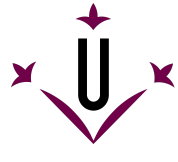
Recommender systems based on collaborative filtering and latent factors

Ramon Béjar Torres

Data mining - Master on Computer Science



Universitat
de Lleida

We are going to discuss an approach to build a recommender system based on using the whole information stored in the user-product ranking matrix. That is, a matrix where:

$$M[i,j] = \begin{cases} \text{ranking information about user i and product j} \\ Nan \text{ if such ranking is not yet known} \end{cases}$$

We assume that such matrix gives **explicit rankings** provided by users to the products they have bought.

Because we use the whole information in such matrix to predict unknown entries, we say that we follow a collaborative (or global) filtering approach.

The approach based on latent factors, is based on two complementary assumptions:

1. User i can be encoded as a vector of c feature values, that encode the preferences of the user for different categories/kinds of products.
2. Product j can also be encoded as a vector of c feature values, that encode the weight that the product has in each one of these categories.

So, how much an user likes a product will be measured as the **dot product** of these two vectors.

U-V matrix factorization for latent factors discovery

The approach to discover latent factors that characterize users and products, is the one based on U-V matrix factorization. The problem formulation is that we want to find a factorization of our user-product matrix as the product of two matrices U and V, where:

1. Each user will be represented as a row vector with c factors in matrix U, so U will be a matrix with m rows and c columns.
2. Each product will be represented as a column vector with c factors in matrix V, so V will be a matrix with c rows and n columns.

Then, given our user-product rating matrix, with m rows (users) and n columns (products) we want to find matrices U and V such:

$$\overbrace{\begin{pmatrix} U_{1,1} & \cdots & U_{1,c} \\ & \cdots & \\ & \vdots & \\ U_{m,1} & \cdots & U_{m,c} \end{pmatrix}}^{U\,(m\times c)} \times \overbrace{\begin{pmatrix} V_{1,1} & \cdots & V_{1,n} \\ & \cdots & \\ & \vdots & \\ V_{c,1} & \cdots & V_{c,n} \end{pmatrix}}^{V\,(c\times n)} = \overbrace{\begin{pmatrix} M_{1,1} & \cdots & M_{1,n} \\ & \cdots & \\ & \vdots & \\ M_{m,1} & \cdots & V_{m,n} \end{pmatrix}}^{M(m\times n)}$$

Remember that in general the matrix M may have empty entries, but our U-V factorization will provide values for all the entries, so when we say an U-V factorization is good for a partially filled matrix M, we usually mean that it agrees with the value for the filled entries of M. Once we have this factorization, observe that the value of any entry (i, j) of the matrix M, even for unknown entries, is predicted multiplying the row i of matrix U by column j of matrix V, that is:

$$\hat{M}(i, j) = \sum_{k=1}^{c} U_{i,k} * V_{k,j}$$

Because it may be not possible to find such **exact** factorization with the desired number of latent factors, finding such factorization is actually presented as an optimization problem, where the goal is to find a factorization with the smallest RMSE error (error computed over the filled entries of M):

$$\mathrm{RMSE}(U, V, M) = \sqrt{\frac{\sum_{i,j}(U(row_i) \cdot V(col_j) - M(i,j))^2}{\#\ known\ entries}}$$

where the summatory is over entries (i, j) of the matrix M with known values.

However, because we are really more interested in being able to **predict unknown values** of the matrix M, than in predicting the known ones, to avoid **overfitting** to the known values and have a high error rate for unknown entries, the optimization algorithms usually also allow to consider alternative objective functions that incorporate regularized terms to control the tradeoff between:

- RMSE error
- Prediction of unknown entries.

In general, the regularized terms have the goal to penalize too complex models that are very sensitive to small changes in the input. That is, to penalize models where many values of row_i (user i) or $column_j$ (product j) are away from zero.

Optimization algorithms for U-V matrix factorization

We may consider different algorithms for finding a good U-V factorization (with a small RMSE error or any other generalized error function). These are the two main ones:

1. Gradient descend: Using the same approach we presentend for finding a linear model, but this time applied to the parameters of our model (the coefficients of the matrices U and V)

2. Alternating Least Squares: If we keep fixed one of the two matrices (U or V), the problem becomes a quadratic optimization problem that can be optimally solved in P-time. So, we perform an iterative process where we fix one of them, find the optimal one for the other, and in the next iteration we exchange the roles: the second one is fixed and the first one is optimized. This process is repeated until we reach a fixed point.

To know more about this problems and their solving algorithms, a good source of information is the paper:

> *Yehuda Koren, Robert Bell and Chris Volinsky. Matrix factorization techniques for recommender systems. In Computer Journal, IEEE press, Vol 42(8), 2009. https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf (https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf)*

and also the book about data mining for big data we recommended for this course.

