

# 大数据计算及应用

## 作业 2：推荐系统

---

### 实验报告

---

June 15, 2020

曹元议 1711425

王雨奇 1711299

段非 1711264

# Contents

<b>1</b>	<b>概述</b>	<b>1</b>
1.1	推荐系统产生背景	1
1.2	推荐系统常用算法简介	1
1.2.1	协同过滤	1
1.2.2	基于 NLP 处理	2
1.2.3	基于关联规则	2
1.2.4	基于矩阵分解	2
1.2.5	基于用户行为数据的模型推荐算法	2
1.3	SVD 算法简介	2
1.3.1	为什么使用 SVD 算法	4
1.3.2	引入物品属性的优化	4
1.4	推荐系统评测指标	4
<b>2</b>	<b>数据集说明</b>	<b>5</b>
<b>3</b>	<b>代码细节说明</b>	<b>9</b>
3.1	算法整体流程	10
3.2	SVD_improved 类的初始化	10
3.3	训练集和测试集的划分	12
3.4	数据集统计	14
3.5	训练基础 SVD 模型	21
3.6	线性回归拟合物品属性	25
3.7	测试训练过的模型	30
3.8	预测打分结果	32
<b>4</b>	<b>实验结果分析</b>	<b>33</b>
4.1	遇到的问题及总结	40

# 1 概述

## 1.1 推荐系统产生背景

随着移动互联网的快速发展，我们进入了信息爆炸时代。当前通过互联网提供服务的平台越来越多，相应的提供的服务种类 (购物，视频，新闻，音乐，婚恋，社交等) 层出不穷。

同时，随着社会的发展，受教育程度的提升，每个人都有表现自我个性的欲望。随着互联网的发展，出现了非常多的可以表达自我个性的产品，如微信朋友圈，微博，抖音，快手等，每个人的个性喜好特长有了极大展示的空间。“长尾理论”也很好的解释了多样化物品中的非畅销品可以满足人们多样化的需求，这些需求加起来不一定比热门物品产生的销售额小。

并且随着社会的进步，物质生活条件的改善，大家不必再为生存下来而担忧，所以大家有越来越多的需求是非生存需求，比如看书，看电影，购物等，而这些非生存的需求往往在很多时候是不确定的，比如给你推荐一部电影，如果符合你的口味，你可能会很喜欢。

总结上面提到的三点，当今时代可选择的商品和服务这么多，而不同人的兴趣偏好又是截然不同，并且在特定场景下，个人对自己的需求不是很明确。在这三个背景驱动下，推荐系统应运而生。个性化推荐系统是解决上述三个矛盾的最有效的方法和工具之一。

## 1.2 推荐系统常用算法简介

### 1.2.1 协同过滤

协同过滤 (Collaborative Filtering, 简称 CF) 是利用集体智慧的一个典型的方法，集体智慧指的是从大量人的行为数据中收集，从大多数推个例，常常指的是大量行为中的共性部分。来看一下来自百度百科的解释：协同过滤简单来说是利用某兴趣相投、拥有共同经验之群体的喜好来推荐用户感兴趣的信息，个人通过合作的机制给予信息相当程度的回应（如评分）并记录下来以达到过滤的目的进而帮助别人筛选信息，回应不一定局限于特别感兴趣的，特别不感兴趣信息的纪录也相当重要。协同过滤分为两种：基于用户的 CF 和基于物品的 CF。

- 基于用户的 CF (User CF): 基于用户对商品的偏好找到相邻邻居用户，然后将邻居用户喜欢的推荐给当前用户。就是将用户和商品之间建立一个向量的关系，以此来挖掘用户之间的相似度，找到 K 邻居后，根据邻居的西安四度权重和对商品的偏好，预测当前用户没有偏好的商品，得到一个排序的商品列表作为推荐。

- 基于物品的 CF 的原理和 User CF 类似，Item CF 是将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有偏好的物品，计算得到一个排序的物品列表作为推荐。

### 1.2.2 基于 NLP 处理

由于公司的数据很多都不是结构化的，所以就用到了 NLP 算法，算法主要涉及了像文本相似度（COS 相似度；Jaccard 相似度）、文本距离（欧式距离；曼哈顿距离；闵科夫斯基距离，马氏距离）等为基础的基本推荐算法。

### 1.2.3 基于关联规则

关联规则的本质也是一种协同过滤，有很多应用场景，最经典的莫过于啤酒和尿布（基于购物车，收藏栏，转发统计，点赞栏 Apriori 算法，FP-Tree，RefixSpan 算法，SimRank 算法）

### 1.2.4 基于矩阵分解

基于打分矩阵情况下的推荐。常用算法：FunkSVD 算法, BiasSVD 算法。

### 1.2.5 基于用户行为数据的模型推荐算法

对于淘宝，京东这些电商平台，用户会有大量的行为数据留下，这部分数据中很多是结构化的数据，这时就可以用模型算法来进行推荐。像简单判别模型推荐 SVM、LR、KNN 等，以及复杂判别模型推荐算法 FM、FFM、wide&deep、deepFM、NFM、AFM 等。

## 1.3 SVD 算法简介

SVD 本身是将矩阵进行奇异值分解的办法，但是在推荐系统中也可以使用类似的方法。

用户的评分行为可以表示成一个评分矩阵  $R$ ，其中  $R_{ui}$  就是用户  $u$  对物品  $i$  的评分。但是，用户不会对所有的物品评分，所以这个矩阵里有很多元素都是空的，这些空的元素称为缺失值（missing value）。SVD 在推荐系统中的应用思路就是把评分问题转为矩阵分解问题，即将评分矩阵  $R$  分解为  $Q^T$  和  $P$ ， $Q^T$  和  $P$  的乘积就是评分矩阵  $R'$ ， $R'$  中对应的元素的值就是评分的预测值。 $Q$  和  $P$  矩阵的意义稍后再说明。

由于用户对物品的打分矩阵中的元素并没有填满，因此就需要通过机器学习的手段去学习到打分矩阵被分解之后得到的矩阵。它在推荐系统中的原理也属于 LFM（latent factor model）隐语义模型的原理。

SVD 法的预测公式如下所示：

$$\widehat{r_{ui}} = \mu + b_i + b_u + q_i^T p_u$$

以上公式的参数解释如下所示：

- $\mu$ : 训练集中所有记录的评分的全局平均数。在不同网站中，因为网站定位和销售的物品不同，网站的整体评分分布也会显示出一些差异。比如有些网站中的用户就是喜欢打高分，而另一些网站的用户就是喜欢打低分。而全局平均数可以表示网站本身对用户评分的影响。
- $b_u$ : 用户偏置（user bias）项。这一项表示了用户的评分习惯中和物品没有关系的那种因素。比如有些用户就是比较苛刻，对什么东西要求都很高，那么他的评分就会偏低，而有些用户比较宽容，对什么东西都觉得不错，那么他的评分就会偏高。
- $b_i$ : 物品偏置（item bias）项。这一项表示了物品接受的评分中和用户没有什么关系的因素。比如有些物品本身质量就很高，因此获得的评分相对都比较高，而有些物品本身质量很差，因此获得的评分相对都会比较低。
- $p$ : 用户因子矩阵。例如，给定一个用户  $u$ ，向量  $p_u$  的维度值为因子的个数， $p_u$  的每一个分量就是因子的大小（越大则表明用户对这个因子的偏好程度更高）
- $q$ : 物品因子矩阵。例如，给定一个物品  $i$ ，向量  $q_i$  的维度值为因子的个数， $q_i$  的每一个分量就是因子的大小（越大则表明这个物品拥有这个因子的程度更深）

其中  $q_i^T p_u$  是向量  $q_i$  的转置与向量  $p_u$  的点积。

为了学习模型的  $b_u$ 、 $b_i$ 、 $p$ 、 $q$  参数，我们需要让以下正则化的平方误差最小：

$$\sum_{(u,i) \in \alpha} (r_{ui} - \mu - b_i - b_u - q_i^T p_u)^2 + \lambda_4 (b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$$

其中  $\lambda_4$  是正则化参数，防止过拟合，其值一般通过交叉验证来获得。

这个最小化过程一般可以使用随机梯度下降法（SGD）去实现，这里就不做推导了，直接给出递推式如下：

$$b_u \leftarrow b_u + \gamma \cdot (e_{ui} - \lambda_4 \cdot b_u)$$

$$b_i \leftarrow b_i + \gamma \cdot (e_{ui} - \lambda_4 \cdot b_i)$$

$$q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda_4 \cdot q_i)$$

$$p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda_4 \cdot p_u)$$

其中  $e_{ui}$  为真实值与预测值的差值，即： $e_{ui} = r_{ui} - \widehat{r}_{ui}$ 。 $\gamma$  为学习率。

按照上述式子进行递推计算就可以得到  $b_u$ 、 $b_i$ 、 $p$ 、 $q$  参数了。一般来说 Netflix 推荐的  $\lambda_4$  和  $\gamma$  的值分别为： $\lambda_4=0.02$ 、 $\gamma=0.005$

### 1.3.1 为什么使用 SVD 算法

经过课上的学习可以发现，UserCF 预测的精度最低、ItemCF 次之，因此不考虑使用 CF 来实现。而 SVD 算法不仅推荐的精度比前两者更高，而且还可以把推荐为题转换为机器学习问题，而且实现起来也比较容易。不采用精度更高的 SVD++ 算法的原因是，SVD++ 算法虽然考虑了用户的隐式反馈信息，但是这会大大增加计算量，而且 SVD++ 算法相比 SVD 算法改进程度也不是太大。因此，我们采用了 SVD 算法。

### 1.3.2 引入物品属性的优化

本次实验我们采用线性拟合的方式，使用 `itemAttribute.txt` 提供的信息，使用物品属性去拟合真实值与预测值的差值  $e_{ui}$ ，公式如下：

$$\widehat{e}_{ui} = a \times attr_1 + b \times attr_2 + c$$

其中系数  $a$ 、 $b$ 、 $c$  通过最小二乘法求得。这里我们引入 Python 的 `scipy` 包的 `optimize` 模块，为每个用户去计算  $a$ 、 $b$ 、 $c$  参数。

