## 推荐系统入门（二）：协同过滤（附代码）

## 引言

协同过滤（Collaborative Filtering）算法是推荐系统中最早诞生的算法之一，并且在推荐系统领域也是使用最多最普遍的推荐算法，协同过滤的原理是**首先给目标用户找出其相似用户，通过相似用户的喜好进而挖掘出目标用户的喜好**。这里面其实就涉及到协同过滤算法的两个主要的功能，**其一是目标用户找出相似用户的过程，这个过程可以称为预测过程，其二是相似用户的喜好挖掘目标用户的喜好，这个过程可以称为推荐过程**，在预测过程我们通过会使用目标用户的行为及日志等用户信息去挖掘目标用户的**偏好模型**，然后通过这个偏好模型再去找出相似带有该偏好模型的用户，而在推荐过程我们会采用这些带有相似偏好模型的用户喜好的物品推荐给**目标用户**。其实这个过程可以用日常生活中的**例子**来阐述，我们去看电影的时候，会经常遇到看什么电影的问题，当看到电影传单上所有的电影时，我们又很难确定看哪个电影，而这个时候我们通常会问朋友或同学，通过去询问或者打听看哪部电影，而通常问的这个朋友或者同学是兴趣爱好相似比较信任的。这个过程可以很生动的描述推荐系统的核心思想。

所谓协同过滤，基本思想是**根据用户之前的喜好以及其他兴趣相近的用户的选择来给用户推荐物品**(基于对用户历史行为数据的挖掘发现用户的喜好偏向， 并预测用户可能喜好的产品进行推荐)，**一般是仅仅基于用户的行为数据（评价、购买、下载等）, 而不依赖于项的任何附加信息（物品自身特征）或者用户的任何附加信息（年龄，** **性别等）**。目前应用比较广泛的协同过滤算法是**基于邻域的方法**， 而这种方法主要有下面两种算法：

* **基于用户的协同过滤算法(UserCF)**: 给用户推荐和他兴趣相似的其他用户喜欢的产品
* **基于物品的协同过滤算法(ItemCF)**: 给用户推荐和他之前喜欢的物品相似的物品

不管是UserCF还是ItemCF算法，非常重要的步骤之一就是计算用户和用户或者物品和物品之间的相似度，所以下面先整理常用的相似性度量方法，然后再对每个算法的具体细节进行展开。

推荐系统最核心的部分是推荐算法，推荐算法的不同则会导致推荐系统中推荐的准确性发生偏差。对于协同过滤推荐算法，其流程如下图所示，主要分为三步：**第一，收集用户偏好，第二，找到相似的用户和物品，第三，形成相似推荐**。

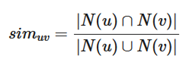
图 协同过滤算法的流程

图 协同过滤算法的流程

## 1. 相似性度量方法

### 1.1 杰卡德(Jaccard)相似系数

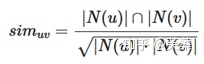
这个是衡量两个集合的相似度一种指标。两个用户u和v交互商品交集的数量占这两个用户交互商品并集的数量的比例，称为两个集合的杰卡德相似系数，用符号simu,v*表示，其中N(u),N(v)分别表示用户u和用户v交互商品的集合。*



由于杰卡德相似系数一般无法反映具体用户的评分喜好信息， 所以常用来评估用户是否会对某商品进行打分，而不是预估用户会对某商品打多少分。

### 1.2 余弦相似度

余弦相似度衡量了两个向量的夹角，夹角越小越相似。首先从集合的角度描述余弦相似度，相比于Jaccard公式来说就是分母有差异，不是两个用户交互商品的并集的数量，而是两个用户分别交互的商品数量的乘积，公式如下：



从向量的角度进行描述，令矩阵A为用户-商品交互矩阵(因为是TopN推荐并不需要用户对物品的评分，只需要知道用户对商品是否有交互就行)，即矩阵的每一行表示一个用户对所有商品的交互情况，有交互的商品值为1没有交互的商品值为0，矩阵的列表示所有商品。若用户和商品数量分别为m，n的话，交互矩阵A就是一个m行n列的矩阵。此时用户的相似度可以表示为(其中u· v指的是[向量点积](https://www.zhihu.com/search?q=%E5%90%91%E9%87%8F%E7%82%B9%E7%A7%AF&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank))：

IMG_259

上述用户-商品交互矩阵在现实情况下是非常的稀疏了，为了避免存储这么大的稀疏矩阵，在计算用户相似度的时候一般会采用集合的方式进行计算。理论上向量之间的相似度计算公式都可以用来计算用户之间的相似度，但是会根据实际的情况选择不同的用户相似度度量方法。

这个在具体实现的时候， 可以使用cosine\_similarity进行实现：

from sklearn.metrics.pairwise import cosine\_similarity

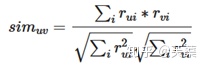
i = [1, 0, 0, 0]

j = [1, 0.5, 0.5, 0]

consine\_similarity([a, b])

### 1.3 皮尔逊相关系数

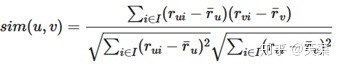
皮尔逊相关系数的公式与余弦相似度的计算公式非常的类似，首先对于上述的余弦相似度的计算公式写成求和的形式,其中r*ui*，*r*vi分别表示用户u和用户v对商品i是否有交互(或者具体的评分值)：



如下是皮尔逊相关系数计算公式，其中r*ui*，*r*vi*分别表示用户u和用户v对商品i是否有交互(或者具体的评分值)，*

IMG_261

分别表示用户u和用户v交互的所有商品交互数量或者具体评分的平均值。



所以相比余弦相似度，皮尔逊相关系数通过使用用户的平均分对各独立评分进行修正，减小了用户评分偏置的影响。具体实现， 我们也是可以调包， 这个计算方式很多， 下面是其中的一种：

from scipy.stats import pearsonr

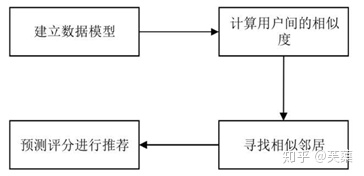
i = [1, 0, 0, 0]

j = [1, 0.5, 0.5, 0]

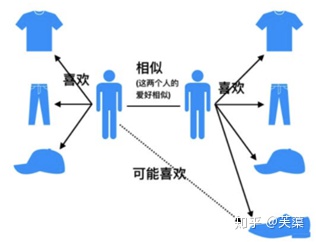
pearsonr(i, j)

下面是**基于用户协同过滤**和**基于物品协同过滤**的原理讲解。

## 2. 基于用户的协同过滤

图 User CF算法的工作过程

基于用户的协同过滤，简称User CF，思想其实比较简单，**首先是根据目标用户对物品的偏好找到兴趣相近的用户作为相邻用户，然后将相邻用户的喜爱的作品推荐给目标用户**；即**当一个用户A需要个性化推荐的时候，我们可以先找到和他有相似兴趣的其他用户，然后把那些用户喜欢的，而用户A没有听说过的物品推荐给A**。



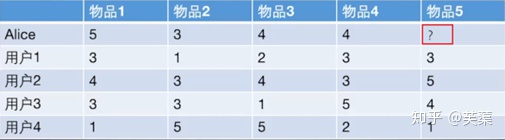
**UserCF算法主要包括两个步骤：**

· 找到和目标用户兴趣相似的集合

· 找到这个集合中的用户喜欢的， 且目标用户没有听说过的物品推荐给目标用户。

上面的两个步骤中， 第一个步骤里面， 我们会**基于前面给出的[相似性度量](https://www.zhihu.com/search?q=%E7%9B%B8%E4%BC%BC%E6%80%A7%E5%BA%A6%E9%87%8F&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)的方法找出与目标用户兴趣相似的用户**， 而第二个步骤里面， 如何基于相似用户喜欢的物品来对目标用户进行推荐呢？ 这个要**依赖于目标用户对相似用户喜欢的物品的一个喜好程度**， 那么如何衡量这个程度大小呢？ 为了更好理解上面的两个步骤， 下面拿一个具体的例子把两个步骤具体化。

**以下图为例，此例将会用于本文各种算法中**



给用户推荐物品的过程可以**形象化为一个猜测用户对商品进行打分的任务**，上面表格里面是5个用户对于5件物品的一个打分情况，就可以理解为用户对物品的喜欢程度

应用UserCF算法的两个步骤：

· 首先根据前面的这些打分情况(或者说已有的[用户向量](https://www.zhihu.com/search?q=%E7%94%A8%E6%88%B7%E5%90%91%E9%87%8F&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)）计算一下Alice和用户1， 2， 3， 4的相似程度， 找出与Alice最相似的n个用户

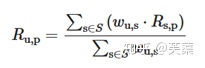
· 根据这n个用户对物品5的评分情况和与Alice的相似程度会猜测出Alice对物品5的评分， 如果评分比较高的话， 就把物品5推荐给用户Alice， 否则不推荐。

**关于第一个步骤，** **上面已经给出了计算两个用户相似性的方法，** **这里不再过多赘述，** **这里主要解决第二个问题，** **如何产生最终结果的预测。**

**最终结果的预测**

根据上面的几种方法， 我们可以计算出向量之间的相似程度， 也就是可以计算出Alice和其他用户的相近程度， 这时候我们就可以选出与Alice最相近的前n个用户， 基于他们对物品5的评价猜测出Alice的打分值， 那么是怎么计算的呢？

这里常用的方式之一是**利用用户相似度和相似用户的评价加权平均获得用户的评价预测**， 用下面式子表示：



这个式子里面， 权重w*u,s是用户u和用户s的相似度， R*s,p是用户s对物品p的评分。

还有一种方式如下， 这种方式考虑的更加前面， 依然是用户相似度作为权值， 但后面不单纯的是其他用户对物品的评分， 而是**该物品的评分与此用户的所有评分的差值进行加权平均，** **这时候考虑到了有的用户内心的评分标准不一的情况**， 即有的用户喜欢打高分， 有的用户喜欢打低分的情况。

IMG_267

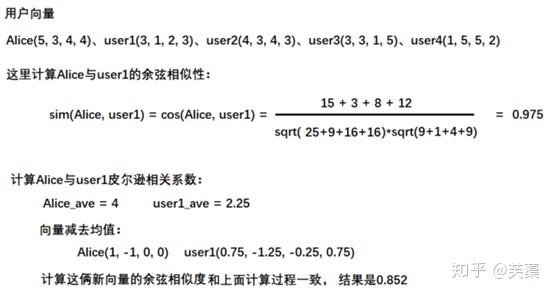
所以这一种计算方式更为推荐。下面的计算将使用这个方式。

在获得用户u对不同物品的评价预测后， 最终的推荐列表根据预测评分进行排序得到。 至此，基于用户的协同过滤算法的推荐过程完成。

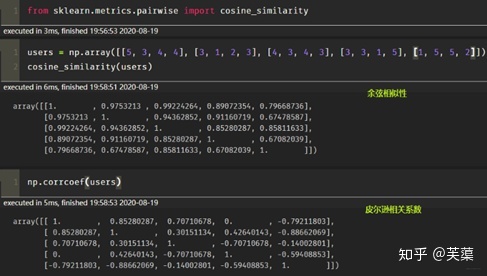
根据上面的问题， 下面手算一下：

Aim: 猜测Alice对物品5的得分：

**1）计算Alice与其他用户的相似度（这里使用皮尔逊相关系数）**



这里我们使用皮尔逊相关系数， 也就是Alice与用户1的相似度是0.85。同样的方式， 我们就可以计算与其他用户的相似度， 这里可以使用numpy的相似度函数得到用户的**相似性矩阵**：



从这里看出, Alice用户和用户2,用户3,用户4的相似度是0.7，0, -0.79。 所以如果n=2， 找到与Alice最相近的两个用户是用户1， 和Alice的相似度是0.85， 用户2， 和Alice相似度是0.7

**2）根据相似度用户计算Alice对物品5的最终得分** 用户1对物品5的评分是3， 用户2对物品5的打分是5， 那么根据上面的计算公式， 可以计算出Alice对物品5的最终得分是

IMG_270IMG_271

**3）根据用户评分对用户进行推荐** 这时候， 我们就得到了Alice对物品5的得分是4.87， 根据Alice的打分对物品排个序从大到小：物品1>物品5>物品3=物品4>物品2 这时候，如果要向Alice推荐2款产品的话， 我们就可以推荐物品1和物品5给Alice

至此， 基于用户的协同过滤算法原理介绍完毕。

### 2.1 UserCF编程实现

这里简单的通过编程实现上面的案例，为后面的大作业做一个热身， 梳理一下上面的过程其实就是三步： **计算用户[相似性矩阵](https://www.zhihu.com/search?q=%E7%9B%B8%E4%BC%BC%E6%80%A7%E7%9F%A9%E9%98%B5&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)、得到前n个相似用户、计算最终得分**。

所以我们下面的程序也是分为这**三步**：

**1）首先，** **先把数据表给建立起来** 这里我采用了字典的方式， 之所以没有用pandas， 是因为上面举得这个例子其实是个个例， 在真实情况中， 我们知道， 用户对物品的打分情况并不会这么完整， 会存在大量的空值， 所以矩阵会很稀疏， 这时候用DataFrame， 会有大量的NaN。故这里用字典的形式存储。 用两个字典， 第一个字典是物品-用户的评分映射， 键是物品1-5， 用A-E来表示， 每一个值又是一个字典， 表示的是每个用户对该物品的打分。 第二个字典是用户-物品的评分映射， 键是上面的五个用户， 用1-5表示， 值是该用户对每个物品的打分。

# 定义数据集， 也就是那个表格， 注意这里我们采用字典存放数据， 因为实际情况中数据是非常稀疏的， 很少有情况是现在这样

def loadData():

items={'A': {1: 5, 2: 3, 3: 4, 4: 3, 5: 1},

'B': {1: 3, 2: 1, 3: 3, 4: 3, 5: 5},

'C': {1: 4, 2: 2, 3: 4, 4: 1, 5: 5},

'D': {1: 4, 2: 3, 3: 3, 4: 5, 5: 2},

'E': {2: 3, 3: 5, 4: 4, 5: 1}

}

users={1: {'A': 5, 'B': 3, 'C': 4, 'D': 4},

2: {'A': 3, 'B': 1, 'C': 2, 'D': 3, 'E': 3},

3: {'A': 4, 'B': 3, 'C': 4, 'D': 3, 'E': 5},

4: {'A': 3, 'B': 3, 'C': 1, 'D': 5, 'E': 4},

5: {'A': 1, 'B': 5, 'C': 5, 'D': 2, 'E': 1}

}

return items,users

items, users = loadData()

item\_df = pd.DataFrame(items).T

user\_df = pd.DataFrame(users).T

**2）计算用户相似性矩阵** 这个是一个共现矩阵, 5\*5，行代表每个用户， 列代表每个用户， 值代表用户和用户的相关性，这里的思路是这样， 因为要求用户和用户两两的相关性， 所以需要用双层循环遍历用户-物品评分数据， 当不是同一个用户的时候， 我们要去遍历物品-用户评分数据， 在里面去找这两个用户同时对该物品评过分的数据放入到这两个用户向量中。 因为正常情况下会存在很多的NAN， 即可能用户并没有对某个物品进行评分过， 这样的不能当做用户向量的一部分， 没法计算相似性。 还是看代码吧， 感觉不太好描述：

"""计算用户相似性矩阵"""

similarity\_matrix = pd.DataFrame(np.zeros((len(users), len(users))), index=[1, 2, 3, 4, 5], columns=[1, 2, 3, 4, 5])

# 遍历每条用户-物品评分数据

for userID in users:

for otheruserId in users:

vec\_user = []

vec\_otheruser = []

if userID != otheruserId:

for itemId in items: # 遍历物品-用户评分数据

itemRatings = items[itemId] # 这也是个字典 每条数据为所有用户对当前物品的评分

if userID in itemRatings and otheruserId in itemRatings: # 说明两个用户都对该物品评过分

vec\_user.append(itemRatings[userID])

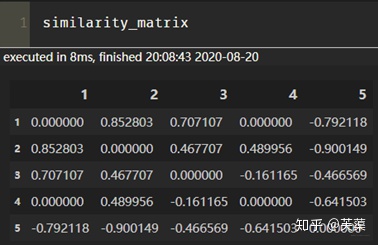
vec\_otheruser.append(itemRatings[otheruserId])

# 这里可以获得相似性矩阵(共现矩阵)

similarity\_matrix[userID][otheruserId] = np.corrcoef(np.array(vec\_user), np.array(vec\_otheruser))[0][1]

#similarity\_matrix[userID][otheruserId] = cosine\_similarity(np.array(vec\_user), np.array(vec\_otheruser))[0][1]

这里的similarity\_matrix就是我们的用户相似性矩阵， 长下面这样：



有了相似性矩阵， 我们就可以得到与Alice最相关的前n个用户。

**3）计算前n个相似的用户**

"""计算前n个相似的用户"""

n = 2

similarity\_users = similarity\_matrix[1].sort\_values(ascending=False)[:n].index.tolist() # [2, 3] 也就是用户1和用户2

**4）计算最终得分** 这里就是上面的那个公式了。

"""计算最终得分"""

base\_score = np.mean(np.array([value for value in users[1].values()]))

weighted\_scores = 0.

corr\_values\_sum = 0.

for user in similarity\_users: # [2, 3]

corr\_value = similarity\_matrix[1][user] # 两个用户之间的相似性

mean\_user\_score = np.mean(np.array([value for value in users[user].values()])) # 每个用户的打分平均值

weighted\_scores += corr\_value \* (users[user]['E']-mean\_user\_score) # 加权分数

corr\_values\_sum += corr\_value

final\_scores = base\_score + weighted\_scores / corr\_values\_sum

print('用户Alice对物品5的打分: ', final\_scores)

user\_df.loc[1]['E'] = final\_scores

user\_df

结果如下：



至此，我们就用代码完成了上面的小例子，有了这个评分，我们其实就可以对该用户做推荐了。这其实就是微型版的UserCF的工作过程了。

**注意：基于用户协同过滤的完整代码参考源代码文件中的UserCF.py**

**2.2 UserCF优缺点**

User-based算法存在两个重大问题：

1） **数据稀疏性**。一个大型的电子商务推荐系统一般有非常多的物品，用户可能买的其中不到1%的物品，不同用户之间买的物品重叠性较低，导致算法无法找到一个用户的邻居，即偏好相似的用户。**这导致UserCF不适用于那些正反馈获取较困难的应用场景**(如酒店预订，大件商品购买等低频应用)

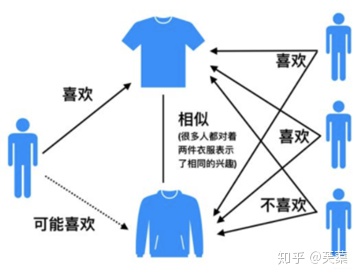
2） **算法扩展性**。基于用户的协同过滤需要维护用户[相似度矩阵](https://www.zhihu.com/search?q=%E7%9B%B8%E4%BC%BC%E5%BA%A6%E7%9F%A9%E9%98%B5&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)以便快速的找出Topn相似用户，该矩阵的存储开销非常大，存储空间随着用户数量的增加而增加，**不适合用户数据量大的情况使用**。

由于UserCF技术上的两点缺陷，导致很多电商平台并没有采用这种算法，而是采用了ItemCF算法实现最初的推荐系统。

## 3. 基于物品的协同过滤

图 ltem CF 算法的工作过程

基于物品的协同过滤, 简称 Item CF，基本思想是**预先根据所有用户的历史偏好数据计算物品之间的相似性，然后把与用户喜欢的物品相类似的物品推荐给用户**。比如物品a和c非常相似，因为喜欢a的用户同时也喜欢c，而用户A喜欢a，所以把c推荐给用户A。**Item CF算法并不利用物品的内容属性计算物品之间的相似度，主要通过分析用户的行为记录计算物品之间的相似度，该算法认为，物品a和物品c具有很大的相似度是因为喜欢物品a的用户大都喜欢物品c**。

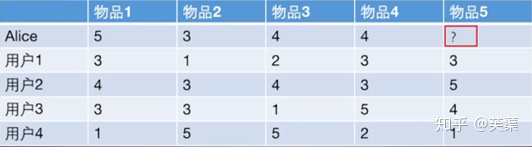


**基于物品的协同过滤算法主要分为两步：**

· 计算物品之间的相似度

· 根据物品的相似度和用户的历史行为给用户生成推荐列表（购买了该商品的用户也经常购买的其他商品）

基于物品的协同过滤算法和基于用户的协同过滤算法很像， 所以我们这里直接还是拿上面Alice的那个例子来看。



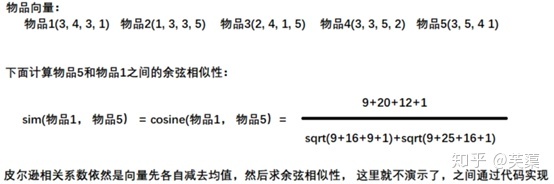
如果想知道Alice对物品5打多少分， 基于物品的协同过滤算法会这么做：

· 首先计算一下物品5和物品1， 2， 3， 4之间的相似性(它们也是向量的形式， 每一列的值就是它们的向量表示， 因为ItemCF认为物品a和物品c具有很大的相似度是因为喜欢物品a的用户大都喜欢物品c， 所以就可以基于每个用户对该物品的打分或者说喜欢程度来向量化物品)

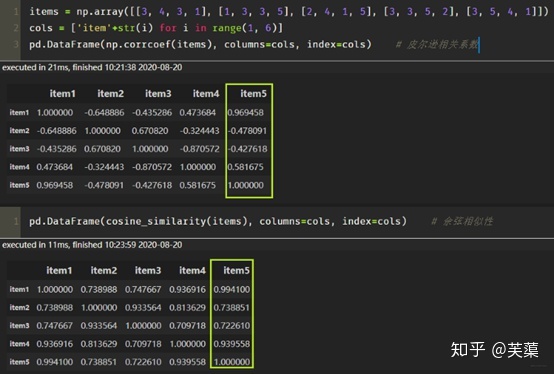
· 找出与物品5最相近的n个物品

· 根据Alice对最相近的n个物品的打分去计算对物品5的打分情况

**下面我们就可以具体计算一下，** **首先是步骤1：**



由于计算比较麻烦， 这里直接用python计算了：



根据皮尔逊相关系数， 可以找到与物品5最相似的2个物品是item1和item4(n=2)， 下面基于上面的公式计算最终得分：

IMG_279IMG_280

这时候依然可以向Alice推荐物品5。下面也是简单编程实现一下， 和上面的差不多：

"""计算物品的相似矩阵"""

similarity\_matrix = pd.DataFrame(np.ones((len(items), len(items))), index=['A', 'B', 'C', 'D', 'E'], columns=['A', 'B', 'C', 'D', 'E'])

# 遍历每条物品-用户评分数据

for itemId in items:

for otheritemId in items:

vec\_item = [] # 定义列表， 保存当前两个物品的向量值

vec\_otheritem = []

#userRagingPairCount = 0 # 两件物品均评过分的用户数

if itemId != otheritemId: # 物品不同

for userId in users: # 遍历用户-物品评分数据

userRatings = users[userId] # 每条数据为该用户对所有物品的评分， 这也是个字典

if itemId in userRatings and otheritemId in userRatings: # 用户对这两个物品都评过分

#userRagingPairCount += 1

vec\_item.append(userRatings[itemId])

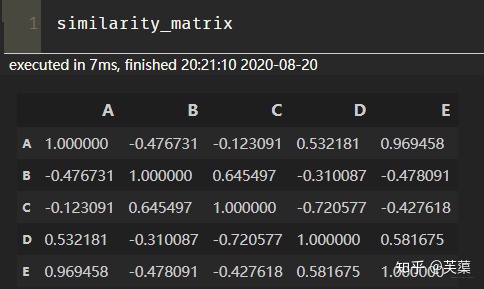
vec\_otheritem.append(userRatings[otheritemId])

# 这里可以获得相似性矩阵(共现矩阵)

similarity\_matrix[itemId][otheritemId] = np.corrcoef(np.array(vec\_item), np.array(vec\_otheritem))[0][1]

#similarity\_matrix[itemId][otheritemId] = cosine\_similarity(np.array(vec\_item), np.array(vec\_otheritem))[0][1]

这里就是物品的相似度矩阵了， 大概长下面这个样子：



**然后也是得到与物品5相似的前n个物品，** **计算出最终得分来。**

"""得到与物品5相似的前n个物品"""

n = 2

similarity\_items = similarity\_matrix['E'].sort\_values(ascending=False)[:n].index.tolist() # ['A', 'D']

"""计算最终得分"""

base\_score = np.mean(np.array([value for value in items['E'].values()]))

weighted\_scores = 0.

corr\_values\_sum = 0.

for item in similarity\_items: # ['A', 'D']

corr\_value = similarity\_matrix['E'][item] # 两个物品之间的相似性

mean\_item\_score = np.mean(np.array([value for value in items[item].values()])) # 每个物品的打分平均值

weighted\_scores += corr\_value \* (users[1][item]-mean\_item\_score) # 加权分数

corr\_values\_sum += corr\_value

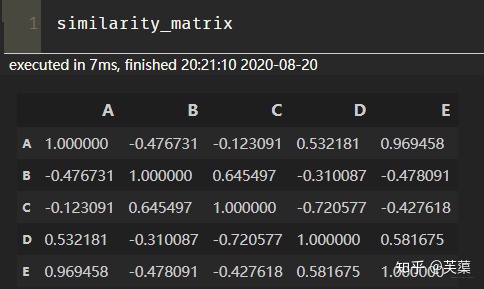
final\_scores = base\_score + weighted\_scores / corr\_values\_sum

print('用户Alice对物品5的打分: ', final\_scores)

user\_df.loc[1]['E'] = final\_scores

user\_df

结果如下：



**注意：基于商品的协同过滤算法的完整代码参考源代码文件中的ItemCF.py**

## 4. 算法评估

由于**UserCF**和**ItemCF**结果评估部分是共性知识点， 所以在这里统一标识。 这里介绍评测指标：

**1)召回率**

**对用户u推荐N个物品记为R(u), 令用户u在测试集上喜欢的物品集合为T(u)**， 那么召回率定义为：

IMG_283

这个意思就是说，**在用户真实购买或者看过的影片里面，模型真正预测出了多少，这个考察的是模型推荐的一个全面性**。

**2)准确率**

准确率定义为：

IMG_284

这个意思再说，**在推荐的所有物品中，用户真正看的有多少，这个考察的是模型推荐的一个准确性**。为了提高准确率，模型需要把非常有把握的才对用户进行推荐，所以这时候就减少了推荐的数量，而这往往就损失了全面性，真正预测出来的会非常少，所以实际应用中应该综合考虑两者的平衡。

**3)覆盖率**

覆盖率**反映了推荐算法发掘长尾的能力，覆盖率越高，说明推荐算法越能将长尾中的物品推荐给用户**。

IMG_285

该覆盖率表示最终的推荐列表中包含多大比例的物品。如果所有物品都被给推荐给至少一个用户，那么覆盖率是100%。

**4)新颖度**

**用推荐列表中物品的平均流行度度量推荐结果的新颖度**。如果推荐出的物品都很热门，说明推荐的新颖度较低。由于物品的流行度分布呈长尾分布，所以为了流行度的平均值更加稳定， 在计算平均流行度时对每个物品的流行度取对数。

## 5. 协同过滤算法的权重改进



**（1）基础算法**

**图1**为最简单的计算**物品相关度**的公式，分子为同时喜好itemi和itemj的用户数

**（2）对热门物品的惩罚**

**图1**存在一个问题，如果 item-j 是很热门的商品，导致很多喜欢 item-i 的用户都喜欢 item-j，这时 **wij就会非常大**。同样，几乎所有的物品都和 item-j 的相关度非常高，这显然是不合理的。所以**图2**中分母通过引入 N(j) 来对 item-j 的热度进行**惩罚**。如果物品很热门，那么N(j)就会越大，对应的权重就会变小。

**（3）对热门物品的进一步惩罚**

如果 item-j 极度热门，上面的算法还是不够的。举个例子，《[Harry Potter](https://www.zhihu.com/search?q=Harry+Potter&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)》非常火，买任何一本书的人都会购买它，即使通过**图2**的方法对它进行了惩罚，但是《Harry Potter》仍然会获得很高的相似度。这就是推荐系统领域著名的 Harry Potter Problem。

如果需要进一步对热门物品惩罚，可以继续修改公式为如**图3**所示，通过调节参数 α，α 越大，惩罚力度越大，热门物品的相似度越低，整体结果的平均热门程度越低。

**（4）对活跃用户的惩罚**

同样的，Item-based CF 也需要考虑活跃用户（即一个活跃用户（专门做刷单）可能买了非常多的物品）的影响，活跃用户对物品相似度的贡献应该小于不活跃用户。**图4**为集合了该权重的算法。

## 6.协同过滤推荐算法存在的问题

协同过滤推荐算法给我们带来便利的同时，也存在一些问题，在实际应用过程中我们经常会遇到，以下就存在的一些**典型的问题进行了分析**。

### 6.1 数据稀疏性的问题

推荐系统面临的问题中最大的问题就是**数据稀疏性问题**，它**会影响推荐系统推荐的准确度**，在个性化推荐系统中，协同过滤是最常用的技术，数据稀疏性影响了协同过滤的结果，因为在很多推荐系统中只有很小一部分评分，当在评分很小的情况下，推荐的准确度也会下降。

目前来说解决数据稀疏问题最好的算法是**降维技术**，比如**[奇异值SV](https://www.zhihu.com/search?q=%E5%A5%87%E5%BC%82%E5%80%BCSV&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)D分解**。**降维技术的关键是通过原始矩阵通过计算来求其低维度的相似来降低矩阵的维度**，有一点弊端就是**数据量大的时候运算成本很高**，好在最近几年云计算发展迅猛，这些计算量都可以放在运动，来弥补计算量的问题，除了降维技术之外**人口统计过滤**也是解决数据稀疏性问题的很好的方法，**人口统计过滤是通过用户资料的对比找到用户相似，然后通过相似的用户来进行推荐**，这种方式可以解决数据量很稀疏的情况下使用。

### 6.2 冷启动问题

**[冷启动问题](https://www.zhihu.com/search?q=%E5%86%B7%E5%90%AF%E5%8A%A8%E9%97%AE%E9%A2%98&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)**也被认作为推荐系统面临的问题中关键之一。**冷启动产生原因是新用户进入系统没有用户行为数据**，所以就不存在收集用户偏好的过程，进而也就很难进行有效的推荐。解决用户冷启动的问题可以采用上一小节中的**人口统计过滤**，其针对于冷启动也会有很好的效果，除此之外只能通过业务上调整去改变并获得用户的一些资料，从而进行有效推荐，具体的业务有**利用用户注册信息、社交网络挖掘、启用用户信息**。

具体做法是通过用户信息进行**用户分类**，利用社交网络导入用户的好友关系，然后通过好友关系来进行**相关推荐**，还可以通过用户的首次访问不立即给推荐反馈结果，而是通过引导页用户填写用户感兴趣的内容，然后通过这些收集到的准确信息来进行**准确的推荐**，一般我们会通过引导用户填写比较有代表性、区分性以及多样性的内容去进行选择，比如在**豆瓣FM**中，一般会采用热门精选的音乐以及分组好的音乐来给用户进行选择。解决内容冷启动问题可以通过物品的信息来解决新物品的推荐问题，一般会采用**物品标签进行推荐**，通过抽象出新物品的标签以及利用老物品的标签进行**匹配**，匹配成功就进行相关度的推荐。而系统的冷启动可以通过**专家发挥专家的作用**，在系统冷启动环节中，可以引入专家的知识，利用专家的知识构建起物品的相关度表，然后通过表格进行相关度的分析进行推荐。

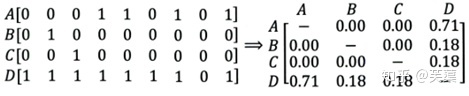
### 6.3 扩展性问题

推荐系统中通过推荐算法获得之后，因为用户之前没有足够多的对物品进行评分，有时候很难掌握用户的各方面兴趣和需求，这就会导致出现**过拟合现象**，这一类现象就是推荐系统中的**扩展性问题**，这类问题对于推荐系统的体验上是一个很大的问题。

扩展性问题在实际应用中是无可避免的，其中**受数据集大小的扩展，数据完整性等问题的限制**，对于数据完整性的问题，通常是引入**随机性**，使算法收敛到全局最优或者逼近全局最优，但是推荐系统运行过程是动态变化的，同时也会**受冷启动影响**，用户/对象/喜好数据达到一定规模时，性能这个时候也会存在一些瓶颈，这个时候的**大数据处理和算法的可扩展性**成为推荐算法技术需要实施并迫切需要解决的问题。

### 6.4 泛化能力弱的问题

协同过滤算法存在的问题之一就是**泛化能力弱**， 即协同过滤无法将两个物品相似的信息推广到其他物品的相似性上。 导致的问题是**热门物品具有很强的头部效应，** **容易跟大量物品产生相似，** **而尾部物品由于特征向量稀疏，** **导致很少被推荐**。 比如下面这个例子：



A, B, C, D是物品， 看右边的物品共现矩阵， 可以发现物品D与A、B、C的相似度比较大， 所以很有可能将D推荐给用过A、B、C的用户。 但是物品D与其他物品相似的原因是因为D是一件热门商品， 系统无法找出A、B、C之间相似性的原因是其特征太稀疏， 缺乏相似性计算的直接数据。 所以这就是协同过滤的天然缺陷：**推荐系统头部效应明显，** **处理稀疏向量的能力弱**。

为了解决这个问题，同时增加模型的泛化能力，2006年，**矩阵分解技术(Matrix Factorization,MF**)被提出，该方法在**协同过滤共现矩阵**的基础上，使用更稠密的隐向量表示用户和物品，挖掘用户和物品的**隐含兴趣和隐含特征**，在一定程度上弥补协同过滤模型处理稀疏矩阵能力不足的问题。

### 6.5 其它一些问题和挑战

推荐系统在推荐的过程中还会存在**隐私问题、噪声问题、解释性问题、合理性、新颖性**等等，这些问题在推荐系统中很有挑战性，但是解决这些问题出现的同时也会改变推荐准确度，所以我们需要权衡这些问题和推荐之间的联系，目前很多算法都有在改进相关的问题，改进的措施也是多维度的。

## 扩展

1.**什么时候使用UserCF，什么时候使用ItemCF？为什么？**

答案：

1）UserCF 由于是基于用户相似度进行推荐， 所以具备更强的社交特性， 这样的特点非常适于**用户少，** **物品多，** **时效性较强的场合**， 比如新闻推荐场景， 因为新闻本身兴趣点分散， 相比用户对不同新闻的兴趣偏好， 新闻的及时性，热点性往往更加重要， 所以正好适用于发现热点，跟踪热点的趋势。 另外还具有推荐新信息的能力， 更有可能发现惊喜, 因为看的是人与人的相似性, 推出来的结果可能更有惊喜，可以发现用户潜在但自己尚未察觉的兴趣爱好。

对于用户较少， 要求时效性较强的场合， 就可以考虑UserCF。

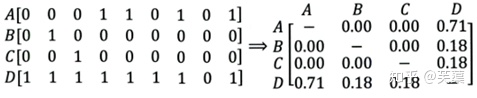
2）ItemCF 这个更适用于兴趣变化较为稳定的应用， 更接近于个性化的推荐， 适合**物品少，用户多，用户兴趣固定持久，** **物品更新速度不是太快的场合**， 比如推荐艺术品， 音乐， 电影。 下面是UserCF和ItemCF的优缺点对比： （来自[项亮](https://www.zhihu.com/search?q=%E9%A1%B9%E4%BA%AE&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)推荐系统实践）

2.**协同过滤在计算上有什么缺点？有什么比较好的思路可以解决（缓解）？**

答案：

**较差的稀疏向量处理能力**

第一个问题就是**泛化能力弱**，即协同过滤无法将两个物品相似的信息推广到其他物品的相似性上。导致的问题是**热门物品具有很强的头部效应，容易跟大量物品产生相似，而尾部物品由于特征向量稀疏，导致很少被推荐**。比如下面这个例子：

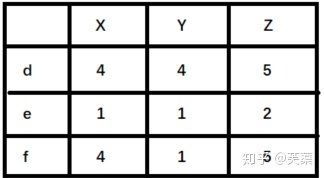


A, B, C, D是物品， 看右边的物品共现矩阵， 可以发现物品D与A、B、C的相似度比较大， 所以很有可能将D推荐给用过A、B、C的用户。 但是物品D与其他物品相似的原因是因为D是一件热门商品， 系统无法找出A、B、C之间相似性的原因是其特征太稀疏， 缺乏相似性计算的直接数据。 所以这就是协同过滤的天然缺陷：**推荐系统头部效应明显，** **处理稀疏向量的能力弱**。

为了解决这个问题，同时增加模型的泛化能力，2006年，**矩阵分解技术(Matrix Factorization, MF**)被提出，该方法在协同过滤共现矩阵的基础上，使用更稠密的隐向量表示用户和物品， 挖掘用户和物品的隐含兴趣和隐含特征，在一定程度上弥补协同过滤模型处理稀疏矩阵能力不足的问题。 具体细节等后面整理，这里先铺垫一下。

**3.上面介绍的相似度计算方法有什么优劣之处？**

cosine相似度还是比较常用的，一般效果也不会太差，但是对于评分数据不规范的时候，也就是说，存在有的用户喜欢打高分，有的用户喜欢打低分情况的时候，有的用户喜欢乱打分的情况，这时候consine相似度算出来的结果可能就不是那么准确了，比如下面这种情况：



这时候，如果用余弦相似度进行计算，会发现用户d和用户f比较相似，而实际上，如果看这个商品喜好的一个趋势的话，其实d和e比较相近，只不过e比较喜欢打低分，d比较喜欢打高分。 所以对于这种用户评分偏置的情况，余弦相似度就不是那么好了，可以考虑使用下面的皮尔逊相关系数。

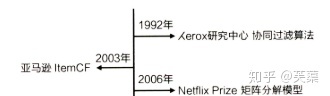
4.**协同过滤还存在其他什么缺陷？有什么比较好的思路可以解决（缓解）？**

答案：

协同过滤的特点就是完全没有利用到物品本身或者是用户自身的属性， 仅仅利用了用户与物品的交互信息就可以实现推荐，比较简单高效， 但这也是它的一个短板所在， 由于无法有效的引入用户年龄， 性别，商品描述，商品分类，当前时间，地点等一系列用户特征、物品特征和上下文特征， 这就造成了有效信息的遗漏，不能充分利用其它[特征数据](https://www.zhihu.com/search?q=%E7%89%B9%E5%BE%81%E6%95%B0%E6%8D%AE&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)。

为了解决这个问题， 在推荐模型中引用更多的特征，**推荐系统慢慢的从以协同过滤为核心到了以逻辑回归模型为核心**， 提出了能够综合不同类型特征的机器学习模型。

演化图左边的时间线梳理完毕：



## 实战

**UserCF.py**

import pandas as pd

import numpy as np

import warnings

import random, math, os

from tqdm import tqdm

from sklearn.model\_selection import train\_test\_split

warnings.filterwarnings('ignore')

# 评价指标

# 推荐系统推荐正确的商品数量占用户实际点击的商品数量

def Recall(Rec\_dict, Val\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Val\_dict: 用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

hit\_items = 0

all\_items = 0

for uid, items in Val\_dict.items():

rel\_set = items

rec\_set = Rec\_dict[uid]

for item in rec\_set:

if item in rel\_set:

hit\_items += 1

all\_items += len(rel\_set)

return round(hit\_items / all\_items \* 100, 2)

# 推荐系统推荐正确的商品数量占给用户实际推荐的商品数

def Precision(Rec\_dict, Val\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Val\_dict: 用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

hit\_items = 0

all\_items = 0

for uid, items in Val\_dict.items():

rel\_set = items

rec\_set = Rec\_dict[uid]

for item in rec\_set:

if item in rel\_set:

hit\_items += 1

all\_items += len(rec\_set)

return round(hit\_items / all\_items \* 100, 2)

# 所有被推荐的用户中,推荐的商品数量占这些用户实际被点击的商品数量

def Coverage(Rec\_dict, Trn\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Trn\_dict: 训练集用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

rec\_items = set()

all\_items = set()

for uid in Rec\_dict:

for item in Trn\_dict[uid]:

all\_items.add(item)

for item in Rec\_dict[uid]:

rec\_items.add(item)

return round(len(rec\_items) / len(all\_items) \* 100, 2)

# 使用平均流行度度量新颖度,如果平均流行度很高(即推荐的商品比较热门),说明推荐的新颖度比较低

def Popularity(Rec\_dict, Trn\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Trn\_dict: 训练集用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

pop\_items = {}

for uid in Trn\_dict:

for item in Trn\_dict[uid]:

if item not in pop\_items:

pop\_items[item] = 0

pop\_items[item] += 1

pop, num = 0, 0

for uid in Rec\_dict:

for item in Rec\_dict[uid]:

pop += math.log(pop\_items[item] + 1) # 物品流行度分布满足长尾分布,取对数可以使得平均值更稳定

num += 1

return round(pop / num, 3)

# 将几个评价指标[指标函数](https://www.zhihu.com/search?q=%E6%8C%87%E6%A0%87%E5%87%BD%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)一起调用

def rec\_eval(val\_rec\_items, val\_user\_items, trn\_user\_items):

print('recall:', Recall(val\_rec\_items, val\_user\_items))

print('precision', Precision(val\_rec\_items, val\_user\_items))

print('coverage', Coverage(val\_rec\_items, trn\_user\_items))

print('Popularity', Popularity(val\_rec\_items, trn\_user\_items))

def get\_data(root\_path):

# 读取数据

rnames = ['user\_id', 'movie\_id', 'rating', 'timestamp']

ratings = pd.read\_csv(os.path.join(root\_path, 'ratings.dat'), sep='::', engine='python', names=rnames)

# 分割训练和验证集

trn\_data, val\_data, \_, \_ = train\_test\_split(ratings, ratings, test\_size=0.2)

trn\_data = trn\_data.groupby('user\_id')['movie\_id'].apply(list).reset\_index()

val\_data = val\_data.groupby('user\_id')['movie\_id'].apply(list).reset\_index()

trn\_user\_items = {}

val\_user\_items = {}

# 将数组构造成字典的形式{user\_id: [item\_id1, item\_id2,...,item\_idn]}

for user, movies in zip(\*(list(trn\_data['user\_id']), list(trn\_data['movie\_id']))):

trn\_user\_items[user] = set(movies)

for user, movies in zip(\*(list(val\_data['user\_id']), list(val\_data['movie\_id']))):

val\_user\_items[user] = set(movies)

return trn\_user\_items, val\_user\_items

def User\_CF\_Rec(trn\_user\_items, val\_user\_items, K, N):

'''

trn\_user\_items: 表示训练数据，格式为：{user\_id1: [item\_id1, item\_id2,...,item\_idn], user\_id2...}

val\_user\_items: 表示验证数据，格式为：{user\_id1: [item\_id1, item\_id2,...,item\_idn], user\_id2...}

K: Ｋ表示的是相似用户的数量，每个用户都选择与其最相似的K个用户

N: N表示的是给用户推荐的商品数量，给每个用户推荐相似度最大的N个商品

'''

# 建立item->users倒排表

# 倒排表的格式为: {item\_id1: {user\_id1, user\_id2, ... , user\_idn}, item\_id2: ...} 也就是每个item对应有那些用户有过点击

# 建立倒排表的目的就是为了更好的统计用户之间共同交互的商品数量

print('建立倒排表...')

item\_users = {}

for uid, items in tqdm(trn\_user\_items.items()): # 遍历每一个用户的数据,其中包含了该用户所有交互的item

for item in items: # 遍历该用户的所有item, 给这些item对应的用户列表添加对应的uid

if item not in item\_users:

item\_users[item] = set()

item\_users[item].add(uid)

# 计算用户协同过滤矩阵

# 即利用item-users倒排表统计用户之间交互的商品数量，用户协同过滤矩阵的表示形式为：sim = {user\_id1: {user\_id2: num1}, user\_id3:{user\_id4: num2}, ...}

# 协同过滤矩阵是一个双层的字典，用来表示用户之间共同交互的商品数量

# 在计算用户协同过滤矩阵的同时还需要记录每个用户所交互的商品数量，其表示形式为: num = {user\_id1：num1, user\_id2:num2, ...}

sim = {}

num = {}

print('构建协同过滤矩阵...')

for item, users in tqdm(item\_users.items()): # 遍历所有的item去统计,用户两辆之间共同交互的item数量

for u in users:

if u not in num: # 如果用户u不在字典num中，提前给其在字典中初始化为0,否则后面的运算会报key error

num[u] = 0

num[u] += 1 # 统计每一个用户,交互的总的item的数量

if u not in sim: # 如果用户u不在字典sim中，提前给其在字典中初始化为一个新的字典,否则后面的运算会报key error

sim[u] = {}

for v in users:

if u != v: # 只有当u不等于v的时候才计算用户之间的相似度

if v not in sim[u]:

sim[u][v] = 0

sim[u][v] += 1

# 计算用户相似度矩阵

# 用户协同过滤矩阵其实相当于是余弦相似度的分子部分,还需要除以分母,即两个用户分别交互的item数量的乘积

# 两个用户分别交互的item数量的乘积就是上面统计的[num字典](https://www.zhihu.com/search?q=num%E5%AD%97%E5%85%B8&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)

print('计算相似度...')

for u, users in tqdm(sim.items()):

for v, score in users.items():

sim[u][v] = score / math.sqrt(num[u] \* num[v]) # 余弦相似度分母部分

# 对验证数据中的每个用户进行TopN推荐

# 在对用户进行推荐之前需要先通过相似度矩阵得到与当前用户最相思的前K个用户，

# 然后对这K个用户交互的商品中除当前测试用户训练集中交互过的商品以外的商品计算最终的[相似度分数](https://www.zhihu.com/search?q=%E7%9B%B8%E4%BC%BC%E5%BA%A6%E5%88%86%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":"268401920"}" \t "_blank)

# 最终推荐的候选商品的相似度分数是由多个用户对该商品分数的一个累加和

print('给测试用户进行推荐...')

items\_rank = {}

for u, \_ in tqdm(val\_user\_items.items()): # 遍历测试集用户，给测试集中的每个用户进行推荐

items\_rank[u] = {} # 初始化用户u的候选item的字典

for v, score in sorted(sim[u].items(), key=lambda x: x[1], reverse=True)[:K]: # 选择与用户u最相思的k个用户

for item in trn\_user\_items[v]: # 遍历相似用户之间交互过的商品

if item not in trn\_user\_items[u]: # 如果相似用户交互过的商品，测试用户在训练集中出现过，就不用进行推荐，直接跳过

if item not in items\_rank[u]:

items\_rank[u][item] = 0 # 初始化用户u对item的相似度分数为０

items\_rank[u][item] += score # 累加所有相似用户对同一个item的分数

print('为每个用户筛选出相似度分数最高的Ｎ个商品...')

items\_rank = {k: sorted(v.items(), key=lambda x: x[1], reverse=True)[:N] for k, v in items\_rank.items()}

items\_rank = {k: set([x[0] for x in v]) for k, v in items\_rank.items()} # 将输出整合成合适的格式输出

return items\_rank

if \_\_name\_\_ == "\_\_main\_\_":

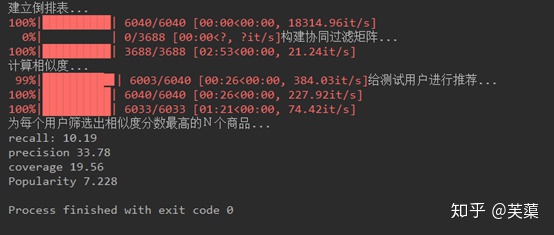
root\_path = './data/ml-1m/'

trn\_user\_items, val\_user\_items = get\_data(root\_path)

rec\_items = User\_CF\_Rec(trn\_user\_items, val\_user\_items, 80, 10)

rec\_eval(rec\_items, val\_user\_items, trn\_user\_items)

**结果：**



**ItemCF.py**

import pandas as pd

import numpy as np

import warnings

import random, math, os

from tqdm import tqdm

from sklearn.model\_selection import train\_test\_split

warnings.filterwarnings('ignore')

# 评价指标

# 推荐系统推荐正确的商品数量占用户实际点击的商品数量

def Recall(Rec\_dict, Val\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Val\_dict: 用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

hit\_items = 0

all\_items = 0

for uid, items in Val\_dict.items():

rel\_set = items

rec\_set = Rec\_dict[uid]

for item in rec\_set:

if item in rel\_set:

hit\_items += 1

all\_items += len(rel\_set)

return round(hit\_items / all\_items \* 100, 2)

# 推荐系统推荐正确的商品数量占给用户实际推荐的商品数

def Precision(Rec\_dict, Val\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Val\_dict: 用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

hit\_items = 0

all\_items = 0

for uid, items in Val\_dict.items():

rel\_set = items

rec\_set = Rec\_dict[uid]

for item in rec\_set:

if item in rel\_set:

hit\_items += 1

all\_items += len(rec\_set)

return round(hit\_items / all\_items \* 100, 2)

# 所有被推荐的用户中,推荐的商品数量占这些用户实际被点击的商品数量

def Coverage(Rec\_dict, Trn\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Trn\_dict: 训练集用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

rec\_items = set()

all\_items = set()

for uid in Rec\_dict:

for item in Trn\_dict[uid]:

all\_items.add(item)

for item in Rec\_dict[uid]:

rec\_items.add(item)

return round(len(rec\_items) / len(all\_items) \* 100, 2)

# 使用平均流行度度量新颖度,如果平均流行度很高(即推荐的商品比较热门),说明推荐的新颖度比较低

def Popularity(Rec\_dict, Trn\_dict):

'''

Rec\_dict: 推荐算法返回的推荐列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

Trn\_dict: 训练集用户实际点击的商品列表, 形式:{uid: {item1, item2,...}, uid: {item1, item2,...}, ...}

'''

pop\_items = {}

for uid in Trn\_dict:

for item in Trn\_dict[uid]:

if item not in pop\_items:

pop\_items[item] = 0

pop\_items[item] += 1

pop, num = 0, 0

for uid in Rec\_dict:

for item in Rec\_dict[uid]:

pop += math.log(pop\_items[item] + 1) # 物品流行度分布满足长尾分布,取对数可以使得平均值更稳定

num += 1

return round(pop / num, 3)

# 将几个评价指标指标函数一起调用

def rec\_eval(val\_rec\_items, val\_user\_items, trn\_user\_items):

print('recall:', Recall(val\_rec\_items, val\_user\_items))

print('precision', Precision(val\_rec\_items, val\_user\_items))

print('coverage', Coverage(val\_rec\_items, trn\_user\_items))

print('Popularity', Popularity(val\_rec\_items, trn\_user\_items))

def get\_data(root\_path):

# 读取数据

rnames = ['user\_id','movie\_id','rating','timestamp']

ratings = pd.read\_csv(os.path.join(root\_path, 'ratings.dat'), sep='::', engine='python', names=rnames)

# 分割训练和验证集

trn\_data, val\_data, \_, \_ = train\_test\_split(ratings, ratings, test\_size=0.2)

trn\_data = trn\_data.groupby('user\_id')['movie\_id'].apply(list).reset\_index()

val\_data = val\_data.groupby('user\_id')['movie\_id'].apply(list).reset\_index()

trn\_user\_items = {}

val\_user\_items = {}

# 将数组构造成字典的形式{user\_id: [item\_id1, item\_id2,...,item\_idn]}

for user, movies in zip(\*(list(trn\_data['user\_id']), list(trn\_data['movie\_id']))):

trn\_user\_items[user] = set(movies)

for user, movies in zip(\*(list(val\_data['user\_id']), list(val\_data['movie\_id']))):

val\_user\_items[user] = set(movies)

return trn\_user\_items, val\_user\_items

def Item\_CF(trn\_user\_items, val\_user\_items, K, N):

'''

trn\_user\_items: 表示训练数据，格式为：{user\_id1: [item\_id1, item\_id2,...,item\_idn], user\_id2...}

val\_user\_items: 表示验证数据，格式为：{user\_id1: [item\_id1, item\_id2,...,item\_idn], user\_id2...}

K: Ｋ表示的是相似商品的数量，为每个用户交互的每个商品都选择其最相思的K个商品

N: N表示的是给用户推荐的商品数量，给每个用户推荐相似度最大的N个商品

'''

# 建立user->item的倒排表

# 倒排表的格式为: {user\_id1: [item\_id1, item\_id2,...,item\_idn], user\_id2: ...} 也就是每个用户交互过的所有商品集合

# 由于输入的训练数据trn\_user\_items,本身就是这中格式的，所以这里不需要进行额外的计算

# 计算商品协同过滤矩阵

# 即利用user-items倒排表统计商品与商品之间被共同的用户交互的次数

# 商品协同过滤矩阵的表示形式为：sim = {item\_id1: {item\_id２: num1}, item\_id３: {item\_id４: num２}, ...}

# 商品协同过滤矩阵是一个双层的字典，用来表示商品之间共同交互的用户数量

# 在计算商品协同过滤矩阵的同时还需要记录每个商品被多少不同用户交互的次数，其表示形式为: num = {item\_id1：num1, item\_id２:num2, ...}

sim = {}

num = {}

print('构建相似性矩阵．．．')

for uid, items in tqdm(trn\_user\_items.items()):

for i in items:

if i not in num:

num[i] = 0

num[i] += 1

if i not in sim:

sim[i] = {}

for j in items:

if j not in sim[i]:

sim[i][j] = 0

if i != j:

sim[i][j] += 1

# 计算物品的相似度矩阵

# 商品协同过滤矩阵其实相当于是余弦相似度的分子部分,还需要除以分母,即两个商品被交互的用户数量的乘积

# 两个商品被交互的用户数量就是上面统计的num字典

print('计算协同过滤矩阵．．．')

for i, items in tqdm(sim.items()):

for j, score in items.items():

if i != j:

sim[i][j] = score / math.sqrt(num[i] \* num[j])

# 对验证数据中的每个用户进行TopN推荐

# 在对用户进行推荐之前需要先通过商品相似度矩阵得到当前用户交互过的商品最相思的前K个商品，

# 然后对这K个用户交互的商品中除当前测试用户训练集中交互过的商品以外的商品计算最终的相似度分数

# 最终推荐的候选商品的相似度分数是由多个相似商品对该商品分数的一个累加和

items\_rank = {}

print('给用户进行推荐．．．')

for uid, \_ in tqdm(val\_user\_items.items()):

items\_rank[uid] = {} # 存储用户候选的推荐商品

for hist\_item in trn\_user\_items[uid]: # 遍历该用户历史喜欢的商品，用来下面寻找其相似的商品

for item, score in sorted(sim[hist\_item].items(), key=lambda x: x[1], reverse=True)[:K]:

if item not in trn\_user\_items[uid]: # 进行推荐的商品一定不能在历史喜欢商品中出现

if item not in items\_rank[uid]:

items\_rank[uid][item] = 0

items\_rank[uid][item] += score

print('为每个用户筛选出相似度分数最高的Ｎ个商品...')

items\_rank = {k: sorted(v.items(), key=lambda x: x[1], reverse=True)[:N] for k, v in items\_rank.items()}

items\_rank = {k: set([x[0] for x in v]) for k, v in items\_rank.items()}

return items\_rank

if \_\_name\_\_ == "\_\_main\_\_":

root\_path = './data/ml-1m/'

trn\_user\_items, val\_user\_items = get\_data(root\_path)

rec\_items = Item\_CF(trn\_user\_items, val\_user\_items, 80, 10)

rec\_eval(rec\_items, val\_user\_items, trn\_user\_items)

**结果：**

