《大数据分析》第 2 次实验：矩阵分解

姓名：刘培源　　学号：2023214278

Note

实验中的代码均在“code/exp2.ipynb”中，由于作业提交空间的限制，并未包含数据。要跑通代码，需要将实验数据中的“netflix_train.txt”，“netflix_test.txt”和“users.txt”放到“code”文件夹下。

题目 1：数据预处理。首先要体会在大数据处理过程中不得不经历的一个步骤：数据清洗及格式化。对应到本次作业的问题，就是需要将输入文件整理成维度为“用户 * 电影”的矩阵 $\mathbf{X}$，其中 $\mathbf{X}_{ij}$ 对应用户 i 对电影 j 的打分。对于分数未知的项，可以采取一些特殊的处理方法，如全定为 0 或另建一个矩阵进行记录哪些已知哪些未知。这一步的输出为两个矩阵，$\mathbf{X}_{\text{train}}$ 和 $\mathbf{X}_{\text{test}}$，分别对应训练集与测试集。

答：我选择了全量数据，并将分数未知的项全部设置成 0，数据处理代码如下：

```python
import numpy as np
from tqdm import tqdm

# 读取user_ids
user_ids = np.loadtxt('users.txt', dtype=np.int32)

# 初始化矩阵 X_train 和 X_test 为 0 矩阵
X_train, X_test = [np.zeros((len(user_ids), 10000)) for _ in range(2)]

# 建立用户ID到矩阵索引的映射
user_id_to_index = {user_id: index for index, user_id in enumerate(user_ids)}

# 读取评分数据并更新矩阵 X_train 和 X_test
with open('netflix_train.txt', 'r') as file:
    print("处理训练数据X_train...")
    for line in tqdm(file):
        user_id, movie_id, score, _ = line.split()
        X_train[user_id_to_index[int(user_id)], int(movie_id) - 1] = float(score)

with open('netflix_test.txt', 'r') as file:
    print("处理测试数据X_test...")
    for line in tqdm(file):
        user_id, movie_id, score, _ = line.split()
        X_test[user_id_to_index[int(user_id)], int(movie_id) - 1] = float(score)

print(f"训练数据X_train的形状为：{X_train.shape}")
print(f"测试数据X_test的形状为：{X_test.shape}")
```

题目 2：协同过滤。协同过滤（Collaborative Filtering）是最经典的推荐算法之一，包含基于 user 的协同过滤和基于 item 的协同过滤两种策略。本次作业需要实现基于用户的协同过滤算法。算法的思路非常简单，当需要判断用户 i 是否喜欢电影 j，只要看与 i 相似的用户，看他们是否喜欢电影 j，并根据相似度对他们的打分进行加权平均。用公式表达如下：

$$\text{score}(i,j) = \frac{\sum_k \{\text{sim}[\mathbf{X}(i), \mathbf{X}(k)] \cdot \text{score}(k,j)\}}{\sum_k \text{sim}[\mathbf{X}(i), \mathbf{X}(k)]} \tag{1}$$

其中，$\mathbf{X}(i)$ 表示用户 i 对所有电影的打分，对应到本次作业的问题中，就是 $\mathbf{X}$ 矩阵中第 i 行对应的 10000 维的向量（未知记为 0）。

$\text{sim}[\mathbf{X}(i), \mathbf{X}(k)]$ 表示用户 i 和用户 k 对于电影打分的相似度，可以采用两个向量的 cos 相似度来表示，即：$\cos(x,y) = \frac{x \cdot y}{|x||y|}$。通过这个公式，就可以对测试集中的每一条记录，计算用户可能的打分。

答：我的实现代码如下：

```python
import time

def predict_ratings_efficient(X_train):
    # 对于X_train进行归一化
    X_train_norm = X_train / np.linalg.norm(X_train, axis=1, keepdims=True)

    # 算出相似矩阵
    S = np.dot(X_train_norm, X_train_norm.T)

    # X_train_binary 为 X_train 的二值化矩阵
    # 用S与X_train_binary相乘，可以直接得到分母的值
    X_train_binary = X_train.copy()
    X_train_binary[X_train_binary != 0] = 1

    # 返回预测矩阵
    return (S @ X_train) / (S @ X_train_binary)

start = time.time()

result = predict_ratings_efficient(X_train)

end = time.time()

# 过滤掉测试集中不存在的值
mask = X_test != 0

# 计算RMSE
RMSE = np.sqrt(np.sum(np.sum((result[mask] - X_test[mask])**2)) / test_length)

print(f"协同过滤算法的RMSE为：{RMSE:.5f}，耗时为：{end - start:.2f}s")

# 计算假设所有打分均为0~5的RMSE
for i in range(6):
    RMSE = np.sqrt(np.sum(np.sum((i - X_test[mask])**2)) / test_length)

print(f"假设所有打分均为{i}的RMSE为：{RMSE:.5f}")
```

