rating_count,
        'item_rating_counter': item_rating_counter,
    }

def parse_test(filename):
    user_count = 0
    item_set = set()
    rating_count = 0
    user_rating_count = []
    item_rating_counter = Counter()
    with open(filename, 'r', encoding='utf-8') as f:
        while True:
            head = f.readline()
            if not head:
                break
            user_count += 1
            user_id, num = head.strip().split('|')
            num = int(num)
            user_rating_count.append(num)
            for _ in range(num):
                line = f.readline()
                if not line:
                    break
                item_id = line.strip()
                item_set.add(item_id)
                rating_count += 1
                item_rating_counter[item_id] += 1
    return {
        'user_count': user_count,
        'item_count': len(item_set),
        'rating_count': rating_count,
        'user_rating_count': user_rating_count,
        'item_rating_counter': item_rating_counter,
    }

def stat_list(lst):
    import numpy as np
    arr = np.array(lst)
    return {
        'min': int(arr.min()),
```

```
        'max': int(arr.max()),
        'mean': float(arr.mean()),
        'std': float(arr.std()),
        'median': float(np.median(arr)),
    }
```

得到结果如下:

训练集

- 用户数: 598
- 物品数: 9077
- 评分数: 90854
- 评分分布: 最小 10，最大 100，均值约 70，标准差约 21，中位数 70
- 每个用户评分数分布: 最小 15，最大 2678，均值约 152，标准差 254，中位数 59
- 每个物品被评分数分布: 最小 1，最大 280，均值约 10，标准差 20.3，中位数 3

测试集

- 用户数: 610
- 交互数: 9982
- 每个用户交互数分布: 最小 5，最大 496，均值约 16.4，标准差 30.2，中位数 16
- 每个物品被交互数分布: 最小 1，最大 56，均值约 2.76，标准差 4.2，中位数 1

训练集与测试集交集

- 训练集和测试集共有物品数: 2971
- 只在测试集出现的物品数: 647
- 训练集用户数: 598，测试集用户数: 610

我们可以得出以下结论:

1. 冷启动问题明显:
    - 测试集有 647 个物品在训练集未出现，意味着需要为这些物品设计特殊的预测策略（如用全局均值、用户均值、物品均值、或基于内容的特征等）。
    - 测试集有部分用户在训练集未出现（测试集用户数比训练集多），同理也要考虑冷启动用户。
2. 数据稀疏且长尾明显:
    - 物品被评分数分布中位数仅 3，均值 10，大部分物品被评分很少，只有极少数物品很热门（最大 280）。

- 用户评分数分布中位数 59，均值 152，说明有少数用户非常活跃，大部分用户评分较少。
- 这说明你的评分矩阵非常稀疏，基于邻域的协同过滤如 UserCF、ItemCF 可能效果有限，可以考虑矩阵分解（如 SVD、ALS）、深度学习（如 AutoRec、NCF）等方法。

3. 评分分布偏高且较均匀:

   评分均值 70，标准差 21，分布较宽但整体偏高。可以考虑归一化评分或直接预测原始分数。

4. 测试集交互量远小于训练集:

   训练集评分数 9 万 +，测试集交互数约 1 万，说明测试集只是部分用户/物品的部分交互，评测时要关注泛化能力。

三、 实验原理

(一) UserCF

UserCF 的核心思想是“物以类聚，人以群分”。它通过分析用户之间的相似性，为用户推荐其他相似用户喜欢的物品。假设如果两个用户在过去的某些行为（如购买商品、观看视频等）上表现出相似性，那么他们在未来的兴趣上也可能是相似的。

- **简单余弦相似度**：将每个用户表示为一个向量，向量的每个元素对应一个物品，如果用户对物品感兴趣，则值为 1，否则为 0。用户之间的相似度可以通过计算向量夹角的余弦值来衡量。

$$\text{sim}(u_1, u_2) = \frac{\sum_{l \in I} 1}{\sqrt{|J_1| \cdot |J_2|}}$$

  其中，J_1 和 J_2 分别是用户 u_1 和 u_2 喜欢的物品集合，$I = J_1 \cap J_2$ 是它们的交集。

- **改进的余弦相似度(User-IIF)**:为了降低热门物品对相似度的影响,引入惩罚因子 $1/\log(1+n_l)$，其中 n_l 是喜欢物品 l 的用户数量。

$$\text{sim}(u_1, u_2) = \frac{\sum_{l \in I} \frac{1}{\log(1+n_l)}}{\sqrt{|J_1| \cdot |J_2|}}$$

推荐流程如下:

1. **选择邻居用户**：根据相似度，选择与目标用户最相似的 K 个用户。
2. **推荐物品**：从这些邻居用户的喜好中，选择目标用户未接触过的物品。
3. **评分预测**：计算目标用户对这些物品的感兴趣程度（通常通过加权平均的方式），并根据评分高低进行排序。
4. **生成推荐列表**：选择评分最高的 N 个物品作为推荐结果。

- **优点**:
  - 更注重社会化，推荐结果更符合用户所在兴趣小组的热门物品。
  - 适用于用户兴趣变化较慢的场景。
- **缺点**:
  - 当用户数量较多时，计算复杂度较高（时间复杂度为 $O(m^2)$，其中 m 是用户数量）。
  - 新用户冷启动问题较难解决。

(二) ItemCF

ItemCF 的核心思想是“如果你喜欢某个物品，那么你可能也会喜欢与该物品相似的其他物品”。它通过计算物品之间的相似度，为用户推荐与其历史行为物品相似的其他物品。

- **简单余弦相似度**：将每个物品表示为一个向量，向量的每个元素对应一个用户，如果用户对物品感兴趣，则值为 1，否则为 0。物品之间的相似度可以通过计算向量夹角的余弦值来衡量。

$$\text{sim}(i_1, i_2) = \frac{\sum_{v \in V} \text{like}(v, i_1) \cdot \text{like}(v, i_2)}{\sqrt{\sum_{u_1 \in W_1} \text{like}^2(u_1, i_1)} \cdot \sqrt{\sum_{u_2 \in W_2} \text{like}^2(u_2, i_2)}}$$

  其中，W_1 和 W_2 分别是喜欢物品 i_1 和 i_2 的用户集合。

- **调整余弦相似度**：在计算物品相似度时，可以对用户评分进行中心化处理，以减少用户评分偏差的影响。

推荐流程如下：

1. **物品相似度计算**：基于用户的评分行为，计算物品两两之间的相似度。
2. **相似物品筛选**：对目标物品，选出与之最相似的 Top-K 物品。
3. **推荐生成或评分预测**：结合用户历史评分和相似度权重，预测用户对尚未交互物品的兴趣程度。

- **优点**：
  - 更注重个性化，推荐结果新颖性较高，容易发现并推荐长尾物品。
  - 当物品数量较少时，计算复杂度较低（时间复杂度为 $O(n^2)$，其中 n 是物品数量）。
- **缺点**：
  - 新物品冷启动问题较难解决。
  - 对于用户数量远大于物品数量的场景，计算效率较低。

(三) MLP

其基本结构如下：

- **神经元（Perceptron）**：MLP 的基本单元是神经元。每个神经元接收多个输入信号，对这些输入信号进行加权求和，然后通过一个非线性激活函数（如 Sigmoid、ReLU 等）输出一个信号。
- **多层结构**：MLP 由多个层次组成，包括输入层、隐藏层和输出层。输入层接收外部输入数据，隐藏层用于提取数据的特征，输出层根据隐藏层的输出生成最终的预测结果。隐藏层可以有多个，层数越多，网络的深度越大，能够学习到更复杂的特征。
- **连接权重**：相邻层的神经元之间通过权重相连。权重是网络的参数，决定了输入信号对输出信号的影响程度。在训练过程中，权重会不断调整以优化网络的性能。

前向传播：

- **输入信号**：输入层接收输入数据，每个输入特征对应一个神经元。

- **加权求和**：每个隐藏层神经元将输入信号乘以对应的权重后求和，再加上一个偏置项（bias），得到一个线性组合的结果。
- **激活函数**：线性组合的结果通过激活函数进行非线性变换，激活函数的作用是引入非线性，使网络能够学习复杂的模式。常见的激活函数包括：
  - **Sigmoid 函数**：输出范围在 (0,1) 之间，具有良好的可微性，但容易出现梯度消失问题。
  - **ReLU 函数（Rectified Linear Unit）**：输出为 $\max(0, x)$，计算简单，能够有效缓解梯度消失问题，但可能导致部分神经元“死亡”。
  - **Tanh 函数**：输出范围在 (-1,1) 之间，是对称的非线性函数，性能通常优于 Sigmoid 函数。
- **逐层传递**：经过激活函数处理后的信号作为下一层神经元的输入，逐层传递，直到输出层。输出层的神经元数量通常与任务的目标类别数或输出维度相关。

损失函数：

- **定义**：损失函数用于衡量网络输出与真实标签之间的差异，是优化网络参数的依据。常见的损失函数包括：
  - **均方误差（MSE）**：适用于回归任务，计算预测值与真实值之间的平方差的平均值。

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

    其中，y_i 是真实值，$\hat{y}_i$ 是预测值，N 是样本数量。
  - **交叉熵损失（Cross-Entropy Loss）**：适用于分类任务，衡量预测概率分布与真实概率分布之间的差异。

$$\text{Cross-Entropy} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

    其中，C 是类别总数，y_i 是真实标签的独热编码，$\hat{y}_i$ 是预测概率。

反向传播：

- **目标**：通过反向传播算法计算损失函数对每个权重的梯度，然后利用梯度下降法更新权重，使损失函数值最小化。
- **计算过程**：
  1. **计算输出层梯度**：根据损失函数对输出层的激活函数求导，得到输出层的误差梯度。
  2. **逐层反向传播**：从输出层开始，逐层向前计算隐藏层的误差梯度。对于每个隐藏层神经元，误差梯度由两部分组成：一是来自上一层的误差梯度，二是当前层激活函数的导数。
  3. **计算权重梯度**：根据误差梯度和输入信号，计算每个权重的梯度。
  4. **更新权重**：使用梯度下降法或其他优化算法（如 Adam、RMSprop 等）更新权重。

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L}{\partial w}$$

     其中，w 是权重，η 是学习率，$\frac{\partial L}{\partial w}$ 是权重的梯度。

训练过程如下:

- **初始化权重**：在训练开始之前，需要对权重进行初始化。常见的初始化方法包括随机初始化、Xavier 初始化和 He 初始化等。合适的初始化方法可以加速训练过程，避免梯度消失或梯度爆炸问题。
- **前向传播**：将训练数据输入网络，通过前向传播计算网络的输出和损失函数值。
- **反向传播**：根据损失函数值，通过反向传播计算权重的梯度，并更新权重。
- **迭代优化**：重复前向传播和反向传播的过程，直到损失函数值收敛或达到预设的训练轮数。

(四) ALS

ALS（Alternating Least Squares）算法是一种基于矩阵分解的协同过滤算法，用于推荐系统中预测用户对物品的评分。其核心思想是将用户-物品评分矩阵分解为两个低秩矩阵：用户因子矩阵（User Factors）和物品因子矩阵（Item Factors）。通过交替最小二乘法优化这两个矩阵，从而实现对未知评分的预测。

1. 矩阵分解

假设用户-物品评分矩阵为 R，其维度为 $m \times n$，其中 m 是用户数量，n 是物品数量。ALS 算法的目标是将 R 分解为两个低秩矩阵 U 和 V：

$$R \approx U \times V^T$$

其中:

- U 是用户因子矩阵，维度为 $m \times k$。
- V 是物品因子矩阵，维度为 $n \times k$。
- k 是因子数量（通常远小于 m 和 n）。

这种分解的目的是用两个较小的矩阵来近似表示原始的评分矩阵，从而简化模型并提高计算效率。

2. 优化目标

ALS 算法通过最小化以下目标函数来优化 U 和 V：

$$\min_{U,V} \sum_{(i,j)\in\Omega} (R_{ij} - U_i \cdot V_j^T)^2 + \lambda(\|U\|^2 + \|V\|^2)$$

其中:

- Ω 是已知评分的集合。
- R_{ij} 是用户 i 对物品 j 的评分。
- U_i 是用户 i 的因子向量。
- V_j 是物品 j 的因子向量。
- λ 是正则化参数，用于防止过拟合。

这个目标函数的目的是最小化预测评分和实际评分之间的差异，同时通过正则化项防止模型过拟合。

3. 交替最小二乘法

ALS 算法通过交替固定一个矩阵，优化另一个矩阵来逐步逼近最优解。具体步骤如下：

- **初始化**

  随机初始化用户因子矩阵 U 和物品因子矩阵 V。通常可以使用正态分布随机初始化：

  $$U \sim \mathcal{N}(0, \sigma^2), \quad V \sim \mathcal{N}(0, \sigma^2)$$

  其中，σ 是标准差，通常取较小的值（如 0.1）。

  随机初始化是为了给算法一个起点，后续通过优化逐步调整这些值。

- **固定 V，优化 U** 对于每个用户 i，固定物品因子矩阵 V，通过最小化以下目标函数来更新 U_i：

  $$\min_{U_i} \sum_{j \in \Omega_i} (R_{ij} - U_i \cdot V_j^T)^2 + \lambda \|U_i\|^2$$

  这是一个关于 U_i 的线性最小二乘问题，可以通过求解以下方程来更新 U_i：

  $$(V_{\Omega_i}^T V_{\Omega_i} + \lambda I) U_i = V_{\Omega_i}^T R_{i,\Omega_i}$$

  其中：

  - V_{Ω_i} 是与用户 i 有评分的物品对应的子矩阵。
  - R_{i,Ω_i} 是用户 i 的评分向量。
  - I 是单位矩阵。

  这个步骤的目的是针对每个用户，根据已知评分和物品因子矩阵，计算出最优的用户因子向量。

- **固定 U，优化 V** 对于每个物品 j，固定用户因子矩阵 U，通过最小化以下目标函数来更新 V_j：

  $$\min_{V_j} \sum_{i \in \Omega_j} (R_{ij} - U_i \cdot V_j^T)^2 + \lambda \|V_j\|^2$$

  这是一个关于 V_j 的线性最小二乘问题，可以通过求解以下方程来更新 V_j：

  $$(U_{\Omega_j}^T U_{\Omega_j} + \lambda I) V_j = U_{\Omega_j}^T R_{\Omega_j,j}$$

  其中：

  - U_{Ω_j} 是与物品 j 有评分的用户对应的子矩阵。
  - $R_{\Omega_j,j}$ 是物品 j 的评分向量。
  - I 是单位矩阵。

  这个步骤的目的是针对每个物品，根据已知评分和用户因子矩阵，计算出最优的物品因子向量。

- **迭代**重复上述步骤，直到收敛或达到预设的迭代次数。

