机器学习算法 day03_协同过滤推荐算法及应用

课程大纲

协同过滤推荐算法原理	协同过滤推荐算法概述
	协同过滤推荐算法思想
	协同过滤推荐算法分析
	协同过滤推荐算法要点
	协同过滤推荐算法实现
协同过滤推荐算法案例	案例需求
	数据规整
	参数设定
	用 Scikili 机器学习算法库实现
	算法检验
	实现推荐
协同过滤推荐算法补充	计算距离的数学公式
	协同过滤算法常见问题

课程目标：

1、理解协同过滤算法的核心思想

2、理解协同过滤算法的代码实现

3、掌握协同过滤算法的应用步骤：数据处理、建模、运算和结果判定

1. CF 协同过滤推荐算法原理

1.1 概述

什么是协同过滤 (Collaborative Filtering，简称 CF)？
首先想一个简单的问题，如果你现在想看个电影，但你不知道具体看哪部，你会怎么做？大部分的人会问问周围的朋友，看看最近有什么好看的电影推荐，而我们一般更倾向于从口味比较类似的朋友那里得到推荐。这就是协同过滤的核心思想。

协同过滤算法又分为**基于用户**的协同过滤算法和**基于物品**的协同过滤算法

1.2 案例需求

如下数据是各用户对各文档的偏好：

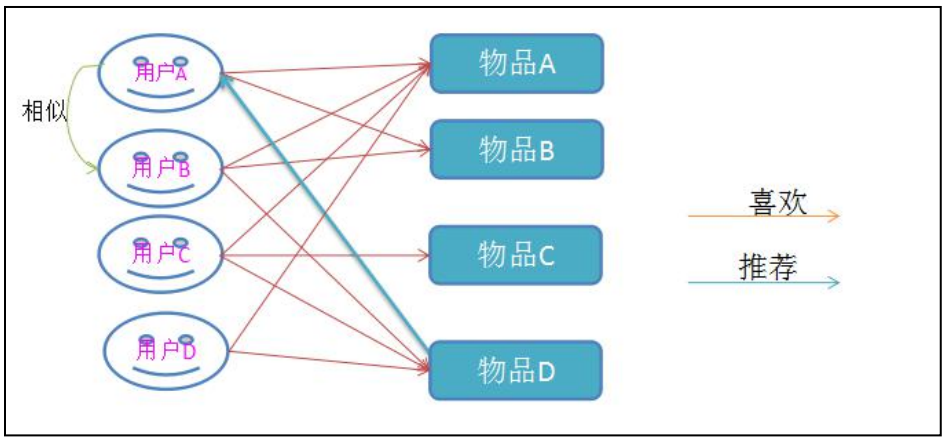
用户/文档	文档 A	文档 B	文档 C	文档 D
用户 A	√	√	推荐？	推荐？
用户 B	√	√		√
用户 C	√		√	√
用户 D	√			√

现在需要基于上述数据，给 A 用户推荐一篇文档

1.3 算法分析

1.3.1 基于用户相似度的分析

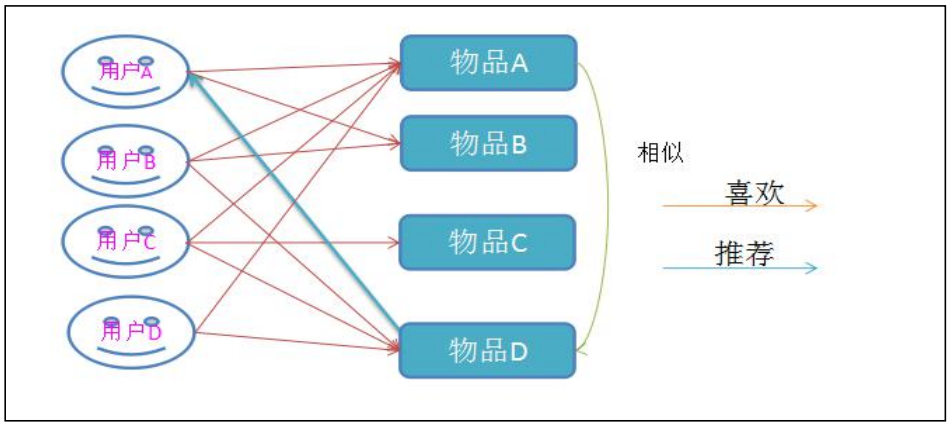
直觉分析：“用户 A/B”都喜欢物品 A 和物品 B，从而**“用户 A/B”的口味最为相近**
因此，为“用户 A”推荐物品时可参考“用户 B”的偏好，从而推荐 D



