南开大学
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

基于 SVD 、SVD++、ItemCF 和 UserCF 推荐系统的设计与实现

成员 1：苗发生 2010224 信息安全 & 法学双学位

成员 2：徐文斌 2010234 计算机卓越班

成员 3：李俞萱 2013572 计算机科学与技术 3 班

指导教师：杨征路

2023 年 6 月 8 日

目录

一、 实验要求 & 实验完成情况

(一) 实验要求

Task: Predict the rating scores of the pairs (u, i) in the Test.txt file.

Dataset：

1. **Train.txt**, which is used for training your models.
2. **Test.txt**, which is used for test.
3. **ItemAttribute.txt**, which is used for training your models (optional).
4. **ResultForm.txt**, which is the form of your result file.

The formats of datasets are explained in the DataFormatExplanation.txt. Note that if you can use ItemAttribute.txt appropriately and improve the performance of the algorithms, additional points (up to 10) can be added to your final course score.

(二) 实验完成情况

我们基于 SVD 和 CF 实现了四套推荐系统，并充分考虑实现过程中的并行处理特性。结果分析中，我们分别通过四套推荐系统得到实验结果，并对结果进行对比分析。

本小组在实现基本作业要求和学有余力的基础上，通过 SVD,SVD++,ItemCF,UserCF 实现了四种推荐系统，在实现的过程中充分考虑了并行处理特征。结果分析中，我们分别通过四套推荐系统得到实验结果，并对结果进行对比分析。具体完成情况如表 1 所示

表 1: 作业要求及本组完成情况

序号	作业要求	是否要求	本组完成情况
1	任何一种算法实现	√	√
2	对用户打分进行预测	√	√
3	数据集的分析	√	√
4	算法实现与分析	√	√
5	使用 ItemAttribute.txt 并提高算法的表现效果		√
6	通过多种算法实现		√
7	多种算法实现结果对比分析		√
8	并行加速		√

二、 基础知识

(一) SVD

奇异值分解（Singular Value Decomposition，简称 SVD）是一种常用的矩阵分解方法，可以将一个矩阵分解为三个矩阵的乘积。SVD 在数据分析、图像处理、推荐系统等领域有广泛应用。

给定一个实数或复数的 $m \times n$ 矩阵 A，SVD 将其分解为以下形式：

$$A = U \sum V^*$$

其中，U 是一个 $m \times m$ 的酉矩阵（即 $U^U = I$），Σ 是一个 $m \times n$ 的矩阵，其中对角线上的元素 $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ 称为奇异值，其余元素为 0，V 是一个 $n \times n$ 的酉矩阵。这里的 V^* 表示 V 的共轭转置。

具体地，U 的列向量称为左奇异向量，V 的列向量称为右奇异向量。奇异值 σ_i 与矩阵 A 的特征值关系密切，可以通过对 A 的特征值分解得到。

在 SVD 分解中，U 的列向量构成了 A 的特征向量的一组正交归一基，V 的列向量构成了 A^*A 的特征向量的一组正交归一基。

SVD 的应用非常广泛，其中一个重要的应用是降维。对于一个高维数据集，我们可以通过 SVD 将其降维到较低的维度，而保留数据的主要特征。具体地，我们可以保留前 k 个奇异值，将其余的奇异值置为 0，然后通过乘积 $U_k\Sigma_k V_k^*$ 来重构原始数据，其中 U_k 和 V_k 分别是 U 和 V 的前 k 列构成的矩阵，Σ_k 是将奇异值除了前 k 个以外的元素置为 0 得到的矩阵。这样可以大大减少数据的维度，同时尽量保留原始数据的信息。

SVD 还有其他一些重要的性质和应用，例如广义逆矩阵的计算、奇异值截断、最小二乘问题的求解等。

(二) SVD++

奇异值分解 ++（Singular Value Decomposition++，简称 SVD++）是在传统的 SVD（奇异值分解）算法的基础上进行改进，通过考虑用户的隐式反馈信息来提高推荐的准确性。下面是 SVD++ 算法的详细介绍。

假设我们有一个用户-物品评分矩阵 R，其中每个元素 r_{ui} 表示用户 u 对物品 i 的评分。SVD++ 算法的目标是通过将评分矩阵分解为低秩的用户特征矩阵和物品特征矩阵的乘积来进行推荐。

首先，我们定义用户 u 的隐式反馈向量 y_u，它表示用户 u 与其交互的物品集合的隐式反馈。隐式反馈可以是用户的点击、购买、浏览历史等行为。对于每个用户 u，我们还定义一个偏置项 b_u，表示用户 u 对物品的整体偏好。

SVD++ 算法的目标函数可以定义为最小化以下形式的损失函数：

$$\min_{p,q,y,b} \sum_{(u,i)\in R_{\text{train}}} (r_{ui} - \hat{r}_{ui})^2 + \lambda_1 \sum_u \|y_u\|^2 + \lambda_2 \sum_u \|p_u\|^2 + \lambda_3 \sum_i \|q_i\|^2 + \lambda_4 \sum_u \|b_u\|^2$$

其中，$(u,i) \in R_{train}$ 表示评分矩阵中的已知评分项，$\hat{r}_{ui}$ 是预测评分，p_u 和 q_i 分别是用户 u 和物品 i 的特征向量，$\lambda_1, \lambda_2, \lambda_3, \lambda_4$ 是正则化参数。

为了计算预测评分 $\hat{r}_{ui}$，我们可以使用以下公式：

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T (p_u + |N(u)|^{-\frac{1}{2}} \sum_{j\in N(u)} y_j)$$

其中，μ 是所有已知评分的平均值，b_i 是物品 i 的偏置项，$N(u)$ 是与用户 u 交互的物品集合，$|N(u)|$ 是集合 $N(u)$ 的大小。

SVD++ 算法的核心思想是通过将用户的隐式反馈信息 y_u 考虑在内，进一步捕捉用户与物品之间的关联。通过最小化目标函数，我们可以通过梯度下降等方法来学习得到用户特征矩阵。

(三) UserCF

用户协同过滤（User-based Collaborative Filtering，简称 UserCF）用于根据用户之间的相似度来推荐物品。UserCF 通过分析用户之间的行为数据，找到具有相似兴趣爱好的用户，并将这些用户喜欢的物品推荐给目标用户。

假设有一个用户-物品矩阵 R，其中每个元素 R_{ij} 表示用户 i 对物品 j 的评分或行为数据。UserCF 的工作流程如下：

1. **相似度计算：**首先，通过某种相似度度量方法（如余弦相似度、皮尔逊相关系数等），计算用户之间的相似度。设用户 i 和用户 j 之间的相似度为 $sim(i,j)$
2. **相似用户选择：**选取与目标用户最相似的 k 个用户，构建相似用户集合 $N(i)$。可以根据相似度值进行排序，选择相似度最高的 k 个用户
3. **物品推荐：**根据相似用户集合 $N(i)$，推荐目标用户可能感兴趣且尚未评价过的物品。一种常见的推荐方法是基于邻居用户的加权评分，即利用相似用户的评分数据为目标用户生成推荐列表。

   具体而言，对于目标用户 i 尚未评价过的物品 j，可以通过以下步骤生成推荐得分：

   $$R'_{ij} = \frac{1}{\sum_{j' \in N(i)} sim(i,j')} \sum_{j' \in N(i)} sim(i,j') \cdot R_{j'j}$$

   其中，R'_{ij} 表示物品 j 的推荐得分，$sim(i,j')$ 表示用户 i 与用户 j' 之间的相似度，$R_{j'j}$ 表示用户 j' 对物品 j 的评分。
4. **推荐列表生成：**根据物品的推荐得分，按照一定的规则（如取 Top-N）生成最终的推荐列表，将推荐物品推荐给目标用户。

UserCF 的优点在于利用用户之间的行为数据进行推荐，避免了对物品的内容信息的依赖，具有较好的可解释性和灵活性。然而，UserCF 也存在一些问题，如数据稀疏性、冷启动问题和推荐偏好的一致性等挑战。

综上所示，用户协同过滤（UserCF）通过计算用户之间的相似度、选择相似用户集合，并利用相似用户的评分数据为目标用户生成推荐列表。UserCF 算法的优点包括可解释性和灵活性，但也存在数据稀疏性、冷启动问题和推荐偏好一致性等挑战。因此，在实际应用中需要综合考虑算法的特点和限制，并结合其他推荐算法进行综合推荐。

（四） ItemCF

物品协同过滤（Item-based Collaborative Filtering，简称 ItemCF）用于为用户提供个性化推荐。它基于物品之间的相似性来推荐用户可能感兴趣的物品。

假设有一个物品-用户矩阵 R，其中 R_{ij} 表示用户 i 对物品 j 的评分。ItemCF 的主要思想是通过计算物品之间的相似性来预测用户对未评价物品的喜好程度。

首先，计算物品之间的相似性矩阵 S。常用的相似性度量方法包括余弦相似度和皮尔逊相关系数等。可以使用以下公式计算物品 i 和物品 j 之间的相似度：

$$S_{ij} = \frac{\sum_{u \in U_{ij}} (R_{ui} - \overline{R_u})(R_{uj} - \overline{R_u})}{\sqrt{\sum_{u \in U_{ij}} (R_{ui} - \overline{R_u})^2} \sqrt{\sum_{u \in U_{ij}} (R_{uj} - \overline{R_u})^2}}$$