  通过不断交替优化用户因子矩阵和物品因子矩阵，逐步逼近最优解。每次迭代都会使预测评分更接近实际评分

4. 预测评分

在优化完成后，可以通过用户因子矩阵 U 和物品因子矩阵 V 来预测未知评分：

$$\hat{R}_{ij} = U_i \cdot V_j^T$$

其中，$\hat{R}_{ij}$ 是用户 i 对物品 j 的预测评分。

通过计算用户因子向量和物品因子向量的点积，得到用户对物品的预测评分

ALS 算法通过矩阵分解和交替最小二乘法，逐步优化用户因子矩阵和物品因子矩阵，从而实现对未知评分的预测。这种方法不仅能够有效处理大规模数据集，还能通过正则化防止过拟合，提高模型的泛化能力。

（五） SVD

奇异值分解（Singular Value Decomposition, SVD）是推荐系统中经典的协同过滤算法，其核心思想是通过矩阵分解技术将用户-物品评分矩阵分解为低维隐因子表示，从而发现用户和物品的潜在特征模式。

1. 矩阵分解基础理论

问题建模：设用户-物品评分矩阵为 $R \in \mathbb{R}^{m \times n}$，其中 m 为用户数量，n 为物品数量。矩阵分解的目标是找到两个低维矩阵：

- 用户隐因子矩阵：$P \in \mathbb{R}^{m \times k}$
- 物品隐因子矩阵：$Q \in \mathbb{R}^{n \times k}$

使得 $R \approx PQ^T$，其中 $k \ll \min(m, n)$ 为隐因子维度。

基本假设：

1. **低秩假设**：用户-物品交互矩阵具有低秩结构，可以用少量隐因子表示
2. **隐因子解释性**：每个隐因子对应某种潜在的用户偏好或物品属性（如电影的类型、用户的年龄群体等）
3. **线性组合假设**：用户对物品的偏好可以表示为隐因子的线性组合

2. FunkSVD 算法原理

FunkSVD 是 Simon Funk 在 Netflix Prize 竞赛中提出的矩阵分解算法，是现代推荐系统矩阵分解方法的基础。

预测模型：对于用户 u 对物品 i 的评分预测：

$$\hat{r}_{ui} = p_u^T q_i = \sum_{f=1}^{k} p_{uf} \cdot q_{if}$$

其中：

- $p_u \in \mathbb{R}^k$ 为用户 u 的隐因子向量
- $q_i \in \mathbb{R}^k$ 为物品 i 的隐因子向量
- k 为隐因子维度

优化目标：最小化已观察评分的预测误差:

$$\min_{P,Q} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - p_u^T q_i)^2 + \lambda(\|P\|_F^2 + \|Q\|_F^2)$$

其中:

- $\mathcal{K}$ 为已观察评分的集合
- λ 为正则化参数，防止过拟合
- $\|\cdot\|_F$ 为 Frobenius 范数

梯度下降求解：对参数的偏导数为:

$$\frac{\partial}{\partial p_{uf}} = -2e_{ui} \cdot q_{if} + 2\lambda p_{uf} \tag{1}$$

$$\frac{\partial}{\partial q_{if}} = -2e_{ui} \cdot p_{uf} + 2\lambda q_{if} \tag{2}$$

其中 $e_{ui} = r_{ui} - \hat{r}_{ui}$ 为预测误差。

3. ClassicalSVD 算法原理

ClassicalSVD 在 FunkSVD 基础上引入全局偏置项，更好地建模数据的整体趋势。
预测模型：

$$\hat{r}_{ui} = \mu + p_u^T q_i$$

其中 μ 为全局平均评分，反映数据集的整体评分水平。
优化目标：

$$\min_{P,Q,\mu} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - p_u^T q_i)^2 + \lambda(\|P\|_F^2 + \|Q\|_F^2)$$

理论优势：

- 全局偏置 μ 能够捕捉数据集的基准评分水平
- 减少了隐因子需要学习的信息量，提高收敛速度
- 对于评分分布不均匀的数据集表现更好

4. BiasSVD 算法原理

BiasSVD 进一步引入用户偏置和物品偏置，全面建模评分的个性化特征。
预测模型：

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$$

其中:

- μ 为全局平均评分
- b_u 为用户 u 的偏置，反映用户的评分倾向（严格/宽松）
- b_i 为物品 i 的偏置，反映物品的受欢迎程度
- $p_u^T q_i$ 为用户-物品的个性化交互

偏置项的物理意义：

- **全局偏置** μ：所有用户对所有物品的平均评分，反映评分系统的基准水平
- **用户偏置** b_u：用户相对于平均水平的评分习惯偏差
  - $b_u > 0$：该用户倾向于给出高评分（宽松型用户）
  - $b_u < 0$：该用户倾向于给出低评分（严格型用户）
- **物品偏置** b_i：物品相对于平均水平的质量偏差
  - $b_i > 0$：该物品普遍受到好评（高质量物品）
  - $b_i < 0$：该物品普遍评分较低（低质量物品）

优化目标：

$$\min_{P,Q,b_u,b_i,\mu} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u^T q_i)^2 + \lambda(\|P\|_F^2 + \|Q\|_F^2 + \sum_u b_u^2 + \sum_i b_i^2)$$

5. SVDattr 算法原理

SVDattr 在 BiasSVD 基础上融入物品属性信息，实现内容感知的协同过滤。

属性建模：设物品 i 具有属性集合 $A_i = \{a_1, a_2, \ldots, a_{|A_i|}\}$，每个属性 a 对应一个属性嵌入向量 $e_a \in \mathbb{R}^d$。

属性聚合：物品 i 的属性特征表示为：

$$\text{attr}_i = \frac{1}{|A_i|} \sum_{a\in A_i} e_a$$

属性融合：通过线性变换将属性特征投影到隐因子空间：

$$\text{attr\_factors}_i = W \cdot \text{attr}_i$$

其中 $W \in \mathbb{R}^{k\times d}$ 为属性融合矩阵。

增强预测模型：

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T(q_i + \text{attr\_factors}_i)$$

理论优势：

- 缓解数据稀疏性问题，特别是对新物品的推荐
- 提供内容和协同过滤的融合建模
- 增强模型的可解释性和泛化能力

(六) NeuralCF

神经协同过滤（Neural Collaborative Filtering, NeuralCF）是深度学习在推荐系统中的重要应用，旨在克服传统矩阵分解方法仅能建模线性关系的局限性。

1. 传统协同过滤的局限性

线性假设的局限：传统矩阵分解假设用户-物品交互可以表示为隐因子的内积：

$$\hat{r}_{ui} = p_u^T q_i$$

这种内积操作限制了模型只能学习用户和物品隐因子之间的线性关系，无法捕捉复杂的非线性交互模式。

表达能力不足：

- 内积操作是对称的，即 $p_u^T q_i = q_i^T p_u$
- 无法建模用户偏好的复杂性和异质性
- 难以处理用户行为的上下文信息和时序特征

2. NeuralCF 架构设计

NeuralCF 通过神经网络架构同时建模线性和非线性用户-物品交互，主要包含两个分支：

广义矩阵分解（GMF）分支： GMF 是传统矩阵分解的神经网络实现：

$$\phi_{GMF}(p_u, q_i) = p_u \odot q_i$$

其中 $\odot$ 表示元素级乘积（Hadamard 乘积）。

多层感知机（MLP）分支： MLP 分支通过深度神经网络学习非线性交互：

$$\phi_{MLP}(p_u, q_i) = \text{MLP}([p_u; q_i])$$

其中 $[p_u; q_i]$ 表示用户和物品嵌入的拼接。

3. 多层感知机的非线性建模

网络结构： MLP 分支采用多层全连接网络：

$$z_1 = \phi_1([p_u; q_i]) = \sigma_1(W_1[p_u; q_i] + b_1) \tag{3}$$
$$z_2 = \phi_2(z_1) = \sigma_2(W_2 z_1 + b_2) \tag{4}$$
$$\vdots \tag{5}$$
$$z_L = \phi_L(z_{L-1}) = \sigma_L(W_L z_{L-1} + b_L) \tag{6}$$

其中：

- W_l, b_l 为第 l 层的权重矩阵和偏置向量
- σ_l 为第 l 层的激活函数（通常为 ReLU）
- L 为网络层数

非线性表达能力：

- 多层非线性变换能够学习复杂的用户-物品交互模式
- 相比内积操作，MLP 可以学习任意的用户-物品交互函数
- 网络深度和宽度控制模型的表达能力

4. 特征融合与预测

NeuMF 融合策略： NeuralCF 通过拼接 GMF 和 MLP 的输出进行特征融合：

$$\hat{r}_{ui} = \sigma(h^T[\phi_{GMF}(p_u^G, q_i^G); \phi_{MLP}(p_u^M, q_i^M)])$$

其中：

- p_u^G, q_i^G 为 GMF 分支的用户和物品嵌入
- p_u^M, q_i^M 为 MLP 分支的用户和物品嵌入
- h 为最终预测层的权重向量
- σ 为 sigmoid 激活函数（用于二分类）或线性激活（用于评分预测）

独立嵌入的设计理念：

- GMF 和 MLP 分支使用独立的嵌入层，增强模型灵活性
- 允许两个分支学习不同粒度的特征表示
- 避免线性和非线性建模之间的参数冲突

5. 优化目标与训练

点对点损失函数：对于评分预测任务，采用均方误差损失：

$$\mathcal{L} = \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \hat{r}_{ui})^2 + \lambda\|\Theta\|_2^2$$

负采样策略：对于隐式反馈数据，采用负采样方法：

- 正样本：用户实际交互的物品，标签为 1
- 负样本：随机采样用户未交互的物品，标签为 0
- 负采样比例通常设为 1:1 到 1:4

(七) AutoInt

AutoInt（Automatic Feature Interaction Learning）模型将 Transformer 架构中的多头自注意力机制引入推荐系统，实现特征交互的自动学习，无需人工设计特征组合。

1. 特征交互学习的重要性

特征交互在推荐中的作用：

- **一阶特征**：单个特征的独立影响（如用户年龄、物品类别）
- **二阶交互**：两个特征的组合效应（如年轻用户喜欢动作电影）
- **高阶交互**：多个特征的复杂组合模式

传统方法的局限：

- 人工特征工程成本高，难以穷举所有有效组合
- 预定义的交互方式（如内积、外积）表达能力有限
- 高阶特征交互的计算复杂度呈指数增长

2. 自注意力机制原理

注意力机制的核心思想：自注意力机制通过计算序列中每个元素与所有元素的相关性，实现动态的特征权重分配。

数学表述：对于输入序列 $X = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^{n \times d}$，自注意力计算为：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

其中：

- $Q = XW_Q$ 为查询矩阵，$W_Q \in \mathbb{R}^{d \times d_k}$
- $K = XW_K$ 为键矩阵，$W_K \in \mathbb{R}^{d \times d_k}$
- $V = XW_V$ 为值矩阵，$W_V \in \mathbb{R}^{d \times d_v}$
- d_k, d_v 为查询/键和值的维度

注意力权重的解释：

$$\alpha_{ij} = \frac{\exp(q_i^T k_j / \sqrt{d_k})}{\sum_{l=1}^{n} \exp(q_i^T k_l / \sqrt{d_k})}$$

α_{ij} 表示特征 i 对特征 j 的注意力权重，反映了两个特征之间的相关性强度。

3. 多头注意力机制

多头设计的动机：单个注意力头可能关注特定类型的特征关系，多头机制允许模型同时关注不同的交互模式。

多头注意力计算：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

其中：

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

参数矩阵：

- $W_i^Q, W_i^K \in \mathbb{R}^{d \times d_k}$：第 i 个头的查询和键投影矩阵
- $W_i^V \in \mathbb{R}^{d \times d_v}$：第 i 个头的值投影矩阵
- $W^O \in \mathbb{R}^{hd_v \times d}$：输出投影矩阵
- 通常设置 $d_k = d_v = d/h$，其中 h 为头数

4. AutoInt 在推荐中的应用

特征序列构建：将用户和物品的嵌入表示构成长度为 2 的序列：

$$X = [\text{user\_emb}, \text{item\_emb}] \in \mathbb{R}^{2 \times d}$$

自动交互学习：通过自注意力机制，模型能够自动学习：

- 用户特征内部的关联性
- 物品特征内部的关联性

- 用户和物品特征之间的交互关系

层次化特征学习：通过多层自注意力堆叠，模型能够学习不同层次的特征交互：

$$X^{(0)} = [\text{user\_emb}, \text{item\_emb}] \tag{7}$$

$$X^{(l)} = \text{LayerNorm}(X^{(l-1)} + \text{MultiHead}(X^{(l-1)})) \tag{8}$$

$$\hat{r}_{ui} = \text{MLP}(\text{Concat}(X^{(L)})) \tag{9}$$

其中 L 为注意力层数。

5. 残差连接与层标准化

残差连接（Residual Connection）：

$$X^{(l)} = X^{(l-1)} + \text{MultiHead}(X^{(l-1)})$$

残差连接的作用：

- 缓解梯度消失问题，支持更深的网络训练
- 保持低层特征信息，避免信息丢失
- 加速网络收敛

层标准化（Layer Normalization）：

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \odot \gamma + \beta$$

其中：

- μ, σ 为输入的均值和标准差
- γ, β 为可学习的缩放和平移参数

层标准化的优势：

- 稳定训练过程，减少内部协变量偏移
- 加速收敛速度
- 减少对初始化的敏感性

6. AutoInt 的理论优势

自动特征发现：

- 无需人工设计特征交互，模型自动学习有效的特征组合
- 注意力权重提供特征重要性的解释
- 适应性强，能够处理不同类型和规模的特征

高阶交互建模：

- 多层注意力能够捕捉任意阶的特征交互

- 相比传统方法，计算复杂度增长更为缓慢
- 能够处理稀疏特征和连续特征的混合输入

模型可解释性：

- 注意力权重矩阵提供特征交互的可视化
- 能够识别对预测最重要的特征组合
- 有助于理解模型的决策过程

计算效率：

- 注意力计算可以并行化，训练效率较高
- 相比穷举所有特征组合，计算复杂度显著降低
- 支持 GPU 加速，适合大规模数据处理

四、 核心代码分析

(一) UserCF

用户协同过滤主要通过计算用户之间的相似度，利用相似用户的评分来预测目标用户的评分

1. 构建用户评分矩阵

- 创建用户和物品的映射字典，将用户和物品的 ID 映射为矩阵的索引。这一步通过遍历训练数据，提取所有用户和物品的唯一 ID，并为它们分配一个唯一的索引。例如，`user_to_idx` 字典将用户 ID 映射为矩阵的行索引，`item_to_idx` 字典将物品 ID 映射为矩阵的列索引。
- 准备构建稀疏评分矩阵所需的数据，包括评分值、用户索引和物品索引。这些数据将用于构建稀疏矩阵，存储用户对物品的评分信息。例如，`values` 存储评分值，`rows` 和 `cols` 分别存储对应的用户索引和物品索引。
- 使用准备好的数据构建稀疏评分矩阵。稀疏矩阵是一种高效存储大规模稀疏数据的数据结构，通过存储非零元素的值（`values`）、行索引（`rows`）和列索引（`cols`），以及矩阵的形状（用户数量和物品数量）来构建。这种方式可以节省大量内存，尤其适用于用户和物品数量庞大的场景。构建完成后，`rating_matrix` 包含了所有用户对物品的评分信息。

