day 2

一、协同过滤算法

协同过滤（collaborative Filtering）：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，基于用户行为数据，而不考虑附加信息

基于领域的方法：

> 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

二、相似性度量方法

1. 杰卡德（Jaccard）相似系数

两个集合的杰卡德：两个用户交互商品的交集数量占这两个用户交互商品并集的数量比例

用户u, 用户v，这两个用户的杰卡德相似系数sim_{uv}, $N(u), N(v)$分别表示u，v交互商品的集合

由于杰卡德下属不反应用户的评分喜好信息，所以常用于评估用户是否对商品评分

$$sim_{uv} = \frac{|N(u) \cap N(v)|}{|N(u)| \cup |N(v)|}$$

2. 余弦相似度

分母不再是两用户交互商品的并集数量，而是分别交互的商品数量乘积

$$sim_{uv} = \frac{|N(u) \cap N(v)|}{|N(u)| \cdot |N(v)|}$$

令 N 为用户 - 商品交互矩阵，得到

$$sim_{uv} = cos(u, v) = \frac{u \cdot v}{|u| \cdot |v|}$$

（对大规模稀疏矩阵，余弦相似度常使用集合方式计算）

求和形式

r_{ui}, r_{vi}分别表示用户u和用户v对商品i是否有交互（或具体的评分值）

$$sim_{uv} = \frac{\sum_i r_{ui} * r_{vi}}{\sqrt{\sum_i r_{ui}^2}\sqrt{\sum_i r_{vi}^2}}$$

python 掉包运行 cosine_similarity

```
from sklearn.metrics.pairwise import cosine_similarity
i = [1, 0, 0, 0]
j = [1, 0.5, 0.5, 0]
cosine_similarity([i, j])
```

3. 皮尔逊相关系数

$\bar{r}_u, \bar{r}_v$分别表示用户u和用户v交互的所有商品交互数量或具体评分的均值

$$sim(u, v) = \frac{\sum_{i \in I}(r_{ui} - \bar{r}_u)(r_{vi} - \bar{r}_v)}{\sqrt{\sum_{i \in I}(r_{ui} - \bar{r}_u)^2}\sqrt{\sum_{i \in I}(r_{vi} - \bar{r}_v)^2}}$$

与余弦相似度比，皮尔逊相关系数使用用户的平均值对各个独立评分进行修正，减小了用户评分偏置的影响，其中python中的一种掉包如下

```
from scipy.stats import pearsonr
i = [1, 0, 0, 0]
j = [1, 0.5, 0.5, 0]
pearsonr(i, j)
```

* 基于一般协同过滤算法，进行权重改进

对热门物品的惩罚，对活跃用户进行惩罚

三、基于用户的协同过滤

1. UserCF基本概念

当一个用户u需要个性化推荐，先找到和他有相似兴趣的其他用户，然后把这些用户喜欢的，而u没听过的物品推荐给u ----- 猜测用户对商品进行打分的任务

a. 找到和目标用户兴趣相似的集合

b. 找到这个集合中的用户喜欢的，而目标用户没听说过的物品推荐给目标用户

2. 最终结果的预测

利用用户相似度和相似用户的平均加权平均获得用户的评价预测

$w_{u,s}$是用户u对用户s的相似度，$R_{s,p}$是用户s对物品p的评分

$$R_{\mathrm{u,p}} = \frac{\sum_{\mathrm{s}\in S}(w_{\mathrm{u,s}} \cdot R_{\mathrm{s,p}})}{\sum_{\mathrm{s}\in S} w_{\mathrm{u,s}}}$$

另一种 method（更全面）：

该物品的评分与此用户的所有评分的差值进行加权平均（考虑到用户内心的评分标准不一样）(?)

$$P_{i,j} = \bar{R}i + \frac{\sum_{k=1}^{n}\left(S_{i,k}\left(R_{k,j} - \bar{R}k\right)\right)}{\sum_{k=1}^{n} S_{j,k}}$$

3. 实例计算

	物品1	**物品2**	**物品3**	**物品4**	**物品5**
Alice	5	3	4	4	**？**
用户1	3	1	2	3	3
用户2	4	3	4	3	5
用户3	3	3	1	5	4
用户4	1	5	5	2	1

a. 计算Alice与其他用户的相似度（皮尔逊相关系数）

通过上表，得到用户向量

这里计算 Alice 和 user1 的余弦相似性：

$$sim(Alice, user1) = cos(Alice, user1) = \frac{15+3+8+12}{\sqrt{25+9+16+16} * \sqrt{9+1+4+9}} = 0.975$$

计算Alice和user1皮尔逊相关系数：

$$Alice\_ave = 4, user1\_ave = 2.25$$

向量减去均值得到

$$Alice(1, -1, 0, 0), user1(0.75, -1.25, -0.25, 0.75)$$

得到新向量的余弦相似度与原余弦相似度一致（？）

使用皮尔逊相关稀疏，同理可以计算得到其他用户的相似度，使用numpy的相似度函数得到用户的相似性矩阵

python_code:

```
user = np.asarray([[5, 3, 4, 4], [3, 1, 2, 3], [4, 3, 4,
3], [3, 3, 1, 5], [1, 5, 5, 2]])
cos_sim = cosine_similarity(user)
corroef = np.corrcoef(user)
print('余弦相似性:\n {}'.format(cos_sim))
print('皮尔逊相关系数:\n {}'.format(corroef))
```

output.txt:

```
余弦相似性:
 [[1.         0.9753213  0.99224264 0.89072354 0.79668736]
 [0.9753213  1.         0.94362852 0.91160719 0.67478587]
 [0.99224264 0.94362852 1.         0.85280287 0.85811633]
 [0.89072354 0.91160719 0.85280287 1.         0.67082039]
 [0.79668736 0.67478587 0.85811633 0.67082039 1.        ]]
皮尔逊相关系数:
 [[ 1.          0.85280287  0.70710678  0.
-0.79211803]
 [ 0.85280287  1.          0.30151134  0.42640143
-0.88662069]
 [ 0.70710678  0.30151134  1.         -0.70710678
-0.14002801]
 [ 0.          0.42640143 -0.70710678  1.
-0.59408853]
 [-0.79211803 -0.88662069 -0.14002801 -0.59408853  1.
]]
```

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

b. 根据相似度用户计算Alice对物品5的最终得分

$$P_{Alice,物品5} = \bar{R}Alice + \frac{\sum_{k=1}^{2}\left(S_{Alice,userk}\left(R_{userk,物品5} - \bar{R}userk\right)\right)}{\sum_{k=1}^{2} S_{Alice,userk}} = 4 + \frac{0.85 * (3 - 2.4) + 0.7 * (5 - 3.8)}{0.85 + 0.7} = 4.87$$

c. 根据用户评分对用户进行推荐

按照得分排序给Alice推荐物品1和物品5

四、UserCF编程实现

1. 计算用户相似性矩阵
2. 得到前n个相似用户
3. 计算最终得分

1. 建立数据表

字典方式存储数据（真实情况中，用户对物品的打分情况会很分散，导致大量的空值，所以矩阵会很稀疏，这个时候如果使用DataFrame，会有大量的NaN，所以用字典形式存储。

> 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表示，每个值也是一个字典，表示每个用户对物品的打分；
>
> 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，值是用户对每个物品的打分

python_code:

```
import pandas as pd

# 定义数据集，注意:采用字典存放数据，因为实际情况中数据是非常稀疏的
```

```
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
print('item_df:\n{}'.format(item_df))
print('user_df:\n{}'.format(user_df))
```

output.txt:

```
item_df:
     1    2    3    4    5
A  5.0  3.0  4.0  3.0  1.0
B  3.0  1.0  3.0  3.0  5.0
C  4.0  2.0  4.0  1.0  5.0
D  4.0  3.0  3.0  5.0  2.0
E  NaN  3.0  5.0  4.0  1.0
user_df:
     A    B    C    D    E
1  5.0  3.0  4.0  4.0  NaN
2  3.0  1.0  2.0  3.0  3.0
3  4.0  3.0  4.0  3.0  5.0
4  3.0  3.0  1.0  5.0  4.0
5  1.0  5.0  5.0  2.0  1.0
```

2. 计算用户相似性矩阵

5*5 的共现矩阵，行和列都代表用户，值代表用户间相似性，考虑两个用户，遍历物品-用户评分数据，在里面找这两个用户同时对该物品的评分数据放到这两个用户想两种。（其中会存在很多NaN值，即无法计算相似性）

python_code:

```
'''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:
            # print('userID:{}'.format(userID))
            # print('otheruserID:{}'.format(otheruserID))
            for itemID in items:
                itemRatings = items[itemID]
                if userID in itemRatings and otheruserID in
itemRatings:
                    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]
print('similarity_matrix:\n{}'.format(similarity_matrix))
```

output.txt:

```
similarity_matrix:
          1         2         3         4         5
1  0.000000  0.852803  0.707107  0.000000 -0.792118
2  0.852803  0.000000  0.467707  0.489956 -0.900149
3  0.707107  0.467707  0.000000 -0.161165 -0.466569
4  0.000000  0.489956 -0.161165  0.000000 -0.641503
5 -0.792118 -0.900149 -0.466569 -0.641503  0.000000
```

得到与Alice最相关的前n个用户

3. 计算前n个相似的用户

python_code:

```
'''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
# ============================== 解析
================================
p1 = similarity_matrix[1]
p2 = p1.sort_values(ascending=False)
p3 = p2.index
# 取出表的第一行
print('1st process: 1column of similarity_matrix:\n{};
type_p1: {}'.format(p1, type(p1)))
# 按照值从大到小排序
#
DataFrame.sort_values(by='##',axis=0,ascending=True,inplace
=False,na_position='last')
# by 指定列名(axis = 0或'index') 或者索引(axis = 1或'columns')
# axis  若axis=0或’index’，则按照指定列中数据大小排序；若axis=1
或’columns’，则按照指定索引中数据大小排序，默认axis=0
# ascending 是否按指定列的数组升序排列，默认为True，即升序排列
```

```
# inplace    是否用排序后的数据集替换原来的数据，默认为False，即不替换
# na_position    {‘first’,‘last’}，设定缺失值的显示位置
print('2nd process: sort_values:\n{}; type_p2: {}'.format(p2, type(p2)))
# .index 将Series类转化为int64index,即整数型索引
# .tolist() 将索引转换为列表
print('3th process: \n{}; type_p3: {}'.format(p3, type(p3)))
# 输出结果
print('similarity_users:\n{}'.format(similarity_users))
```

output.txt:

```
similarity_users:
[2, 3]
```

4. 计算最终得分(Alice 对物品5的打分)

python_code:

```
'''计算最终得分'''
base_score = np.mean(list(users[1].values()))
weighted_scores = 0.
corr_values_sum = 0.
for user in similarity_users:  # [2, 3]
    corr_value = similarity_matrix[1][user]  # 两个用户间的相似性
    mean_user_score = np.mean(list(users[user].values()))  # 用户的打分平均值
    weighted_scores += corr_value * (users[user]['E'] - mean_user_score)
    corr_values_sum += corr_value
final_scores = base_score + weighted_scores / co*rr_values_sum
print('用户Alice对物品5的打分：{}'.format(final_scores))
user_df.loc[1]['E'] = final_scores
print(user_df)
```

output.txt:

```
用户Alice对物品5的打分：4.871979899370592
     A    B    C    D        E
1  5.0  3.0  4.0  4.0  4.87198
2  3.0  1.0  2.0  3.0  3.00000
3  4.0  3.0  4.0  3.0  5.00000
4  3.0  3.0  1.0  5.0  4.00000
5  1.0  5.0  5.0  2.0  1.00000
```

基于用户协同过滤的完整代码参考 UserCF.py 和 UserCF.py

5. UserCF缺点

a. 数据稀疏性

不同用户间购买物品重叠性较低，难以找到偏好相似的用户，导致UserCF不适用于正反馈获取较难的应用场景（如酒店预定，大件商品购买等低频应用）

b. 算法扩展性

需要维护用户相似度矩阵，用于快速找出Topn相似用户，该矩阵的存储开销很大，不适合用户数据量较大的情况使用

五、基于物品的协同过滤

ItemCF：根据用户历史偏好数据，计算物品间相似性，然后将用户喜欢的相似物品推荐给用户。

ItemCF不利用物品内容属性计算相似度，主要通过用户的行为记录计算物品间的相似度。

> - 计算物品间相似度
> - 根据物品相似度和用户历史行为，给用户生产推荐列表

========== 计算 ==========

物品向量：

物品1-5

计算物品5和物品1间的余弦相似性

$$sim(\text{物品1，物品5}) = cosine(\text{物品1，物品5}) = \frac{9+20+12+1}{\sqrt{9+16+9+1}\sqrt{9+25+16+1}}$$

下面使用皮尔逊相关系数计算

python_code:

```
import numpy as np
import pandas as pd
items = np.asarray([[3, 4, 3, 1], [1, 3, 3, 5], [2, 4, 1,
5], [3, 3, 5, 2], [3, 5, 4, 1]])
cols = ['item' + str(i) for i in range(1, 6)]
item_corrcoef = pd.DataFrame(np.corrcoef(items),
columns=cols, index=cols)
print('item_corrcoef:{}'.format(item_corrcoef))
```

output.txt:

```
item_corrcoef:          item1     item2     item3     item4
    item5
item1  1.000000 -0.648886 -0.435286  0.473684  0.969458
item2 -0.648886  1.000000  0.670820 -0.324443 -0.478091
item3 -0.435286  0.670820  1.000000 -0.870572 -0.427618
item4  0.473684 -0.324443 -0.870572  1.000000  0.581675
item5  0.969458 -0.478091 -0.427618  0.581675  1.000000
```

所以得到与物品5最相似的物品是item1和item4(n = 2)，最终得分为：

$$P_{Alice,物品5} = \bar{R}物品5 + \frac{\sum_{k=1}^{2}\left(S_{物品5,物品k}\left(R_{Alice,物品k} - \bar{R}物品k\right)\right)}{\sum_{k=1}^{2} S_{物品k,物品5}} = \frac{13}{4} + \frac{0.97 * (5 - 3.2) + 0.58 * (4 - 3.4)}{0.97 + 0.58} = 4.6$$

1. 建立数据表

这一步和UserCF编程是一致的

```
import pandas as pd
import numpy as np
import math

# 定义数据集，注意:采用字典存放数据，因为实际情况中数据是非常稀疏的
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
print('item_df:\n{}'.format(item_df))
print('user_df:\n{}'.format(user_df))
```

2. 计算物品相似度矩阵

```
'''计算物品的相似矩阵'''
similarity_matrix = pd.DataFrame(np.zeros((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 = []
        # userRatingPairCount = 0  # 两件物品均评过分的用户数
        if itemID != otheritemID:
            for userID in users:  # 遍历用户-物品评分数据
                userRatings = users[userID]
```

```
                if itemID in userRatings and otheritemID in
userRatings:
                    # userRatingPairCount += 1
                    vec_item.append(userRatings[itemID])

 vec_otheritem.append(userRatings[otheritemID])
            # 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]
print('similarity_matrix:\n{}'.format(similarity_matrix))
```

output.txt:

```
similarity_matrix:
          A         B         C         D         E
A  0.000000 -0.476731 -0.123091  0.532181  0.969458
B -0.476731  0.000000  0.645497 -0.310087 -0.478091
C -0.123091  0.645497  0.000000 -0.720577 -0.427618
D  0.532181 -0.310087 -0.720577  0.000000  0.581675
E  0.969458 -0.478091 -0.427618  0.581675  0.000000
```

