智能推荐系统第一次作业：

基于协同过滤的用户评分预测

张宏伟

10205102417

2023 年 4 月 7 日

# 目录

1. [算法介绍](#_bookmark0)

**[1.1](#_bookmark1)** [基于用户的协同过滤算法](#_bookmark1) 2

**[1.2](#_bookmark2)** [基于物品的协同过滤算法](#_bookmark2) 2

1. [核心代码注解](#_bookmark3)

**[2.1](#_bookmark4)** [验证集划分](#_bookmark4) 3

**[2.2](#_bookmark5)** [相似度计算](#_bookmark5) 4

**[2.3](#_bookmark6)** [用户评分预测](#_bookmark6) 7

**[2.4](#_bookmark6)** [预测结果可视化 8](#_bookmark6)

1. [结果分析](#_bookmark7)

**[3.1](#_bookmark18)** [基本公式](#_bookmark18) 8

**[3.2](#_bookmark20)** [算法扩展相似度](#_bookmark20) 10

**[3.3](#_bookmark21)** [退化成均值](#_bookmark21) 10

**[3.4](#_bookmark22)** [UCF、ICF 结合](#_bookmark22) 11

1. [代码运行](#_bookmark24)

**[4.1](#_bookmark25)** [环境](#_bookmark25) 12

**4.2** [运行](#_bookmark26) 13

1. [提交文件列表](#_bookmark27)

**[5.1 C](#_bookmark28)ODE** 13

# 算法介绍

协同过滤（Collaborative Filtering，CF）的核心想法是利用某兴趣相投、拥有共同经验之群体的喜好来推荐用户感兴趣的信息。通过分析目标用户与其他用户的相似度、目标物品与其他物品的相似度，过滤出相似的用户和物品，通过他们的历史行为，帮助预测目标用户对目标物品的喜爱程度。过滤不一定局限于特别相似的，特别不相似信息的记录也相当重要。

协同过滤分为两种，一种是基于用户的协同过滤（User-Based Collaborative Filtering，UCF），另一种是基于物品的协同过滤（Item-Based Collaborative Filtering，ICF）

### 基于用户的协同过滤算法

**UCF** 是通过不同用户对同一件物品的评分来评测用户之间喜好的相似性，基于用户之间的相似性做

出推荐，这种算法给用户推荐和他兴趣相投的其他用户喜欢的物品。

基于用户的协同过滤步骤如下：

1. 给定用户集合 𝑆𝑈、物品集合 𝑆𝐼 、𝑢𝑠𝑒𝑟 − 𝑖𝑡𝑒𝑚 评分矩阵 𝑚𝑎𝑡 [|𝑆𝑈 |] [|𝑆𝐼 |]， 对 𝑆𝑈 中的每一对用户(𝑢𝑎, 𝑢𝑏)，找到他们都评过分的共同物品，加入集合 𝑆𝑖；
2. 考虑𝑢𝑎、𝑢𝑏的个人评分均值和对 𝑖𝑡𝑒𝑚 ∈ 𝑆𝑖 的评分，计算 𝑢𝑎、𝑢𝑏 的相似度，可采用计算等方法瘬 构造相似度矩阵 𝑠𝑖𝑚 [|𝑆𝑈 |] [|𝑆𝑈 |]；
3. 给定目标用户 𝑢 和目标物品 𝑖，对于 𝑘 ∈ 𝑆𝑈，𝑘 ≠ 𝑢，如果 𝑘 评价过目标物品 𝑖，那么将 𝑘 加入 𝑢 的邻居集合 𝑆𝑛；

4. 对于 𝑛 ∈ 𝑆𝑛，用 𝑠𝑖𝑚 [𝑢] [𝑛] 和 𝑚𝑎𝑡 [𝑛] [𝑖]，结合 𝑢 的个人评分习惯，来预测 𝑚𝑎𝑡 [𝑢] [𝑖]，可采用平均值、加权平均值等方法。

### 基于物品的协同过滤算法

**ICF**通过用户对不同物品的评分来评测物品之间的相似性，基于物品之间的相似性做出推荐，即，给用户推荐和他之前喜欢的物品相似的物品。

基于物品的协同过滤步骤如下：

1. 给定用户集合 𝑆𝑈、物品集合 𝑆𝐼 、𝑢𝑠𝑒𝑟 −𝑖𝑡𝑒𝑚 评分矩阵 𝑚𝑎𝑡 [|𝑆𝑈 |] [|𝑆𝐼 |]，对 𝑆𝐼 中的每一对物品 (𝑖𝑎, 𝑖𝑏)，找到用户 𝑢𝑠𝑒𝑟，满足 𝑢𝑠𝑒𝑟 给 𝑖𝑎、𝑖𝑏 都打过分，将这样的 𝑢𝑠𝑒𝑟 加入集合 𝑆𝑢；
2. 对于 𝑢𝑠𝑒𝑟 ∈ 𝑆𝑢，根据 𝑚𝑎𝑡 [𝑢𝑠𝑒𝑟] [𝑖𝑎] 和 𝑚𝑎𝑡 [𝑢𝑠𝑒𝑟] [𝑖𝑏]，计算 𝑖𝑎、𝑖𝑏 的相似度，可采用计算Cosine Similarity（余弦相似度）的方法，构造相似度矩阵 𝑠𝑖𝑚 [|𝑆𝑈 |] [|𝑆𝑈 |]；
3. 给定目标用户 𝑢 和目标物品 𝑖，对于 𝑘 ∈ 𝑆𝐼 ，𝑘 ≠ 𝑖，如果 𝑢 评价过 𝑘，那么将 𝑘 加入 𝑖 的邻居集合𝑆𝑛；

4.对于 𝑛 ∈ 𝑆𝑛，用 𝑠𝑖𝑚 [𝑖] [𝑛] 和 𝑚𝑎𝑡 [𝑢] [𝑛]，来预测 𝑚𝑎𝑡 [𝑢] [𝑖]，可采用平均值、加权平均值等方法。

# 核心代码注解

本节解释代码的核心部分，所列函数非完整代码。主要代码语言为 Python，考虑到计算相似度的部

分由于复杂度较高，用 Python 计算时间较长，不方便调试，后改用 C++ 编写。

### 验证集划分

函数定义：split\_valid\_data(self, path, to\_path, rate, round):

参数含义：

path,：train.csv路径

to\_path：划分好的验证集输出文件路径

rate：验证集条目数占train.csv 总条目数比例

round：改变验证集挑选起点

引入用户活跃度和物品活跃度。用户活跃度定义为该用户评分过的物品总数，物品活跃度定义为给该

物品评过分的用户总数。

为了保证训练数据的完整性，先读入train.csv，构造user\_item评分矩阵，并计算用户、物品活跃度。只挑选用户活跃度和物品活跃度均不低于5的用户-物品-评分条目，选入验证集，并从训练集中去除，更新用户、物品活跃度。

1 unum, inum = self.user\_dict[i[0]], self.item\_dict[i[1]]

# 保证有足够的训练集

if ( self.user\_active[unum] >= 5 and self.item\_active[inum] >= 5 ):

# 写入验证集

writer.writerow(i)

# 从训练集中去除

self.user\_active[unum] -= 1

self.item\_active[inum] -= 1

self.matrix[unum][inum] = 0

cnt += 1

if ( cnt == total ): # 收集满total条验证集

break

由于下一步计算相似度转入C++，划分完训练集后需要将uesr-item评分矩阵写入 .csv 文件中。

# 输出 ui\_matrix

matwriter = csv.writer(open(matrix\_path,'w',newline=''))

for i in range (len(self.matrix)):

matwriter.writerow(self.matrix[i])

### 相似度计算

由于相似度计算的复杂度为 𝑂 (𝑚2𝑛) （对于UCF，𝑚 为用户数量，𝑛 为物品数量；对于 ICF，𝑚 为

物品数量，𝑛 为用户数量），试过用 Python计算时间较长，因此改成C++。

由于 UCF、ICF 的相似度计算过程类似，只是循环的含义不同，此处仅解释 UCF 的用户相似度计算过程。

首先需要读入划分完验证集之后在Python中计算得到的user\_item\_matrix、用户打分平均值user\_ave、

物品得分平均值item\_ave.