对于代码每一行的详细解释已经在注释中给出。注意到这个解法核心代码只有 12-16 的

三行，没有任何 “for” 循环，十分的优雅高效。具体来说就是，公式1的分子可以由 16 行的前半部分得到，分母可以由二值化的 $\mathbf{X}_{\text{train}}$ 与相似矩阵点乘得到，而它们每个元素的除法，就是两个矩阵对应元素相除即可。

代码的输出如下：

```
协同过滤算法的RMSE为： 1.01837，耗时为： 31.10s
假设所有打分均为0的RMSE为： 3.56220
假设所有打分均为1的RMSE为： 2.63002
假设所有打分均为2的RMSE为： 1.77334
假设所有打分均为3的RMSE为： 1.17151
假设所有打分均为4的RMSE为： 1.26497
假设所有打分均为5的RMSE为： 1.95650
```

可以发现，在处理 10000×10000 量级的数据时，只耗时了 $30s$ 左右的时间，这验证了算法的高效性；同时，算法的 RMSE 结果均低于 “假设所有打分为 0-5” 的 RMSE，这验证了算法的有效性。

题目 3：课程里已介绍了矩阵分解的相关知识。对于给定的矩阵 $\mathbf{X}$，可以将其分解为 $\mathbf{U}$、$\mathbf{V}$ 两个矩阵的乘积，使 $\mathbf{UV}$ 的乘积在抑制部分逼近 $\mathbf{X}$。即：$\mathbf{X}_{m*n} \approx \mathbf{U}_{m*k}\mathbf{V}^{\mathsf{T}}_{n*k}$，其中 k 为隐藏的维度，是算法的参数。

基于行为矩阵的低秩假设，可以认为 $\mathbf{U}$ $\mathbf{V}$ 是用户和电影在隐空间的特征表达，它们的乘积矩阵可以用来预测 $\mathbf{X}$ 的未知部分。

本作业可以使用梯度下降法优化求解上述问题。目标函数是：

$$J = \frac{1}{2}\|\mathbf{A} \circ (\mathbf{X} - \mathbf{U}\mathbf{V}^T)\|_F^2 + \lambda\|\mathbf{U}\|_F^2 + \lambda\|\mathbf{V}\|_F^2 \tag{2}$$

其中，$\mathbf{A}$ 是指示矩阵，$\mathbf{A}_{ij} = 1$ 意味着 $\mathbf{X}_{ij}$ 的值为已知，反之亦然。$\circ$ 是阿达马积（即矩阵逐元素相乘）。$\|\cdot\|_F$ 表示矩阵的 Frobenius 范数，计算公式为 $\|\mathbf{A}\|_F = \sqrt{\sum_i \sum_j \mathbf{A}_{ij}^2}$。在目标函数 J 中，第一项为已知值部分，为 $\mathbf{UV}$ 的乘积逼近 $\mathbf{X}$ 的误差；后面的两项是为防止过拟合加入的正则项，λ 为控制正则项大小的参数。

当目标函数取得最小值时，算法得到最优解。可分别对 $\mathbf{U}$ 和 $\mathbf{V}$ 求偏导，结果如下：

$$\frac{\partial J}{\partial \mathbf{U}} = [\mathbf{A} \circ (\mathbf{U}\mathbf{V}^T - \mathbf{X})]\mathbf{V} + 2\lambda\mathbf{U} \tag{3}$$

$$\frac{\partial J}{\partial \mathbf{V}} = [\mathbf{A} \circ (\mathbf{U}\mathbf{V}^T - \mathbf{X})]\mathbf{U} + 2\lambda\mathbf{V} \tag{4}$$

