深入探索基于矩阵分解和协同过滤的个性化推荐系统及其在Netflix电影推荐中的应用

大数据分析B课程第二次实验项目

叶璨铭 (2024214500) ycm24@mails.tsinghua.edu.cn

2024年11月26日星期二

## 1 绪论

## 2 代码与文档格式说明

本文档使用Jupyter Notebook编写，所以同时包括了实验文档和实验代码。

本次实验项目采用了 Quarto + nbdev 的系统来发布Jupyter Notebook, 因而我们的实验文档导出为pdf和html格式可以进行阅读，而我们的代码也导出为python模块形式，可以作为代码库被其他项目使用。

我们这样做的好处是，避免单独管理一堆 .py 文件，防止代码冗余和同步混乱，py文件和pdf文件都是从.ipynb文件导出的，可以保证实验文档和代码的一致性。

|  |
| --- |
| 重要 |
| 可以通过以下命令安装我们实验的代码：  pip install git+https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data.git  我们的代码导出为了python模块形式，通过以下命令导入：  from thu\_big\_data\_ml.big\_data\_analytics.matrix\_decomposition import \* |

https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data.git 是我们本次大数据机器学习课程实验的代码仓库地址。

而这次作业中，我开发的我们课题组的基础依赖库[ScholarlyInfrastructure](https://github.com/THU-CVML/ScholarlyInfrastructure)也相应地进行了代码更新。这个库对标fastcore库，对AI科研经常会用到的一些基础性地、和Python语言的表达力有关的代码进行了整理，比如PyTorch模型检查、清晰的日志、实验参数管理、异常处理、argmax自动函数优化等。接下来我们也会用到这个项目中的一些代码来完成本次作业。

pip install git+https://github.com/THU-CVML/ScholarlyInfrastructure.git

from scholarly\_infrastructure import \*

以上代码库开源在github，欢迎各位同学、老师们提出宝贵意见，或者加入我们的开发一起完善，构建更加优质的科研工具。

|  |
| --- |
| 重要 |
| 本文档具有一定的交互性，建议使用浏览器打开html文件，这样比pdf文件阅读体验更佳。 |

## 3 实验目的与项目要求

老师给我们的要求是

1. 实现基于用户的协同过滤算法，并在测试集上计算RMSE。
2. 使用梯度下降法实现矩阵分解，优化目标函数，并在测试集上评估效果，尝试不同的参数组合。
3. 比较协同过滤和矩阵分解的优缺点。

~~作为Top1大学的学生~~，我们不仅需要完成以上内容，还需要进行一些深入的思考和探索。

1. 算法优化：针对协同过滤的稀疏性问题，我们计划实现基于近邻选择的优化策略，如KNN-based CF，并尝试利用GPU加速相似度计算。
2. 超参数调优：在矩阵分解中，我们将系统地搜索隐空间维度k和正则化参数λ的最佳组合，使用网格搜索结合交叉验证来确定最优参数。
3. 自定义改进：尝试在矩阵分解中引入非线性变换或自定义正则化项，探索是否能进一步提高推荐精度。
4. 全面对比分析：除了RMSE，我们还将从时间复杂度、可解释性、冷启动问题等方面全面比较协同过滤和矩阵分解的优缺点，并通过可视化手段展示实验结果。

事不宜迟，我们开始动手吧！

## 4 1. 实验数据整理

Netflix 在2006年举办了奖金高达100万美元的Netflix Prize竞赛，旨在改进其电影推荐算法。这一竞赛吸引了大量AI开发者投身于推荐系统相关研究，并开辟了推荐算法商业化的先河。 Netflix电影评价数据集包含来自48万用户对1.7万部电影的评价数据，评价数超过100万条。

我们选择了老师提供的Netflix数据集版本（一个子集），并使用里面的全部一万多个用户和一万多个电影的全量矩阵进行实验。

### 4.1 数据集的自动下载

我们在linux服务器中运行代码，因此优雅的方式是直接在服务器上下载数据集，加快速度。首先我们需要确认数据文件夹在哪里，这个是gitignore的一个文件夹。

from thu\_big\_data\_ml.help import data\_path

download\_path = data\_path / "netflix"  
download\_path.mkdir(exist\_ok=True, parents=True)  
download\_path

Path('/home/ye\_canming/repos/assignments/THU-Coursework-Machine-Learning-for-Big-Data/data/netflix')

根据老师分享的清华网盘链接，我们写出如下代码：

urls = [  
 "https://cloud.tsinghua.edu.cn/seafhttp/files/5f936b33-0b44-4878-8c3c-320ad1863c3b/movie\_titles.txt",   
 "https://cloud.tsinghua.edu.cn/seafhttp/files/54d448d0-88c1-4a6d-b0bc-80f1992deeca/netflix\_test.txt",  
 "https://cloud.tsinghua.edu.cn/seafhttp/files/bfafbb0a-30fc-4215-aee2-4b73cc2d6855/netflix\_train.txt",  
 "https://cloud.tsinghua.edu.cn/seafhttp/files/a228e885-c624-4cde-8c28-b88b3846e405/users.txt",  
]  
# 自动提取文件名，得到 movie\_titles.txt  
# filenames = [url[(url.rfind("F")+1):url.rfind("&dl=1")] for url in urls]  
filenames = [url.split("/")[-1] for url in urls]  
filenames

['movie\_titles.txt', 'netflix\_test.txt', 'netflix\_train.txt', 'users.txt']

pip install py3-wget

import py3\_wget  
for url, file in zip(urls, filenames):  
   
 py3\_wget.download\_file(  
 url=url,  
 output\_path=(download\_path/file).as\_posix(),  
 )

### 4.2 数据读取

import pandas as pd  
import numpy as np

由于我们有很多耗时操作，所以我们需要使用joblib库来将已经进行过的操作缓存到硬盘上，以便下次直接从硬盘中读取。

from thu\_big\_data\_ml.help import joblib\_memory

pd.read\_csv = joblib\_memory.cache(pd.read\_csv) # 尽管都是从磁盘读取，cache之前需要11s来读取 训练集， cache之后不需要解析csv的格式，所以快很多，0s左右就可以读取。

现在我们来读取四个csv表格

# 读取用户列表, 根据老师给的文档的信息，文件有 10000 行，每行一个整数，表示用户的 id，文件对应本次作业的所有用户。  
users = pd.read\_csv(download\_path/'users.txt', header=None, names=['user\_id'])  
print(users.shape)  
users.head()

Sun 2024-12-01 22:30:15.807897

INFO (10000, 1) d=484279;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=667773;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

|  | user\_id |
| --- | --- |
| 0 | 305344 |
| 1 | 387418 |
| 2 | 2439493 |
| 3 | 1664010 |
| 4 | 2118461 |

# 读取训练集，文件包含 689 万条用户打分，每行为一条打分，对应的格式为：用户 id 电影 id 分数 打分日期  
# 我们用英文来标注列的名字。  
train\_data = pd.read\_csv(download\_path/'netflix\_train.txt', sep=' ', header=None, names=['user\_id', 'movie\_id', 'rating', 'date'])  
print(train\_data.shape)  
train\_data.head()

Sun 2024-12-01 22:30:19.294096

INFO (6897746, 4) d=766285;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=912141;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

|  | user\_id | movie\_id | rating | date |
| --- | --- | --- | --- | --- |
| 0 | 305344 | 1 | 1 | 2004-02-08 |
| 1 | 305344 | 3 | 2 | 2003-03-23 |
| 2 | 305344 | 4 | 1 | 2003-12-13 |
| 3 | 305344 | 5 | 1 | 2004-12-15 |
| 4 | 305344 | 6 | 1 | 2003-09-23 |

# 读取测试集, 文件包含约 172 万条用户打分，格式与训练集相同。  
test\_data = pd.read\_csv(download\_path/'netflix\_test.txt', sep=' ', header=None, names=['user\_id', 'movie\_id', 'rating', 'date'])  
print(test\_data.shape)  
test\_data.head()

Sun 2024-12-01 22:30:21.216140

INFO (1719466, 4) d=512624;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=366615;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

|  | user\_id | movie\_id | rating | date |
| --- | --- | --- | --- | --- |
| 0 | 2123534 | 6134 | 4 | 2005-05-31 |
| 1 | 2123534 | 677 | 1 | 2005-06-06 |
| 2 | 2123534 | 3182 | 4 | 2005-06-06 |
| 3 | 2123534 | 7882 | 4 | 2005-06-27 |
| 4 | 2123534 | 6099 | 1 | 2005-05-31 |

# 读取电影名称,文件对应每部电影的年份和名称，格式为：电影 id, 年份, 名称, 各项之间用逗号分隔。  
try:  
 movies = pd.read\_csv(download\_path/'movie\_titles.txt', sep=',', header=None, encoding='utf-8')  
except Exception as e:  
 print(f"Error: {e}")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
[Memory] Calling pandas.io.parsers.readers.read\_csv...  
read\_csv(Path('/home/ye\_canming/repos/assignments/THU-Coursework-Machine-Learning-for-Big-Data/data/netflix/movie\_titles.txt'), sep=',', header=None, encoding='utf-8')

Sun 2024-12-01 22:30:21.895338

INFO Error: Error tokenizing data. C error: Expected 3 fields in line 72, saw 4 d=310515;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=708609;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

可以看到我们遇到了一个报错，72行的数据是这样的“72,1974,At Home Among Strangers, A Stranger Among His Own” 一个电影名自己包括分隔符”,“在里面，导致了pandas读取出错， 怎么解决这个问题呢？

我们可以指定 names ，或者改变 sep使用正则表达式，强制要求有三个字段。

try:  
 movies = pd.read\_csv(download\_path / 'movie\_titles.txt', sep=r'\s\*,\s\*', header=None, encoding='utf-8',  
 names=['movie\_id', 'year', 'title'], engine='python')  
except Exception as e:  
 print(f"Error: {e}")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
[Memory] Calling pandas.io.parsers.readers.read\_csv...  
read\_csv(Path('/home/ye\_canming/repos/assignments/THU-Coursework-Machine-Learning-for-Big-Data/data/netflix/movie\_titles.txt'), sep='\\s\*,\\s\*', header=None, encoding='utf-8', names=['movie\_id', 'year', 'title'], engine='python')

Sun 2024-12-01 22:30:22.187627

INFO Error: 'utf-8' codec can't decode byte 0xe9 in position 2872: invalid continuation byte d=512726;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=471953;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

为了解决编码问题，我们使用 chardet 自动检测数据的 编码格式。

pip install chardet

import chardet

with open(download\_path / 'movie\_titles.txt', 'rb') as f:  
 rawdata = f.read()  
encoding = chardet.detect(rawdata)['encoding']  
encoding

'ISO-8859-1'

try:  
 movies = pd.read\_csv(download\_path / 'movie\_titles.txt', sep=r'\s\*,\s\*', header=None, encoding=encoding,   
 names=['movie\_id', 'year', 'title'], engine='python')  
except Exception as e:  
 print(f"Error: {e}")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
[Memory] Calling pandas.io.parsers.readers.read\_csv...  
read\_csv(Path('/home/ye\_canming/repos/assignments/THU-Coursework-Machine-Learning-for-Big-Data/data/netflix/movie\_titles.txt'), sep='\\s\*,\\s\*', header=None, encoding='ISO-8859-1', names=['movie\_id', 'year', 'title'], engine='python')

Sun 2024-12-01 22:30:34.311269

INFO Error: Expected 3 fields in line 72, saw 4. Error could possibly be due to quotes being d=736669;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=177193;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\  
 ignored when a multi-char delimiter is used.

看来就算指定了 names， pandas无法知道我们面对多个逗号的时候处理的逻辑，而sep使用了正则表达式加上两边空白符对于本例无效，因为我们的分隔符实际上是没有空格的逗号。实际上，我们的规则是第一次遇到的两个逗号算数，后面的逗号无效。

我们现在手动使用Python代码来处理这个复杂的逻辑，最后再转换为dataframe。

with open(download\_path / 'movie\_titles.txt', 'r', encoding=encoding) as file:  
 lines = file.readlines()  
lines[:3]

['1,2003,Dinosaur Planet\n', '2,2004,Isle of Man TT 2004 Review\n', '3,1997,Character\n']

movies = []  
for i, line in enumerate(lines):  
 # 去除最后一个 \n  
 line = line.strip()  
 if line=="": # 最后一行空行  
 continue  
 # 前两个逗号算数，第三个逗号无效  
 splits = line.split(',', 2)  
 # print(splits)  
 try:  
 row\_data = [int(splits[0]), int(splits[1]), splits[2]]  
 except Exception as e:  
 print(e)  
 print(line, splits)  
 break  
 # logger.exception(e)  
 movies.append(row\_data)

Sun 2024-12-01 22:30:36.563813

INFO invalid literal for int() with base 10: 'NULL' d=568677;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=423109;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

Sun 2024-12-01 22:30:36.608295

INFO ('4388,NULL,Ancient Civilizations: Rome and Pompeii', ['4388', 'NULL', 'Ancient d=929713;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=222246;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#55\55;;\  
 Civilizations: Rome and Pompeii'])

我们可以看到再次出现报错，有的电影缺失了year这个字段，并且是用”NULL”字符串表示的。我们可以把它当做None，让Pandas处理为NaN。

from scholarly\_infrastructure import default\_on\_exception

default\_on\_exception?

