基于协同过滤和矩阵分解的IMDb电影评分预测

Huibin Mo

1 数据预处理

为保证模型和结果的可靠性，本次实验使用**全量的训练集和测试集数据**，以**数据解读-数据读取-转换为矩阵-统一矩阵尺寸**的步骤进行数据预处理，具体如下：

（1）数据解读

本次实验的数据集共 4 个。①users.txt：记录本次实验的所有用户，共 10000 行，每行一个整数，表示用户 id。②movie_titles.txt：记录本次实验的所有电影，共 17700 部，储存格式为电影 id、年份和电影名称。③netflix_train.txt：训练集，包含 689 万条用户打分，每行为一条打分，存储格式为用户 id、电影 id、分数和打分日期。④netflix_test.txt：测试集，格式与 netflix_train.txt 相同，包含 172 万条用户打分。

为了使训练集与测试集的数据有更好的数据对应，需根据训练集和测试集涉及的用户 id 和电影 id，同时考虑到矩阵中存在空单元格，因此**构建 10000×10000 的矩阵，其中 10000 行对应用户 id 的总数，10000 列涉及电影 id 的总数**。

（2）数据读取

使用 pd.read_csv 函数读取训练集和测试集的数据，并将其转换为 DataFrame 对象。**由于原始数据文件使用空格作为分隔符，因此在 read_csv 函数中设置了 delimiter="\s+"来指定分隔符为一个或多个空格**。同时命名第 1~4 列的列名分别为 user_id、movie_id、score 和 date，方便下一步用 pivot 函数将其转换为矩阵。最终得到两个 DataFrame 对象，分别为训练集 netflix_train 和测试集 netflix_test。

```
1. train_file_path = "../data/netflix_train.txt"
2. test_file_path = "../data/netflix_test.txt"
3. netflix_train = pd.read_csv(train_file_path, header=None, delimiter="\s+",
names=["user_id", "movie_id", "score", "date"])
4. netflix_test = pd.read_csv(test_file_path, header=None, delimiter="\s+",
names=["user_id", "movie_id", "score", "date"])
```

（3）转换为矩阵

使用 **pivot 函数将 netflix_train 和 netflix_test 转换为矩阵形式**，将"user_id"列作为行索引，"movie_id"列作为列索引，"score"列作为矩阵中的值。最终得到训练集矩阵 train 和测试集矩阵 netflix_test。

```
1. matrix_train = netflix_train.pivot(index="user_id", columns="movie_id",
values="score")
2. matrix_test = netflix_test.pivot(index="user_id", columns="movie_id", values="score")
```

（4）统一训练集矩阵和测试集矩阵的尺寸

使用 reindex 函数统一训练集矩阵和测试集矩阵的尺寸。训练集涉及 10000 部电影，测试集涉及 9983 部电影，因此需将测试集矩阵扩大到 10000 列。通过获取 matrix_train 的列名，将 matrix_test 按照相同的列名顺序进行重新索引，确保训练集矩阵和测试集矩阵具有相同的列顺序，方便后续的数据处理和对比分析。最后，使用 fillna 函数将矩阵中的缺失值（NaN）替换为 0。

```
1. columns = matrix_train.columns
2. matrix_test = matrix_test.reindex(columns=columns)
3. matrix_train.fillna(0, inplace=True)
4. matrix_test.fillna(0, inplace=True)
```

2 基于矩阵运算的协同过滤

2.1 代码思路

为减少协同过滤算法的运算时间，直接对训练集矩阵和测试集矩阵进行矩阵运算，以计算相似度-预测评分-计算 RMSE 的思路设计协同过滤算法代码。

（1）创建计算余弦相似度的 similarity 函数

创建 UserCF 类，创建用于计算矩阵相似度的 similarity 函数。为更好进行基于用户的协同过滤算法，创建 UserCF 类。首先，设定**计算两个向量余弦相似度的 similarity 函数**，其思路为：

① np.dot(x_i, x_k.T)使用 np.dot 函数计算向量 x_i 和向量 x_k 的点积。实际上本次实验的向量 x_i 和向量 x_k 均为训练集矩阵。

② np.outer(np.linalg.norm(x_i, axis=1), np.linalg.norm(x_k, axis=1)) 使用 np.linalg.norm 函数计算向量 x_i 和向量 x_k 的范数，接着使用 np.outer 函数计算了两个范数的外积，得到外积矩阵。

③ np.dot(x_i, x_k.T) / np.outer(np.linalg.norm(x_i, axis=1), np.linalg.norm(x_k, axis=1))，即将步骤①计算得到的点积，除以步骤②中得到的外积矩阵，**得到两个向量的余弦相似度**，相似度的计算结果在 0 到 1 之间。

```
1. def similarity(self, x_i, x_k):
2.     return np.dot(x_i, x_k.T) / np.outer(np.linalg.norm(x_i, axis=1),
np.linalg.norm(x_k, axis=1))
```

（2）创建预测评分的 UserCF_predict 函数

进一步在 UserCF 类中，创建**基于用户的协同过滤算法预测用户评分的 UserCF_predict 函数，得到每个用户对未曾评分电影的预测评分 score_predict**。具体思路如下：

① 首先调用 self.similarity(train_matrix, train_matrix)，计算训练集中每个用户之间的相似度，**得到相似度矩阵 matrix_similarity**。