这种就是基于用户的协同过滤算法 UserCF 指导思想

1.3.2 基于物品相似度的分析

直觉分析：物品组合(A,D)被同时偏好出现的次数最多，因而可以认为 **A/D 两件物品的相似度最高**，从而，可以为选择了 A 物品的用户推荐 D 物品



这种就是基于物品的协同过滤算法 ItemCF 指导思想

1.4 算法要点

1.4.1、指导思想

这种过滤算法的有效性基础在于：

1、用户偏好具有相似性，即用户可分类。这种分类的特征越明显，推荐准确率越高

2、物品之间具有相似性，即偏好某物品的人，都很可能也同时偏好另一件相似物品

1.4.2、两种 CF 算法适用的场景

什么情况下使用哪种算法推荐效果会更好？

不同环境下这两种理论的有效性也不同，应用时需做相应调整。

a. 如豆瓣上的文艺作品，用户对其的偏好程度与用户自身的品位关联性较强；适合 UserCF

b. 而对于电子商务网站来说，商品之间的内在联系对用户的购买行为影响更为显著。

1.5 算法实现

总的来说，要实现协同过滤，需要一下几个步骤：

1. 收集用户偏好
2. 找到相似的用户或物品
3. 计算推荐

1.5.1 收集用户偏好

用户有很多方式向系统提供自己的偏好信息，而且不同的应用也可能大不相同，下面举例进行介绍：

用户行为	类型	特征	作用
评分	显式	整数量化值[0,n]	可以得到精确偏好
投票	显式	布尔量化值 0|1	可以得到精确偏好
转发	显式	布尔量化值 0|1	可以得到精确偏好
保存书签	显式	布尔量化值 0|1	可以得到精确偏好
标记书签 Tag	显式	一些单词	需要进一步分析得到偏好
评论	显式	一些文字	需要进一步分析得到偏好

点击流	隐式	一组点击记录	需要进一步分析得到偏好
页面停留时间	隐式	一组时间信息	噪音偏大，不好利用
购买	隐式	布尔量化值 0|1	可以得到精确偏好

1.5.2 原始偏好数据的预处理

❖ 用户行为识别/组合

在一般应用中，我们提取的用户行为一般都多于一种，关于如何组合这些不同的用户行为，比如，可以将用户行为分为“查看”和“购买”等等，然后基于不同的行为，计算不同的用户 / 物品相似度。

类似于当当网或者京东给出的“购买了该图书的人还购买了 ...”，“查看了图书的人还查看了 ...”

❖ 喜好程度加权

根据不同行为反映用户喜好的程度将它们进行加权，得到用户对于物品的总体喜好。

一般来说，显式的用户反馈比隐式的权值大，但比较稀疏，毕竟进行显示反馈的用户是少数；同时相对于“查看”，“购买”行为反映用户喜好的程度更大，但这也因应用而异。

❖ 数据减噪和归一化。

① 减噪：用户行为数据是用户在使用应用过程中产生的，它可能存在大量的噪音和用户的误操作，我们可以通过经典的数据挖掘算法过滤掉行为数据中的噪音，这样可以是我们的分析更加精确。

② 归一化：如前面讲到的，在计算用户对物品的喜好程度时，可能需要对不同的行为数据进行加权。但可以想象，不同行为的数据取值可能相差很大，比如，用户的查看数据必然比购买数据大的多，如何将各个行为的数据统一在一个相同的取值范围中，从而使得加权求和得到的总体喜好更加精确，就需要我们进行归一化处理。最简单的归一化处理，就是将各类数据除以此类中的最大值，以保证归一化后的数据取值在 [0,1] 范围中。

❖ 形成用户偏好矩阵

一般是二维矩阵，一维是用户列表，另一维是物品列表，值是用户对物品的偏好，一般是 [0,1] 或者 [-1, 1] 的浮点数值。

1.5.3 找到相似用户或物品

当已经对用户行为进行分析得到用户喜好后，我们可以根据用户喜好计算相似用户和物品，然后基于相似用户或者物品进行推荐，这就是最典型的 CF 的两个分支：基于用户的 CF 和基于物品的 CF。这两种方法都需要计算相似度，下面我们先看看最基本的几种计算相似度的方法。