之后，可迭代对 $\mathbf{U}$ 和 $\mathbf{V}$ 行梯度下降更新，具体算法如下：

Initialize U and V (very small random value);
Loop until converge:

$$\mathbf{U} = \mathbf{U} - \alpha\frac{\partial J}{\partial \mathbf{U}};$$

$$\mathbf{V} = \mathbf{V} - \alpha\frac{\partial J}{\partial \mathbf{V}};$$

End loops

算法中 α 为学习率，通常根据具体情况选择 0.0001 到 0.1 之前的实数值。算法的收敛条件，可以选择目标函数 J 的变化量小于某个阈值。

题目要求如下：

1. 对于给定 $k = 50, \lambda = 0.01$ 的情况，画出迭代过程中目标函数值和测试集上 RMSE 的变化，给出最终的 RMSE，并对结果进行简单分析。

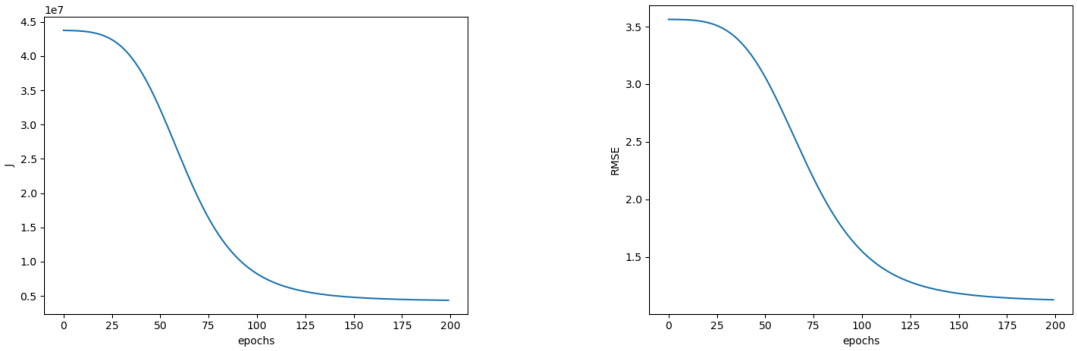
2. 调整 k 的值（如 $20, 50$）和 λ 的值（如 $0.001, 0.1$），比较最终 RMSE 的效果，对结果进行简单分析，选取最优的参数组合。

3. 将题目二和题目三的结果进行对比，讨论两种方法的优缺点。

答：我的矩阵分解的学习代码如下：

```python
import matplotlib.pyplot as plt

def plot_fig(x, y, xl, yl, path):
    plt.plot(x, y)
    plt.xlabel(xl)
    plt.ylabel(yl)
    plt.tight_layout()
    plt.savefig(f"{path}.png")
    plt.close()

def matrix_decomposition(X_train, X_test, test_length, epochs, k = 50, l = 0.01, alpha = 1e-4):
    A = X_train > 0

    U = np.random.randn(X_train.shape[0], k) * 0.1
    V = np.random.randn(X_train.shape[1], k) * 0.1

    mask = X_test != 0

    # 基于梯度下降的矩阵分解
    J, RMSE = np.zeros((epochs)), np.zeros((epochs))
    for i in tqdm(range(epochs)):

        # 对变量求偏导
        temp = A * ((U @ V.T) - X_train)
        dU = temp @ V + 2 * l * U
        dV = temp @ U + 2 * l * V

        # 参数更新
        U -= alpha / (1 + .1 * i) * dU
        V -= alpha / (1 + .1 * i) * dV

        temp = U @ V.T
        J[i] = .5 * np.sum((A * (X_train - temp)) ** 2) + l * (np.sum(U ** 2 + V ** 2))

        RMSE[i] = np.sqrt(np.sum((temp[mask] - X_test[mask]) ** 2) / test_length)


    # 可视化结果
    plot_fig(range(epochs), RMSE, xl='epochs', yl='RMSE', path=f'RMSE_{k}_{l}')
    plot_fig(range(epochs), J, xl='epochs', yl='J', path=f'J_{k}_{l}')

    print(f'k={k}, lambda={l}, RMSE={RMSE[epochs-1]}, J={J[epochs-1]}')
    return RMSE

ks = [20, 50, 100]
lamdas = [1e-3, 1e-2, 1e-1]
RMSEs, times = {}, {}

for k in ks:
    for l in lamdas:
        start = time.time()
        RMSE = matrix_decomposition(X_train, X_test, test_length, epochs=200, k=k, l=l)
        times[(k, l)] = time.time() - start
        RMSEs[(k, l)] = RMSE
```

1. 对于 $k = 50, \lambda = 0.01$ 的情况，迭代过程中目标函数 J 的值和测试集上 RMSE 的变化如图1所示（迭代次数为 200）。可以观察到，随着迭代次数的增加，目标函数 J 的值逐渐减小，并收敛到 $0.4e^7$ 左右，同时 RMSE 的值也逐渐减小，并收敛到 1.1 左右。