② 通过矩阵运算 np.dot(matrix_similarity, train_matrix)，将相似度矩阵 matrix_similarity 与训练集矩阵 train_matrix 相乘，得到一个新的矩阵 score_table，即**对应用户对所有电影的加权评分**。

③ 通过计算每个用户与其他用户的相似度之和，得到一个相似度之和的矩阵 similarity_total，表示**对应用户与其他用户的相似度之和**。

④ 通过 np.divide(score_table, similarity_total)，将 score_table 矩阵中的每个

元素除以相似度之和矩阵 similarity_total 中的对应元素，得到**每个用户对未曾评分电影的预测评分 score_predict**。

```
def UserCF_predict(self, train_matrix):
    # 基于矩阵运算求训练集的相似度
    matrix_similarity = self.similarity(train_matrix, train_matrix)
    # 计算加权求和后的打分值
    score_table = np.dot(matrix_similarity, train_matrix)
    similarity_total = np.sum(np.abs(matrix_similarity), axis=1, keepdims=True)
    score_predict = np.divide(score_table, similarity_total)
    return score_predict
```

（3）预测电影评分

数据获取。调用 1 数据预处理的数据预处理代码（自建的 data_process 函数），得到训练集矩阵 train_matrix 和测试集矩阵 test_matrix。

```
# 调用 data_process 函数，处理得到 train_matrix 和 test_matrix
train_file_path = "../data/netflix_train.txt"
test_file_path = "../data/netflix_test.txt"
train_matrix, test_matrix = data_process(train_file_path, test_file_path)
```

预测电影评分。先定义 UserCF，通过调用自建的 UserCF_predict 函数，以（1）和（2）的思路，预测得到电影评分 score_predict_array。

```
# 基于训练集训练协同过滤算法，并预测用户评分
user_cf = UserCF()
score_predict_array = user_cf.UserCF_predict(train_matrix)
```

（4）计算 RMSE

剔除训练集和预测结果中的零值。将测试集矩阵转换为数组，以提高运算效率。然后，使用 np.nonzero 函数找到测试集中非零数值的位置，得到非零数值的索引。通过这些索引，从测试集矩阵和预测评分矩阵中提取出非零数值，分别存储在 score_actual 和 score_predict 中。

```
# 剔除训练集和预测结果的零值
test_array = test_matrix.values  # 将训练集转化为 array，加快运算效率
nonzero_indices = np.nonzero(test_array)  # 训练集的非零数值的位置
score_actual = test_array[nonzero_indices]  # 获取训练集的非零数值
score_predict = score_predict_array[nonzero_indices]  # 获取预测结果的非零数值
```

计算 RMSE。使用 sklearn.metrics 的 mean_squared_error 函数计算预测评分与实际评分之间的 RMSE。

```
rmse = mean_squared_error(score_actual, score_predict, squared=False)
```

2.2 预测结果

基于矩阵运算的协同过滤算法主要结果如表 1 所示。

①**RMSE**：在**剔除测试集零值**的基础上，测试集实际评分与预测评分的 **RMSE 为 2.5674**。

②算法的时间消耗：由于使用矩阵运算，时间消耗大幅度减少，**训练和预测仅用 39.49s**。其中，对有效数据的训练集矩阵进行算法训练，消耗时长不到 1s。

表 1　基于矩阵运算的矩阵分解算法结果

方法	RMSE	总时长
全矩阵矩阵运算	2.5674	39.49 s

3　基于稀疏矩阵的协同过滤

3.1　数据预处理和生成稀疏矩阵

为保证模型和结果的可靠性，本次实验使用**全量的训练集和测试集数据**，以**数据解读-数据读取-构建索引-训练集和训练集解析-生成稀疏矩阵**的思路步骤进行数据预处理，具体说明如下：

（1）数据解读

为了使训练集与测试集的数据有更好的数据对应，需根据用户 id 和电影 id，同时考虑到矩阵中存在空单元格，因此**构建 10000×17700 的稀疏矩阵，其中 10000 行对应用户 id 的总数，17700 列对应电影 id 的总数**。

（2）数据读取

使用 Python 内置的 **open 函数**读取 4 个数据集，其中 **with 语句**可以保证文件读取完成后自动关闭文件句柄，且不改变数据集的数据。使用 **read 方法**读取文件内容后，再使用 **splitlines 方法**对每行内容进行分割，得到一个以每行内容为元素的列表。

```
# 从 users.txt 文件，读取用户 id 列表
with open('../data/users.txt', 'r') as file:
    users = file.read().splitlines()  # 按 txt 顺序排列用户 id
# 从 movie_titles.txt 文件，读取电影信息列表
with open('../data/movie_titles.txt', 'r', encoding='ISO-8859-1') as file:
    movie_titles = file.read().splitlines()
# 读取 netflix_train.txt 文件
with open('../data/netflix_train.txt', 'r') as file:
    netflix_train_data = file.read().splitlines()
# 读取 netflix_test.txt 文件
with open('../data/netflix_test.txt', 'r') as file:
    netflix_test_data = file.read().splitlines()
```

（3）构建数据索引

对于读取了用户 id 列表，并使用字典 user_dict 将用户 id 映射到索引上，方便后续构建稀疏矩阵。

另外，movie_titles.txt 的电影数据的读取与处理比较特殊。由于电影名称中包含了逗号，因此使用 split 方法进行分割可能会将电影名称分成多个部分。为了避免这种情况，代码中使用 **split(',', 1)方法指定最多只分割成两个部分**，将电影 id 和电影名称分割。创建了一个字典 movie_dict 来存储电影 id 和电影索引的