1.5.4 相似度的计算

相似度的计算，现有的几种基本方法都是基于向量（Vector）的，其实也就是计算两个向量的距离，距离越近相似度越大。
在推荐的场景中，在用户 - 物品偏好的二维矩阵中，我们可以将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，或者将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度。

CF 的常用方法有三种，分别是欧式距离法、皮尔逊相关系数法、余弦相似度法。
为了测试算法，给出以下简单的用好偏好数据矩阵：
行表示三名用户，列表示三个品牌，对品牌的喜爱度按照 1~5 增加。

用户	苹果	小米	魅族
zhangsan	5	5	2
Lisi	3	5	4
wangwu	1	2	5

（1）欧氏距离法

就是计算每两个点的距离，比如 Nike 和 Sony 的相似度 。
数值越小，表示相似度越高。

```
def OsDistance(vector1, vector2):
    sqDiffVector = vector1-vector2
    sqDiffVector=sqDiffVector**2
    sqDistances = sqDiffVector.sum()
    distance = sqDistances**0.5
    return distance
```

（2）皮尔逊相关系数

两个变量之间的相关系数越高，从一个变量去预测另一个变量的精确度就越高，这是因为相关系数越高，就意味着这两个变量的共变部分越多，所以从其中一个变量的变化就可越多地获知另一个变量的变化。如果两个变量之间的相关系数为 1 或-1，那么你完全可由变量 X 去获知变量 Y 的值。

- 当相关系数为 0 时，X 和 Y 两变量无关系。
- 当 X 的值增大，Y 也增大，正相关关系，相关系数在 0.00 与 1.00 之间
- 当 X 的值减小，Y 也减小，正相关关系，相关系数在 0.00 与 1.00 之间
- 当 X 的值增大，Y 减小，负相关关系，相关系数在-1.00 与 0.00 之间

  当 X 的值减小，Y 增大，负相关关系，相关系数在-1.00 与 0.00 之间

相关系数的绝对值越大，相关性越强，相关系数越接近于 1 和-1，相关度越强，相关系数越接近于 0，相关度越弱。
在 python 中用函数 corrcoef 实现，具体方法见参考资料

（3）余弦相似度

通过测量两个向量内积空间的夹角的余弦值来度量它们之间的相似性。0 度角的余弦值是 1，而其他任何角度的余弦值都不大于 1;并且其最小值是-1。从而两个向量之间的角度的余弦值确定两个向量是否大致指向相同的方向。两个向量有相同的指向时，余弦相似度的值为 1；两个向量夹角为 90°时，余弦相似度的值为 0；两个向量指向完全相
反的方向时，余弦相似度的值为-1。在比较过程中，向量的规模大小不予考虑，仅仅考虑到向量的指向方向。余弦相似度通常用于两个向量的夹角小于 90°之内，因此余弦相似度的值为 0 到 1 之间。

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$$

```
def cosSim(inA,inB):
    num = float(inA.T*inB)
    denom = la.norm(inA)*la.norm(inB)
    return 0.5+0.5*(num/denom)
```

*注：本课程的实战案例基于**皮尔逊相关系数法**实现*

1.5.3 计算推荐

UserCF 基于用户相似度的推荐

计算推荐的过程其实就是 KNN 算法的计算过程

ItemCF 基于物品相似度的推荐

算法思路
1. 构建物品的同现矩阵
2. 构建用户对物品的评分矩阵
3. 通过矩阵计算得出推荐结果

推荐结果=用户评分矩阵*同现矩阵

实质：计算各种物品组合的出现次数

	101	102	103	104	105	106	107		U3		R
101	5	3	4	4	2	2	1		2.0		40.0
102	3	3	3	2	1	1	0		0.0		18.5
103	4	3	4	3	1	2	0	X	0.0	=	24.5
104	4	2	3	4	2	2	1		4.0		40.0
105	2	1	1	2	2	1	1		4.5		26.0
106	2	1	2	2	1	2	0		0.0		16.5
107	1	0	0	1	1	0	1		5.0		15.5

2. CF 协同过滤算法 Python 实战

2.1 电影推荐需求

根据一个用户对电影评分的数据集来实现**基于用户相似度**的协同过滤算法推荐，相似度的算法采用皮尔逊相关系数法

数据样例如下：

用户 ID：电影 ID：评分：时间

```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
```

2.2 算法实现

本案例使用的数据分析包为 pandas，Numpy 和 matplotlib

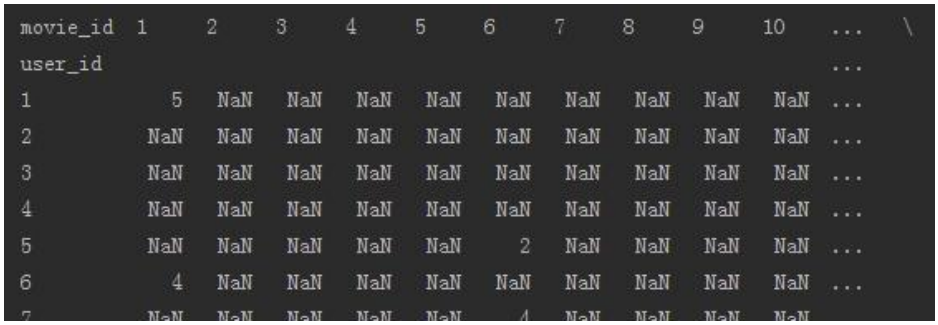
2.2.1 数据规整

首先将评分数据从 ratings.dat 中读出到一个 DataFrame 里：

```
>>> import pandas as pd
>>> from pandas import Series,DataFrame
>>> rnames = ['user_id','movie_id','rating','timestamp']
>>> ratings = pd.read_table(r'ratings.dat',sep='::',header=None,names=rnames)
>>> ratings[:3]
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968

[3 rows x 4 columns]
```

ratings 表中对我们有用的仅是 user_id、movie_id 和 rating 这三列，因此我们将这三列取出，放到一个以 user 为行，movie 为列，rating 为值的表 data 里面。

```
>>> data = ratings.pivot(index='user_id',columns='movie_id',values='rating')  #形成一个透视表
>>> data[:5]
```



movie_id	1	2	3	4	5	6	7	8	9	10	...	\
user_id											...	
1	5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
5	NaN	NaN	NaN	NaN	NaN	2	NaN	NaN	NaN	NaN	...	
6	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
7	NaN	NaN	NaN	NaN	NaN	4	NaN	NaN	NaN	NaN		

可以看到这个表相当得稀疏，填充率大约只有 5%，接下来要实现推荐的第一步是计算 user 之间的相关系数

2.2.2 相关度测算

DataFrame 对象有一个很亲切的方法：

.corr(method='pearson', min_periods=1) 方法，可以对所有列互相计算相关系数。

其中：

method 默认为皮尔逊相关系数，

min_periods 参数，这个参数的作用是设定计算相关系数时的最小样本量，低于此值的一对列将不进行运算。**这个值的取舍关系到相关系数计算的准确性，**因此有必要先来确定一下这个参数。

2.2.3 min_periods 参数测定

测定这样一个参数的基本方法:

- 统计在 min_periods 取不同值时，***相关系数的标准差大小，越小越好***;

但同时又要考虑到，我们的样本空间十分稀疏，min_periods 定得太高会导致出来的结果集太小，所以只能选定一个折中的值。

这里我们测定评分系统标准差的方法为：

- 在 data 中挑选一对重叠评分最多的用户，用他们之间的相关系数的标准差去对整体标准差做点估计。

在此前提下对这一对用户在不同样本量下的相关系数进行统计，观察其标准差变化。

首先，要找出重叠评分最多的一对用户。我们新建一个以 user 为行列的方阵 foo，然后挨

个填充不同用户间重叠评分的个数：

```
>>> foo = DataFrame(np.empty((len(data.index),len(data.index)),dtype=int),index=data.index,columns=data.index)
#print(empt.shape): (6040, 6040)
>>> for i in foo.index:
        for j in foo.columns:
            foo.ix[i,j] = data.ix[i][data.ix[j].notnull()].dropna().count()
```

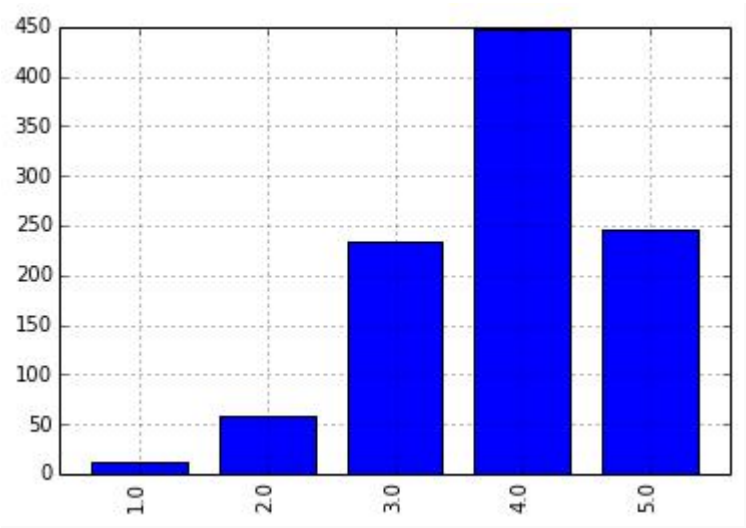
这段代码特别费时间，因为最后一行语句要执行 4000*4000 = 1600 万遍；

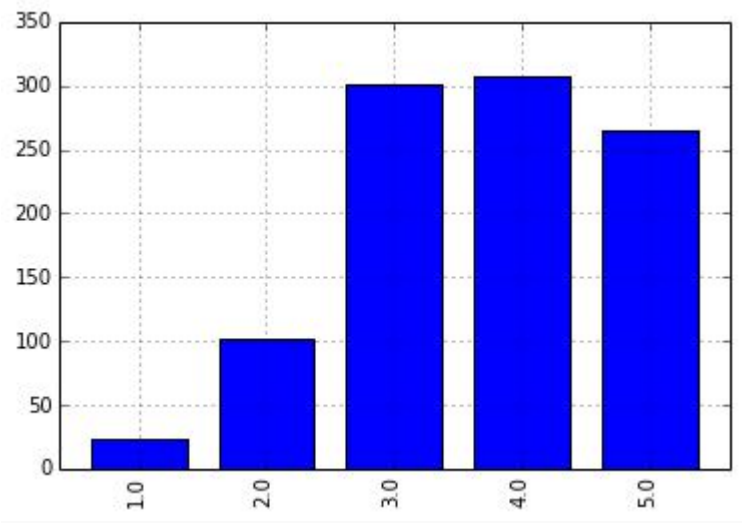
找到的最大值所对应的行列分别为 424 和 4169，这两位用户之间的重叠评分数为 998：

```
>>> for i in foo.index:
        foo.ix[i,i]=0     #先把对角线的值设为 0
>>> ser = Series(np.zeros(len(foo.index)))
>>> for i in foo.index:
        ser[i]=foo[i].max()     #计算每行中的最大值

>>> ser.idxmax()     #返回 ser 的最大值所在的行号
4169

>>> ser[4169]       #取得最大值
998

>>> foo[foo==998][4169].dropna()     #取得另一个 user_id
424         4169
Name: user_id, dtype: float64
```

把 424 和 4169 的评分数据单独拿出来，放到一个名为 test 的表里，另外计算了一下这两个用户之间的相关系数为 0.456，还算不错，另外通过柱状图了解一下他俩的评分分布情况：

```
>>> data.ix[4169].corr(data.ix[424])
0.45663851303413217
>>> test = data.reindex([424,4169],columns=data.ix[4169][data.ix[424].notnull()].dropna().index)
>>> test
movie_id   2     6     10    11    12    17 ...
424            4     4     4     4     1     5 ...
4169           3     4     4     4     2     5 ...

>>> test.ix[424].value_counts(sort=False).plot(kind='bar')
>>> test.ix[4169].value_counts(sort=False).plot(kind='bar')
```





对这俩用户的相关系数统计，我们分别随机抽取 20、50、100、200、500 和 998 个样本值，各抽 20 次。并统计结果：

```
>>> periods_test = DataFrame(np.zeros((20,7)),columns=[10,20,50,100,200,500,998])
>>> for i in periods_test.index:
        for j in periods_test.columns:
            sample = test.reindex(columns=np.random.permutation(test.columns)[:j])
            periods_test.ix[i,j] = sample.iloc[0].corr(sample.iloc[1])


>>> periods_test[:5]
          10        20        50        100       200       500       998
0 -0.306719  0.709073  0.504374  0.376921  0.477140  0.426938  0.456639
1  0.386658  0.607569  0.434761  0.471930  0.437222  0.430765  0.456639
2  0.507415  0.585808  0.440619  0.634782  0.490574  0.436799  0.456639
3  0.628112  0.628281  0.452331  0.380073  0.472045  0.444222  0.456639
4  0.792533  0.641503  0.444989  0.499253  0.426420  0.441292  0.456639

[5 rows x 7 columns]
```

```
>>> periods_test.describe()
                10          20          50          100         200         500    #998
略
count   20.000000   20.000000   20.000000   20.000000   20.000000   20.000000
mean     0.346810    0.464726    0.458866    0.450155    0.467559    0.452448
std      0.398553    0.181743    0.103820    0.093663    0.036439    0.029758
min     -0.444302    0.087370    0.192391    0.242112    0.412291    0.399875
25%      0.174531    0.320941    0.434744    0.375643    0.439228    0.435290
50%      0.487157    0.525217    0.476653    0.468850    0.472562    0.443772
75%      0.638685    0.616643    0.519827    0.500825    0.487389    0.465787
max      0.850963    0.709073    0.592040    0.634782    0.546001    0.513486

[8 rows x 7 columns]
```

从 std 这一行来看，理想的 min_periods 参数值应当为 200 左右(标准差和均值、极值最接近)。

2.2.3 算法检验

为了确认在 min_periods=200 下本推荐算法的靠谱程度，最好还是先做个检验。
具体方法为：在评价数大于 200 的用户中随机抽取 1000 位用户，每人随机提取一个评价另存到一个数组里，并在数据表中删除这个评价。然后基于阉割过的数据表计算被提取出的 1000 个评分的期望值，最后与真实评价数组进行相关性比较，看结果如何。

```
>>> check_size = 1000
>>> check = {}
>>> check_data = data.copy()     #复制一份 data 用于检验，以免篡改原数据
>>> check_data = check_data.ix[check_data.count(axis=1)>200]   #滤除评价数小于 200 的用户
>>> for user in np.random.permutation(check_data.index):
        movie = np.random.permutation(check_data.ix[user].dropna().index)[0]
        check[(user,movie)] = check_data.ix[user,movie]
        check_data.ix[user,movie] = np.nan
        check_size -= 1
        if not check_size:
            break


>>> corr = check_data.T.corr(min_periods=200)
>>> corr_clean = corr.dropna(how='all')
>>> corr_clean = corr_clean.dropna(axis=1,how='all')   #删除全空的行和列
>>> check_ser = Series(check)     #这里是被提取出来的 1000 个真实评分
>>> check_ser[:5]
(15, 593)        4
(23, 555)        3
```

```
(33, 3363)      4
(36, 2355)      5
(53, 3605)      4
dtype: float64
```

接下来要基于 corr_clean 给 check_ser 中的 1000 个用户-影片对计算评分期望。

计算方法为：对与用户相关系数大于 0.1 的其他用户评分进行加权平均，权值为相关系数

```
>>> result = Series(np.nan,index=check_ser.index)
>>> for user,movie in result.index:     #这个循环看着很乱，实际内容就是加权平均而已
        prediction = []
        if user in corr_clean.index:
            corr_set = corr_clean[user][corr_clean[user]>0.1].dropna()     #仅限大于 0.1 的用户
        else:continue
        for other in corr_set.index:
            if    not np.isnan(data.ix[other,movie]) and other != user:#注意 bool(np.nan)==True
                prediction.append((data.ix[other,movie],corr_set[other]))
        if prediction:
            result[(user,movie)] = sum([value*weight for value,weight in prediction])/sum([pair[1] for pair in prediction])


>>> result.dropna(inplace=True)
>>> len(result)#随机抽取的 1000 个用户中也有被 min_periods=200 刷掉的
862
>>> result[:5]
(23, 555)       3.967617
(33, 3363)      4.073205
(36, 2355)      3.903497
(53, 3605)      2.948003
(62, 1488)      2.606582
dtype: float64
>>> result.corr(check_ser.reindex(result.index))
0.436227437429696
>>> (result-check_ser.reindex(result.index)).abs().describe()#推荐期望与实际评价之差的绝对值
count       862.000000
mean          0.785337
std           0.605865
min           0.000000
25%           0.290384
50%           0.686033
75%           1.132256
max           3.629720
dtype: float64
```

862 的样本量能达到 0.436 的相关系数，应该说结果还不错。如果一开始没有滤掉评价数小于 200 的用户的话，那么首先在计算 corr 时会明显感觉时间变长，其次 result 中的样本量会很小，大约 200+个。但因为样本量变小的缘故，相关系数可以提升到 0.5~0.6 。

另外从期望与实际评价的差的绝对值的统计量上看，数据也比较理想。

2.2.4 实现推荐

在上面的检验，尤其是平均加权的部分做完后，推荐的实现就没有什么新东西了。
首先在原始未阉割的 data 数据上重做一份 corr 表：

```
>>> corr = data.T.corr(min_periods=200)
>>> corr_clean = corr.dropna(how='all')
>>> corr_clean = corr_clean.dropna(axis=1,how='all')
```

我们在 corr_clean 中随机挑选一位用户为他做一个推荐列表：

```
>>> lucky = np.random.permutation(corr_clean.index)[0]
>>> gift = data.ix[lucky]
>>> gift = gift[gift.isnull()]       #现在 gift 是一个全空的序列
```

最后的任务就是填充这个 gift：

```
>>> corr_lucky = corr_clean[lucky].drop(lucky)#lucky 与其他用户的相关系数 Series，不包含 lucky 自身
>>> corr_lucky = corr_lucky[corr_lucky>0.1].dropna()     #筛选相关系数大于 0.1 的用户
>>> for movie in gift.index:     #遍历所有 lucky 没看过的电影
        prediction = []
        for other in corr_lucky.index:    #遍历所有与 lucky 相关系数大于 0.1 的用户
            if not np.isnan(data.ix[other,movie]):
                prediction.append((data.ix[other,movie],corr_clean[lucky][other]))
        if prediction:
            gift[movie] = sum([value*weight for value,weight in prediction])/sum([pair[1] for pair in prediction])


>>> gift.dropna().order(ascending=False)     #将 gift 的非空元素按降序排列
movie_id
3245        5.000000
2930        5.000000
2830        5.000000
2569        5.000000
1795        5.000000
981         5.000000
696         5.000000
```

682	5.000000
666	5.000000
572	5.000000
1420	5.000000
3338	4.845331
669	4.660464
214	4.655798
3410	4.624088
…	
2833	1
2777	1
2039	1
1773	1
1720	1
1692	1
1538	1
1430	1
1311	1
1164	1
843	1
660	1
634	1
591	1
56	1

Name: 3945, Length: 2991, dtype: float64

3、CF 协同过滤算法补充

3.1、计算距离的数学公式

欧几里德距离（Euclidean Distance）
最初用于计算欧几里德空间中两个点的距离，假设 x，y 是 n 维空间的两个点，它们之间的欧几里德距离是：

$$d(x,y)=\sqrt{(\sum(x_i-y_i)^2)}$$

可以看出，当 n=2 时，欧几里德距离就是平面上两个点的距离。

当用欧几里德距离表示相似度，一般采用以下公式进行转换：距离越小，相似度越大

$$sim(x,y)=\frac{1}{1+d(x,y)}$$

皮尔逊相关系数（Pearson Correlation Coefficient）

皮尔逊相关系数一般用于计算两个定距变量间联系的紧密程度，它的取值在 [-1，+1] 之间。

$$p(x,y)=\frac{\sum x_i y_i-n\overline{xy}}{(n-1)s_x s_y}=\frac{n\sum x_i y_i-\sum x_i\sum y_i}{\sqrt{n\sum x_i^2-(\sum x_i)^2}\sqrt{n\sum y_i^2-(\sum y_i)^2}}$$

sx, sy 是 x 和 y 的样品标准偏差。

Cosine 相似度（Cosine Similarity）

Cosine 相似度被广泛应用于计算文档数据的相似度：

$$T(x,y)=\frac{x\bullet y}{\|x\|^2\times\|y\|^2}=\frac{\sum x_i y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}}$$

Tanimoto 系数（Tanimoto Coefficient）

Tanimoto 系数也称为 Jaccard 系数，是 Cosine 相似度的扩展，也多用于计算文档数据的相似度：

$$T(x,y)=\frac{x\bullet y}{\|x\|^2+\|y\|^2-x\bullet y}=\frac{\sum x_i y_i}{\sqrt{\sum x_i^2}+\sqrt{\sum y_i^2}-\sum x_i y_i}$$