Let's consider the same example matrix we used in our notebook about clustering algorithms. In particular, let's consider a data base of users, where for each user we store the ratings given by the user to different movies. We first consider the matrix full of entries, but later we consider variations of this matrix where some of the entries will be empty.

In [4]:
```
# Example data
#
# We have 10 users, and 10 movies: STW1, STW2, STW3, STW4, STW5, STW6
#                                  T1, T2, T3 and BaT
# Each entry i,j is the rating given by the user in the range [-5.0,5.0]
# We can observe that we have 4 clear Star Wars fans (that they also like a
# little bit Terminator movies)
# We also have four clear Terminator fans (that they also like a little STWs mo
vies)
# Finally, we have two clear Breakfast at tiffannies fans (BaT), that they do n
ot
# like too much science-fiction movies

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 [5]:
```
from pyspark.mllib.recommendation import ALS, MatrixFactorizationModel, Rating

# We need a function to convert our matrix format to a RDD of
# pyspark.mllib.recommendation.Rating objects:
#
#    Rating(int(userid),int(productid),float(rating)
#
# BEWARE: this function works with the whole matrix in the driver memory,
# obtaining a python list representation of the ratings to finally get the
# RDD. A better (more scalable) version should do this from a RDD with the matr
ix entries
# loaded from a source file, not from a python matrix in main memory

def convertMatrixToRatings( matrix ):
    ratings = []
    for user,userrow in enumerate(matrix):
        for product,productrating in enumerate(userrow):
            ratings.append( Rating( int(user) , int(product), float(productrati
ng) ) )
    return ratings
```

In [6]:

```
ratings = sc.parallelize( convertMatrixToRatings( usersandmovies ) )

# Build the recommendation model using Alternating Least Squares
rank = 3
numIterations = 10
model = ALS.train(ratings, rank, numIterations)
```

In [7]:

```
# Evaluate the model on training data
# First, get the data without the rating values (only user-product IDs)
testdata = ratings.map(lambda p: (p[0], p[1]))

# Next, Get the predictions obatined with our U-V factorization model
predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))
# join ((u,p), V) and ((u,p), W) to get ((u,p), (V, W))
ratesAndPreds = ratings.map( lambda r: ((r[0], r[1]), r[2]) ).join(predictions)

## Compute Mean Square error
MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
print("Mean Squared Error = " + str(MSE))
```

```
Mean Squared Error = 0.08856746050130206
```

In [8]:
```
# We can also get the U-V factorization, as the set of user features (latent fa
ctors) of
# the U matrix

print (" Users latent factors:")
for userfactors in model.userFeatures().sortByKey().collect():
    print (userfactors)
```

```
 Users latent factors:
(0, array('d', [-0.8929557204246521, 1.3610984086990356, -0.650032043457031
2]))
(1, array('d', [-0.7475835680961609, 1.4006414413452148, -0.429874211549758
9]))
(2, array('d', [-0.8469208478927612, 1.4760514497756958, 0.2404583543539047
2]))
(3, array('d', [-0.34039148688316345, 1.4697706699371338, -0.1793191432952880
9]))
(4, array('d', [0.6612030863761902, 0.7702144980430603, 2.4479691982269287]))
(5, array('d', [0.8096885681152344, 0.8089669942855835, 2.1741440296173096]))
(6, array('d', [0.6402640342712402, 0.8823555707931519, 2.0161702632904053]))
(7, array('d', [0.8778318166732788, 0.9204967617988586, 2.5905468463897705]))
(8, array('d', [-1.7340666055679321, -0.9847530126571655, 0.2968831658363342
3]))
(9, array('d', [-1.7340666055679321, -0.9847530126571655, 0.2968831658363342
3]))
```

In [9]:
```
# and the set of product features of the V matrix:
print ("\n Products latent factors:")
for productfactors in model.productFeatures().sortByKey().collect():
    print (productfactors)
```