调用函数：

read\_matrix(string path)

void read\_list(string path, double \* dst){

ifstream inFile(path, ios::in);

if ( ! inFile ) {

printf("Open file failed. \n");

return;

}

int i = 0;

string line, field;

while (getline(inFile, line)){// 按行读取CSV文件中的数据

string field;

istringstream sin(line); // 将整行字符串line读入到字符串流sin中

getline(sin, field, ',');

dst[i] = atof(field.c\_str());

++ i;

}

inFile.close();

cout << "total lines: " << i << endl;

cout << "finish reading" << endl;

}

然后计算用户相似度(以余弦相似度为例)。

函数定义：user\_sim\_cal\_cosine(string to\_path)

参数含义：to\_path：保存相似度矩阵的文件路径

void user\_sim\_cal\_cosine(string to\_path) {

ofstream outFile(to\_path, ios::out);

if (!outFile) {

printf("Open file failed. \n");

return;

}

for (int ua = 0; ua < U; ++ua) {

sim[ua][ua] = 1; // 自己和自己的相似度为1

for (int ub = ua + 1; ub < U; ++ub) {

double numerator = 0, denominator\_a = 0, denominator\_b = 0;

int simcnt = 0;

for (int item = 0; item < I; ++item) {

double r\_ai = mat[ua][item];

double r\_bi = mat[ub][item];

if (r\_ai > 0 && r\_bi > 0) { // a和b同时给i打过分

++simcnt;

numerator += r\_ai \* r\_bi;

denominator\_a += r\_ai \* r\_ai;

denominator\_b += r\_bi \* r\_bi;

}

}

if (simcnt > 0) {

sim[ua][ub] = numerator / (sqrt(denominator\_a) \* sqrt(denominator\_b));

sim[ub][ua] = sim[ua][ub];

}

else {

sim[ua][ub] = sim[ub][ua] = 0; // 没有共同打分

}

}

for (int ub = 0; ub < U; ++ub) { // 写入一行

outFile << sim[ua][ub];

outFile << (ub == U - 1 ? '\n' : ',');

}

}

outFile.close();

cout << "finish writing similarity" << endl;

}

对与其他相似度计算公式也进行了尝试：

**Pearson相似度**

void user\_sim\_cal\_pearson(string to\_path){

for ( int ua = 0; ua < U; ++ ua ){

sim[ua][ua] = 1; // 自己和自己的相似度为1

for ( int ub = ua + 1; ub < U; ++ ub ){

double r\_a = user\_ave[ua];

double r\_b = user\_ave[ub];

double E = 0, Sa = 0, Sb = 0;

int simcnt = 0;

for ( int item = 0; item < I; ++ item ){

double r\_ai = mat[ua][item];

double r\_bi = mat[ub][item];

if ( r\_ai > 0 && r\_bi > 0 ){ // a和b同时给i打过分

++ simcnt;

E += ( r\_ai - r\_a ) \* ( r\_bi - r\_b );

Sa += ( r\_ai - r\_a ) \* ( r\_ai - r\_a );

Sb += ( r\_bi - r\_b ) \* ( r\_bi - r\_b );

}

}

if ( simcnt > 0 && Sa \* Sb != 0 ){

sim[ua][ub] = E / sqrt(Sa \* Sb);

}else{ // 没有共同打分或某一用户所有商品打分都一样

sim[ua][ub] = 0;

}

sim[ub][ua] = sim[ua][ub];

}

}

}

**Improved Cosine Similarity (User-IIF)**

void user\_sim\_cal\_user\_iif(string to\_path) {

vector<int> item\_cnt(I, 0);

for (int item = 0; item < I; ++item) {

for (int user = 0; user < U; ++user) {

if (mat[user][item] > 0) {

++item\_cnt[item];

}

}

}

for (int ua = 0; ua < U; ++ua) {

sim[ua][ua] = 1; // 自己和自己的相似度为1

double u\_a\_iif = 0;

for (int item = 0; item < I; ++item) {

if (mat[ua][item] > 0) {

u\_a\_iif += log(1 + U / item\_cnt[item]);

}

}

for (int ub = ua + 1; ub < U; ++ub) {

double numerator = 0, denominator\_a = 0, denominator\_b = 0;

int simcnt = 0;

double u\_b\_iif = 0;

for (int item = 0; item < I; ++item) {

double r\_ai = mat[ua][item];

double r\_bi = mat[ub][item];

if (r\_ai > 0 && r\_bi > 0) { // a和b同时给i打过分

++simcnt;

numerator += (r\_ai - u\_a\_iif) \* (r\_bi - u\_b\_iif);

denominator\_a += pow(r\_ai - u\_a\_iif, 2);

denominator\_b += pow(r\_bi - u\_b\_iif, 2);

}

}

if (simcnt > 0) {

denominator\_a = sqrt(denominator\_a);

denominator\_b = sqrt(denominator\_b);

double sim\_value = numerator / (denominator\_a \* denominator\_b);

sim[ua][ub] = sim\_value;

sim[ub][ua] = sim\_value;

}

else {

sim[ua][ub] = sim[ub][ua] = 0; // 没有共同打分

}

}

}

}

### 用户评分预测

同样以 UCF为例。按相似度由高到低遍历所有邻居，如果该邻居给目标物品打过分，那么记录相关

数值。遍历完后，根据相应公式计算预测值。将预测出来的评分限制到【1，5】的区间。

def predict\_1rating(self, unum, inum):

'''

根据共现矩阵，计算一次rating

'''

r\_u = self.user\_ave\_score[unum]

total, weigh = 0, 0

# i := neighbor index

cnt = 0

for u in range(len(self.user\_dict)): # 遍历所有用户

unum\_b = int(self.neighbor[unum][u]) # 当前考察的邻居

if ( unum == unum\_b ):

continue

r\_bi = self.matrix[unum\_b][inum]

if ( r\_bi != 0 ): # 邻居给inum打过分

r\_b = self.user\_ave\_score[unum\_b] # 该邻居打分均值

total += ( r\_bi - r\_b ) \* self.sim[unum][unum\_b]

weigh += self.sim[unum][unum\_b]

cnt += 1

# 调整 top-K 邻居 ###############################

if ( cnt == 1000 ): # 凑齐 cnt 个邻居

break

if ( weigh == 0 ):

pred\_ui = self.all\_user\_ave

self.disable += 1

else:

pred\_ui = r\_u + total / weigh

if ( pred\_ui < 1 ):

pred\_ui = 1

if ( pred\_ui > 5 ):

pred\_ui = (self.all\_user\_ave+5)/2

return np.squeeze(pred\_ui)

### **2.4**预测结果可视化

为了便于观察预测结果的分布情况了，在测试集上的拟合效果，将预测的结果可视化

核心代码：

# 可视化预测结果

plt.style.use('ggplot')

plt.figure(figsize=(15, 4))

plt.scatter([i for i in range(len(predicted\_list))], gt\_rate\_list, alpha=0.8, label='rate')

plt.scatter([i for i in range(len(predicted\_list))], predicted\_list, alpha=0.8, label='predicted rate')

plt.legend(loc=[1, 1], fontsize=10)

plt.title('Prediction results on the validation set (RMSE: %.5f)'%RMSE)

plt.xlabel('index')

plt.ylabel('score')

# 结果分析

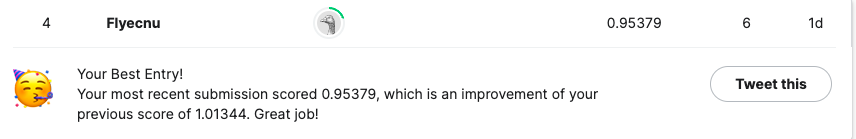
尝试了UCF、ICF 的不同公式、参数、组合。榜单上的提交大致如下：

尝试了User-IIF 、Pearson correlation、 Cosine Similarity不同相似度进行计算

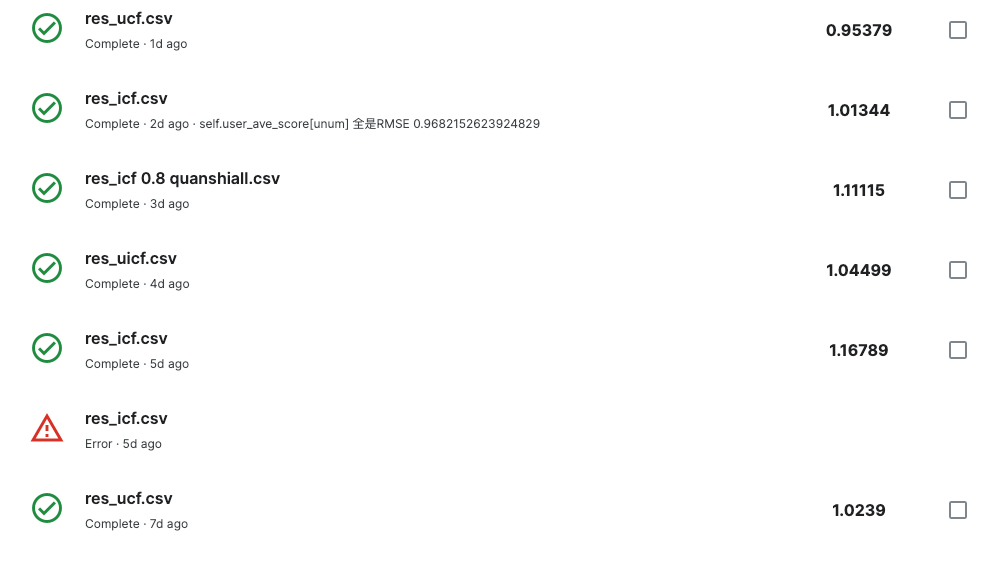
尝试了用Flyod 对相似度矩阵进行优化

因提交次数有限，未对本地所有预测结果进行提交

**最好结果：**Score: 0.95379



**提交列表**



下面将预测分为4部分作进一步的说明。

### 基本公式

前1,2,6次提交尝试了UCF、ICF及其基本的相似度计算公式。

对 UCF来说，相似度计算采用Cosine Similarity 来计算。第 [1](#_bookmark8)次使用Cosine Similarity作为相似度，考虑最相似的 3 位邻居；第6次使用余弦相似度，考虑所有邻居，按相似度加权平均，并进行Flyod优化。

对ICF来说，相似度计算可以采用余弦相似度（提交 2 ），本地尝试了课件上另有一个降低活跃用户权重的余弦相似度计算公式。

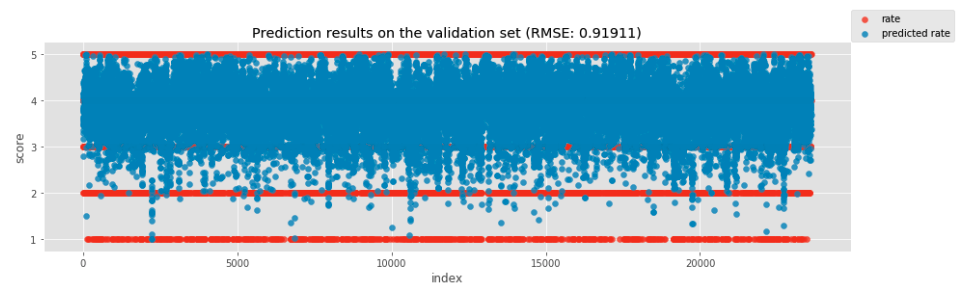


图1第6次提交在验证集上的预测结果

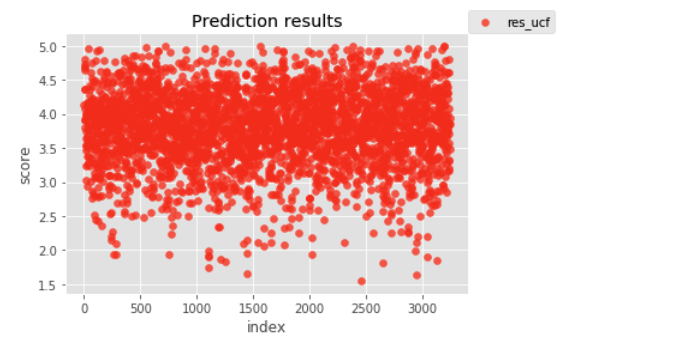


图2: 第6次提交在测试集上的预测结果

这 3 次提交总得来说差别较大，预测点比较分散，能涵盖 [1*,* 5] 的评分取值区间。同时预测错误带来的误差比较大。

* 1. **Floyd** 算法扩展相似度

考虑到数据集较小，相似度矩阵十分稀疏，想尝试能否按照“A和B 相似，B 和 C 相似，那么 A 和C 相似” 以及“A 和 B 相似，B 和 C 不相似，那么 A 和 C 不相似” 的逻辑来填充相似度矩阵。

在 Flyod最短路径算法的基础上修改，用sim[A][C] \* sim[C][B]; 来更新 sim[A][B] 。

代码如下：

for ( int k = 0; k < U; ++ k )

for ( int i = 0; i < U; ++ i )

for ( int j = 0; j < U; ++ j )

if ( sim[i][j] == 0 && sim[i][k] \* sim[k][j] > 0 )

sim[i][j] = sim[i][k] \* sim[k][j];

使用这种方法确实能使得相似度矩阵不那么稀疏，但是相似度相乘后得到的相似度往往值很小，参考

价值不大。但根据在kaggle上的测试结果，这种方法对预测准确度有一定提高。

### 退化成均值

由于相似度矩阵比较稀疏，且尝试填充没有起到较好的效果，只能考虑公式的退化。对于大部分用户、物品之间计算出的相似度是算不出来的，找不到可以使用的邻居，预测全部采用用户个人打分均值或所有用户打分均值。

另外，由于数据集太小，可以认为共同交互比较少的用户或物品之间算出来的相似度并不准确，不予

考虑，仍旧退化成均值。

第4，5次提交中对共同交互的次数做了一个限制：对于两个物品，如果没有超过5 5个用户同时给它们打过分

那么就认为无法计算它们之间的相似度。

这一约束导致验证集瘯测试集上只有个别点采用了预测，其他都由均值填充。

和图[1](#_bookmark19)相比，相当于几乎没做预测，不符合数据集整体的分布规律， RASE的值比较大。

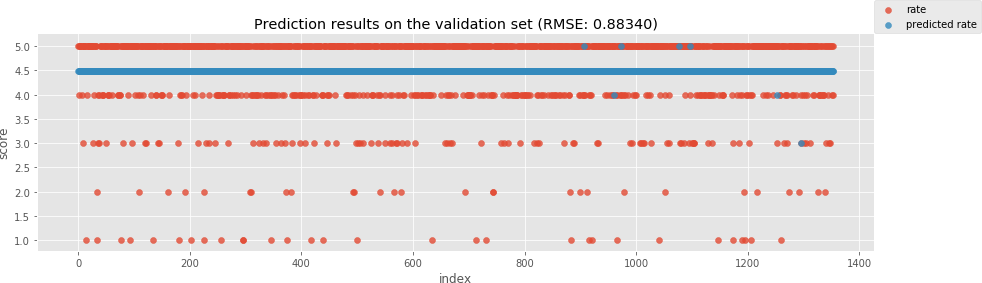


图3:第5次提交在验证集上的预测结果

## UCF-ICF 结合

最后的第[4](#_bookmark17)次提交尝试了综合上述提交，UCF 与 ICF 结合的方法。用户相似度的计算采用 Pearson correlation，物品相似度的计算采用余弦相似度，计算相似度时，共同交互至少为2，认为能够计算相似度。 UCF的预测将相似度阈值定在 0.1（考虑到基于用户的预测是在用户个人打分均值基准上浮动，相似度小的用户影响本身就比较小）， ICF 的预测将相似度定在0.6。无法获得有效邻居的退化成所有用户打分均值。最后将 UCF得到的预测值和 ICF 得到的预测值相加取平均。

结合 UCF、ICF 后，可以从图 [4](#_bookmark23) 看出，本地预测效果较好，但是实际打分结果不理想。

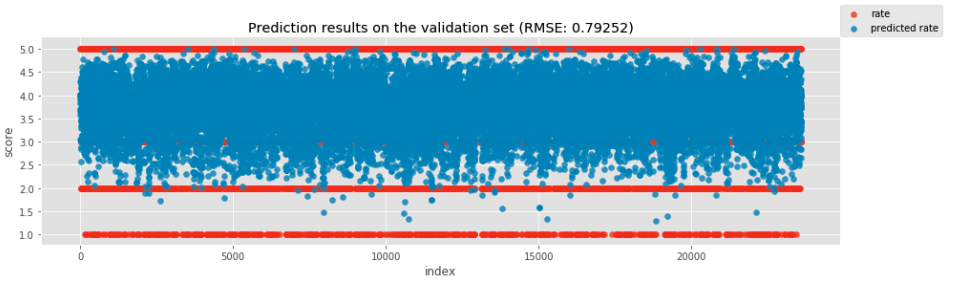


图4: 第4次提交在验证集上的预测结果

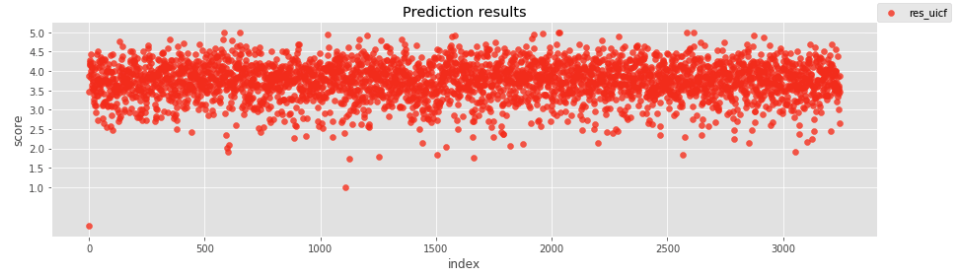


图5:第¥次提交在测试集上的预测结果

# 代码运行

### 环境

PYthon环境：

* + - Anaconda 3.5.2-Windows-x86-64
    - Python 3.7

C++环境：

* + - gcc vcrsion 8.2.0

运行文件夹目录：

* + - Assign-1-data

-train.csv

-text.csv

* + - Assign-1-result
    - usimcal.cpp
    - isimcal.cpp
    - ucf.ipynb
    - iucf.ipynb
    - icf.ipynb

### 运行

代码以.ipynb 和 .cpp 格式的文件提交。

**UCF** 运行方式：

依次执行ucf.ipynb中的 cell，完成导入数据集、划分验证集、计算平均值等工作。

直到写有“# 计算 similarity C++”的cell，去运行usimcal.cpp，计算并导出用户相似度矩阵。

然后回到ucf.ipynb，继续执行cell，完成导入相似度矩阵、邻居排序，以及在验证集上验证、在测试

集上测试推荐效果，并且可视化推荐结果。

**ICF** 运行方式：

依次执行icf.ipynb中的 cell，完成导入数据集、划分验证集、计算平均值等工作。

直到写有“# 计算 similarity C++”的cell，去运行isimcal.cpp，计算并导出用户相似度矩阵。

然后回到icf.ipynb，继续执行cell，完成导入相似度矩阵、邻居排序，以及在验证集上验证、在测试

集上测试推荐效果，并且可视化推荐结果。

**UICF** 运行方式：

依次执行uicf.ipynb中的 cell，完成导入数据集、划分验证集、计算平均值等工作。

直到写有“# 计算 similarity C++”的cell，去运行usimcal.cpp和isimcal.cpp，计算并导出用户相似度矩阵。

然后回到uicf.ipynb，继续执行cell，完成导入相似度矩阵、邻居排序，以及在验证集上验证、在测试

集上测试推荐效果，并且可视化推荐结果。

备注：

* 相似度矩阵文件导出、导入路径需匹配；
* simcal.cpp 中有多个计算相似度的函数可选，代表不同的计算方法；
* 训练集不改变的话，调整预测用的相似度阈值，不需要重新计算相似度矩阵；
* 代码中涉及到修改预测所用阈值的部分已框出。

# 提交文件列表

## Code

* + - ucf.ipynb：基于用户的协同过滤算法代码
    - icf.ipynb：基于物品的协同过滤算法代码
    - uicf.ipynb：基于用户、物品混合的协同过滤算法代码
    - usimcal.cpp：计算用户相似度矩阵的代码
    - isimcal.cpp：计算物品相似度矩阵的代码

## SimMatrix

* + - usim.csv：用户相似度矩阵
    - isim.csv：物品相似度矩阵