```python
def process_rating_matrix(train_data, batch_size=5000):
    def create_mapping(data_dict, key_type='user'):
        if key_type == 'user':
            keys = list(data_dict.keys())
        else:  # item
            keys = list({item for ratings in data_dict.values() for item, _
                in ratings})
        return {key: idx for idx, key in enumerate(keys)}

    def prepare_matrix_data(data_dict, user_map, item_map):
        total_entries = sum(len(ratings) for ratings in data_dict.values())
        values = np.zeros(total_entries)
        rows = np.zeros(total_entries, dtype=int)
        cols = np.zeros(total_entries, dtype=int)

        current_idx = 0
        for user, ratings in data_dict.items():
            user_idx = user_map[user]
            for item, score in ratings:
                values[current_idx] = score
                rows[current_idx] = user_idx
                cols[current_idx] = item_map[item]
                current_idx += 1

        return values, rows, cols

    # 创建用户和物品的映射
    user_to_idx = create_mapping(train_data, 'user')
```

```
    item_to_idx = create_mapping(train_data, 'item')

    # 准备矩阵数据
    matrix_values, row_indices, col_indices = prepare_matrix_data(
        train_data, user_to_idx, item_to_idx
    )

    # 构建稀疏矩阵
    rating_matrix = csr_matrix(
        (matrix_values, (row_indices, col_indices)),
        shape=(len(user_to_idx), len(item_to_idx))
    )
```

2. 计算用户之间的相似度

- 计算用户之间的相似度矩阵，这是用户协同过滤算法的核心步骤，用于后续的评分预测。具体过程如下：
  - 使用余弦相似度计算用户之间的相似度。余弦相似度通过计算两个用户评分向量之间的夹角余弦值来衡量它们的相似性。值越接近 1，表示两个用户越相似。例如，对于用户 u 和用户 v，它们的相似度 $\text{sim}(u, v)$ 可以表示为：

$$\text{sim}(u, v) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

    其中，$\mathbf{u}$ 和 $\mathbf{v}$ 分别是用户 u 和 v 的评分向量。
  - 分批计算相似度矩阵，以避免一次性计算大规模相似度矩阵导致内存不足。每次只计算一部分用户与其他所有用户的相似度，然后将结果存储到相似度矩阵中。例如，`similarity_matrix` 是一个大小为 $n \times n$ 的矩阵，其中 n 是用户数量。每次计算一个批次的用户与其他所有用户的相似度，并将结果存储到 `similarity_matrix` 中。
- 分批计算用户之间的相似度矩阵，以避免内存不足的问题。具体步骤如下：
  - 遍历用户，按照指定的批次大小（`batch_size`）分批次处理用户。例如，用户总数为 n，批次大小为 `batch_size`，则每次处理 `batch_size` 个用户。
  - 对于每个批次，计算从当前批次起始索引（`batch_start`）到结束索引（`batch_end`）的用户与其他所有用户的相似度。例如，对于当前批次的用户索引范围 [`batch_start`, `batch_end`)，计算这些用户与其他所有用户的相似度。
  - 将计算得到的相似度矩阵存储到最终的相似度矩阵（`similarity_matrix`）中，直到所有用户都被处理完毕。例如，将每个批次计算得到的相似度矩阵存储到 `similarity_matrix` 的对应位置，直到所有用户都被处理完毕。

```
def compute_batch_similarity(matrix, start_pos, end_pos):
        return cosine_similarity(matrix[start_pos:end_pos], matrix)
def create_empty_similarity_matrix(size):
        return np.zeros((size, size))

# 获取矩阵维度
```

```
    matrix_size = rating_matrix.shape[0]

    # 创建空的相似度矩阵
    similarity_matrix = create_empty_similarity_matrix(matrix_size)

    # 分批计算相似度
    for batch_start in tqdm(range(0, matrix_size, batch_size),
                            desc="计算用户相似度",
                            mininterval=0.1,
                            initial=0,
                            bar_format='{desc}: {percentage:3.0f}%|{bar}| {
                                n_fmt}/{total_fmt} [{elapsed}<{remaining}]'):
        batch_end = min(batch_start + batch_size, matrix_size)
        batch_similarity = compute_batch_similarity(rating_matrix,
            batch_start, batch_end)
        similarity_matrix[batch_start:batch_end] = batch_similarity

    return rating_matrix, similarity_matrix, user_to_idx, item_to_idx
```

3. 用户评分预测

- **对每个用户进行预测**
  - 遍历测试数据中的每个用户，跳过不在 `user_map` 中的用户。对于每个用户，获取其在评分矩阵中的索引（`user_idx`），并从相似度矩阵中提取该用户与其他所有用户的相似度（`user_similarities`）。这一步是为了确定哪些用户与当前用户相似，以便后续使用这些相似用户的评分来预测当前用户的评分。
- **获取最相似的 `top_k` 个用户**
  - 从相似度矩阵中获取与当前用户最相似的 `top_k` 个用户。具体过程如下：
    - 使用 `np.argsort` 对相似度（`user_similarities`）进行排序，得到从高到低的索引。
    - $[1 : \texttt{top\_k} + 1]$ 跳过当前用户（相似度最高的用户是自己），取前 `top_k` 个用户。这样可以确保只考虑与当前用户最相似的其他用户，而不是包括当前用户自己。
- **获取这些用户的评分**
  - 获取最相似的 `top_k` 个用户的评分矩阵（`similar_users_ratings`）。这一步是为了获取这些相似用户对所有物品的评分，以便后续计算当前用户对特定物品的预测评分。
- **计算预测分数**

  对当前用户需要预测评分的每个物品，计算预测评分。具体过程如下：
  - 遍历当前用户需要预测评分的每个物品（`item`）。
  - 如果物品不在 `item_map` 中，则跳过该物品。
  - 获取物品在评分矩阵中的索引（`item_idx`）。

- 提取最相似的 top_k 个用户对当前物品的评分（item_ratings），并将其转换为一维数组。
- 过滤掉未评分的用户（评分值为零的用户），得到有效的评分（valid_ratings）。
- 如果存在有效的评分,则计算这些评分的平均值作为当前物品的预测评分(predicted_score)。
- 将预测评分存储到 predicted_scores 列表中,每个元素是一个元组 (item, predicted_score)。

通过以上步骤,可以为当前用户预测其对每个物品的评分,并将这些预测评分存储到 predicted_scores 列表中。最终，将所有用户的预测评分结果存储到 predictions 字典中。

```
# 进行用户评分预测
def predict_user_ratings(similarity_matrix, rating_matrix, user_map, item_map, test_data, top_k=500):
    predictions = defaultdict(list)

    # 获取所有用户和物品的ID列表
    user_ids = list(user_map.keys())
    item_ids = list(item_map.keys())

    # 对每个用户进行预测
    for user in tqdm(test_data.keys(),
                     desc="预测用户评分",
                     mininterval=0.1,
                     initial=0,
                     bar_format='{desc}: {percentage:3.0f}%|{bar}| {n_fmt}/{total_fmt} [{elapsed}<{remaining}]'):
        if user not in user_map:
            continue

        user_idx = user_map[user]
        user_similarities = similarity_matrix[user_idx]

        # 获取最相似的top_k个用户
        similar_users = np.argsort(user_similarities)[::-1][1:top_k+1]

        # 获取这些用户的评分
        similar_users_ratings = rating_matrix[similar_users]

        # 计算预测分数
        predicted_scores = []
        for item in test_data[user]:
            if item not in item_map:
                continue

            item_idx = item_map[item]
            item_ratings = similar_users_ratings[:, item_idx].toarray().flatten()

```

```
            # 只考虑有评分的用户
            valid_ratings = item_ratings[item_ratings > 0]
            if len(valid_ratings) > 0:
                predicted_score = np.mean(valid_ratings)
                predicted_scores.append((item, predicted_score))

        # 按预测分数排序
        predicted_scores.sort(key=lambda x: x[1], reverse=True)
        predictions[user] = predicted_scores

    return predictions
```

(二) ItemCF

1. 数据处理流程

`read_train_full(filename)` 用于读取训练集文件，并构建用户-物品评分矩阵和物品-用户评分矩阵，同时返回评分三元组列表。

```
from collections import defaultdict

def read_train_full(filename):
    user_items = defaultdict(dict)  # 用户–物品评分矩阵
    item_users = defaultdict(dict)  # 物品–用户评分矩阵
    ratings = []  # 评分三元组列表

    with open(filename, 'r', encoding='utf-8') as f:
        while True:
            head = f.readline()  # 读取用户信息行
            if not head:  # 如果文件结束，退出循环
                break
            user_id, num = head.strip().split('|')  # 解析用户ID和评分数量
            num = int(num)  # 转换为整数
            for _ in range(num):  # 遍历该用户的评分记录
                line = f.readline()  # 读取评分信息行
                if not line:  # 如果文件结束，退出循环
                    break
                item_id, score = line.strip().split()  # 解析物品ID和评分
                score = float(score)  # 转换为浮点数
                user_items[user_id][item_id] = score  # 构建用户–物品评分矩阵
                item_users[item_id][user_id] = score  # 构建物品–用户评分矩阵
                ratings.append((user_id, item_id, score))  # 添加到评分三元组
                    列表
    return user_items, item_users, ratings
```

`read_test(filename)` 用于读取测试集文件，并解析用户需要预测评分的物品列表。

