SmartCat

DATA ⟶ KNOWLEDGE ⟶ POWER

IMPROVED R IMPLEMENTATION OF COLLABORATIVE FILTERING FOR RECOMMENDER SYSTEMS

In this work we present an R implementation of memory-based collaborative filtering which is:

- **SIGNIFICANTLY FASTER (EXECUTION TIME IS DECREASED BY A AN ORDER OF MAGNITUDE)**
- **APPLICABLE TO LARGE DATASETS ON WHICH CLASSIC IMPLEMENTATION MIGHT RUN OUT OF MEMORY.**

WHAT ARE OUR GOALS?

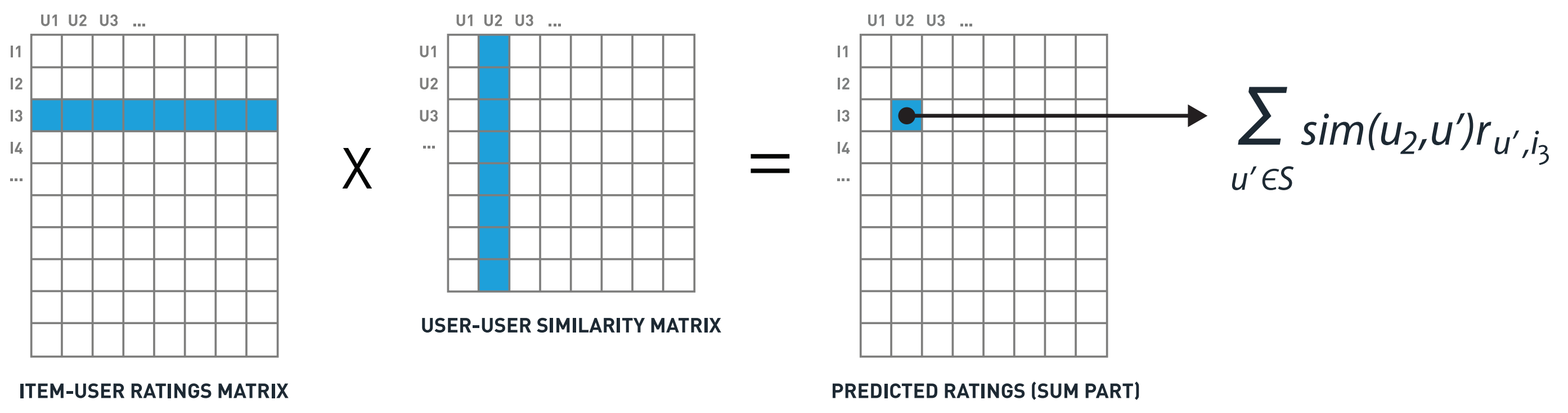
Memory-based collaborative filtering (CF) is a common technique that make recommendations by using information about similarities between users or items:

- USER-BASED CF: To predict how user U will rate item I, we take information about how similar users have rated that item.
- ITEM-BASED CF: To predict how user U will rate item I, we take information about how he has rated similar items.

Here are the exact formulas for user-based CF. In order to calculate prediction about how user u will rate item i($r_{u,i}$), we aggregate over ratings that users similar to **u** have given to item **i**. The more similar a user is, the more influence his rating has to the overall prediction.

$$r_{u,i} = w \sum_{u' \in S} sim(u,u') r_{u',i} \qquad w = \frac{1}{\sum_{u' \in S} |sim(u,u')|}$$

Predictions are usually not calculated in a loop, but rather using matrix multiplication since it is a much faster operation. Here is an example for predicting how user U2 will rate item I3 (blue cells):



$$\sum_{u' \in S} sim(u_2,u') r_{u',i_3}$$

By multiplying two matrices we get instant results for all predictions (not only for U2-I3). Obviously, in case there are a lot of users or items, matrices might become large. Then, the calculation of all predictions require a lot of memory and processing time. Therefore, the main issues of memory-based CF are related to:

- **EXECUTION TIME**
- **SCALABILITY**

We address these issues using new approach and compare it to commonly used R package 'recommenderlab'.