映射关系。

```
# 创建字典来存储用户 id 和索引的映射关系
user_dict = {user_id: index for index, user_id in enumerate(users)}
# 创建字典来存储电影 id 和电影名的映射关系
movie_dict = {}
for line in movie_titles:
    movie_id, movie_name = line.split(',', 1)  # 将 line 按逗号分隔，最多分隔成 2 个字符串
    movie_dict[movie_id] = movie_name  # 将电影 id 和电影名存入字典中
```

（4）训练集和测试集数据解析

训练集和测试集的预处理过程相同，都是解析原始数据并将用户、电影和评分数据存入相应的列表中。需要注意的是，在将电影 id 转换为电影索引时，需要减去 1 以匹配稀疏矩阵的索引。最终得到的 **rows、cols 和 data 三个列表分别对应 CSR 格式稀疏矩阵中的行、列和评分数据**。

```
# 创建训练集稀疏矩阵的行、列和数据列表
rows_train = []
cols_train = []
data_train = []
# 解析 netflix_train_data 并将数据存入相应的列表中
for line in netflix_train_data:
    user_id, movie_id, score, date = line.split(' ')  # 将 line 按空格分隔
    user_index = user_dict[user_id]  # 获取用户的索引
    movie_index = int(movie_id) - 1  # 获取电影的索引，减去 1 以匹配矩阵索引
    rows_train.append(user_index)
    cols_train.append(movie_index)
    data_train.append(float(score))

# 创建测试集稀疏矩阵的行、列和数据列表
rows_test = []
cols_test = []
data_test = []
# 解析 netflix_test_data 并将数据存入相应的列表中
for line in netflix_test_data:
    user_id, movie_id, score, date = line.split(' ')
    user_index = user_dict[user_id]
    movie_index = int(movie_id) - 1  # 减去 1 以匹配矩阵索引
    rows_test.append(user_index)
    cols_test.append(movie_index)
    data_test.append(float(score))
```

（5）生成稀疏矩阵

创建训练集和测试集的 CSR 格式的稀疏矩阵。使用 Python 内置 scipy.sparse 模块中的 **csr_matrix 函数构建 CSR 格式的稀疏矩阵**。该函数传入 3 个参数：数据列表、行列表和列列表。其中，行列表和列列表分别对应原始数据中的用户和

电影索引，数据列表则是对应的评分数据。同时，指定稀疏矩阵的形状，即用户数和电影数。**得到的训练集和测试集稀疏矩阵，均为 10000 行×17700 列的稀疏矩阵**，其中训练集有效数据为 6897746 条，测试集有效数据为 1719466 条。

保存为 NPZ 格式的稀疏矩阵文件。使用 Python 内置 scipy.sparse 模块中的 **save_npz 函数将稀疏矩阵保存为 CSR 格式文件**，保存到本地，方便后续调用。由于稀疏矩阵比较大，因此使用压缩后的 NPZ 格式进行保存。后面使用 scipy.sparse.load_npz 函数读取这些稀疏矩阵文件，得到稀疏矩阵对象。

```
1. # 构建 CSR 格式的稀疏矩阵
2. sparse_matrix_train = csr_matrix((data_train, (rows_train, cols_train)),
shape=(len(users), len(movie_titles)))
3. sparse_matrix_test = csr_matrix((data_test, (rows_test, cols_test)),
shape=(len(users), len(movie_titles)))
4. # 将稀疏矩阵保存为 NPZ 格式文件
5. save_npz("../data/sparse_matrix_train.npz", sparse_matrix_train)
6. save_npz("../data/sparse_matrix_test.npz", sparse_matrix_test)
```

3.2 协同过滤代码思路

为进一步检验协同过滤算法的可靠性，基于训练集稀疏矩阵和测试集稀疏矩阵，使用遍历的方法进行算法训练和评分预测，以**计算相似度-预测评分-计算 RMSE** 的思路设计协同过滤算法代码。

（1）创建计算相似度矩阵的 compute_similarity_matrix 函数

首先**创建 UserCF 类，封装了基于用户的协同过滤算法**。在类的初始化函数__init__中，导入训练集稀疏矩阵 train_data，用于训练协同过滤算法。同时初始化一些变量，如相似度矩阵 similarity_matrix 和用于记录运算时长的变量。

创建用于计算相似度矩阵的 compute_similarity_matrix 函数。

① 调用 toarray()方法，将稀疏矩阵**转换为数组形式**，以便进行向量化计算。

② **每个用户的评分进行去中心化处理**，即通过计算每个用户的评分平均值，并将平均值与原始评分相减，得到去中心化的评分矩阵。

③ 通过矩阵运算**计算用户相似度矩阵**。通过 np.dot(centered_train_array, centered_train_array.T)，计算去中心化的评分矩阵与其转置的矩阵乘积。然后，通过 np.outer(np.linalg.norm(centered_train_array, axis=1), np.linalg.norm(centered_train_array, axis=1))计算去中心化的评分矩阵每行的范数的外积。最后，通过除法运算将矩阵相除，得到用户的**余弦相似度矩阵 similarity_matrix**。