```
def read_test(filename):
    test_data = []  # 测试数据列表

```

```
    with open(filename, 'r', encoding='utf-8') as f:
        while True:
            head = f.readline() # 读取用户信息行
            if not head: # 如果文件结束，退出循环
                break
            user_id, num = head.strip().split('|') # 解析用户ID和预测任务数量
            num = int(num) # 转换为整数
            items = [] # 当前用户需要预测的物品列表
            for _ in range(num): # 遍历该用户的预测任务
                line = f.readline() # 读取物品信息行
                if not line: # 如果文件结束，退出循环
                    break
                item_id = line.strip() # 解析物品ID
                items.append(item_id) # 添加到物品列表
            test_data.append((user_id, items)) # 添加到测试数据列表
    return test_data
```

2. 余弦相似度计算

余弦相似度是一种常用的相似性度量方法，用于衡量两个物品的用户评分向量之间的相似性。范围为 [0, 1]。如果两个物品没有共同的用户评分，则返回 0.0。

具体计算流程如下：

1. 提取两个物品的用户评分字典：`users1` 和 `users2` 分别是 `item1` 和 `item2` 的用户评分字典，格式为 `{user_id: score}`。
2. 找到两个物品的共同用户：使用集合操作 `&` 找到两个物品的共同用户。如果没有共同用户，返回 0.0，因为无法计算相似度。
3. 提取共同用户的评分向量：遍历共同用户，提取每个用户对 `item1` 和 `item2` 的评分，分别存储到 `v1` 和 `v2` 中。
4. 检查向量是否为零向量：使用 `np.linalg.norm` 计算向量的范数（即向量的长度）。如果任一向量的范数为 0，说明该向量是零向量，无法计算余弦相似度，因此返回 0.0。
5. 计算余弦相似度：

   余弦相似度公式为：

$$\text{cosine similarity} = \frac{\mathbf{v1} \cdot \mathbf{v2}}{\|\mathbf{v1}\| \times \|\mathbf{v2}\|}$$

```
def cosine_similarity(item_users, item1, item2):
    users1 = item_users[item1]
    users2 = item_users[item2]
    common_users = set(users1.keys()) & set(users2.keys())
    if not common_users:
        return 0.0
    v1 = np.array([users1[u] for u in common_users])
    v2 = np.array([users2[u] for u in common_users])
    if np.linalg.norm(v1) == 0 or np.linalg.norm(v2) == 0:
```

```
        return 0.0
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
```

3. 预测

具体步骤如下:

1. **处理冷启动用户和冷启动物品**。
   - 如果用户是冷启动用户（即用户不在 `user_items` 中），无法找到该用户的评分信息。
     - 如果目标物品在 `item_users` 中，返回该物品的平均评分。
     - 如果目标物品也不在 `item_users` 中，返回默认评分（这里假设为 70.0）。
   - 如果物品是冷启动物品（即物品不在 `item_users` 中），无法找到该物品的评分信息。
     - 返回该用户的平均评分。
2. **计算与目标物品相似的物品**。
   - 遍历该用户评分过的所有物品（`user_items[user_id]`）。
   - 跳过目标物品本身（`neighbor == item_id`）。
   - 跳过与目标物品没有共同评分用户的物品。
   - 计算目标物品与当前物品的相似度（使用 `cosine_similarity` 函数）。
   - 如果相似度高于阈值（`sim_threshold`），将该物品及其相似度添加到列表 `sims` 中。
3. **如果存在高相似度的邻居物品，使用加权平均公式计算预测评分**。
   - 如果存在高相似度的邻居物品（`sims` 不为空）:
     - 按相似度从高到低排序，并取前 `k` 个最相似的物品。
     - 使用加权平均公式计算预测评分:

$$\text{predicted rating} = \frac{\sum(\text{similarity} \times \text{rating})}{\sum|\text{similarity}|}$$

     - 如果分母不为零，返回计算结果。
4. **如果没有高相似度的邻居物品，遍历所有物品，找与目标物品有共同评分用户的前 K 个最相似物品**。
   - 如果没有高相似度的邻居物品（`sims` 为空）:
     - 遍历所有物品，计算目标物品与每个物品的相似度。
     - 跳过目标物品本身（`neighbor == item_id`）。
     - 跳过用户未评分的物品。
     - 跳过与目标物品没有共同评分用户的物品。
     - 计算相似度并存储到列表 `sim_all` 中。
     - 按相似度从高到低排序，并取前 `k` 个最相似的物品。
     - 使用加权平均公式计算预测评分。

5. **如果仍然无法计算出预测评分，返回该用户的平均评分。**

```
def predict(user_id, item_id, user_items, item_users, k=20, sim_threshold
    =0.5):
    # 冷启动用户
    if user_id not in user_items:
        if item_id in item_users:
            return np.mean(list(item_users[item_id].values()))
        else:
            return 70.0
    # 冷启动物品
    if item_id not in item_users:
        return np.mean(list(user_items[user_id].values()))
    # 只用与目标物品有共同评分用户的物品计算相似度
    sims = []
    for neighbor in user_items[user_id]:
        if neighbor == item_id:
            continue
        # 只考虑有共同评分用户的物品
        if len(set(item_users[item_id].keys()) & set(item_users[neighbor].
            keys())) == 0:
            continue
        sim = cosine_similarity(item_users, item_id, neighbor)
        if sim > sim_threshold:
            sims.append((neighbor, sim))
    # 如果有高相似度邻居，使用它们
    if sims:
        sims = sorted(sims, key=lambda x: x[1], reverse=True)[:k]
        numerator, denominator = 0.0, 0.0
        for neighbor, sim in sims:
            numerator += sim * user_items[user_id][neighbor]
            denominator += abs(sim)
        if denominator > 0:
            return numerator / denominator
    # 否则，遍历所有物品，找与目标物品有共同评分用户的前K个最相似物品
    sim_all = []
    for neighbor in item_users:
        if neighbor == item_id:
            continue
        if user_id not in user_items or neighbor not in user_items[user_id]:
            continue
        if len(set(item_users[item_id].keys()) & set(item_users[neighbor].
            keys())) == 0:
            continue
        sim = cosine_similarity(item_users, item_id, neighbor)
        sim_all.append((neighbor, sim))
    sim_all = sorted(sim_all, key=lambda x: x[1], reverse=True)[:k]
    numerator, denominator = 0.0, 0.0
```

```
    for neighbor, sim in sim_all:
        numerator += sim * user_items[user_id][neighbor]
        denominator += abs(sim)
    if denominator > 0:
        return numerator / denominator
    # 最后兜底：用用户均值
    return np.mean(list(user_items[user_id].values()))
```

(三) MLP 网络

1. 数据处理流程

`read_data(file_path, is_train=True)` 用于读取数据文件，将文件内容解析为一个字典，键为用户 ID，值为用户评分的物品列表（训练数据）或用户需要预测的物品列表（测试数据）。

1. **打开文件并读取所有行**:

2. **逐行解析数据**:

```
while i < len(lines):
    user_info = lines[i].strip().split('|')
    user_id = int(user_info[0])
    item_count = int(user_info[1])
    i += 1
```

每行的格式为 `user_id|item_count`，表示用户 ID 和该用户评分的物品数量。
使用 `strip` 去除行首尾的空白字符，然后用 `split('|')` 按 `|` 分割。
将用户 ID 和物品数量转换为整数。

3. **解析物品信息**:

- 如果是训练数据:

```
items = [list(map(int, lines[i+j].strip().split())) for j in range(item_count
    )]
```

每个用户有 `item_count` 行物品信息，每行格式为 `item_id score`。
使用列表推导式，将每行的物品 ID 和评分转换为整数，并存储为二维列表。

- 如果是测试数据:

```
items = [int(lines[i+j]) for j in range(item_count)]
```

每个用户有 `item_count` 行物品 ID，每行只有一个物品 ID。
使用列表推导式，将每行的物品 ID 转换为整数，并存储为一维列表。

4. **更新索引并存储数据**:

```
i += item_count
data[user_id] = items
```

更新索引 `i`，跳过已经处理的物品行。
将用户 ID 和对应的物品信息存储到字典 `data` 中。

`process_train_data(data_dict)` **将训练数据字典转换为一个列表，每个元素为** `[user_id, item_id, score]`。

`process_test_data(data_dict)` **将测试数据字典转换为一个列表，每个元素为** `[user_id, item_id]`。

数据划分策略：

`split_train_test(raw_data, test_size=0.1, random_state=2)` **使用** `sklearn.model_selection.train_test_split` **方法随机划分数据。**

`split_train_test2(raw_data, random_state=2)` **按用户划分数据，确保每个用户至少有一个样本在验证集中。**

```
def split_train_test(raw_data, test_size=0.1, random_state=2):
    return train_test_split(raw_data, test_size=test_size, random_state=
        random_state)

def split_train_test2(raw_data, random_state=2):
    random.seed(random_state)
    user_records = defaultdict(list)
    for row in raw_data:
        user_records[row[0]].append(row)
    train_data, val_data = [], []
    for records in user_records.values():
        if len(records) == 1:
            train_data.extend(records)
        else:
            idx = random.randint(0, len(records)-1)
            val_data.append(records[idx])
            train_data.extend([r for i, r in enumerate(records) if i != idx])
    return train_data, val_data
```

1. **初始化随机种子**：使用 `random.seed` 设置，确保结果的可重复性。
2. **按用户分组**：使用 `defaultdict` 创建一个字典，键为用户 ID，值为该用户的所有记录。遍历原始数据，将每个记录按用户 ID 分组。
3. **划分训练集和验证集**：遍历每个用户的记录列表：
   - 如果该用户只有一个记录，则将该记录分配到训练集。
   - 如果该用户有多个记录：
     - 随机选择一个记录作为验证集。
     - 其余记录分配到训练集。

2. MLP 网络构建

我们构建了如下的 MLP 网络，通过嵌入层将用户和物品映射到低维空间，再通过多层全连接层学习用户和物品之间的复杂关系，最终输出评分预测。

```
class MLP(nn.Module):
    def __init__(self, n_user, d_user, n_item, d_item, dropout=0.4):
        super().__init__()
```

```
        self.embedding_user = nn.Embedding(n_user, d_user)
        self.embedding_item = nn.Embedding(n_item, d_item)
        self.fc_layers = nn.Sequential(
            nn.Linear(d_user + d_item, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.1),
            nn.Dropout(dropout),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.LeakyReLU(0.1),
            nn.Dropout(dropout),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.LeakyReLU(0.1),
            nn.Dropout(dropout),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
    def forward(self, user_ids, item_ids):
        vec_user = self.embedding_user(user_ids)
        vec_item = self.embedding_item(item_ids)
        x = torch.cat((vec_user, vec_item), 1)
        return self.fc_layers(x) * 100
```

1. **嵌入层**：将离散的用户 ID 和物品 ID 映射到低维的连续向量空间，使得模型能够捕捉到用户和物品的内在特征。

   - `nn.Embedding` 是 PyTorch 提供的嵌入层，用于将离散的用户 ID 和物品 ID 映射到低维的连续向量空间。
   - `n_user` 和 `n_item` 分别是用户和物品的总数，表示嵌入层的输入维度。
   - `d_user` 和 `d_item` 分别是用户和物品嵌入的维度，表示嵌入层的输出维度。

2. **全连接层**：通过多层全连接层，模型可以学习到用户和物品特征之间的复杂非线性关系。

   - `nn.Sequential` 是一个容器，用于将多个层按顺序堆叠起来。
   - `nn.Linear` 是全连接层，用于将输入向量转换为更高维度的向量。
     - 第一层的输入维度是 `d_user + d_item`，因为用户嵌入向量和物品嵌入向量会被拼接在一起。
     - 每一层的输出维度分别是 256、128、64 和 1。
   - `nn.BatchNorm1d` 是批量归一化层，用于对每一层的输出进行归一化处理，加速训练过程并提高模型的稳定性。
   - `nn.LeakyReLU` 是激活函数，用于引入非线性。它的参数 `0.1` 表示负半轴的斜率。
   - `nn.Dropout` 是 Dropout 层，用于防止过拟合。它的参数 `dropout` 表示丢弃的比例。
   - `nn.Sigmoid` 是 Sigmoid 激活函数，将输出值缩放到 [0, 1] 的范围内。最后再乘以 100，将评分范围调整为 [0, 100]。

3. **前向传播函数 `forward`：**定义输入数据如何通过网络进行计算，最终得到输出结果。它接收用户 ID 和物品 ID 的张量。
4. 使用嵌入层将用户 ID 和物品 ID 分别映射到低维向量空间，得到用户嵌入向量 `vec_user` 和物品嵌入向量 `vec_item`。
5. **向量拼接**：
   - 使用 `torch.cat` 将用户嵌入向量和物品嵌入向量按列拼接在一起，形成一个新的输入向量 `x`。这个向量包含了用户和物品的信息，将作为后续全连接层的输入。
6. **通过全连接层序列**：
   - 将拼接后的向量 `x` 输入到全连接层序列 `self.fc_layers` 中，经过多层计算后得到输出。
   - 输出值经过 `Sigmoid` 激活函数缩放到 [0, 1]，再乘以 100，将评分范围调整为 [0, 100]。
7. **激活函数和归一化**：激活函数（如 `LeakyReLU`）引入非线性，使模型能够捕捉复杂的模式；批量归一化（`BatchNorm1d`）加速训练并提高模型的稳定性。
8. **Dropout**：防止过拟合，提高模型的泛化能力。
9. **输出层**：最终输出一个评分值，范围为 [0, 100]。

3. 最佳参数选择

