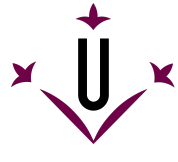
Recommender systems based on item-to-item collaborative filtering

Ramon Béjar Torres

Data mining - Master on Computer Science



Universitat de Lleida

In this notebook we consider how to build recommender systems based on item-to-item (where items are the products of the on-line store) collaborative filtering.

The approach to recommend new items to an user is based on finding SIMILAR items to the ones already bought by the user, or similar to the ones we know the user likes or is interested in. So, what we use in this kind of recommender systems is a way to compare two items, but based on the information from the set of users that bought those products.

This approach is a good one when the user-item matrix represents which products were bought by each user, but the typical user row vector is sparse (there are few purchases with respect to the total number of items of the on-line store).

This item-centered approach allows us to compute recommendations for any single item bought by any user.

Even in the extreme case where the user has only bought one item, the system will be able to compute good recommendations if the total number of purchases of the on-line store is high enough.

We can also consider not only which items the user bought, but also wich items he has browsed in the online store frequently, or any other source of implicit knowledge about the interests of the user.

In any case, the main issue here will be **how to compare items of the on-line store** and how to predict wich ones will be more interesting for the user.

So, with respect to the approach based on latent factors:

- Here we also perform global filtering because we consider the whole set of users and items to build the system
- But we compute explicitely similarity measures between any pair of products, instead of decomposing users and products as vectors of latent factors to predict the matching between users and products.

Comparing items based on set of common costumers

The first step towards discovering (mining) similar items is how to measure the similarity between two items in the store catalog.

If we assume that the information we have for every item is the set of users that bought that item, the global filtering approach is based on comparing ALL the costumers of the store, checking how many costumers have bought both the two items we want to compare.

> *The idea is that the similarity measure should be higher when the two items have a higher number of common costumers.*

So, a very direct way of measuring the distance would be to count the number of common costumers, and normalize it by the total number of costumers of the store.

The cosine distance

However, when we consider user-item matrices that incorporate more complex information about user-item pairs, for example the satisfaction degree of the costumer with the item, it is better to consider other kinds of similarity measures. One widely used distance measure is the cosine distance.

Given the angle θ between two product vectors v1 and v2, the cosine distance is defined as the cosine of the angle θ, that we can compute from the components of both vectors as follows:

$$cosdis(v_1, v_2) = cos(\theta) = \frac{v_1 \cdot v_2}{\sqrt{\sum_i v_1[i]^2}\sqrt{\sum_i v_2[i]^2}}$$

Some key properties:

- The value ranges in [-1,1]
- -1 means they are opposite vectors ($\theta = 180$)
- 0 means they are orhogonal vectors ($\theta = 90$)
- 1 means they are equal or proportional (same direction) ($\theta = 0$)

We can compute this measure only when both vectors are non-zero (the module of any of them is not zero)

So, when used as a similarity measure, observe the cosine distance can signal from totally different products (-1) to totally similar ones (1)

For working on item-to-item based collaborative filtering, we must work with the columns of the user-item matrix, the same matrix that we used with global filtering based on finding latent factors.

From now on, we will assume that items are already given as vectors, representing their corresponding column vectors in the user-item matrix.

In [2]:
```
#  Compute the cosine distance between vectors vec1 and vec2, represented
#  as dense lists: with all the elements (non-zero and zero) values present
def CosineDistance( vec1, vec2 ):
    dot = 0.0
    v1rs = 0.0
    v2rs = 0.0
    for i in range(len(vec1)):
        dot += (vec1[i]*vec2[i])
        v1rs += (vec1[i]*vec1[i])
        v2rs += (vec2[i]*vec2[i])
    v1rs = math.sqrt(v1rs)
    v2rs = math.sqrt(v2rs)
    return dot/(v1rs*v2rs)
```

However, working with sparse vectors is a better approach if we consider that the fraction of users that have bought a product will be low.

Binary feature vectors

Consider for example the following set of 4 item vectors, each one indicating which costumers, from a total set of 4 costumers, have bought (1) or have not bought (0) the item

In [3]:
```
itemvectors = [ [1,1,0,0], [0,1,0,1], [0,1,0,0], [1,0,1,0] ]
```

In [4]:
```
for i1, vec1 in enumerate(itemvectors):
    for i2, vec2 in enumerate(itemvectors):
        if (i1 < i2):
          print ("Cosine distance between ", i1, "and", i2, ":", CosineDistance
( vec1, vec2 ))
```

```
Cosine distance between  0 and 1 : 0.4999999999999999
Cosine distance between  0 and 2 : 0.7071067811865475
Cosine distance between  0 and 3 : 0.4999999999999999
Cosine distance between  1 and 2 : 0.7071067811865475
Cosine distance between  1 and 3 : 0.0
Cosine distance between  2 and 3 : 0.0
```

Negative and positive features vectors

As we have said, the cosine distance is also defined when our item vectors contain values in a range that includes negative values. This is the case where:

- negative values mean *negative* ratings
- positive values mean positive ratings
- 0 mean a neutral rating (or no rating at all). For example, consider the following set of items.

In [5]:
```
itemvectors2 = [ [-3,-3,-3,-2], [-2,-2,-2,-1], [1,1,1,1], [3,3,3,3] ]
```

In [6]:
```
for i1, vec1 in enumerate(itemvectors2):
    for i2, vec2 in enumerate(itemvectors2):
        if (i1 < i2):
          print ("Cosine distance between ", i1, "and", i2, ":", CosineDistance
( vec1, vec2 ))
```

```
Cosine distance between  0 and 1 : 0.996270962773436
Cosine distance between  0 and 2 : -0.987829161147262
Cosine distance between  0 and 3 : -0.987829161147262
Cosine distance between  1 and 2 : -0.9707253433941511
Cosine distance between  1 and 3 : -0.9707253433941511
Cosine distance between  2 and 3 : 1.0
```

Observe that in this case, the cosine distance ranges from -1 to 1, where -1 means *totally opposite vectors*, and 1 means totally aligned vectors, without giving relevance to the magnitude of the vectors.

Similarity in the users-movies example

Let's consider what information the cosine distance provides when comparing movies in our users-movies example:

In [7]:
```
# We have 10 users, and 10 movies: STW1, STW2, STW3, STW4, STW5, STW6
#                                  T1, T2, T3 and BaT
usersandmovies = [ [3,3,3,5,5,4, 3,3,-1, -1], \
                   [3,3,3,5,5,4, 4,2,0, -1], \
                   [3,3,4,5,5,4, 4,4,1, 0], \
                   [4,3,3,4,5,4, 3,3,1, -1], \
                   [1,1,1,0,1,1, 5,4,2, -1], \
                   [1,2,1,0,1,1, 4,4,2, -1], \
                   [1,2,2,1,1,1, 4,4,2, -1], \
                   [1,2,2,1,1,0, 5,4,3, -1], \
                   [-2,-3,-2,0,-2,-1, 0,0,-1,4], \
                   [-2,-3,-2,0,-2,-1, 0,0,-1,4]   ]
```

In [8]:
```
def getMovieVector( usersandmovies, j ):
    return [ usersandmovies[u][j] for u in range(len(usersandmovies))]
```

Similarity between the first four star wars movies:

In [9]:
```
swmovies = [ getMovieVector( usersandmovies, j ) for j in range(4)]
for i1, vec1 in enumerate(swmovies):
    for i2, vec2 in enumerate(swmovies):
        if (i1 < i2):
          print ("Cosine distance between ", i1, "and", i2, ":", CosineDistance
( vec1, vec2 ))
```

```
Cosine distance between  0 and 1 : 0.9554528219538876
Cosine distance between  0 and 2 : 0.9668114046189884
Cosine distance between  0 and 3 : 0.8808819893600309
Cosine distance between  1 and 2 : 0.9698160576680458
Cosine distance between  1 and 3 : 0.772771255454225
Cosine distance between  2 and 3 : 0.8762691935871222
```

Similarity between terminator movies:

In [10]:
```
tmovies = [ getMovieVector( usersandmovies, j ) for j in [6,7,8]]
for i1, vec1 in enumerate(tmovies):
    for i2, vec2 in enumerate(tmovies):
        if (i1 < i2):
          print ("Cosine distance between ", i1, "and", i2, ":", CosineDistance
( vec1, vec2 ))
```

```
Cosine distance between  0 and 1 : 0.9824666109905364
Cosine distance between  0 and 2 : 0.7681373347487839
Cosine distance between  1 and 2 : 0.7767356373806175
```

Similarity between the first four star wars movies and terminator movies:

In [11]:
```
for i1,vec1 in enumerate(swmovies):
    for i2,vec2 in enumerate(tmovies):
        print ("Cosine distance between ", i1, "and", i2, ":", CosineDistance(
vec1, vec2 ))
```

```
Cosine distance between  0 and 0 : 0.7393877297305068
Cosine distance between  0 and 1 : 0.73431307102251
Cosine distance between  0 and 2 : 0.4495530025457533
Cosine distance between  1 and 0 : 0.7762444233578819
Cosine distance between  1 and 1 : 0.7741809609998153
Cosine distance between  1 and 2 : 0.5989849814784503
Cosine distance between  2 and 0 : 0.8135251376128875
Cosine distance between  2 and 1 : 0.8113625732861212
Cosine distance between  2 and 2 : 0.552422157047008
Cosine distance between  3 and 0 : 0.6859384573602509
Cosine distance between  3 and 1 : 0.6673778622698389
Cosine distance between  3 and 2 : 0.18302666282596894
```

So observe that many star wars movies can be considered to be *similar* to many terminator movies (from the point of view of our set of users). But not all such pairs are clearly similar. For example:

- Similarity for (SW1,T3) : 0.44
- Similarity for (SW4,T3) : 0.18

Similarity between Star Wars I and Breakfast at Tiffanys:

In [12]:
```
vec1, vec2 = getMovieVector( usersandmovies, 0 ), getMovieVector( usersandmovie
s, 9 )
print ("Cosine distance between 0 and 1 :", CosineDistance( vec1, vec2 ))
```

```
Cosine distance between 0 and 1 : -0.6477502756312957
```

So, in this case we get a negative value, because it seems that Star Wars and Breakfast at Tiffanys are very opposite movies...

Mining similar items based on item-to-item global filtering

Let's now consider the approach for recommending products to the Amazon costumers presented in the paper:

> *Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-Item Collaborative Filtering. In IEEE INTERNET COMPUTING. 2003*

In that paper you will find only the overall idea. The approach is the one we have explained before, use some similarity measure between products to recommend relevant products. We do not know what are the current similarity measures used by Amazon, but we will use the cosine distance in this notebook to develop our recommender system.

The main building block of the Amazon recomender system is their algorithm to compute similarity between any pair of items in their on-line catalog. This is the pseudo-code of the mentioned Amazon item-to-item similarity mining algorithm:

```
def computeSimilarityBetweenProducts( I ):
 for each item i1 in product catalog I:
    for each customer C who purchased i1:
       for each item i2 purchased by customer C:
          record that *a customer* (C) purchased i1 and i2:
             store that (i1,i2) were purchased by a same user
    for each item i2 in product catalog I:
        compute the similarity between i1 and i2
```

This algorithm can be though as an **off-line** algorithm, the similarity between products should be computed as a background process, and only be recomputed when there are a significant number of changes in the purchases database. What is the worst-case and real complexity of this algorithm ?

Consider the following observations (where M is the number of costumers and N the number of items of the on-line store:

1. The worst-case complexity is $O(N^2 M)$. This is the case when almost any user has bought any item of the store.
2. However, if we assume that many costumers have very few purchases (let's say a constant number), the real complexity is more closer to $O(N\ M)$.
3. The complexity can be further reduced if we only consider a sample (subset) of costumers to compute the similarity between products. Of course, this will produce an approximation of the real similarity values between products.

For computing the similarity, we can use the cosine distance, or any other similarity measure we think is good for our application domain. Observe that in some sense, it is reasonable to think that:

> *the similarity between a pair of products (i1,i2) will be a number proportional to the total number of customers that purchased both i1 and i2*,

Distributed computation

If we only record whether an user buys a product (1/0 binary feature vectors), then observe that the similarity measure depends only on the total number of such customers, and not the particular ones.

So, the computation needed to compute the similarity between i1 and i2 can be thought as some kind of *reduce* (by Key) operation between all pairs (i1,i2) produced by different users C. That is, if we consider (i1,i2) as the key, and for example "1" as the value for each different user C that bought i1 and i2, the (key,value) to produce would be:

```
((i1,i2), 1)  for each user C that bought i1 and i2
```

and then reduce by key (for example summing up the values) all such (key,value) pairs. Of course, to get a normalized similarity measure the sum should be divided by the maximun number of users.

The output format of the data set computed by such global filtering algorithm could be something like the following. For each item I we have a list of (item,similarity) pairs:

I : [(j1,sim(I,j1), (j2,sim(I,j2), ..., (jI,sim(I,jI)]

where the set of items j1, j2, ... , jI is the set of items that have at least one common costumer (user) with I and so their similarity is > 0. We will refer to the (distributed) data set that contains such information for all the items as the **rddSimilarityPairs** in the rest of this notebook.

If we instead consider the distributed computation of the cosine distance, in the more general case with negative and positive feature values, the computation could be as follows:

1. Map every pair of user-product ratings **with the same user u** to the values they contribute in the final cosine distance between p1 and p2:
$$(u, p_1, r_1), (u, p_2, r_2) \rightarrow ((p_1, p_2), (r_1 r_2, r_1^2, r_2^2))$$
2. Reduce all the previous key-value pairs, with the same key as:
$$((p_1, p_2), (pra_{1,2}, ra_1^2, ra_2^2)) + ((p_1, p_2), (prb_{1,2}, rb_1^2, rb_2^2)) \rightarrow$$
$$((p_1, p_2), (pra_{1,2} + prb_{1,2}, ra_1^2 + rb_1^2, ra_2^2 + rb_2^2))$$
3. Compute the cosine distance combining the reduced values in a final map:
$$((p_1, p_2), (\sum_u r_1 r_2, \sum_u r_1^2, \sum_u r_2^2)) \rightarrow \frac{\sum_u r_1 r_2}{\sqrt{\sum_u r_1^2}\sqrt{\sum_u r_2^2}}$$

Observe that the previous solution is not necessarily the most efficient way to compute the cosine distance:

- It produces redundant information: info for (p_1, p_2) will be the same one as the info for (p_2, p_1). This could be fixed filtering the pairs we combine in step 1 to those with $p_1 < p_2$.
- The sum of squares $\sum_u r_i^2$ for item i is computed as many times as different pairs of items we build with i. So, to save some computation, the term $\sum_u r_i^2$ could be instead computed only once, and then reused when computing the cosine distance between item i and any other item j.

Recommending similar items to a previously bought one

As a first recommender system, consider the case where we want to recommend similar items to one just bought by an user, or to one recently browsed by the user. So, we want to focus on similar items to a particular one.

Here you have a possible pseudo-code for a recommender system for that particular case. The input is the user we are considering and the item we want to use as the base item to get the recommendations. Observe that the primary use of this algorithm would be "on-line": every time an user makes a new purchase or browses a new item it would be desirable to get such focused recommendations.

```
def RecommendKmostSimilar( RDD rddSimilarityPairs, user U, item I, int K ):
     rdd1 = getSimilarItems( rddSimilarityPairs, I )
     rdd2 = EraseItemsAlreadyBought( rdd1, U )
     rdd3 = rdd2.sortBySimilarity()
     bestK = rdd3.takeFirstKItems( K )

     return bestK
```

This function assumes that we have a previously computed data set, the rddSimilarityPairs, that should be the one computed by the similarity mining algorithm of the previous section (or by any other algorithm that provides such data set).

Implementation

Let's see a possible implementation in spark of the four steps of the previous algorithm. Let's first consider the following similarity information data set, that for simplicity we will consider that is stored in a plain ASCII file. In a final application this similarity information would be stored in a database.

Observe that if we assume that the similarity between products does not change significantly very frequently, we can compute the similarity in a off-line algorithm, and only run on-line with an RDD that contains only the similarity between the current target product I and the other products with similarity greater than zero.

1 2,0.6 4,0.3

2 1,0.6

3 4,0.7 5,0.8

4 3,0.7 5,0.4 1,0.3

5 3,0.8 4,0.4

6 7,0.3

7 6,0.3

0. Load similarity info from file to RDD

Before computing recommendations, we load the file similarityPairsInfo_1.txt to get the desired RDD data set

In [14]:
```
rddSimilarityPairs = sc.textFile('similarityPairsInfo_1.txt').map( parseSimilar
ityInfo )
```

Let's take a look to check if the file was well parsed:

In [15]:
```
rddSimilarityPairs.collect()
```

Out[15]:
```
[(1, [(2, 0.6), (4, 0.3)]),
 (2, [(1, 0.6)]),
 (3, [(4, 0.7), (5, 0.8)]),
 (4, [(3, 0.7), (5, 0.4), (1, 0.3)]),
 (5, [(3, 0.8), (4, 0.4)]),
 (6, [(7, 0.3)]),
 (7, [(6, 0.3)])]
```

Observe that this format for storing similarity info is efficient towards finding the complete info for an item, but has some redundancy in the information.

1. Filter similarity info related to item I

Next, we can filter from such rdd data set only the similarity information related to our input item I:

In [16]:
```
def getSimilarItems( rddSimilarityPairs, I ):
   return  rddSimilarityPairs.filter( lambda x : x[0] == I )
```

Let's test the function with item 4:

In [17]:
```
rdd1 = getSimilarItems( rddSimilarityPairs, 4 )
rdd1.collect()
```

Out[17]:
```
[(4, [(3, 0.7), (5, 0.4), (1, 0.3)])]
```

2. Filter elements not already bought by user U

The next step is to retain only items not bought by user U. Again, the implementation is highly dependent on whether we assume the set of purchases of user U fits into a single machine or it must be distributed. Let's assume here that his/her ser of purchases fits into one machine, so we can read it into the following function. We assume this format:

user1 item11 item12 ... item1N

user2 item11 item12 ... item1N

. . .

userN item11 item12 ... item1N

where the set of items in line i is the set of items bought by user i

In [18]:
```
# In this function we remove the set of already items by U, and also
# remove the item key to get the final set
# similar and filtered items as a single list
def RemoveBoughtItems( itemsSimilarToI, U ):
   purchases = getPurchases( U )
   print ( " Purchases by user ", U, " : ", purchases )
   return itemsSimilarToI.flatMap( lambda x : [it for it  in x[1] if it[0] not
in purchases ] )
```

We will consider the next example file of user purchases ('purchases.txt'):

```
1    2  3
2    3  4
3    4  5  6
4    6  1
```

Let's check it filtering out the purchases of user 3:

In [20]:
```
rdd2 = RemoveBoughtItems( rdd1, 3 )
rdd2.collect()
```

```
 Purchases by user  3  :  [4, 5, 6]
```

Out[20]:
```
[(3, 0.7), (1, 0.3)]
```

3. Sort them and take the most K similar items

The last two steps are sort the resulting set of items by similarity and taking the first k items. We can do this with spark with a single action:

In [21]:
```
# Take only the first most similar non-bought item
rdd2.takeOrdered(1, key=lambda x: -x[1])
```

Out[21]:
```
[(3, 0.7)]
```

The meaning of empty entries

Observe that in the cosine distance formula:

$$cosdis(v_1, v_2) = cos(\theta) = \frac{v_1 \cdot v_2}{\sqrt{\sum_i v_1[i]^2}\sqrt{\sum_i v_2[i]^2}}$$

the contribution of user i will be 0 if the entries $v_1[i]$ and $v_2[i]$ are equal to 0. That is, it would be like simplifying the vectors eliminating the entry i.

But then, in case the values represent ratings like in the movies example, what should be the contribution of that entry if the value is not 0, but empty (user i did not give ratings for movies v_1 and v_2)?

Assuming 0 means empty rating

For example, assume the two modified vectors for star wars 1 and star wars 4, where the 0 represents NO rating (instead of neutral ratings):

In [25]:
```
sw1 = getMovieVector( usersandmovies, 0 )
sw4 = getMovieVector( usersandmovies, 3 )
sw1[4] = sw1[5] = 0
print ("Cosine distance between sw1 ", sw1, "and sw4 ", sw4, ":", CosineDistanc
e( sw1, sw4 ))
```

```
Cosine distance between sw1  [3, 3, 3, 4, 0, 0, 1, 1, -2, -2] and sw4  [5, 5,
5, 4, 0, 0, 1, 1, 0, 0] : 0.8973484983270362
```

If we increase the empty entries that coincide for a same user:

In [26]:
```
sw1[8] = sw1[9] = 0
print ("Cosine distance between sw1 ", sw1, "and sw4 ", sw4, ":", CosineDistanc
e( sw1, sw4 ))
```

```
Cosine distance between sw1  [3, 3, 3, 4, 0, 0, 1, 1, 0, 0] and sw4  [5, 5, 5,
4, 0, 0, 1, 1, 0, 0] : 0.9738516810963532
```

So, having less ratings can induce a higher value for the cosine distance, when the non-empty entries are very similar, even if they are very few with respect to the total number

of users.

Recommending based on the whole set of previous purchases

Sometimes Amazon also sends emails to costumers sending recommendations based on *all* (or a subset of) his/her previous purchases, instead of based on a single recent one.

Observe that in this case, a same item could be recommended but with different similarity values, as the result of being similar to different purchases of the given user (but with a different similarity value with each purchase).

That is, to have for a particular item i, that we consider it for recommendation, a set of different similarity values:

$$(i, p_1, s_1), (i, p_2, s_2), \ldots, (i, p_m, s_m)$$

if item i was found similar to different previous purchases $p_1, p_2, \ldots, p_m$ of the user

So, we should **aggregate** in some way the different similarity values

$$s_1, s_2, \ldots, s_m$$

obtained for a same item. How can we aggregate the different similarities between a given product and a set of different products ?

- We could use the average/median value of the set similarity values for a same item
- We could use some kind of weighted sum, that gives more relevance to items where the similarity info was computed using the info of more users

  $$wSum = \sum_i w_i s_i$$

  where w_i represents the weight (relevance) of the similarity value s_i towards computing the final value. Observe that setting $w_i = 1/m$ gives the regular average value, that gives all the values the same relevance

Measuring similarity in other ways

As we have seen, the key component of this kind of recommender systems is the way to compute similarity between items. We have presented a very common one, the cosine distance, but there are some others, such as the Jaccard distance or the Pearson correlation coefficient.

Here you have a good source of information about different ways to measure similarity for recommender systems:

> *"A new user similarity model to improve the accuracy of collaborative filtering" by Haifeng Liu, Zheng Hu, Ahmad Mian, Hui Tian, Xuzhen Zhu.*
> *https://doi.org/10.1016/j.knosys.2013.11.006 (https://doi.org/10.1016/j.knosys.2013.11.006)*