(a) 目标函数 J 的值的变化　　(b) 测试集上 RMSE 的变化

图 1: $k = 50, \lambda = 0.01$ 情况下 J 的值和测试集上 RMSE 的变化。

最终的 RMSE 的值为 1.12639。

2. 我选取了 $k = 20, 50, 100, \lambda = 0.001, 0.01, 0.1$ 的组合（迭代次数为 200），代码输出如下：

```
k=20, lambda=0.001, RMSE=1.199704901009163, J=4946663.924701192
k=20, lambda=0.01, RMSE=1.193266476469468, J=4892023.083354976
k=20, lambda=0.1, RMSE=1.2111765109038708, J=5045634.62173047
k=50, lambda=0.001, RMSE=1.127843347966499, J=4357843.756894563
k=50, lambda=0.01, RMSE=1.1263914350087676, J=4348016.115486492
k=50, lambda=0.1, RMSE=1.1408531197635965, J=4466907.596315169
k=100, lambda=0.001, RMSE=1.123459384773856, J=4310706.040847416
k=100, lambda=0.01, RMSE=1.1265778376868703, J=4331229.973901362
k=100, lambda=0.1, RMSE=1.1312125926037473, J=4373110.854280531
```

RMSE 的变化如2所示。可以观察到，当固定 k 的时候，λ 的取值趋向于 0.1 比较好，当固定 λ 的时候，k 的取值越大 RMSE 越好，但是随之而来的是更久的训练时间，如表1所示。最好的参数组合是 $k = 100, \lambda = 0.001$ 时，RMSE 为 1.123459。

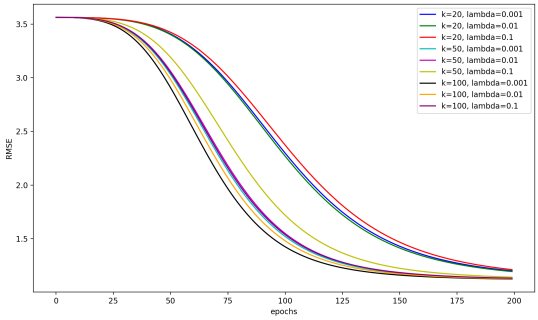


图 2: 不同 k 和 λ 组合的 RMSE 曲线

(k, λ)	Time (s)
(20, 0.001)	270.44
(20, 0.01)	252.51
(20, 0.1)	253.65
(50, 0.001)	312.49
(50, 0.01)	327.92
(50, 0.1)	295.85
(100, 0.001)	362.14
(100, 0.01)	359.44
(100, 0.1)	360.43

表 1: 不同 k 和 λ 组合的耗时

3. **协同过滤与矩阵分解算法的对比：**针对本次实验任务，我协同过滤跑出来的 RMSE 为 1.01837，矩阵分解跑出来的最好的 RMSE 为 1.12346，协同过滤的效果略好于矩阵分解。下面是对这两种算法优缺点的分析：

   协同过滤：

   - 优点：
     - 计算效率高，因为仅需依据用户与商品关联矩阵 S 计算，其中 S_{ij} 表示用户 i 和商品 j 的关联程度。
     - 基于用户行为数据，不需其他先验知识，能够灵活适应用户偏好的变化。
     - 当用户行为数据丰富时，能够提供更加准确的推荐。
   - 缺点：
     - 空间复杂度较高，需要维护用户与商品的相似度矩阵，复杂度为 $O(m \times n)$，其中 m 和 n 分别为用户数和商品数。
     - 需要大量的用户行为数据（显性或隐性）。
     - 假设用户兴趣仅由过往行为决定，忽略了上下文环境的影响。
     - 在数据稀疏的情况下，推荐性能下降。

   矩阵分解：

   - 优点：
     - 空间复杂度较低，仅需存储物品和用户的隐向量矩阵 P 和 Q，复杂度为 $O(k(m+n))$，其中 k 为隐特征的数量。
     - 强大的泛化能力，能在一定程度上解决数据稀疏问题。
     - 提供良好的可扩展性和灵活性，适用于各种规模的数据集。
   - 缺点：

- 计算时间较长，尤其是在大规模数据集上（实验中是协同过滤算法的 5 倍以上）。
- 通常仅使用共现矩阵，无法有效整合用户、商品及上下文特征，可能损失部分语义信息。