其中，U_{ij} 表示同时对物品 i 和物品 j 评分的用户集合，$\overline{R_u}$ 表示用户 u 的评分均值。

然后，对于用户 u，通过以下公式计算对未评分物品 j 的预测评分 $\hat{R}_{uj}$：

$$\hat{R}_{uj} = \overline{R_u} + \frac{\sum_{i \in I_u} S_{ij}(R_{ui} - \overline{R_u})}{\sum_{i \in I_u} |S_{ij}|}$$

其中，I_u 表示用户 u 评价过的物品集合。

最后，根据预测评分的高低为用户生成推荐列表。

（五） 基于 UserCF 和 ItemCF 实现起来的区别

ItemCF（基于物品的协同过滤）和 UserCF（基于用户的协同过滤）是两种不同的推荐算法，它们在实现推荐系统时的逻辑上存在一些区别：

1. **数据处理方式：** 在 ItemCF 中，主要关注物品之间的相似度计算和推荐物品的选择；而在 UserCF 中，主要关注用户之间的相似度计算和推荐给用户的物品选择
2. **相似度计算：** 在 ItemCF 中，通过分析用户对物品的评分行为，计算不同物品之间的相似度，以确定物品的相似度矩阵；而在 UserCF 中，通过分析用户的评分行为，计算不同用户之间的相似度，以确定用户的相似度矩阵
3. **推荐生成方式：** 在 ItemCF 中，根据用户历史评分信息和物品的相似度矩阵，计算出推荐物品的评分预测值，并选取预测值高的物品作为推荐结果；而在 UserCF 中，根据用户之间的相似度矩阵和用户的历史评分信息，计算出推荐物品的评分预测值，并选取预测值高的物品作为推荐结果
4. **推荐结果解释：** 在 ItemCF 中，推荐的物品是与用户历史喜好相似的其他物品；而在 UserCF 中，推荐的物品是其他与用户相似的用户喜欢的物品

总体而言，ItemCF 更注重挖掘物品之间的相似性，以找到用户可能喜欢的相似物品；而 UserCF 更注重挖掘用户之间的相似性，以找到与用户兴趣相似的其他用户，从而推荐他们喜欢的物品。因此，两种算法在推荐系统中的实现逻辑会有所区别。

三、 数据集

（一） 数据集划分方法

我们采用流出法（Holdout Method）将原始数据集划分为训练集和测试集。流出法的基本思想是将原始数据集中的一部分样本数据留出作为测试集，而将剩余的数据作为训练集。这样可以在训练阶段使用训练集对模型进行参数估计和学习，并在测试阶段使用测试集评估模型的性能和泛化能力。

在具体实验中，我们对原始数据集的划分方法和步骤如下：

1. **数据集划分比例选择：** 首先需要确定将原始数据集划分为训练集和测试集的比例。划分比例的选择通常会受到数据集大小、任务需求和算法性能评估的要求等因素的影响。本次实验中，我们将数据集划分为将数据集划分为 70% 的训练集和 30% 的测试集。
2. **随机划分：** 将原始数据集中的样本数据随机划分为训练集和测试集。确保训练集和测试集中的样本数据是独立且随机选择的，以减小样本选择的偏差性。
3. **训练集和测试集的使用：** 使用训练集对模型进行训练和参数估计。在训练阶段，模型通过学习训练集中的样本数据来调整自身的参数和特征表示。在训练完成后，使用测试集对模型进行评估和性能测试。测试集中的样本数据没有参与模型的训练过程，可以用于评估模型在未见过的数据上的预测能力和泛化性能
4. **性能评估：** 使用测试集对模型在未知数据上的性能进行评估，我们采用的评估指标包括准确率、精确率、召回率、F1 值等。

(二) 数据集划分结果

通过上述流出法对数据集进行划分后，得到的数据集实际情况如下所示：

表 2: 数据集真实情况

数据集	用户数量	商品数量	评分数量
训练集	19835	624961	5001507
测试集	19835	624961	119010

对数据集进行统计，统计结果如下：

表 3: 数据集统计结果

Statistics	ratings	item attr1	item attr2
Size	5001507	464932	443687
Sparseness	99.96%	8.33%	12.52%
Mean	49.50	314596	311482
S.D.	38.22	180956	179777

(三) 数据可视化

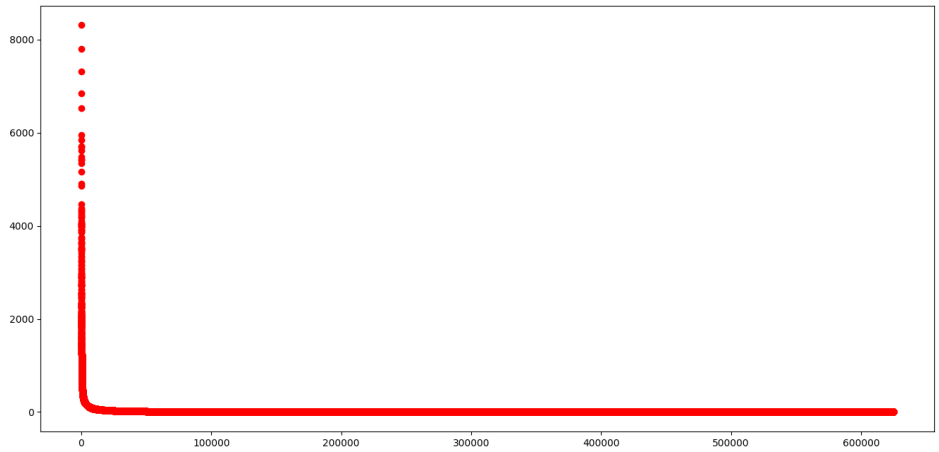
为了分析数据集特点，我们通过 python 来生成长尾图，具体的 python 代码如下所示：

生成长尾图

```python
# coding=utf-8
import matplotlib.pyplot as plt
import io


items = []
x = []
y = []

with io.open('./longTailData.txt', encoding='utf-8') as f:
    lineNo = 0
    totalTimes = 0
    for line in f:
        x.append(lineNo)
        items.append(int(line.split()[0]))
        y.append(float(line.split()[1]))
        totalTimes = totalTimes + y[lineNo]
        lineNo = lineNo + 1
    # totalTimes = 19835
    # y = map(lambda x:x / totalTimes, y)


plt.plot(x, y, 'ro')
```

```
plt.show()
```

该程序所做出的长尾图如下所示：



（四） 数据集划和预处理分关键代码实现

1. 数据集的读取

我们充分利用面向对象的思想来编写本次大作业的代码，在数据的读取中，我们封装和实现 DataAnalyser 类，数据的读取和操作通过该类中的 analyse() 函数实现，analyse() 是数据读取和分析的关键，具体实现如下：

数据的读取和分析

```
#include "DataAnalyser.h"

#include <fstream>
#include <set>
#include <iostream>
#include <vector>
#include <algorithm>

static bool cmp(std::pair<int, int> &a, std::pair<int, int> &b)
{
    return a.second > b.second;
}

std::map<std::string, int> DataAnalyser::analyse()
{
    std::map<int, int> rateTimes;
    std::set<int> users;
    std::set<int> items;
    int ratings = 0;
    int minUserId = 0x3f3f3f3f;
    int maxUserId = 0;
    int minItemId = 0x3f3f3f3f;
    int maxItemId = 0;
```

```
24      std::map<std::string, int> dataInfo;
25      std::ifstream readStream;
26      readStream.open(dataPath);
27      do
28      {
29          std::string header;
30          readStream >> header;
31          int userId, rateNum;
32          int sepPos = header.find_first_of("|");
33          userId = atoi(header.substr(0, sepPos).c_str());
34          rateNum = atoi(header.substr(sepPos + 1, header.size() - sepPos - 1).
                c_str());
35          users.insert(userId);
36          ratings += rateNum;
37          minUserId = std::min(userId, minUserId);
38          maxUserId = std::max(userId, maxUserId);
39          int itemId, score;
40          while (rateNum)
41          {
42              if (!isTest)
43              {
44                  readStream >> itemId >> score;
45                  rateTimes[itemId]++;
46              }
47              else
48                  readStream >> itemId;
49              items.insert(itemId);
50              minItemId = std::min(itemId, minItemId);
51              maxItemId = std::max(itemId, maxItemId);
52              rateNum--;
53          }
54      } while (!readStream.eof());
55      readStream.close();
56      dataInfo["userNum"] = users.size();
57      dataInfo["itemNum"] = items.size();
58      dataInfo["rateNum"] = ratings;
59      dataInfo["minUserId"] = minUserId;
60      dataInfo["maxUserId"] = maxUserId;
61      dataInfo["minItemId"] = minItemId;
62      dataInfo["maxItemId"] = maxItemId;
63
64      if (!isTest)
65      {
66          std::ofstream longTailData;
67          longTailData.open("./longTailData.txt");
68          std::vector<std::pair<int, int>> allItemRatedTimes;
69          for (int i = minItemId; i <= maxItemId; i++)
70              allItemRatedTimes.push_back(std::make_pair(i, rateTimes[i]));
```

```
        sort(allItemRatedTimes.begin(), allItemRatedTimes.end(), cmp);
        for (auto pa : allItemRatedTimes)
            longTailData << pa.first << " " << pa.second << "\n";
        longTailData.close();
    }

    return dataInfo;
}
```