④ 处理 NaN 值和负数值。计算相似度矩阵的时间复杂度较高，因为需要进行矩阵乘法和归一化操作。

```
1. class UserCF:
2.     """基于用户的协同过滤算法"""
3.
4.     def __init__(self, train_data):
5.         """初始化函数"""
6.         self.train_data = train_data
7.         self.similarity_matrix = None
```

```
        self.time_total = 0  # 用于记录运算总时长，初始化为 0
        self.time_train = 0
        self.time_predict_total = 0

    def compute_similarity_matrix(self):
        """计算相似度矩阵"""
        start_time_train = time.time()
        # 将稀疏矩阵转换为数组形式，便于进行向量化计算
        train_array = self.train_data.toarray()
        # 计算每个用户的评分平均值，并将平均值与原始评分相减，得到去中心化的评分矩阵
        centered_train_array = train_array - np.mean(train_array, axis=1).reshape((-1, 1))
        # 计算用户相似度矩阵
        similarity_matrix = np.dot(centered_train_array, centered_train_array.T) / np.outer(np.linalg.norm(centered_train_array, axis=1), np.linalg.norm(centered_train_array, axis=1))
        similarity_matrix[np.isnan(similarity_matrix)] = 0  # 处理 nan 值
        similarity_matrix[similarity_matrix < 0] = 0  # 处理负数值
        self.similarity_matrix = similarity_matrix
        end_time_train = time.time()
        self.time_train = end_time_train - start_time_train  # 计算消耗的时间
        print("Compute similarity matrix time: {:.4f}s".format(self.time_train))  # 输出消耗的时间
        self.time_total += self.time_train
```

（2）创建预测评分的 predict_score 函数

①给定用户 ID 和电影 ID，**获取观看过该电影的用户列表**。

②对于每个观看过该电影的用户，计算与目标用户的相似度，并根据相似度和观看用户的评分计算**预测评分**。

③对预测评分进行一些处理，如处理 NaN 值。预测评分的时间复杂度取决于观看过该电影的用户数量。

```
def predict_score(self, user_id, movie_id):
        """预测评分"""
        start_time_predict = time.time()
        predict_score = 0

        # 获取观看过该电影的用户的列表
        users_with_movie = self.train_data[:, movie_id].nonzero()[0]

        if len(users_with_movie) != 0:
            # 计算用户与其他用户的相似度，同时计算预测评分
            similarity_sum = 0
            for other_user_id in users_with_movie:
                similarity = self.similarity_matrix[user_id, other_user_id]
```

```
                predict_score += similarity * (self.train_data[other_user_id,
movie_id] - np.mean(self.train_data[other_user_id, :]))
                similarity_sum += similarity

            if similarity_sum != 0:
                predict_score /= similarity_sum

        if np.isnan(predict_score):
            predict_score = 0
        else:
            end_time_predict = time.time()
            duration_predict = end_time_predict - start_time_predict
            self.time_predict_total += duration_predict
            self.time_total += duration_predict

        return predict_score
```

（3）预测评分

数据获取。读取训练集稀疏矩阵和测试集稀疏矩阵。

```
sparse_matrix_train = load_npz("../data/sparse_matrix_train.npz")
sparse_matrix_test = load_npz("../data/sparse_matrix_test.npz")
```

预测评分。创建了一个 UserCF 的实例，并调用 compute_similarity_matrix 函数计算相似度矩阵。使用协同过滤算法对测试集中的每个用户-电影对进行评分预测。通过遍历测试集的非零元素，获取用户 ID 和电影 ID，调用 predict_score 函数预测评分，并记录预测评分和实际评分。

```
# 训练算法
user_cf = UserCF(sparse_matrix_train)  # 基于训练集，训练协同过滤算法
user_cf.compute_similarity_matrix()

# 基于协同过滤算法，预测电影评分
predictions = []
actual_score = []
for user_id, movie_id in zip(*sparse_matrix_test.nonzero()):
    actual_score = sparse_matrix_test[user_id, movie_id]
    predicted_score = user_cf.predict_score(user_id, movie_id)
    predictions.append(predicted_score)  # 记录协同过滤算法预测的电影评分
    actual_score.append(actual_score)  # 记录测试集中的实际电影评分
```

（4）计算 RMSE

计算 RMSE。使用 sklearn.metrics 的 mean_squared_error 函数计算预测评分与实际评分之间的 RMSE。

```
rmse = mean_squared_error(predictions, actual_score, squared=False)
```

3.3 预测结果

基于稀疏矩阵的协同过滤算法主要结果如表 1 所示。

①RMSE：测试集实际评分与预测评分的 **RMSE 仅为 1.0264**，表明算法预测效果优秀。由于基于稀疏矩阵的协同过滤算法使用了 for 循环遍历每个数值，导致训练得到的协同过滤算法更为精确可靠，因此 RMSE 要低于上文基于矩阵运算的协同过滤。

②算法的时间消耗：由于使用矩阵运算，时间消耗大幅度减少，**训练和预测仅用 41.32s**。其中，对有效数据的训练集矩阵进行算法训练，消耗时长不到 1s。

表 2　基于稀疏矩阵的矩阵分解算法结果

方法	实验	RMSE	总时长	训练时长	测试时长
稀疏矩阵	第 1 次实验（更强 CPU）	1.0264	31.06 h	2.97 h	28.08 h
	第 2 次实验	1.0264	39.55 h	3.85 h	35.70 h