Signature: default\_on\_exception(func, default\_value=None, verbose=False)  
Docstring: <no docstring>  
File: ~/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/nucleus.py  
Type: function

int\_or\_none = default\_on\_exception(lambda x: int(x), default\_value=None)  
movies = []  
for i, line in enumerate(lines):  
 # 去除最后一个 \n  
 line = line.strip()  
 if line=="": # 最后一行空行  
 continue  
 # 前两个逗号算数，第三个逗号无效  
 splits = line.split(',', 2)  
 row\_data = [int\_or\_none(splits[0]), int\_or\_none(splits[1]), splits[2]]  
 movies.append(row\_data)

# 将列表转换为 DataFrame  
movies = pd.DataFrame(movies, columns=['movie\_id', 'year', 'title'])  
print(movies.shape)  
movies.head()

Sun 2024-12-01 22:30:40.463529

INFO (17770, 3) d=775512;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=218252;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

|  | movie\_id | year | title |
| --- | --- | --- | --- |
| 0 | 1 | 2003.0 | Dinosaur Planet |
| 1 | 2 | 2004.0 | Isle of Man TT 2004 Review |
| 2 | 3 | 1997.0 | Character |
| 3 | 4 | 1994.0 | Paula Abdul's Get Up & Dance |
| 4 | 5 | 2004.0 | The Rise and Fall of ECW |

### 4.3 数据预处理为评分矩阵X

from scipy.sparse import csr\_matrix

csr\_matrix是Compressed Sparse Row matrix，适合用于行操作比较多的场景。在这个推荐系统的问题中，用户是行，电影是列，每个元素是用户的评分。由于用户和电影的数量都很大，而且每个用户只评过很少的电影，所以这个矩阵会非常稀疏。用稀疏矩阵来表示可以节省内存和计算时间。

那么怎么从刚才的原始数据得到评分矩阵呢？首先，我先对用户ID和电影ID分别建立一个字典，这个字典把行的索引（比如0-9999这些用户）和”user\_id”实际的值去做映射。我们必须做这一步，不然索引是不安全的。我们使用行列的索引而非原始的user\_id来做后续的操作。

# 对用户ID进行排序，并映射到行索引  
sorted\_users = users.sort\_values('user\_id')  
user\_to\_row = {user\_id: idx for idx, user\_id in enumerate(sorted\_users['user\_id'])}  
# 对电影ID进行排序，并映射到列索引  
sorted\_movies = movies.sort\_values('movie\_id')  
movie\_to\_col = {movie\_id: idx for idx, movie\_id in enumerate(sorted\_movies['movie\_id'])}

但是需要注意的是，data不一定用到了所有的user\_id 和 movie\_id, 所以实际上movie用到的只有老师说的”10000”多个，而不是 17770个。

train\_data['user\_id'].nunique(), train\_data['movie\_id'].nunique(), test\_data['user\_id'].nunique(), test\_data['movie\_id'].nunique()

(10000, 10000, 10000, 9983)

首先我们还需要搞明白训练集和测试集的关系是什么，他们的用户和电影是否重叠呢？

# 集合取交集  
len(set(train\_data['user\_id'].unique()) & set(test\_data['user\_id'].unique())), len(set(train\_data['movie\_id'].unique()) & set(test\_data['movie\_id'].unique()))

(10000, 9983)

|  |
| --- |
| 注记 |
| 一开始我以为，测试集和训练集的划分是根据不同的用户来操作的，我们需要用已知用户的评分去预测未知用户的评分，现在看来并不是这样的，一个用户实际上在训练集和测试集上都有出现，但是是不同的评分数据。我们接下来推荐系统中实际上是通过其他用户在训练集的评分来补全我们用户的评分，然后再用测试集已知的我们这个用户的评分来计算损失。 |

我们再取并集来看，可以看出不管是训练集还是测试集，一共出现的都只有10000个电影和10000个用户。

# 集合取并集  
len(set(train\_data['user\_id'].unique()) | set(test\_data['user\_id'].unique())), len(set(train\_data['movie\_id'].unique()) | set(test\_data['movie\_id'].unique()))

(10000, 10000)

为了让矩阵是，而不是，我们的索引需要压缩一下。

movies\_set = set(train\_data['movie\_id'].unique()) | set(test\_data['movie\_id'].unique())  
movie\_to\_col = {movie\_id: idx for idx, movie\_id in enumerate(movies\_set)}

现在我们可以安全地从 train\_data 和 tets\_data 得到 了，这个矩阵在《深度学习推荐系统》一书中又称作“共现矩阵”。

在scipy里，csr\_matrix的构造函数可以接受三个参数： values, (rows, cols)，以及形状。 比如 values可以是 [10, 8], rows=[0, 1], cols=[3, 2], 这就表示在 (0, 3)和 (1, 2)两个坐标上矩阵的值是 10 和 8， 其他位置都是0（表示评分未知）。

这样我们就成功规避了老师提到的“计算机计算能力”不足无法处理矩阵的问题。 当然这里只是解决了存储问题，后续这个矩阵怎么去做进一步运算是值得思考的。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L11)

get\_rating\_matrix

get\_rating\_matrix (data:pandas.core.frame.DataFrame)

# @joblib\_memory.cache # data 哈希本身需要很大的开销，不宜 cache  
def get\_rating\_matrix(data:pd.DataFrame)-> csr\_matrix:  
 rows = data['user\_id'].map(user\_to\_row)  
 cols = data['movie\_id'].map(movie\_to\_col)  
 values = data['rating']  
   
 return csr\_matrix((values, (rows, cols)), shape=(len(user\_to\_row), len(movie\_to\_col)))

X\_train = get\_rating\_matrix(train\_data)  
X\_test = get\_rating\_matrix(test\_data)  
X\_train.shape, X\_test.shape, type(X\_train), X\_train.dtype

((10000, 10000), (10000, 10000), <class 'scipy.sparse.\_csr.csr\_matrix'>, dtype('int64'))

如果有些操作不被稀疏矩阵支持，我们可能还是需要稠密矩阵，我们把稠密矩阵也给出。

X\_train\_dense:np.array = X\_train.toarray() # todense也可以  
X\_test\_dense:np.array = X\_test.toarray() # todense也可以  
X\_train\_dense.shape, X\_test\_dense.shape, type(X\_train\_dense), X\_train\_dense.dtype

((10000, 10000), (10000, 10000), <class 'numpy.ndarray'>, dtype('int64'))

## 5 2. 基于用户的协同过滤实现

协同过滤思路很简单，“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”，这里需要注意几个关键的点： 1. 判断用户i和用户j的相似性，是通过训练集i和j的相似性，而不是通过测试集用户i和用户j的相似性，也不是通过测试集i和训练集j的相似性，因为测试集本质上不是矩阵，而是单个评分，每次测试的时候我们知道的只有 user\_id和movie\_id，不能利用测试集的其他信息来判断相似性。 2. 判断用户i是否喜欢电影j，用户i自身和自己很相似，但是因为缺失了数据所以填写了0，万万不可以用矩阵中缺省值“0”加入评分的平均计算，所以寻找“与i相似的用户”时，需要排除用户i自己。

由于协同过滤有很多个不同的版本，而项目文档给出的公式缺乏对我上面说到的几个问题的严格定义，因此我们参考[《深度学习推荐系统》一书中的定义](https://yd.qq.com/web/reader/b7732f20813ab7c33g015deakaab325601eaab3238922e53)来实现代码。 提出协同过滤方法的论文是 《Amazon.com Recommenders Item-to-Item Collaborative Filtering》。

此外，协同过滤和机器学习方法有一定的关联， - 协同过滤和使用cos相似度的weighted的K近邻（sklearn中的KNeighborsRegressor(weights='distance', metric='cosine')）比较相似，区别KNN一般预测一个值或者类别，这里预测整个一行。以及刚才我们提到的第2点，协同过滤不能使用用户自己。 - 计算一个矩阵每一行之间的相似度，把相似度矩阵存下来，也叫做Kernel Method，存下来的矩阵叫做Gram矩阵。

由于K近邻代码上和协同过滤还是有所区别，我们这里不使用K近邻实现推荐系统。

### 5.1 用户相似度计算

我们首先计算每个用户互相之间的相似度。

import torch

# 尝试在GPU上面算，期待会快一些

X\_train\_torch = torch.from\_numpy(X\_train\_dense).float()  
X\_train\_torch = X\_train\_torch.to("cuda")  
try:  
 cosine\_sim = torch.nn.functional.cosine\_similarity(X\_train\_torch[:, None], X\_train\_torch[None, :], dim=2)  
except Exception as e:  
 print(e)

Sun 2024-12-01 22:31:03.171040

INFO CUDA out of memory. Tried to allocate 3725.29 GiB. GPU 0 has a total capacity of 23.69 GiB d=985658;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=91352;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\  
 of which 18.69 GiB is free. Process 1612869 has 1.91 GiB memory in use. Process 1631877 has   
 1.66 GiB memory in use. Including non-PyTorch memory, this process has 1.41 GiB memory in   
 use. Of the allocated memory 1.12 GiB is allocated by PyTorch, and 0 bytes is reserved by   
 PyTorch but unallocated. If reserved but unallocated memory is large try setting   
 PYTORCH\_CUDA\_ALLOC\_CONF=expandable\_segments:True to avoid fragmentation. See documentation   
 for Memory Management   
 (https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)

可惜PyTorch似乎不支持稀疏矩阵，我们暂时无法通过PyTorch在CUDA上面完成余弦相似度的计算。

我们还是使用CPU来做。根据sklearn的文档

X : {array-like, sparse matrix} of shape (n\_samples\_X, n\_features) Input data.

sklearn支持通过稀疏的方式，针对稀疏矩阵进行优化的前提下，来计算相似度，速度应该是合理的。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L23)

get\_similarities

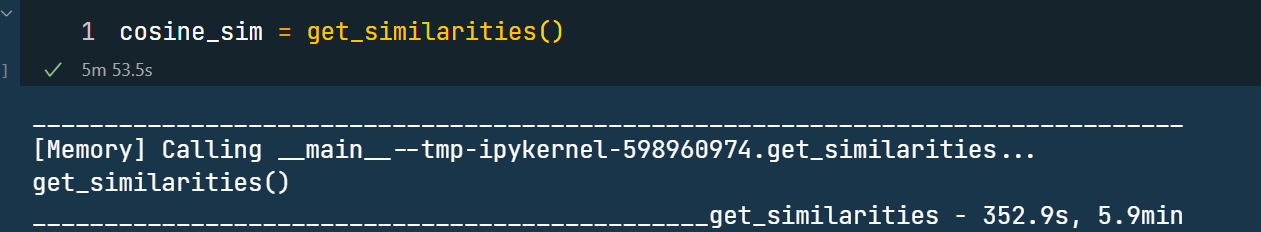
get\_similarities ()

from sklearn.metrics.pairwise import cosine\_similarity

@joblib\_memory.cache  
def get\_similarities():  
 simularities = cosine\_similarity(X\_train)  
 return simularities

# from thu\_big\_data\_ml.matrix\_decomposition import get\_similarities

cosine\_sim = get\_similarities()

我们经过 352.9s, 5.9min 的时间，成功计算出 10000 x 10000 的余弦相似度矩阵。 

由于我们使用了 joblib缓存结果, 第二次加载这个矩阵只需要 3.3s 。

cosine\_sim.shape, type(cosine\_sim), cosine\_sim.dtype

((10000, 10000), <class 'numpy.ndarray'>, dtype('float64'))

cosine\_sim[:5, :5]

array([[1. , 0.29042362, 0.27409518, 0.29692921, 0.34769348],  
 [0.29042362, 1. , 0.29676954, 0.2805517 , 0.23722667],  
 [0.27409518, 0.29676954, 1. , 0.39591191, 0.32875298],  
 [0.29692921, 0.2805517 , 0.39591191, 1. , 0.30646964],  
 [0.34769348, 0.23722667, 0.32875298, 0.30646964, 1. ]])

值得注意的是，cosine\_similarity和cosine\_distances有所不同，这两个加在一起是1，不要搞混了。

### 5.2 “推理”协同过滤模型

按照协同过滤算法本身来说，现在我们本来只需要对测试集的每一个评分开始预测就行了，对测试集有评分的每一个用户和每一个电影，按照公式去计算训练集中相似的用户的评分的加权平均数就可以了。

但是我们可以把思路理地更清晰一些，实际上训练集确定的情况下，整个训练集的矩阵每一行是什么情况都是可以计算的，未来测试集只需要查询对应的结果即可，我们按照机器学习的术语，把这个过程称为“推理”（而刚才计算相似度矩阵的过程可以称为“训练”），得到的结果是新的 10000 x 10000的矩阵，每一行的评分是通过原始矩阵其他行的打分加权平均计算得到的。

如此一来，我们的操作是向量化的。

首先，根据我上面提到的“第二点”，我们需要排除用户自身和自身的相似度，不能用用户自身的评分来预测。实际上，更加合理的做法是计算平均数的时候排除评分为0的情况。

我们可以使用 np.nanmean() 来在忽略缺失值的情况下计算平均值。

example\_array = np.array([np.nan, np.nan, 1])  
np.nanmean(example\_array) # 正确的结果是 1.0，错误的结果是 1/3

1.0

# 如果是加权平均数，也是同理  
example\_weights = [1, 2, 3]  
mask = np.isnan(example\_array)  
weights = np.where(mask, np.nan, example\_weights)  
weighted\_sum = np.nansum(example\_array \* weights)  
weight\_sum = np.nansum(weights)  
weighted\_mean = weighted\_sum / weight\_sum  
weighted\_mean # 应当仍是1 , 而不是 0.5

1.0

理论清晰了，刚才我们已经知道怎么对向量做操作，现在我们可以编写代码来排除没有评分的列的情况下求加权平均数，对整个矩阵去做操作。

X\_train\_dense = X\_train\_dense.astype(np.float64)

from copy import deepcopy  
from tqdm import tqdm

# 评分范围是 1-5, 所以0就是缺失值，刚才稀疏矩阵的默认设为了0，我们需要nan  
X\_train\_dense\_nan = deepcopy(X\_train\_dense)  
X\_train\_dense\_nan[X\_train\_dense\_nan==0] = np.nan  
X\_train\_dense\_nan[0]

array([nan, nan, nan, ..., nan, nan, nan])

np.ones((2, 3)).sum(axis=0)

array([2., 2., 2.])

