



南開大學
Nankai University

网络空间安全学院
大数据计算及应用实验报告

推荐系统

姓名：孙璐

学号：2112060

专业：信息安全

2024 年 6 月 15 日

目录

1 作业要求	2
2 数据集分析	2
2.1 训练集数据分析	6
2.2 测试集数据分析	7
2.3 长尾效应	7
2.4 用户活跃度分析	9
2.5 物品受欢迎程度分析	12
3 SVD	14
3.1 SVD 算法分析	14
3.1.1 特征分解	14
3.1.2 SVD 的说明	14
3.2 代码实现	17
3.3 推荐算法实验结果	27
3.4 其他尝试	28
4 SVD++	31
4.1 算法说明	31
4.2 代码实现	32
4.3 推荐算法实验结果	38
5 UserCF	39
5.1 UserCF 算法说明	39
5.1.1 用户相似度度量	39
5.1.2 计算目标用户对要推荐物品的评分	40
5.2 代码实现	41
5.3 推荐算法实验结果	51
6 ItemCF	51
6.1 ItemCF 算法说明	51
6.1.1 物品相似度度量	52
6.1.2 计算目标用户对要推荐物品的评分	52
6.2 代码实现	52
6.3 推荐算法实验结果	55

1 作业要求

预测 test.txt 文件中各对 (u, i) 的评分分数。

数据集：

- Train.txt, 用于训练模型。
- Test.txt, 用于测试。
- ItemAttribute.txt, 用于训练模型（可选）。
- ResultForm.txt, 是结果文件的格式。

数据集的格式在 DataFormatExplanation.txt 中有解释。请注意，如果能适当地使用 ItemAttribute.txt 并提高算法的性能，可以额外获得最高 10 分的加分，计入您的最终课程成绩。

在本物品中，需要报告 Test.txt 文件中未知各对 (u, i) 的预测评分分数。可以使用课程中学习的任何算法，也可以使用其他资源（如 MOOC）。

一组需要撰写一份关于该物品的报告。报告应包括但不限于以下内容：

- 数据集的基本统计信息（例如：用户数量、评分数量、物品数量等）；
- 算法的详细说明；
- 推荐算法的实验结果（均方根误差、训练时间、空间占用量）；
- 对算法进行理论分析或实验分析。

数据集解释 (DataFormatExplanation.txt) 格式如下：

train.txt

<user id>|<numbers of rating items>
<item id> <score>

test.txt

<user id>|<numbers of rating items>
<item id>

item.txt

<item id>|<attribute_1>|<attribute_2>('None' means this item is not belong to any of attribute_1/2)

2 数据集分析

对 train.txt 和 test.txt 进行数据分析。其中 user_rating_counts、item_rating_counts、score_counts：分别存储每个用户的评分次数、每个物品被评分的次数、不同评分值出现的次数；userNum 记录总的

用户数、itemNum 记录总的物品数、rateNum 记录所有用户对所有物品的评分总数；minUserId、max-UserId、minItemId、maxItemId 用来记录用户和物品的最小和最大 ID。

打开 train.txt 或 test.txt 读取数据，train.txt 和 test.txt 的数据集格式如下所示

```
train.txt
<user id>|<numbers of rating items>
<item id>    <score>
```

```
test.txt
<user id>|<numbers of rating items>
<item id>
```

逐行读取数据，更新用户集合、物品集合、最大最小物品 ID 用户 ID 等，如果数据集是训练集，需统计物品的评分次数和评分的分布，如果数据集的是测试集，只需统计物品 ID。训练集文件读取完成后，计算每个用户和物品的平均评分次数，找出评分次数最多和最少的物品，找出最常见和最不常见的评分及其次数，并计算评分的中位数。

最后将所有物品按评分次数从高到低排序。

Listing 1: data_analysis

```

1
2 def data_analysis(dataPath, isTest):
3     user_rating_counts = {} # 存储每个用户的评分次数
4     item_rating_counts = {} # 存储每个物品被评分的次数
5     score_counts = {} # 存储不同评分的次数
6     users = set() # 存储所有用户ID
7     items = set() # 存储所有物品ID
8     ratings = 0 # 总评分数
9     minUserId = float('inf') # 最小用户ID，初始为正无穷大
10    maxUserId = 0 # 最大用户ID，初始为0
11    minItemId = float('inf') # 最小物品ID，初始为正无穷大
12    maxItemId = 0 # 最大物品ID，初始为0
13    dataInfo = {} # 存储最终的统计信息
14
15    with open(dataPath, 'r') as readStream:
16        for line in readStream:
17            if not line.strip():
18                continue
19            header = line.strip() # 读取并去除行首尾空白字符
20            sepPos = header.find("|") # 找到分隔符的位置
21            userId = int(header[:sepPos]) # 提取用户ID
22            rateNum = int(header[sepPos + 1:]) # 提取评分数量
```

```
23     users.add(userId)  # 将用户ID添加到集合中
24     ratings += rateNum  # 增加总评分数
25     minUserId = min(userId, minUserId)  # 更新最小用户ID
26     maxUserId = max(userId, maxUserId)  # 更新最大用户ID
27
28     # 统计用户的评分次数
29     if userId in user_rating_counts:
30         user_rating_counts[userId] += rateNum
31     else:
32         user_rating_counts[userId] = rateNum
33
34     # 统计每个物品的评分次数和不同评分的次数
35     for _ in range(rateNum):
36         if not isTest:  # 如果不是测试集
37             itemId, score = map(int,
38                                 readStream.readline().strip().split())  #
39                                 读取物品ID和评分
40             if itemId in item_rating_counts:  #
41                 如果物品ID已经存在于字典中
42                 item_rating_counts[itemId] += 1  # 增加评分次数
43             else:
44                 item_rating_counts[itemId] = 1  #
45                 否则初始化评分次数为1
46
47             if score in score_counts:  # 如果当前分数已存在于字典中
48                 score_counts[score] += 1  # 增加分数出现次数
49             else:
50                 score_counts[score] = 1  #
51                 初始化当前分数出现次数为1
52
53         else:  # 如果是测试集
54             itemId = int(readStream.readline().strip())  #
55             只读取物品ID
56             items.add(itemId)  # 将物品ID添加到集合中
57             minItemId = min(itemId, minItemId)  # 更新最小物品ID
58             maxItemId = max(itemId, maxItemId)  # 更新最大物品ID
59
60 if not isTest:
61     # 计算平均每个用户的评分次数和平均每个物品的评分次数
62     avg_user_ratings = sum(user_rating_counts.values()) /
63                         len(user_rating_counts)
64     avg_item_ratings = sum(item_rating_counts.values()) /
```

```
len(item_rating_counts)

# 找出评分次数最多和最少的物品
max_rated_item = max(item_rating_counts,
                      key=item_rating_counts.get)
min_rated_item = min(item_rating_counts,
                      key=item_rating_counts.get)

# 找出最常见的评分和次数
most_common_score = max(score_counts, key=score_counts.get)
most_common_score_count = score_counts[most_common_score]

# 找出最不常见的评分和次数
least_common_score = min(score_counts, key=score_counts.get)
least_common_score_count = score_counts[least_common_score]

# 计算评分的中位数
scores = [score for score, count in score_counts.items() for _
          in range(count)]
median_score = median(scores)

# 填充统计信息
dataInfo["userNum"] = len(users) # 用户数
dataInfo["itemNum"] = len(items) # 物品数
dataInfo["rateNum"] = ratings # 总评分数
dataInfo["minUserId"] = minUserId # 最小用户ID
dataInfo["maxUserId"] = maxUserId # 最大用户ID
dataInfo["minItemId"] = minItemId # 最小物品ID
dataInfo["maxItemId"] = maxItemId # 最大物品ID

if not isTest:
    dataInfo["avg_user_ratings"] = avg_user_ratings #
        平均每个用户的评分次数
    dataInfo["avg_item_ratings"] = avg_item_ratings #
        平均每个物品的评分次数
    dataInfo["max_rated_item"] = max_rated_item # 评分次数最多的物品
    dataInfo["min_rated_item"] = min_rated_item # 评分次数最少的物品
    dataInfo["most_common_score"] = most_common_score # 最常见的评分
    dataInfo["most_common_score_count"] = most_common_score_count #
        最常见评分出现的次数
    dataInfo["least_common_score"] = least_common_score #
        最不常见的评分
```

```

92     dataInfo["least_common_score_count"] = least_common_score_count
93         # 最不常见评分出现的次数
94     dataInfo["median_score"] = median_score # 评分的中位数
95
96     if not isTest: # 如果不是测试集
97         with open("./Data-RecommandationSystem/Long_Tail_Data.txt", 'w')
98             as longTailData: # 打开文件写入长尾数据
99             allItemRatedTimes = [(i, item_rating_counts.get(i, 0)) for i
100                 in range(minItemId, maxItemId + 1)] #
101                 获取所有物品的评分次数
102             allItemRatedTimes.sort(key=lambda x: x[1], reverse=True) #
103                 按评分次数从高到低排序
104             for item in allItemRatedTimes: # 写入每个物品的评分次数
105                 longTailData.write(f"{item[0]} {item[1]}\n")
106
107     return dataInfo # 返回统计信息

```

2.1 训练集数据分析

统计训练集的信息

Listing 2: 统计训练集的信息

```

1
2 # 统计训练集的信息
3 train_result = data_analysis(train_data_path, False)
4 for key, value in train_result.items():
5     print(key, ":", value)

```

```

In [5]: # 统计训练集的信息
train_result = data_analysis(train_data_path, False)
for key, value in train_result.items():
    print(key, ":", value)

userNum : 19835
itemNum : 455691
rateNum : 5001507
minUserId : 0
maxUserId : 19834
minItemId : 0
maxItemId : 624960
avg_user_ratings : 252.1556339803378
avg_item_ratings : 10.975654555389507
max_rated_item : 147073
min_rated_item : 342087
most_common_score : 0
most_common_score_count : 1535563
least_common_score : 7
least_common_score_count : 247
median_score : 50

```

训练集数据分析结果如下:

训练集中共有 19835 位用户, 455691 件物品, 总评分次数达到 5001507 次。其中最小的用户 ID 为 0, 最大的用户 ID 是 19834, 最小的物品 ID 是 0, 最大的物品 ID 是 624960。平均每个用户给

252.1556339803378 (252) 个商品打分, 平均每个商品被评分 10.975654555389507 (11) 次, 其中评分次数最多的物品 ID 是 147073, 评分次数最少的物品 ID 是 342087, 用户最常见的评分是 0, 出现了 2535563 次, 最不常见的评分是 7, 出现了 247 次, 评分的中位数是 50。

2.2 测试集数据分析

统计测试集的信息

Listing 3: 统计测试集的信息

```
1
2 # 统计测试集的信息
3 test_result = data_analysis(test_data_path, True)
4 for key, value in test_result.items():
5     print(key, ":", value)
```

```
In [6]: # 统计测试集的信息
test_result = data_analysis(test_data_path, True)
for key, value in test_result.items():
    print(key, ":", value)

userNum : 19835
itemNum : 28242
rateNum : 119010
minUserId : 0
maxUserId : 19834
minItemId : 29
maxItemId : 624932
```

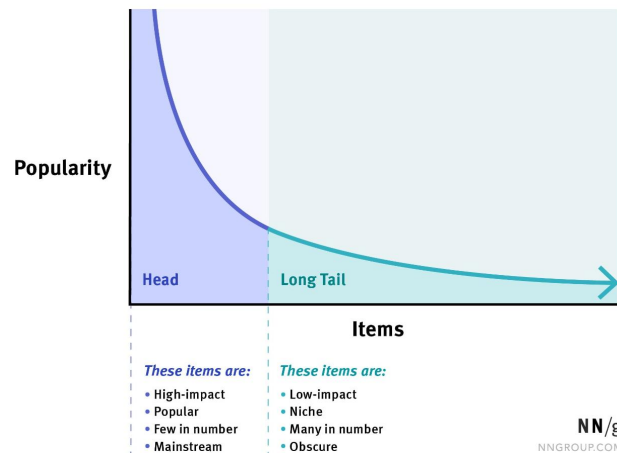
测试集数据分析结果如下:

测试集中共有 19835 位用户, 28242 件物品, 需评分次数为 119010 次。其中最小的用户 ID 为 0, 最大的用户 ID 是 19834, 最小的物品 ID 是 29, 最大的物品 ID 是 624932。

2.3 长尾效应

长尾效应 (Long Tail Effect) 是指在某些领域, 特别是互联网和媒体行业中, 商品或内容的销量或受欢迎程度分布不均衡的现象。具体来说, 少数热门商品或内容会获得大量的销售或关注, 而大量的冷门商品或内容虽然单个销量或关注度较低, 但其总量却占据了一个很大的市场份额。头部为少数非常受欢迎的商品或内容, 销售量或关注度非常高。长尾为大量不太受欢迎的商品或内容, 单个的销售量或关注度较低, 但总和占据了很大的市场份额。

The Long Tail



读取物品及其评分次数数据，将物品按评分次数从高到低排序，计算并绘制评分次数的分布图，标出长尾部分的位置，并返回长尾部分在总数据中的占比。这里认为评分次数小于等于 15 次的物品属于长尾部分。

Listing 4: 长尾效应分析

```

1
2 # 长尾效应分析
3 def analyze_longtail(data_path, threshold):
4     item_rating_counts = {} # 存储每个物品被评分的次数
5
6     # 读取数据并统计每个物品的评分次数
7     with open(data_path, 'r') as file:
8         for line in file:
9             if not line.strip():
10                 continue
11             item_id, rating_count = map(int, line.strip().split())
12             item_rating_counts[item_id] = rating_count
13
14     # 按评分次数从高到低排序
15     sorted_items = sorted(item_rating_counts.items(), key=lambda x:
16                             x[1], reverse=True)
17
18     # 计算长尾部分的数据量和总数据量
19     total_items = len(sorted_items)
20     tail_items = [item for item in sorted_items if item[1] <= threshold]
21     tail_items_count = len(tail_items)
22
23     # 计算长尾部分的占比

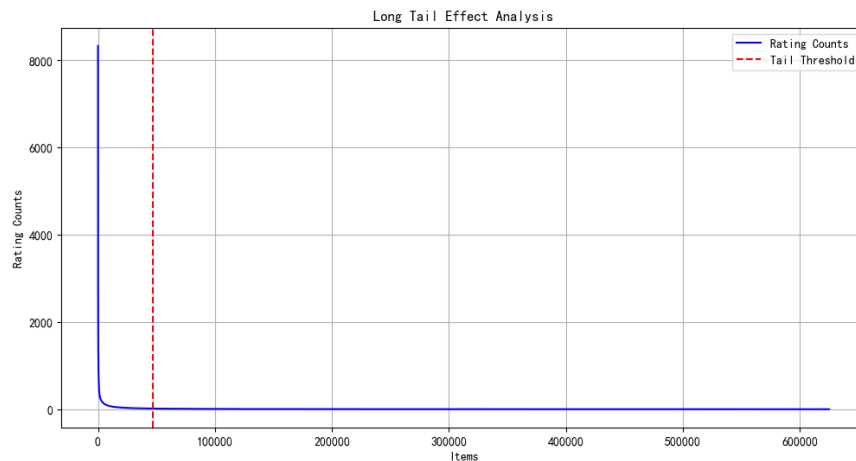
```

```

23     tail_ratio = tail_items_count / total_items
24
25     # 绘制长尾效应图
26     plt.figure(figsize=(12, 6))
27     plt.plot(range(1, total_items + 1), [count for _, count in
28         sorted_items], color='blue', label='Rating Counts')
29     plt.axvline(x=total_items - tail_items_count, color='red',
30         linestyle='--', label='Tail Threshold')
31     plt.title('Long Tail Effect Analysis')
32     plt.xlabel('Items')
33     plt.ylabel('Rating Counts')
34     plt.legend()
35     plt.grid(True)
36     plt.show()
37
38     return tail_ratio
39
40 threshold = 15 # 评分次数小于等于15次的物品属于长尾部分
41 tail_ratio = analyze_longtail(longtail_data_path, threshold)
42 print(f"长尾部分的占比: {tail_ratio:.4%}")

```

绘制出的长尾效应图如下:



如果将评分次数小于等于 15 次的物品认为属于长尾部分, 那么当前长尾部分占比为 92.4725%。

2.4 用户活跃度分析

统计每个用户的评分次数, 绘制直方图, 直方图的横轴表示评分次数, 纵轴表示具有特定评分次数的用户数量。

Listing 5: 用户活跃度分析

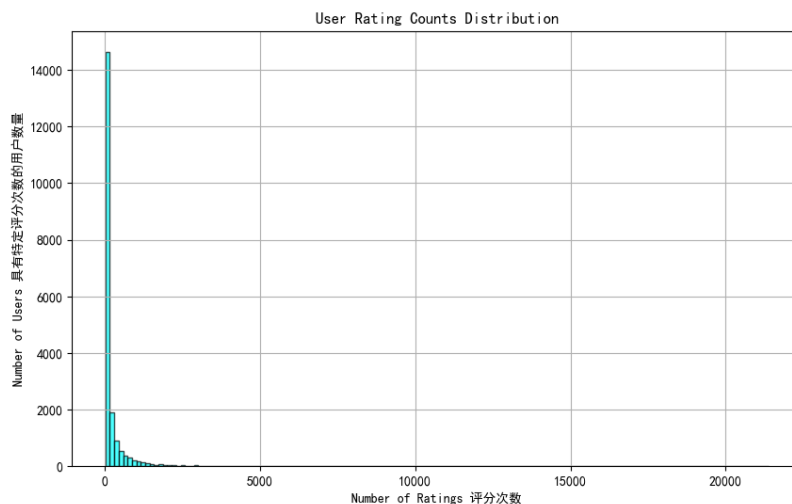
```
2 # 绘图完成用户活跃度分析，分析用户的评分次数分布
3 def plot_user_rating_distribution(dataPath, isTest):
4     user_rating_counts = {} # 存储每个用户的评分次数
5     item_rating_counts = {} # 存储每个物品被评分的次数
6     score_counts = {} # 存储不同评分的次数
7     users = set() # 存储所有用户ID
8     items = set() # 存储所有物品ID
9     ratings = 0 # 总评分数
10    minUserId = float('inf') # 最小用户ID，初始为正无穷大
11    maxUserId = 0 # 最大用户ID，初始为0
12    minItemId = float('inf') # 最小物品ID，初始为正无穷大
13    maxItemId = 0 # 最大物品ID，初始为0
14    dataInfo = {} # 存储最终的统计信息
15
16    with open(dataPath, 'r') as readStream:
17        for line in readStream:
18            if not line.strip():
19                continue
20            header = line.strip() # 读取并去除行首尾空白字符
21            sepPos = header.find("|") # 找到分隔符的位置
22            userId = int(header[:sepPos]) # 提取用户ID
23            rateNum = int(header[sepPos + 1:]) # 提取评分数量
24            users.add(userId) # 将用户ID添加到集合中
25            ratings += rateNum # 增加总评分数
26            minUserId = min(userId, minUserId) # 更新最小用户ID
27            maxUserId = max(userId, maxUserId) # 更新最大用户ID
28
29            # 统计用户的评分次数
30            if userId in user_rating_counts:
31                user_rating_counts[userId] += rateNum
32            else:
33                user_rating_counts[userId] = rateNum
34
35            # 统计每个物品的评分次数和不同评分的次数
36            for _ in range(rateNum):
37                if not isTest: # 如果不是测试集
38                    itemId, score = map(int,
39                                         readStream.readline().strip().split()) #
40                                         读取物品ID和评分
39                    if itemId in item_rating_counts: #
40                        如果物品ID已经存在于字典中
41                        item_rating_counts[itemId] += 1 # 增加评分次数
```

```

41         else:
42             item_rating_counts[itemId] = 1 #
43             否则初始化评分次数为1
44
45         if score in score_counts:# 如果当前分数已存在于字典中
46             score_counts[score] += 1# 增加分数出现次数
47         else:
48             score_counts[score] = 1 #
49             初始化当前分数出现次数为1
50
51         else: # 如果是测试集
52             itemId = int(readStream.readline().strip()) #
53             只读取物品ID
54
55             items.add(itemId) # 将物品ID添加到集合中
56             minItemId = min(itemId, minItemId) # 更新最小物品ID
57             maxItemId = max(itemId, maxItemId) # 更新最大物品ID
58
59     if not isTest:
60         plt.figure(figsize=(10, 6))
61         plt.hist(user_rating_counts.values(), bins=150, color='cyan',
62                 edgecolor='black', alpha=0.7)
63         plt.title('User Rating Counts Distribution')
64         plt.xlabel('Number of Ratings 评分次数')
65         plt.ylabel('Number of Users 具有特定评分次数的用户数量')
66         plt.grid(True)
67         plt.show()

```

绘制直方图如下：



2.5 物品受欢迎程度分析

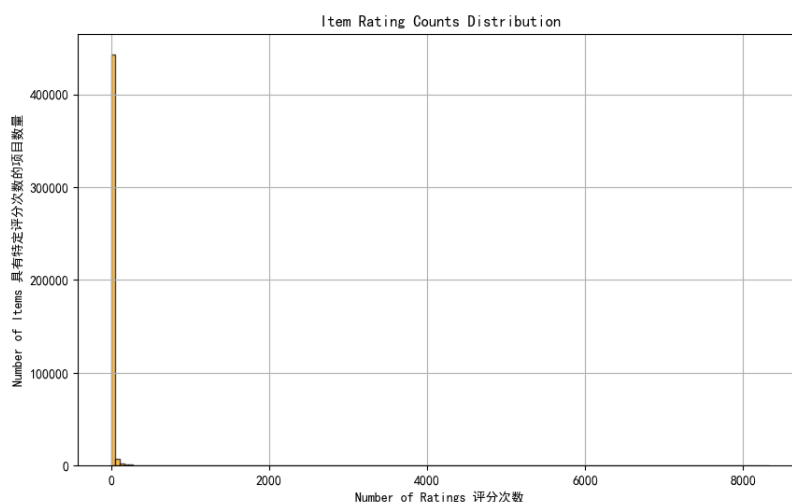
统计每个物品的评分次数，绘制直方图，直方图的横轴表示评分次数，纵轴表示具有特定评分次数的物品数量。这里假定只要用户给了评分，用户对该商品就是有期待的，不管评分是高还是低。

Listing 6: 物品受欢迎程度分析

```
1
2 # 绘图完成物品受欢迎程度分析，分析每个物品的评分次数分布
3 def plot_item_rating_distribution(dataPath, isTest):
4     user_rating_counts = {} # 存储每个用户的评分次数
5     item_rating_counts = {} # 存储每个物品被评分的次数
6     score_counts = {} # 存储不同评分的次数
7     users = set() # 存储所有用户ID
8     items = set() # 存储所有物品ID
9     ratings = 0 # 总评分数
10    minUserId = float('inf') # 最小用户ID，初始为正无穷大
11    maxUserId = 0 # 最大用户ID，初始为0
12    minItemId = float('inf') # 最小物品ID，初始为正无穷大
13    maxItemId = 0 # 最大物品ID，初始为0
14    dataInfo = {} # 存储最终的统计信息
15
16    with open(dataPath, 'r') as readStream:
17        for line in readStream:
18            if not line.strip():
19                continue
20            header = line.strip() # 读取并去除行首尾空白字符
21            sepPos = header.find("|") # 找到分隔符的位置
22            userId = int(header[:sepPos]) # 提取用户ID
23            rateNum = int(header[sepPos + 1:]) # 提取评分数量
24            users.add(userId) # 将用户ID添加到集合中
25            ratings += rateNum # 增加总评分数
26            minUserId = min(userId, minUserId) # 更新最小用户ID
27            maxUserId = max(userId, maxUserId) # 更新最大用户ID
28
29            # 统计用户的评分次数
30            if userId in user_rating_counts:
31                user_rating_counts[userId] += rateNum
32            else:
33                user_rating_counts[userId] = rateNum
34
35            # 统计每个物品的评分次数和不同评分的次数
36            for _ in range(rateNum):
37                if not isTest: # 如果不是测试集
```

```
38         itemId, score = map(int,
39                               readStream.readline().strip().split()) #
40         读取物品ID和评分
41     if itemId in item_rating_counts: #
42         如果物品ID已经存在于字典中
43         item_rating_counts[itemId] += 1 # 增加评分次数
44     else:
45         item_rating_counts[itemId] = 1 #
46         否则初始化评分次数为1
47
48     if score in score_counts: # 如果当前分数已存在于字典中
49         score_counts[score] += 1 # 增加分数出现次数
50     else:
51         score_counts[score] = 1 #
52         初始化当前分数出现次数为1
53
54     else: # 如果是测试集
55         itemId = int(readStream.readline().strip()) #
56         只读取物品ID
57         items.add(itemId) # 将物品ID添加到集合中
58         minItemId = min(itemId, minItemId) # 更新最小物品ID
59         maxItemId = max(itemId, maxItemId) # 更新最大物品ID
60
61 if not isTest:
62     plt.figure(figsize=(10, 6))
63     plt.hist(item_rating_counts.values(), bins=150, color='orange',
64              edgecolor='black', alpha=0.7)
65     plt.title('Item Rating Counts Distribution')
66     plt.xlabel('Number of Ratings 评分次数')
67     plt.ylabel('Number of Items 具有特定评分次数的物品数量')
68     plt.grid(True)
69     plt.show()
```

绘制直方图如下：



3 SVD

3.1 SVD 算法分析

3.1.1 特征分解

特征值和特征向量定义如下：

A 是 n 阶方阵, 如果数 λ 和 n 维非零向量 x 使得 $Ax = \lambda x$ (即 $(A - \lambda E)x = 0$, 转化成求解 $|A - \lambda E| = 0$) 成立, 则称 λ 为 A 的特征值, 向量 x 为 A 的对应特征值 λ 的特征向量。

假设我们得到矩阵 A 的 n 个特征值 $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$, 以及对应的特征向量 $\{w_1, w_2, \dots, w_n\}$ 。如果这 n 个互异的特征向量是线性无关的 (即 A 可以对角化), 那么矩阵 A 可以表示如下：

$$A = W \Sigma W^{-1}$$

其中 W 是这 n 维特征向量张成的 $n \times n$ 维矩阵, Σ 则是以这 n 个特征值为主对角线的 $n \times n$ 维矩阵。一般我们会将 W 的这 n 个特征向量标准化, 即使 $\|w_i\|_2 = 1$, 或者 $w_i^T w_i = 1$, 此时 W 的 n 个特征向量为标准正交基, 满足 $W^T W = I$, 即 $W^T = W^{-1}$ 。此时说 W 为酉矩阵。此时特征分解可以表示为：

$$A = W \Sigma W^T$$

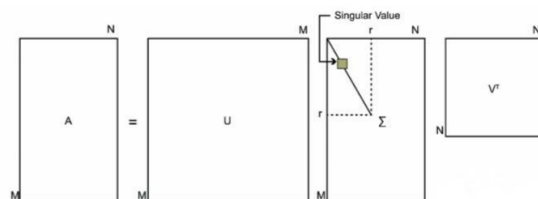
3.1.2 SVD 的说明

特征值分解需要的前提是矩阵分解对象必须是方阵, 因此引入了 SVD。对于任意矩阵 $m \times n$,

$$A = U \Sigma V^T$$

其中, U 是一个 $m \times m$ 的酉矩阵, Σ 是一个 $m \times n$ 的对角矩阵, 对角元素的值称为奇异值, V 是一个 $n \times n$ 的酉矩阵。

其图形式化表示如下



如果想要求 SVD 分解后的 U, Σ, V 这三个矩阵

- 求解 V : 将 A 转置与 A 做矩阵乘法, 得到一个 $n \times n$ 的方阵 $A^T A$ 。进而对其进行特征分解:

$$(A^T A)v_i = \lambda_i v_i$$

求得方阵 $A^T A$ 的 n 个特征值和对应的 n 个特征向量 v_i 。于是所有特征向量张成一个 $n \times n$ 的矩阵 V , 则就是我们 SVD 中要求解的 V 。一般将 V 中的每个特征向量叫做 A 的右奇异向量。

- 求解 U : 同样, 将 A 与 A 的转置做矩阵乘法, 得到一个 $m \times m$ 的方阵 AA^T 。进行特征分解:

$$(AA^T)u_i = \lambda_i u_i$$

求得方阵 AA^T 的 m 个特征值和对应的 m 个特征向量 u_i 。同样所有特征向量张成一个 $m \times m$ 的矩阵 U 。这就是 SVD 中的 U 矩阵, 一般将 U 的每个特征向量叫做 A 的左奇异向量。

- 求解 Σ : 求出每个奇异量 σ 即可

$$A = U\Sigma V^T \Rightarrow AV = U\Sigma V^T V \Rightarrow AV = U\Sigma \Rightarrow Av_i = \sigma_i u_i \Rightarrow \sigma_i = Av_i / u_i$$

这样便可求出每个奇异值了。

Recap: SVD

Remember SVD:

- A : Input data matrix
- U : Left singular vecs
- V : Right singular vecs
- Σ : Singular values

So in our case:

"SVD" on Netflix data: $R \approx Q \cdot P^T$

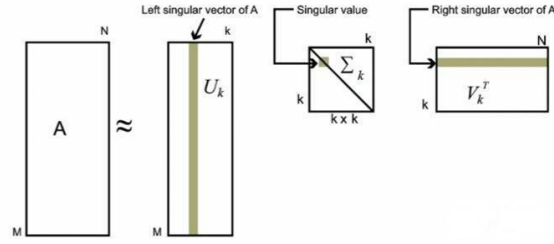
$A = R, Q = U, P^T = \Sigma V^T$

$\hat{r}_{xi} = q_i \cdot p_x$

特征分解中的特征值类似, 在奇异值矩阵中也是按照从大到小排列, 而且奇异值的减少特别的快, 在很多情况下, 前 10% 甚至 1% 的奇异值的和就占了全部的奇异值之和的 99% 以上的比例。也就是说, 我们也可以用最大的 k 个的奇异值和对应的左右奇异向量来近似描述矩阵。也就是说:

$$A_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T \approx U_{m \times k} \Sigma_{k \times k} V_{k \times n}^T$$

其中 k 要比 n 小的很多, 也就是一个大的矩阵 A 可以用三个小的矩阵 $U_{m \times k}, \Sigma_{k \times k}, V_{k \times n}^T$ 来表示。示意图如下, 也现在的矩阵 A 只需要灰色部分的三个小的矩阵就可以近似描述



在将 SVD 技术用在推荐系统的过程中，其基本思想是将用户-物品评分矩阵分解成多个低维矩阵的乘积，从而发现潜在的用户和物品特征，进而进行推荐。构建一个用户-物品评分矩阵 R ，其中 R_{ij} 表示用户 i 对物品 j 的评分，使用 SVD 进行矩阵分解。

有一个用户-物品评分矩阵 R ，其大小为 $m \times n$ ，将其分解为三个矩阵的乘积：

$$R = U \Sigma V^T$$

其中：

- U 是一个 $m \times k$ 的正交矩阵，表示用户特征矩阵。
- Σ 是一个 $k \times k$ 的对角矩阵，对角线上的元素是奇异值，表示特征的重要程度。
- V 是一个 $n \times k$ 的正交矩阵，表示物品特征矩阵。

为了简化计算，通常选择前 k 个最大的奇异值来近似原始矩阵 R ，得到：

$$R \approx U_k \Sigma_k V_k^T$$

其中：

- U_k 是一个 $m \times k$ 的矩阵，包含前 k 个奇异值对应的左奇异向量。
- Σ_k 是一个 $k \times k$ 的对角矩阵，包含前 k 个最大的奇异值。
- V_k 是一个 $n \times k$ 的矩阵，包含前 k 个奇异值对应的右奇异向量。

通过计算用户特征矩阵和物品特征矩阵的乘积，可以得到用户对某商品的预测评分矩阵。 $\hat{R} = U_k \Sigma_k V_k^T$ 。其中， \hat{R}_{ij} 表示预测的用户 i 对物品 j 的评分。

计算 RMSE

Latent Factor Models

users				
1	3		5	4
2	4	1	2	3
3	2	5	4	2
4	3	4	2	5
5	3	2	2	4

factors		
1	-4	2
-5	6	5
-2	3	5
1.1	2.1	-3
-7	2.1	-2
-1	7	3

users									
1.1	-2	3	5	-2	-5	8	-4	3	14
-8	7	5	14	3	-1	14	2.9	-7	12
2.1	-4	6	1.7	2.4	9	-3	4	8	-7

$R \approx PQ^T$

- ❑ SVD isn't defined when entries are missing!
- ❑ Use specialized methods to find P, Q
- $\min_{P, Q} \sum_{(i,x) \in R} (r_{xi} - q_i \cdot p_x)^2 \quad \hat{r}_{xi} = q_i \cdot p_x$
- Note:
 - ❑ We don't require cols of P, Q to be orthogonal/unit length
 - ❑ P, Q map users/movies to a latent space
 - ❑ The most popular model among Netflix contestants

27

如果不考虑迭代过程，只关注于静态的矩阵分解本身，对一个 $m \times n$ 矩阵进行 SVD 分解时间复杂度大概是 $O(mn \min(m, n))$ 。空间复杂度主要存储矩阵占用了很大的空间，原始评分矩阵 R 的占用大小为 $m \times n$ ，分解后矩阵 U 大小为 $m \times k$ (k 是选择奇异值的数量，维度)，奇异值矩阵 Σ 大小为 $k \times k$ ，分解后的矩阵 V 大小为 $n \times k$ ，所以 SVD 算法整体占用的空间非常大，空间复杂度较高。

3.2 代码实现

初始化 SVD 的各个成员变量，包括用户特征向量和物品特征向量的维度、是否包含物品属性、物品属性文件路径，训练集矩阵，物品被评分次数，物品属性，用户偏差，物品偏差，用户特征向量，物品特征向量，平均评分，用户数量等信息。

Listing 7: SVD

```

1
2 class SVD:
3     def __init__(self, dim, has_itemAttribute, itemAttributeFile):
4         self.dim = dim # 用户特征向量和物品特征向量的维度
5         self.has_itemAttribute = has_itemAttribute # 是否包含物品属性
6         self.itemAttributeFile = itemAttributeFile # 物品属性文件
7
8         self.trainMatrix = [] # 训练集的矩阵
9         self.validationMatrix = [] # 验证集的矩阵
10        self.itemRatedCount = [] # 物品被评分的次数
11        self.itemAttributes = [] # 物品属性
12
13        # 用户偏差、物品偏差、用户特征向量和物品特征向量
14        self.userBias = []
15        self.itemBias = []
16        self.userFeatures = []
17        self.itemFeatures = []
18
19        self.meanRating = 0 # 平均评分

```

```

20     self.userNum = 0 # 用户数量
21     self.itemNum = 0 # 物品数量
22     # self.rmse = None

```

processHeader() 负责处理每个用户数据文件每一行的头部，提取用户 ID 和该用户评分数量。

在 processData() 中处理训练集 train.txt。计算用户 ID 和物品 ID 的最大值和最小值，将用户的评分数据存储到 user_ratings 字典中，逐行读取数据文件中的内容，读取用户 ID、评分数量及用户评分的商品信息更新最大最小 ID，通过最大最小 ID 计算用户和物品的数量。

然后初始化用户偏差、物品偏差、用户特征向量和物品特征向量等参数。用户偏差和物品偏差初始化为 0，用户特征向量和物品特征想来那个初始化为 0 到 dim 之间的随机整数，除以 dim 进行归一化。最后，将数据按照比例（这里是 0.1）划分为训练集和验证集。

Listing 8: SVD

```

1
2 class SVD:
3     ...
4     def processHeader(self, header):
5         sep_pos = header.find("|") # 找到分隔符的位置
6         userId = int(header[:sep_pos]) # 提取用户ID
7         rateNum = int(header[sep_pos + 1:]) # 提取评分数量
8         return userId, rateNum
9
10    def initialize_variables(self, lines):
11        maxUserId = 0
12        minUserId = float('inf')
13        maxItemId = 0
14        minItemId = float('inf')
15
16        for line in lines:
17            if not line.strip():
18                continue
19            parts = line.strip().split()
20            userId, rateNum = self.processHeader(parts[0])
21            minUserId = min(userId, minUserId)
22            maxUserId = max(userId, maxUserId)
23
24            for _ in range(rateNum):
25                itemId, score = map(int, lines.pop(0).strip().split())
26                minItemId = min(itemId, minItemId)
27                maxItemId = max(itemId, maxItemId)
28
29        return maxUserId, minUserId, maxItemId, minItemId

```

```
30
31 def processData(self, dataPath, validation_ratio=0.1):
32     maxUserId = 0
33     minUserId = float('inf')
34     maxItemId = 0
35     minItemId = float('inf')
36
37     user_ratings = {}
38
39     with open(dataPath, 'r') as read_stream:
40         for line in read_stream:
41             if not line.strip():
42                 continue
43             parts = line.strip().split()
44             userId, rateNum = self.processHeader(parts[0])
45             minUserId = min(userId, minUserId)
46             maxUserId = max(userId, maxUserId)
47
48             if userId not in user_ratings:
49                 user_ratings[userId] = []
50
51             # 遍历每个用户的评分信息
52             for _ in range(rateNum):
53                 itemId, score = map(int,
54                                     read_stream.readline().strip().split()) #
55                                     读取物品ID和评分
56                 user_ratings[userId].append((itemId, score))
57                 minItemId = min(itemId, minItemId)
58                 maxItemId = max(itemId, maxItemId)
59
60             # 计算用户数量和物品数量
61             self.userNum = maxUserId - minUserId + 1
62             self.itemNum = maxItemId - minItemId + 1
63
64             # 初始化用户偏差、物品偏差、用户特征向量和物品特征向量等参数
65             self.user_bias = np.zeros(self.userNum)
66             self.item_bias = np.zeros(self.itemNum)
67             # self.user_features = np.random.rand(self.userNum, self.dim) /
68             # self.dim
69             # self.item_features = np.random.rand(self.itemNum, self.dim) /
70             # self.dim
71             self.user_features = np.random.randint(low=0, high=self.dim,
```

```

        size=(self.userNum, self.dim)).astype(float)/ self.dim
68 self.item_features = np.random.randint(low=0, high=self.dim,
        size=(self.itemNum, self.dim)).astype(float)/ self.dim
69
70 #
    用户-物品评分矩阵，每一行代表一个用户，每一列代表一个物品，列表中的元素是
71 self.trainMatrix = [[] for _ in range(self.userNum)]
72 self.validationMatrix = [[] for _ in range(self.userNum)]
73 self.item_rated_count = [0] * self.itemNum #
    每个元素表示对应物品被评分的次数
74 print("Go to generate trainMatrix ...")
75 self.generateTrainMatrix(dataPath)
76
77 for userId, ratings in user_ratings.items():
78     random.shuffle(ratings)
79     split_point = int(len(ratings) * (1 - validation_ratio))
80     self.validationMatrix[userId] = ratings[split_point:]
81
82 def init_model_parameters(self):
83     random.seed()
84
85     # 初始化用户偏差和物品偏差
86     self.user_bias = np.zeros(self.userNum) # 初始化用户偏差为0
87     self.item_bias = np.zeros(self.itemNum) # 初始化物品偏差为0
88
89     # 初始化用户特征向量
90     self.user_features = np.random.randint(low=0, high=self.dim,
        size=(self.userNum, self.dim)).astype(float)/ self.dim
91     # self.user_features = np.random.rand(self.userNum, self.dim) /
        self.dim
92
93     # 初始化物品特征向量
94     # self.item_features = np.random.rand(self.itemNum, self.dim) /
        self.dim
95     self.item_features = np.random.randint(low=0, high=self.dim,
        size=(self.itemNum, self.dim)).astype(float)/ self.dim
96
97     print("init_model_parameters finished ...")

```

load_item_attributes() 从属性文件 itemAttribute.txt 中加载物品的属性信息，将属性信息存储在 self.itemAttributes 列表中，用物品 ID 索引。

Listing 9: SVD

```

1
2 class SVD:
3     ...
4     def load_item_attributes(self):
5         # 初始化属性列表, 初始值为 (-1, -1)
6         self.itemAttributes = [(-1, -1) for _ in range(self.itemNum)] #
            初始化属性列表
7         with open(self.itemAttributeFile, 'r') as input_file:
8             for line in input_file:
9                 parts = line.strip().split('|')
10                itemId = int(parts[0]) # 获取物品 ID
11                attr1 = int(parts[1]) if parts[1] != "None" else -1 #
                    获取第一个属性
12                attr2 = int(parts[2]) if parts[2] != "None" else -1 #
                    获取第二个属性
13                # 将物品 ID 与属性对应起来
14                self.itemAttributes[itemId] = (attr1, attr2)
15        print("load_item_attributes finished ...")

```

processRatings() 处理每个用户的评分数据, 并将其添加到训练集的矩阵中, userId 对应的子列表中添加一个元组 (itemId, actual_rating)。累加计算评分等待后续计算平均分并记录每个物品被评分的次数。

Listing 10: SVD

```

1
2 class SVD:
3     ...
4     def processRatings(self, stream, userId, rateNum):
5         for _ in range(rateNum):
6             itemId, actual_rating= map(int,
7                 stream.readline().strip().split()) # 读取物品ID和评分
8
9             #
                用户-物品评分矩阵, 每一行代表一个用户, 每一列代表一个物品, 列表中的元
10            self.trainMatrix[userId].append((itemId, actual_rating))#
                分配到训练集
11            self.mean_rating += actual_rating# 累加评分以计算平均值
            self.item_rated_count[itemId] += 1# 记录物品被评分的次数

```

调用 generateTrainMatrix() 逐行读取数据集, 调用 processHeader() 处理每个用户第一行的数据提取用户 ID 和评分数量, 调用 processRatings() 处理每个用户的每个商品的评分数据加到训练集矩

阵，记录每个物品的被评分次数。处理完数据集的所有数据后计算平均评分，调用 `saveTrainMatrix()` 保存训练集矩阵。

Listing 11: SVD

```

1
2 class SVD:
3     ...
4     def generateTrainMatrix(self, dataPath):
5         random.seed()
6         train_num = 0
7
8         with open(dataPath, 'r') as readStream:
9             for line in readStream:
10                 if not line.strip():
11                     continue
12                 parts = line.strip().split()
13                 userId, rateNum = self.processHeader(parts[0])
14                 self.processRatings(readStream, userId, rateNum)
15                 train_num += rateNum
16
17         print("generateTrainMatrix finished ...")
18         self.mean_rating /= train_num # 计算训练集评分的平均值
19         self.saveTrainMatrix()
20         # print("TrainMatrix saved ...")
21
22     def saveTrainMatrix(self):
23         with open("./Data-RecommandationSystem/SVD_TrainMatrix.pkl",
24                 'wb') as file:
25             pickle.dump(self.trainMatrix, file)
26         print("Train matrix saved ...")