注：两次实验使用相同的代码与数据集。由于实验 CPU 有差异，导致了程序运行时长的差异。第 1 次实验 CPU 为 Intel Xeon Gold 6330，第 2 次实验 CPU 为 Intel Xeon Gold 6326，前者的线程数、缓存等更优。**本次实验报告主要分析第 1 次实验的结果**。

4　基于梯度下降的矩阵分解

4.1 代码思路

基于训练集稀疏矩阵和测试集稀疏矩阵，使用矩阵分解方法进行算法训练和评分预测，以**不同参数组合的矩阵分解-计算 RMSE-绘图**的思路设计矩阵分解算法代码。

（1）创建矩阵分解的 matrix_decomposition 函数

matrix_decomposition 函数用于训练矩阵分解算法和预测评分。输入参数有训练集稀疏矩阵、隐空间维度 k、学习率 alpha、正则化参数 lambd、最大迭代次数 max_iter 和阈值 threshold。函数返回分解后的左矩阵 U、右矩阵 V，以及每次迭代的目标函数值 error_values 和 RMSE 值 rmse_values。具体思路如下：

① **初始化矩阵**。根据训练集的形状，初始化左矩阵 U 和右矩阵 V。其中，U 的形状为(num_users, k)，V 的形状为(num_movies, k)，k 为隐空间维度，num_users 为用户数量，num_users 为电影数量。

② **获取训练集中的非零元素**。使用 nonzero 方法获取训练集中非零元素的行和列索引，然后使用 A1 属性获取非零元素的值，用于计算预测评分和目标函数值。

③ **迭代更新矩阵，记录迭代过程中的目标函数值和 RMSE**。循环迭代更新矩阵 U 和 V，直到达到最大迭代次数或目标函数值小于阈值。在每次迭代中，首先**计算预测评分**，即计算 U 和 V 的乘积，并计算预测评分与实际评分的差异。然后，根据差异计算梯度，并使用学习率 alpha 和正则化参数 lambd 更新矩阵 U 和 V。接下来，计算目标函数值，包括差异的平方和正则化项。最后，计算测试集上的 RMSE，并将目标函数值和 RMSE 值分别添加到 error_values 和 rmse_values 中。

④ **返回**分解后的**左矩阵 U、右矩阵 V**，以及**每次迭代的目标函数值**

error_values 和 RMSE 值 rmse_values。

```
def matrix_decomposition(sparse_matrix_train, k, alpha, lambd, max_iter, threshold):
    # 初始化 U 和 V 矩阵
    num_users, num_movies = sparse_matrix_train.shape
    U = np.random.rand(num_users, k)
    V = np.random.rand(num_movies, k)

    # 获取稀疏矩阵中非零元素的行、列和值
    rows, cols = sparse_matrix_train.nonzero()
    values = sparse_matrix_train[rows, cols].A1

    error_values = []  # 用于存储每次迭代后的目标函数值
    rmse_values = []  # 用于存储每次迭代后测试集上的 RMSE 值
    for i in range(max_iter):
        # 计算预测评分
        dot_product = np.sum(U[rows] * V[cols], axis=1)
        diff = dot_product - values
        # 计算梯度
        grad_U = diff[:, None] * V[cols] + 2 * lambd * U[rows]
        grad_V = diff[:, None] * U[rows] + 2 * lambd * V[cols]
        # 更新 U 和 V
        U[rows] -= alpha * grad_U
        V[cols] -= alpha * grad_V
        # 计算目标函数值
        error = np.sum(diff ** 2) / 2 + lambd * (np.sum(U[rows] ** 2) +
np.sum(V[cols] ** 2))
        error_values.append(error)  # 记录目标函数值

        # 计算测试集的 RMSE
        predict_score = U.dot(V.T)
        rmse = calculate_rmse(predict_score, sparse_matrix_test)
        rmse_values.append(rmse)

        if error < threshold:
            break

    return U, V, error_values, rmse_values
```

（2）创建计算 RMSE 的 calculate_rmse 函数

calculate_rmse 函数用于计算 RMSE（均方根误差），输入参数为预测评分 predict_score 和真实评分 actual_score。获取预测评分 predict_score 和真实评分 actual_score 的非零值，并进一步计算 RMSE。

```
def calculate_rmse(predict_score, actual_score):
    mask = actual_score.nonzero()  # 获取非零值的位置
```

```
    predict_score = predict_score[mask]  # 获取预测评分的非零值
    actual_score = actual_score[mask].A1  # 获取测试集实际评分的非零值
    return np.sqrt(np.mean((predict_score - actual_score) ** 2))  # 计算 RMSE
```

（3）数据获取与参数设定

使用 load_npz 函数读取训练集稀疏矩阵和测试集稀疏矩阵。按题目要求设定参数，其中**隐空间维度 k 的取值为 50 和 20**；**正则化参数 lambd 的取值为 0.01、0.001 和 0.1**；学习率 alpha 为 0.01。

```
# 读取训练集和测试集
sparse_matrix_train = load_npz("../data/sparse_matrix_train.npz")
sparse_matrix_test = load_npz("../data/sparse_matrix_test.npz")
# 设定参数
k_values = [50, 20]  # 隐空间维度
lambda_values = [0.01, 0.001, 0.1]  # 正则化参数
alpha_range = [0.01]  # 学习率
max_iter = 1000  # 最大迭代次数
threshold = 1e-4  # 域值
```