这样，引入物品属性优化之后的评分预测公式为：

$$\widehat{r}_{ui} = \mu + b_i + b_u + q_i^T p_u + \widehat{e}_{ui}$$

## 1.4 推荐系统评测指标

我们在实验中使用的评测指标是 RMSE，即均方根误差。公式如下所示：

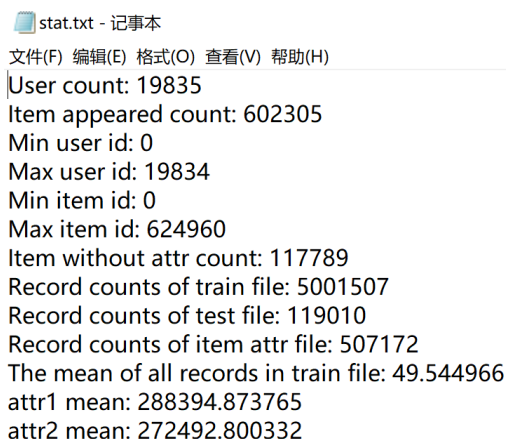
$$\text{RMSE} = \frac{\sqrt{\sum_{(u,i) \in T} (r_{ui} - \widehat{r}_{ui})^2}}{|T|}$$

其中， $r_{ui}$  分数的真实值， $\widehat{r}_{ui}$  是分数的预测值，集合  $T$  是测试集， $|T|$  是测试集的大小。

评分预测的目的就是找到好的模型，从而最小化测试集的 RMSE。

## 2 数据集说明

以下是我们在程序运行时生成的统计文件`stat.txt`的内容：



```
stat.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
User count: 19835
Item appeared count: 602305
Min user id: 0
Max user id: 19834
Min item id: 0
Max item id: 624960
Item without attr count: 117789
Record counts of train file: 5001507
Record counts of test file: 119010
Record counts of item attr file: 507172
The mean of all records in train file: 49.544966
attr1 mean: 288394.873765
attr2 mean: 272492.800332
```

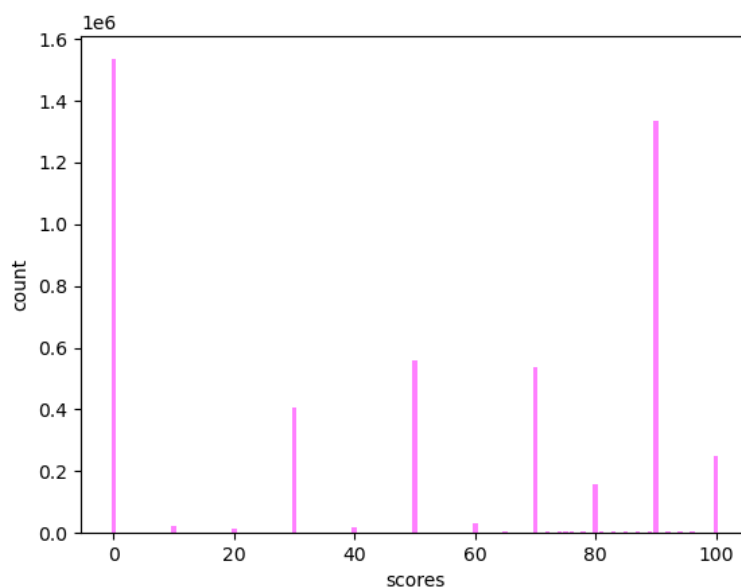
**Figure 1:** `stat.txt`

可以发现：`train.txt`、`test.txt`、`itemAttribute.txt`的记录数分别为 5000107（训练集评分总数）、119010、507172。原始数据集的用户数为 19835，在`train.txt`、`test.txt`、`itemAttribute.txt`出现过的物品总数为 602305。最小的用户 id 为 0，最大用户 id 为 1；最小的物品 id 为 0，最大物品 id 为 624960（所以物品数应该是 624961 个）。`train.txt`所有得分的平均值为 49.544966。`itemAttribute.txt`中属性 1 的平均值约为 288394，属性 2 的平均值约为 272492（选择的计算方式是将`None`处理为 0）。

从理论上来说，稠密矩阵大小应该是  $19835 \times 602305 = 12396101435$ 。而训练集评分总数只有 5000107 个（约占 0.04%）。由此可见这次的评分矩阵是非常稀疏的。

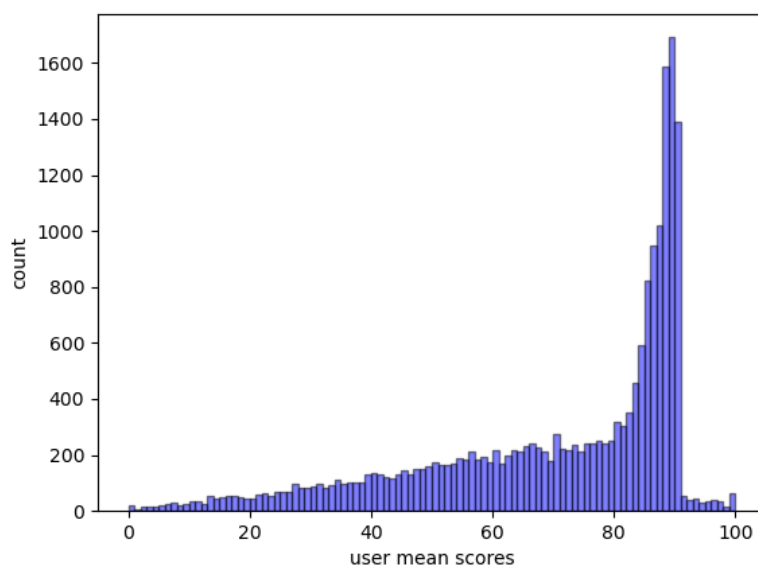
除此之外，我们还生成了每个用户的平均值文件`userMean.csv`、物品得分平均值文件`itemMean.csv`、打分分布情况文件`rateDetail.csv`。由于比较占空间所以未在压缩包中提供，可以运行之后查看。