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L59)

get\_X\_train\_weighted

get\_X\_train\_weighted ()

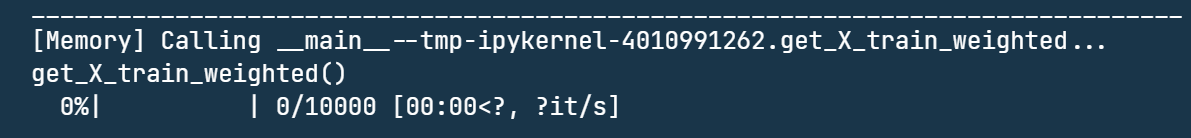
def compute\_weighted\_sum\_on\_matrix(cosine\_sim, X\_train\_dense\_nan):  
# 创建一个与 X\_train\_dense 相同大小的矩阵，用于存储加权平均数  
 X\_train\_weighted = np.zeros\_like(X\_train\_dense\_nan)  
  
 # 遍历 X\_train\_dense 的每一行  
 for i in tqdm(range(X\_train\_dense\_nan.shape[0])):  
 # 获取第 i 行的权重  
 weights = cosine\_sim[i, :]  
   
 # 复制这个权重到整个矩阵的维度，方便后面掩码操作  
 weights = np.repeat(weights, X\_train\_dense\_nan.shape[1]).reshape(X\_train\_dense\_nan.shape[0], X\_train\_dense\_nan.shape[1])  
   
   
 # 创建一个掩码，是 nan的就是True  
 mask = np.isnan(X\_train\_dense\_nan)  
   
   
 # 将权重中的对应位置设置为 np.nan  
 weights = np.where(mask, np.nan, weights)  
   
 # 计算加权平均数，忽略 np.nan 值  
 X\_train\_weighted[i, :] = np.nansum(X\_train\_dense\_nan \* weights, axis=0) / np.nansum(weights, axis=0)  
  
 # X\_train\_weighted 现在是一个 mxn 的矩阵，其中每一行是忽略 np.nan 的加权平均数  
 return X\_train\_weighted  
  
@joblib\_memory.cache  
def get\_X\_train\_weighted():  
 return compute\_weighted\_sum\_on\_matrix(cosine\_sim, X\_train\_dense\_nan)

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L32)

compute\_weighted\_sum\_on\_matrix

compute\_weighted\_sum\_on\_matrix (cosine\_sim, X\_train\_dense\_nan)

# X\_train\_weighted = get\_X\_train\_weighted()

可以看到速度非常慢，不可接受（4分钟连第一个计算也没有完成），我们需要进行一些优化。 

#### 5.2.1 速度优化——矩阵乘法

如果我们不考虑加权平均数需要忽略没有评分的其他用户这一问题，直接把0当做是有效的评分，并且基于相似度直接计算加权平均数，那么我们马上就发现其实矩阵乘法 cosine\_sim @ X\_train\_dense 是有意义的，这就等价于每一行的 cosine\_sim 线性组合 X\_train\_densede 所有行向量。这个离加权平均数只差了一个归一化的步骤，新的矩阵的每一行需要 除以 cosine\_sim 那一行的和。

weight\_sum = cosine\_sim.sum(axis=1) # 每一行的和  
weight\_sum.shape

(10000,)

X\_train\_pred\_wrong = (cosine\_sim @ X\_train\_dense) / weight\_sum[:, np.newaxis] # 升维变成一列向量，这样才能每一行去除这个值而不是每一列去除。  
X\_train\_pred\_wrong.shape

(10000, 10000)

X\_train\_pred\_wrong[0]

array([0.0410934 , 0.00174947, 0.10677668, ..., 0.00775722, 0.01182744,  
 0.02292204])

当然上面得到的结果是错误的，我们看到里面的值都是“0.0410934”之类的，明显小于1，不可能是实际的评分取值，这就是因为没有考虑到刚才我们说的排除没有和商品交互的用户提供的0评分的问题。

我们仔细观察上面的公式，可以发现 cosine\_sim @ X\_train\_dense 其实计算的是正确的，因为评分为0的时候自然不会算进去，问题出在/ weight\_sum[:, np.newaxis] 这一步把评分为0的用户也算了进去。

因此，正确的解决办法是，对每一个用户和每一个物品，都有不同的 weight\_sum。 给定一个用户的时候，确定了和其他用户的相似性，但是不同物品有不同的nan mask，从而weight的加和有所不同。

我们经过一段时间的思考，可以注意到，这其实也是可以通过一个矩阵乘法来解决的。首先我们定义mask矩阵，如果是0（缺失评分）mask就是1，否则就是0。我们注意到(1-mask)矩阵，如果当做是一行列向量，那么每一个列向量就表示了物品对用户的选择，也就是这个物品有哪些用户是有评分的。使用这个列向量去对cosine\_sim进行线性组合，其含义就是在进行一个选择性求和，只有非nan的地方才会相加。

mask = np.isnan(X\_train\_dense\_nan)  
weight\_sum\_correct = cosine\_sim@(1-mask)  
weight\_sum\_correct.shape

(10000, 10000)

X\_train\_pred = (cosine\_sim @ X\_train\_dense) / weight\_sum\_correct  
X\_train\_pred.shape

(10000, 10000)

最终我们的公式非常的简洁 (cosine\_sim @ X\_train\_dense) / (cosine\_sim @ (X\_train\_dense!=0)), 其中 @ 表示矩阵乘法， / 表示逐元素相除。

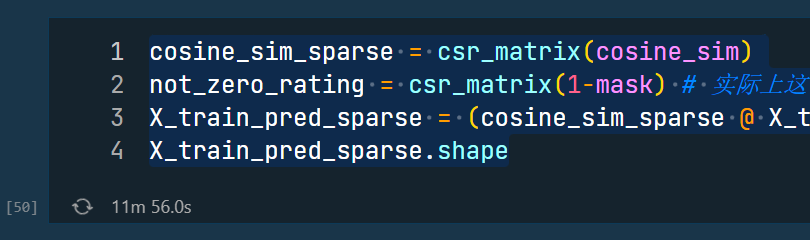
X\_train\_pred[0]

array([3.57833886, 1.76862034, 3.38810491, ..., 2.66767709, 1.94945024,  
 2.63015681])

可以看到这一次我们得到的数值是正确的，是1-5的评分能得到的平均数。 网络上很多协同过滤的博客参差不齐，都没有提到这个公式，都是用for去写的，而《深度学习推荐系统》一书也没有涉及到这个细节。这里我们自己通过数学推导，终于得到了正确而高效的计算公式。

刚才我们是用稠密矩阵来实现对应的公式的，由于矩阵乘法已经被numpy库深度优化，所以我们的计算很高效。下面我们尝试稀疏矩阵实现同样的计算。

cosine\_sim\_sparse = csr\_matrix(cosine\_sim)  
not\_zero\_rating = csr\_matrix(1-mask) # 实际上这个很稀疏，很多元素都是0  
X\_train\_pred\_sparse = (cosine\_sim\_sparse @ X\_train) / (cosine\_sim\_sparse @ not\_zero\_rating)  
X\_train\_pred\_sparse.shape  
np.allclose(X\_train\_pred, X\_train\_pred\_sparse.toarray())

可以发现稀疏矩阵的速度在这个数据规模上和这个问题上，比稠密矩阵的还要慢不少，无法在合理的时间内完成。 

这可能是因为我们得到的最终结果是稠密的，经过矩阵乘法之后稀疏矩阵就不再稀疏，因而算法并不高效，需要算出10000\*10000个点，还不如稠密矩阵表示。

### 5.3 评价协同过滤模型

现在我们有了 X\_train\_pred, 包括了对所有用户在所有物品上的评分，接下来只需要与 X\_test 对比就可以了。

但是需要注意的是，X\_test 并没有包含所有用户在所有物品上的评分，如果直接对这两个矩阵展开为向量去做RMSE，得到的结果当然是错误的。因为这样错误地把0评分和预测评分去做比较。

我们可以for遍历所有的测试数据，然后去查询 X\_train\_pred 中对应的评分，然后再去计算指标。我们也可以对矩阵去进行操作。

test\_mask = (X\_test\_dense!=0) # 获得一个 mask，在老师给的文档中也称为“指示矩阵”  
masked\_X\_train\_pred = test\_mask \* X\_train\_pred # 把 X\_test\_dense 没有评分的地方的 X\_train\_pred 也变成0 （这里通过乘法，乘1乘0实现，也可以通过赋值实现）  
n = test\_mask.sum() # 计算有评分的数量  
rmse = np.sqrt(((X\_test\_dense-masked\_X\_train\_pred)\*\*2).sum() / n) # 计算 RMSE，因为没有评分的地方大家都是0，所以计算出来是对的。  
rmse, n

(1.0183690394072489, 1719466)

我们也可以使用 PyTorch 来计算 RMSE，做出一个loss function。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L69)

masked\_rmse\_loss

masked\_rmse\_loss (reconstructed:torch.Tensor, matrix:torch.Tensor,  
 verbose:bool=False, do\_ensure\_tensor:bool=True)

import torch

def ensure\_tensor(x:np.ndarray|list, device:str="cuda:2"):  
 # 确保输入loss函数的数据是张量  
 return torch.tensor(x).to(device)  
def masked\_rmse\_loss(reconstructed:torch.Tensor, matrix:torch.Tensor, verbose:bool=False, do\_ensure\_tensor:bool=True)->torch.Tensor:  
 # 首先确保类型是tensor，而且device想同，这样才能在GPU上算Loss  
 if do\_ensure\_tensor:  
 reconstructed = ensure\_tensor(reconstructed)  
 matrix = ensure\_tensor(matrix)  
 # 获得一个 mask，在老师给的文档中也称为“指示矩阵”  
 test\_mask = (matrix!= 0)  
 n = test\_mask.sum()  
 if verbose:  
 print(f"Number of non-zero elements in the matrix: {n}")  
 # 只对mask后的元素做Loss计算  
 masked\_matrix = matrix[test\_mask]  
 masked\_reconstructed = reconstructed[test\_mask]  
 return torch.sqrt(torch.mean((masked\_reconstructed - masked\_matrix)\*\*2))

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L66)

ensure\_tensor

ensure\_tensor (x:numpy.ndarray|list, device:str='cuda:2')

masked\_rmse\_loss(ensure\_tensor(X\_train\_pred), ensure\_tensor(X\_test\_dense), verbose=True)

/tmp/ipykernel\_1705708/3678547022.py:4: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires\_grad\_(True), rather than torch.tensor(sourceTensor).  
 return torch.tensor(x).to(device)

Wed 2024-11-27 18:38:57.454612

INFO Number of non-zero elements in the matrix: 1719466 d=124771;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=143256;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

tensor(1.0184, device='cuda:2', dtype=torch.float64)

刚才我们算出了在测试集上的RMSE，同理，其实我们也可以计算在训练集上的RMSE，这应该并不是0，因为协同过滤算法不仅考虑了原本用户自己的评分，还把相似用户的评分考虑进来了。

masked\_rmse\_loss(ensure\_tensor(X\_train\_pred), ensure\_tensor(X\_train\_dense), verbose=True)

/tmp/ipykernel\_1705708/3678547022.py:4: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires\_grad\_(True), rather than torch.tensor(sourceTensor).  
 return torch.tensor(x).to(device)

Wed 2024-11-27 18:39:01.089009

INFO Number of non-zero elements in the matrix: 6897746 d=638104;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=740898;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

tensor(1.0127, device='cuda:2', dtype=torch.float64)

可以看出对于UserCF算法，训练集和测试集的RMSE误差差不多，训练集稍微低一些。

## 6 3. 基于梯度下降的矩阵分解算法实现

我们首先复习一下吴老师课程讲解的内容。

老师课上强调，矩阵分解可以把PCA、SVD、LDA三个重要的方法统一起来。

我们可以使用 PyTorch 来实现梯度下降矩阵分解，但是这里我们同时也尝试一下 jax 来实现。因为 jax 是 谷歌提出的新一代框架，结合了 NumPy 的接口和自动微分功能，以及强大的编译优化，据一些研究指出具有较好的性能。

pip install -U "jax[cuda12]"  
pip install flax  
pip install treescope

### 6.1 模型定义

首先我们需要定义我们的模型，具体来说需要定义模型的参数 U和V。U和V建模了评分矩阵。

# 定义分解矩阵的大小  
k = 50 # 隐向量维度  
m, n = X\_train\_torch.shape

import torch  
import torch.nn as nn  
import torch.optim as optim

为了让代码规范一些，我们把参数写为 module 形式。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L90)

MatrixFactorization

MatrixFactorization (n\_users:int, n\_items:int, k)

\*Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes::

import torch.nn as nn  
import torch.nn.functional as F  
  
class Model(nn.Module):  
 def \_\_init\_\_(self) -> None:  
 super().\_\_init\_\_()  
 self.conv1 = nn.Conv2d(1, 20, 5)  
 self.conv2 = nn.Conv2d(20, 20, 5)  
  
 def forward(self, x):  
 x = F.relu(self.conv1(x))  
 return F.relu(self.conv2(x))

Submodules assigned in this way will be registered, and will have their parameters converted too when you call :meth:to, etc.