相似邻居的计算

介绍完相似度的计算方法，下面我们看看如何根据相似度找到用户-物品的邻居，常用的挑选邻居的原则可以分为两类：

1) 固定数量的邻居：K-neighborhoods 或者 Fix-size neighborhoods

不论邻居的“远近”，只取最近的 K 个，作为其邻居。

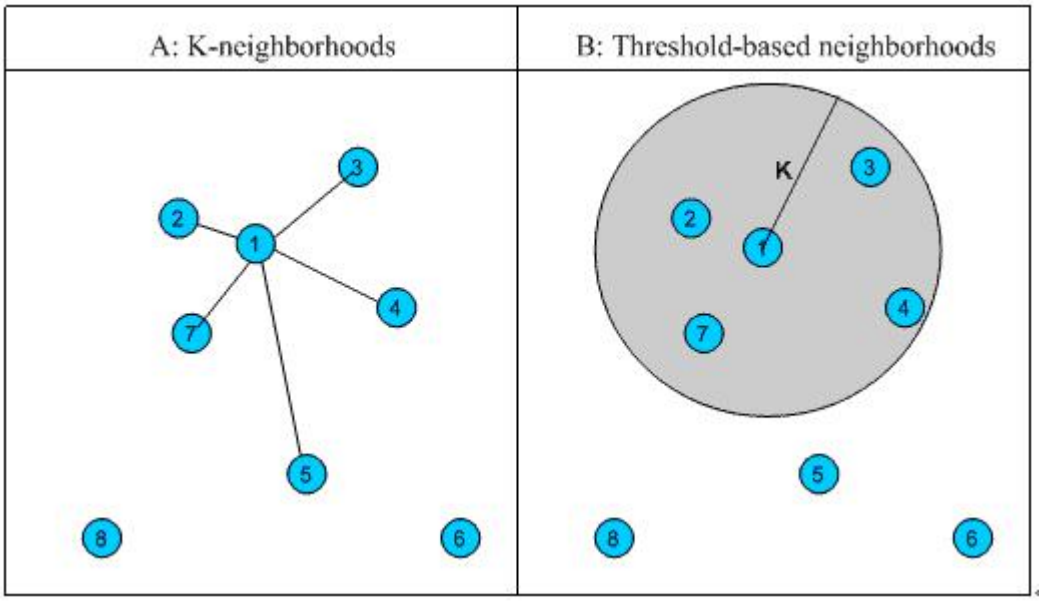
如下图中的 A，假设要计算点 1 的 5- 邻居，那么根据点之间的距离，我们取最近的 5 个点，分别是点 2，点 3，点 4，点 7 和点 5。但很明显我们可以看出，这种方法对于孤立点的计算效果不好，因为要取固定个数的邻居，当它附近没有足够多比较相似的点，就被迫取一些不太相似的点作为邻居，这样就影响了邻居相似的程度，比如下图中，点 1 和点 5 其实并不是很相似。

2) 基于相似度门槛的邻居：Threshold-based neighborhoods

与计算固定数量的邻居的原则不同，基于相似度门槛的邻居计算是对邻居的远近进行最大值的限制，落在以当前点为中心，距离为 K 的区域中的所有点都作为当前点的邻居，这种方

法计算得到的邻居个数不确定，但相似度不会出现较大的误差。
如下图中的 B，从点 1 出发，计算相似度在 K 内的邻居，得到点 2，点 3，点 4 和点 7，这种方法计算出的邻居的相似度程度比前一种优，尤其是对孤立点的处理。

图：相似邻居计算示意图



3.2、协同过滤算法常见问题

虽然协同过滤是一种比较省事的推荐方法，但在某些场合下并不如利用元信息推荐好用。协同过滤会遇到的两个常见问题是

1) 稀疏性问题——因用户做出评价过少，导致算出的相关系数不准确
2) 冷启动问题——因物品获得评价过少，导致无“权”进入推荐列表中

都是样本量太少导致的（上例中也使用了至少 200 的有效重叠评价数）。
因此在对于新用户和新物品进行推荐时，使用一些更一般性的方法效果可能会更好。比如：

- 给新用户推荐更多平均得分超高的电影；
- 把新电影推荐给喜欢类似电影（如具有相同导演或演员）的人。

后面这种做法需要维护一个物品分类表，这个表既可以是基于物品元信息划分的，也可是通过聚类得到的。