以下是通过`rateDetail.csv`得出的打分分布情况，可以发现用户的评分都集中在 0、30、50、70、80、90、100：



**Figure 2:** 打分分布情况

以下是根据`userMean.csv`得出的用户打分平均值分布，可以发现平均分集中在 80 到 90。意味着用户的打分总体来说还是比较宽容的：

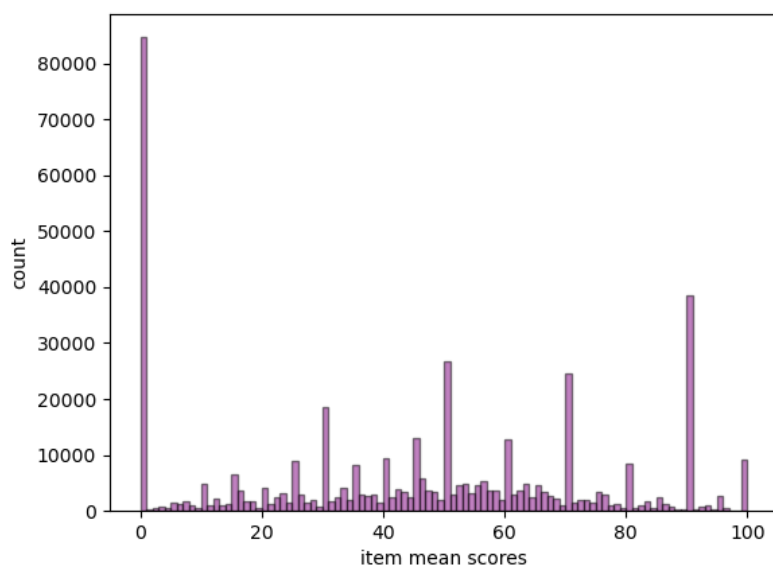


**Figure 3:** 用户均分分布情况

以下是根据`itemMean.csv`得出的物品得分平均值分布，可以发现平均分分布和上

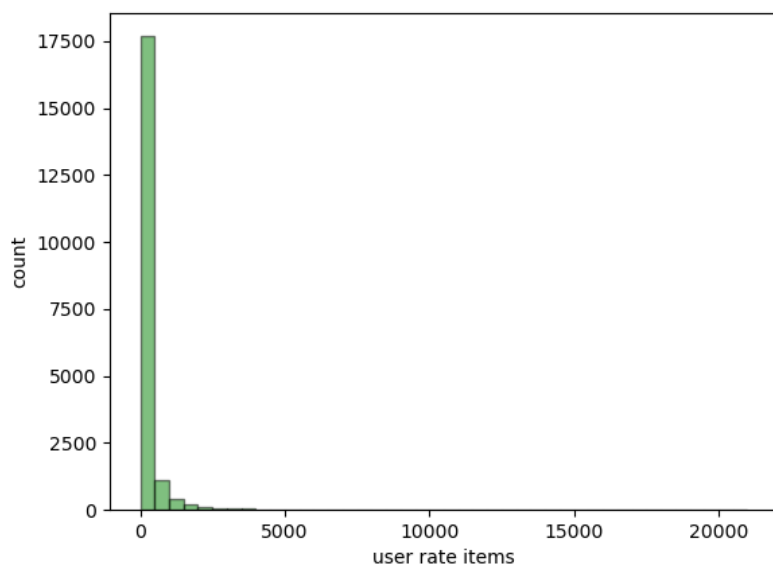


面的总体打分分布情况比较类似：

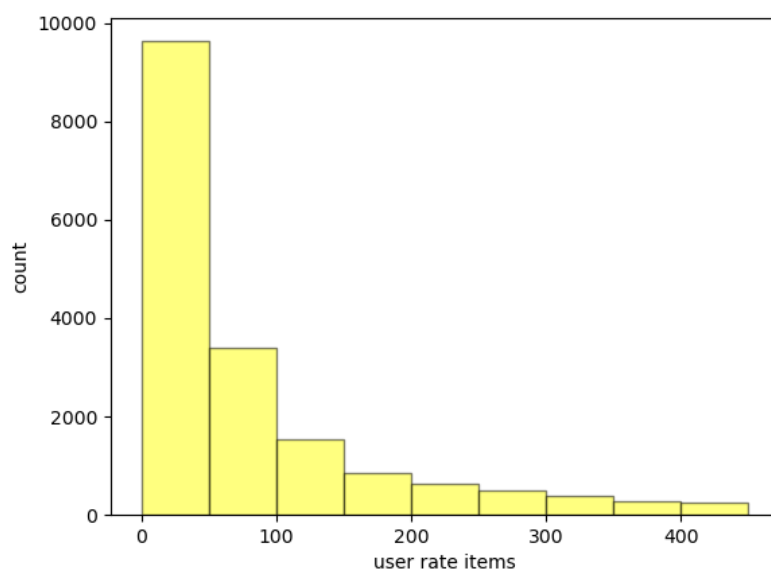


**Figure 4:** 物品均分分布情况

以下是根据`userMean.csv`得出的用户打分次数分布，可以发现打分次数集中在 50 以内，说明用户总体打分的意愿还是不够强：

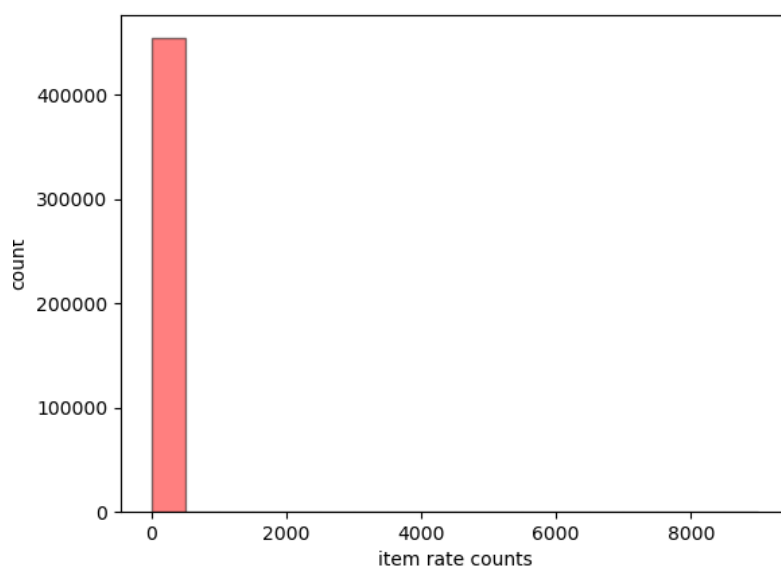


**Figure 5:** 用户打分次数分布情况（一个柱子的刻度为 500）

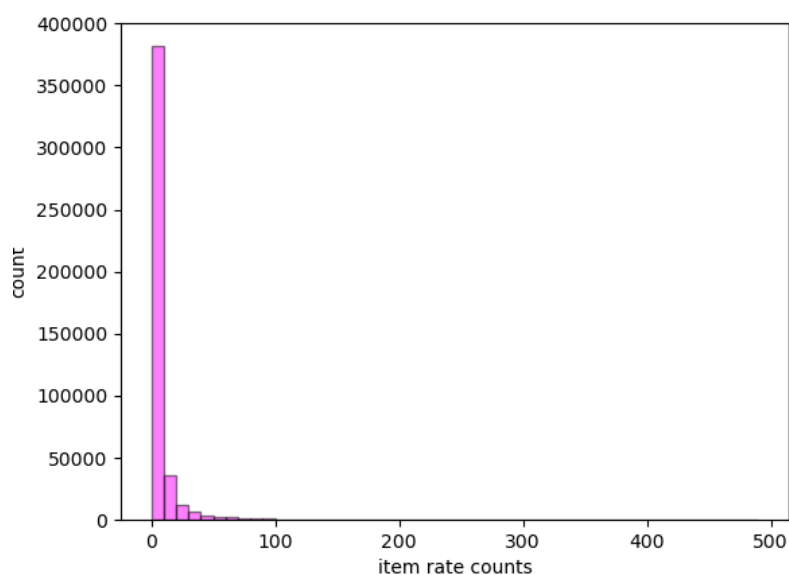


**Figure 6:** 用户打分次数在 500 以内的分布情况（一个柱子的刻度为 50）

以下是根据 `itemMean.csv` 得出的物品被评分打分次数分布，可以发现大多数物品被评分次数仅仅在 10 以内，也可以侧面地反映用户总体打分的意愿确实不够强，以及很多物品并没有被用户所发现：



**Figure 7:** 物品被评分次数分布情况（一个柱子的刻度为 500）



**Figure 8:** 物品被评分次数在 500 以内的分布情况（一个柱子的刻度为 10）

### 3 代码细节说明

本次实验的所有代码使用 Python 语言编写。下面是本次实验的源码结构：

```
RecommendationSystem/  
├── algorithm  
│   └── SVD_improved.py  
├── data-new  
│   ├── train.txt  
│   ├── test.txt  
│   └── itemAttribute.txt  
└── recommand.py
```

`algorithm`文件夹包含了本次实验必须运行的算法代码`SVD_improved.py`（由主程序`recommand.py`进行调用）。`data-new`文件夹包含了实验用到的数据集源文件。

### 3.1 算法整体流程

我们将算法的所有功能都封装到了类 `SVD_improved` 中，在源文件 `algorithm/SVD_improved.py` 中。

`recommand.py` 是本次实验的主程序。

程序的大致流程为：统计源数据文件并做部分初始化、划分训练集和测试集、训练基础 SVD 模型、线性回归拟合物品属性、测试训练过的模型、预测打分结果。

下面将分别介绍这几部分代码。

### 3.2 SVD\_improved 类的初始化

这里实现的主要是对 `SVD_improved` 类的成员变量进行初始化，包括算法运行中的一些参数。下面对一些变量的含义进行了说明：

- `_n_user`: 用户总人数；
- `_n_item`: 物品总个数；
- `_train_mean`: 训练集中所有评分的全局平均数。
- `_learn_rate`: 学习率，即梯度下降中的步长，经过查阅相关资料，我们将它设置为 Netflix 提供的推荐值 0.005。
- `_lambda`: 正则化参数，为了防止过拟合，我们将它设置为 Netflix 提供的推荐值 0.02。
- `_n_factors`: 隐式因子数，即物品的特征数，这里将它设置为 10。
- `_n_epochs`: 训练时的迭代次数，这里我们将它设置为 20。
- `_scale`: 用户可以给物品打分的范围，这里 “(0,100)” 代表着用户的打分范围是 0-100。
- `_user_dict`: 存储实际的用户 id 与该用户在程序中的编号的映射。
- `_item_dict`: 存储实际的物品 id 与该物品在程序中的编号的映射。
- `_sparse_matrix`: 以 “用户 id、物品 id、分值” 三元组的形式存储用户对某个物品的打分。
- `_sparse_matrix_with_err`: 在 `_sparse_matrix` 的基础上又加了最后一列：分数预测值与分数真实值的差值。

- `_bu`: 用户偏置 (user bias) 项。
- `_bi`: 物品偏置 (item bias) 项
- `_p`: 用户隐因子矩阵。
- `_q`: 物品隐因子矩阵。

下面是SVD\_improved类的构造函数:

**algorithm/SVD\_improved.py**

```
12 def __init__(self, n_factors=10, n_epochs=20, learn_rate=0.005,  
13 _lambda=0.02, scale=(0,100)):  
14     super().__init__()  
15     self._TRAIN_FILE = '../data-new/train.txt'  
16     self._TEST_FILE = '../data-new/test.txt'  
17     self._ATTR_FILE = '../data-new/itemAttribute.txt'  
18     self._TRAIN_SET = 'trainset.csv'  
19     self._TEST_SET = 'testset.csv'  
20     self._BU_VECTOR = 'bu_vector.dat'  
21     self._BI_VECTOR = 'bi_vector.dat'  
22     self._P_MATRIX = 'p_matrix.dat'  
23     self._Q_MATRIX = 'q_matrix.dat'  
24     self._SPRASE = 'sprase.dat'  
25     self._all_time = 0.0  
26     self._do_divide = False # 是否要重新训练  
27     self._do_train = False # 是否要重新划分训练集和测试集  
28     self._n_user = 0  
29     self._n_item = 0  
30     self._train_mean = 0.0  
31     self._learn_rate = learn_rate  
32     self._lambda = _lambda  
33     self._n_factors = n_factors  
34     self._n_epochs = n_epochs  
35     self._scale = scale  
36     self._user_dict = {}  
37     self._item_dict = {}  
38     self._sprase_matrix = []  
39     self._sprase_matrix_with_err = []  
40     self._bu = None  
41     self._bi = None  
42     self._p = None  
43     self._q = None
```

```
43     self._item_attrs = {}
44     self._user_param = {}
45     self._user_item_attrs = defaultdict(list) # format: { user1:
        [(item1_id, item1.err, item1.attr1, item1.attr2), ...], ... }
```

### 3.3 训练集和测试集的划分

`divide_train_and_test_set()`函数会将`train.txt`文件分割成训练集、测试集，并把它分别写到`trainset.csv`和`testset.csv`中。其中，训练集用来训练我们的SVD模型，即学习参数`bu`、`bi`、`p`、`q`，测试集用来测试模型的准确度，并由此计算出RMSE。代码如下：

```
algorithm/SVD_improved.py

368 def divide_train_and_test_set(self, test=2):
369     assert test >= 0 and test <= 10
370     print('Dividing train and test set.....')
371     start = timeit.default_timer()
372     train_file = []
373     with open(self._TRAIN_FILE, 'r') as f:
374         train_file = f.readlines()
375         f.close()
376
377     user_id = ''
378     item_id = ''
379     rate_str = ''
380     trainset = open(self._TRAIN_SET, 'w')
381     testset = open(self._TEST_SET, 'w')
382     rank = test / 10
383
384     for train in train_file:
385         line = train.strip()
386         if line.find('|') != -1:
387             user_id, user_item_count = line.split('|')
388         else:
389             if line == "":
390                 continue
391             item_id, rate_str = line.split()
392             if (random.random() >= rank):
```

```
393         trainset.write('%s,%s,%s\n' % (user_id, item_id,
394                                     rate_str))
395     else:
396         testset.write('%s,%s,%s\n' % (user_id, item_id, rate_str))
397
398     trainset.close()
399     testset.close()
400
401     end = timeit.default_timer()
402     self._all_time += (end - start)
403     print('Time cost: %fs' % (end - start))
```

划分训练集和测试集的方法采用的就是最简单的方法,即根据随机数的值来划分。函数的参数 `test` 表示测试集所占总数据的比重,例如这里 `test=2` 就说明测试集的大小占 `train.txt` 中数据的  $2/10 = 0.2 = 20\%$ 。对于 `train.txt` 中的数据,程序会通过随机数 `random()` 的使用来把它们随机分到训练集、测试集之中,存储形式是“用户实际 id、物品实际 id、分值”的三元组。

例如,对于 `train.txt` 的前 11 行数据。即用户 0 对十个商品的打分。

```
0|41
507696 90
137915 90
22757 90
120328 90
123025 90
131263 90
147073 80
175835 90
180037 90
192496 90
```

Figure 9: `train.txt` 的前 11 行数据

程序会把它们分到 `trainset.csv` 和 `testset.csv` 中 (每次重新划分的结果都不一样,下面只是一次划分的结果)。

0	137915	90			
0	120328	90			
0	131263	90			
0	147073	80			
0	180037	90			
0	192496	90			
			0	507696	90
			0	22757	90
			0	123025	90
			0	175835	90

(a) trainset.csv

(b) testset.csv

Figure 10: 用户 0 对前十个商品的打分被分到 trainset.csv 和 testset.csv

### 3.4 数据集统计

`stat()` 函数主要用来统计 `train.txt`、`test.txt` 和 `itemAttribute.txt` 中的数据信息，并做一些初始化操作（例如填充映射 `self._user_dict` 和 `self._item_dict`）。首先初始化一些用于辅助统计的变量。

```

algorithm/SVD_improved.py
146 def stat(self):
147     self._all_time = 0.0
148     print('In %s ..... ' % (self.__class__.__name__ + '.' +
149         sys.getframe().f_code.co_name + '()'))
149     start = timeit.default_timer()
150     attr_lines = []
151     train_lines = []
152     test_lines = []
153     with open(self._ATTR_FILE, 'r') as f:
154         attr_lines = f.readlines()
155         f.close()
156     with open(self._TRAIN_FILE, 'r') as f:
157         train_lines = f.readlines()
158         f.close()
159     with open(self._TEST_FILE, 'r') as f:
160         test_lines = f.readlines()
161         f.close()
162
163     train_records = 0 # 训练集有多少条数据，初始化为 0
164     test_records = 0 # 测试集有多少条数据，初始化为 0
165     user_rates = {} # format: { userid1: [sco1, sco2, ...], ...}
166     item_rates = {} # format: { itemid1: [sco1, sco2, ...], ...}
167     rate_num = [0 for _ in range(0, self._scale[1] - self._scale[0] + 1)]
168     user_id = ''

```



```

169     item_id = ''
170     rate_str = ''
171     uid = 0
172     iid = 0
173     rate_ = 0.0
174     global_mean = 0.0 # 训练集中所有评分的全局平均数
175     min_user_id = sys.maxsize # 用户 id 最小值
176     max_user_id = 0 # 用户 id 最大值
177     min_item_id = sys.maxsize # 物品 id 最小值
178     max_item_id = 0 # 物品 id 最大值
179     item_has_attr = {}
180     attr_lines_ = []

```

然后处理itemAttribute.txt文件，统计物品的属性特征。

#### algorithm/SVD\_improved.py

```

183     print("Processing item attr file.....")
184     # 将属性值 None 先处理为 0 (原数据中没有 0 的属性值)
185     for attr_line in attr_lines:
186         line = attr_line.strip()
187         if line == "":
188             continue
189         item_id, attr1, attr2 = line.split('|')
190         if attr1 == 'None':
191             attr1 = 0
192         else:
193             attr1 = int(attr1)
194         if attr2 == 'None':
195             attr2 = 0
196         else:
197             attr2 = int(attr2)
198         iid = int(item_id)
199         # 更新 max_item_id
200         if max(iid, max_item_id) == iid:
201             max_item_id = iid
202         # 更新 min_item_id
203         if min(iid, min_item_id) == iid:
204             min_item_id = iid
205         # 建立实际的物品 id 与物品在数据集中的编号的映射

```

```

206         self._item_dict[iid] = self._n_item
207         # 将物品及其属性存储到 attr_lines_ 中
208         item_has_attr[iid] = (iid, attr1, attr2)
209         attr_lines_.append((iid, attr1, attr2))
210         self._n_item += 1
211         attr_records += 1
212
213     # 统计所有属性的平均值
214     attr1_sum = 0.0
215     attr2_sum = 0.0
216     for item_id, attr1, attr2 in attr_lines_:
217         attr1_sum += attr1
218         attr2_sum += attr2
219
220     attr1_sum /= len(attr_lines_)
221     attr2_sum /= len(attr_lines_)
222
223     # 将缺失属性信息的 item 使用平均值处理
224     attr_list = []
225     item_no_attrs = 0
226     for i in range(min_item_id, max_item_id + 1):
227         try:
228             attr_list.append([item_has_attr[i][1], item_has_attr[i][2]])
229         except KeyError:
230             attr_list.append([attr1_sum, attr2_sum])
231             item_no_attrs += 1

```

接下来是统计train.txt、test.txt两个文件，其中统计了用户数，物品数，最大、最小用户 id，最大、最小物品 id，两个文件中的评分数，以及train.txt中全部评分的平均数。

#### algorithm/SVD\_improved.py

```

229     print("Processing train file.....")
230     for train in train_file:
231         line = train.strip()
232         if line.find('|') != -1:
233             user_id, user_item_count = line.split('|')
234             uid = int(user_id)
235             try:

```

```
236         u = self.user_dict[uid]
237     except KeyError:
238         user_rates[uid] = []
239         # 建立实际的用户 id 与该用户在数据集中的编号的映射
240         self.user_dict[uid] = self.n_user
241         # 更新 max_user_id
242         if max(uid, max_user_id) == uid:
243             max_user_id = uid
244         # 更新 min_user_id
245         if min(uid, min_user_id) == uid:
246             min_user_id = uid
247         self.n_user += 1
248     else:
249         if line == "":
250             continue
251         item_id, rate_str = line.split()
252         train_records += 1
253         iid = int(item_id)
254         rate_ = float(rate_str)
255         # 统计所有分值的分布情况
256         rate_num[math.floor(rate_)] += 1
257         global_mean += rate_
258         # 将用户的打分加到 user_rates 中, 其目的是在之后计算用户打分平均值
259         user_rates[uid].append(rate_)
260     try:
261         i = self._item_dict[iid]
262     except KeyError:
263         # 建立实际的物品 id 与该物品在数据集中的编号的映射
264         self._item_dict[iid] = self._n_item
265         # 更新 max_item_id
266         if max(iid, max_item_id) == iid:
267             max_item_id = iid
268         # 更新 min_item_id
269         if min(iid, min_item_id) == iid:
270             min_item_id = iid
271         self._n_item += 1
272     try:
273         item_rates[iid].append(rate_)
274     except KeyError:
275         item_rates[iid] = []
```

```
276         # 将物品获得的分数加到 item_rates 中, 其目的是在之后计算物品获取
277         # 的分数平均值
278         item_rates[iid].append(rate_)
279     # 计算打分的平均值
280     global_mean /= train_records
281
282     print("Processing test file.....")
283     for test in test_file:
284         line = test.strip()
285         if line.find('|') != -1:
286             user_id, user_item_count = line.split('|')
287             uid = int(user_id)
288             try:
289                 u = self.user_dict[uid]
290             except KeyError:
291                 user_rates[uid] = []
292                 # 建立实际的物品 id 与该物品在数据集中的编号的映射
293                 self.user_dict[uid] = self.n_user
294                 # 更新 max_user_id
295                 if max(uid, max_user_id) == uid:
296                     max_user_id = uid
297                 # 更新 min_user_id
298                 if min(uid, min_user_id) == uid:
299                     min_user_id = uid
300                 self.n_user += 1
301             else:
302                 if line == "":
303                     continue
304                 test_records += 1
305                 iid = int(line)
306                 try:
307                     i = self.item_dict[iid]
308                 except KeyError:
309                     self.item_dict[iid] = self.n_item
310                     if max(iid, max_item_id) == iid:
311                         max_item_id = iid
312                     if min(iid, min_item_id) == iid:
313                         min_item_id = iid
314                     self.n_item += 1
```

然后将重新整理的属性信息并输出到itemAttribute.csv文件（第一列是物品实际 id，第二列是属性 1 的值，第三列是属性 2 的值），然后打印数据的统计信息。

#### algorithm/SVD\_improved.py

```

304     print("Generating new item attr file.....")
305     with open('itemAttribute.csv', 'w') as f:
306         for i in range(len(attr_list)):
307             f.write('%d,%d,%d\n' % (i, attr_list[i][0], attr_list[i][1]))
308         f.close()
309
310     # 打印统计信息
311     print("Making statistics files.....")
312     with open('stat.txt', 'w') as statf:
313         statf.write('User count: %d\n' % self._n_user)
314         statf.write('Item appeared count: %d\n' % self._n_item)
315         statf.write('Min user id: %d\n' % min_user_id)
316         statf.write('Max user id: %d\n' % max_user_id)
317         statf.write('Min item id: %d\n' % min_item_id)
318         statf.write('Max item id: %d\n' % max_item_id)
319         statf.write('Item without attr count: %d\n' % item_no_attrs)
320         statf.write('Record counts of train file: %d\n' % train_records)
321         statf.write('Record counts of test file: %d\n' % test_records)
322         statf.write('Record counts of item attr file: %d\n' %
323                     attr_records)
324         statf.write('The mean of all records in train file: %f\n' %
325                     global_mean)
326         statf.write('attr1 mean: %f\n' % attr1_sum)
327         statf.write('attr2 mean: %f\n' % attr2_sum)
328         statf.close()

```

将用户打分的平均值写入userMean.csv文件，形式是用户实际 id、打分平均值、打分总数。

#### algorithm/SVD\_improved.py

```

327     with open('userMean.csv', 'w') as userMean:
328         s = sorted(user_rates.items(), key = lambda x : x[0])
329         for u, r in s:
330             userMean.write(str(u))
331             userMean.write(',')
332             sum = 0.0
333             for rate in r:

```

```

334         sum += rate
335         # 计算打分平均值
336         if len(r) != 0:
337             sum /= len(r)
338         userMean.write(str(sum))
339         userMean.write(',')
340         userMean.write(str(len(r)))
341         userMean.write('\n')
342     userMean.close()

```

结果如下图所示, 例如第一行就表示用户 0 的打分平均值是 77.31707, 他一共对 41 个物品进行了打分。

	A	B	C
1	0	77.31707	41
2	1	89.869	229
3	2	51.38614	101

Figure 11: userMean.csv 文件

将物品获得的打分的平均值写入itemMean.csv文件, 形式是物品实际 id、打分平均值、获得的打分总数。

```

algorithm/SVD_improved.py
343     with open('itemMean.csv', 'w') as itemMean:
344         s = sorted(item_rates.items(), key = lambda x : x[0])
345         for i, r in s:
346             itemMean.write(str(i))
347             itemMean.write(',')
348             sum = 0.0
349             for rate in r:
350                 sum += rate
351             if len(r) != 0:
352                 sum /= len(r)
353             itemMean.write(str(sum))
354             itemMean.write(',')
355             itemMean.write(str(len(r)))
356             itemMean.write('\n')
357     itemMean.close()

```

结果如下图所示, 例如第一行就表示物品 0 的获得的用户打分平均值是 0, 共有 1 个用户对该物品进行了打分。

	A	B	C
1	0	50	1
2	1	0	1
3	2	0	1

Figure 12: itemMean.csv 文件

最后, 将所有分数的分布情况写入文件 `rateDetail.csv`, 然后给出该函数 (即统计和初始化过程) 所用的时间。

```

algorithm/SVD_improved.py
359     with open('rateDetail.csv', 'w') as rateDetail:
360         for i, j in enumerate(rate_num):
361             rateDetail.write('%d,%d\n' % (i + self._scale[0], j))
362         rateDetail.close()
363
364     end = timeit.default_timer()
365     self._all_time += (end - start)
366     print('Time cost in %s: %fs' % (self.__class__.__name__ + '.' +
        sys._getframe().f_code.co_name + '()', end - start))

```

### 3.5 训练基础 SVD 模型

首先通过 `_prepare()` 函数做训练模型之前的初始化准备工作。`bu`、`bi` 在一开始只要初始化成全 0 向量, 初始化 `P`、`Q` 矩阵的方法很多, 一般都是将这两个矩阵用随机数填充, 但随机数的大小还是有讲究的, 根据经验, 随机数需要和  $1/\sqrt{F}$  成正比 (来自项亮的《推荐系统实践》第 189 页), `F` 即隐式因子数, 在代码中表示为 `self._n_factors`。

```

algorithm/SVD_improved.py
404     def _prepare(self):
405         print('In %s ..... ' % (self.__class__.__name__ + '.' +
            sys._getframe().f_code.co_name + '()'))
406         start = timeit.default_timer()
407         train = []
408         with open(self._TRAIN_SET, 'r') as f:
409             train = f.readlines()
410             f.close()

```

```

411
412     sqrt_dim = math.sqrt(self._n_factors)
413
414     self._bu = [0.0 for _ in range(0, self._n_user)]
415     self._bi = [0.0 for _ in range(0, self._n_item)]
416     self._p = [[random.random() / sqrt_dim for _ in range(0,
417 self._n_factors)] for _ in range(0, self._n_user)]
418     self._q = [[random.random() / sqrt_dim for _ in range(0,
419 self._n_factors)] for _ in range(0, self._n_item)]
420
421     self.records = 0
422
423     for train_line in train:
424         line = train_line.strip()
425         if line == "":
426             continue
427         user_id, item_id, rate_str = line.split(',')
428         uid = self._user_dict[int(user_id)]
429         iid = self._item_dict[int(item_id)]
430         rate_ = float(rate_str)
431         self._sprase_matrix.append((uid, iid, rate_))
432         self._train_mean += rate_
433         self.records += 1
434
435     self._train_mean /= self.records
436
437     end = timeit.default_timer()
438     self._all_time += (end - start)
439     print('Time cost in %s: %fs' % (self.__class__.__name__ + '.' +
440 sys._getframe().f_code.co_name + '()', end - start))

```

然后读入训练集`trainset.csv`的数据到稀疏矩阵`self._sprase_matrix`中。

然后通过`train()`函数完成迭代训练基础 SVD 模型的功能，主要是利用随机梯度下降法（SGD）不断地修正参数  $b_u$ 、 $b_i$ 、 $p_u$ 、 $q_i$ ，并计算每次迭代后得到的训练集 RMSE。



**algorithm/SVD\_improved.py**

```

445     def train(self):
446         print('Preparing.....')
447         # 调用 prepare() 初始化 bu,bi,p,q
448         self._prepare()
449         print('Training.....')
450
451         # 开始迭代训练模型
452         for epoch in range(0, self._n_epochs):
453             rmse = 0.0
454             start = timeit.default_timer()
455             self._sprase_matrix_with_err = []
456             for u, i, r in self._sprase_matrix:
457                 # 计算预测的分值
458                 rp = self._train_mean + self._bu[u] + self._bi[i] +
459                     self._dot(u, i)
460                 # 计算完分数之后一定要作边界检查
461                 if (rp < self._scale[0]):
462                     rp = float(self._scale[0])
463                 if (rp > self._scale[1]):
464                     rp = float(self._scale[1])
465                 # 计算真实值与预测值之间的差
466                 err = r - rp
467                 # 修正参数 bu
468                 self._bu[u] += self._learn_rate * (err - self._lambda *
469                     self._bu[u])
470                 # 修正参数 bi
471                 self._bi[i] += self._learn_rate * (err - self._lambda *
472                     self._bi[i])
473                 for k in range(0, self._n_factors):
474                     # trick 1: 每次迭代计算矩阵之前需要保存原值, 否则越迭代值越大,
475                     # 最终导致超过浮点数范围
476                     qik = self._q[i][k]
477                     puk = self._p[u][k]
478                     # 修正参数 qik
479                     self._p[u][k] += self._learn_rate * (err * qik -
480                         self._lambda * puk)
481                     # 修正参数 puk
482                     self._q[i][k] += self._learn_rate * (err * puk -
483                         self._lambda * qik)

```

```

478         rmse += err ** 2
479         self._sprase_matrix_with_err.append((u, i, r, err))
480     # 更新 rmse
481     rmse /= self.records
482     rmse = math.sqrt(rmse)
483     # trick 2: 每次迭代之后需要降低学习率, 以让结果尽快收敛
484     self._learn_rate *= 0.5
485     end = timeit.default_timer()
486     print('RMSE in epoch %d: %f' % (epoch, rmse))
487     self._all_time += (end - start)
488     print('Time cost: %fs' % (end - start))

```

其中`_dot()`函数用于计算P矩阵和Q矩阵的某一行某一点的点积:

#### **algorithm/SVD\_improved.py**

```

439     def _dot(self, u, i):
440         sum = 0.0
441         for k in range(0, self._n_factors):
442             sum += (self._p[u][k] * self._q[i][k])
443         return sum

```

迭代训练完成后, 将数据保存至文件中, 这样节省了每次运行都要训练模型的开销。如果已经存在训练结果, 就可以直接读取矩阵, 预测打分结果。

具体需要保存的内容为`self._bu`、`self._bi`、`self._p`、`self._q`这 4 个矩阵或向量的大小和内容, 以及`self._sprase_matrix_with_err`和`self._train_mean`。

#### **algorithm/SVD\_improved.py**

```

481     print('Iteration finished')
482     print('Writing train result data to files.....')
483     start = timeit.default_timer()
484     with open(self._BU_VECTOR, 'wb') as f:
485         f.write(struct.pack('i', self._n_user))
486         for bu in self._bu:
487             f.write(struct.pack('d', bu))
488         f.close()
489     with open(self._BI_VECTOR, 'wb') as f:
490         f.write(struct.pack('i', self._n_item))
491         for bi in self._bi:

```

```

492         f.write(struct.pack('d', bi))
493     f.close()
494     with open(self._P_MATRIX, 'wb') as f:
495         f.write(struct.pack('i', self._n_user))
496         f.write(struct.pack('i', self._n_factors))
497         for p in self._p:
498             for pi in p:
499                 f.write(struct.pack('d', pi))
500     f.close()
501     with open(self._Q_MATRIX, 'wb') as f:
502         f.write(struct.pack('i', self._n_item))
503         f.write(struct.pack('i', self._n_factors))
504         for q in self._q:
505             for qi in q:
506                 f.write(struct.pack('d', qi))
507     f.close()
508     with open(self._SPRASE, 'wb') as f:
509         f.write(struct.pack('d', self._train_mean))
510         f.write(struct.pack('i', self.records))
511         for u, i, r, err in self._sprase_matrix_with_err:
512             f.write(struct.pack('i', u))
513             f.write(struct.pack('i', i))
514             f.write(struct.pack('d', r))
515             f.write(struct.pack('d', err))
516     f.close()
517     end = timeit.default_timer()
518     self._all_time += (end - start)
519     print('Time cost: %fs' % (end - start))

```

### 3.6 线性回归拟合物品属性

训练完模型之后,就可以用它来测试划分出的训练集,以验证模型的准确度,并得出 RMSE。在这一步中,我们使用了 `itemAttribute.txt` 文件,即物品的属性来降低 RMSE,具体方法是使用每个物品的两个属性来二元线性拟合打分预测结果。这一部分通过 `_prepare_item_attr()`、`_linear()` 和 `linear()` 函数来实现,下面分别介绍。

`_prepare_item_attr()` 函数主要是将经过处理并重新生成的 `itemAttribute.csv` 的数据加载到 `_item_attrs` 和 `_user_item_attrs` 中。前者以物品 id 为索引,存储了每个物品的两个属性;后者以用户 id 为索引,每一项的内容是物品 id、真实值和预测值之

间的差值、物品第一个属性、物品第二个属性。

```

algorithm/SVD_improved.py

522 def _prepare_item_attr(self):
523     item_lines = []
524     with open('itemAttribute.csv', 'r') as f:
525         item_lines = f.readlines()
526         f.close()
527
528     for item_line in item_lines:
529         line = item_line.strip()
530         if line == "":
531             continue
532         item_id, attr1, attr2 = line.split(',')
533         try:
534             # 存储了每个物品的两个属性
535             self._item_attrs[self._item_dict[int(item_id)]] =
                    (float(attr1), float(attr2))
536         except KeyError:
537             pass
538
539     for u, i, r, err in self._sprase_matrix_with_err:
540         # 物品 id、残差、物品第一个属性、物品第二个属性
541         self._user_item_attrs[u].append((i, err, self._item_attrs[i][0],
                    self._item_attrs[i][1]))

```

接下来 `_linear()` 函数的功能是构造残差函数，使用最小二乘法拟合打分真实值与预测值之间的残差，公式是  $err = a \times attr_1 + b \times attr_2 + c$ ，最后返回  $a$ ， $b$ ， $c$  三个参数。

```

algorithm/SVD_improved.py

541 def _linear(self, user, user_item_attr):
542     # equation: err = a*attr1 + b*attr2 + c
543     # 为每个用户分析出方程参数
544     def func(x, y, p): # 回归函数
545         a, b, c = p
546         return a * x + b * y + c
547     def residuals(p, z, x, y): # 残差函数
548         return z - func(x, y, p)
549     l = len(self._user_item_attrs[user])
550     # 测试集中用户 user 打分的所有物品的第一个属性

```

```

551     x = np.array([self._user_item_attrs[user][i][2] for i in range(0,
552         1)])
553     # 测试集中用户 user 打分的所有物品的第二属性
554     y = np.array([self._user_item_attrs[user][i][3] for i in range(0,
555         1)])
556     # 物品真实值与预测值之间的残差
557     z = np.array([self._user_item_attrs[user][i][1] for i in range(0,
558         1)])
559     plsq = optimize.leastsq(residuals, [0, 0, 0], args=(z, x, y)) # 最小
560     # 二乘法拟合
561     a, b, c = plsq[0] # 获得拟合结果
562     return (a, b, c)

```

`linear()`函数实现了二元线性拟合的部分的整体功能，其中调用了上面提到的`_prepare_item_attr()`、`_linear()`两个函数。

在这里，程序顺便对是否重新划分训练集和测试集、是否要重新训练也作了判断。如果需要就分别调用`divide_train_and_test_set()`和`train()`函数，如果不需要就直接读取训练矩阵。

#### algorithm/SVD\_improved.py

```

558     def linear(self):
559         # 是否需要重新分割训练集和测试集
560         if self._do_divide:
561             self.divide_train_and_test_set()
562         elif os.path.exists(self._TRAIN_SET) == False or
563             os.path.exists(self._TEST_SET) == False:
564             self.divide_train_and_test_set()
565
566         # 是否需要重新训练模型
567         if self._do_train:
568             self.train()
569         elif os.path.exists(self._BI_VECTOR) == False or
570             os.path.exists(self._BU_VECTOR) == False or
571             os.path.exists(self._P_MATRIX) == False or
572             os.path.exists(self._Q_MATRIX) == False or
573             os.path.exists(self._SPRASE) == False:
574             self.train()
575         else: # 如果有训练结果文件，就可以直接读取训练矩阵了
576             print('Loading train data from file.....')

```

```
572     start = timeit.default_timer()
573     self._bi = []
574     self._bu = []
575     self._p = []
576     self._q = []
577     with open(self._BU_VECTOR, 'rb') as f:
578         byte_str = f.read(4)
579         user_len = struct.unpack('i', byte_str)[0]
580         for i in range(0, user_len):
581             byte_str = f.read(8)
582             l = struct.unpack('d', byte_str)[0]
583             self._bu.append(l)
584     f.close()
585     with open(self._BI_VECTOR, 'rb') as f:
586         byte_str = f.read(4)
587         item_len = struct.unpack('i', byte_str)[0]
588         for i in range(0, item_len):
589             byte_str = f.read(8)
590             l = struct.unpack('d', byte_str)[0]
591             self._bi.append(l)
592     f.close()
593     with open(self._P_MATRIX, 'rb') as f:
594         byte_str = f.read(4)
595         user_len = struct.unpack('i', byte_str)[0]
596         byte_str = f.read(4)
597         factor_len = struct.unpack('i', byte_str)[0]
598         for i in range(0, user_len):
599             new_list = []
600             for j in range(0, factor_len):
601                 byte_str = f.read(8)
602                 l = struct.unpack('d', byte_str)[0]
603                 new_list.append(l)
604             self._p.append(new_list)
605     f.close()
606     with open(self._Q_MATRIX, 'rb') as f:
607         byte_str = f.read(4)
608         item_len = struct.unpack('i', byte_str)[0]
609         byte_str = f.read(4)
610         factor_len = struct.unpack('i', byte_str)[0]
611         for i in range(0, item_len):
```

```

612         new_list = []
613         for j in range(0, factor_len):
614             byte_str = f.read(8)
615             l = struct.unpack('d', byte_str)[0]
616             new_list.append(l)
617         self._q.append(new_list)
618     f.close()
619     with open(self._SPRASE, 'rb') as f:
620         byte_str = f.read(8)
621         self._train_mean = struct.unpack('d', byte_str)[0]
622         byte_str = f.read(4)
623         self.records = struct.unpack('i', byte_str)[0]
624         for ii in range(0, self.records):
625             byte_str = f.read(4)
626             u = struct.unpack('i', byte_str)[0]
627             byte_str = f.read(4)
628             i = struct.unpack('i', byte_str)[0]
629             byte_str = f.read(8)
630             r = struct.unpack('d', byte_str)[0]
631             byte_str = f.read(8)
632             err = struct.unpack('d', byte_str)[0]
633             self._sprase_matrix_with_err.append((u, i, r, err))
634     f.close()
635     end = timeit.default_timer()
636     self._all_time += (end - start)
637     print('Time cost: %fs' % (end - start))

```

然后调用 `_prepare_item_attr()` 函数, 将 `itemAttribute.csv` 的数据加载到 `_item_attrs` 和 `_user_item_attrs` 中:

**algorithm/SVD\_improved.py**

```

637     print('Loading item attrs.....')
638     start = timeit.default_timer()
639     self._prepare_item_attr()
640     end = timeit.default_timer()
641     self._all_time += (end - start)
642     print('Time cost: %fs' % (end - start))

```

然后调用 `_linear(k, v)` 来拟合真实值与预测值之间的残差。其中, `k` 为用户 `id`, `v` 为存储了物品 `id`、残差、物品属性的 `dict`。

**algorithm/SVD\_improved.py**

```
644     print('Linear analysis.....')
645     start = timeit.default_timer()
646     for k, v in self._user_item_attrs.items():
647         # 最小二乘法拟合残差
648         param = self._linear(k, v)
649         # 将参数 a,b,c 存储在数组 _user_param 中
650         self._user_param[k] = param
651     end = timeit.default_timer()
652     self._all_time += (end - start)
653     print('Time cost: %fs' % (end - start))
```

### 3.7 测试训练过的模型

`test_model()` 函数用来测试刚刚训练好的模型, 程序会使用测试集来验证模型的准确度, 并得到测试集的 RMSE。在这里, 我们分别计算了 SVD 模型预测的得分 (rp1) 的 RMSE, 以及通过线性拟合修正预测得分之后的结果的 (rp2) RMSE, 这些结果都被写到了文件 `model_test_result.csv` 中。

**algorithm/SVD\_improved.py**

```
656     def test_model(self):
657         self.linear()
658         print('Model testing.....')
659         start = timeit.default_timer()
660         records = 0
661         rmse = 0.0
662         rmse_improved = 0.0
663         resultf = open('model_test_result.csv', 'w')
664         with open(self._TEST_SET, 'r') as f:
665             for line in f.readlines():
666                 if line == "":
667                     continue
668                 user_id, item_id, rate_str = line.split(',')
669                 u = self._user_dict[int(user_id)]
670                 i = self._item_dict[int(item_id)]
671                 r = float(rate_str)
672                 # svd 预测得到的得分
```



```

673         rp1 = self._train_mean + self._bu[u] + self._bi[i] +
        self._dot(u, i)
674         # 经过线性拟合修正后的得分
675         rp2 = rp1 + self._linear_predict(u, i)
676         # 确保得分在范围内
677         if (rp1 < self._scale[0]):
678             rp1 = float(self._scale[0])
679         if (rp1 > self._scale[1]):
680             rp1 = float(self._scale[1])
681         err = r - rp1
682         rmse += err ** 2
683         # 确保得分在范围内
684         if (rp2 < self._scale[0]):
685             rp2 = float(self._scale[0])
686         if (rp2 > self._scale[1]):
687             rp2 = float(self._scale[1])
688         err = r - rp2
689         rmse_improved += err ** 2
690         resultf.write('%d,%d,%f,%f,%f\n' % (int(user_id),
        int(item_id), rp1, rp2, r))
691         records += 1
692     rmse /= records
693     rmse = math.sqrt(rmse)
694     rmse_improved /= records
695     rmse_improved = math.sqrt(rmse_improved)
696     f.close()
697     resultf.close()
698     end = timeit.default_timer()
699     print('RMSE in test set: %f' % rmse)
700     print('Improved RMSE in test set: %f' % rmse_improved)
701     self._all_time += (end - start)
702     print('Time cost: %fs' % (end - start))

```

其中 `_linear_predict()` 用于预测真实值和预测值的差值。函数会通过公式  $err = a \times attr_1 + b \times attr_2 + c$  来计算这个差值, 用来对模型预测的打分做出修正。

#### algorithm/SVD\_improved.py

```

656     def _linear_predict(self, u, i):

```

```

657         return self._user_param[u][0] * self._item_attrs[i][0] +
           self._user_param[u][1] * self._item_attrs[i][1] +
           self._user_param[u][2]

```

文件`model_test_result.csv`的每一行共有五项, 分别是用户 id、物品 id、模型预测打分值`rp1`、经过线性拟合修正后的打分值`rp2`、用户对该物品打分的真实值。

下图中为`model_test_result.csv`的前两行数据, 举例来说, 第一行就表示模型预测的用户 0 对物品 507696 的打分约为 85, 经过线性拟合修正后约为 83, 而用户 0 对该物品打分的真实值为 90。

0	507696	85.36971	83.35402	90
0	137915	87.3749	85.3592	90

Figure 13: 文件 `model_test_result.csv` 的前两行数据

### 3.8 预测打分结果

最后, 程序会对`test.txt`文件指定的用户、物品对的打分进行预测, 并将预测结果按照要求的格式写入到`result1.txt`和`result2.txt`中。由于我们通过线性拟合对打分进行了优化, 因此`result1.txt`和`result2.txt`分别存储了优化前后的分值。

```

algorithm/SVD_improved.py

700     def predict(self):
701         print('Predicting.....')
702         start = timeit.default_timer()
703         test_file = []
704         with open(self._TEST_FILE, 'r') as f:
705             test_file = f.readlines()
706             f.close()
707         user_id = ''
708         item_id = ''
709         uid = 0
710         iid = 0
711         resultf1 = open('result1.txt', 'w') # 优化前
712         resultf2 = open('result2.txt', 'w') # 优化后
713         for test in test_file:
714             line = test.strip()
715             if line.find('|') != -1:
716                 user_id, user_item_count = line.split('|')

```

```
717         uid = int(user_id)
718         resultf1.write(line + '\n')
719         resultf2.write(line + '\n')
720     else:
721         if line == "":
722             continue
723         iid = int(line)
724         u = self._user_dict[uid]
725         i = self._item_dict[iid]
726         # svd 预测得到的得分
727         rp1 = self._train_mean + self._bu[u] + self._bi[i] +
             self._dot(u, i)
728         # 经过线性拟合修正后的得分
729         rp2 = rp1 + self._linear_predict(u, i)
730         # 确保得分在范围内
731         if (rp1 < self._scale[0]):
732             rp1 = float(self._scale[0])
733         if (rp1 > self._scale[1]):
734             rp1 = float(self._scale[1])
735         if (rp2 < self._scale[0]):
736             rp2 = float(self._scale[0])
737         if (rp2 > self._scale[1]):
738             rp2 = float(self._scale[1])
739         resultf1.write('%d %f \n' % (iid, rp1))
740         resultf2.write('%d %f \n' % (iid, rp2))
741
742     resultf1.close()
743     resultf2.close()
744     end = timeit.default_timer()
745     self._all_time += (end - start)
746     print('Time cost: %fs' % (end - start))
```

## 4 实验结果分析

一次运行的结果如下（包括统计文件、划分训练集和测试集、模型训练、模型测试、评分预测），这次指定了 **-g** 和 **-t** 两个参数，表明要重新划分训练集和测试集、重新进行训练；**-m** 参数的含义为查看堆内存详细信息：

```
G:\RecommandationSystem>python -u recommand.py -g -t -m
In SVD_improved.stat() .....
Processing item attr file.....
Processing train file.....
Processing test file.....
Generating new item attr file.....
Making statistics files.....
Time cost in SVD_improved.stat(): 16.544869s
Dividing train and test set.....
Time cost: 8.705590s
Preparing.....
In SVD_improved._prepare() .....
Time cost in SVD_improved._prepare(): 9.482014s
Training.....
RMSE in epoch 0: 27.096739
Time cost: 42.041090s
RMSE in epoch 1: 22.715155
Time cost: 44.713885s
RMSE in epoch 2: 21.110017
Time cost: 43.225252s
RMSE in epoch 3: 20.390432
Time cost: 42.376421s
RMSE in epoch 4: 20.002774
Time cost: 44.863886s
RMSE in epoch 5: 19.774146
Time cost: 44.408153s
RMSE in epoch 6: 19.626631
Time cost: 43.807765s
RMSE in epoch 7: 19.528458
Time cost: 45.021466s
RMSE in epoch 8: 19.462330
Time cost: 44.100183s
RMSE in epoch 9: 19.421008
Time cost: 45.185566s
RMSE in epoch 10: 19.398327
Time cost: 45.307052s
RMSE in epoch 11: 19.386584
Time cost: 43.608304s
RMSE in epoch 12: 19.380626
Time cost: 42.699368s
RMSE in epoch 13: 19.377627
```

**Figure 14:** 一次运行的结果

```

Time cost: 45.294164s
RMSE in epoch 14: 19.376123
Time cost: 43.520387s
RMSE in epoch 15: 19.375370
Time cost: 42.388137s
RMSE in epoch 16: 19.374993
Time cost: 45.739232s
RMSE in epoch 17: 19.374805
Time cost: 43.882090s
RMSE in epoch 18: 19.374710
Time cost: 42.659244s
RMSE in epoch 19: 19.374663
Time cost: 45.183617s
Iteration finished
Writing train result data to files.....
Time cost: 9.952104s
Loading item attrs.....
Time cost: 4.194129s
Linear analysis.....
Time cost: 4.939171s
Model testing.....
RMSE in test set: 26.476013
Improved RMSE in test set: 26.457478
Time cost: 11.334226s
Predicting.....
Time cost: 1.997122s
Process memory usage: 1705.863281 MB
Total time cost: 947.174487s
Partition of a set of 30568179 objects. Total size = 1683770817 bytes.
  Index  Count  %      Size  % Cumulative  % Kind (class / dict of class)
    0 12656785  41 970710168  58 970710168  58 tuple
    1 15868814  52 380851536  23 1351561704  80 float
    2 642598    2 236705584   14 1588267288  94 list
    3    883    0 43609472    3 1631876760  97 dict (no owner)
    4 1245824   4 34889668     2 1666766428  99 int
    5 39937     0 6297224      0 1673063652  99 str
    6 59519     0 1904608      0 1674968260  99 numpy.float64
    7 17611     0 1505261      0 1676473521 100 bytes
    8  8918     0 1291272      0 1677764793 100 types.CodeType
    9  8032     0 1092352      0 1678857145 100 function
<313 more rows. Type e.g. '_more' to view.>

```

Figure 15: 一次运行的结果

第一次运行完之后, 紧接着再运行一次。这次不指定 **-g** 和 **-t** 参数:

```

G:\RecommendationSystem>python -u recommand.py -m
In SVD_improved.stat() .....
Processing item attr file.....
Processing train file.....
Processing test file.....
Generating new item attr file.....
Making statistics files.....
Time cost in SVD_improved.stat(): 16.926092s
Loading train data from file.....
Time cost: 8.845090s
Loading item attrs.....
Time cost: 3.647066s
Linear analysis.....
Time cost: 4.227270s
Model testing.....
RMSE in test set: 26.476013
Improved RMSE in test set: 26.457478
Time cost: 11.186985s
Predicting.....
Time cost: 1.220466s
Process memory usage: 1599.769531 MB
Total time cost: 46.052968s
Partition of a set of 34684624 objects. Total size = 1586585981 bytes.
  Index  Count   %    Size  % Cumulative  % Kind (class / dict of class)
    0 8657243  25 682743160  43 682743160  43 tuple
    1 16048471  46 385163304  24 1067906464  67 float
    2 9182158  26 257107020  16 1325013484  84 int
    3 642598  2 200958840  13 1525972324  96 list
    4 883  0 43609472  3 1569581796  99 dict (no owner)
    5 39933  0 6297020  0 1575878816  99 str
    6 59519  0 1904608  0 1577783424  99 numpy.float64
    7 17611  0 1505261  0 1579288685 100 bytes
    8 8918  0 1291272  0 1580579957 100 types.CodeType
    9 8032  0 1092352  0 1581672309 100 function
<313 more rows. Type e.g. '_more' to view.>

```

Figure 16: 第二次运行的结果

上面在训练中进行了 20 次迭代, 每次迭代的训练集 RMSE 趋势图如下所示:

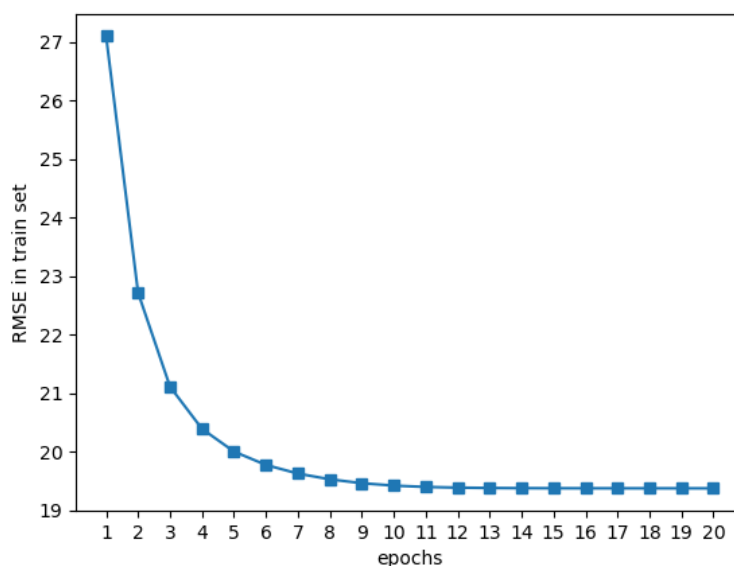


Figure 17: 训练集 RMSE 趋势图

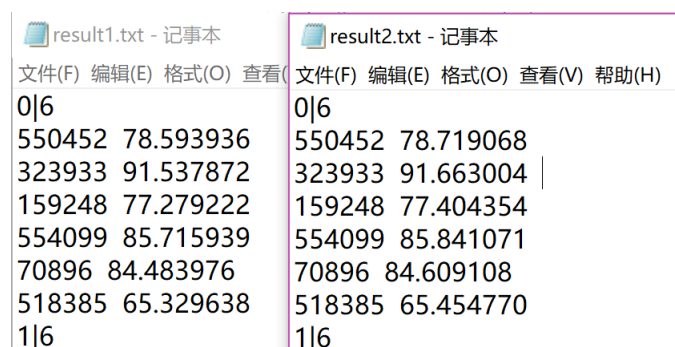
可以发现，迭代的次数越多，训练集的 RMSE 就会越小，并且减小的幅度也会越来越小（直到趋势线慢慢与横轴平行）。这符合我们的预期。

在上述的截图中，引入物品属性进行优化之前测试集的 RMSE 是 26.476013，优化之后的测试集 RMSE 是 26.457478。说明引入物品属性对打分真实值和和预测值的差值进行线性拟合确实起到了一定的优化作用。

上面在训练的时候进行一次迭代的用时为 40 多秒（确实是程序中最慢的一个环节），程序运行总时间约为 947 秒。而第二次重新运行的时候，由于本地已经存在训练集文件、测试集文件以及训练结果文件，而且没有指定命令行参数 **-g**（重新划分训练集和测试集）和 **-t**（重新进行训练），程序会自行查找并加载训练结果文件，因此第二次运行的时间仅有 46 秒。

分析两次运行的过程中堆内存空间的使用情况，可以发现 **tuple** 类型所占比重最大，因为程序中的 **\_sprase\_matrix** 和 **\_sprase\_matrix\_with\_err** 存储的是训练集的内容，为 **list** 类型；而这两个 **list** 的每一个元素的类型都是 **tuple** 类型，这两个 **list** 的长度都等于训练集的长度（约 400 多万）。由此可见，尽管对于这么大的数据集使用了稀疏矩阵的方式进行存储，但是空间消耗还是非常大。由此可以想到，如果使用了稠密矩阵的方式来存储训练集的内容，那么空间开销之大可想而知。

这两次运行生成的结果文件 **result1.txt**（引入物品属性优化前）和 **result2.txt**（引入物品属性优化后）的部分内容如下所示：



result1.txt - 记事本	result2.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)	文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
0 6	0 6
550452 78.593936	550452 78.719068
323933 91.537872	323933 91.663004
159248 77.279222	159248 77.404354
554099 85.715939	554099 85.841071
70896 84.483976	70896 84.609108
518385 65.329638	518385 65.454770
1 6	1 6

**Figure 18:** 两个结果文件对比

经过我们的调参和反复训练，将 SVD 正则化参数设为 0.02、学习率设为 0.005、隐式因子数设为 10 可以起到比较好的推荐效果。由于训练集、测试集的内容是随机的、 $P$  矩阵和  $Q$  矩阵的内容也是随机初始化的，因此每次进行训练得到的结果都不尽相同。但大致方向是相同的，每次测试集的 RMSE 都可以在 26.4 左右。下面是经过多次反复训练，得到的最好的结果，经过物品属性优化的测试集 RMSE 可以达到 26.38（在压缩包中提供的是与这个结果相对应的训练集文件、测试集文件、结果文件、训练文件），与此结果相对应的程序运行截图如下所示：



```
PS G:\aaaa\RecommandationSystem> python -u "g:\aaaa\RecommandationSystem\recommand.py" -t
In SVD_improved.stat() .....
Processing item attr file.....
Processing train file.....
Processing test file.....
Generating new item attr file.....
Making statistics files.....
Time cost in SVD_improved.stat(): 17.108423s
Preparing.....
In SVD_improved._prepare() .....
Time cost in SVD_improved._prepare(): 8.482650s
Training.....
RMSE in epoch 0: 27.082739
Time cost: 41.009008s
RMSE in epoch 1: 22.680463
Time cost: 41.660536s
RMSE in epoch 2: 21.074589
Time cost: 42.651089s
RMSE in epoch 3: 20.348530
Time cost: 42.731513s
RMSE in epoch 4: 19.954170
Time cost: 42.453561s
RMSE in epoch 5: 19.719886
Time cost: 43.894444s
RMSE in epoch 6: 19.569276
Time cost: 42.793658s
RMSE in epoch 7: 19.468873
Time cost: 42.082670s
RMSE in epoch 8: 19.402058
Time cost: 42.542721s
RMSE in epoch 9: 19.360807
Time cost: 42.311185s
RMSE in epoch 10: 19.338279
Time cost: 43.465472s
RMSE in epoch 11: 19.326638
```

Figure 19: 最好的结果截图

```
RMSE in epoch 10: 19.338279
Time cost: 43.465472s
RMSE in epoch 11: 19.326638
Time cost: 42.727386s
RMSE in epoch 12: 19.320737
Time cost: 42.166238s
RMSE in epoch 13: 19.317769
Time cost: 43.147953s
RMSE in epoch 14: 19.316280
Time cost: 43.087631s
RMSE in epoch 15: 19.315535
Time cost: 42.240873s
RMSE in epoch 16: 19.315162
Time cost: 42.983678s
RMSE in epoch 17: 19.314976
Time cost: 43.233479s
RMSE in epoch 18: 19.314882
Time cost: 42.000902s
RMSE in epoch 19: 19.314836
Time cost: 42.821564s
Iteration finished
Writing train result data to files.....
Time cost: 9.882417s
Loading item attrs.....
Time cost: 4.224750s
Linear analysis.....
Time cost: 4.911151s
Model testing.....
RMSE in test set: 26.405304
Improved RMSE in test set: 26.386859
Time cost: 11.204366s
Predicting.....
Time cost: 1.129677s
Total time cost: 908.948995s
PS G:\aaaa\RecommadationSystem>
```

Figure 20: 最好的结果截图

与此结果对应的result1.txt和result2.txt已经在压缩包中提供。

## 4.1 遇到的问题及总结

1. 这次为了编程上的方便所以使用了 Python 实现, 因为 Python 不仅语法简单而且还有很多好用的库提供支持。虽然确实感觉到了使用 Python 编程确实很方便, 但是带来了最大的问题就是训练的时间特别长, 而一次迭代的性能瓶颈就是矩阵的运算。由于 Python 并没有 openmp 这样的可以自动并行化的模块, 我们也没有学习过将矩阵运算作并行化处理的方法, 所以就要寻求其他加速的方式。我们也尝试使用了多进程 multiprocessing 包去处理, 但是效果也没有明显提升(代码未附在压缩包中)。通过查阅资料也发现numba库的jit可以即时编译, 而且只需要装饰一下函数或者类就够了, 然后就尝试使用了一下, 才发现使用numba的条件还是非常苛刻的, 它需要在编译期间就要知道变量的类型, 所以就总是失败, 如果要

能成功使用numba库可能得把代码重新写一遍，由于时间关系也没有实现。还有一个办法就是使用 C++ 去改写本次实验的代码，但也由于时间关系没来得及去改写。我们只能采取将训练的结果保存在本地的手段，这样下次运行程序的时候不用重复训练了，以减少训练的巨大代价，但是这会比较占用磁盘空间。因此训练速度慢是本次实验的一个不足之处，需要后续进行补充。

2. 本次实验虽然尝试使用了线性回归，将物品属性作为参数去拟合打分真实值和 SVD 预测值之差，通过这个来修正最终预测的得分。但是从测试集的 RMSE 的改进程度来看，这个改进效果并不是特别明显（但是也有些许改进）。而且分析model\_test\_result.csv可以发现部分修正的预测值比原先的预测值更加偏离实际的打分值。虽然我们也尝试了使用了非线性回归去解决这个问题，但是效果更不明显。我们总结原因如下：

- 也许处理缺失的物品属性信息的方式不太妥当。由数据集分析这一节可以发现总共最多有 624961 个物品 id，但是itemAttribute.txt只有 507172 个物品的属性信息。因此物品属性信息的缺失值非常多。而如何填补这个缺失值是个学问。我们只是简单粗暴地用两个属性的平均值（使用已有的物品属性信息计算出来的）来填补缺失属性信息的这部分物品属性信息，然后将None用 0 来替换（表示它不属于这两个属性的任何一个属性），之后也没有训练出更合适的物品属性值。这带来的问题就是在线性拟合的时候，可能有大量物品的属性值是重复的，而已知自变量的值越多，线性拟合的结果才会更准确。结果这些重复的属性值就弱化了这个效果（重复的自变量值在实际的效果只能算作一个值），导致真正已知的不重复的自变量的值减少，自然线性拟合的结果就没那么准确了。因此也许使用插值的方式去处理缺失的物品属性信息会更好一些，这样带来重复值的机会就会少一些。
- 也许需要通过拟合更多的参数（例如P矩阵和Q矩阵的隐因子）来增加线性回归的准确度，但是这样会让程序变得更慢。

3. 由于这次使用了Python,Python有很多强大的程序内存分析工具,例如memory\_profiler，但是这个工具最大的缺点就是要分析每一行代码的内存占用，这会让本身就很慢的程序变得更慢。因此我们只好使用上次实验的老办法，查看进程所占用的内存空间，然后借用guppy3库来查看程序堆内存的使用情况。通过这两种方式也可以看出这次实验的程序占用的内存空间非常大，不仅仅是因为数据集本来就大，也有变量分配得不够合理的问题。例如有一些冗余的、没什么用的变量可以删去（例如\_sprase\_matrix和\_sprase\_matrix\_with\_err有大量重复的元素，如果引入了\_sprase\_matrix\_with\_err之后可以将\_sprase\_matrix的空间

强制回收，还有一些用于数据集统计的变量可以清除掉，这样就可以大幅度减少内存占用)，或者发生了内存泄漏也会导致程序占用的内存空间非常大。因此这也是一个不足之处，需要改进。但是这也是一个折中的过程，在优化内存方面需要考虑的因素是：优化了内存使用可能对程序的效率所带来的影响。