```

`predict_rating()` 根据训练好的 SVD 模型参数，预测用户 `user_id` 对物品 `item_id` 的评分。使用模型中学习到的用户偏差、物品偏差以及用户特征向量和物品特征向量的内积，加上全局平均评分，来预测评分值，并确保预测评分在 0 到 100 之间。

SVD 模型预测用户 u 对物品 i 的评分 \hat{r}_{ui} 计算公式如下

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{q}_i^T \mathbf{p}_u$$

其中：

- \hat{r}_{ui} 表示用户 u 对物品 i 的预测评分。
- μ 是所有已知评分的全局平均值。
- b_u 是用户 u 的用户偏差。

- b_i 是物品 i 的物品偏差。
- \mathbf{q}_i 是物品 i 的特征向量。
- \mathbf{p}_u 是用户 u 的特征向量。

这些特征向量 \mathbf{q}_i 和 \mathbf{p}_u 是在模型训练得到的，它们表示了用户和物品在一个潜在的特征空间中的表达。

Listing 12: SVD

```

1
2 class SVD:
3     ...
4     def predict_rating(self, user_id, item_id):
5         #
6         根据SVD模型的公式计算用户user_id对物品item_id的预测评分predicted_rating
7         predicted_rating = self.mean_rating + self.user_bias[user_id] +
8         self.item_bias[item_id] + np.dot(self.user_features[user_id],
        self.item_features[item_id])
9         # 限制预测评分在0到100之间
10        return min(max(predicted_rating, 0), 100)

```

calculate_rmse 计算训练集上的均方根误差 (RMSE)，评估模型在训练数据上的预测精度。遍历所有用户和其评分过的物品，使用 predict_rating 函数预测评分并计算实际评分与预测评分的误差，计算总误差的平方和，累计误差的平方除以评分的商品数量并计算平方根，得到最终的均方根误差。

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{u,i} (\hat{r}_{ui} - r_{ui})^2}$$

其中:

- RMSE 表示均方根误差
- N 表示所有用户的评分次数
- \hat{r}_{ui} 表示模型预测的用户 u 对物品 i 的评分。
- r_{ui} 表示的用户 u 对物品 i 的实际评分。

Listing 13: SVD

```

1
2 class SVD:
3     ...
4     def calculate_rmse(self):
5         rating_count = 0
6         total_error = 0
7

```



```

8      # 遍历所有用户
9      for userId in range(self.userNum):
10         # 遍历该用户评分过的所有物品
11         for itemId, actual_rating in self.trainMatrix[userId]:
12             # 使用 predict_rating 函数预测用户对物品的评分
13             predicted_rating = self.predict_rating(userId, itemId)
14             error = actual_rating - predicted_rating #
15                 计算实际评分与预测评分的误差
16             total_error += error ** 2 # 累计误差的平方
17             rating_count += 1
18
19         # 计算训练集上的均方根误差
20         train_rmse = math.sqrt(total_error / rating_count)
21         print(f"train RMSE: {train_rmse}")

```

train_epoch() 进行一轮训练，对每个用户和其评分过的物品，计算预测评分与实际评分的误差，并根据误差和学习率更新用户偏差 (user_bias)、物品偏差 (item_bias)、用户特征向量 (user_features) 和物品特征向量 (item_features)。

train() 则首先进行数据预处理，初始化模型参数，加载物品属性，然后进行迭代训练。每个 epoch 中，遍历每个用户及其评分过的物品，计算预测评分并更新参数。每轮每隔 3000 个用户打印一次训练进度信息，每轮训练结束后计算并输出训练集上的 RMSE，学习率按指数衰减。

具体的更新规则如下：

- 更新用户偏差 (User Bias)：

$$b_u^{(t+1)} = b_u^{(t)} + \eta \cdot (e_{ui} - \lambda \cdot b_u^{(t)})$$

$b_u^{(t)}$ 是第 t 次迭代时的用户偏差， η 是学习率， e_{ui} 是实际评分与预测评分之间的误差， λ 是正则化参数。

- 更新物品偏差 (Item Bias)：

$$b_i^{(t+1)} = b_i^{(t)} + \eta \cdot (e_{ui} - \lambda \cdot b_i^{(t)})$$

$b_i^{(t)}$ 是第 t 次迭代时的物品偏差， η 是学习率， e_{ui} 是实际评分与预测评分之间的误差， λ 是正则化参数。

- 更新用户特征向量 (User Feature Vector)：

$$p_u^{(t+1)} = p_u^{(t)} + \eta \cdot (e_{ui} \cdot q_i^{(t)} - \lambda \cdot p_u^{(t)})$$

$p_u^{(t)}$ 是第 t 次迭代时的用户特征向量， $q_i^{(t)}$ 是第 t 次迭代时的物品特征向量， η 是学习率， e_{ui} 是实际评分与预测评分之间的误差， λ 是正则化参数。

- 更新物品特征向量 (Item Feature Vector)：

$$q_i^{(t+1)} = q_i^{(t)} + \eta \cdot (e_{ui} \cdot p_u^{(t)} - \lambda \cdot q_i^{(t)})$$

$q_i^{(t)}$ 是第 t 次迭代时的物品特征向量, $p_u^{(t)}$ 是第 t 次迭代时的用户特征向量, η 是学习率, e_{ui} 是实际评分与预测评分之间的误差, λ 是正则化参数。

Listing 14: SVD

```

1
2 class SVD:
3     ...
4     def train_epoch(self, learning_rate, regularization):
5         for userId in range(self.userNum):
6             for itemId, actual_rating in self.trainMatrix[userId]:
7                 predicted_rating = self.predict_rating(userId, itemId)
8                 error = actual_rating - predicted_rating
9
10                self.user_bias[userId] += learning_rate * (error -
11                    regularization * self.user_bias[userId])
12                self.item_bias[itemId] += learning_rate * (error -
13                    regularization * self.item_bias[itemId])
14                user_features_backup =
15                    np.copy(self.user_features[userId])
16
17                self.user_features[userId] += learning_rate * (error *
18                    self.item_features[itemId] - regularization *
19                    self.user_features[userId])
20                self.item_features[itemId] += learning_rate * (error *
21                    user_features_backup - regularization *
22                    self.item_features[itemId])
23
24    def train(self, trainFile, initial_learning_rate, regularization,
25        epochs, validation_ratio=0.1):
26        print("Preprocessing .....")
27        self.processData(trainFile, validation_ratio)
28        self.init_model_parameters()
29        if self.has_itemAttribute:
30            self.load_item_attributes()
31        print("Training .....")
32        start_time = time.time()
33
34        print("epochs:", epochs)
35        print("UserNum: ", self.userNum)
36        learning_rate = initial_learning_rate
37
38        for epoch in range(1, epochs + 1):

```

```
31     # self.train_epoch(learning_rate, regularization)
32     # 遍历所有用户
33     for userId in range(self.userNum):
34         # 遍历该用户评分过的物品
35         for itemId, actual_rating in self.trainMatrix[userId]:
36             predicted_rating = self.predict_rating(userId,
37                 itemId) # 预测评分
38             error = actual_rating - predicted_rating # 计算误差
39
40             # 更新用户偏差项
41             self.user_bias[userId] += learning_rate * (error -
42                 regularization * self.user_bias[userId])
43
44             # 更新物品偏差项
45             self.item_bias[itemId] += learning_rate * (error -
46                 regularization * self.item_bias[itemId])
47
48             user_features_backup =
49                 np.copy(self.user_features[userId]) # 备份用户向量
50
51             # 更新用户向量和物品向量
52             self.user_features[userId] += learning_rate * (error
53                 * self.item_features[itemId] - regularization *
54                 self.user_features[userId])
55             self.item_features[itemId] += learning_rate * (error
56                 * user_features_backup - regularization *
57                 self.item_features[itemId])
58
59             # 每轮每隔3000个用户打印一次信息
60             if userId % 3000 == 0:
61                 print(f"Epoch {epoch}, {userId} users processed ...")
62
63             print(f"Epoch {epoch} finished ...")
64             self.calculate_rmse()
65             # 学习率衰减
66             # learning_rate = initial_learning_rate / (1 + decay_rate *
67                 epoch)
68             learning_rate *= 0.9999
69
70             # 记录训练结束时间
71             end_time = time.time()
72             print(f"Training time: {end_time - start_time} seconds")
```

predict() 使用训练好的模型进行预测，并将预测结果写入文件。

Listing 15: SVD

```

1
2 class SVD:
3     ...
4     def predict(self, predictFile):
5         print("Predicting .....")
6         with open("./Data-RecommandationSystem/Res_SVD.txt", 'w') as
            output:
7             with open(predictFile, 'r') as readStream:
8                 for line in readStream:
9                     parts = line.strip().split('|')
10                    userId = int(parts[0])
11                    output.write(f"{userId}|{len(parts[1:])}\n") #
                        写入用户ID和评分数量
12                    ratings = parts[1:]
13                    for rating in ratings:
14                        itemId = int(readStream.readline().strip()) #
                            读取物品ID和评分
15                        predicted_rating = self.predict_rating(userId,
                            itemId)
16                        output.write(f"{itemId}
                            {int(predicted_rating)}\n") #
                            写入物品ID和预测评分
17                    print("Prediction finished ...")

```

3.3 推荐算法实验结果

这里假设用户特征向量和物品特征向量的维度是 200，并且使用了物品属性文件，初始学习率是 0.001，正则化参数为 1.5，迭代训练 50 轮，分割验证集系数为 0.1。

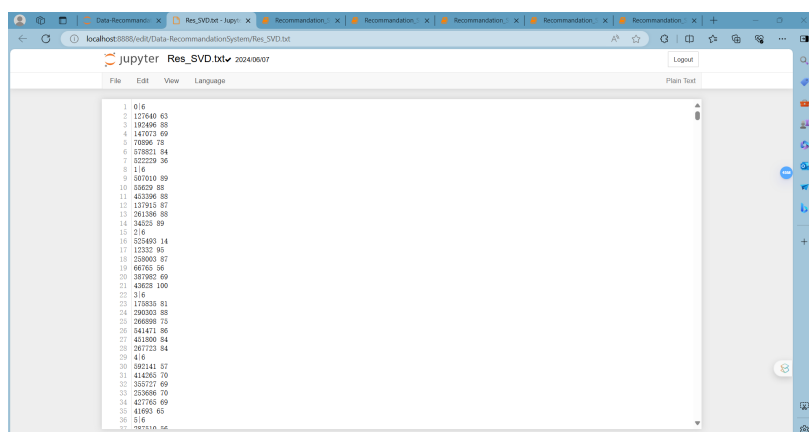
Listing 16: SVD

```

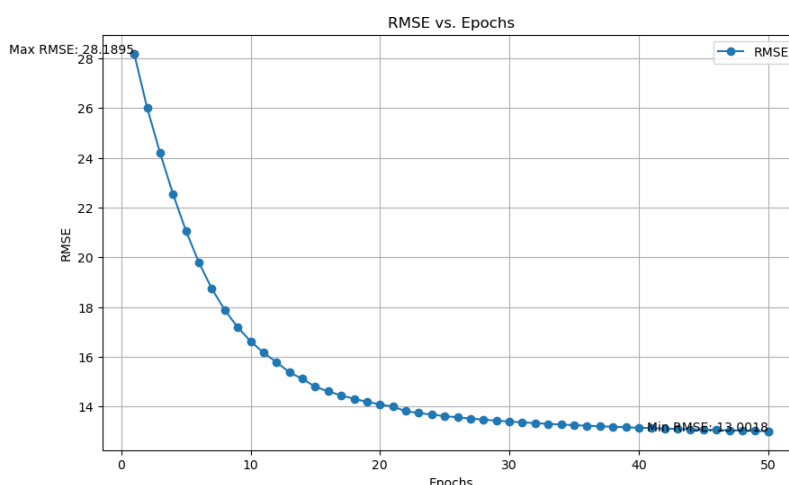
1
2 svd = SVD(200, True, "./Data-RecommandationSystem/itemAttribute.txt")
3 svd.train("./Data-RecommandationSystem/train.txt", 0.001, 1.5, 50, 0.1)
4 svd.predict("./Data-RecommandationSystem/test.txt")

```

预测结果文件 Res_SVD.txt 如下：



最开始的几轮 RMSE 值为 30 左右，且下降的很快，在训练到第 50 轮后，RMSE 值降低到 12-13 左右。如果再继续训练，虽然 RMSE 下降的越来越慢，但可以继续下降。曾尝试训练 150 轮，RMSE 可以降低到 11-12 之间。



在训练 50 轮的前提下，第一轮的 RMSE 值为 28.189479245705037，第 50 轮的 RMSE 值为 13.101790692729472。总的训练时间为 4758s（在 jupyter 上的训练时间）。

但在本次作业实现的 4 种算法中，SVD 算法的训练时间是最短的，且预测效果是最好的。其他 4 种算法的训练时间大概是 1-2 个小时（就没有再记录准确的训练时间了）。推测原因可能是因为用的 jupyter 运行程序，当把代码文件下载到本地进行运行时，运行时间缩短了一半左右。同时本次作业实现的四种算法中都用到了 item_Attributes.txt 属性文件以提高模型的预测准确率。

如果想要进一步缩短程序时间，可以进一步优化代码结构，减少不必要的循环，并采用并程序计算、分块矩阵运算等方法进一步提高程序运行效率。

3.4 其他尝试

使用 Surprise 库进行 SVD 算法的实现，将所有数据处理成 user_id, item_id, rating 的格式，写入 train_processed.txt 文件中。使用 Surprise 库中的 Reader 类指定数据的格式和分隔符，以及评分范围，使用 Dataset.load_from_file 方法从处理后的文件 train_processed.txt 中加载数据，并将其转换为 Surprise 库中的 Dataset 对象 data。