```
lr_list = [0.01, 0.001, 0.0005]
epoch_list = [5,10]
dropout_list = [0.2,0.4]
d_user_list = [64, 100, 128]
d_item_list = [64, 100, 128]
```

定义了超参数的候选列表，使用找到的最佳超参数组合重新训练模型。

- `lr_list`：学习率的候选值。
- `epoch_list`：训练轮数的候选值。
- `dropout_list`：Dropout 比例的候选值。
- `d_user_list` 和 `d_item_list`：用户和物品嵌入维度的候选值。

最终得到的最佳参数组合为：

```
Best params:
lr=0.01, epoch=5, dropout=0.4, d_user=128, d_item=64, val RMSE=17.7169
```

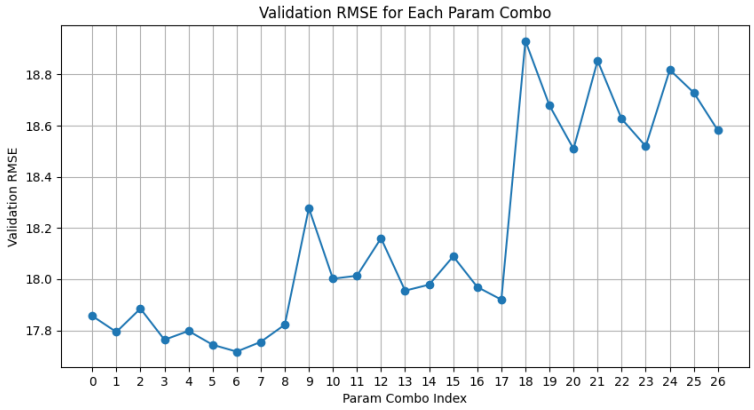


图 1: param_search_rmse

4. 训练验证架构

定义损失函数和优化器

```
loss_func = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-5)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)
```

- 使用均方误差（MSE）作为损失函数。
- 使用 Adam 优化器，学习率为 `lr`，并添加权重衰减（`weight_decay=1e-5`）以防止过拟合。
- 使用 `ReduceLROnPlateau` 学习率调度器，当验证集损失不再下降时，降低学习率。

批量处理数据

```
total_loss, total_mse, total_mae, all_labels, all_preds = 0, 0, 0, [], []
n_batches = (len(data_) + batch_size - 1) // batch_size
for i in range(0, len(data_), batch_size):
    batch = np.array(data_[i:i+batch_size])
    users = torch.tensor(batch[:,0], dtype=torch.long, device=device)
    items = torch.tensor(batch[:,1], dtype=torch.long, device=device)
    ratings = torch.tensor(batch[:,2], dtype=torch.float, device=device)
    optimizer.zero_grad()
    with torch.set_grad_enabled(phase=='Train'):
        outputs = model(users, items).squeeze()
        loss = loss_func(outputs, ratings)
        if phase=='Train':
            loss.backward()
            optimizer.step()
```

- 遍历每个批量：
  - 提取用户 ID、物品 ID 和评分，将这些数据转换为张量，并移动到指定设备。
  - 清空优化器的梯度。

– 前向传播，计算模型的输出和损失。

– 如果是训练阶段，反向传播并更新模型参数。

计算性能指标

```
preds = outputs.cpu().detach().numpy()
labs = ratings.cpu().numpy()
all_labels.extend(labs)
all_preds.extend(preds)
total_loss += loss.item()
total_mse += mean_squared_error(labs, preds)
total_mae += mean_absolute_error(labs, preds)
avg_loss = total_loss/(n_batches)
avg_rmse = math.sqrt(total_mse/(n_batches))
avg_mae = total_mae/(n_batches)
r2 = r2_score(all_labels, all_preds) if len(set(all_labels)) > 1 else float('
    nan')
mape = np.mean(np.abs((np.array(all_labels) - np.array(all_preds)) / (np.
    array(all_labels) + 1e-8))) * 100
```

- 将模型的输出和真实标签转换为 NumPy 数组，记录所有标签和预测值。
- 累加损失、MSE 和 MAE。

记录训练历史

```
if phase=='Train':
    history['train_loss'].append(avg_loss)
    history['train_rmse'].append(avg_rmse)
    history['train_mae'].append(avg_mae)
    history['train_r2'].append(r2)
    history['train_mape'].append(mape)
else:
    history['valid_loss'].append(avg_loss)
    history['valid_rmse'].append(avg_rmse)
    history['valid_mae'].append(avg_mae)
    history['valid_r2'].append(r2)
    history['valid_mape'].append(mape)
    scheduler.step(avg_loss)
    if avg_rmse < best_rmse:
        best_rmse = avg_rmse
        counter = 0
        torch.save(model.state_dict(), f"{model_dir}/ckpt.model")
        if verbose:
            print(f'Saving model with RMSE {avg_rmse:.3f}')
    else:
        counter += 1
    if counter >= patience:
        if verbose:
            print('Early stopping!')
```

```
25            break
```

- 如果是训练阶段，记录训练指标；如果是验证阶段，记录验证指标。
- 使用学习率调度器调整学习率。
- 如果当前验证集 RMSE 是最佳值，保存模型，并重置早停计数器。
- 如果连续多个轮数验证集 RMSE 没有改善，触发早停机制。

`predict_test` 使用训练好的模型对测试数据进行评分预测，并将预测结果写入到指定的输出文件中。

- 逐行读取测试数据文件：
  - 如果行包含 `|`，表示这是一个新的用户记录的开始。首先处理之前的用户记录（如果有）：
    - 将用户 ID 和物品 ID 列表转换为张量，并移动到指定设备。
    - 使用模型进行预测，得到评分结果。
    - 将预测结果格式化为字符串并添加到 `results` 列表中。
    - 读取新的用户 ID，并重置 `users` 和 `items` 列表。
  - 如果行不包含 `|`，表示这是物品 ID，将其添加到 `items` 列表中。

```
with open(test_file, 'r') as f:
    for line in f:
        if '|' in line:
            if users:
                users_t = torch.tensor(users, dtype=torch.long).to(device)
                items_t = torch.tensor(items, dtype=torch.long).to(device)
                ratings = model(users_t, items_t).squeeze()
                results.extend(f'{items[i]}␣␣{ratings[i].item()}\n' for i in
                    range(len(ratings)))
            results.append(line)
            user_id = int(line.split('|')[0])
            users, items = [], []
        else:
            item_id = int(line.strip())
            users.append(user_id)
            items.append(item_id)
if users:
    users_t = torch.tensor(users, dtype=torch.long).to(device)
    items_t = torch.tensor(items, dtype=torch.long).to(device)
    ratings = model(users_t, items_t).squeeze()
    results.extend(f'{items[i]}␣␣{ratings[i].item()}\n' for i in range(len(
        ratings)))
```

(四) ALS 算法

1. 构建评分矩阵

我们需要根据训练数据构建评分矩阵。评分矩阵是一个二维矩阵，其行对应用户，列对应物品，矩阵中的值表示用户对物品的评分。在实际应用中，评分矩阵通常是稀疏的，因为每个用户只对少数物品进行了评分。构建评分矩阵的目的是为了将用户和物品之间的评分关系以矩阵的形式表示出来，这样便于后续的矩阵运算和模型优化。我们通过 train_data 构建了评分矩阵 rating_matrix，其中 train_data 是一个字典，键为用户 ID，值为用户评分的物品列表

```
def build_rating_matrix(train_data, user_to_index, item_to_index):
    """构建评分矩阵"""
    num_users = len(user_to_index)
    num_items = len(item_to_index)
    rating_matrix = np.zeros((num_users, num_items), dtype=np.float32)

    for user, user_items in train_data.items():
        for item, rating in user_items:
            rating_matrix[user_to_index[user], item_to_index[item]] = rating

    return rating_matrix
```

2. 更新因子矩阵

ALS 算法的核心思想是交替更新用户和物品的因子矩阵。具体来说，我们先固定物品的因子矩阵 item_matrix，更新用户的因子矩阵 user_matrix；然后固定用户的因子矩阵 user_matrix，更新物品的因子矩阵 item_matrix。这个过程会不断重复，直到模型收敛。在每次更新过程中，我们通过最小化预测评分与实际评分之间的误差来优化因子矩阵。这个优化过程涉及到求解一个最小二乘问题，通常使用线性代数的方法来解决。通过这种方式，我们可以逐步调整因子矩阵，使得模型能够更好地拟合训练数据。在更新因子矩阵时，我们使用了正则化参数 regularization 来防止过拟合，确保模型具有良好的泛化能力

```
def update_factors(matrix, other_matrix, rating_matrix, regularization,
    num_factors, is_user=True):
    """更新用户或物品因子矩阵"""
    num_entities = matrix.shape[0]
    for i in range(num_entities):
        if is_user:
            ratings = rating_matrix[i, :]
            non_zero = ratings.nonzero()[0]
            if len(non_zero) > 0:
                submatrix = other_matrix[non_zero, :]
                ratings_sub = ratings[non_zero]
        else:
            ratings = rating_matrix[:, i]
            non_zero = ratings.nonzero()[0]
            if len(non_zero) > 0:
                submatrix = other_matrix[non_zero, :]
                ratings_sub = ratings[non_zero]
```

```
17
18          if len(non_zero) > 0:
19              A = submatrix.T @ submatrix + regularization * np.eye(num_factors
                    )
20              b = submatrix.T @ ratings_sub
21              matrix[i, :] = np.linalg.solve(A, b)
22      return matrix
```

3. 模型训练

als 函数是实现交替最小二乘法（ALS）的核心函数，用于训练模型。首先对训练数据进行预处理，提取所有用户和物品的列表，并为每个用户和物品创建唯一的索引映射，以便在后续的矩阵操作中快速定位。

接着，函数调用 build_rating_matrix 构建评分矩阵，该矩阵以用户为行、物品为列，记录了用户对物品的评分信息。随后，初始化用户和物品的因子矩阵，这些矩阵将用于表示用户和物品在特征空间中的向量。在训练过程中，函数通过多次迭代交替更新用户和物品的因子矩阵。每次迭代中，先固定物品的因子矩阵，更新用户的因子矩阵；再固定用户的因子矩阵，更新物品的因子矩阵。这一过程通过最小化预测评分与实际评分之间的误差来优化因子矩阵。

每次更新后，使用均方根误差（RMSE）评估当前模型的性能，并记录下来。如果当前模型的 RMSE 比之前的最佳 RMSE 更低，则更新最佳 RMSE 并保存当前的因子矩阵。最后，函数返回最终的用户矩阵、物品矩阵以及用户和物品的索引映射字典，这些结果将用于后续的评分预测和推荐。

```
1  def als(train_data, num_factors=10, num_iterations=10, regularization=0.1):
2      """交替最小二乘法实现推荐系统"""
3      # 数据准备
4      users = sorted(train_data.keys())
5      items = sorted({item for user_items in train_data.values() for item, _ in
           user_items})
6
7      # 创建映射
8      user_to_index, item_to_index = create_mapping_dicts(users, items)
9
10     # 构建评分矩阵
11     rating_matrix = build_rating_matrix(train_data, user_to_index,
           item_to_index)
12
13     # 初始化
14     user_matrix, item_matrix = initialize_matrices(len(users), len(items),
           num_factors)
15
16     # 训练过程
17     history = []
18     best_rmse = float('inf')
19     best_matrices = None
20
21     for epoch in range(num_iterations):
22         # 更新因子
```

```
        user_matrix = update_factors(user_matrix, item_matrix, rating_matrix,
            regularization, num_factors, True)
        item_matrix = update_factors(item_matrix, user_matrix, rating_matrix,
            regularization, num_factors, False)

        # 评估
        current_rmse = evaluate_model(user_matrix, item_matrix, rating_matrix
            )
        history.append(current_rmse)

        # 保存最佳模型
        if current_rmse < best_rmse:
            best_rmse = current_rmse
            best_matrices = (user_matrix.copy(), item_matrix.copy())