.. note:: As per the example above, an \_\_init\_\_() call to the parent class must be made before assignment on the child.

ivar training: Boolean represents whether this module is in training or evaluation mode. :vartype training: bool\*

|  | **Type** | **Details** |
| --- | --- | --- |
| n\_users | int |  |
| n\_items | int | 定义分解矩阵的大小 |
| k |  | 隐向量维度 |

class MatrixFactorization(nn.Module):  
 def \_\_init\_\_(self, n\_users:int, n\_items:int, # 定义分解矩阵的大小  
 k # 隐向量维度  
 ):  
 super().\_\_init\_\_()  
 self.U = nn.Parameter(torch.randn(n\_users, k))  
 self.V = nn.Parameter(torch.randn(n\_items, k))  
   
 def forward(self):  
 return torch.matmul(self.U, self.V.t())

model = MatrixFactorization(m, n, k)  
model().shape

torch.Size([10000, 10000])

我们再用 jax/flax 框架来尝试一下。

import jax  
import jax.numpy as jnp  
from jax import grad, jit, vmap # 这三个函数在 jax 中叫做 "transformations", 意思是对函数进行操作的函数（也可以说是泛函、算子），这三个函数分别作用是 求导，即时编译，向量化。  
import jax.random as jrandom # 为了和 torch.random 做区分，我们导入叫做 jrandom

# 再尝试一下jax

# jax不需要 requires\_grad 因为求导的逻辑不一样，jax待会会直接定义导函数。  
# 在 jax 当中随机数非常重要。 jax的函数是纯函数，不能有副作用，而且输入相同时输出必须相同。所以随机数必须通过显式地传入随机种子，来让随机函数是纯函数。  
key = jrandom.PRNGKey(42)  
jU = jrandom.normal(key, (m, k))  
jV = jrandom.normal(key, (n, k))

Sun 2024-12-01 22:32:30.655660

INFO Unable to initialize backend 'rocm': module 'jaxlib.xla\_extension' has no attribute d=533246;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/jax/\_src/xla\_bridge.py\xla\_bridge.py;;\:d=959507;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/jax/\_src/xla\_bridge.py#906\906;;\  
 'GpuAllocatorConfig'

Sun 2024-12-01 22:32:30.700398

INFO Unable to initialize backend 'tpu': INTERNAL: Failed to open libtpu.so: libtpu.so: d=145576;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/jax/\_src/xla\_bridge.py\xla\_bridge.py;;\:d=226984;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/jax/\_src/xla\_bridge.py#906\906;;\  
 cannot open shared object file: No such file or directory

我们可以直接使用jax对 jU 和 jV 去做梯度下降，但是我们写得更加规范一些，现在我们使用flax框架把矩阵分解也写为一个Module。

注意，根据 https://github.com/google/flax , flax 有两套不同的API，一套是 from flax import linen, 一套是 from flax import nnx。 > Released in 2024, Flax NNX is a new simplified Flax API that is designed to make it easier to create, inspect, debug, and analyze neural networks in JAX. Flax NNX evolved from the Flax Linen API, which was released in 2020 by engineers and researchers at Google Brain in close collaboration with the JAX team.

后者与PyTorch更加接近，是flax的最新设计，我们决定使用后者。

from flax import linen as jnn # 为了和 torch.nn 做区分，我们导入叫做 jnn，和flax官方的写法不同  
from flax import nnx # 导入 nnx 库，里面包含了一些常用的网络层  
from fastcore.all import store\_attr # 导入 fastcore 基础库的 store\_attr 函数，用来方便地存储类的属性，这样Python面向对象写起来不那么冗长。 请 pip install fastcore。  
import treescope # flax 的 可视化

# 定义 MatrixFactorization 模型  
# 注意 flax 使用了 Python标准库 `dataclasses`， 因为面向对象的定义风格更加简洁，使用类变量specify了init的参数。  
class JaxMatrixFactorization(nnx.Module):  
 def \_\_init\_\_(self, n\_users:int, n\_items:int, # 定义分解矩阵的大小  
 k:int, # 隐向量维度  
 \*, rngs: nnx.Rngs # 在 jax 中随机种子非常重要。  
 ):  
 super().\_\_init\_\_()  
 # store\_attr()  
 key = rngs.params()  
 self.U = nnx.Param(jrandom.normal(key, (n\_users, k)))  
 self.V = nnx.Param(jrandom.normal(key, (n\_items, k)))  
 # 如果我们还有子模块，flax要求把 rngs 传递下去。  
 def \_\_call\_\_(self):  
 # return jnp.dot(self.U, self.V.T) # 不能使用这个，因为 nnx.Param 和 jnp.array 不一样，会导致 jax 编译错误，这个是jax的bug。  
 return self.U @ self.V.T

# 现在我们可以构建这个模型  
rngs=nnx.Rngs(params=42)  
jmodel = JaxMatrixFactorization(m, n, k, rngs=rngs)  
print(jmodel().shape)  
nnx.display(jmodel)

Sun 2024-12-01 22:32:36.521590

INFO (10000, 10000) d=182805;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=300023;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

可见目前来看新版的 nnx 接口与PyTorch基本一样，只不过强调随机数种子的传递。旧版的linen则是有较大的不同，它并不能在类中储存状态，需要递归地从模型中调用初始化函数整合出一个大的参数向量出来。

### 6.2 损失函数定义

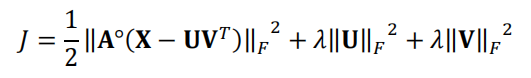


image.png

注意在 PyTorch 的设计哲学中，正则化项是通过优化器weight\_decay选项来实现的，而不是在损失函数中计算。 这是因为损失函数通常输入是 (y\_pred, y\_true) 的形式，不包含模型参数，这样才能把逻辑解耦开来。所以如果我们在loss就实现 l2 regularization，是很麻烦而且没有必要的。而优化器对模型的参数有完全的访问权，可以精确控制哪些部分的参数使用什么学习率和什么正则化。

所以我们下面损失函数暂时忽略 ，稍后会加入回来。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L123)

jax\_masked\_mse\_loss

jax\_masked\_mse\_loss (reconstructed:jax.Array, matrix:jax.Array)

# 定义PyTorch的损失函数  
def masked\_mse\_loss(reconstructed:torch.Tensor, matrix:torch.Tensor)->torch.Tensor:  
 observed\_indices = torch.where(matrix != 0) # A 矩阵，表示哪里是有评分的，只在有评分的地方算loss。  
 return 0.5\*torch.mean((reconstructed[observed\_indices] - matrix[observed\_indices])\*\*2)  
  
# 同理，定义 jax 损失函数  
def jax\_masked\_mse\_loss(reconstructed:jnp.ndarray, matrix:jnp.ndarray)->jnp.ndarray:  
 observed\_indices = jnp.where(matrix != 0) # A 矩阵，表示哪里是有评分的，只在有评分的地方算loss。  
 return 0.5\*jnp.mean((reconstructed[observed\_indices] - matrix[observed\_indices])\*\*2)

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L118)

masked\_mse\_loss

masked\_mse\_loss (reconstructed:torch.Tensor, matrix:torch.Tensor)

# 测试一下我们刚写的函数  
masked\_mse\_loss(model(), model()) # 应当是0

tensor(0., grad\_fn=<MulBackward0>)

jax\_masked\_mse\_loss(jmodel(), jmodel()) # 应当是0

Array(0., dtype=float32)

可以看到 jax 和 torch 基本都是遵循numpy接口的，所以写法基本一样。由于PyTorch的求导方式是动态构建计算图，所以我们在上面的输出可以看到 tensor 带有一个 grad\_fn 属性，用来记录计算图，而 jax 的输出则没有。

### 6.3 3.a 梯度下降训练分解矩阵U和V (k=50, λ=0.01)

# 参考推荐系统库 https://surprise.readthedocs.io/en/stable/matrix\_factorization.html 的默认参数设置，  
# lmd = 2e-2  
lr = 5e-3  
  
# 参考老师给的设置  
lmd = 0.01  
k = 50  
  
# 收敛条件  
max\_epochs = 100000  
# required\_rmse = 1.0  
# required\_delta\_loss = 1e-4  
required\_delta\_loss = 1e-6

#### 6.3.1 PyTorch 实现训练矩阵分解模型

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L131)

train\_matrix\_factorization

train\_matrix\_factorization (X\_train\_dense:<built-infunctionarray>,  
 X\_test\_dense:<built-infunctionarray>,  
 k:int=50, lmd:float=0.02, lr:float=0.005,  
 max\_epochs:int=100000,  
 required\_delta\_loss:float=0.01,  
 random\_state=42, device='cuda:4')

import lightning as L # PyTorch Lightning库，这里我们只是用它来固定随机数种子  
from tqdm import tqdm # 进度条库

def train\_matrix\_factorization(X\_train\_dense:np.array, X\_test\_dense:np.array, k:int = 50,   
 lmd:float = 2e-2, lr:float = 5e-3, max\_epochs:int = 100000,   
 required\_delta\_loss:float = 1e-2,  
 random\_state=42, device = 'cuda:4'):  
 # 设置 PyTorch 随机种子，防止结果不一致。  
 L.seed\_everything(random\_state)   
 # 输入数据转为 PyTorch 张量  
 X\_train\_torch = torch.from\_numpy(X\_train\_dense).to(device)  
 X\_test\_torch = torch.from\_numpy(X\_test\_dense).to(device)  
 # 模型定义  
 m, n = X\_train\_torch.shape  
 model = MatrixFactorization(m, n, k).to(device)  
 model = torch.compile(model)  
 # 优化器定义  
 optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight\_decay=lmd)  
 # 指标  
 metrics = []  
 # 优化循环  
 bar = tqdm(range(max\_epochs))  
 previous\_loss = 0  
 for epoch in bar:  
 optimizer.zero\_grad()  
 pred\_matrix = model()  
 loss = masked\_mse\_loss(pred\_matrix, X\_train\_torch)  
 loss.backward()  
 optimizer.step()  
 # 计算指标  
 with torch.no\_grad():  
 # RMSE  
 train\_rmse = masked\_rmse\_loss(pred\_matrix, X\_train\_torch, do\_ensure\_tensor=False)  
 test\_rmse = masked\_rmse\_loss(pred\_matrix, X\_test\_torch, do\_ensure\_tensor=False)  
 # loss 变化  
 loss\_item = loss.item()  
 delta\_loss = abs(loss\_item - previous\_loss)  
 previous\_loss = loss\_item  
   
 metric = dict(  
 loss=loss\_item,  
 train\_rmse = train\_rmse.item(),  
 test\_rmse = test\_rmse.item(),   
 delta\_loss = delta\_loss  
 )  
 # 指标记录  
 metrics.append(metric)  
 bar.set\_postfix(\*\*metric)  
 # 中止条件  
 if delta\_loss<required\_delta\_loss:  
 break  
   
 return model, metrics

model, metrics = train\_matrix\_factorization(X\_train\_dense, X\_test\_dense, k=k, lmd=lmd, lr=lr, max\_epochs=max\_epochs, required\_delta\_loss=required\_delta\_loss)

INFO: Seed set to 42

Sun 2024-12-01 22:33:57.929346

INFO Seed set to 42 d=288389;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/lightning/fabric/utilities/seed.py\seed.py;;\:d=256787;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/lightning/fabric/utilities/seed.py#57\57;;\

9%|▉ | 8861/100000 [05:14<53:57, 28.15it/s, delta\_loss=9.83e-7, loss=0.229, test\_rmse=0.943, train\_rmse=0.676]

metrics\_df = pd.DataFrame(metrics)  
metrics\_df

|  | loss | train\_rmse | test\_rmse | delta\_loss |
| --- | --- | --- | --- | --- |
| 0 | 31.268774 | 7.908069 | 7.916553 | 3.126877e+01 |
| 1 | 30.852995 | 7.855316 | 7.867649 | 4.157791e-01 |
| 2 | 30.443527 | 7.803016 | 7.819154 | 4.094677e-01 |
| 3 | 30.040384 | 7.751179 | 7.771080 | 4.031429e-01 |
| 4 | 29.643579 | 7.699816 | 7.723437 | 3.968048e-01 |
| ... | ... | ... | ... | ... |
| 8857 | 0.228548 | 0.676088 | 0.943370 | 1.043081e-06 |
| 8858 | 0.228547 | 0.676087 | 0.943389 | 1.013279e-06 |
| 8859 | 0.228546 | 0.676085 | 0.943408 | 1.013279e-06 |
| 8860 | 0.228545 | 0.676084 | 0.943427 | 1.043081e-06 |
| 8861 | 0.228544 | 0.676082 | 0.943446 | 9.834766e-07 |

#### 6.3.2 Jax/Flax 实现训练矩阵分解模型

from functools import partial  
import optax

# 训练函数  
# @nnx.jit # nnx 提供的 算子，进行自动状态管理  
@partial(nnx.jit, static\_argnums=(3, )) # 静态参数，用于固定模型和优化器参数, 这个是为了解决一些不能被 jit 但是可以hash的参数，这里不能这样解决。参考 https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeError  
def train\_step(jmodel:JaxMatrixFactorization, joptimizer:nnx.Optimizer, X\_train\_jnp:jnp.ndarray, criterion=jax\_masked\_mse\_loss):  
 def loss\_fn(jmodel:JaxMatrixFactorization):  
 pred\_matrix = jmodel()  
 return criterion(pred\_matrix, X\_train\_jnp)  
 loss, grads = nnx.value\_and\_grad(loss\_fn)(jmodel) # 这是另一个nnx算子，自动求导函数、  
 joptimizer.update(grads) # In place 的更新操作，更新优化器参数和模型参数。  
 return loss

# 构建jax的优化器  
joptimizer = nnx.Optimizer(jmodel, optax.adamw(lr))  
X\_train\_jnp = jnp.array(X\_train\_dense) # 从numpy array 转换为 jax array  
try:  
 loss = train\_step(jmodel, joptimizer, X\_train\_jnp)   
except Exception as e:  
 print(e)

Sun 2024-12-01 22:39:27.752790

INFO Abstract tracer value encountered where concrete value is expected: traced array with shape d=288389;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=256787;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\  
 int32[]   
 The size argument of jnp.nonzero must be statically specified to use jnp.nonzero within JAX   
 transformations.   
 The error occurred while tracing the function train\_step at   
 /tmp/ipykernel\_2142274/4108683153.py:3 for jit. This concrete value was not available in   
 Python because it depends on the value of the argument X\_train\_jnp.   
   
 See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeError

这是 jax jit编译时会遇到的典型问题，问题的根源在于我们刚才定义 [jax\_masked\_mse\_loss](https://Open-Book-Studio.github.io/THU-Coursework-Machine-Learning-for-Big-Data/coding_projects/big_data_analytics/P2_Matrix-Decomposition/decomposition.html#jax_masked_mse_loss) 函数的时候，使用到了 jnp.where 函数（使用到 jnp.nonzero函数），这个函数如果输入并不知道 size，就无法进行编译。

我们现在来改进一下这个函数。刚才 where 是选择了 indices 出来，所以导致了计算具有动态性，不利于jax的静态编译，我们可以使用另一种计算方式。

@jit  
def compilable\_jax\_masked\_mse\_loss(reconstructed:jnp.ndarray, matrix:jnp.ndarray)->jnp.ndarray:  
 mask = (matrix!=0) # 获得一个 mask，在老师给的文档中也称为“指示矩阵”  
 masked\_reconstructed = mask \* reconstructed # 把 matrix 没有评分的地方的 reconstructed 也变成0 （这里通过乘法，乘1乘0实现，也可以通过赋值实现）  
 n = mask.sum() # 计算有评分的数量  
 rmse = jnp.sqrt(((matrix-masked\_reconstructed)\*\*2).sum() / n) # 计算 RMSE，因为没有评分的地方大家都是0，所以计算出来是对的。  
 return rmse

loss = compilable\_jax\_masked\_mse\_loss(jmodel(), jmodel())  
loss, float(loss) # 单个scalar转化为float

(Array(0., dtype=float32), 0.0)

使用 jit 算子对 compilable\_jax\_masked\_mse\_loss 操作，然后计算一次，可以看到计算成功了，这个函数是可以被jax编译的。

现在我们可以来优化了。

import optuna

# @jit  
def train\_matrix\_factorization\_jax(X\_train\_dense:np.array, X\_test\_dense:np.array, k:int = 50,   
 lmd:float = 2e-2, lr:float = 5e-3, max\_epochs:int = 100000, required\_delta\_loss:float = 1e-3,  
 random\_state=42,  
 trial:optuna.Trial=None, critical\_metric="test\_rmse" # 用于下一章节的调优的 Pruning   
 ):  
 X\_train\_jnp = jnp.array(X\_train\_dense) # 从numpy array 转换为 jax array  
 X\_test\_jnp = jnp.array(X\_test\_dense) # 从numpy array 转换为 jax array  
 # 模型定义  
 m, n = X\_train\_jnp.shape  
 rngs=nnx.Rngs(params=random\_state)  
 jmodel = JaxMatrixFactorization(m, n, k, rngs=rngs)  
 # 优化器  
 joptimizer = nnx.Optimizer(jmodel, optax.adamw(learning\_rate=lr, weight\_decay=lmd))  
 jmetrics = [] # 指标  
 bar = tqdm(range(max\_epochs))  
 previous\_loss:float = 0.0  
 for epoch in bar:  
 pred\_matrix = jmodel()  
 loss = train\_step(jmodel, joptimizer, X\_train\_jnp, criterion=compilable\_jax\_masked\_mse\_loss)  
   
 # 指标记录  
 train\_rmse = jnp.sqrt(loss)  
 test\_rmse = jnp.sqrt(compilable\_jax\_masked\_mse\_loss(pred\_matrix, X\_test\_jnp))  
 # 有关收敛  
 loss\_item = float(loss) # 把单个 scalar 转化为 float， 但是需要注意这个会让jit无法在外层编译  
 delta\_loss:float = abs(loss\_item - previous\_loss)  
 previous\_loss = loss\_item  
   
 metric = dict(  
 loss = loss\_item,   
 train\_rmse = float(train\_rmse),  
 test\_rmse = float(test\_rmse),   
 delta\_loss = delta\_loss  
 )  
 # 指标的记录  
 jmetrics.append(metric)  
 bar.set\_postfix(\*\*metric)  
 # optuna调参记录  
 if trial is not None:  
 for k, v in metric.items():  
 trial.set\_user\_attr(k, v)  
 trial.report(metric[critical\_metric], step=epoch)  
 if trial.should\_prune():  
 raise optuna.TrialPruned()  
  
 # 收敛条件  
 if delta\_loss<required\_delta\_loss:  
 break  
 return jmodel, jmetrics

# 设置 jax 计算的设备  
jax.default\_device = jax.devices("gpu")[1]  
jax.default\_device

CudaDevice(id=1)

jmodel, jmetrics = train\_matrix\_factorization\_jax(X\_train\_dense, X\_test\_dense, k=k, lmd=lmd, lr=lr, max\_epochs=max\_epochs, required\_delta\_loss=required\_delta\_loss)

6%|▌ | 5503/100000 [01:23<23:58, 65.69it/s, delta\_loss=8.34e-7, loss=0.676, test\_rmse=0.978, train\_rmse=0.822]

jmetrics\_df = pd.DataFrame(jmetrics)  
jmetrics\_df

|  | loss | train\_rmse | test\_rmse | delta\_loss |
| --- | --- | --- | --- | --- |
| 0 | 7.955531 | 2.820555 | 2.821863 | 7.955531e+00 |
| 1 | 7.902468 | 2.811133 | 2.813115 | 5.306339e-02 |
| 2 | 7.849845 | 2.801758 | 2.804412 | 5.262232e-02 |
| 3 | 7.797671 | 2.792431 | 2.795755 | 5.217409e-02 |
| 4 | 7.745949 | 2.783154 | 2.787145 | 5.172205e-02 |
| ... | ... | ... | ... | ... |
| 5499 | 0.675748 | 0.822039 | 0.978253 | 1.251698e-06 |
| 5500 | 0.675747 | 0.822038 | 0.978269 | 1.370907e-06 |
| 5501 | 0.675746 | 0.822038 | 0.978279 | 1.192093e-06 |
| 5502 | 0.675745 | 0.822037 | 0.978296 | 1.072884e-06 |
| 5503 | 0.675744 | 0.822036 | 0.978304 | 8.344650e-07 |

我们很明显看到 jax 的矩阵分解优化速度比PyTorch要快很多。不仅收敛的epochs数少了一半，而且每一个epoch需要的训练时间也更少。

* 为什么收敛的epochs数少了呢？可以看到是因为 jax 的初始值就更好一些。尽管 [jax.random.normal](https://jax.readthedocs.io/en/latest/_autosummary/jax.random.normal.html) 与 [torch.randn](https://pytorch.org/docs/stable/generated/torch.randn.html) 确实是等价的，而且 随机种子都是42，但是jax得到的初始UV确实更好。
* 为什么 jax 比 PyTorch 快呢？我们的 PyTorch 已经是使用了良好的代码实践，使用了 PyTorch 2.0 的 torch.compile功能，并且在不需要求导的情况下使用了 torch.no\_grad()，然而仍然是打不过 jax。有人说这是因为 jax 支持 JIT，这确实没错，XLA技术具有强大的性能，但是 PyTorch 的 torch.compile 也未尝不利。参考[文章](https://zhuanlan.zhihu.com/p/660342914)，实际上问题的关键在于 JAX 强调 函数式编程和无状态设计，所有的函数都是纯函数，没有副作用，因此编译优化更简单，而 PyTorch 则是面向对象编程，有状态的设计，因而需要更多的编译优化。

为了确定文档中说的 jax.random.normal 与 torch.randn 等价，这里我们可以使用 KS 拟合优度假设检验，验证这两个方法得到的随机数是否来自于同一个分布。

from scipy.stats import kstest, norm

# 生成 PyTorch 随机数  
L.seed\_everything(42)  
torch\_samples = torch.randn(1000)  
# 生成 JAX 随机数  
jax\_samples = jax.random.normal(jrandom.PRNGKey(42), (1000,))  
res\_torch = kstest(torch\_samples, norm.cdf, alternative='two-sided')  
res\_jax = kstest(jax\_samples, norm.cdf, alternative='two-sided')  
res\_mutual = kstest(jax\_samples, torch\_samples, alternative='two-sided')  
if res\_torch.pvalue > 0.05:  
 print(f"无法拒绝零假设，所以我们得接受torch就是服从标准正态分布")  
if res\_jax.pvalue > 0.05:  
 print(f"无法拒绝零假设，所以我们得接受jax就是服从标准正态分布")  
if res\_jax.pvalue > 0.05:  
 print(f"无法拒绝零假设，所以我们得接受这两个分布是一样的。")  
res\_torch, res\_jax, res\_mutual

INFO: Seed set to 42

Sun 2024-12-01 22:41:01.445898

INFO Seed set to 42 d=709570;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/lightning/fabric/utilities/seed.py\seed.py;;\:d=776646;file:///home/ye\_canming/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/lightning/fabric/utilities/seed.py#57\57;;\

Sun 2024-12-01 22:41:02.598415

INFO 无法拒绝零假设，所以我们得接受torch就是服从标准正态分布 d=288389;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=256787;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

Sun 2024-12-01 22:41:02.617294

INFO 无法拒绝零假设，所以我们得接受jax就是服从标准正态分布 d=709570;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=776646;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

Sun 2024-12-01 22:41:02.634285

INFO 无法拒绝零假设，所以我们得接受这两个分布是一样的。 d=442417;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=33326;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

(  
 KstestResult(statistic=0.019189070906030226, pvalue=0.848033539038892, statistic\_location=-0.38583097, statistic\_sign=1),  
 KstestResult(statistic=0.034437055011013185, pvalue=0.18222928193437893, statistic\_location=-0.63853437, statistic\_sign=1),  
 KstestResult(statistic=0.041, pvalue=0.37012017606173, statistic\_location=-0.81669724, statistic\_sign=1)  
)

我们可以看到统计上来说确实是等价的，但是得到的结果在相同随机数种子下确实是不同的数值。而且由于 res\_torch 的 p值比 res\_jax 的更高，我们可以认为 torch 的随机结果反而更加接近标准正态分布。

从 jax random模块的[文档](https://jax.readthedocs.io/en/latest/jax.random.html)来看，jax采用的随机数生成算法是”threefry2x32”，设计的目的是为了更好的并行化和向量化。

根据[这篇学术论文](https://arxiv.org/abs/2401.17345), NumPy, TensorFlow, 和 PyTorch 三个框架使用的随机数生成算法分别是 Mersenne Twister, PCG, and Philox。因此，PyTorch生成随机数的算法与Jax有所不同，这也就导致了我们刚才都选择了42作为随机种子，但是得到的UV矩阵初值有所不同，jax的随机矩阵在矩阵分解这个问题上似乎更好。

#### 6.3.3 画出迭代过程中目标函数值和测试集上 RMSE 的变化

我们这里尝试使用 plotly 而非 matplotlib 库来绘制图片，因为使用html或者jupyter notebook查看本文档的时候，plotly 能展示交互性的图表。

pip install plotly

import pandas as pd  
import plotly.graph\_objects as go  
from plotly.subplots import make\_subplots  
from thu\_big\_data\_ml.help import plt, pio

pio.renderers.default = "notebook"

由于 目标函数值 （MSE）的量纲和 RMSE不同，我们可以使用高级的绘图技巧，在同一张图上使用不同的纵轴来绘制不同的曲线，以方便比较。参考[官方文档](https://plotly.com/python/multiple-axes/) 和 [中文翻译版本](https://daobook.github.io/plotly-book/tutorial/fundamentals/15_multiple-axes.html)。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L192)

draw\_metrics\_df

draw\_metrics\_df (df:pandas.core.frame.DataFrame, title:str='Metrics')

def draw\_metrics\_df(df:pd.DataFrame, title:str='Metrics'):  
 # 使用 plotly 创建图表  
 fig = go.Figure()  
 # 需要支持第二纵轴  
 fig = make\_subplots(specs=[[{"secondary\_y": True}]], figure=fig)  
  
 # 绘制 loss 曲线  
 fig.add\_trace(go.Scatter(x=df.index, y=df['loss'], name='Loss', mode='lines'), secondary\_y=False)  
 fig.add\_trace(go.Scatter(x=df.index[1:], y=df['delta\_loss'][1:], name='Delta Loss', mode='lines'), secondary\_y=False)  
  
 # 绘制 train\_rmse 和 test\_rmse 曲线  
 fig.add\_trace(go.Scatter(x=df.index, y=df['train\_rmse'], name='Train RMSE', mode='lines'), secondary\_y=True)  
 fig.add\_trace(go.Scatter(x=df.index, y=df['test\_rmse'], name='Test RMSE', mode='lines'), secondary\_y=True)  
   
 # 更新坐标轴  
 fig.update\_yaxes(title\_text="Loss", titlefont\_color="red", tickfont\_color="red", secondary\_y=False)  
 fig.update\_yaxes(title\_text="RMSE", titlefont\_color="blue", tickfont\_color="blue", secondary\_y=True)  
   
 # 标题  
 fig.update\_layout(title=dict(  
 text=title  
 ),)  
 return fig

fig = draw\_metrics\_df(metrics\_df, title="Metrics of PyTorch Based Matrix Factorization Training")  
fig

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): text/html

fig = draw\_metrics\_df(jmetrics\_df, title="Metrics of Jax Based Matrix Factorization Training")  
fig

Unable to display output for mime type(s): text/html

如图所示，Loss 和 Delta Loss 使用的是左侧的坐标轴，单位为评分的平方，而Train RMSE 和 Test RMSE 使用的是右侧的坐标轴，单位为评分。Delta Loss是用来决定收敛条件的，是Loss的差分的绝对值。

最终的PyTorch和Jax分别的测试集RMSE为

metrics[-1]['test\_rmse'], jmetrics[-1]['test\_rmse']

(0.94344562292099, 0.9783037304878235)

我们对上面的图进行一些分析：

1. Loss和Train RMSE 的趋势是一致的，因为他们之间是平方关系；他们随着训练时间的增加而减小，一开始减小的速度比较快，后面变化趋于平缓。
2. Delta Loss 也随着时间逐渐减小。
3. Test RMSE 一开始先是快速降低，随后缓慢降低，最后反而开始上升。这就说明到了后期出现了过拟合现象，测试损失反而在上升。可以使用 Early Stop等方法缓解此问题。
4. Jax 和 PyTorch 的曲线趋势相似，但是初值不相同，这个问题我们已经在上文中详细讨论，这里不再赘述。

### 6.4 3.b 矩阵分解参数调优

#### 6.4.1 科学调参的原则与方法

我在以前的作业当中详细描述了谷歌AI团队《深度学习调优指南》的思想，涉及到的概念包括目标元参数、冗余元参数和固定元参数，贝叶斯优化、演化计算、近似随机搜索，科学实验的控制变量法与调参实验设计中的探索与利用等。这里我们不再赘述，需要的话可以阅读[这次作业的文档](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/notebooks/coding_projects/P1_KNN/kd_tree.ipynb) 以及 谷歌[Deep Learning Tuning Playbook](https://github.com/google-research/tuning_playbook)原文。

简单总结来说，调参的本质是进行科学实验，通过控制变量法，探究在冗余和固定元参数（a.k.a. 无关变量）搜索到最优或者保持固定时，目标元参数（a.k.a. 目标超参数、自变量）对因变量（a.k.a. 评价指标）的影响。

而对于调参实验的结果，我们可以使用我们这门课当中学习的假设检验方法，来探究目标元参数的取值（比如选择自己提出的方法还是其他人的方法）之间导致因变量的差异是否具有统计显著性，以及哪一个元参数的取值最好。这个可以参考我们[上一次作业ANOVA的文档](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/notebooks/coding_projects/P1_ANOVA/anova.ipynb)，这里我们同样不再赘述。

#### 6.4.2 搜索空间定义

本次作业中，我们可以认为 k 的取值是 **目标元参数**，我们实验的目的就是探究 不同k的取值对于矩阵分解的效果有何影响。直观地来说，k越大，我们矩阵可以容纳的秩就越大，表达能力就更强，应当具有更好的分解效果，这是我们的实验假设。具体而言，k决定了模型参数的大小，而李航《统计学习方法》中介绍的泛化误差上界定理，[Open AI 对 LLM 探究的 Scaling Law](https://openai.com/index/scaling-laws-for-neural-language-models/) 都指出模型参数和模型的泛化误差具有重要的关系。我们通过本次调参实验可以尝试看看有无类似的规律。

而 以及 学习率 lr 的取值则是 **冗余元参数** 或者 **固定超参数**。由于作业要求我们搜索 ，所以它是 冗余元参数，我们需要为每一个 k 的取值搜素它的值调到最优，而为了节省计算资源，lr则可以当做固定超参数。为了提高实验速度，我们把required\_delta\_loss设的比上一节要大一些，并且使用 jax 来做分解。

为了定义搜索空间，我们使用dataclass，要求传入函数的参数是强类型，而且有一个随机概率分布，这样方便定义调参。这里用到我自己写的scholarly\_infrastructure库的一个核心功能experiment\_setting，对Python标准的dataclass进行了改进。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L225)

MatrixFactorizationSetting

MatrixFactorizationSetting (k:int=20, lmd:float=0.001, lr:float=0.005,  
 required\_delta\_loss:float=0.001)

from scholarly\_infrastructure.rv\_args.nucleus import RandomVariable, experiment\_setting  
from optuna.distributions import IntDistribution, FloatDistribution, CategoricalDistribution  
from typing import Optional, Union  
from dataclasses import asdict  
import optuna

@experiment\_setting  
class MatrixFactorizationSetting:  
 # 隐因子数量 k  
 k: int = ~RandomVariable( # ~ 表示 服从 这个分布，这是我写的库实现的语法糖  
 default=20,  
 description="Number of latent factors.",  
 # distribution=IntDistribution(low=1, high=128, log=True) # 我们假设 k 和 RMSE满足某种“scaling law”，所以使用 log 分布。  
 distribution=CategoricalDistribution([1, 4, 16, 64])   
 )  
   
 # 正则化参数 λ  
 lmd: float = ~RandomVariable(  
 default=0.001,  
 description="Regularization parameter.",  
 distribution=FloatDistribution(0.001, 0.1, log=True)  
 )  
   
 # 学习率  
 lr: float = ~RandomVariable(  
 default = 5e-3,   
 description="Learning rate.",  
 distribution=FloatDistribution(1e-4, 1e-1, log=True) # 虽然本次我们实验不调，但是其固有分布也是要写清楚的，下次实验或许要调。  
 )  
   
 # 精度要求  
 required\_delta\_loss: float = ~RandomVariable(  
 default = 1e-3,   
 description="Required Delta Loss, a.k.a. tolerance",  
 distribution=FloatDistribution(1e-8, 1e-1, log=True)  
 )

# 可以展示文档， MatrixFactorizationSetting 是一个 dataclass，也是一个 experiment\_setting  
MatrixFactorizationSetting.show\_dataframe\_doc()[:1]

|  | name | type | default | default\_factory | init | repr | hash | compare | metadata | kw\_only | description | distribution |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | k | <class 'int'> | 20.0 | <dataclasses.\_MISSING\_TYPE object at 0x7f42182... | True | True | None | True | None | <dataclasses.\_MISSING\_TYPE object at 0x7f42182... | Number of latent factors. | CategoricalDistribution(choices=(1, 4, 16, 64)) |

固定元参数定义

fixed\_meta\_params = MatrixFactorizationSetting(lr=5e-3, # surprise库的值  
 required\_delta\_loss=1e-4)  
frozen\_rvs = {"lr", "required\_delta\_loss"}

#### 6.4.3 定义目标函数

由于我们数据不是很多，我们直接就使用 测试集作为验证集，允许使用验证集来进行模型选择（包括调参、Early Stopping等）。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/tree.py#L163)

objective

objective (trial:optuna.trial.\_trial.Trial, critical\_metric='test\_rmse')

def objective(trial:optuna.Trial, critical\_metric="test\_rmse"):  
 # experiment\_setting 类具有 optuna\_suggest 方法，可以自动推荐一个设置出来。  
 config:MatrixFactorizationSetting = MatrixFactorizationSetting.optuna\_suggest(  
 trial, fixed\_meta\_params, frozen\_rvs=frozen\_rvs)  
 # 调用上面写好的 train\_matrix\_factorization\_jax 函数  
 jmodel, jmetrics = train\_matrix\_factorization\_jax(X\_train\_dense, X\_test\_dense,   
 trial=trial, critical\_metric=critical\_metric, # 为了进行搜索剪枝，传入这两个参数  
 \*\*asdict(config))  
 # 假如采取最后的  
 # best\_metric = jmetrics[-1][critical\_metric]  
 # 假如允许 Early Stop  
 best\_metric = min(map(lambda m: m[critical\_metric], jmetrics))  
 return best\_metric

#### 6.4.4 执行调参搜索

from thu\_big\_data\_ml.help import runs\_path  
from optuna.samplers import \*  
from optuna.pruners import \*  
import json

study\_path = runs\_path / "optuna\_studies.db"  
sqlite\_url = f"sqlite:///{study\_path}"

study = optuna.create\_study(  
 study\_name="matrix factorization hpo 11.27 4.0", # 3.0 使用 1e-3  
 storage=sqlite\_url,   
 load\_if\_exists=True,   
 # sampler=QMCSampler(seed=42), # 谷歌建议  
 sampler=TPESampler(seed=42), # 谷歌建议  
 pruner=HyperbandPruner(), # 通过中间结果来决定是否停止搜索  
 direction="minimize")  
study.set\_user\_attr("contributors", ["Ye Canming"])  
study.set\_user\_attr("fixed\_meta\_parameters", json.dumps(asdict(fixed\_meta\_params)))

[I 2024-12-01 22:41:19,109] Using an existing study with name 'matrix factorization hpo 11.27 4.0' instead of creating a new one.

接下来运行

study.optimize(objective, n\_trials=10)

0%| | 1/100000 [00:00<26:39:19, 1.04it/s, delta\_loss=0.0242, loss=5.33, test\_rmse=2.31, train\_rmse=2.31]  
[I 2024-12-01 22:41:26,704] Trial 60 pruned.   
 0%| | 1/100000 [00:01<28:29:17, 1.03s/it, delta\_loss=0.0674, loss=8.68, test\_rmse=2.95, train\_rmse=2.95]  
[I 2024-12-01 22:41:34,347] Trial 61 pruned.   
 0%| | 9/100000 [00:01<3:55:06, 7.09it/s, delta\_loss=0.0616, loss=8.17, test\_rmse=2.87, train\_rmse=2.86]   
[I 2024-12-01 22:41:38,169] Trial 62 pruned.   
 0%| | 1/100000 [00:00<24:50:02, 1.12it/s, delta\_loss=0.0644, loss=8.68, test\_rmse=2.95, train\_rmse=2.95]  
[I 2024-12-01 22:41:41,686] Trial 63 pruned.   
 0%| | 9/100000 [00:01<4:08:01, 6.72it/s, delta\_loss=0.0569, loss=8.22, test\_rmse=2.88, train\_rmse=2.87]   
[I 2024-12-01 22:41:45,774] Trial 64 pruned.   
 0%| | 1/100000 [00:00<17:05:00, 1.63it/s, delta\_loss=0.0259, loss=5.33, test\_rmse=2.31, train\_rmse=2.31]  
[I 2024-12-01 22:41:48,977] Trial 65 pruned.   
 0%| | 3/100000 [00:01<9:38:54, 2.88it/s, delta\_loss=0.00837, loss=4.07, test\_rmse=2.02, train\_rmse=2.02]   
[I 2024-12-01 22:41:53,200] Trial 66 pruned.   
 0%| | 1/100000 [00:01<49:53:26, 1.80s/it, delta\_loss=0.0253, loss=5.33, test\_rmse=2.31, train\_rmse=2.31]  
[I 2024-12-01 22:41:58,560] Trial 67 pruned.   
 0%| | 1/100000 [00:00<19:19:37, 1.44it/s, delta\_loss=0.0271, loss=5.33, test\_rmse=2.31, train\_rmse=2.31]  
[I 2024-12-01 22:42:03,995] Trial 68 pruned.   
 0%| | 1/100000 [00:01<31:06:24, 1.12s/it, delta\_loss=0.0255, loss=5.33, test\_rmse=2.31, train\_rmse=2.31]  
[I 2024-12-01 22:42:08,464] Trial 69 pruned.

study.optimize(objective, n\_trials=30)

study.optimize(objective, n\_trials=20)

#### 6.4.5 分析实验结果

首先加载调参实验的结果，实验分析的时候，不用重新再跑一次实验，这两个是分开的

study = optuna.load\_study(  
 study\_name="matrix factorization hpo 11.27 4.0",   
 storage=sqlite\_url)

##### 6.4.5.1 最好的参数组合是哪个？

study.best\_value, study.best\_params

(0.9159189462661743, {'k': 16, 'lmd': 0.07791470666172784})

##### 6.4.5.2 Optuna 可视化分析结果

可以通过下面的命令对我们刚才保存调参结果的数据库文件进行可视化展示：

optuna-dashboard sqlite:///optuna\_studies.db --port 18081

from optuna.visualization import plot\_intermediate\_values  
from optuna.visualization import plot\_param\_importances  
from optuna.visualization import plot\_optimization\_history

plot\_optimization\_history(study)

Unable to display output for mime type(s): text/html

我们进行了60次实验，随着实验的进行，目标函数越来越好。注意有很多实验因为被提前剪枝了所以没有在图中显示出来，这些实验的目标函数值太大了，不会优于显示在图中的实验。

plot\_intermediate\_values(study)

Unable to display output for mime type(s): text/html

图中我们可以看出，不同的实验的loss下降曲线是有所不同，有的曲线是向上凸的。

plot\_param\_importances(study)

Unable to display output for mime type(s): text/html

从这个图我们可以发现，相比于正则化项 lambda的取值，k的取值明显对泛化性能更加重要。

##### 6.4.5.3 以 k 为目标元参数进行假设检验分析——期望改进分析

期望改进分析是说如果有多个方法可以选择的时候，我提出的方法随机地应用于某一个人使用的元参数组合时，我的方法的期望性能相比于采纳其他方法，是否显著地更好。

# 首先我们拿到实验的所有数据  
df = study.trials\_dataframe()  
df.head()

|  | number | value | datetime\_start | datetime\_complete | duration | params\_k | params\_lmd | user\_attrs\_delta\_loss | user\_attrs\_loss | user\_attrs\_test\_rmse | user\_attrs\_train\_rmse | system\_attrs\_completed\_rung\_0 | system\_attrs\_completed\_rung\_1 | system\_attrs\_completed\_rung\_2 | system\_attrs\_completed\_rung\_3 | system\_attrs\_completed\_rung\_4 | system\_attrs\_completed\_rung\_5 | system\_attrs\_completed\_rung\_6 | state |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0.961816 | 2024-11-27 21:57:37.217204 | 2024-11-27 21:58:54.987898 | 0 days 00:01:17.770694 | 4 | 0.002051 | 0.000100 | 0.916142 | 0.961816 | 0.957153 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | COMPLETE |
| 1 | 1 | 0.919173 | 2024-11-27 21:58:55.112755 | 2024-11-27 22:01:19.734468 | 0 days 00:02:24.621713 | 16 | 0.026070 | 0.000100 | 0.805639 | 0.919173 | 0.897574 | 2.309793 | 2.299495 | 2.269680 | 2.190724 | 2.011784 | 1.097557 | 0.927841 | COMPLETE |
| 2 | 2 | 1.117243 | 2024-11-27 22:01:20.259350 | 2024-11-27 22:01:49.182465 | 0 days 00:00:28.923115 | 4 | 0.002310 | 0.007060 | 1.239060 | 1.117243 | 1.113131 | 2.021852 | 2.018061 | 2.007246 | 1.979886 | 1.928901 | 1.117243 | NaN | PRUNED |
| 3 | 3 | 0.920926 | 2024-11-27 22:01:49.277027 | 2024-11-27 22:04:11.261289 | 0 days 00:02:21.984262 | 16 | 0.003823 | 0.000100 | 0.809304 | 0.920926 | 0.899613 | 2.299917 | 2.270888 | 2.193796 | 2.017946 | 1.108612 | 0.929011 | NaN | COMPLETE |
| 4 | 4 | 1.533228 | 2024-11-27 22:04:17.347049 | 2024-11-27 22:04:40.633025 | 0 days 00:00:23.285976 | 1 | 0.008168 | 0.013748 | 2.349351 | 1.533228 | 1.532759 | 1.921702 | 1.920640 | 1.917642 | 1.910277 | 1.898259 | 1.533228 | NaN | PRUNED |

# 用于后续指定列进行分析  
target = "value" # 实际上是 test\_rmse  
treatment = "params\_k"

# 只需要分析表格中我们感兴趣的列  
intersted\_cols = [c for c in df.columns if c.startswith("params") or "user\_attrs" in c or c=="value"]  
dfi = df[intersted\_cols].dropna(subset=[target])  
dfi[treatment] = dfi[treatment].astype("int")  
dfi.head()

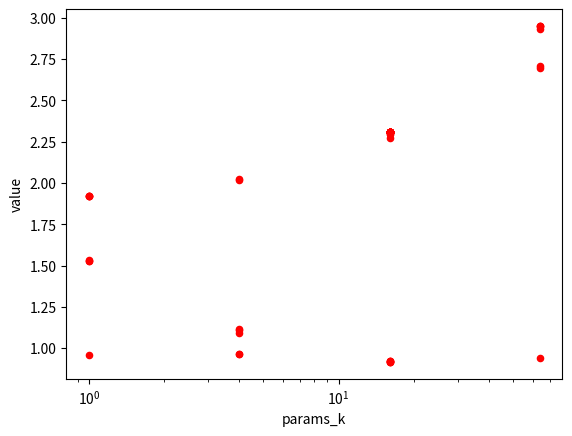
|  | value | params\_k | params\_lmd | user\_attrs\_delta\_loss | user\_attrs\_loss | user\_attrs\_test\_rmse | user\_attrs\_train\_rmse |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0.961816 | 4 | 0.002051 | 0.000100 | 0.916142 | 0.961816 | 0.957153 |
| 1 | 0.919173 | 16 | 0.026070 | 0.000100 | 0.805639 | 0.919173 | 0.897574 |
| 2 | 1.117243 | 4 | 0.002310 | 0.007060 | 1.239060 | 1.117243 | 1.113131 |
| 3 | 0.920926 | 16 | 0.003823 | 0.000100 | 0.809304 | 0.920926 | 0.899613 |
| 4 | 1.533228 | 1 | 0.008168 | 0.013748 | 2.349351 | 1.533228 | 1.532759 |

import seaborn as sns

fig, ax = plt.subplots()  
ax.set\_xscale('log')  
# sns.boxplot(data=dfi, x=treatment, y=target, ax=ax)  
dfi.plot(x=treatment, y=target, ax=ax, kind='scatter', c='red')

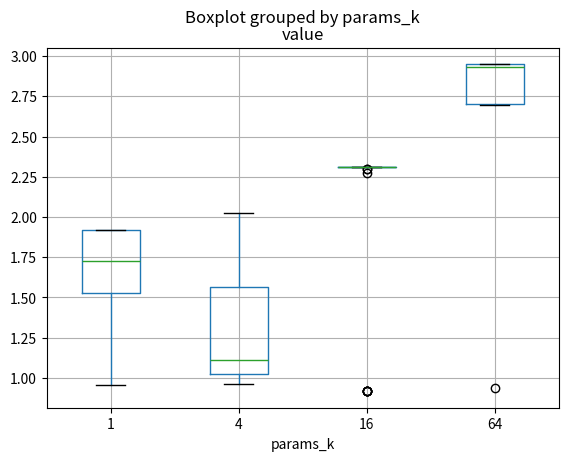
不同k的RMSE箱线图对比

<Axes: xlabel='params\_k', ylabel='value'>



dfi.boxplot(by=treatment, column=target)

<Axes: title={'center': 'value'}, xlabel='params\_k'>

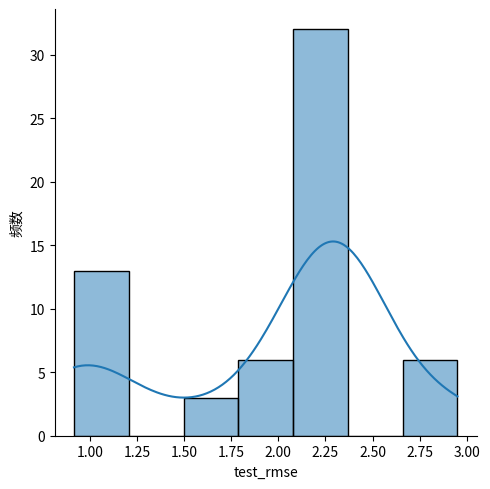


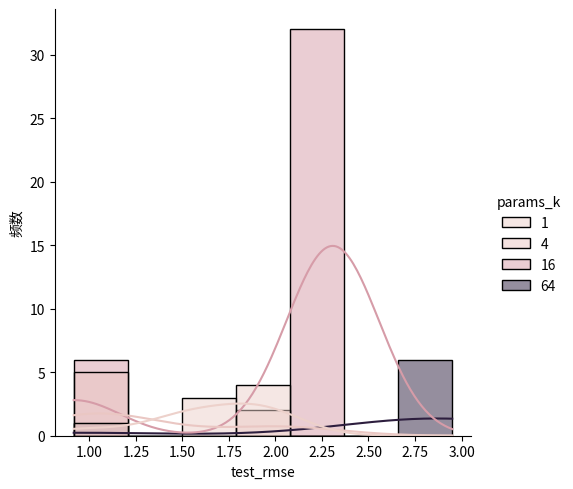
我们首先画出了散点图和箱线图，可以直观感觉到，k小的时候，test\_rmse的上界反而更小，而平均值则是在k=4时取最好。

现在我们来画经验分布图，包括没有分组前和分组之后的数据的分布图。

sns.displot(data=dfi, x=target, kde=True, kind="hist")  
plt.ylabel("频数")  
plt.xlabel("test\_rmse")  
sns.displot(data=dfi, x=target, hue=treatment, kde=True, kind="hist")  
plt.ylabel("频数")  
plt.xlabel("test\_rmse")

Text(0.5, 9.444444444444438, 'test\_rmse')





刚才进行了可视化分析，现在进行严格的假设检验。

我们在同时比较多个方法，可以用上一次作业的ANOVA检验吗？ 由于我们使用了nbdev系统，我上次写作业的代码已经变成了成熟的python模块，我们可以直接调用我之前写的代码，看看ANOVA的条件是否满足。 由于上次写的[test\_normality\_group](https://Open-Book-Studio.github.io/THU-Coursework-Machine-Learning-for-Big-Data/coding_projects/P1_ANOVA/anova.html#test_normality_group)需要大量数据才有效，我们需要换别的假设检验。

[source](https://github.com/Open-Book-Studio/THU-Coursework-Machine-Learning-for-Big-Data/blob/main/thu_big_data_ml/big_data_analytics/matrix_decomposition.py#L305)

test\_normality\_small\_sample

test\_normality\_small\_sample (df, interesting\_col, hue\_col='群类别',  
 transform=None)

from thu\_big\_data\_ml.big\_data\_analytics.anova import test\_normality\_group, homogeneity\_of\_variance  
from scipy import stats  
from statsmodels.stats.diagnostic import lilliefors  
from thu\_big\_data\_ml.big\_data\_analytics.anova import auto\_anova\_for\_df, auto\_kruskal\_for\_df

# 根据上次我们ANOVA作业调研的结果， Shapiro-Wilk 和 Lilliefors 是适合小样本情况下的正态检验方法。  
def test\_normality\_small\_sample(df, interesting\_col, hue\_col='群类别', transform=None):  
 if transform is None:  
 transform = lambda x: x  
 grouped\_data = df.groupby(hue\_col)  
 normality\_results = {}  
 for name, group in grouped\_data:  
 normality\_result = {}  
 data\_column = group[interesting\_col].to\_numpy()  
 data\_column = transform(data\_column)  
   
 # Shapiro-Wilk test  
 res = stats.shapiro(data\_column)  
 normality\_result['Shapiro-Wilk'] = "Not Normal" if res.pvalue < 0.05 else "Normal"  
   
 # Lilliefors test   
 res = lilliefors(data\_column)  
 normality\_result['Lilliefors'] = "Not Normal" if res[1] < 0.05 else "Normal"  
   
 normality\_results[name] = normality\_result  
 return pd.DataFrame(normality\_results)

test\_normality\_small\_sample(dfi, interesting\_col=target, hue\_col=treatment)

|  | 1 | 4 | 16 | 64 |
| --- | --- | --- | --- | --- |
| Shapiro-Wilk | Not Normal | Not Normal | Not Normal | Not Normal |
| Lilliefors | Normal | Not Normal | Not Normal | Not Normal |

homogeneity\_of\_variance(dfi, interesting\_col=target, hue\_col=treatment)

The variances are not homogeneous!

{  
 'ratio\_largest\_to\_smallest': 2.158185623471692,  
 'bartlett\_result': BartlettResult(statistic=3.6668498840499297, pvalue=0.29975821213270615)  
}

对于正态性，k=1的时候根据Lilliefors检验，是正态分布，其他情况下拒绝了是正态分布的假设。 对于方差齐性，根据拇指法则，方差不齐，但是根据假设检验，方法齐性满足。

那么我们可以勉强使用。

auto\_anova\_for\_df(dfi, interesting\_col=target, hue\_col=treatment)

|  | Source | Sum of Squares (SS) | Degrees of Freedom (df) | Mean Square (MS) | F | p | p\_excel |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Between | 6.865326 | 3 | 2.288442 | 8.42717 | 0.000103 | 0.000103 |
| 1 | Within | 15.207092 | 56 | 0.271555 | NaN | NaN | NaN |
| 2 | Total | 22.072418 | 59 | NaN | NaN | NaN | NaN |

尽管ANOVA的使用条件不满足，但是我们勉强使用ANOVA，可以看到ANOVA方法的p值很小，很显著地不同k之间存在显著差异！

接下来我们使用 Kruskal-Wallis 检验，这个检验不需要方差齐性和正态性满足，就可以对多列样本进行比较。

res = auto\_kruskal\_for\_df(dfi, interesting\_col=target, hue\_col=treatment)  
if res.pvalue < 0.05:  
 print(f"There is a significant difference between the different {treatment}.")  
res

Wed 2024-11-27 23:14:42.491233

INFO There is a significant difference between the different params\_k. d=665822;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py\nucleus.py;;\:d=179451;file:///home/ye\_canming/repos/novelties/cv/ScholarlyInfrastructure/scholarly\_infrastructure/logging/nucleus.py#53\53;;\

KruskalResult(statistic=21.605648342166887, pvalue=7.879126651465415e-05)

Kruskal-Wallis H-test tests的零假设是 不同组的中位数之间没有显著差异。看来我们可以拒绝原假设，认为不同k的矩阵分解的RMSE的中位数有显著差异。

刚才我们检验出来存在显著差异，但是不知道具体是谁比谁大有显著性，所以我们还需要使用post-hoc类型的假设检验来进行进一步的分析。

首先刚才检验的是中位数有显著差异，我们来求一下分组的中位数。

# 分组中位数  
medians = df.groupby(treatment)[target].median().sort\_values(ascending=True)  
order = medians.index[:] # 中位数的排序  
medians

params\_k  
4 1.111837  
1 1.726936  
16 2.309724  
64 2.931462  
Name: value, dtype: float64

Dunn’s test就是针对 Kruskal-Wallis one-way analysis 对应的 post-hoc 检验方法（之一，还有其他的）。

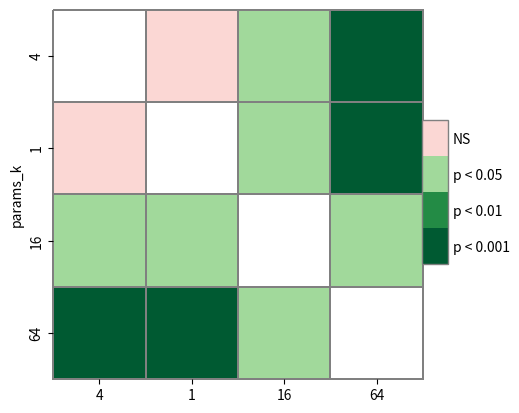
import scikit\_posthocs as sp  
from scikit\_posthocs import posthoc\_dunn

res = posthoc\_dunn(dfi, val\_col=target, group\_col=treatment,   
 sort=True, p\_adjust='holm')  
  
# 按照中位数大小排序  
res = res.reindex(order)  
res = res[order]  
res

|  | 4 | 1 | 16 | 64 |
| --- | --- | --- | --- | --- |
| params\_k |  |  |  |  |
| 4 | 1.000000 | 0.835664 | 0.033024 | 0.000609 |
| 1 | 0.835664 | 1.000000 | 0.035515 | 0.000703 |
| 16 | 0.033024 | 0.035515 | 1.000000 | 0.035515 |
| 64 | 0.000609 | 0.000703 | 0.035515 | 1.000000 |

# 对 post hoc检验的结果进行热力图可视化  
heatmap\_args = {'linewidths': 0.25, 'linecolor': '0.5', 'clip\_on': False, 'square': True, 'cbar\_ax\_bbox': [0.80, 0.35, 0.04, 0.3]}  
sp.sign\_plot(res, \*\*heatmap\_args)

(<Axes: ylabel='params\_k'>, <matplotlib.colorbar.Colorbar object at 0x7ef55c5e1b40>)



从结果表可以看出，k=4和k=1的矩阵分解方法都非常显著地优于k=16和k=64的方法，是最好的两个参数。而k=16又显著地由于k=64的情况。而k=4和k=1之间，由于实验数据不足，无法得出明确的结论。

## 7 4. 基于用户的协同过滤方法与矩阵分解算法的结果与优缺点对比

### 7.1 理论上的优缺点对比

[《深度学习推荐系统》](https://yd.qq.com/web/reader/b7732f20813ab7c33g015deakaab325601eaab3238922e53)一书中对这两个方法都有所介绍，其中提到一些观点，这里我阅读总结如下，其中括号部分是我针对本次项目情况的思考和疑惑

基于用户的协同过滤方法(UserCF)的优缺点

1. 优点：符合人们直觉上的“兴趣相似的朋友喜欢的物品，我也喜欢”的思想。因为是基于用户的相似度来进行推荐。书上认为这个方法非常直观、可解释性很强。（本次Netflix数据集就是假设其他和我对电影评分相似的朋友评价过的电影如果我去看也会有相似的评分）
2. 缺点：用户相似度矩阵的存储开销非常大。在实际的互联网应用中，用户数远远大于物品数（？一定是这样吗，根据[分析报告](https://pdf.dfcfw.com/pdf/H3_AP202407261638343337_1.pdf), 淘宝用户数量是10亿，而2000万商家的商品数量是数十亿，不好说谁比谁多），为了快速找到最相似的n个用户，需要存储用户相似度矩阵（？真的是这样吗，KD树、FAISS等数据结构可以很高效地找到k近邻），因为用户数数以亿计，所以这个相似度矩阵很大，随着用户的增长以的速度快速增长，系统难以承受。（本次数据集中，物品是电影，数量和用户一样多，相似度矩阵确实很大）
3. 缺点：新用户推荐问题。用户的历史数据向量是稀疏的，而对于一个只有几次购买记录的新用户来说，很难找到和他相似的用户。不只是新用户，如果正反馈获取是很困难的应用场景，比如酒店预订和大件商品预订，所有用户都很稀疏，互相之间的相似度都很低。
4. 优点：推荐具有社交属性，可以让用户快速得知兴趣相似的人喜欢什么，比较适合新闻推荐场景。因为不同人的新闻兴趣点比较分散，所以新闻的及时性和热点性比用户对新闻的兴趣偏好更重要（？所以这不就说明UserCF没有用吗，为什么王喆这里说userCF能发现和追踪热点呢）。
5. 缺点：对于兴趣变化稳定的应用不太适合。比如电商场景中，某段时间内用户会倾向于某一类热点商品，这个时候商品向量的相似度比用户向量的相似度是更加有价值的。（本次数据集NetFlix属于这个场景，因为看电影的评分是相对固定的偏好）
6. 缺点：泛化能力较差，因为无法把两个用户或者物品相似信息推广到其他用户和其他物品的相似上，导致新用户或者冷门物品太过稀疏，很少被推荐。
7. 缺点：只能利用共现矩阵也就是用户和物品的评分交互信息，无法有效引入用户年龄、性别，商品类别、描述等额外的信息，相比逻辑回归等机器学习模型而言使用的信息太少，所以泛化能力有限。（本次项目当中，Netflix数据集就包括了电影的标题信息，如果进行自然语言处理提取特征，可以提升推荐效果）

矩阵分解技术的优缺点

1. 优点：泛化能力相对协同过滤更好。使用稠密的隐向量来表示用户和物品，所以可以挖掘到用户的隐含兴趣和物品的潜在特征，可以弥补协同过滤处理稀疏矩阵能力不足的问题。
2. 缺点：和协同过滤的缺点7一样，也只是利用了共现矩阵的信息，没有利用到其他信息。
3. 缺点：王喆指出，k越大，隐向量的表达能力越强，但是容易过拟合，所以泛化能力会相应降低。所以工程中需要多次实验才能找到最优的k值。
4. 优点：相比协同过滤，空间复杂度低，因为不需要存储mxm的相似度矩阵，只需要存储用户和物品的隐向量。（本次项目中 MatrixFactorization model 的参数量确实是比协同过滤的小）
5. 优点：更好的扩展性和灵活性。矩阵分解的最终产出是用户和物品的隐向量矩阵，与深度学习中的Embedding思想不谋而合（真的一样吗），因此矩阵分解的结果也非常便于与其它特征进行组合和拼接，并便于与深度学习网络进行无缝结合。

### 7.2 实验结果比较

我们可以来看看实验结果是否支持上面的理论分析。

协同过滤方法我们得到的训练集RMSE是1.0127, 测试集RMSE是1.0184。

对于给定 k=50, λ=0.01 的情况下，使用lr=5e-3, required\_delta\_loss=1e-6，Pytorch和 jax 在训练集上的RMSE是分别是6.760268e-01和8.220364e-01, 测试集上的RMSE分别是 9.441460e-01和9.783037e-01，

在没有进行调参的情况下，矩阵分解的性能轻松超越了协同过滤方法，在训练集和测试集上都更小，**可见矩阵分解的表达能力和泛化能力都优于协同过滤，确实符合理论分析中所说，使用稠密的隐向量表示用户和物品是可以挖掘到用户的隐含兴趣和物品的潜在特征的，弥补了协同过滤能力不足的问题**。

在实现协同过滤的时候，我们确实进行了多次优化，才成功计算出最后的推荐值，而且整个系统中最慢的步骤和存储量最大的步骤就是求解相似度矩阵，即使使用了sklearn的方法进行了稀疏矩阵相似度的优化，还是耗费了十多分钟的时间，**相比矩阵分解实验所需要的时间长，存储量也大，符合理论分析中协同过滤的第二点和矩阵分解的第四点**。

metrics\_df.iloc[-1], jmetrics\_df.iloc[-1]

(  
 loss 2.285061e-01  
train\_rmse 6.760268e-01  
test\_rmse 9.441460e-01  
delta\_loss 9.992896e-07  
Name: 8898, dtype: float64,  
 loss 6.757439e-01  
train\_rmse 8.220364e-01  
test\_rmse 9.783037e-01  
delta\_loss 8.344650e-07  
Name: 5503, dtype: float64  
)

而在lr=5e-3, required\_delta\_loss=1e-4条件下，搜索k和λ我们发现，λ对于RMSE的影响不大，而k的影响却很大。我们通过严格的假设检验分析，发现了k=1和k=4的矩阵分解方法显著优于k=16和k=64的矩阵分解方法，而k=16的矩阵分解方法又显著地优于k=64的矩阵分解方法。从平均值来看，k=4的RMSE比k=1的RMSE要小，但是缺乏统计学证据，差异并不显著。

由此**可以验证矩阵分解理论分析第三点**，即 k越大，隐向量的表达能力越强，但是容易过拟合，所以泛化能力会相应降低。所以k=4在我们这次实验中是折中的最优最优选择。可见，在实际工程应用中，需要调解k这个元参数是矩阵分解算法的一个缺点。我们使用optuna调参也消耗了大量的时间，即使使用了最先进的Tree Prazen优化方法和Hyperband剪枝方法。

刚才我们做的调参分析是期望改进分析，是说参数随机选取的情况下，哪个k期望的性能最好。如果参数不是随机选取，而是通过调参优化到最优呢？这也是一个公平比较各个方法的另外一个分析手段。我们列出下表：

# 其他无关变量调到最优时，目标自变量对指标的影响。  
best\_rows = dfi.loc[dfi.groupby(treatment)[target].idxmin()]  
best\_rows[['params\_k', 'user\_attrs\_test\_rmse', 'user\_attrs\_train\_rmse', 'user\_attrs\_delta\_loss']]

|  | params\_k | user\_attrs\_test\_rmse | user\_attrs\_train\_rmse | user\_attrs\_delta\_loss |
| --- | --- | --- | --- | --- |
| 7 | 1 | 0.958201 | 0.956666 | 0.0001 |
| 0 | 4 | 0.961816 | 0.957153 | 0.0001 |
| 15 | 16 | 0.915919 | 0.893912 | 0.0001 |
| 6 | 64 | 0.937775 | 0.851572 | 0.0001 |

如上表所示，随着k的增大，train\_rmse 会变小，所以说k越大，模型的表达能力越强。而对于test\_rmse，在最优参数的情况下，k=16是最小的，而不是k=64最小。因而k=16在调参到最优的情况下，即具有k=1和k=4不具备的表达能力，也可以避免像k=64那样过拟合。

综上所述，我们的实验结果基本支持书上给出的分析，我们通过本次实验深入了解了两种方法的实现细节和性能差异，受益匪浅。