关键部分代码解释：

1. **创建变量：** 函数开始时，创建了一些必要的变量，包括 rateTimes（记录每个物品的评分次数），users（记录用户的集合），items（记录物品的集合），ratings（记录评分总数），minUserId 和 maxUserId（记录用户 ID 的最小值和最大值），minItemId 和 maxItemId（记录物品 ID 的最小值和最大值），以及 dataInfo（存储数据分析结果的 map）
2. **打开数据集：** 通过 readStream.open(dataPath) 打开数据文件，dataPath 是 DataAnalyser 类中的一个成员变量，表示数据文件的路径
3. **读取数据并进行分析：** 通过循环读取数据文件中的每一行，并进行相应的数据分析操作。具体过程如下：
   (a) 读取每行的头部信息，包含了用户 ID 和评分次数
   (b) 提取用户 ID 和评分次数，并更新相关的变量和集合
   (c) 读取每个评分记录的物品 ID 和评分（如果不是测试数据），并更新相关的变量和集合
   (d) 记录最小和最大的用户 ID 和物品 ID
   (e) 统计每个物品的评分次数
4. 关闭数据文件，数据分析完成
5. 构造并返回数据分析结果：将分析得到的信息存储在 dataInfo 中，并返回该 map 作为分析结果
6. 长尾数据保存（如果不是测试数据）：如果不是测试数据（即 isTest 为 false），则将物品的评分次数按照从高到低的顺序保存在./longTailData.txt 文件中

总结来看，analyse() 函数通过读取数据文件，统计用户数、物品数、评分总数等信息，并可选择保存长尾数据。返回的 dataInfo 存储了这些分析结果

2. 数据集的划分和预处理

我们采用面向对象的思想，本部分的数据集划分，只需要实现 splitData () 函数即可，该函数将给定数据集按照指定的比例划分为训练集和测试集，并对训练集进行一些统计和预处理操作，为后续的推荐系统训练和评估提供数据基础，具体实现如下：

数据集的划分

```
void SVD::splitData(std::string dataPath, float rate)
{
    srand(time(0));
```

```
    int trainNum = 0; // 用于求平均值
    std::ifstream readStream;
    readStream.open(dataPath);
    do
    {
        std::string header;
        readStream >> header;
        int userId, rateNum;
        int sepPos = header.find_first_of("|");
        userId = atoi(header.substr(0, sepPos).c_str());
        rateNum = atoi(header.substr(sepPos + 1, header.size() - sepPos - 1).
            c_str());
        int exceptTestNum = (int)(rateNum * rate);
        int trueTestNum = 0;
        int itemId, score;
        while (rateNum)
        {
            readStream >> itemId >> score;

            if (abs(rand()) % 100 / 100.0 < rate && trueTestNum <
                exceptTestNum)
            {
                testMatrix[userId].push_back(std::make_pair(itemId, score));
                trueTestNum++;
            }
            else
            {
                trainMatrix[userId].push_back(std::make_pair(itemId, score));
                mean += score;
                trainNum++;
                clickPerItem[itemId]++;
            }
            rateNum--;
        }
        while (trueTestNum < exceptTestNum)
        {
            auto it = trainMatrix[userId].begin();
            while (it != trainMatrix[userId].end())
            {
                if (abs(rand()) % 100 / 100.0 < rate)
                {
                    mean -= (*it).second;
                    trainNum--;
                    testMatrix[userId].push_back(*it);
                    clickPerItem[(*it).first]--;
                    it = trainMatrix[userId].erase(it);
                    trueTestNum++;
                    if (trueTestNum >= exceptTestNum)
```

```
                        break;
                    }
                    else
                    {
                        it++;
                    }
                }
            }
        } while (!readStream.eof());
        readStream.close();
        mean /= trainNum;
}
```

该函数可归纳为如下几个步骤：

1. 读取每行数据，获取用户 ID 和评分数量
2. 根据划分比例计算测试集中的预期评分数量
3. 循环读取评分记录，并根据随机数将评分添加到测试集或训练集
4. 更新训练集的均值、训练样本数量和每个物品的点击数
5. 当测试集中的评分数量不足预期时，从训练集中移动评分记录到测试集
6. 计算训练集的评分均值

四、 核心代码实现

(一) Makefile

我们的程序是在 Linux 上进行编写，所使用的编译器为 g++，通过编写 Makefile 文件，来编译和构建 SVD 程序. 编写 Makefile 的目的是自动化地进行编译、链接和运行的步骤，并提供一些额外的功能。

Makefile

```
cc = g++
prom = SVD
deps = $(shell find ./ -name "*.h")
src = $(shell find ./ -name "*.cpp")
obj = $(src:%.cpp=%.o)

FLAG = -fopenmp -lpthread -std=c++17 -O2

$(prom): $(obj)
	@ $(cc) -o $(prom) $(obj) $(FLAG)

%.o: %.cpp $(deps)
	@ $(cc) -c $< -o $@ $(FLAG)

```

```
run: $(prom)
	@ ./$(prom)

clean:
	@ rm -rf $(obj) $(prom)
```

Makefile 文件解释：

1. **编译和构建：**通过执行 make 命令，根据规则和依赖关系，可以自动编译源文件，生成目标文件，最终将目标文件链接在一起生成目标程序
2. **源文件和头文件查找：**通过使用 find 命令，程序会在当前目录及其子目录中查找所有的源文件（.cpp）和头文件（.h），并将它们的路径保存在变量 src 和 deps 中
3. **编译选项：**通过设置变量 FLAG，可以指定编译选项，如使用 OpenMP 多线程库、链接 pthread 库、使用 C++17 标准以及启用优化级别 2
4. **目标程序运行：**通过执行 make run 命令，可以运行生成的目标程序
5. **清理：**通过执行 make clean 命令，可以删除生成的目标文件和目标程序，以便进行清理

(二) SVD 实现

1. 初始化模型参数

我们通过封装 initModelArg() 函数初始化用户和物品的偏置项以及对应的隐含因子矩阵，通过随机生成的初始值，为模型建立起初始状态。具体实现如下：

初始化模型参数

```
void SVD::initModelArg()
{
    srand(time(0));
    for (int i = 0; i < userNum; i++)
    {
        bu[i] = 0;
        for (int j = 0; j < dim; j++)
            pu[i][j] = abs(rand()) % dim / (1.0 * dim);
    }
    for (int i = 0; i < itemNum; i++)
    {
        bi[i] = 0;
        for (int j = 0; j < dim; j++)
            qi[i][j] = abs(rand()) % dim / (1.0 * dim);
    }
}
```

代码逻辑解释：

1. 对每个用户循环进行操作：
   (a) 将 bu[i] 初始化为 0

(b) 对每个维度循环进行操作：生成一个范围在 [0, dim) 之间的随机数，并将其赋值给 pu[i][j]

2. 对每个物品循环进行操作：

(a) 将 bi[i] 初始化为 0

(b) 对每个维度循环进行操作：生成一个范围在 [0, dim) 之间的随机数，并将其赋值给 qi[i][j]

2. 计算每个物品的 RMSE

我们通过封装 calRMSEPerItem() 函数来统计每个物品的预测误差情况，并将结果写入文件以供后续分析和评估模型性能

计算每个物品的 RMSE

```cpp
void SVD::calRMSEPerItem()
{
    for (int i = 0; i < userNum; i++)
    {
        for (auto node : testMatrix[i])
            trainMatrix[i].push_back(node);
    }
    std::map<int, std::pair<ld, int>> item2RMSE;
    for (int userId = 0; userId < userNum; userId++)
    {
        for (auto node : testMatrix[userId])
        {
            int itemId = node.first;
            ld score = node.second;
            ld rui = predictEXT(userId, itemId);
            ld eui = score - rui;
            item2RMSE[itemId].first += eui * eui;
            item2RMSE[itemId].second++;
        }
    }
    std::ofstream output;
    output.open("./rmsePerItem.txt");
    for (int itemId = 0; itemId < itemNum; itemId++)
    {
        ld rmse = 0;
        if (item2RMSE[itemId].second > 0)
            rmse = sqrt(item2RMSE[itemId].first / item2RMSE[itemId].second);
        output << itemId << " " << rmse << std::endl;
    }
    output.close();
}
```

代码逻辑解释：

1. 将测试数据中的每个评分节点（node）复制到对应用户的训练数据中，以便后续使用完整的训练数据进行预测
2. 创建一个映射（map）对象 item2RMSE，用于记录每个物品的误差平方和以及对应的评分数量
3. 对每个用户的测试数据中的每个评分节点进行如下操作：
   (a) 获取评分节点的物品 ID（itemId）和评分值（score）
   (b) 使用 predictEXT 函数预测该用户对该物品的评分值（rui）
   (c) 计算预测评分值与实际评分值之间的误差（eui）
   (d) 更新 item2RMSE 中对应物品的误差平方和和评分数量
4. 打开一个输出文件，命名为"rmsePerItem.txt"
5. 对每个物品循环进行如下操作：
   (a) 计算该物品的均方根误差（rmse），如果该物品的评分数量大于 0
   (b) 将物品 ID 和对应的均方根误差写入输出文件

3. 计算整体的 RMSE

我们通过封装 calRMSE() 函数来评估模型在训练集和测试集上的预测性能，通过计算均方根误差来衡量模型的拟合程度和预测准确性，具体实现如下：

计算训练集和测试集的 RMSE

```
void SVD::calRMSE()
{
    int trainNum = 0;
    ld trainRMSE = 0;
    int testNum = 0;
    ld testRMSE = 0;
    for (int userId = 0; userId < userNum; userId++)
    {
        trainNum += trainMatrix[userId].size();
        for (auto node : trainMatrix[userId])
        {
            int itemId = node.first;
            ld score = node.second;
            ld rui = predictUI(userId, itemId);
            ld eui = score - rui;
            trainRMSE += eui * eui;
        }
    }

    for (int userId = 0; userId < userNum; userId++)
    {
        testNum += testMatrix[userId].size();
        for (auto node : testMatrix[userId])
```

```
        {
            int itemId = node.first;
            ld score = node.second;
            ld rui = predictEXT(userId, itemId);
            ld eui = score - rui;
            testRMSE += eui * eui;
        }
    }
    trainRMSE = sqrt(trainRMSE / trainNum);
    testRMSE = sqrt(testRMSE / testNum);

    std::cout << "train RMSE: " << trainRMSE << "\ttest RMSE: " << testRMSE
        << std::endl;
}
```

代码逻辑解释：

1. 初始化变量 trainNum、trainRMSE、testNum 和 testRMSE，用于统计训练集和测试集的样本数量和误差平方和
2. 对每个用户循环进行操作：
   (a) 累加该用户训练数据的样本数量到 trainNum 变量
   (b) 对该用户的训练数据中的每个评分节点进行操作：
      i. 获取评分节点的物品 ID（itemId）和评分值（score）
      ii. 使用 predictUI 函数预测该用户对该物品的评分值（rui）
      iii. 计算预测评分值与实际评分值之间的误差（eui）
      iv. 累加误差平方到 trainRMSE 变量
3. 对每个用户循环进行操作：
   (a) 累加该用户测试数据的样本数量到 testNum 变量
   (b) 对该用户的测试数据中的每个评分节点进行操作：
      i. 获取评分节点的物品 ID（itemId）和评分值（score）
      ii. 使用 predictEXT 函数预测该用户对该物品的评分值（rui）
      iii. 计算预测评分值与实际评分值之间的误差（eui）
      iv. 累加误差平方到 testRMSE 变量
4. 计算训练集的均方根误差（trainRMSE），将误差平方和除以训练集样本数量后取平方根
5. 计算测试集的均方根误差（testRMSE），将误差平方和除以测试集样本数量后取平方根
6. 输出训练集和测试集的均方根误差

4. 预测用户对物品的评分

我们通过封装 predictUI(int uid, int iid) 函数来实现用户 uid 对物品 iid 的评分预测功能，具体原理为通过结合全局平均值、用户偏置项、物品偏置项和用户-物品的相似性，可以得出对用户对物品的评分预测。修正步骤则是为了确保评分值在合理的范围内。

预测用户对物品的评分

```
ld SVD::predictUI(int uid, int iid)
{
    ld rui = mean + bu[uid] + bi[iid] + product(pu[uid], qi[iid], dim);
    if (rui > 100)
        rui = 100;
    else if (rui < 0)
        rui = 0;
    return rui;
}
```

关键代码逻辑：

1. 据用户 uid 和物品 iid 的索引，从相应的数组和矩阵中获取对应的模型参数
2. 计算评分预测值 rui，该值由以下四部分组成：
   (a) 全局平均值 mean
   (b) 用户偏置项 bu[uid]，表示用户 uid 的整体评分倾向
   (c) 物品偏置项 bi[iid]，表示物品 iid 的整体评分倾向
   (d) 用户向量 pu[uid] 和物品向量 qi[iid] 之间的内积，通过 product 函数计算得到，表示用户 uid 和物品 iid 之间的相似度或相关性
3. 对 rui 进行一些修正：
   (a) 如果 rui 大于 100，将其设置为 100，限制评分在 0 到 100 之间
   (b) 如果 rui 小于 0，将其设置为 0，限制评分在 0 到 100 之间
4. 返回修正后的评分预测值 rui 作为函数的结果

5. 扩展预测

我们通过封装 predictEXT(int uid, int iid) 函数来进行扩展预测，当训练数据不足或物品冷启动时，通过计算与目标物品相似度较高的其他物品的加权评分来改进预测结果，具体实现如下：

扩展预测

```
ld SVD::predictEXT(int uid, int iid)
{
    ld res = predictUI(uid, iid);
    if (hasKNN && trainMatrix[uid].size() < itemNum / 10000 && clickPerItem[
        iid] < userNum / 1000)
    {
        ld knnRes = 0;
        ld simSum = 0;
        auto items = knn(uid, iid);
        for (int i = 0; i < items.size(); i++)
        {
            knnRes += predictUI(uid, items[i].first) * items[i].second;
            simSum += items[i].second;
```

```
13          }
14          if (simSum != 0)
15          {
16              ld knnWeight = 0.3;
17              knnRes /= simSum;
18              res = (1 - knnWeight) * res + knnWeight * knnRes;
19              if (res > 100)
20                  res = 100;
21              if (res < 0)
22                  res = 0;
23          }
24      }
25      return res;
26  }
```

代码核心逻辑：

1. 调用 predictUI(uid, iid) 函数，获取基础的评分预测值 res
2. 如果满足一定条件（具体条件为 hasKNN 为真、用户 uid 在训练数据中的评分数量小于总物品数量的 1/10000、物品 iid 在训练数据中被评分的用户数量小于总用户数量的 1/1000），则执行以下步骤：
   (a) 调用 knn(uid, iid) 函数，获取与物品 iid 最相似的物品列表 items
   (b) 初始化 knnRes 为 0，表示最终的 KNN 预测评分值，simSum 为 0，表示物品相似度的累加和
   (c) 遍历 items 列表，对于每个物品，执行以下步骤：
      i. 调用 predictUI(uid, items[i].first) 函数，获取用户 uid 对物品 items[i].first 的评分预测值
      ii. 将该预测值乘以 items[i].second，得到部分 KNN 预测评分的累加值 knnRes
      iii. 将 items[i].second 累加到 simSum 中，计算物品相似度的累加和
   (d) 如果 simSum 不等于 0，则执行以下步骤：
      i. 设置 knnWeight 为 0.3，表示 KNN 预测评分的权重
      ii. 将 knnRes 除以 simSum，得到加权平均后的 KNN 预测评分值
      iii. 使用加权平均的预测评分值更新 res，将其作为最终的评分预测值
      iv. 如果 res 大于 100，则将其限制为 100
      v. 如果 res 小于 0，则将其限制为 0
3. 返回最终的评分预测值 res 作为函数的结果

扩展预测的目的是在基础评分预测的基础上，通过 KNN 方法对用户对物品的评分进行扩展预测。如果满足特定条件，会计算用户 uid 与物品 iid 最相似的物品列表，并根据其相似度加权计算 KNN 预测评分值。最终，使用加权平均的预测评分值对基础评分预测进行修正。

6. 计算与目标物品相似性较高的其他物品

我们通过封装 knn(int uid, int iid) 用于找到与物品 iid 最相似的 k 个物品，具体实现如下：

计算与目标物品相似性较高的其他物品

```cpp
std::vector<std::pair<int, ld>> SVD::knn(int uid, int iid)
{
    std::vector<std::pair<int, ld>> res;
    // 这里先用train吧
    ld y1, y2;
    y1 = item2Attri[iid].first;
    y2 = item2Attri[iid].second;
    if (y1 == -1 || y2 == -1)
        return res;
    for (int i = 0; i < trainMatrix[uid].size(); i++)
    {
        int itemId = trainMatrix[uid][i].first;
        int score = trainMatrix[uid][i].second;
        ld x1, x2;
        x1 = item2Attri[itemId].first;
        x2 = item2Attri[itemId].second;
        if (iid == itemId || x1 == -1 || x2 == -1)
            continue;
        res.push_back(std::make_pair(itemId, sim(x1, x2, y1, y2)));
    }
    auto cmp = [&](std::pair<int, ld> &a, std::pair<int, ld> &b)
    {
        return a.second > b.second;
    };
    std::sort(res.begin(), res.end(), cmp);
    if (res.size() > k)
        res.resize(k);
    return res;
}
```

代码关键逻辑为：

1. 创建一个空的结果向量 res，用于存储最相似的物品
2. 获取物品 iid 的属性值 y1 和 y2，分别表示其两个属性的取值
3. 如果 y1 或 y2 为-1（表示属性值未知），则返回空的结果向量 res
4. 遍历用户 uid 在训练数据中评分的物品，对于每个物品执行以下步骤：
   (a) 获取物品 itemId 和对应的评分 score
   (b) 获取物品 itemId 的属性值 x1 和 x2，分别表示其两个属性的取值
   (c) 如果 itemId 与 iid 相同，或者 x1 或 x2 为-1（表示属性值未知），则跳过当前物品
   (d) 计算物品 itemId 与物品 iid 之间的相似度，使用 sim(x1, x2, y1, y2) 函数计算相似度，并将 (itemId, 相似度) 的键值对添加到结果向量 res 中

5. 定义一个自定义的比较函数 cmp，用于对结果向量 res 按照相似度降序进行排序
6. 使用 std::sort 函数对结果向量 res 进行排序，排序依据是相似度，使用自定义的比较函数 cmp 进行比较
7. 如果结果向量 res 的大小大于 k，则将其截断为只包含前 k 个元素

7. 模型训练

我们通过封装 train () 函数来实现 SVD 模型的训练，具体实现如下：

模型训练

```
void SVD::train(std::string trainFile, ld alpha, ld lambda, float rate, int
    iterNum, bool saveModel)
{
    std::cout << "------------------preprocess start----------------" << std
        ::endl;
    processData(trainFile, rate);
    initModelArg();
    if (hasKNN)
        openKNN();
    std::cout << "------------------preprocess over----------------" << std::
        endl;
    std::cout << "------------------train start----------------" << std::endl
        ;
    for (int iter = 1; iter <= iterNum; iter++)
    {
        for (int userId = 0; userId < userNum; userId++)
        {
            for (auto node : trainMatrix[userId])
            {
                int itemId = node.first;
                ld score = node.second;
                ld rui = predictUI(userId, itemId);
                ld eui = score - rui;

                // update arg
                bu[userId] = bu[userId] + alpha * (eui - lambda * bu[userId])
                    ;
                bi[itemId] = bi[itemId] + alpha * (eui - lambda * bi[itemId])
                    ;

                for (int k = 0; k < dim; k++)
                {
                    ld tmp_pu = pu[userId][k];
                    pu[userId][k] = pu[userId][k] + alpha * (eui * qi[itemId
                        ][k] - lambda * pu[userId][k]);
                    qi[itemId][k] = qi[itemId][k] + alpha * (eui * tmp_pu -
                        lambda * qi[itemId][k]);
```

```
                }
            }
            if (userId % 5000 == 0)
            {
                std::cout << "epoch " << iter << " process " << userId << "
                    user" << std::endl;
            }
        }
        // 打印RMSE信息
        std::cout << "epoch" << iter << ": \t";
        calRMSE();
        // 越往后，学习率应该越小，越趋于稳定。
        alpha *= 0.99;
        // lambda *= 1.05;
    }

    // 画图用
    // calRMSEPerItem();
}
```

代码逻辑：

1. **数据预处理步骤：**
   - 调用 `processData` 函数对训练数据进行预处理，划分为训练集和测试集，并初始化模型参数。
   - 如果设置了 KNN，调用 `openKNN` 函数打开 KNN 方法。
2. **迭代训练过程：**
   - 进行 `iterNum` 次迭代。
   - 对于每个用户 `userId`，遍历其在训练集中评分的物品。
   - 对于每个物品，执行以下步骤：
     - 获取物品 ID `itemId` 和对应的评分 `score`。
     - 根据当前模型参数预测用户 `userId` 对物品 `itemId` 的评分，得到预测评分 `rui`。
     - 计算实际评分与预测评分之间的误差 `eui`。
     - 更新模型参数：
       * 更新用户偏置 `bu[userId]`：`bu[userId] = bu[userId] + alpha * (eui - lambda * bu[userId])`
       * 更新物品偏置 `bi[itemId]`：`bi[itemId] = bi[itemId] + alpha * (eui - lambda * bi[itemId])`
       * 对于每个维度 `k`，计算临时变量 `tmp_pu = pu[userId][k]`
       * 更新用户向量 `pu[userId][k]`：`pu[userId][k] = pu[userId][k] + alpha * (eui * qi[itemId][k] - lambda * pu[userId][k])`：
       * 更新物品向量 `qi[itemId][k]`：`qi[itemId][k] = qi[itemId][k] + alpha * (eui * tmp_pu - lambda * qi[itemId][k])`

- 每当 userId 是 5000 的倍数时，输出当前迭代轮次和处理的用户数。
- 在每轮迭代结束后，调用 calRMSE 函数计算并输出训练集和测试集的 RMSE 值。
- 调整学习率 alpha，乘以 0.99，以减小学习率。
- 如果需要保存模型，可以在此处添加保存模型的代码。

3. **训练结束后的额外步骤：**
   - 如果需要保存模型，可以在训练结束后添加保存模型的代码。
   - 调用 calRMSEPerItem 函数计算每个物品的 RMSE 值，以便后续分析使用。

(三) SVD++ 实现

1. 基于 SVD++ 的实现和基于 SVD 实现的主要区别

SVD++ 是在传统的 SVD（奇异值分解）算法的基础上进行改进，通过考虑用户的隐式反馈信息来提高推荐的准确性。二者在实现过程中主要有如下区别：

1. **特征向量的计算：** 在基于 SVD 的代码中，特征向量是通过独立的用户特征和物品特征计算的。而在基于 SVD++ 的代码中，特征向量还考虑了用户在训练集中的交互项，通过累加这些交互项的特征向量来得到增强的特征向量
2. **预测评分计算：** 在基于 SVD 的代码中，预测评分仅考虑用户特征和物品特征的乘积。而在基于 SVD++ 的代码中，预测评分除了考虑用户特征和物品特征的乘积外，还考虑了用户在训练集中的交互项对预测评分的影响
3. **RMSE 计算：** 在基于 SVD 的代码中，RMSE 的计算只考虑了用户对应的评分项。而在基于 SVD++ 的代码中，RMSE 的计算不仅考虑了用户对应的评分项，还考虑了用户在训练集中的交互项

总体上，基于 SVD++ 的代码在预测评分和 RMSE 计算中考虑了更多的因素，尤其是用户在训练集中的交互项，从而可能提供更准确的预测和评估结果。故在 SVD++ 算法的分析中，我们只分析和 SVD 实现中不相同的地方，相同地方不再赘述。

2. 计算整体的 RMSE

在 SVD++ 中，我们通过封装 calRMSE() 函数，来计算 SVD++ 实现下的 RMSE 值，具体实现如下：

计算整体的 RMSE

```
void SVDPP::calRMSE()
{
    int trainNum = 0;
    ld trainRMSE = 0;
    int testNum = 0;
    ld testRMSE = 0;
    ld *z = new ld[dim];

    for (int userId = 0; userId < userNum; userId++)
    {
```

```
        ld yjSum = trainMatrix[userId].size();
        yjSum = sqrt(yjSum);
        memset(z, 0, sizeof(z));
        for (auto [j, s] : trainMatrix[userId])
            for (int k = 0; k < dim; k++)
                z[k] += yj[j][k];
        for (int k = 0; k < dim; k++)
        {
            z[k] /= yjSum;
            z[k] += pu[userId][k];
        }
        trainNum += trainMatrix[userId].size();
        for (auto node : trainMatrix[userId])
        {
            int itemId = node.first;
            ld score = node.second;
            ld rui = mean + bu[userId] + bi[itemId] + product(qi[itemId], z,
                dim);
            rui = stdRui(rui);
            ld eui = score - rui;
            trainRMSE += eui * eui;
        }
    }

    for (int userId = 0; userId < userNum; userId++)
    {
        ld yjSum = trainMatrix[userId].size();
        yjSum = sqrt(yjSum);
        memset(z, 0, sizeof(z));
        for (auto [j, s] : trainMatrix[userId])
            for (int k = 0; k < dim; k++)
                z[k] += yj[j][k];
        for (int k = 0; k < dim; k++)
        {
            z[k] /= yjSum;
            z[k] += pu[userId][k];
        }
        testNum += testMatrix[userId].size();
        for (auto node : testMatrix[userId])
        {
            int itemId = node.first;
            ld score = node.second;
            ld rui = mean + bu[userId] + bi[itemId] + product(qi[itemId], z,
                dim);
            rui = stdRui(rui);
            ld eui = score - rui;
            testRMSE += eui * eui;
        }
```

```
    }
    trainRMSE = sqrt(trainRMSE / trainNum);
    testRMSE = sqrt(testRMSE / testNum);

    std::cout << "train RMSE: " << trainRMSE << "\ttest RMSE: " << testRMSE
        << std::endl;
}
```

关键逻辑解释：

1. 初始化变量：创建变量 trainNum 和 testNum，并将它们初始化为 0，创建变量 trainRMSE 和 testRMSE，并将它们初始化为 0，创建大小为 dim 的数组 z
2. 遍历每个用户：使用 userId 迭代用户。对于每个用户，首先计算用户在训练集中评分的数量，并将其存储在 yjSum 中。然后将数组 z 的元素全部初始化为 0
3. 计算 z 的值：遍历该用户在训练集上的每个评分，其中 j 表示物品 ID，s 表示对应的评分。在循环中，将物品 j 的特征向量 yj[j] 的每个元素与 z 的对应元素相加
4. 更新 z：计算 z 的平均值，首先将其每个元素除以 yjSum，然后将其与用户特征向量 pu[userId] 的对应元素相加，得到更新后的 z
5. 计算训练集 RMSE：遍历该用户在训练集上的每个评分，其中 node 包含物品 ID 和对应的评分。计算预测评分 rui，通过加上 mean、bu[userId]、bi[itemId] 以及物品特征向量 qi[itemId] 与 z 的内积（使用 product 函数）。然后使用 stdRui 函数对 rui 进行标准化处理，计算预测误差 eui，并将其平方后累加到 trainRMSE 中
6. 遍历每个用户（测试集）：使用相同的逻辑，计算该用户在测试集上的评分数量，并遍历每个评分，计算预测评分 rui，计算预测误差 eui，并将其平方后累加到 testRMSE 中
7. 计算 RMSE：将 trainRMSE 和 testRMSE 分别除以相应的评分数量，并对其进行平方根运算，得到训练集和测试集的 RMSE 值

相比于 SVD 算法，基于 SVD++ 算法在计算用户特征向量时考虑了用户在训练集中的交互项（通过累加 yj[j] 到 z 中），并在预测评分时加入了对用户交互项的考虑（通过与 qi[itemId] 的内积）。这些改进旨在提高预测准确性和模型性能。

在基于 SVD++ 的代码中，使用交互项 yj[j] 来增强用户特征向量 pu[userId]，以更好地捕捉用户与物品的交互信息。通过计算 z 的平均值，将用户特征向量和交互项的信息相结合，提供了更丰富的用户表示。

在预测评分时，除了考虑用户和物品的特征向量乘积，还考虑了用户的交互项。通过计算用户特征向量 pu[userId] 和交互项 z 的加权和，并加上全局偏差项（mean、bu[userId]、bi[itemId]），得到最终的预测评分 rui。这样可以更准确地预测用户对物品的评分。

通过在训练集和测试集上计算 RMSE，可以评估基于 SVD++ 的模型在训练数据和测试数据上的预测性能。RMSE 是一种常用的衡量预测误差的指标，通过计算预测误差的平方和的均值，并取平方根，来衡量模型的预测准确性。在这段代码中，分别计算训练集和测试集上的 RMSE，并将结果输出。

综上所述，基于 SVD++ 的代码相比于基于 SVD 的代码，在用户特征的表示和预测评分的计算上引入了用户在训练集中的交互项，以提高模型的表达能力和预测准确性。这些改进使得模型能够更好地捕捉用户和物品之间的复杂关系，从而提高推荐系统的性能。

3. 模型训练

我们通过封装 train () 函数来实现 SVD++ 模型的训练，具体实现如下：

模型训练

```
void SVDPP::train(std::string trainFile, ld alpha, ld lambda, float rate, int iterNum, bool saveModel)
{
    std::cout << "------------------preprocess start----------------" << std::endl;
    processData(trainFile, rate);
    initModelArg();
    std::cout << "------------------preprocess over----------------" << std::endl;
    std::cout << "------------------train start----------------" << std::endl;
    ld *z = new ld[dim];
    for (int iter = 1; iter <= iterNum; iter++)
    {
        for (int userId = 0; userId < userNum; userId++)
        {
            assert(trainMatrix[userId].size() != 0);
            if (trainMatrix[userId].size() == 0)
                continue;
            ld yjSum = trainMatrix[userId].size();
            yjSum = sqrt(yjSum);
            memset(z, 0, sizeof(z));
            for (auto [j, s] : trainMatrix[userId])
                for (int k = 0; k < dim; k++)
                    z[k] += yj[j][k];
            for (int k = 0; k < dim; k++)
            {
                z[k] /= yjSum;
                z[k] += pu[userId][k];
            }
            for (auto node : trainMatrix[userId])
            {
                int itemId = node.first;
                ld score = node.second;

                ld rui = mean + bu[userId] + bi[itemId] + product(qi[itemId], z, dim);
                rui = stdRui(rui);
                ld eui = score - rui;

                // update arg
                bu[userId] = bu[userId] + alpha * (eui - lambda * bu[userId]);
                bi[itemId] = bi[itemId] + alpha * (eui - lambda * bi[itemId])
```

```
                    ;

                // #pragma omp parallel for num_threads(4)
                for (int k = 0; k < dim; k++)
                {
                    pu[userId][k] = pu[userId][k] + alpha * (eui * qi[itemId
                        ][k] - lambda * pu[userId][k]);
                }
                // 这里为了充分利用cache, 把他们三个的更新分开
                for (auto [j, s] : trainMatrix[userId])
                    // #pragma omp parallel for num_threads(4)
                    for (int k = 0; k < dim; k++)
                        yj[j][k] = yj[j][k] + alpha * (eui * qi[itemId][k] /
                            yjSum - lambda * yj[j][k]);
                // #pragma omp parallel for num_threads(4)
                for (int k = 0; k < dim; k++)
                    qi[itemId][k] = qi[itemId][k] + alpha * (eui * z[k] -
                        lambda * qi[itemId][k]);
            }
            if (userId % 1000 == 0)
            {
                std::cout << "epoch " << iter << " process " << userId << "
                    user" << std::endl;
            }
        }
        // 打印RMSE信息
        std::cout << "epoch" << iter << ": \t";
        calRMSE();
        // 越往后, 学习率应该越小, 越趋于稳定。
        alpha *= 0.99;
    }
}
```

SVD++ 模型训练步骤如下：

1. **读取训练数据并进行预处理：**首先，通过调用 processData(trainFile, rate) 函数对训练数据进行处理，其中 trainFile 是训练数据文件的路径，rate 是用于划分训练集和测试集的比例。该函数会将数据加载到 trainMatrix 和 testMatrix 中，并计算平均评分 mean 以及每个用户对应的交互项 yj

2. **初始化模型参数：**通过调用 initModelArg() 函数，初始化模型参数。这里涉及到一些参数的初始化，如用户偏差项 bu、物品偏差项 bi、用户特征向量 pu 和物品特征向量 qi 等

3. **迭代训练：**使用 iterNum 指定的迭代次数，进行模型的训练。在每次迭代中，对每个用户进行处理

4. **计算用户特征向量加权和：**首先，计算每个用户在训练集中的交互项的平方根和 yjSum。然后，对于每个用户，遍历其训练集中的每个物品，计算用户特征向量加权和 z。这里使用了 memset 函数将 z 初始化为 0，并通过累加计算交互项的加权和

5. **预测评分和参数更新：**对于每个用户的每个物品，计算预测评分 rui，并计算预测误差 eui。然后，根据预测误差更新模型参数，包括用户偏差项 bu、物品偏差项 bi、用户特征向量 pu、物品特征向量 qi 以及交互项 yj。更新参数时使用了学习率 alpha 和正则化项 lambda
6. **打印训练进度和计算 RMSE：**在每个迭代的最后，打印当前迭代的进度（处理了多少个用户）并计算并输出训练集和测试集上的 RMSE
7. 调整学习率：在每次迭代后，通过乘以 0.99 来逐渐降低学习率 alpha，以便训练过程趋于稳定

这段代码实现了基于 SVD++ 的模型的训练过程，包括了参数初始化、迭代训练、预测评分和参数更新等步骤。通过多次迭代，模型逐渐学习并优化参数，以提高对用户评分的预测准确性。

（四） ItemCF 实现

在实现 ItemCF 的过程中，我们考虑 deviation 以消除偏差，得到尽可能准确的打分结果。

1. 计算每个物品的平均打分

在读入 train.txt 后，我们调用 calculate_item_avg() 函数来计算每个物品的平均打分

计算每个物品的平均打分

```
void calculate_item_avg() { // get the average score of specific item
    for (int i = 0; i < max_item; i++) {
        if (ratingNum_item[i] != 0) {
            itemAVG[i] /= ratingNum_item[i];
        }
        else {
            rating blank; blank.id = -1; blank.score = 0;
            Data[i].push_back(blank);
            ratingNum_item[i]++;
        }
    }
}
```

代码实现细节：

1. 使用一个循环遍历所有物品，循环变量为 i，从 0 到 max_item - 1
2. 在循环内部，首先检查 ratingNum_item[i] 是否不等于零，即该物品的评分数是否不为零
3. 如果评分数不为零,则执行此操作:将物品 i 的总分 itemAVG[i] 除以评分数 ratingNum_item[i]，得到平均分数
4. 如果评分数为零，则执行以下操作：
   (a) 创建一个临时的 rating 结构体对象 blank
   (b) 将 blank 的 id 成员变量设置为 -1，表示无效值
   (c) 将 blank 的 score 成员变量设置为 0，表示零分数
   (d) 将 blank 添加到 Data[i] 的末尾，即将其作为一个评分数据添加到物品 i 的评分列表中
   (e) 将评分数 ratingNum_item[i] 增加 1，表示物品 i 的评分数增加了一个

2. 计算两个物品之间的 Pearson 相似度

我们封装 calculate_PearsonSim(int item1, int item2) 函数来计算两个物品之间的 Pearson 相似度，具体实现如下：

计算两个物品之间的 Pearson 相似度

```
double calculate_PearsonSim(int item1, int item2) {
    int item_less = (ratingNum_item[item1] <= ratingNum_item[item2]) ? item1
        : item2;
    int item_more = (ratingNum_item[item1] <= ratingNum_item[item2]) ? item2
        : item1;
    int num = (ratingNum_item[item1] <= ratingNum_item[item2]) ?
        ratingNum_item[item1] : ratingNum_item[item2];
    int count = 0;

    if (Data[item_less][0].id == -1 || Data[item_more][0].id == -1) {
        return 0;
    }
    // calculate the numerator
    double numerator = 0;
    for (int i = 0; i < num; i++) {
        int iter = binarySearch(Data, item_more, ratingNum_item[item_more],
            Data[item_less][i].id);
        if (iter != -1) {
            count++;
            numerator += (Data[item_more][iter].score - itemAVG[item_more]) *
                (Data[item_less][i].score - itemAVG[item_less]);
        }
    }
    // only consider items rated by over 20 users to avoid too much bias
    if (count < 20) {
        return 0;
    }
    // calculate the denominator
    double sum1 = 0;
    for (int i = 0; i < ratingNum_item[item1]; i++) {
        sum1 += (Data[item1][i].score - itemAVG[item1]) * (Data[item1][i].
            score - itemAVG[item1]);
    }
    double sqrt1 = sqrt(sum1);

    double sum2 = 0;
    for (int i = 0; i < ratingNum_item[item2]; i++) {
        sum2 += (Data[item2][i].score - itemAVG[item2]) * (Data[item2][i].
            score - itemAVG[item2]);
    }
    double sqrt2 = sqrt(sum2);

    if (sqrt1 == 0 || sqrt2 == 0) {
```

```
        return 0;
    }

    double similarity = numerator / (sqrt1 * sqrt2);
    return similarity;
}
```

代码核心逻辑：

1. 定义一个函数 calculate_PearsonSim，用于计算物品之间的 Pearson 相似度
2. 根据评分数的大小，将物品编号 item1 和 item2 分为较少评分数的物品和较多评分数的物品
3. 如果较少评分数的物品或较多评分数的物品的第一个评分数据的 id 为-1（无效值），则返回 0
4. **计算相似度的分子部分：**使用循环遍历较少评分数的物品的评分数据，在较多评分数的物品的评分数据中使用二分查找，找到匹配的评分数据，增加 count 计数器，并计算评分数据的差值乘积并累加到 numerator 中
5. 如果有效的用户评分数量小于 20，则返回 0，以避免过多的偏差
6. 计算相似度的分母部分：使用循环遍历较少评分数的物品的评分数据，计算评分数据与该物品的平均分的差值的平方并累加到相应的求和变量中。计算分母的平方根
7. 计算相似度的值：将分子除以分母的乘积，并将结果赋给变量 similarity 作为两个物品的相似度

3. 计算两个物品之间的属性相似度

我们定义 calculate_AttributeSim(int item1, int item2) 函数来计算 item1 和 item2 之间的属性相似度

计算两个物品之间的属性相似度

```
double calculate_AttributeSim(int item1, int item2) {
    if (attrList[item1].norm == 0 || attrList[item2].norm == 0) {
        return 0;
    }
    double a = (attrList[item1].attr1 * attrList[item2].attr1 + attrList[
        item1].attr2 * attrList[item2].attr2);
    double b = (attrList[item1].norm * attrList[item2].norm);
    double attribute_similarity = a / b;
    return attribute_similarity;
}
```

代码解释：calculate_AttributeSim 函数用于计算物品之间的属性相似度。我们将物品的两个属性视为该物品的特征向量，则两个特征向量的余弦相似度就是物品间的属性相似度。如果任意一个物品的属性归一化值为 0，直接返回 0。

4. 结合 Pearson 相似度和属性相似度，综合计算两个物品之间的相似度

在函数 calculate_Sim(int item1, int item2) 中，我们结合 Pearson 相似度和属性相似度，综合计算两个物品之间的相似度，具体实现如下：

结合 Pearson 相似度和属性相似度，综合计算两个物品之间的相似度

```
double calculate_Sim(int item1, int item2) {
    int item_min = min(item1, item2);
    int item_max = max(item1, item2);
    int iter = binarySearch(SimMap, item_min, SimMap[item_min].size(),
        item_max);
    if (iter != -1) {
        return SimMap[item_min][iter].score;
    }
    double Pearson, Similarity, Attribute = 0;
    Pearson = calculate_PearsonSim(item1, item2);
    if (consider_attr)
        Attribute = calculate_AttributeSim(item1, item2);
    Similarity = (consider_attr==true)?(Pearson + Attribute)/2:Pearson;

    rating similarity1; similarity1.id = item_max; similarity1.score =
        Similarity;
    SimMap[item_min].push_back(similarity1);
    return Similarity;
}
```

核心逻辑：

1. 通过比较物品编号的大小，确定较小编号为 item_min，较大编号为 item_max
2. 利用二分查找在 SimMap 中寻找是否已经计算过这两个物品之间的相似度，如果有对应的相似度记录，则直接返回相似度值
3. 如果没有对应的相似度记录，则分别调用 calculate_PearsonSim 和 calculate_AttributeSim 函数计算 Pearson 相似度和属性相似度，并根据是否考虑属性相似度的设置，计算综合相似度
4. 将计算得到的相似度存储到 SimMap 中，并返回相似度的值

5. 预测用户对物品的打分

我们在 predict(int user_id, int item_id) 函数中预测用户对物品的打分。打分结果充分考虑了 deviation 对分数的影响。具体实现如下：

预测用户对物品的打分

```
double predict(int user_id, int item_id) {
    double sumSim = 0;
    double sumSim_item = 0;
    for (int i = 0; i < max_item; i++) {
        if (i == item_id)
```

```
            continue;
        int iter = binarySearch(Data, i, ratingNum_item[i], user_id);
        if (iter != -1) {
            double sim = calculate_Sim(item_id, i);
            if (sim > 0) {
                // consider deviations
                double baseline = avg + (userAVG[user_id] - avg) + (itemAVG[i
                    ] - avg);
                sumSim += sim;
                sumSim_item += sim * (Data[i][iter].score - baseline);
            }
        }
    }
    if (sumSim != 0 && sumSim_item != 0) {
        // consider deviations
        double baseline = avg + (userAVG[user_id] - avg) + (itemAVG[item_id]
            - avg);
        double predict_score = baseline + sumSim_item / sumSim;
        if (predict_score > 100) {
            return 100;
        }
        else if (predict_score < 0) {
            return 0;
        }
        return predict_score;
    }
    else {
        return userAVG[user_id];
    }
}
```

1. 初始化变量 sumSim 和 sumSim_item 为 0，用于计算相似度的加权和以及相似度与评分偏差的乘积的加权和
2. 使用循环遍历所有物品，排除目标物品 item_id
3. 在物品 i 的评分数据中使用二分查找，查找用户 user_id 的评分数据
4. 如果找到匹配的评分数据，计算物品 item_id 和物品 i 之间的相似度 sim
5. 如果相似度大于 0，则考虑偏差项，并计算基线值 baseline
6. 更新相似度的加权和 sumSim 和相似度与评分偏差的乘积的加权和 sumSim_item
7. 如果 sumSim 和 sumSim_item 都不为 0，则计算预测评分
8. 考虑偏差项，并计算基线值 baseline
9. 据计算得到的预测评分 predict_score 进行范围限制，如果超过 100，则返回 100，如果低于 0，则返回 0

10. 如果 sumSim 和 sumSim_item 都为 0，表示无法进行准确的预测，返回用户 user_id 的平均评分 userAVG[user_id]

（五） UserCF 实现

在本部分，和 ItemCF 实现中重叠的部分，我们不予赘述，我们只讲解 UserCF 和 ItemCF 算法实现中的不同点。

1. 计算用户之间的 Pearson 相关系数

在计算 Pearson 相关系数的 calculate_PearsonSim(int user1, int user2) 函数中，我们计算用户之间的 Pearson 相关系数用于衡量两个用户之间的相似度，而非像 ItemCF 中计算物品间的相似度。具体实现如下：

计算用户之间的 Pearson 相关系数

```
double calculate_PearsonSim(int user1, int user2) {
    int user_less = (ratingNum[user1] <= ratingNum[user2]) ? user1 : user2;
    int user_more = (ratingNum[user1] <= ratingNum[user2]) ? user2 : user1;
    int num = (ratingNum[user1] <= ratingNum[user2]) ? ratingNum[user1] :
        ratingNum[user2];

    // calculate the numerator
    double numerator = 0;
    for (int i = 0; i < num; i++) {
        int j = binarySearch(Data, user_more, ratingNum[user_more], Data[
            user_less][i].id);
        if (j != -1) {
            numerator += (Data[user_less][i].score - userAVG[user_less]) * (
                Data[user_more][j].score - userAVG[user_more]);
        }
    }

    // calculate the denominator
    double sum1 = 0;
    for (int i = 0; i < ratingNum[user1]; i++) {
        sum1 += (Data[user1][i].score - userAVG[user1]) * (Data[user1][i].
            score - userAVG[user1]);
    }
    double sqrt1 = sqrt(sum1);

    double sum2 = 0;
    for (int i = 0; i < ratingNum[user2]; i++) {
        sum2 += (Data[user2][i].score - userAVG[user2]) * (Data[user2][i].
            score - userAVG[user2]);
    }
    double sqrt2 = sqrt(sum2);

    if (sqrt1 == 0 || sqrt2 == 0) {
```

```
        return 0;
    }

    double similarity = numerator / (sqrt1 * sqrt2);
    return similarity;
}
```

核心逻辑:

1. **确定用户数量和评分数量：**根据用户 user1 和用户 user2 的评分数量，确定评分较少的用户为 user_less，评分较多的用户为 user_more，并记录评分数量为 num
2. **计算分子：**遍历评分数量 num 次，通过二分查找在用户 user_more 的评分数据中找到与用户 user_less 对应的物品评分，计算评分差值的乘积，并累加到分子 numerator 中
3. **计算分母：**分别计算用户 user1 和用户 user2 的评分差值平方和, 分别累加到 sum1 和 sum2 中
4. **判断除数为 0：**如果 sqrt1 或 sqrt2 为 0，表示其中一个用户没有评分差值，无法计算相似度，返回 0
5. **计算相似度：**根据分子 numerator、sqrt1 和 sqrt2 计算皮尔逊相关系数，将分子除以分母得到相似度 similarity
6. **返回结果：**返回计算得到的相似度 similarity 作为两个用户之间的相似度指标

总结来说，该函数根据用户评分数据，计算了两个用户之间的皮尔逊相关系数，用于衡量它们之间的相似度。相似度的值在-1 到 1 之间，值越接近 1 表示两个用户越相似，值越接近-1 表示两个用户越不相似，值为 0 表示两个用户之间没有线性相关性。

2. 预测用户对物品的评分

UserCF 打分过程中也充分考虑了 deviation 对分数的影响。具体实现如下

预测用户对物品的评分

```
double predict(int user_id, int item_id) {
    double sumSim = 0;
    double sumSim_item = 0;
    for (int i = 0; i < user; i++) {
        if (i == user_id)
            continue;
        int j = binarySearch(Data, i, ratingNum[i], item_id);
        if (j != -1) {
            double sim = calculate_PearsonSim(user_id, i);
            if (sim > 0) {
                double baseline = avg + (userAVG[i] - avg) + (get_item_avg(
                    Data[i][j].id) - avg);
                sumSim += sim;
                sumSim_item += sim * (Data[i][j].score - baseline);
            }
        }
```

```
    }
    if (sumSim != 0 && sumSim_item != 0) {
        double baseline = avg + (userAVG[user_id] - avg) + (get_item_avg(
            item_id) - avg);
        double predict_score = baseline + sumSim_item / sumSim;
        if (predict_score > 100) {
            return 100;
        }
        else if (predict_score < 0) {
            return 0;
        }
        return predict_score;
    }
    else {
        return userAVG[user_id];
    }
}
```

代码核心逻辑：

1. **初始化变量：** 设置初始值 sumSim 和 sumSim_item 为 0，用于累加相似度和相似度乘以评分差值的和
2. **遍历用户：** 对于每个用户 i，跳过当前用户 user_id
3. **查找评分：** 通过二分查找 binarySearch() 函数，在用户 i 的评分数据中查找是否存在物品 item_id 的评分信息
4. **计算相似度：** 如果找到评分信息，计算用户 user_id 和用户 i 之间的相似度 sim，使用 calculate_PearsonSim() 函数
5. **计算基线值：** 如果相似度大于 0，计算基线值 baseline，包括整体平均评分 avg、用户 i 的平均评分 userAVG[i]、物品 Data[i][j].id 的平均评分
6. **累加相似度和差值：**将相似度 sim 累加到 sumSim 中,将相似度乘以评分差值 (Data[i][j].score - baseline) 累加到 sumSim_item 中
7. **预测评分：** 如果 sumSim 和 sumSim_item 都不为 0，计算基线值 baseline，然后计算预测评分 predict_score，包括基线值加上相似度乘以评分差值除以相似度的结果
8. **范围限制：** 如果预测评分大于 100，将其限制为 100；如果预测评分小于 0，将其限制为 0
9. **返回结果：** 返回预测评分结果。如果 sumSim 和 sumSim_item 都为 0，说明没有可用的相似度和评分差值信息，返回用户 user_id 的平均评分 userAVG[user_id] 作为预测结果

总结来说，通过遍历其他用户的评分信息，计算相似度并加权预测评分，基于用户历史评分和相似度的差异来预测用户对特定物品的评分。

五、 实验结果分析

四个算法的训练时间、预测时间、空间消耗与 RMSE 信息如表4所示，算法结合 ItemAttribute.txt 后的情况如表5所示

(一) 基本结果分析

表 4: 通过四种算法实现推荐系统的结果对比分析

算法名称	训练时间	预测时间	空间消耗	RMSE
SVD	201.045s(30epoch)	6.467s	1480M	25.6202
SVD++	14214s(30epoch)	14.806s	101M	33.4657
ItemCF	-	6.5h	160M	28.61
UserCF	-	1.8h	63M	32.437

表 5: 是否使用 ItemAttribute.txt 以及使用后效果是否得到提升

算法名称	是否使用	效果是否得到提升
SVD	是	25.6202->25.6016
SVD++	否 (SVD++ 训练慢且效果差)	-
ItemCF	是	28.61->34.075
UserCF	否	-

从表中看出，在我们实现的四种算法中，SVD 算法虽然空间消耗最大，却在预测时间与预测准确度上都有着最佳的表现，且使用 ItemAttribute.txt 考虑物品属性后性能也得到了提升。与之相反，协同过滤算法在当前数据集中具有着较差的表现，考虑属性后的效果也并不理想。

六、 实验中遇到的问题及对应的解决方案

在本次大作业中，我们充分考虑了实验过程中可能遇到的问题，并针对性的提出应对解决方案，本次实验中，我们遇到的问题及对应的解决方案如下：

1. **数据稀疏性问题：**推荐系统中常常存在数据稀疏性，即用户和物品之间的评分数据非常有限。这可能导致算法难以准确地捕捉到用户和物品之间的关联关系。解决方案包括：

   (a) 使用降维技术（如 SVD）来减少数据的维度，提高推荐的准确性

   (b) 使用正则化技术（如岭回归或 Lasso 回归）来缓解数据稀疏性问题

   (c) 结合其他信息，如用户的历史行为、物品的属性等，来增强推荐的准确性

2. **冷启动问题：**冷启动是指在推荐系统中遇到新用户或新物品时的问题。由于缺乏用户或物品的历史评分数据，难以进行个性化推荐。解决方案包括：

   (a) 对于新用户，可以采用基于内容的推荐方法，根据用户的个人信息、偏好等特征进行推荐

   (b) 对于新物品，可以利用物品的属性信息、标签等进行推荐，或者采用基于流行度的推荐方法，推荐热门物品给用户

3. **算法效率问题：**一些推荐算法可能在处理大规模数据集时效率较低。解决方案包括：

   (a) 优化算法实现，利用并行计算、矩阵分解技术等提高算法的效率

   (b) 使用近似算法或采样方法来减少计算量，同时保持较高的推荐准确性

4. **过拟合问题：**在使用基于模型的推荐算法时，可能会遇到过拟合问题，即模型在训练集上表现良好，但在测试集或实际应用中表现不佳。解决方案包括：

(a) 引入正则化项（如 L2 正则化）来惩罚模型的复杂度，防止过拟合

(b) 使用交叉验证技术来选择合适的模型参数，避免过拟合

(c) 增加数据的多样性，包括引入新的用户和物品，扩大训练集的规模，减少过拟合

七、 参考文献

1. Hoecker A, Kartvelishvili V. SVD approach to data unfolding[J]. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 1996, 372(3): 469-481.

2. Kalman D. A singularly valuable decomposition: the SVD of a matrix[J]. The college mathematics journal, 1996, 27(1): 2-23.

3. Abdi H. Singular value decomposition (SVD) and generalized singular value decomposition[J]. Encyclopedia of measurement and statistics, 2007, 907: 912.

4. Wu Y, Li Y, Qian R. NE-UserCF: Collaborative filtering recommender system model based on NMF and E2LSH[J]. International Journal of Performability Engineering, 2017, 13(5): 610.

5. Thangarasu N, Rajalakshmi R, Manivasagam G, et al. Performance of re-ranking techniques used for recommendation method to the user CF-Model[J]. International Journal of Data Informatics and Intelligent Computing, 2022, 1(1): 30-38.

6. Zheng Q. A novel collaborative filtering algorithm by making recommendations from curious users[C]//International Conference on Computer Graphics, Artificial Intelligence, and Data Processing (ICCAID 2022). SPIE, 2023, 12604: 1169-1178.

7. 潘丽芳, 张大龙, 李慧. 基于用户的协同过滤 (UserCF) 新闻推荐算法研究 [J]. 山西师范大学学报: 自然科学版, 2018, 32(4): 26-30.

8. 朱郁筱, 吕琳媛. 推荐系统评价指标综述 [J]. 电子科技大学学报, 2012, 41(002): 163-175.

9. 朱扬勇, 孙婧. 推荐系统研究进展 [J]. 计算机科学与探索, 2015, 9(5): 513-525.

10. 吴丽花, 刘鲁. 个性化推荐系统用户建模技术综述 [J]. 情报学报, 2006, 25(1): 55-62.

11. 朱冲, 蔡其锑. 基于 SVD 协同过滤的餐饮推荐系统设计 [J]. 浙江树人大学学报 (自然科学版), 2018 (2): 1-5.