        # 输出进度
        print(f"训练进度: {epoch + 1}/{num_iterations} | 当前RMSE: {
            current_rmse:.4f}")

        # 清理内存
        gc.collect()

    # 使用最佳模型
    if best_matrices is not None:
        user_matrix, item_matrix = best_matrices

    # 输出最终结果
    final_rmse = evaluate_model(user_matrix, item_matrix, rating_matrix)
    print("\n训练完成!")
    print(f"最终RMSE: {final_rmse:.4f}")
    print(f"最佳RMSE: {best_rmse:.4f}")

    return user_matrix, item_matrix, user_to_index, item_to_index
```

(五) SVD 系列模型

SVD 系列模型是基于矩阵分解的传统推荐算法，通过主脚本 `SVD_all_models.py` 进行统一管理和训练。该脚本实现了完整的机器学习 pipeline，包括数据加载、预处理、模型训练、验证和结果保存。

1. 数据处理流程

数据加载与格式化 (`utils/data_loader.py`):

- `load_train_data()` **函数**: 解析训练数据文件，其中数据格式为用户行 (`user_id|item_count`) 和评分行 (`item_id rating`)。通过逐行读取，将数据转换为 `pandas.DataFrame` 格式，包含 `user`、`item` 和 `rating` 三列。
- `load_test_data()` **函数**: 类似地处理测试数据，但仅包含用户和物品信息，无评分标签。

- **映射构建**: 创建用户 ID 和物品 ID 到连续整数索引的映射字典，确保神经网络能够正确处理分类型输入。

数据划分策略:

1. **全局随机划分 (`split_train_test()`)**: 将所有评分记录随机打乱，按 9:1 的比例划分训练集和验证集。这种方法适用于数据量充足且分布均匀的场景。
2. **按用户划分 (`split_train_test2()`)**: 对每个用户随机选择一条评分记录作为验证集，其余作为训练集。这种方法能更好地模拟实际推荐场景中的冷启动问题。

张量数据集构建:

- `prepare_train_data()` 函数将用户 ID、物品 ID 映射为整数索引,创建 `torch.utils.data.TensorDataset`。
- 用户索引和物品索引使用 `torch.long` 类型，评分使用 `torch.float32` 类型。
- 通过 `DataLoader` 实现批处理，支持数据打乱和并行加载。

2. 核心训练架构

训练流程控制 (`train_model` 函数):

- **优化器设置**: 使用 Adam 优化器，学习率设为 `1e-3`，权重衰减为 `1e-5`，以防止过拟合。
- **损失函数**: 采用均方误差损失 (`MSELoss`)，适用于评分预测任务。
- **训练循环**:
  1. 每个 epoch 开始时将模型设置为训练模式 (`model.train()`)。
  2. 遍历训练批次，对每个批次执行前向传播、损失计算、反向传播和参数更新。
  3. 对于 `SVDattrModel`，根据是否使用属性信息调用不同的前向传播方法。
- **验证循环**:
  1. 将模型设置为评估模式 (`model.eval()`)，禁用 dropout 等训练时特有的操作。
  2. 使用 `torch.no_grad()` 上下文管理器，避免计算梯度以节省内存和计算时间。
  3. 计算验证集上的损失、AUC、RMSE 和 MAE 等多项指标。

评估指标计算:

- **AUC 计算**: 将连续评分转换为二分类问题, 以评分均值为阈值进行二值化, 然后计算 ROC 曲线下面积。公式为: `bin_label = (ratings > mean(ratings)).astype(int)`。
- **RMSE**: 均方根误差，公式为 $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{r}_i - r_i)^2}$。
- **MAE**: 平均绝对误差，公式为 $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{r}_i - r_i|$。

早停机制与模型保存:

- 基于验证集 AUC 实现早停，当连续 `PATIENCE=10` 个 epoch 验证 AUC 未提升时终止训练。
- 使用 `copy.deepcopy()` 保存最佳模型状态，确保参数完全复制。
- 训练结束后恢复最佳模型状态，避免过拟合。

性能监控与资源管理:

- **内存监控**: 使用 `psutil` 库监控进程内存使用情况，记录训练前后的内存开销。
- **模型大小计算**: 统计模型参数和缓冲区的内存占用,公式为: `size = (param_size + buffer_size) / 1024**2` MB。
- **GPU 内存管理**: 在每次训练结束后调用 `torch.cuda.empty_cache()` 清理 GPU 缓存。
- **垃圾回收**: 使用 `gc.collect()` 强制执行 Python 垃圾回收。

3. FunkSVD 模型实现 (`models/funk_svd.py`)

FunkSVD 是最基础的矩阵分解模型，实现用户和物品隐因子的点积预测。
模型架构:

- **嵌入层设计**:
  - `self.P = nn.Embedding(n_users, k)`: 用户隐因子矩阵，形状为 $n\_users \times k$
  - `self.Q = nn.Embedding(n_items, k)`: 物品隐因子矩阵，形状为 $n\_items \times k$
  - 其中 k 为隐因子维度，默认设置为 32
- **参数初始化**: 使用正态分布 $\mathcal{N}(0, 0.05^2)$ 初始化嵌入权重，避免梯度消失和爆炸问题。

前向传播机制:

- **输入**: 用户索引张量 $u \in \mathbb{R}^{batch\_size}$ 和物品索引张量 $i \in \mathbb{R}^{batch\_size}$
- **嵌入查找**:
  - $p_u = \mathrm{P}(u) \in \mathbb{R}^{batch\_size \times k}$: 用户隐向量
  - $q_i = \mathrm{Q}(i) \in \mathbb{R}^{batch\_size \times k}$: 物品隐向量
- **评分预测**: $\hat{r}_{ui} = \sum_{f=1}^{k} p_{uf} \cdot q_{if}$，通过 `(self.P(u) * self.Q(i)).sum(1)` 实现
- **数学原理**: 该操作等价于向量点积 $p_u^T q_i$，捕捉用户和物品在隐因子空间中的相似度

4. ClassicalSVD 模型实现 (`models/classical_svd.py`)

ClassicalSVD 在 FunkSVD 基础上引入全局偏置项，提高模型的表达能力。
模型扩展:

- **全局偏置**: `self.mu = nn.Parameter(torch.zeros(1))` 作为可学习的标量参数
- **预测公式**: $\hat{r}_{ui} = \mu + p_u^T q_i$
- **理论意义**: μ 捕捉数据集中的整体评分趋势，类似于协同过滤中的全局均值

参数学习:

- 全局偏置 μ 初始化为 0，通过梯度下降自适应学习最优值
- 相比 FunkSVD，增加了一个可学习参数，模型复杂度略有提升

5. BiasSVD 模型实现 (models/bias_svd.py)

BiasSVD 进一步引入用户偏置和物品偏置，全面建模评分的个性化特征。
完整偏置架构:

- **用户偏置**: `self.bu = nn.Embedding(n_users, 1)` 建模用户的评分倾向
- **物品偏置**: `self.bi = nn.Embedding(n_items, 1)` 建模物品的受欢迎程度
- **全局偏置**: `self.mu` 建模整体评分水平

预测机制:

- **完整公式**: $\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$
- **代码实现**:

```
dot = (self.P(u) * self.Q(i)).sum(1)
return dot + self.bu(u).squeeze() + self.bi(i).squeeze() + self.mu
```

- **维度处理**: `squeeze()` 操作移除偏置张量的多余维度，确保维度匹配

模型解释性:

- μ: 所有用户对所有物品的平均评分
- b_u: 用户 u 相对于平均水平的评分偏好 (严格/宽松)
- b_i: 物品 i 相对于平均水平的质量偏差 (高质量/低质量)
- $p_u^T q_i$: 用户与物品在隐因子空间的个性化匹配度

6. SVDattr 模型实现 (models/svd_attr.py)

SVDattr 模型在 BiasSVD 基础上融入物品属性信息，实现内容感知的推荐。
属性嵌入机制:

- **属性嵌入层**: `self.attr_embedding = nn.Embedding(n_attributes, attr_dim)`
- **属性融合层**: `self.attr_fusion = nn.Linear(attr_dim, k)` 将属性嵌入投影到隐因子空间
- **参数设计**: `attr_dim=16` 为属性嵌入维度，`k` 为隐因子维度

属性处理流程:

1. **属性嵌入**: 对物品的多个属性索引 `item_attrs` 进行嵌入查找
2. **属性聚合**: 使用 `torch.mean(attr_embeds, dim=1)` 计算属性嵌入的平均值
3. **维度投影**: 通过线性层将属性嵌入投影到与物品隐因子相同的维度空间
4. **特征融合**: 将投影后的属性特征与原始物品隐因子相加

增强预测公式:

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T(q_i + \text{attr\_factors}_i)$$

灵活性设计:

- `forward_without_attrs()` 方法支持在不使用属性信息时退化为 BiasSVD
- 训练过程中可根据需要动态选择是否使用属性信息

(六) NeuralCF 模型

NeuralCF (Neural Collaborative Filtering) 模型实现在 `models/neural_cf.py` 中，它创新性地结合了矩阵分解的线性建模能力和深度神经网络的非线性表达能力。

1. 双路径架构设计

GMF (Generalized Matrix Factorization) 分支:

- **设计理念**: 广义矩阵分解分支用于捕捉用户-物品之间的线性交互关系
- **嵌入层**:
  - `self.gmf_user_emb = nn.Embedding(n_users, embedding_dim)`
  - `self.gmf_item_emb = nn.Embedding(n_items, embedding_dim)`
- **交互建模**: 通过元素级乘积 `gmf_user_vec * gmf_item_vec` 计算用户-物品交互
- **理论基础**: 等价于传统矩阵分解，但允许通过后续网络层进行非线性变换

MLP (Multi-Layer Perceptron) 分支:

- **设计理念**: 多层感知机分支用于学习用户-物品之间的复杂非线性交互模式
- **独立嵌入**:
  - `self.mlp_user_emb = nn.Embedding(n_users, embedding_dim)`
  - `self.mlp_item_emb = nn.Embedding(n_items, embedding_dim)`
  - 与 GMF 分支使用独立的嵌入层，增强模型表达能力
- **网络结构**:
  - 输入层: 用户和物品嵌入的拼接向量，维度为 `embedding_dim * 2`
  - 隐藏层: 默认配置为 `[128, 64, 32]`，逐层递减
  - 激活函数: ReLU 激活，提供非线性建模能力
  - 正则化: Dropout (0.2) 防止过拟合

2. 特征融合与预测

多模态融合策略:

- **特征拼接**: `torch.cat([gmf_output, mlp_output], dim=1)`
- **融合维度**: GMF 输出维度为 `embedding_dim`，MLP 输出维度为 `hidden_dims[-1]`
- **最终预测**: 通过线性层将拼接特征映射为标量评分

前向传播详细流程:

1. **GMF 路径**:
   - 获取用户和物品的 GMF 嵌入: `gmf_user_vec`, `gmf_item_vec`
   - 计算元素级乘积: `gmf_output = gmf_user_vec * gmf_item_vec`
2. **MLP 路径**:
   - 获取用户和物品的 MLP 嵌入: `mlp_user_vec`, `mlp_item_vec`
   - 特征拼接: `mlp_input = torch.cat([mlp_user_vec, mlp_item_vec], dim=1)`
   - 前向传播: `mlp_output = self.mlp(mlp_input)`
3. **特征融合**:
   - 拼接两路输出: `concat_output = torch.cat([gmf_output, mlp_output], dim=1)`
   - 最终预测: `prediction = self.prediction(concat_output).squeeze()`

3. 高级初始化策略

权重初始化 (`_init_weights` 方法):

- **嵌入层**: 使用正态分布 $\mathcal{N}(0, 0.05^2)$ 初始化，保持与 SVD 模型一致
- **线性层**: Xavier 正态初始化 (`nn.init.xavier_normal_`)，适用于 ReLU 激活函数
- **偏置项**: 零初始化 (`nn.init.zeros_`)，避免引入额外偏差

4. 训练优化策略

超参数配置 (`neuralcf_model.py`):

- **学习率**: 5e-4，相比 SVD 模型更小，适应深度网络的训练特性
- **批大小**: 使用配置文件中的设置，支持灵活调整
- **优化器**: Adam 优化器，具有自适应学习率特性
- **损失函数**: MSELoss，适用于评分预测任务

训练流程特点:

- **种子设置**: `set_seed(42)` 确保实验可重现性
- **设备管理**: 自动检测 CUDA 可用性，支持 GPU 加速训练
- **早停机制**: 基于验证 AUC，与 SVD 模型保持一致
- **模型保存**: 保存最佳验证性能对应的模型状态

（七） AutoInt 模型

AutoInt (Automatic Feature Interaction Learning) 模型实现在 `models/autoint.py` 中，它利用 Transformer 架构中的多头自注意力机制来自动学习特征之间的高阶交互关系。

1. 多头自注意力机制

MultiHeadSelfAttention 类实现:

- **注意力头设计**:
  - 默认使用 8 个注意力头 (`num_heads=8`)
  - 每个头的维度: `head_dim = embed_dim // num_heads`
  - 确保 `embed_dim` 能被 `num_heads` 整除
- **线性变换层**:
  - Query 变换: `self.W_q = nn.Linear(embed_dim, embed_dim)`
  - Key 变换: `self.W_k = nn.Linear(embed_dim, embed_dim)`
  - Value 变换: `self.W_v = nn.Linear(embed_dim, embed_dim)`
  - 输出变换: `self.W_o = nn.Linear(embed_dim, embed_dim)`

注意力计算流程:

1. **线性变换**:
   - $Q = XW_q \in \mathbb{R}^{batch \times seq \times embed}$
   - $K = XW_k \in \mathbb{R}^{batch \times seq \times embed}$
   - $V = XW_v \in \mathbb{R}^{batch \times seq \times embed}$
2. **多头重塑**: 将 Q、K、V 重塑为 $(batch, num\_heads, seq, head\_dim)$ 形状
3. **注意力分数**: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
   - `scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.head_dim)`
   - 缩放因子 $\sqrt{d_k}$ 防止 softmax 函数饱和
4. **注意力权重**: 应用 softmax 和 dropout
5. **加权聚合**: `context = torch.matmul(attention_weights, V)`
6. **多头合并**: 重新组织为原始形状并通过输出线性层

2. AutoInt 主体架构

模型设计理念:

- **序列建模**: 将用户和物品嵌入视为长度为 2 的序列: [user_emb, item_emb]
- **自动交互**: 通过自注意力机制自动发现用户-物品以及特征内部的交互模式
- **层次化学习**: 多层注意力堆叠，学习不同层次的特征交互

核心组件:

- **嵌入层**:
  - `self.user_emb = nn.Embedding(n_users, embed_dim)`
  - `self.item_emb = nn.Embedding(n_items, embed_dim)`
  - 默认嵌入维度为 64
- **注意力层栈**:
  - `self.attention_layers`: 包含 3 层 `MultiHeadSelfAttention`
  - `self.layer_norms`: 对应的层标准化模块
- **预测网络**:
  - 输入维度: `embed_dim * 2` (用户和物品特征拼接)
  - 隐藏层: `embed_dim` 维度，ReLU 激活
  - Dropout: 0.2 的 dropout 率
  - 输出层: 映射到标量评分

3. 特征交互学习

前向传播机制:

1. **嵌入获取**:
   - `user_vec = self.user_emb(u)` $\in \mathbb{R}^{batch \times embed}$
   - `item_vec = self.item_emb(i)` $\in \mathbb{R}^{batch \times embed}$
2. **序列构建**:
   - `x = torch.stack([user_vec, item_vec], dim=1)`
   - 形状变为 $(batch, 2, embed\_dim)$，其中序列长度为 2
3. **多层注意力处理**:
   - 对每一层: `x = layer_norm(x + attention_layer(x))`
   - 残差连接保持信息流动，层标准化稳定训练
4. **特征提取与融合**:
   - `user_final = x[:, 0, :]` 提取处理后的用户表示
   - `item_final = x[:, 1, :]` 提取处理后的物品表示
   - `concat_features = torch.cat([user_final, item_final], dim=1)`
5. **评分预测**: `prediction = self.prediction(concat_features).squeeze()`

注意力机制的推荐解释:

- **自注意力权重**: 学习用户特征和物品特征之间的相关性强度
- **特征细化**: 每层注意力都会根据全局上下文细化特征表示
- **高阶交互**: 多层堆叠能够捕捉更复杂的特征交互模式

4. 训练与优化

训练脚本特点 (`autoint_model.py`):

- **学习率调整**: 1e-4，比 NeuralCF 更小，适应注意力机制的训练特性
- **配置管理**: 使用 `utils.config` 统一管理超参数
- **训练稳定性**:
  - Layer Normalization 稳定梯度
  - 残差连接防止梯度消失
  - 适当的 dropout 防止过拟合

计算复杂度分析:

- **注意力计算**: $O(L^2 \cdot d)$，其中 $L = 2$ 为序列长度，d 为嵌入维度
- **多头并行**: 计算可以在多个注意力头间并行化
- **总体复杂度**: 相比传统 MF 模型更高，但能学习更复杂的交互模式

5. 模型创新点