使用了 Surprise 库中的 SVD 算法，设置学习率为 0.001，正则化参数为 1.5，迭代训练 20 次。每

轮调用 Surprise 库中模型对象 `self.model` 的 `fit` 方法，在训练集 `trainset` 上训练模型，使用训练好的模型进行预测，并计算预测结果与实际结果的 RMSE，保存模型并使用模型预测结果。

Listing 17: SVD(else)

```

1
2 class RecommendationSystem:
3     def __init__(self, train_file, test_file, item_file=None):
4         self.train_file = train_file
5         self.test_file = test_file
6         self.item_file = item_file
7         self.model = None
8
9     def load_data(self):
10        print("Loading and processing training data...")
11        train_data = []
12        with open(self.train_file, 'r') as f:
13            lines = f.readlines()
14            for i in range(len(lines)):
15                if '|' in lines[i]:
16                    user_id, num_ratings = map(int, lines[i].split('|'))
17                    for j in range(1, num_ratings + 1):
18                        item_id, rating = map(int, lines[i + j].split())
19                        train_data.append((user_id, item_id, rating))
20        train_file = './Data-RecommandationSystem/train_processed.txt'
21        with open(train_file, 'w') as f:
22            for user_id, item_id, rating in train_data:
23                f.write(f"{user_id} {item_id} {rating}\n")
24        print("Training data processed and saved.")
25
26        reader = Reader(line_format='user item rating', sep=' ',
27                          rating_scale=(0, 100))
28        data = Dataset.load_from_file(train_file, reader=reader)
29        return data
30
31    def train(self, initial_learning_rate=0.005, regularization=0.02,
32              epochs=20):
33        print("Starting training process...")
34        data = self.load_data()
35        # trainset, testset = train_test_split(data, test_size=0.2)
36        print("Training and test sets created.")
37
38        # 初始化模型

```

```

37     self.model = SVD(n_epochs=1, lr_all=initial_learning_rate,
38                       reg_all=regularization)
39
40     for epoch in range(1, epochs + 1):
41         print(f"Starting epoch {epoch}...")
42         self.model.fit(trainset)
43
44         # 在测试集上评估模型
45         # predictions = self.model.test(testset)
46         predictions = self.model.test(trainset)
47         rmse = accuracy.rmse(predictions, verbose=True)
48         print(f"Epoch {epoch} RMSE: {rmse}")
49
50     print("Model training completed.")
51
52     # 保存模型
53     print("Saving the trained model...")
54     with open('./Data-RecommandationSystem/svd_model.pkl', 'wb') as
55         f:
56         pickle.dump(self.model, f)
57     print("Model saved.")
58
59     def predict(self):
60         print("Starting prediction process...")
61         if self.model is None:
62             print("Loading the trained model...")
63             with open('./Data-RecommandationSystem/svd_model.pkl', 'rb')
64                 as f:
65                 self.model = pickle.load(f)
66             print("Model loaded.")
67
68         with open(self.test_file, 'r') as f:
69             lines = f.readlines()
70             results = []
71             for i in range(len(lines)):
72                 if '|' in lines[i]:
73                     user_id, num_ratings = map(int, lines[i].split('|'))
74                     results.append(f"{user_id}|{num_ratings}\n")
75                     for j in range(1, num_ratings + 1):
76                         item_id = int(lines[i + j].strip())
77                         pred = self.model.predict(user_id, item_id).est
78                         results.append(f"{item_id} {int(pred)}\n")

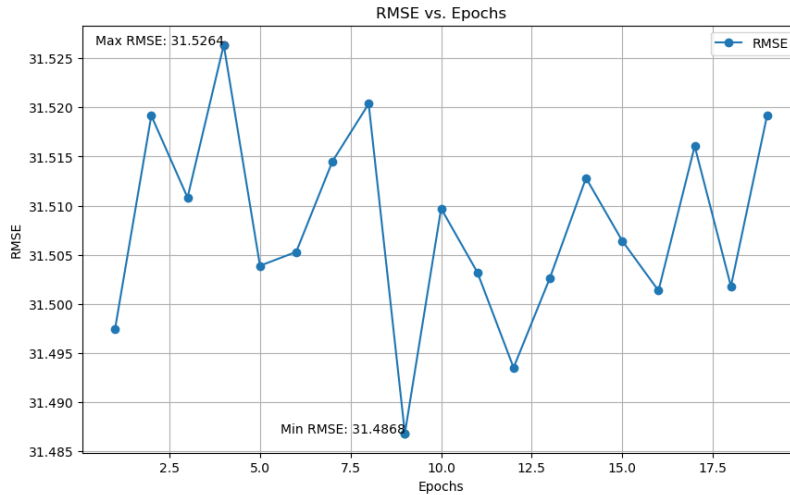
```

```

76
77     print("Saving the prediction results...")
78     with open('./Data-RecommandationSystem/Res_SVD_copy1.txt', 'w')
79         as f:
80         f.writelines(results)
81     print("Prediction finished and results saved.")

```

每一轮的 RMSE 基本都在 31 左右，且每轮次之间 RMSE 没有明显的变化。



4 SVD++

4.1 算法说明

SVD++ 是在 SVD 的基础上，进一步考虑了隐式反馈（如用户的浏览历史、点击记录等未评分行为），从而提高推荐的准确性

1. 用户隐式反馈：

- 隐式反馈指用户与物品的交互行为，不直接给出评分，但这些行为暗示了用户的兴趣。

2. 扩展模型：

- SVD++ 模型加入了隐式反馈的用户特征向量 y_j ：

$$\hat{R}_{ij} = \mathbf{u}_i^T (\mathbf{v}_j + \sum_{k \in N(i)} y_k)$$

其中， $N(i)$ 是用户 i 与之有隐式反馈的物品集合。

3. 用户特征向量更新：

- 在 SVD++ 中，用户特征向量 \mathbf{u}_i 由两个部分组成：

$$\mathbf{u}_i = \mathbf{p}_i + \frac{1}{\sqrt{|N(i)|}} \sum_{k \in N(i)} y_k$$

其中, \mathbf{p}_i 是用户的显式特征向量, $\sum_{k \in N(i)} y_k$ 是隐式反馈特征向量的和。

如果不考虑迭代过程, 只关注于静态的矩阵分解本身, 下面进行时间复杂度和空间复杂度的分析

SVD++ 的时间复杂度与 SVD 类似, 主要的计算开销在于分解评分矩阵和优化模型参数, 对一个 $m \times n$ 矩阵进行 SVD 分解时间复杂度大概是 $O(mn \min(m, n))$ 。因为需要存储额外的用户隐式反馈特征的信息, 空间复杂度略高于 SVD。

4.2 代码实现

这里只展示代码中与 SVD 不同的部分

SVD++ 考虑到了物品交互向量试图增加对物品特征的考量

Listing 18: SVD++

```

1
2 class SVD_PlusPlus:
3     ...
4     def __init__(self, dim):
5         ...
6         self.itemInteraction = [] # 存储物品交互向量
7         ...

```

在 `process_data()` 中对物品交互向量进行初始化, 初始化为一个零矩阵, `self.itemNum` 是物品的数量, `self.dim` 是隐特征向量的维度。

Listing 19: SVD++

```

1
2 class SVD_PlusPlus:
3     ...
4     def process_data(self, data_path):
5         ...
6         self.itemInteraction = np.zeros((self.itemNum, self.dim)) #
           物品交互向量
7         ...

```

`predict_rating()` 根据训练好的 SVD 模型参数, 预测用户 `user_id` 对物品 `item_id` 的评分。使用模型中学习到的用户偏差、物品偏差以及用户特征向量和物品特征向量的内积, 加上全局平均评分, 来预测评分值, 并确保预测评分在 0 到 100 之间。

`predict_rating` 函数预测评分并计算实际评分与预测评分的误差, 计算总误差的平方和, 累计误差的平方除以评分的商品数量并计算平方根, 得到最终的均方根误差。

SVD++ 模型中用户 u 对物品 i 的预测评分公式如下:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left(p_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j \right)$$

其中:

- \hat{r}_{ui} 是预测的用户 u 对物品 i 的评分。
- μ 是所有评分的全局平均值。
- b_u 是用户 u 的用户偏差。
- b_i 是物品 i 的物品偏差。
- q_i 是物品 i 的特征向量。
- p_u 是用户 u 的特征向量。
- $N(u)$ 是用户 u 交互过的物品集合。
- y_j 是物品 j 的交互向量。

Listing 20: SVD++

```

1
2 class SVD_PlusPlus:
3     ...
4     def predict_rating(self, user_id, item_id, interaction_vector):
5         # 预测评分
6         predicted_rating = self.mean_rating + self.user_bias[user_id] +
7             self.item_bias[item_id] + np.dot(self.item_features[item_id],
8             interaction_vector)
9         # 限制预测评分在0到100之间
10        return min(max(predicted_rating, 0), 100)

```

calculate_rmse() 计算训练集上的 RMSE, interaction_vector 是物品的交互向量, 初始化为全零向量。user_interactions 是一个列表, 每个元素是用户的交互向量, 初始化为全零向量。遍历所有用户, 对每个用户的评分列表中的物品进行处理。计算每个用户评分的物品数量的平方根 y_sums[user_id], 并累加每个物品的交互向量到 user_interactions[user_id]。将每个用户的交互向量除以 y_sums[user_id], 然后加上对应的用户特征向量 self.user_features[user_id]。

使用 self.predict_rating() 方法预测用户对物品的评分 predicted_rating。计算预测评分与实际评分的误差 error, 并累加误差的平方到 total_error。在 SVD++ 的 RMSE 计算中, 因为用到了 predict_rating() 进行对物品评分的预测, 所以在 RMSE 计算中考虑到了物品交互向量。

Listing 21: SVD++

```

1
2 class SVD_PlusPlus:
3     ...
4     def calculate_rmse(self):
5         rating_count = 0 # 评分总数
6         total_error = 0 # 总误差

```

```
7     interaction_vector = np.zeros(self.dim) # 物品的交互向量
8
9     # 预计算用户的交互向量和y值
10    user_interactions = [np.zeros(self.dim) for _ in
11                          range(self.userNum)]
12    y_sums = [0] * self.userNum
13
14    # 遍历所有用户
15    for user_id in range(self.userNum):
16        rated_items = self.trainMatrix[user_id]
17        if not rated_items:
18            continue
19
20        y_sums[user_id] = math.sqrt(len(rated_items)) #
21        # 用户评分的物品数量
22
23        # 遍历该用户评分过的所有物品，累加每个物品的交互向量
24        for item_id, actual_rating in rated_items:
25            user_interactions[user_id] +=
26                self.itemInteraction[item_id]
27
28        # 将交互向量的每个分量除以 yj_sum，然后加上对应用户的特征向量
29        user_interactions[user_id] /= y_sums[user_id]
30        user_interactions[user_id] += self.user_features[user_id]
31
32    # 遍历该用户评分过的所有物品，计算预测评分和误差
33    for user_id in range(self.userNum):
34        rated_items = self.trainMatrix[user_id]
35        if not rated_items:
36            continue
37
38        interaction_vector = user_interactions[user_id]
39
40        for item_id, actual_rating in rated_items:
41            predicted_rating = self.predict_rating(user_id, item_id,
42                                                    interaction_vector)
43            error = actual_rating - predicted_rating
44            total_error += error ** 2
45            rating_count += 1
46
47    train_rmse = math.sqrt(total_error / rating_count)
48    print(f"train RMSE: {train_rmse}")
```

在 SVD++ 的 train 训练迭代中, 增加了对物品交互向量 (self.itemInteraction) 的考虑。

遍历每个用户, 预计算当前用户评分的物品数量的平方根 $y_sums[user_id]$, 用户的交互向量 $user_interaction$ 。对每个用户, 遍历其评分过的物品, 累加当前物品 $item_id$ 的交互向量到 $user_interactions[user_id]$ 中, $user_interactions[user_id]$ 中的每个分量除以 $y_sums[user_id]$, 实现归一化。将归一化后的交互向量 $user_interactions[user_id]$ 加上当前用户的特征向量 $self.user_features[user_id]$ 。

遍历每个用户, 如果当前用户有评分记录, 获取当前用户之间计算好的交互想来那个, 循环遍历当前用户评分过的每个物品及实际评分, 并使用 $predict_rating$ 进行评分预测, 计算预测评分与真实评分的误差。

然后更新用户偏差项、物品偏差、用户向量、物品向量和物品交互向量。

具体的更新规则如下:

- 更新用户偏差 (User Bias):

$$b_u^{(t+1)} = b_u^{(t)} + \eta \cdot (e_{ui} - \lambda \cdot b_u^{(t)})$$

$b_u^{(t)}$ 是第 t 次迭代时的用户偏差, η 是学习率, e_{ui} 是实际评分与预测评分之间的误差, λ 是正则化参数。

- 更新物品偏差 (Item Bias):

$$b_i^{(t+1)} = b_i^{(t)} + \eta \cdot (e_{ui} - \lambda \cdot b_i^{(t)})$$

$b_i^{(t)}$ 是第 t 次迭代时的物品偏差, η 是学习率, e_{ui} 是实际评分与预测评分之间的误差, λ 是正则化参数。

- 更新用户特征向量 (User Feature Vector):

$$p_u^{(t+1)} = p_u^{(t)} + \eta \cdot (e_{ui} \cdot q_i^{(t)} - \lambda \cdot p_u^{(t)})$$

$p_u^{(t)}$ 是第 t 次迭代时的用户特征向量, $q_i^{(t)}$ 是第 t 次迭代时的物品特征向量, η 是学习率, e_{ui} 是实际评分与预测评分之间的误差, λ 是正则化参数。

- 更新物品特征向量 (Item Feature Vector):

$$q_i^{(t+1)} = q_i^{(t)} + \eta \cdot (e_{ui} \cdot p_u^{(t)} - \lambda \cdot q_i^{(t)})$$

$q_i^{(t)}$ 是第 t 次迭代时的物品特征向量, $p_u^{(t)}$ 是第 t 次迭代时的用户特征向量, η 是学习率, e_{ui} 是实际评分与预测评分之间的误差, λ 是正则化参数。

•

$$y_i^{(t+1)} = y_i^{(t)} + \eta \left(\frac{e_{ui} \cdot q_i^{(t)}}{y_sums} - \lambda \cdot y_i^{(t)} \right)$$

$y_i^{(t)}$ 是第 t 次迭代的物品交互向量, $q_i^{(t)}$ 是第 t 次迭代时的物品特征向量, η 是学习率, e_{ui} 是实际评分与预测评分之间的误差, y_sums 是归一化因子, λ 是正则化参数。

Listing 22: SVD++

```
2 class SVD_PlusPlus:
3     ...
4     def train(self, train_file, learning_rate, regularization, epochs):
5         print("Preprocessing .....")
6         self.process_data(train_file)
7         self.init_model_parameters()
8         print("Training .....")
9
10        print("epochs:", epochs)
11        print("UserNum: ", self.userNum)
12        t1 = time.time()
13
14        interaction_vector = np.zeros(self.dim)
15
16        # 遍历所有用户
17        for epoch in range(1, epochs + 1):
18            # self.train_epoch(learning_rate, regularization)
19            # 预计算用户的交互向量和y值
20            user_interactions = [np.zeros(self.dim) for _ in
21                                range(self.userNum)]
22            y_sums = [0] * self.userNum
23
24            for user_id in range(self.userNum):
25                rated_items = self.trainMatrix[user_id]
26                if not rated_items:
27                    continue
28
29                # 计算当前用户评分的物品数量
30                y_sums[user_id] = math.sqrt(len(rated_items)) #
31                # 用户评分的物品数量
32                # yj_sum = math.sqrt(len(self.trainMatrix[user_id]))
33                # interaction_vector.fill(0) # 初始化交互向量为全零
34
35                # 遍历该用户评分过的所有物品，累加每个物品的交互向量
36                for item_id, actual_rating in self.trainMatrix[user_id]:
37                    # interaction_vector += self.itemInteraction[item_id]
38                    user_interactions[user_id] +=
39                        self.itemInteraction[item_id]
40
41                # 将交互向量的每个分量除以
42                yj_sum, 然后加上对应用户的特征向量
43                # interaction_vector /= yj_sum
```

```
40         # interaction_vector += self.user_features[user_id]
41         user_interactions[user_id] /= y_sums[user_id]
42         user_interactions[user_id] += self.user_features[user_id]
43
44     for user_id in range(self.userNum):
45         rated_items = self.trainMatrix[user_id]
46         if not rated_items:
47             continue
48
49         interaction_vector = user_interactions[user_id]
50
51         # 遍历该用户评分过的物品
52         for item_id, actual_rating in rated_items:
53             predicted_rating = self.predict_rating(user_id,
54                                                     item_id, interaction_vector) # 预测评分
55             error = actual_rating - predicted_rating # 计算误差
56
57             # 更新用户偏差项物品偏差项
58             self.user_bias[user_id] += learning_rate * (error -
59                                                         regularization * self.user_bias[user_id])
60             self.item_bias[item_id] += learning_rate * (error -
61                                                         regularization * self.item_bias[item_id])
62
63             # 更新用户向量和物品向量
64             self.user_features[user_id] += learning_rate *
65                 (error * self.item_features[item_id] -
66                 regularization * self.user_features[user_id])
67             self.item_features[item_id] += learning_rate *
68                 (error * interaction_vector - regularization *
69                 self.item_features[item_id])
70
71             # 更新物品交互向量
72             for j, _ in self.trainMatrix[user_id]:
73                 self.itemInteraction[j] += learning_rate *
74                     (error * self.item_features[item_id] /
75                     y_sums[user_id] - regularization *
76                     self.itemInteraction[j])
77
78         # 每轮每隔300个用户打印一次信息
79         if user_id % 300 == 0:
80             print(f"Epoch {epoch}, {user_id} users processed
81                   ...")
```

```

71
72     print(f"Epoch {epoch} finished ...")
73     self.calculate_rmse()
74     learning_rate *= 0.9999
75
76     t2 = time.time()
77     print(f"Training time: {t2 - t1} seconds")

```

4.3 推荐算法实验结果

这里假设用户特征向量和物品特征向量的维度是 20，并且使用了物品属性文件，初始学习率是 0.001，正则化参数为 1.5，迭代训练 10 轮。

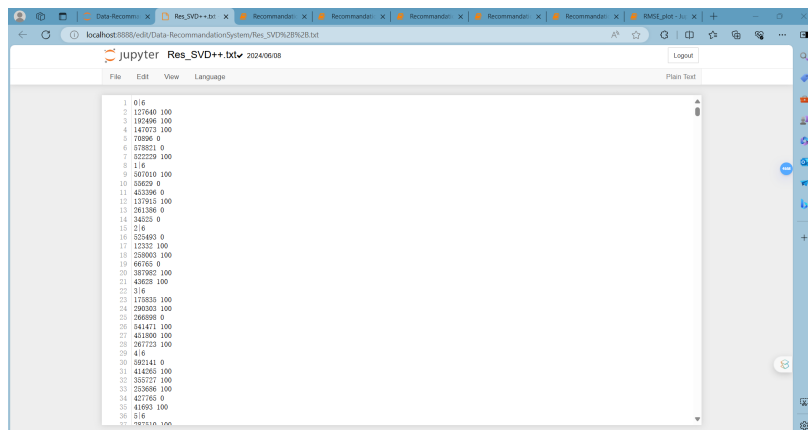
Listing 23: SVD

```

1
2 svd_plus = SVD_PlusPlus(20)
3 svd_plus.train("./Data-RecommandationSystem/train.txt", 0.001, 1.5, 10)

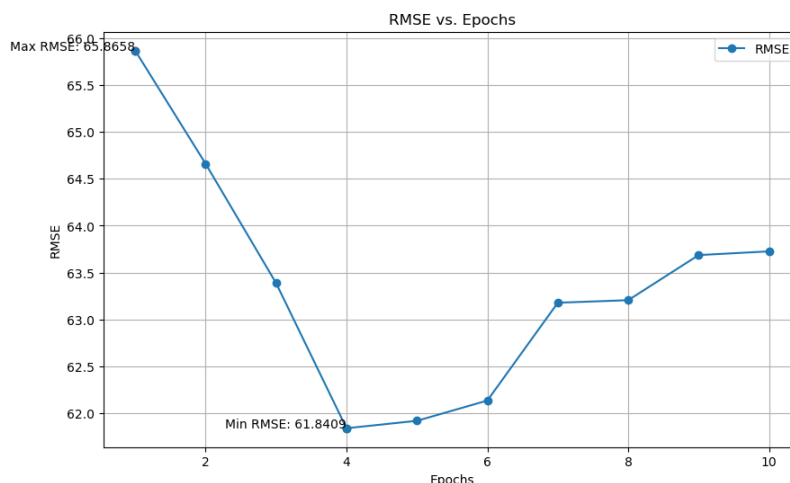
```

预测结果文件 Res_SVD++.txt 如下，可以看到 SVD++ 的预测结果很差：



不仅预测结果很差，而且运行时间很长，大概需要 2-3 个小时。

最开始的 RMSE 值为 65 左右，在训练到第 10 轮后，RMSE 值基本稳定在 63.7 左右。在前 10 轮中，最低的 RMSE 出现在第四轮。



5 UserCF

5.1 UserCF 算法说明

基于用户的协同过滤算法的思想就是，具有类似特征的人群，他们喜欢的东西很多也是一样的。因此，在推荐系统中，假设要为 A 用户推荐物品，可以通过寻找他的“邻居”（与 A 具有相似兴趣的用户）。把那些用户喜欢的，而 A 用户却不曾听说的东西推荐给 A。

UserCF 算法主要包含了两大步骤：

- 第 1 个步骤是，要找到与待推荐用户兴趣相似的用户集合。也就是需要计算目标用户和其余用户的相似度，找到与目标用户最相似的前 n 个用户。
- 第 2 个步骤是，选出这些相似用户喜欢的，并且目标用户没有关注的物品，将它们推荐给目标用户。根据这 n 个用户对某个物品的评分情况与目标用户的相似度猜测目标用户对这个物品的评分，如果评分最终很高，就把这个物品推荐给 A，否则不推荐。

5.1.1 用户相似度度量

- Jaccard 相似系数

Jaccard 相似系数就是两个用户 A 和 B 产生交集的物品数量占两个用户所有物品并集的比例。如果两个人购买的物品交集越多说明两个人的购买欲望相同，两个人的兴趣爱好就越相同。

假设有两个集合 A 和 B，它们的 Jaccard 相似系数 $J(A, B)$ 定义为：

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

其中，

- $|A \cap B|$ 表示集合 A 和 B 的交集的元素个数，
- $|A \cup B|$ 表示集合 A 和 B 的并集的元素个数。

Jaccard 相似系数的取值范围是 $[0, 1]$ ，值越接近于 1 表示两个集合越相似，值越接近于 0 表示两个集合差异越大。

- 皮尔逊相关系数

皮尔逊相关系数 (Pearson correlation coefficient) 是一种衡量两个变量之间线性关系强度和方向的统计量。它用于衡量两个变量之间的线性关系程度, 其取值范围在 $[-1, 1]$ 之间。越接近 1 越是正相关, 越接近 -1 越是负相关, 如果为 0, 则表明二者不相关。

皮尔逊相关系数 $r_{X,Y}$ 衡量了两个数据集 $X = \{x_1, x_2, \dots, x_n\}$ 和 $Y = \{y_1, y_2, \dots, y_n\}$ 之间的线性关系强度。它的定义如下:

$$r_{X,Y} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}}$$

其中, \bar{x} 和 \bar{y} 分别是 X 和 Y 的均值。

皮尔逊相关系数 $r_{X,Y}$ 的取值范围为 -1 到 1 :

- 当 $r_{X,Y} = 1$ 时, 表示 X 和 Y 完全正相关;
- 当 $r_{X,Y} = -1$ 时, 表示 X 和 Y 完全负相关;
- 当 $r_{X,Y} = 0$ 时, 表示 X 和 Y 之间没有线性关系, 但不排除存在其他类型的关系 (如非线性关系)。

5.1.2 计算目标用户对要推荐物品的评分

利用邻居用户的评分数据, 结合其与目标用户的相似度, 对目标用户尚未评分的物品进行预测评分。常用的方法包括加权平均或加权。

用户 u 对物品 i 的预测评分 $\hat{r}_{u,i}$ 可以计算为:

$$\hat{r}_{u,i} = \frac{\sum_{v \in N(u)} \text{sim}(u, v) \cdot r_{v,i}}{\sum_{v \in N(u)} |\text{sim}(u, v)|}$$

其中,

- $N(u)$ 是与用户 u 最相似的一组邻居用户集合,
- $\text{sim}(u, v)$ 是用户 u 和用户 v 之间的相似度,
- $r_{v,i}$ 是邻居用户 v 对物品 i 的实际评分。

如果考虑到用户评分标准不一致的情况, 可以采用评分差值加权平均的方式预测评分。

用户 u 对物品 i 的预测评分 $\hat{r}_{u,i}$ 可以计算为:

$$\hat{r}_{u,i} = \bar{B}_u + \frac{\sum_{k \in K_u} \text{sim}(u, k) \cdot (r_{k,i} - \bar{B}_k)}{\sum_{k \in K_u} \text{sim}(u, k)}$$

其中,

- $\hat{r}_{u,i}$ 用户 u 对物品 i 的预测评分, $r_{k,i}$ 是邻居用户 k 对物品 i 的实际评分
- \bar{B}_u 为考虑了全局、用户和物品三个层面的评分加权后的用户 u 的基准均分, 表示用户 u 对物品的基础偏好, \bar{B}_k 为考虑了全局、用户和物品三个层面的评分加权后的用户 k 的基准均分, 表示用户 k 对物品的基础偏好。
- $\text{sim}(u, k)$ 是用户 u 和用户 k 之间的相似度。
- K_u 是与用户 u 最相似的一组邻居用户的集合

5.2 代码实现

初始化 UserCF 测试集、训练集、物品属性文件、用户数量、最大物品数量及全局平均评分

Listing 24: UserCF

```

1
2 class UserCF:
3     def __init__(self, test_path="./Data-RecommandationSystem/test.txt",
4                 train_path="./Data-RecommandationSystem/train.txt",
5                 attribute_file="./Data-RecommandationSystem/itemAttribute.txt"):
6         self.test_path = test_path # 测试集路径
7         self.train_path = train_path # 训练集路径
8         self.attribute_file = attribute_file # 属性文件路径
9         self.user_num = 0 # 用户数量
10        self.max_item = 0 # 最大物品数量
11        self.global_avg = 0 # 全局平均评分

```

initialize_variables() 初始化最大最小用户 ID、最大最小物品 ID，逐行读取训练集文件，获取用户 ID 和评分数量，更新最大最小用户 ID、最大最小物品 ID 计算用户数量 user_num 和物品数量 max_item。

创建大小为 user_num 的数组记录每个用户的平均评分 self.user_avg，并初始化为 0；创建大小为 max_item 的数组记录每个物品的平均评分 self.item_avg，并初始化为 0；创建大小为 user_num 的数组记录每个用户的评分数量。创建一个长度为 self.user_num 的空列表 self.data，每个元素是一个空列表用于存储每个用户的评分数据，每个评分数据是一个元组 (item_id, rating)。初始化物品属性列表 self.attr_list 存储每个物品的属性信息，后续通过 load_item_attributes 方法读取和填充具体的属性数据。

Listing 25: UserCF

```

1
2 class UserCF:
3     ...
4     # 获取user_num和max_item,并进行初始化
5     def initialize_variables(self):
6         maxUserId = 0
7         minUserId = float('inf')
8         maxItemId = 0
9         minItemId = float('inf')
10
11        with open(self.train_path, 'r') as read_stream:
12            for line in read_stream:
13                if not line.strip():
14                    continue
15                parts = line.strip().split()

```

```

16         userId, rateNum = self.process_header(parts[0])
17         minUserId = min(userId, minUserId)
18         maxUserId = max(userId, maxUserId)
19
20         # 遍历每个用户的评分信息
21         for _ in range(rateNum):
22             itemId, score = map(int,
23                                 read_stream.readline().strip().split()) #
24                                 读取物品ID和评分
25             minItemId = min(itemId, minItemId)
26             maxItemId = max(itemId, maxItemId)
27
28         # 计算用户数量和物品数量
29         self.user_num = maxUserId - minUserId + 1
30         self.max_item = maxItemId - minItemId + 1
31
32         # 初始化
33         self.user_avg = np.zeros(self.user_num) # 用户平均评分
34         self.item_avg = np.zeros(self.max_item) # 物品平均评分
35         self.rating_num = np.zeros(self.user_num, dtype=int) #
36         用户评分数量
37         self.attr_list = [None] * self.max_item # 物品属性列表
38         self.data = [[] for _ in range(self.user_num)] # 用户评分数据
39         print("initialize_variables done ...")