```
 Products latent factors:
(0, array('d', [-0.09388906508684158, 2.150222063064575, -0.313575178384780
9]))
(1, array('d', [0.39158087968826294, 2.266158103942871, -0.2143813520669937]))
(2, array('d', [-0.15328878164291382, 2.194775342941284, -0.088012576103210
5]))
(3, array('d', [-1.4750709533691406, 2.553581714630127, -0.2178601771593094]))
(4, array('d', [-0.667518675327301, 3.071117639541626, -0.45548638701438904]))
(5, array('d', [-0.7910909652709961, 2.3469951152801514, -0.274608999490737
9]))
(6, array('d', [-1.0103960037231445, 2.2092952728271484, 1.427252173423767]))
(7, array('d', [-0.8621616959571838, 1.9098454713821411, 1.2768445014953613]))
(8, array('d', [0.41049206256866455, 0.4889748692512512, 0.6707598567008972]))
(9, array('d', [-1.4981082677841187, -1.190435528755188, 0.5192508697509766]))
```

Can you identify significant differences between the latent factors for different user groups and product groups ? Observe that the two "Breakfast at Tiffany's" fans have clearly different latent factors than the others users

Let's next check what happens if there are some missing values in the user-product matrix. Consider the following function to randomly erase k entries from each row of the matrix:

Prediction model with incomplete data

Let's consider now the extraction of some of the elements of the user-product matrix. Consider randomly eliminating k elements from each row.

In [20]:
```
def convertMatrixToRatingsWithBlanks( matrix, k ):
    ratings = []
    for user,userrow in enumerate(matrix):
        size = len(userrow)
        blanks = random.sample(range(size), k)
        for product,productrating in enumerate(userrow):
            if (product not in blanks):
              ratings.append( Rating( int(user) , int(product), float(productra
ting) ) )
    return ratings
```

In [70]:
```
ratings2 = sc.parallelize( convertMatrixToRatingsWithBlanks( usersandmovies, 4
) )

# Build the recommendation model using Alternating Least Squares
rank = 3
numIterations = 10
model2 = ALS.train(ratings2, rank, numIterations)
```

In [72]:
```
# Evaluate the model on training data
# First, get the data without the rating values (only user-product IDs)
testdata2 = ratings2.map(lambda p: (p[0], p[1]))

# Next, Get the predictions obatined with our U-V factorization model
predictions2 = model2.predictAll(testdata2).map(lambda r: ((r[0], r[1]), r[2]))
# join ((u,p), V) and ((u,p), W) to get ((u,p), (V, W))
ratesAndPreds2 = ratings2.map( lambda r: ((r[0], r[1]), r[2]) ).join(prediction
s2)

# Make predictions also for the whole dataset
predictAll = model2.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndPredsAll = ratings.map( lambda r: ((r[0], r[1]), r[2]) ).join(predictAl
l)

## Compute Mean Square error
MSE2 = ratesAndPreds2.map(lambda r: (r[1][0] - r[1][1])**2).mean()
MSEAll = ratesAndPredsAll.map(lambda r: (r[1][0] - r[1][1])**2).mean()

print("Mean Squared Error = " + str(MSE2) +  " Mean Square Error over complete
 data: "+str(MSEAll) )
```

```
Mean Squared Error = 0.011390722198382629 Mean Square Error over complete dat
a: 0.845641482048498
```

Results for different values for k (eliminated entries from each row):

k	MSE	MSE for all data
1	0.075	0.10
2	0.066	0.17
3	0.037	0.33
4	0.011	0.84

But different executions with the same k will give you slightly different results.

So, the error can decrease when there are less entries available to fit a good model, but this does not imply that the model predition on unkown entries will be also better. Actually, a problem of this factorization approach is that to be able to get a good factor vector for a given user, we need enough known entries for that user.

In contrast, there are recommender systems for products in on-line stores based on direct product-to-product comparison that allow to give recommendations for users from any single previously bought (or ranked) product of that user. So, even if the user only made ONE purchase the system will be able to give recommendations, as soon as the total number of user purchases in the on-line store is high enough.

Collaborative Filtering for Implicit Feedback Datasets

We can also consider data where there is no explicit user feedback. That is, when the entry $M(i, j)$ does not contain an explicit rating of the user, but some **aggregation** of values that can be used to infer something about the preference of user i for product j

For example:

- In a web shop, $M(i, j)$ could indicate the number of times that the user i clicked on product j
- In a on-line video shop, $M(i, j)$ could indicate the fraction of watched time for movie j by user i
- We could also give more relevance if the movie was watched with less interruptions.

In general, we can set up $M(i, j)$ to represent some aggregation of indicators such that the more inputs we have, the higher the value of $M(i, j)$, and so the higher our confidence with respect to the preference of user i for product j.

To know more about the ALS algorithm (and other global filtering algorithms) in spark, check the documentation page:

https://spark.apache.org/docs/3.0.0/ml-collaborative-filtering.html (https://spark.apache.org/docs/3.0.0/ml-collaborative-filtering.html)

To know more about the model used when we assume that the matrix contains implicit feedback data, you can check here one possible model:

"Collaborative Filtering for Implicit Feedback Datasets".
http://yifanhu.net/PUB/cf.pdf (http://yifanhu.net/PUB/cf.pdf)

In [ ]: