实验报告：基于协同过滤算法的用户相似性计算与推荐系统

姓名：蔡卓强 学号：521021910445

1. 实验背景

1.1 基于物品的协同过滤

- 算法原理：基于用户的历史行为，找出与目标用户兴趣相似的一组用户，然后利用这组用户的行为来预测目标用户对物品的喜好程度
- 算法步骤：
    1. 计算用户之间的相似度矩阵，相似度可取余弦相似度、皮尔逊相关系数等；
    2. 找出与目标用户最相似的K个用户；
    3. 综合这K个用户对物品的评分，预测目标用户对未评过的物品的评分
- 优点：简单直观，易于实现
- 缺点：在用户数目庞大时计算复杂度高，且对新用户和长尾物品推荐效果不佳

1.2 基于用户的协同过滤

- 算法原理：基于物品之间的相似度来进行推荐。如果一个用户喜欢某个物品，那么和这个物品相似的其他物品也可能会被用户喜欢
- 算法步骤：
    1. 计算用户之间的相似度矩阵，相似度可取余弦相似度、皮尔逊相关系数等；
    2. 对于目标用户，根据已评分的物品和这些物品的相似度，预测他对未评过的物品的评分
- 优点：计算复杂度相对较低，适合于大规模数据
- 缺点：对物品数目庞大的情况下计算复杂度高，且不太适用于新物品的推荐

2. 实验说明

> 在运行代码时，需要将当前目录设置为`CollaborativeFiltering`的根目录。
>
> 更加详细的提交记录，可以参考Github仓库。

通过实现一个用户协同过滤推荐系统，掌握基本的推荐算法原理和实现方法，了解推荐系统的关键步骤和评价指标。同时，本次实验额外实现了基于物品协同过滤的推荐系统，由于不是本次实验的重点，相应代码只会在结尾简要说明。

3. 基于用户的协同过滤

3.1 数据读取与处理

通过 `UserCF` 类的构造函数读取数据文件并解析。数据文件的每一行包含用户ID、物品ID和评分。

```
std::ifstream f(dataPath, std::ios::in);
if (!f.good()) {
    throw std::runtime_error("open file '" + dataPath + "' error");
}
```

3.2 数据集划分

`divideDataset` 函数用于将原始数据集划分为训练集和测试集。通过指定一个分割比例 `pivot`，将每个用户的评分随机分配到训练集或测试集中。

3.2.1代码及注释

```
static std::tuple<mat_t<int>, mat_t<int>> divideDataset(const mat_t<int> &data, int pivot)
{
    // 检查 pivot 是否在有效范围内
    if (pivot < 0 || pivot > 100) {
        throw std::runtime_error("pivot should be in [0, 100]");
    }
    // 获取每个用户的电影数量和用户数量
    const size_t nItems = data.front().size();
    const size_t nUsers = data.size();
    // 初始化随机数生成器和分布
    std::default_random_engine randomEngine;
    std::uniform_int_distribution<int> distribution(0, 99);
    // 初始化训练集和测试集矩阵，并预留空间
    mat_t<int> trainSet, testSet;
    trainSet.reserve(nUsers);
    testSet.reserve(nUsers);
    // 遍历每个用户的数据
    for (const auto &d : data) {
        // 初始化当前用户的训练集和测试集行
        vec_t<int> trainLine(nItems, 0), testLine(nItems, 0);
        // 遍历当前用户的每个电影评分
        for (size_t i = 0; i < nItems; i++) {
            // 根据 pivot 随机决定该评分是属于训练集还是测试集
            if (distribution(randomEngine) < pivot) {
                trainLine[i] = d[i]; // 加入训练集
            } else {
                testLine[i] = d[i]; // 加入测试集
            }
        }
        // 将当前用户的行加入训练集和测试集矩阵
        trainSet.emplace_back(std::move(trainLine));
        testSet.emplace_back(std::move(testLine));
    }
    // 返回训练集和测试集
    return {trainSet, testSet};
}
```

3.3 用户相似性计算

3.3.1皮尔逊相似度计算

皮尔逊相似度是通过计算两个用户评分之间的线性相关性来衡量相似度。计算过程包括以下步骤：

1. **计算每个用户的平均评分**：对每个用户的所有评分求平均值。
2. **找到共同评分的电影**：找出两个用户都评分过的电影。
3. **计算评分的差值**：对每个共同评分的电影，计算每个用户的评分与其平均评分的差值。
4. **计算分子和分母**：
   - 分子：所有共同评分的电影的评分差值的乘积之和。
   - 分母：两个用户评分差值的平方和的平方根的乘积。

皮尔逊相似度的计算公式如下：

$$\text{similarity} = \frac{\sum_{i \in I} (r_{ui} - \bar{r}_u) \cdot (r_{vi} - \bar{r}_v)}{\sqrt{\sum_{i \in I} (r_{ui} - \bar{r}_u)^2} \cdot \sqrt{\sum_{i \in I} (r_{vi} - \bar{r}_v)^2}}$$

其中，r_{ui}和r_{vi}分别表示用户u和用户v对电影i的评分，$\bar{r}_u$和$\bar{r}_v$分别表示用户u和用户v的平均评分，I表示共同评分的电影集合。

皮尔逊相似度计算的代码实现如下：

```
// 计算用户间的皮尔逊相似度
[[maybe_unused]] auto computeUserPearsonSimilarity = [&](size_t user1, size_t user2) {
    // 计算每个用户的平均分和评分过的电影列表
    vec_t<double> averageScore(nUsers, 0); // 存储每个用户的平均评分
    mat_t<int> scoredFilms(nUsers); // 存储每个用户评分过的电影列表

    for (size_t user = 0; user < nUsers; user++) {
        double sumScore = 0; // 累积评分总和
        int count = 0; // 评分电影的数量

        for (size_t film = 0; film < nItems; film++) {
            if (data[user][film] != 0) { // 如果用户对该电影评分不为0
                sumScore += data[user][film]; // 将评分累加到总和
                scoredFilms[user].emplace_back(film); // 记录该电影
                count++;
            }
        }

        // 计算用户的平均评分，如果该用户没有评分，则平均评分为0
        averageScore[user] = count > 0 ? sumScore / count : 0;
    }

    // 获取用户1和用户2评分过的电影列表
    const auto &films1 = scoredFilms[user1];
    const auto &films2 = scoredFilms[user2];

    // 找到用户1和用户2共同评分的电影
    vec_t<int> commonFilms;
    set_intersection(films1.begin(), films1.end(), films2.begin(), films2.end(),
back_inserter(commonFilms));

    double numerator = 0, denominator1 = 0, denominator2 = 0;

    // 计算皮尔逊相似度的分子和分母
    for (const auto &film: commonFilms) {
        double diff1 = data[user1][film] - averageScore[user1]; // 用户1的评分与平均评分的差值
        double diff2 = data[user2][film] - averageScore[user2]; // 用户2的评分与平均评分的差值
        numerator += diff1 * diff2; // 分子：差值乘积的和
        denominator1 += diff1 * diff1; // 分母的一部分：用户1差值平方和
        denominator2 += diff2 * diff2; // 分母的另一部分：用户2差值平方和
    }

    // 计算最终的皮尔逊相似度
    double similarity = (denominator1 == 0 || denominator2 == 0) ? 0 : numerator /
sqrt(denominator1 * denominator2);
    similarityMatrix[user1][user2] = similarity; // 更新相似度矩阵
    similarityMatrix[user2][user1] = similarity; // 对称更新相似度矩阵
};
```

3.3.2 余弦相似度计算

余弦相似度通过计算两个用户评分向量之间的夹角来衡量相似度，主要关注评分的方向，而不是评分的绝对值。计算过程包括以下步骤：

1. **计算评分向量的点积**：对两个用户的所有评分进行点积运算。
2. **计算评分向量的模**：分别计算两个用户评分向量的模（平方和的平方根）。
3. **计算相似度**：点积除以两个评分向量的模的乘积。

余弦相似度的计算公式如下：

$$\text{similarity} = \frac{\sum_{i \in I} r_{ui} \cdot r_{vi}}{\sqrt{\sum_{i \in I} r_{ui}^2} \cdot \sqrt{\sum_{i \in I} r_{vi}^2}}$$

其中，r_{ui}和r_{vi}分别表示用户u和用户v对电影i的评分，表示所有评分的电影集合。

余弦相似度计算的代码实现如下：

```
// 计算用户间的余弦相似度
[[maybe_unused]] auto computeUserCosineSimilarity = [&](size_t user1, size_t user2) {
    double numerator = 0, denominator1 = 0, denominator2 = 0;

    // 计算余弦相似度的分子和分母
    for (size_t film = 0; film < nItems; film++) {
        double score1 = data[user1][film]; // 用户1对电影的评分
        double score2 = data[user2][film]; // 用户2对电影的评分
        numerator += score1 * score2; // 分子：评分乘积的和
        denominator1 += score1 * score1; // 分母的一部分：用户1评分平方和
        denominator2 += score2 * score2; // 分母的另一部分：用户2评分平方和
    }

    // 计算最终的余弦相似度
    double similarity = (denominator1 == 0 || denominator2 == 0) ? 0 : numerator /
(sqrt(denominator1) * sqrt(denominator2));
    similarityMatrix[user1][user2] = similarity; // 更新相似度矩阵
    similarityMatrix[user2][user1] = similarity; // 对称更新相似度矩阵
};
```

3.3.3对比与分析

皮尔逊相似度

1. **优点**：
   - 考虑了评分的平均值，可以有效消除评分偏差的影响。
   - 更适合有显著评分偏移的情况。
2. **缺点**：
   - 计算较复杂，需要计算每个用户的平均评分和评分差值。
   - 当评分数量较少或没有共同评分时，相似度计算可能不准确。
3. **性能**：
   - 相对较快，因为只需要计算共同评分的电影。
4. **命中率**：
   - 由于考虑了评分偏差，命中率可能较低，但对于某些数据集，推荐的准确性较高。

余弦相似度

1. **优点**：
   - 计算简单，只需要计算评分向量的点积和模。
   - 不考虑评分的平均值，更适合评分稀疏的数据集。
2. **缺点**：
   - 对于评分偏差较大的情况，可能不够准确。
   - 如果用户的评分集中在某个区间，余弦相似度可能会失去区分度。
3. **性能**：
   - 计算较慢，因为需要对所有评分的电影进行计算。

4. **命中率**：
    - 通常命中率较高，因为简单的向量相似度计算能更好地捕捉用户之间的相似性。

3.4寻找最相似的前K个用户

`getKSimilarMatrix` 函数用于从给定的用户相似度矩阵中提取每个用户的前 `mNSimilar` 个最相似的用户，并返回一个矩阵，其中包含这些相似用户的相似度和用户索引。

3.4.1代码及注释

```
[[nodiscard]] mat_t<std::pair<double, size_t>> getKSimilarMatrix(const mat_t<double>
&similarMatrix) const {
    // 获取用户数量
    const size_t nUsers = similarMatrix.size();
    // 初始化结果矩阵，每个用户有一个向量用于存储前mNSimilar个最相似的用户及其相似度
    mat_t<std::pair<double, size_t>> result(nUsers);

    // 遍历每个用户
    for (size_t user = 0; user < nUsers; user++) {
        // 用于存储当前用户与其他所有用户的相似度
        std::vector<std::pair<double, size_t>> kSimUsers;
        // 遍历其他所有用户
        for (size_t other = 0; other < nUsers; other++) {
            // 跳过自身
            if (user != other) {
                // 将相似度和用户索引作为对存入kSimUsers
                kSimUsers.emplace_back(similarMatrix[user][other], other);
            }
        }
        // 对kSimUsers部分排序，提取前mNSimilar个最相似用户
        std::partial_sort(kSimUsers.begin(), kSimUsers.begin() + mNSimilar,
kSimUsers.end(), std::greater<>());
        // 将前mNSimilar个相似用户存入结果矩阵
        result[user] = vec_t<std::pair<double, size_t>>(kSimUsers.begin(),
kSimUsers.begin() + mNSimilar);
    }
    // 返回结果矩阵
    return result;
}
```

3.5 推荐生成

`recommend` 函数根据给定用户的相似用户矩阵 `kSimilarMatrix`，生成推荐电影列表。该函数会根据相似用户的评分，为当前用户推荐评分最高的 `mNRecommend` 部电影。

3.5.1代码及注释

```
std::vector<std::pair<double, size_t>> recommend(const mat_t<std::pair<double, size_t>>
&kSimilarMatrix, const int &user) const {
    // 获取当前用户的相似用户列表
    auto &simUsers = kSimilarMatrix[user];
    // 获取当前用户的评分记录
    auto &userScore = mTrainSet[user];
    // 获取电影数量
    const size_t nFilms = userScore.size();
    // 记录当前用户已观看的电影
    std::unordered_set<int> watchedFilm;
    for (int film = 0; film < nFilms; film++) {
        if (userScore[film]) {
            watchedFilm.insert(film);
```

```
        }
    }
    // 初始化电影评分数组
    std::vector<double> filmScore(nFilms, 0);
    // 遍历相似用户，累加加权评分
    for (const auto &[weight, other]: simUsers) {
        // 获取相似用户的评分记录
        auto &otherScores = mTrainSet[other];
        for (int film = 0; film < nFilms; film++) {
            // 跳过当前用户已观看的电影
            if (watchedFilm.count(film)) {
                continue;
            }
            // 获取相似用户对当前电影的评分
            auto score = otherScores[film];
            if (score != 0) {
                // 加权累加评分
                filmScore[film] += weight * score;
            }
        }
    }
    // 创建包含电影评分及其索引的向量
    std::vector<std::pair<double, size_t>> relatedFilms;
    for (size_t i = 0; i < nFilms; i++) {
        relatedFilms.emplace_back(filmScore[i], i);
    }
    // 部分排序，提取前 mNRecommend 个评分最高的电影
    std::partial_sort(relatedFilms.begin(), relatedFilms.begin() + mNRecommend,
relatedFilms.end(), std::greater<>());
    // 调整大小为 mNRecommend
    relatedFilms.resize(mNRecommend);
    // 返回推荐结果
    return relatedFilms;
}
```

3.6推荐系统评估

`test` 函数对推荐系统进行评估，计算命中率（hit-rate）。

3.6.1代码及注释

```
[[nodiscard]] double test() const {
    // 获取训练集中的用户数量
    const size_t nUsers = mTrainSet.size();
    // 初始化命中计数器
    size_t hit = 0;
    // 计算用户之间的相似度矩阵
    const auto similarityMatrix = calculateSimilarity(mTrainSet);
    // 获取每个用户的前 mNSimilar 个最相似的用户
    const auto kSimilarMatrix = getKSimilarMatrix(similarityMatrix);
    // 遍历每个用户
    for (size_t user = 0; user < nUsers; user++) {
        // 为当前用户生成推荐电影列表
        auto recommendFilms = recommend(kSimilarMatrix, int(user));
        // 检查推荐的电影是否在测试集中出现
        for (const auto &[weight, film] : recommendFilms) {
            if (mTestSet[user][film] != 0) {
                hit += 1; // 如果推荐的电影在测试集中出现，则命中计数器加1
            }
        }
```

```
    }
    // 计算推荐的总数
    size_t recommendCount = mNRecommend * nUsers;
    // 计算命中率
    double hitRate = double(hit) / double(recommendCount);
    // 返回命中率
    return hitRate;
}
```

4. 实验结果

本实验通过计算余弦相似度并生成推荐列表，最终对推荐系统的命中率进行了评估。以下是测试结果的关键输出：

```
auto t0 = std::chrono::system_clock::now();
std::cout <<"nSimilar : 50 , nRecommend : 5\n";
UserCF userCf("./data/ratings.dat", 50, 5);

auto t1 = std::chrono::system_clock::now();
std::cout << "hit-rate = " << userCf.test() << '\n';
auto t2 = std::chrono::system_clock::now();

auto dt1 = double(std::chrono::duration_cast<std::chrono::milliseconds>(t1 - t0).count()) /
1000;
auto dt2 = double(std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count()) /
1000;

std::cout << dt1 << "s for loading data\n" << dt2 << "s for testing hit rate\n";
```

实验结果分析

从实验结果可以看出，余弦相似度的命中率较高，但计算耗时较长。具体实验数据如下：

- **命中率**：0.218709
- **加载数据时间**：0.69秒
- **测试命中率时间**：5.457秒

这表明使用余弦相似度能够提供较高的推荐准确性，但在大规模数据集上计算效率较低。

结论

1. **命中率与效率权衡**：
    - 皮尔逊相似度适用于对效率要求较高的场景，尤其是当用户评分存在明显偏差时。
    - 余弦相似度适用于对推荐准确性要求较高的场景，但计算时间较长，需要在性能和准确性之间进行权衡。
2. **推荐系统优化方向**：
    - 可以结合两种相似度计算方法，根据实际情况选择合适的相似度计算方式。
    - 引入并行计算或分布式计算，提升相似度计算的效率。
    - 结合更多上下文信息或时间因素，进一步提高推荐系统的命中率和准确性。

5.基于物品的协同过滤（`bonus`）

> 由于基于物品的协同过滤与基于用户的协同过滤的大体流程与函数都大致相同，本章节只着重介绍几个核心函数：
>
> 1. `calculateSimilarity` 函数通过统计物品间的共现次数并进行归一化计算，生成物品相似度矩阵。
> 2. `getKSimilarMatrix` 函数从相似度矩阵中提取每个物品的前 `mNSimilar` 个相似物品。

3. `recommend` 函数根据用户的评分和相似物品矩阵生成推荐列表。

1. `calculateSimilarity` 函数

```
static mat_t<double> calculateSimilarity(const mat_t<int>& data) {
    // 如果输入数据为空，抛出异常
    if (data.empty()) {
        throw std::runtime_error("empty data");
    }

    // 获取物品数量
    size_t nItems = data.front().size();
    // 初始化每个物品的评分次数
    vec_t<int> popularItems(nItems, 0);
    // 初始化物品间的相似度矩阵
    mat_t<double> similarityMatrix(nItems, vec_t<double>(nItems, 0));

    // 遍历每个用户的评分记录
    for (const auto& user : data) {
        // 检查每个用户的评分记录是否与物品数量一致
        if (user.size() != nItems) {
            throw std::runtime_error("invalid data");
        }

        // 记录用户评分的物品
        std::vector<size_t> scoredFilms;
        for (size_t film = 0; film < nItems; film++) {
            if (user[film] != 0) {
                // 统计每个物品的评分次数
                popularItems[film] += 1;
                // 记录用户评分的物品
                scoredFilms.emplace_back(film);
            }
        }

        // 统计物品间的共现次数
        for (auto film1 : scoredFilms) {
            for (auto film2 : scoredFilms) {
                if (film1 != film2) {
                    similarityMatrix[film1][film2] += 1;
                }
            }
        }
    }

    // 计算物品间的相似度
    for (size_t i = 0; i < nItems; i++) {
        for (size_t j = 0; j < nItems; j++) {
            if (similarityMatrix[i][j] != 0) {
                similarityMatrix[i][j] /= std::sqrt(double(popularItems[i] *
popularItems[j]));
            }
        }
    }

    // 返回相似度矩阵
    return similarityMatrix;
}
```

2. `getKSimilarMatrix` 函数

```
mat_t<std::pair<double, size_t>> getKSimilarMatrix [[nodiscard]] (const mat_t<double>&
similarMatrix) const {
    // 获取物品数量
    const size_t nFilms = similarMatrix.size();
    // 初始化结果矩阵
    mat_t<std::pair<double, size_t>> result;
    result.reserve(nFilms);

    // 遍历每个物品的相似度向量
    for (const auto& simFilms : similarMatrix) {
        // 用于存储当前物品与其他物品的相似度
        std::vector<std::pair<double, size_t>> kSimFilms;
        kSimFilms.reserve(nFilms);

        // 将相似度和物品索引作为对存入 kSimFilms
        for (size_t sfilm = 0; sfilm < nFilms; sfilm++) {
            kSimFilms.emplace_back(simFilms[sfilm], sfilm);
        }

        // 对 kSimFilms 进行降序排序，提取前 mNSimilar 个相似物品
        std::sort(kSimFilms.begin(), kSimFilms.end(), std::greater<>());
        kSimFilms.resize(mNSimilar);

        // 将排序后的 kSimFilms 存入结果矩阵
        result.emplace_back(std::move(kSimFilms));
    }

    // 返回结果矩阵
    return result;
}
```

3. `recommend` 函数

```
std::vector<std::pair<double, size_t>> recommend [[nodiscard]] (
    const mat_t<std::pair<double, size_t>>& kSimilarMatrix, const vec_t<int>& userScore)
const {
    // 获取物品数量
    const size_t nFilms = userScore.size();
    // 记录用户已评分的物品
    std::vector<size_t> watchedFilms;
    for (size_t film = 0; film < nFilms; film++) {
        if (userScore[film] != 0) {
            watchedFilms.emplace_back(film);
        }
    }

    // 初始化推荐列表
    std::vector<std::pair<double, size_t>> relatedFilms;
    relatedFilms.reserve(nFilms);
    for (size_t film = 0; film < nFilms; film++) {
        relatedFilms.emplace_back(0, film);
    }

    // 累加相似物品的评分
    for (auto film : watchedFilms) {
        auto rating = userScore[film];
        const auto& kSimFilms = kSimilarMatrix[film];
        for (const auto& [w, sfilm] : kSimFilms) {
```

```
            // 跳过用户已评分的物品
            if (std::binary_search(watchedFilms.cbegin(), watchedFilms.cend(), sfilm)) {
                continue;
            }
            // 加权累加评分
            relatedFilms[sfilm].first += (w * rating);
        }
    }

    // 对推荐列表进行降序排序，提取前 mNRecommend 个推荐物品
    std::sort(relatedFilms.begin(), relatedFilms.end(), std::greater<>());
    relatedFilms.resize(mNRecommend);

    // 返回推荐结果
    return relatedFilms;
}
```