```

process_header() 与前面一致，都是用于处理每个用户数据的第一行，提取用户 ID 和评分数量。

在 process_data() 中读取每个用户的评分数据，将 (item_id, score) 添加到 data[user_id] 中，user_avg[user_id] 累加用户对评分商品的评分以用于遍历完每个用户的评分数据后的每个用户的平均评分的计算，并对 self.data[user_id] 列表中的评分数据按照物品 ID 进行排序。在读完所有用户的数据后，计算全局平均评分 self.global_avg 等于所有评分的总和除以评分的总数量。

Listing 26: UserCF

```

1
2 class UserCF:
3     ...
4     def process_header(self, header):
5         sep_pos = header.find("|") # 找到分隔符的位置
6         user_id = int(header[:sep_pos]) # 提取用户ID
7         rate_num = int(header[sep_pos + 1:]) # 提取评分数量
8         return user_id, rate_num
9
10    # 读取训练集

```

```

11     def process_data(self):
12         total_sum = 0
13         total_count = 0
14         maxUserId = 0
15         minUserId = float('inf')
16         maxItemId = 0
17         minItemId = float('inf')
18
19         with open(self.train_path, 'r') as file:
20             for line in file:
21                 if not line.strip():
22                     continue
23                 # 每个用户的第一行
24                 parts = line.strip().split()
25                 user_id, rate_num = self.process_header(parts[0])
26                 self.rating_num[user_id] = rate_num
27                 total_count += rate_num
28
29                 # 遍历每个用户的评分信息
30                 for _ in range(rate_num):
31                     item_id, score = map(int,
32                                           file.readline().strip().split())
33                     self.data[user_id].append((item_id, score))
34                     self.user_avg[user_id] += score
35                     total_sum += score
36
37                 #
38                 # 在读取完每个用户的所有评分数据将用户评分数据按物品ID排序
39                 self.user_avg[user_id] /= self.rating_num[user_id] #
40                 # 计算用户平均评分
41                 self.data[user_id].sort(key=lambda x: x[0]) #
42                 # 对用户评分数据按物品ID排序
43
44         self.global_avg = total_sum / total_count
45         print("Read train.txt ...")
46         # self.initialize_variables()

```

对 item_id 物品,如果已经计算过该物品的平均评分,则直接返回缓存中的值。如果没有计算过,则遍历所有用户的评分数据,如果当前评分数据中包含 item_id 的评分,则累加当前评分 total 和该物品的评分数量 count, 如果 count 不为 0, 则计算物品的平均评分为 total / count, 存储到 self.item_avg[item_id] 中, 如果 count 为 0 则默认评分为 0。

Listing 27: UserCF

```

1
2 class UserCF:
3     ...
4     def get_item_avg(self, item_id):
5         # 如果已经计算过该物品的平均评分, 则直接返回缓存中的平均评分
6         if self.item_avg[item_id] != 0:
7             return self.item_avg[item_id]
8
9         total = 0 # 初始化物品评分总和
10        count = 0 # 初始化物品评分数量
11
12        # 遍历所有用户
13        for i in range(self.user_num):
14            # 遍历当前用户的所有评分数据
15            for item in self.data[i]:
16                # 如果评分数据中包含目标物品, 则累加评分和评分数量
17                if item[0] == item_id:
18                    total += item[1]
19                    count += 1
20
21        # 计算目标物品的平均评分
22        self.item_avg[item_id] = total / count if count != 0 else 0
23        # 返回目标物品的平均评分
24        return self.item_avg[item_id]

```

load_item_attributes() 从属性文件中读取每个物品的属性信息。逐行读取属性文件, 使用 line.strip().split('|') 方法, 去除每行两端的空白字符并按 | 分割, 得到物品 ID 和属性信息的列表 parts, parts[0] 第一个元素为物品 ID, 第二个元素表示第一个属性值如果是 None 则转换为 0, 否则转成对应的整数, 第三个元素表示第二个属性值如果是 None 则转换为 0, 否则转成对应的整数。计算两个属性值的平方和的根。将物品的属性信息存储为字典形式 'attr1': attr1, 'attr2': attr2, 'norm': norm, 以物品 ID 为键。

Listing 28: UserCF

```

1
2 class UserCF:
3     ...
4     def load_item_attributes(self):
5         with open(self.attribute_file, 'r') as file:
6             # lines = file.readlines()
7             for line in file:
8                 if line.strip() == "":
9                     break

```

```

10         parts = line.strip().split('|')
11         item_id = int(parts[0]) # 获取物品 ID
12         attr1 = int(parts[1]) if parts[1] != "None" else 0 #
            获取第一个属性
13         attr2 = int(parts[2]) if parts[2] != "None" else 0 #
            获取第二个属性
14         norm = np.sqrt(attr1 * attr1 + attr2 * attr2) #
            计算属性的欧几里得范数
15         self.attr_list[item_id] = {'attr1': attr1, 'attr2':
            attr2, 'norm': norm} # 将物品属性信息存储到
            self.attr_list 中的字典中
16         print("Read itemAttribute.txt done ...")

```

计算物品间两个属性向量间的属性相似度，判断物品 item1 和 item2 是否在 self.attr_list 中，如果其中一个物品的属性向量或属性列表为 None，则直接返回相似度为 0。

如果两个物品的属性向量和属性列表都不为 None，则计算两个物品属性向量点积即对应维度的乘积之和，计算两个物品属性向量的乘积，计算属性相似度 attribute_similarity。

Listing 29: UserCF

```

1
2 class UserCF:
3     ...
4     def calculate_attribute_sim(self, item1, item2):
5         # 如果item1或item2的属性列表为None，则相似度0
6         if self.attr_list[item1] is None or self.attr_list[item2] is
            None:
7             return 0
8         # 如果item1或item2的属性向量的范数为0，则相似度0
9         if self.attr_list[item1]['norm'] == 0 or
            self.attr_list[item2]['norm'] == 0:
10             return 0
11         # 计算两个物品属性向量的点积
12         a = self.attr_list[item1]['attr1'] *
            self.attr_list[item2]['attr1'] +
            self.attr_list[item1]['attr2'] *
            self.attr_list[item2]['attr2']
13         # 计算两个物品属性向量范数的乘积
14         b = self.attr_list[item1]['norm'] * self.attr_list[item2]['norm']
15         # 计算属性相似度，即点积除以范数的乘积
16         attribute_similarity = a / b
17         # 返回属性相似度
18         return attribute_similarity

```

计算两个用户基于其评分数据的皮尔逊相关系数，用于推荐系统中基于用户的协同过滤推荐。根据两个用户评分数量确定 user_less 和 user_more 用户。遍历两个用户的评分数据，找到共同评分的物品 ID 集合，如果共同评分集合为空则两个用户相似度为 0。

计算两个用户共同评分物品上评分与各自平均评分的差值的乘积之和，计算两个用户评分差的平方和 sum1 和 sum2 和两个用户评分差的平方和的平方根乘积以计算皮尔逊相关系数。如果分母 denominator 为零，则返回相似度为 0。

Pearson 相关系数 $\rho_{X,Y}$ 是衡量两个变量 X 和 Y 之间线性关系强度和方向的统计量。它的定义如下公式所示：

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y}$$

其中，

- $\text{cov}(X,Y)$ 是 X 和 Y 的协方差，
- σ_X 和 σ_Y 分别是 X 和 Y 的标准差。

协方差 $\text{cov}(X,Y)$ 的计算公式为：

$$\text{cov}(X,Y) = \frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})$$

其中 n 是样本数量， X_i 和 Y_i 是样本中的观察值， \bar{X} 和 \bar{Y} 分别是 X 和 Y 的样本均值。

标准差 σ_X 和 σ_Y 的计算公式为：

$$\sigma_X = \sqrt{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2}$$

$$\sigma_Y = \sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i - \bar{Y})^2}$$

Pearson 相关系数 $\rho_{X,Y}$ 的取值范围是从 -1 到 1 ：

- $\rho_{X,Y} = 1$ 表示 X 和 Y 之间存在完全正线性关系，
- $\rho_{X,Y} = -1$ 表示 X 和 Y 之间存在完全负线性关系，
- $\rho_{X,Y} = 0$ 表示 X 和 Y 之间不存在线性关系。

在推荐系统中，Pearson 相关系数常用于计算用户或物品之间的相似度，用于协同过滤推荐。

在函数 calculate_user_pearson_sim 中，遍历 u 的所有评分记录，对于每条记录 (item_id, score)，如果 item_id 也在 v 的评分记录中，则计算

$$\text{numerator} = \sum_{i \in \text{common_items}} (r_{ui} - \bar{r}_u) \cdot (r_{vi} - \bar{r}_v)$$

其中 r_{ui} 是用户 u 对物品 i 的评分， \bar{r}_u 是用户 u 的平均评分， r_{vi} 是用户 v 对物品 i 的评分， \bar{r}_v 是用户 v 的平均评分

下面计算皮尔逊相关系数的分母部分，计算两个用户评分差的平方和 sum1 和 sum2 和两个用户评分差的平方和的平方根乘积：

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

如果分母不是 0，则计算皮尔逊相关系数

$$\text{similarity} = \frac{\text{numerator}}{\text{denominator}}$$

Listing 30: UserCF

```

1
2 class UserCF:
3     ...
4     # 计算两个用户之间的皮尔逊相似度
5     def calculate_user_pearson_sim(self, user1, user2):
6
7         # 根据两个用户的评分数确定评分数较少的用户和评分数较多的用户
8         user_less = user1 if self.rating_num[user1] <=
            self.rating_num[user2] else user2
9         user_more = user2 if self.rating_num[user1] <=
            self.rating_num[user2] else user1
10
11        # 获取两个用户中评分数较少的数量
12        num = min(self.rating_num[user1], self.rating_num[user2])
13
14        # 初始化皮尔逊相似度的分子部分
15        numerator = 0
16
17        # 存储每个用户评分的物品ID
18        user_less_items = {self.data[user_less][i][0] for i in
            range(self.rating_num[user_less])}
19        user_more_items = {self.data[user_more][j][0] for j in
            range(len(self.data[user_more]))}
20
21        # 找到共同评分的物品ID集合
22        common_items = user_less_items.intersection(user_more_items)
23
24        if not common_items:
25            return 0
26
27        # 计算皮尔逊相似度的分子部分

```



```

28     numerator = sum((self.data[user_less][i][1] -
29                     self.user_avg[user_less]) *
30                     (self.data[user_more][j][1] -
31                     self.user_avg[user_more]))
32     for i in range(self.rating_num[user_less])
33     for j in range(len(self.data[user_more]))
34     if self.data[user_less][i][0] in common_items and
35     self.data[user_more][j][0] in common_items)
36
37     # 计算用户1和用户2评分差的平方和
38     sum1 = sum((self.data[user_less][i][1] -
39                 self.user_avg[user_less]) ** 2 for i in
40                 range(self.rating_num[user_less]))
41     sum2 = sum((self.data[user_more][j][1] -
42                 self.user_avg[user_more]) ** 2 for j in
43                 range(len(self.data[user_more])))
44
45     # 计算用户1和用户2评分差的平方和的平方根
46     denominator = np.sqrt(sum1) * np.sqrt(sum2)
47
48     if denominator == 0:
49         return 0
50
51     # 皮尔逊相似度
52     similarity = numerator / denominator
53     return similarity

```

使用 predict 进行预测。sum_sim 初始化相似度的总和和相似度加权评分的总和。遍历所有用户，对每个非当前用户的用户 i 检查用户数据 `self.data[i]` 是否有目标物品 `item_id` 的评分，如果用户 i 对目标物品有评分，则使用 `calculate_user_pearson_sim` 方法计算 `user_id` 和用户 i 之间的皮尔逊相似度 `sim`。只考虑相似度大于 0 的情况，获取用户 i 对物品 `item_id` 的评分 `rating_i`，计算基准评分，然后累加相似度和加权评分差。

如果相似度 `sim > 0`，则考虑该用户 i 对物品 `item_id` 的评分，并计算其与基准评分之间的差异加权：

$$\text{sum_sim_item} += \text{sim} \times (\text{rating_i} - \text{baseline})$$

其中，`rating_i` 是用户 i 对物品 `item_id` 的评分。

基准评分考虑了全局、用户和物品三个层面的评分偏差。通过加权组合这些因素，可以更精确地调整预测评分，以提高推荐系统的准确性和个性化程度。

$$\text{baseline} = \text{global_avg} + (\text{user_avg}[\text{user_id}] - \text{global_avg}) + (\text{get_item_avg}(\text{item_id}) - \text{global_avg})$$

`global_avg`是所有用户所有评分的平均值, `user_avg[user_id]` 是用户 `user_id` 所有评分平均值, `get_item_avg(item_id)` 是物品 `item_id` 所有评分的平均值。

如果存在相似度大于 0 的相似用户, 则计算加权平均以预测用户对物品的评分。如果找不到任何相似用户则返回用户 `user_id` 的平均评分作为预测值, 并保证评分在 0-100 的范围里。

如果存在至少一个相似用户 i , 则计算预测评分:

$$\text{predict_score} = \text{baseline} + \frac{\text{sum_sim_item}}{\text{sum_sim}}$$

这里将相似用户的加权评分差平均到基准评分上, 以预测用户 `user_id` 对物品 `item_id` 的评分。

`write_predictions` 方法用于生成和保存预测结果到文件, 逐行读取测试集文件, 使用 `predict` 方法预测测试集中每个用户对每个物品的评分。

Listing 31: UserCF

```

1
2 class UserCF:
3     ...
4     def predict(self, user_id, item_id):
5         sum_sim = 0 # 初始化相似度的总和
6         sum_sim_item = 0 # 初始化相似度加权评分总和
7
8         # 遍历所有用户
9         for i in range(self.user_num):
10             if i == user_id: # 跳过当前用户
11                 continue
12
13             # 判断用户i是否有对item_id的评分
14             if any(item[0] == item_id for item in self.data[i]):
15                 # 计算当前用户与用户i的Pearson相似度
16                 sim = self.calculate_user_pearson_sim(user_id, i)
17
18                 if sim > 0: # 只考虑正相似度
19                     # 获取用户i对item_id的评分和基准评分
20                     for item in self.data[i]:
21                         if item[0] == item_id:
22                             rating_i = item[1]
23                             break
24
25                     baseline = self.global_avg + (self.user_avg[i] -
26                                                         self.global_avg) + (self.get_item_avg(item_id) -
27                                                         self.global_avg)
28                     sum_sim += sim # 累加相似度
29                     sum_sim_item += sim * (rating_i - baseline) #

```

累加加权评分差

```
28
29 # 如果相似度总和和加权评分差总和都不为零
30 if sum_sim != 0 and sum_sim_item != 0:
31     # 重新计算基准评分
32     baseline = self.global_avg + (self.user_avg[user_id] -
33                                     self.global_avg) + (self.get_item_avg(item_id) -
34                                                         self.global_avg)
35     # 计算预测评分
36     predict_score = baseline + sum_sim_item / sum_sim
37     # 限制预测评分在0到100之间
38     return min(100, max(0, predict_score))
39
40 else:
41     # 如果没有找到任何相似用户，则返回用户的平均评分
42     return self.user_avg[user_id]
43
44 def write_predictions(self):
45     print("Predicting ...")
46     with open("./Data-RecommandationSystem/Res_UserCF.txt", 'w') as
47         file:
48         with open(self.test_path, 'r') as test_file:
49             user_count = 0
50             for line in test_file:
51                 if not line.strip():
52                     continue
53                 parts = line.strip().split('|')
54                 user_id = int(parts[0])
55                 file.write(f"{user_id}|{len(parts[1:])}\n")
56                 ratings = parts[1:]
57
58                 for _ in ratings:
59                     item_id = int(test_file.readline().strip())
60                     predict_rating = int(self.predict(user_id,
61                                                         item_id))
62                     file.write(f"{item_id} {predict_rating}\n")
63
64                 user_count += 1
65                 # 每预测300个用户输出一提示信息
66                 if user_count % 300 == 0:
67                     print(f"Predicted {user_count} users...")
```

```
65 print("Prediction done ...")
```

5.3 推荐算法实验结果

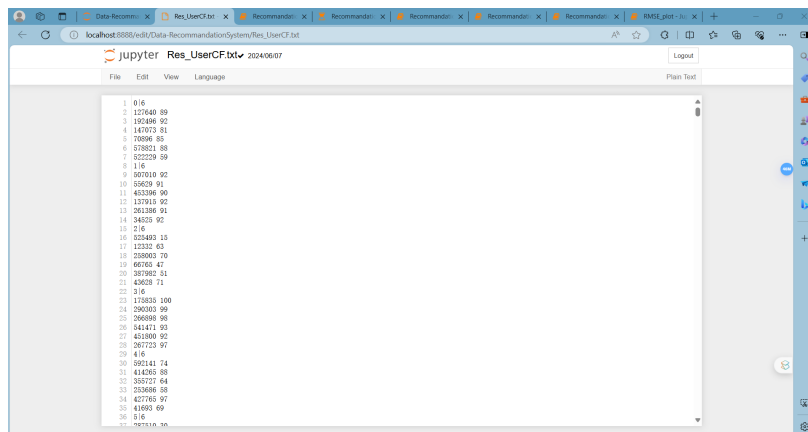
Listing 32: UserCF

```

1
2 class UserCF:
3     ...
4     def run(self):
5         start = time.time()
6         self.initialize_variables()
7         self.process_data()
8         self.write_predictions()
9         end = time.time()
10        print(f"overall running time: {end - start}")
11
12 user_cf = UserCF()
13 user_cf.run()
```

调用 user_cf 的 run() 函数进行训练并预测。

预测结果如下：



该算法运行时间较长，大概 1 小时左右。

6 ItemCF

6.1 ItemCF 算法说明

基于物品的协同过滤算法思想是，具有类似被用户喜欢的物品，可能也会被同一用户喜欢。因此，在推荐系统中，要为用户推荐物品时，可以通过寻找与用户已喜欢物品相似的其他物品进行推荐。

ItemCF 算法主要包含了两大步骤：

- 第 1 个步骤是找到与用户喜欢的物品相似的其他物品集合计算物品之间的相似度。
- 第 2 个步骤是，选出与用户喜欢的物品相似度高、但是用户没有关注的物品，将它们推荐给用户。根据用户对相似物品的评分情况和物品之间的相似度，预测用户对这些物品的评分。

6.1.1 物品相似度度量

- 皮尔逊相关系数

皮尔逊相关系数 (Pearson correlation coefficient) 是一种衡量两个变量之间线性关系强度和方向的统计量。它用于衡量两个变量之间的线性关系程度，其取值范围在 $[-1, 1]$ 之间。越接近 1 越是正相关，越接近 -1 越是负相关，如果为 0，则表明二者不相关。

$$\text{sim}(i, j) = \frac{\sum_{u \in U_{i,j}} (r_{u,i} - \bar{r}_u)(r_{u,j} - \bar{r}_u)}{\sqrt{\sum_{u \in U_{i,j}} (r_{u,i} - \bar{r}_u)^2 \sum_{u \in U_{i,j}} (r_{u,j} - \bar{r}_u)^2}}$$

其中， $U_{i,j}$ 是同时对物品 i 和物品 j 进行评分的用户集合， \bar{r}_u 是用户 u 的平均评分。

6.1.2 计算目标用户对要推荐物品的评分

在 ItemCF 中，预测用户 u 对物品 i 的评分 $\hat{r}_{u,i}$ 可以采用以下公式：

$$\hat{r}_{u,i} = \frac{\sum_{j \in I_u} \text{sim}(i, j) \cdot r_{u,j}}{\sum_{j \in I_u} |\text{sim}(i, j)|}$$

其中，

- I_u 是用户 u 喜欢的物品集合。
- $\text{sim}(i, j)$ 是物品 i 和物品 j 之间的相似度。
- $r_{u,j}$ 是用户 u 对物品 j 的实际评分。

如果考虑到用户评分标准不一致的情况，可以采用评分差值加权平均的方式预测评分。

用户 u 对物品 i 的预测评分 $\hat{r}_{u,i}$ 可以计算为：

$$\hat{r}_{u,i} = \bar{B}_i + \frac{\sum_{j \in N(i,u)} \text{sim}(i, j) \cdot (r_{u,j} - \bar{B}_j)}{\sum_{j \in N(i,u)} \text{sim}(i, j)}$$

其中，

- $\hat{r}_{u,i}$ 用户 u 对物品 i 的预测评分， $r_{u,j}$ 是用户 u 对物品 j 的实际评分
- \bar{B}_i 为考虑了全局、用户和物品三个层面的评分加权后的物品 i 的基准均分，表示物品 i 的基础受欢迎度， \bar{B}_j 为考虑了全局、用户和物品三个层面的评分加权后的物品 j 的基准均分，表示物品 j 的基础受欢迎度。
- $\text{sim}(i, j)$ 是物品 i 和物品 j 之间的相似度。
- $N(i, u)$ 是与物品 i 最相似的一组物品集合，用户 u 对这些物品进行了评分。

6.2 代码实现

这里仅展示 ItemCF 与 UserCF 不同的部分

初始化用户的平均评分，物品的平均评分，用户评分数量，每个物品的被评分次数，物品属性列表，用户评分数据，物品之间相似性的映射关系，训练集中的物品评分数据。

Listing 33: ItemCF

```

1
2 class ItemCF:
3     ...
4     def initialize_variables(self):
5         ...
6         # 初始化
7         self.user_avg = np.zeros(self.user_num) # 用户平均评分
8         self.item_avg = np.zeros(self.num_items) # 物品平均评分
9         self.rating_count_user = np.zeros(self.user_num, dtype=int) #
            用户评分数量
10        self.rating_count_item = np.zeros(self.num_items, dtype=int) #
            每个物品被评分的数量
11        self.attr_list = [None] * self.num_items # 物品属性列表
12        self.data = [[] for _ in range(self.user_num)] # 用户评分数据
13        self.similarity_map = [[] for _ in range(self.num_items)] #
            物品之间相似性的映射关系
14        self.training_data = [[] for _ in range(self.num_items)] #
            存储训练集中的物品评分数据
15        print("initialize_variables done ...")

```

衡量两个物品在用户评分上的相似程度，找到两个物品的共同评分用户，遍历这些共同用户，计算各自物品评分与平均评分之差的乘积之和 (numerator) 和评分差的平方和 (sum_sq1, sum_sq2)，根据皮尔逊相关系数的公式计算相似度。

Listing 34: ItemCF

```

1
2 class ItemCF:
3     ...
4     # 评估两个物品在用户评分上的相似程度
5     def calculate_item_rating_pearson_sim(self, item1, item2):
6         # 确定评分数量较少和较多的物品
7         item_less = item1 if self.rating_count_item[item1] <=
            self.rating_count_item[item2] else item2
8         item_more = item2 if self.rating_count_item[item1] <=
            self.rating_count_item[item2] else item1
9
10        # 获取评分数量的最小值
11        num = min(self.rating_count_item[item1],
            self.rating_count_item[item2])
12
13        # 初始化皮尔逊相关系数的分子和各自物品的评分差值的平方和

```

```
14     numerator = 0
15     sum_sq1 = 0
16     sum_sq2 = 0
17
18     # 如果两个物品中任意一个没有评分数据，则相似度为 0
19     if self.training_data[item_less][0][0] is None or
20        self.training_data[item_more][0][0] is None:
21         return 0
22
23     # 使用集合存储每个物品的用户评分数据
24     item_less_ratings = {user_id: rating for user_id, rating in
25                          self.training_data[item_less]}
26     item_more_ratings = {user_id: rating for user_id, rating in
27                          self.training_data[item_more]}
28
29     # 找出共同评分用户
30     common_users =
31         set(item_less_ratings.keys()).intersection(set(item_more_ratings.keys()))
32
33     for user_id in common_users:
34         rating_less = item_less_ratings[user_id]
35         rating_more = item_more_ratings[user_id]
36
37         # 计算各自物品的评分与平均评分之差的乘积之和
38         numerator += (rating_less - self.item_avg[item_less]) *
39                      (rating_more - self.item_avg[item_more])
40
41         # 计算各自物品评分与平均评分之差的平方和
42         sum_sq1 += (rating_less - self.item_avg[item_less]) ** 2
43         sum_sq2 += (rating_more - self.item_avg[item_more]) ** 2
44
45     # 计算用户1和用户2评分差的平方和的平方根
46     denominator = np.sqrt(sum_sq1) * np.sqrt(sum_sq2)
47
48     # 如果任何一个平方根为 0，则相似度为 0
49     if denominator == 0:
50         return 0
51
52     # 皮尔逊相似度
53     similarity = numerator / denominator
54     return similarity
```

calculate_overall_sim() 结合物品评分的皮尔逊相关系数和物品属性相似度，计算两个物品的总体相似度。将皮尔逊相似度和属性相似度进行加权平均，作为最终的相似度得分。

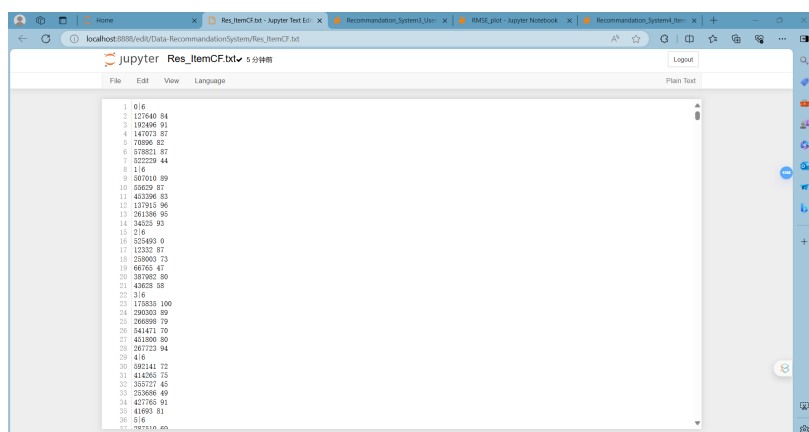
Listing 35: ItemCF

```
1
2 class ItemCF:
3     ...
4     # 综合计算两种相似度
5     def calculate_overall_sim(self, item1, item2):
6         # 确定两个物品的较小和较大的编号
7         min_item = min(item1, item2)
8         max_item = max(item1, item2)
9
10        #
11        检查是否已经计算过这两个物品的相似度，如果计算过则直接返回缓存中的值
12        for sim_item in self.similarity_map[min_item]:
13            if sim_item['id'] == max_item:
14                return sim_item['score']
15
16        # 如果缓存中没有，则计算两个物品的皮尔逊相似度和属性相似度
17        pearson_sim = self.calculate_item_rating_pearson_sim(item1,
18            item2)
19        attribute_sim = self.calculate_attribute_sim(item1, item2)
20
21        # 综合计算得到最终的相似度得分
22        sim_score = (pearson_sim + attribute_sim) / 2
23
24        # 将计算结果存入相似度缓存
25        self.similarity_map[min_item].append({'id': max_item, 'score':
26            sim_score})
27
28        # 返回计算得到的相似度得分
29        return sim_score
```

6.3 推荐算法实验结果

调用 item_cf 的 run() 函数进行训练并预测。

预测结果如下：



```
1 0.6  
2 127640 84  
3 195496 91  
4 147073 87  
5 70996 82  
6 579821 87  
7 522229 44  
8 1.6  
9 567018 89  
10 50629 87  
11 433396 83  
12 127915 86  
13 561386 95  
14 24535 83  
15 2.6  
16 625493 9  
17 12352 87  
18 528003 73  
19 66765 87  
20 387982 89  
21 43628 58  
22 3.6  
23 173633 100  
24 290303 89  
25 266998 79  
26 541471 79  
27 401800 80  
28 267723 84  
29 4.6  
30 982141 72  
31 414263 75  
32 385727 45  
33 253698 49  
34 427165 91  
35 41693 81  
36 9.6  
37 767116 69
```

该算法运行时间较长，大概 1 小时左右。