3. 计算前n个相似的用户

同理得到与物品5相似的前n个物品，并计算出最终得分

python_code:

```
'''得到与物品5相似的前n个物品，计算出最终得分来'''
n = 2
similarity_items =
similarity_matrix['E'].sort_values(ascending=False)
[:n].index.tolist()
print('similarity_items:\n{}'.format(similarity_items))
```

output.txt:

```
similarity_items:
['A', 'D']
```

4. 计算最终得分（Alice 对物品5的打分）

python_code:

```
'''得到与物品5相似的前n个物品，计算出最终得分来'''
n = 2
similarity_items =
similarity_matrix['E'].sort_values(ascending=False)
[:n].index.tolist()
print('similarity_items:\n{}'.format(similarity_items))

base_score = np.mean(list(items['E'].values()))
weighted_scores = 0.
corr_values_sum = 0.
for item in similarity_items:  # ['A', 'D']
    corr_value = similarity_matrix['E'][item]  #  物品之间的
相似性
    mean_item_score = np.mean(list(items[item].values()))
 # 每个物品的打分平均值
    weighted_scores += corr_value * (users[1][item] -
mean_item_score)
    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:\n{}'.format(final_scores))
user_df.loc[1]['E'] = final_scores
print('user_df:\n{}'.format(user_df))
```

output.txt:

```
用户Alice对物品5的打分:
4.6
user_df:
     A    B    C    D    E
1  5.0  3.0  4.0  4.0  4.6
2  3.0  1.0  2.0  3.0  3.0
3  4.0  3.0  4.0  3.0  5.0
4  3.0  3.0  1.0  5.0  4.0
5  1.0  5.0  5.0  2.0  1.0
```

基于用户协同过滤的完整代码参考 ItemCF_test.py 和 ItemCF.py

itemCF在更多场景被使用

六、算法评估

对UserCF和ItemCF统一说明评测指标

1. 召回率

令用户u推荐的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$T(u)$

召回率定义为：

$$\text{Recall} = \frac{\sum_u |R(u) \cap T(u)|}{\sum_u |T(u)|}$$

理解：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， 考察模型推荐的全面性

2. 准确率

$$\text{Precision} = \frac{\sum_u \mid R(u) \cap T(u)|}{\sum_u |R(u)|}$$

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

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

$$\text{Coverage } = \frac{|\bigcup_{u \in U} R(u)|}{|I|}$$

最终的推荐列表中包含多大比例的物品

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。

七、协同过滤算法存在的问题

1. 泛化能力弱

无法将两个物品相似的信息推广到其他物品的相似性上

> 热门物品具有很强的头部效应，与大量物品产生相似
>
> 尾部物品由于特征向量稀疏，很少被推荐

解决方案：2006年，矩阵分解技术（Matrix Factorization MF被提出）

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，从而挖掘用户和物品的隐含兴趣和隐含特征。

2. 信息利用率低

完全没有利用物品或用户本身的属性，虽然简单高效，但是遗漏了很多有效信息，不能充分利用其他特征数据：用户年龄，性别，商品描述，商品分类，当前时间，地点等用户特征，物品特征，上下文特征

八、课后思考

1. 什么时候使用UserCF，什么时候使用ItemCF，为什么

> - UserCF基于用户相似度推荐，具有更强的社交性特性，适用于用户少，物品多，时效性强场合：如新闻推荐--新闻兴趣分散，及时性，热点性更重要；有利于发现新信息，找到用户自己没察觉的兴趣；**用户少，时效强**

- 兴趣稳定，个性化推荐；物品少，用户多，用户兴趣持久，物品更新慢；艺术品，音乐，电影

2. 上面两种相似度计算方法有什么优劣

皮尔逊相关系数对比cosine相似度解决了用户打分习惯偏置的问题