（4）基于不同的参数组合，训练模型和预测评分

使用嵌套的循环遍历不同的参数组合。对于每个参数组合，**调用 matrix_decomposition 函数进行矩阵分解算法的训练**，得到分解后的左矩阵 U、右矩阵 V，以及每次迭代的目标函数值 error_values 和 RMSE 值 rmse_values。然后，使用分解后的左矩阵 U 和右矩阵 V 计算预测评分。接着，**调用 calculate_rmse 函数计算 RMSE 值**，并将结果保存到 results 中。

```
results = []  # 记录预测评分
models = []  # 记录模型
for k in k_values:
    for lambd in lambda_values:
        for alpha in alpha_range:
            U, V, error_values, rmse_values = matrix_decomposition(sparse_matrix_train, k, alpha, lambd, max_iter, threshold)  # 训练矩阵分解算法

            predict_score = U.dot(V.T)  # 计算预测评分
            # 计算 RMSE
            rmse = calculate_rmse(predict_score, sparse_matrix_test)
            rmse_values.append(rmse)
            # 保存结果和模型
            results.append((k, lambd, alpha, train_time, predict_time, rmse, error_values, rmse_values))
            models.append((k, lambd, alpha, U, V))
```

（5）绘制迭代过程中的目标函数值和 RMSE

使用 matplotlib.pyplot 库绘制目标函数值和 RMSE 值随迭代次数变化的图表。由于方法相似，以下只展示迭代过程中的目标函数值的绘图代码。

```
# 绘制迭代过程中的目标函数值变化图
plt.figure()
for res in results:
    error_values = res[6]
    plt.plot(range(len(error_values)), error_values, label=f"k={res[0]}, Lambda={res[1]}, Alpha={res[2]}")
plt.xlabel('Iteration')
plt.ylabel('Objective Function Value')
plt.title('Objective Function Value over Iterations')
plt.legend(loc='upper left', bbox_to_anchor=(-0.1, -0.2), ncol=2)
plt.tight_layout()
plt.savefig('../result/md/figure/objective_function_md_{}.png'.format(end_time_str), dpi=600, bbox_inches='tight')
```

4.2 预测结果

4.2.1 k=50，λ=0.01

基于 4.1 的矩阵分解代码，得到了不同参数组合下的矩阵分解算法预测结果（表 3），并对迭代过程中的目标函数值（图 1）和 RMSE（图 2）进行绘图。

当隐空间维度 k 为 50、正则化参数 λ 为 0.01（参数组合 1）时，矩阵分解算法的最终 RMSE 为 1.6475，是 6 种参数组合中第 3 小，优于其他 2 种 k=50 的参数组合。

时间消耗为 6.03 h，其中训练时长几乎占据所有的时长。时间消耗与隐空间维度有关，k=50 的参数组合的时间消耗明显大于 k=20 的参数组合，相同隐空间维度的时间消耗相近。

从迭代过程中的**目标函数值**（图 1）来看，**参数组合 1 的目标函数值是 6 种参数组合中第 3 小的**，大于参数组合 4（k=20，λ=0.01）和参数组合 5（k=20，λ=0.01）。但收敛明显慢于这 2 种参数组合，迭代次数为 1000 时，目标函数值仍保持下降趋势。

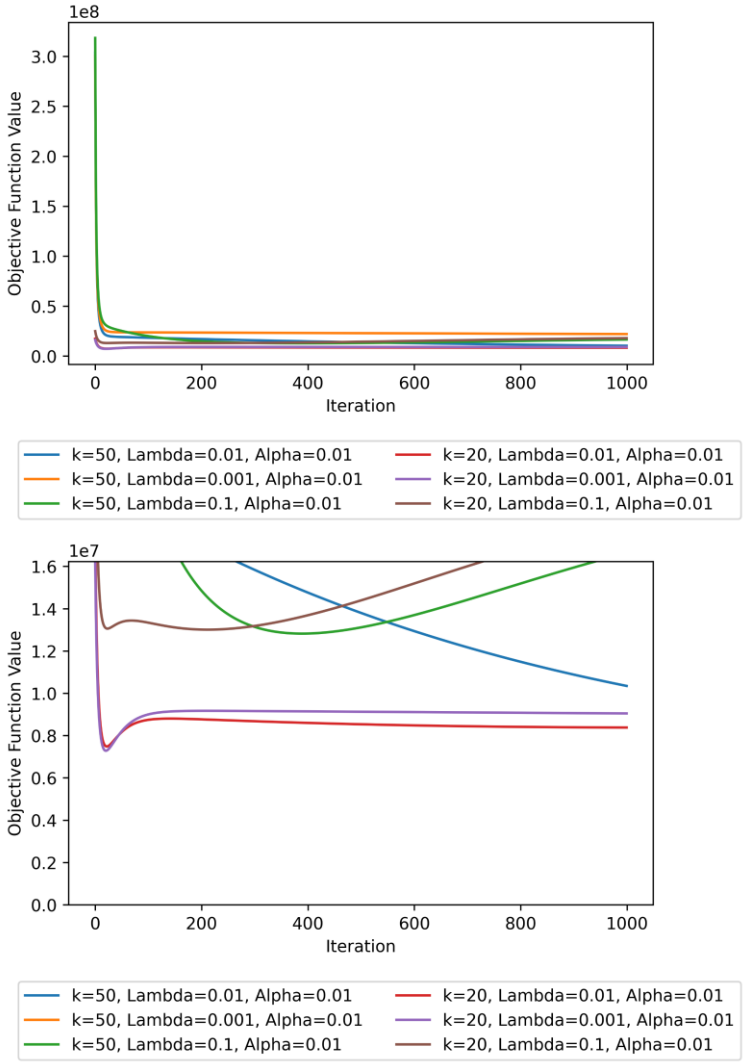
从迭代过程中的 **RMSE**（图 2）来看，**参数组合 1 的 RMSE 是 6 种参数组合中第 3 小的**。迭代次数为 1000 时，RMSE 仍保持下降趋势。