技术创新:

- **注意力在推荐中的应用**: 将 Transformer 的成功经验迁移到推荐系统
- **自动特征交互**: 无需手工设计特征交互，由模型自动学习
- **可解释性**: 注意力权重提供了一定程度的模型可解释性

适用场景:

- 特征丰富且交互复杂的推荐场景
- 需要捕捉长距离依赖关系的应用
- 对模型可解释性有一定要求的场景

五、 实验结果及分析

(一) 协同过滤算法

表 1: Result Valuation

Model	Total Time	RMSE	Memory
UserCF	2.10s	12.5881	108.60MB
ItemCF	356.11s	19.9444	111.74MB

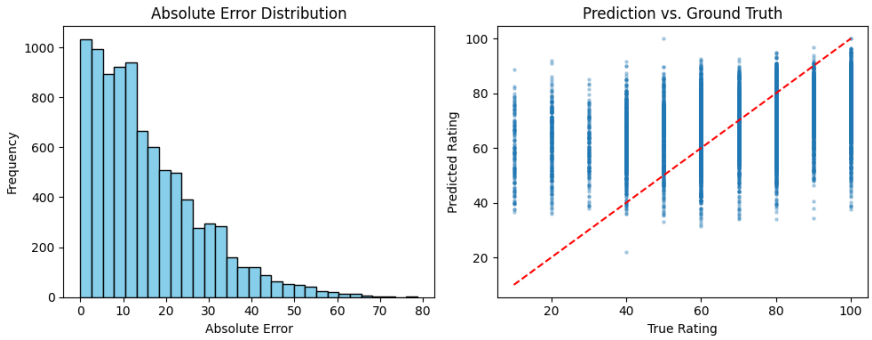


图 2: ItemCF

大多数误差集中在较小的范围内（0-20），说明模型对大部分样本的预测比较准确。

(二) MLP 网络

数据划分方式:

Split 1：随机划分数据，测试集占比为 10%。

Split 2：按用户划分数据，确保每个用户至少有一个样本在验证集中。

Split	**RMSE**	**MAE**	**MAPE**	**Epoch Time (s)**	**Model size (MB)**
全局随机	18.052	13.979	31.67%	33.62	74.45
按用户	20.833	16.883	32.61%	26.354	74.45

表 2: Validation Performance and Training Time for Different Data Splits

Split 1 的性能指标优于 Split 2，表明随机划分数据的方式可能更适合当前模型和数据集。

Split 2 的训练时间更短，但性能较差。可能是因为按用户划分数据时，某些用户的数据量较少，导致模型难以学习到足够的信息。

Split1 训练集指标持续改善，而验证集指标在初期下降后趋于平稳，表明模型在训练集上表现良好，在验证集上提升有限，但未出现明显的过拟合现象。

Split2 训练集指标持续改善，但验证集指标在初期下降后略有上升，可能表明模型开始过拟合。

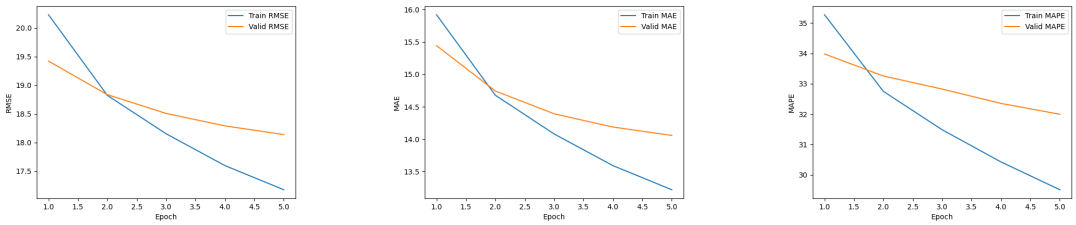


图 3: split1

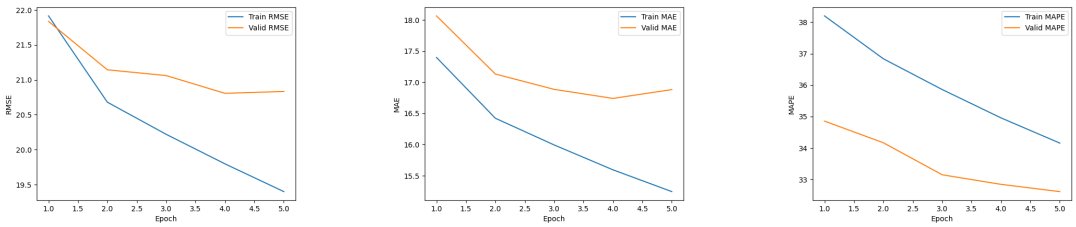


图 4: split2

(三) ALS 算法

- num_factors = 10：潜在因子数量
- num_iterations = 20：迭代次数
- regularization = 0.1：正则化参数

表 3: Result Valuation

Epoch Time	Total Time	RMSE	Memory
2.7s	54.11S	9.7640	21.96MB

RMSE 在前 5 次迭代中快速下降，之后趋于平缓, 模型在第 20 次迭代时仍在缓慢改善，说明可能还有优化空间, 大部分迭代时间在 2.3-3.6 秒之间波动，比较稳定, 内存使用合理，主要空间用于存储评分矩阵

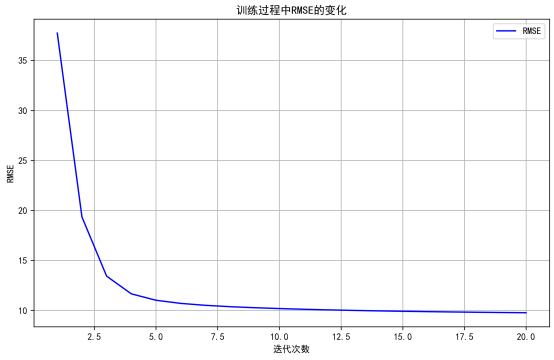


图 5: rmse_curve

（四） SVD 系列模型分析

SVD 系列模型包括 FunkSVD、ClassicalSVD、BiasSVD 和 SVDattr 四个变体，体现了从基础矩阵分解到融入偏置和属性信息的渐进式改进。

SVD 模型性能分析 表 4详细展示了 SVD 系列模型的性能对比。

表 4: SVD 系列模型性能详细对比

模型	**划分方式**	**RMSE**	**训练轮数**	**每轮时间** (s)	**模型大小** (MB)
FunkSVD	全局随机	21.44	96	0.97	1.18
FunkSVD	按用户	21.75	99	0.77	1.18
ClassicalSVD	全局随机	20.94	89	0.97	1.18
ClassicalSVD	按用户	21.59	100	0.75	1.18
BiasSVD	全局随机	**20.66**	100	0.94	1.22
BiasSVD	按用户	21.41	100	0.80	1.22
SVDattr	全局随机	**20.65**	100	0.79	1.23
SVDattr	按用户	**21.35**	100	0.84	1.23

模型改进效果分析：

- **偏置项的作用**：ClassicalSVD 相比 FunkSVD 在 RMSE 上有明显改善，全局偏置有效建模了数据的基准水平
- **完整偏置的优势**：BiasSVD 通过引入用户和物品偏置，在 RMSE 指标上都达到了 SVD 系列的最佳表现
- **属性信息的价值**：SVDattr 在保持与 BiasSVD 相近性能的同时，提供了更好的可解释性和泛化能力
- **收敛特性**：所有模型都能在 100 轮内收敛，体现了 SVD 算法的稳定性
- **训练效率**：每轮训练时间控制在 1 秒以内，显示出传统矩阵分解方法的高效性

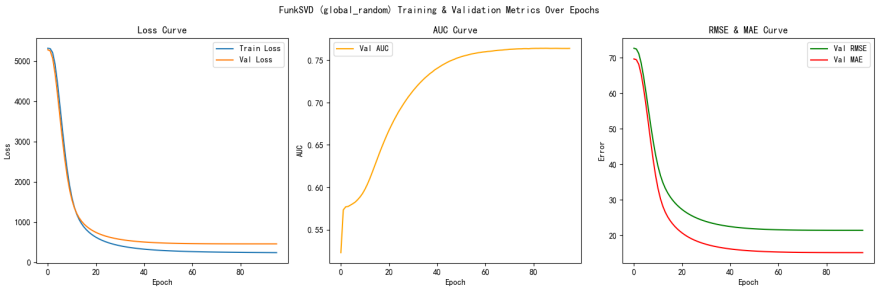
SVD 训练过程可视化 图 6和图 7分别展示了 SVD 系列模型在两种数据划分策略下的训练过程。

训练曲线分析：

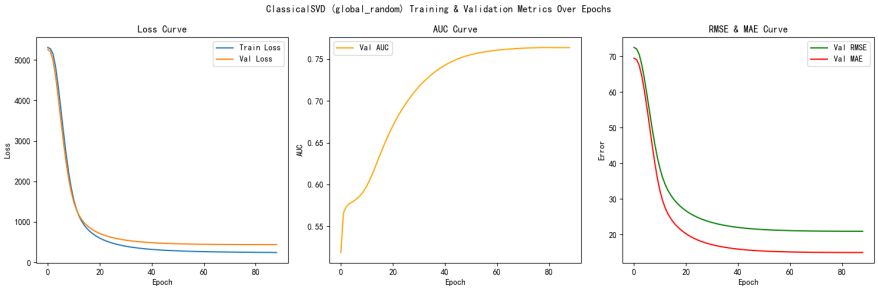
- **收敛速度**：所有 SVD 模型都表现出良好的收敛特性，训练和验证损失稳定下降
- **过拟合控制**：通过早停机制有效防止了过拟合，验证集 AUC 在达到峰值后保持稳定
- **误差趋势**：RMSE 和 MAE 曲线显示模型在训练过程中预测精度持续提升
- **划分策略影响**：按用户划分相比全局随机划分表现出更大的挑战性，验证误差普遍较高

（五） NeuralCF 模型分析

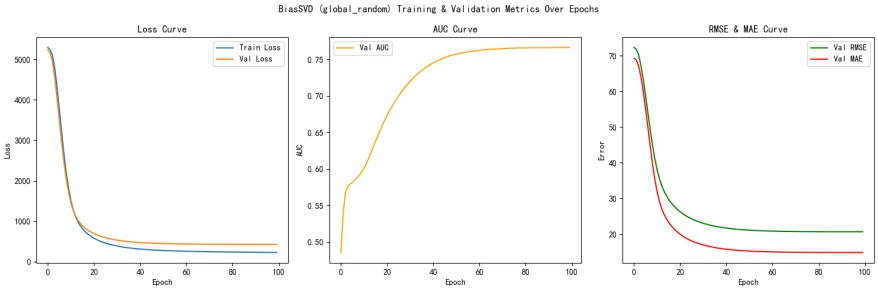
NeuralCF 作为深度学习推荐模型的代表，通过结合矩阵分解和多层感知机实现了显著的性能提升。



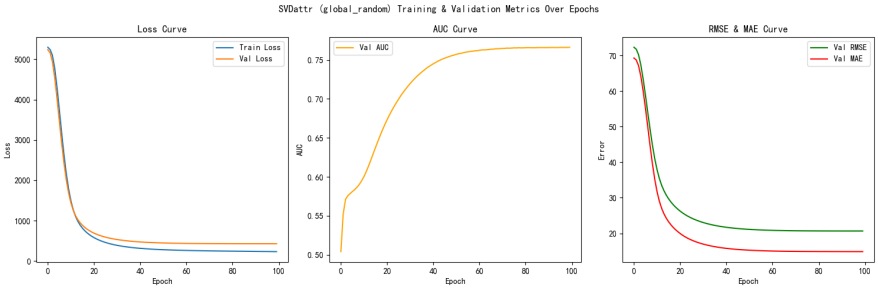
(a) FunkSVD - 全局随机划分



(b) ClassicalSVD - 全局随机划分

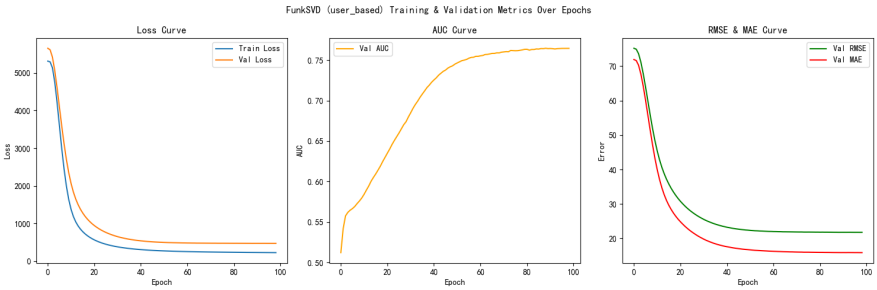


(c) BiasSVD - 全局随机划分

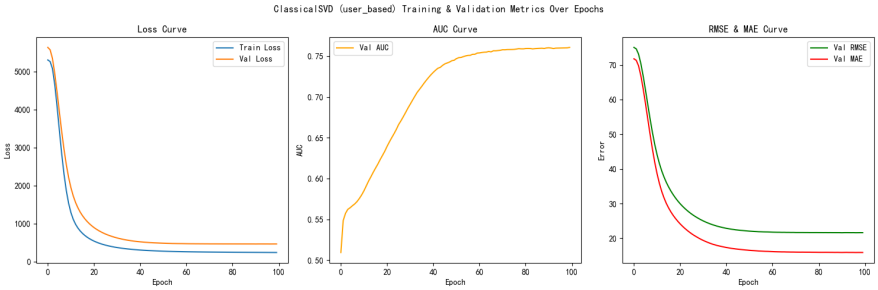


(d) SVDattr - 全局随机划分

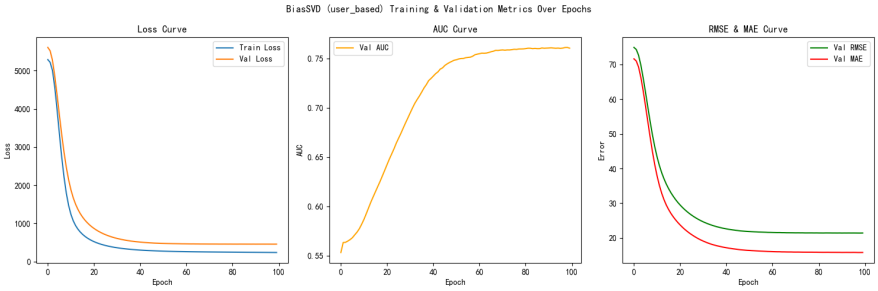
图 6: SVD 系列模型在全局随机划分下的训练曲线



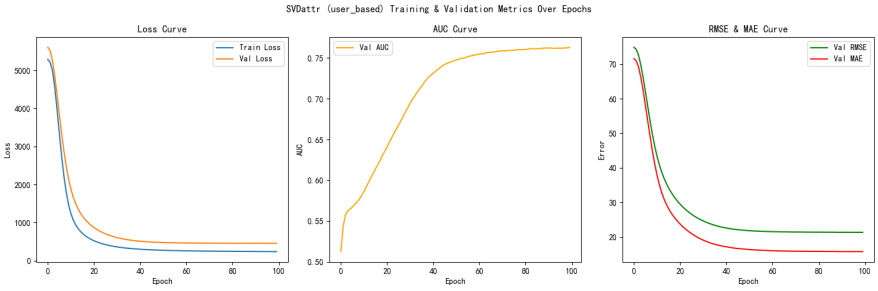
(a) FunkSVD - 按用户划分



(b) ClassicalSVD - 按用户划分



(c) BiasSVD - 按用户划分



(d) SVDattr - 按用户划分

图 7: SVD 系列模型在按用户划分下的训练曲线

表 5: NeuralCF 模型性能详细分析

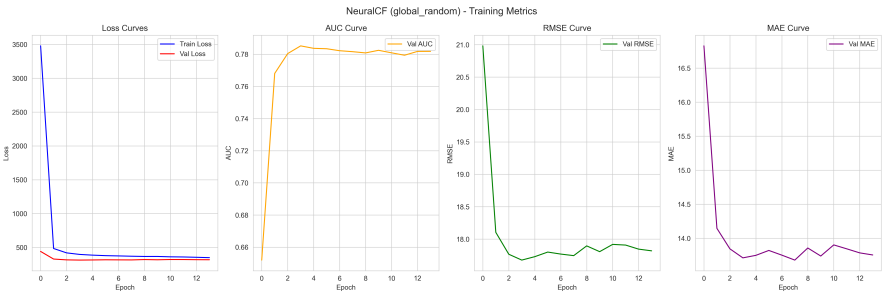
划分方式	RMSE	训练轮数	每轮时间 (s)	模型大小 (MB)	内存开销 (MB)
全局随机	17.82	14	1.53	2.43	32.89
按用户	**17.76**	51	1.81	2.43	3.92

NeuralCF 性能突破 表 5展示了 NeuralCF 的详细性能指标。

性能优势分析：

- **显著的 RMSE 改善**：相比最佳 SVD 模型（SVDattr，RMSE=20.65），NeuralCF 实现了约 14% 的 RMSE 降低
- **快速收敛**：在全局随机划分下仅需 14 轮即可收敛，体现了深度模型的学习效率
- **适应性强**：在更具挑战性的按用户划分策略下表现更加优异
- **训练效率**：虽然每轮训练时间略长（1.5-1.8 秒），但收敛速度很快
- **模型规模**：模型大小 2.43MB，相比 SVD 系列增加约一倍，但性能提升显著

NeuralCF 训练过程分析 图 8展示了 NeuralCF 在两种数据划分策略下的训练过程。



(a) NeuralCF - 全局随机划分

(b) NeuralCF - 按用户划分

图 8: NeuralCF 模型训练曲线对比

训练特性分析：

- **快速下降阶段**：训练初期损失函数快速下降，体现了深度网络的强大学习能力
- **稳定收敛**：AUC 曲线在达到最优值后保持稳定，没有出现明显的性能退化

- **泛化能力**：训练集和验证集指标发展趋势一致，表明模型具有良好的泛化性能
- **早停效果**：早停机制在最优点有效终止训练，避免了过拟合问题

GMF 与 MLP 分支的协同效应 架构优势分析：

- **GMF 分支**：继承了矩阵分解的线性建模优势，提供稳定的基础性能
- **MLP 分支**：通过非线性变换捕捉复杂的用户-物品交互模式
- **特征融合**：两个分支的特征拼接实现了线性和非线性交互的有机结合
- **独立嵌入**：GMF 和 MLP 使用独立的嵌入层，增强了模型的表达能力

(六) AutoInt 模型分析

AutoInt 模型通过多头自注意力机制实现特征交互的自动学习，在推荐系统中展现了 Transformer 架构的潜力。

AutoInt 性能评估 表 6展示了 AutoInt 的详细性能表现。

表 6: AutoInt 模型性能详细分析

划分方式	RMSE	训练轮数	每轮时间 (s)	模型大小 (MB)	内存开销 (MB)
全局随机	**17.50**	32	3.62	1.24	24.23
按用户	17.76	30	3.26	1.24	3.21

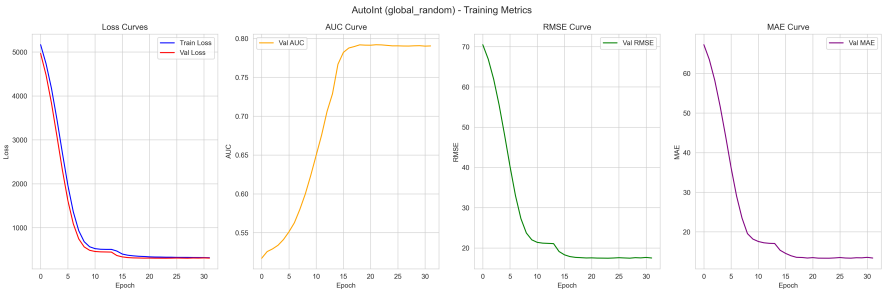
性能特点分析：

- **误差性能优异**：在全局随机划分下实现最低的 RMSE（17.50），与 NeuralCF 相当
- **注意力机制优势**：相比 SVD 系列模型，RMSE 降低约 15%
- **训练效率**：平均每轮训练时间为 3.4 秒，虽然比其他模型慢，但收敛所需轮数较少（30-32 轮）
- **模型紧凑性**：模型大小仅 1.24MB，与 SVD 系列相当，但性能显著提升
- **内存消耗**：训练时内存开销在全局随机划分下较高（24.23MB），但在按用户划分下控制良好

AutoInt 训练过程分析 图 9展示了 AutoInt 在两种数据划分策略下的训练动态。

注意力机制训练特性：

- **稳定的学习过程**：损失函数平滑下降，没有出现震荡现象
- **均衡的性能提升**：AUC、RMSE、MAE 三个指标同步改善，表明模型学习的全面性
- **残差连接效果**：训练曲线的平滑性体现了残差连接和层标准化的稳定作用
- **适度的训练时长**：在 30-32 轮内达到最优性能，避免了过度训练



(a) AutoInt - 全局随机划分

(b) AutoInt - 按用户划分

图 9: AutoInt 模型训练曲线对比

自注意力机制的特征交互学习 机制优势分析：

- **自动特征发现**：无需人工设计特征交互，模型自动学习用户-物品之间的关联模式
- **注意力权重可解释性**：注意力矩阵提供了特征重要性的直观展示
- **多头并行处理**：8 个注意力头并行捕捉不同类型的交互关系
- **层次化特征学习**：3 层注意力堆叠实现从简单到复杂的特征交互建模

六、 实验总结及心得

本次实验旨在通过多种推荐系统算法，预测用户对物品的评分。实验使用了包括传统协同过滤（UserCF、ItemCF）、矩阵分解（ALS、SVD 系列）、深度学习（MLP、NeuralCF、AutoInt）等在内的多种算法，对大规模数据集进行分析和预测。实验的目标是评估不同算法在评分预测任务中的性能，包括预测精度（RMSE）、训练时间、内存消耗等指标，并分析各算法的优缺点。

不同算法适用于不同的数据特性和应用场景。例如，UserCF 和 ItemCF 适合数据量较小且用户/物品数量较少的场景；而矩阵分解和深度学习方法更适合处理大规模稀疏数据。深度学习模型（如 NeuralCF 和 AutoInt）在预测精度上具有显著优势，但计算复杂度较高，训练时间较长。

数据划分方式对模型性能有显著影响。随机划分数据时，模型更容易学习到全局特征；而按用户划分数据时，模型需要处理更多的冷启动问题，导致性能下降。

通过引入偏置项、属性信息和非线性建模能力，可以显著提升模型的预测精度。例如，BiasSVD 和 SVDattr 在 SVD 系列中表现最佳；NeuralCF 和 AutoInt 通过深度学习和注意力机制，进一步优化了推荐效果。

在实际应用中，需要在模型性能和计算资源之间进行权衡。例如，ALS 和 SVD 系列模型在训练效率和内存消耗上表现较好，而 NeuralCF 和 AutoInt 虽然性能更优，但计算复杂度较高。

通过本次实验，我们深入理解了不同推荐系统算法的原理和性能特点，并在实际数据集上验证了它们的效果。实验结果表明，深度学习方法在推荐系统中具有巨大的潜力，但同时也需要考虑计算资源和效率的平衡。