表 3　不同参数组合的矩阵分解算法结果

实验	组合	参数设置		RMSE	总时长	训练时长	测试时长
第 1 次实验	1	k=50	λ=0.01	1.6475	6.03 h	6.03 h	0.48 s
（更强 CPU）	2	k=50	λ=0.001	2.5241	6.03 h	6.03 h	0.56 s
	3	k=50	λ=0.1	1.7923	6.07 h	6.07 h	0.69 s
	4	k=20	λ=0.01	1.4995	2.70 h	2.70 h	0.56 s
	5	k=20	λ=0.001	1.6151	2.71 h	2.71 h	0.65 s
	6	k=20	λ=0.1	1.8983	2.70 h	2.70 h	0.63 s
第 2 次实验	1	k=50	λ=0.01	1.7269	6.14 h	6.14 h	0.48 s
	2	k=50	λ=0.001	2.4114	6.16 h	6.16 h	0.61 s

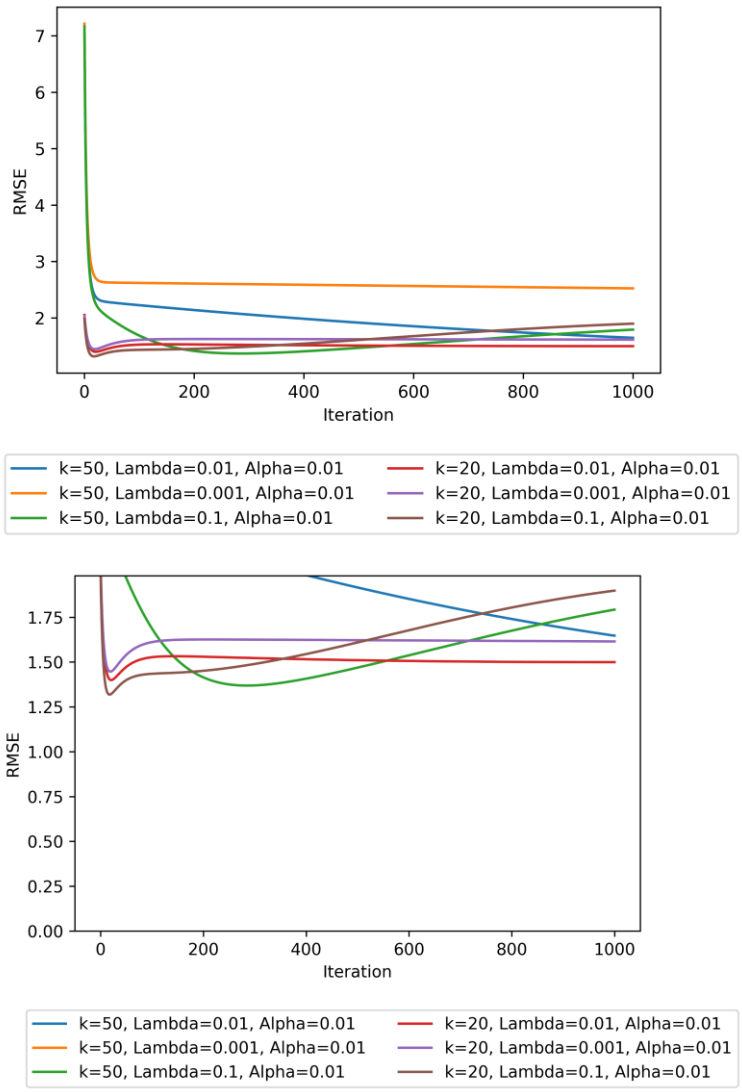
	3	k=50	λ =0.1	1.7785	6.23 h	6.23 h	0.64 s
	4	k=20	λ =0.01	1.5470	2.80 h	2.79 h	0.62 s
	5	k=20	λ =0.001	1.5599	2.81 h	2.81 h	0.62 s
	6	k=20	λ =0.1	1.8938	2.83 h	2.83 h	0.56 s

注：两次实验使用相同的代码与数据集。由于实验 CPU 有差异，导致了程序运行时长的差异。第 1 次实验 CPU 为 Intel Xeon Gold 6330，第 2 次实验 CPU 为 Intel Xeon Gold 6326，前者的线程数、缓存等更优。**本次实验报告主要分析第 1 次实验的结果**。



注：下图为缩小 y 轴范围的图

图 1　基于梯度下降的矩阵分解算法的迭代过程中的目标函数值



注：下图为缩小 y 轴范围的图

图 2　基于梯度下降的矩阵分解算法的迭代过程中的 RMSE

4.2.2　参数组合对比与择优

通过表 3 可以看出，**参数组合 4（k=20，λ=0.01）的效果最优**。进一步以参数组合 4 为基准，对比不同参数组合的 RMSE 相对值和总时长相对值（表 4）。

从最终 RMSE 来看，**参数组合 4 的最终 RMSE 最小**，为 1.4995。其他参数组合的 RMSE 是参数组合 4 的 107.71%~168.33%。

从时间消耗上来看，**参数组合 4 的总时长并列最短**，为 2.70 h。由于隐空间维度为 20，所以模型训练和预测时长明显短于参数组合 1~3。

从迭代过程中的**目标函数值**（图 1）来看，**参数组合 4 的目标函数值是 6 种参数组合中最小的**。迭代次数约为 200 时，目标函数值趋近收敛。

从迭代过程中的 **RMSE**（图 2）来看，**参数组合 1 的 RMSE 是 6 种参数组合中最小的**。迭代次数约为 400 时，RMSE 趋近收敛。

表 4　不同参数组合的矩阵分解算法结果对比

组合	参数设置		RMSE	RMSE 相对值	总时长	总时长相对值
1	k=50	λ=0.01	1.6475	109.87%	6.03 h	223.33%
2	k=50	λ=0.001	2.5241	168.33%	6.03 h	223.33%
3	k=50	λ=0.1	1.7923	119.53%	6.07 h	224.81%
4	k=20	λ=0.01	1.4995	100.00%	2.70 h	100.00%
5	k=20	λ=0.001	1.6151	107.71%	2.71 h	100.37%
6	k=20	λ=0.1	1.8983	126.60%	2.70 h	100.00%

注：相对值以 k=20、λ=0.01 的实验为基准。采用第 1 次实验结果。

5　方法对比

对协同过滤算法和矩阵分解算法的预测精度、时间消耗等进行对比总结（表 5），进一步总结出两种算法的优缺点：

表 5　两种算法结果对比

算法	模型设置		RMSE	相对值	总时长	相对值	训练时长	测试
基于用户的	矩阵运算		2.5674		39.49 s		0.05 s	39.44 s
协同过滤	稀疏矩阵		1.0264	68.45%	31.06 h	1150.37%	2.97 h	28.08 h
基于梯度下降的	k=50	λ=0.01	1.6475	109.87%	6.03 h	223.33%	6.03 h	0.48 s
矩阵分解	k=50	λ=0.001	2.5241	168.33%	6.03 h	223.33%	6.03 h	0.56 s
	k=50	λ=0.1	1.7923	119.53%	6.07 h	224.81%	6.07 h	0.69 s
	k=20	λ=0.01	1.4995	100.00%	2.70 h	100.00%	2.70 h	0.56 s
	k=20	λ=0.001	1.6151	107.71%	2.71 h	100.37%	2.71 h	0.65 s
	k=20	λ=0.1	1.8983	126.60%	2.70 h	100.00%	2.70 h	0.63 s

注：相对值以 k=20、λ=0.01 的基于梯度下降的矩阵分解实验为基准。

（1）协同过滤

①优点：

- **预测精度更高：**协同过滤算法利用用户之间的相似性来进行推荐，能够捕捉到用户的偏好和行为模式，因此在预测用户对电影的评分时具有高的准确性。因此**协同过滤算法的 RMSE 为本次实验最优，仅为 1.0264，RMSE 相较于最优的矩阵分解算法下降 31.55%**。
- **算法实现更简单：**协同过滤算法主要基于用户之间的电影评分相似性来进行推荐，不需要对数据进行复杂的预处理，算法相对简单。
- **适用范围更广：**协同过滤算法不需要领域知识，不需要事先了解物品或用户的特征，只需要根据用户的历史行为进行推荐，适用于各种不同领域的推荐系统。同时协同过滤算法可以根据用户的实时行为调整推荐结果，更好地适应用户的兴趣变化。可以动态地反映用户的偏好变化，提供个性化的推荐。

②缺点：

- **时间消耗更长：**当用户和电影的数量非常多时，用户之间的相似性计算和推荐计算会变得非常耗时，导致**协同过滤算法总时长是矩阵分解算法的 11.50 倍**。

- **冷启动问题**：对于新用户或新电影，由于因为没有足够的电影评分数据来计算相似度，协同过滤算法无法准确预测该用户的电影评分。

（2）矩阵分解

①优点：

- **算法运算效率更高**：矩阵分解算法只需要对评分矩阵进行矩阵分解，调整的主要参数主要是分解矩阵的维数和正则化项系数，模型复杂度较低，需要的运算时间相对较短。因此**单个参数组合的矩阵分解算法时间消耗明显少于协同过滤算法**。
- **处理数据稀疏性问题**：矩阵分解算法可以通过对用户-电影评分矩阵的分解，提取出隐含的用户和物品特征，从而解决数据稀疏性问题。
- **个性化推荐**：矩阵分解算法可以根据用户和电影评分的隐含特征进行个性化的预测与推荐，更准确地预测用户对电影的评分。通过学习用户和物品的隐含特征，它可以捕捉到更细粒度的用户偏好和物品特性。因此矩阵分解适用于较大规模的推荐系统

②缺点：

- **预测精度可能较低**：矩阵分解算法可能会丢失一些重要的用户-电影关联信息，导致预测精度较低，**RMSE 低于基于用户的协同过滤算法**。
- **参数调优成本较高**：矩阵分解算法涉及到矩阵运算和优化算法，参数调优相对复杂，需要选择合适的隐空间维度和正则化参数，需要进行多次迭和调整参数，可能需要更多的时间和计算资源，因此**本次实验的 6 个参数组合矩阵分解的运算总时长要多于协同过滤算法**。