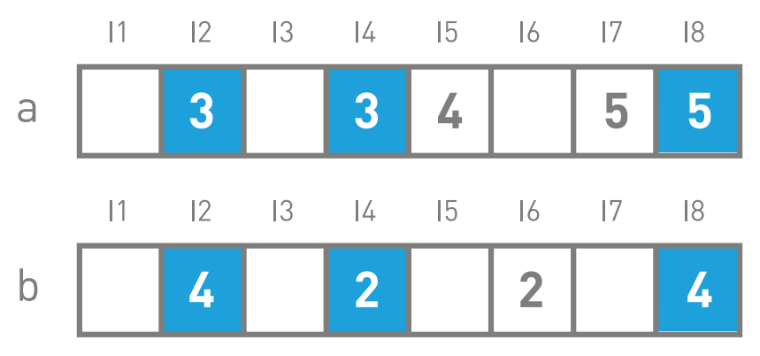
EXECUTION TIME IMPROVEMENT

The main steps used in the implementation of user-based CF are as follows (the same approach applies for item-based CF):

- Take a ratings matrix and optionally normalize ratings.
- Calculate similarities between users.
- Optionally keep k most similar users (k highest values per column) in the user-user similarity matrix.
- Calculate predictions and denormalize them in case normalization was performed in the first step

SPARSENESS ▶ OPTIMIZATION

The characteristic of rating matrices is that they are very sparse (users typically rate only few items, if any). For calculating similarities the key optimization was achieved by using functions optimized for sparse data. Here is an example of how to efficiently calculate correlation using **CROSSPROD** (R function optimized for multiplication of sparse vectors/matrices). The example shows users a and b (items rated by both users are marked as I_{ab}. Final formula can be generalized to matirces.



	I1	I2	I3	I4	I5	I6	I7	I8
a		3		3	4		5	5
b		4		2		2		4

$$cor(a,b) = \frac{\sum_{i \in I_{ab}} (r_{ai} - \overline{r_a}) * (r_{bi} - \overline{r_b})}{\sqrt{\sum_{i \in I_{ab}} (r_{ai} - \overline{r_a})^2} * \sqrt{\sum_{i \in I_{ab}} (r_{bi} - \overline{r_b})^2}} = \frac{\sum_{i \in I_{ab}} (\hat{r_{ai}}) * (\hat{r_{bi}})}{\sqrt{\sum_{i \in I_{ab}} (\hat{r_{ai}})^2} * \sqrt{\sum_{i \in I_{ab}} (\hat{r_{bi}})^2}}$$

$$cor(a,b) = \frac{crossprod(\hat{a},\hat{b})}{\sqrt{crossprod(\hat{a}^2, b! = null)} * \sqrt{crossprod(a! = null, \hat{b}^2)}}$$

Another optimization we made was regarding filtering on k nearest neighbours. We grouped all the values from the similarity matrix by column and applied a function that finds the k-th highest value per column. This was implemented using the R 'data.table' package.

EVALUATION

The comparison of our implementation vs. 'recommenderlab' was performed using the following setup:

- Center normalization and cosine measure to calculate similarities
- 10-fold cross validation
- The evaluation was performed on a single machine with 16 GB RAM

USER-BASED CF:

	Our implementation		Recommenderlab	
	rmse	exec time	rmse	exec time
k=100	0.938	4.7s	0.991	139.5s
k=300	0.931	5.6s	0.999	145.43s
k=ALL	0.932	6.4s	0.985	142.1s

ITEM-BASED CF:

	Our implementation		Recommenderlab	
	rmse	exec time	rmse	exec time
k=100	0.938	6.6s	0.940	302.7s
k=300	0.925	7.7s	0.927	241.6s
k=ALL	0.940	8.7s	1.593	185.4s

COMPARISON ON 100K MOVIELENS DATASET (~ 100, 000 RATINGS)

USER-BASED CF:

	Our implementation		Recommenderlab	
	rmse	exec time	rmse	exec time
k=100	0.902	116.2s	0.964	6321.1s
k=300	0.891	132.7s	0.971	6408.4s
k=ALL	0.901	311.2s	1.006	7403.5s

ITEM-BASED CF:

	Our implementation		Recommenderlab	
	rmse	exec time	rmse	exec time
k=100	0.894	69.7s	0.894	3301.5s
k=300	0.886	77.1s	0.886	3300.3s
k=ALL	0.910	165.4s	1.596	3427.0s

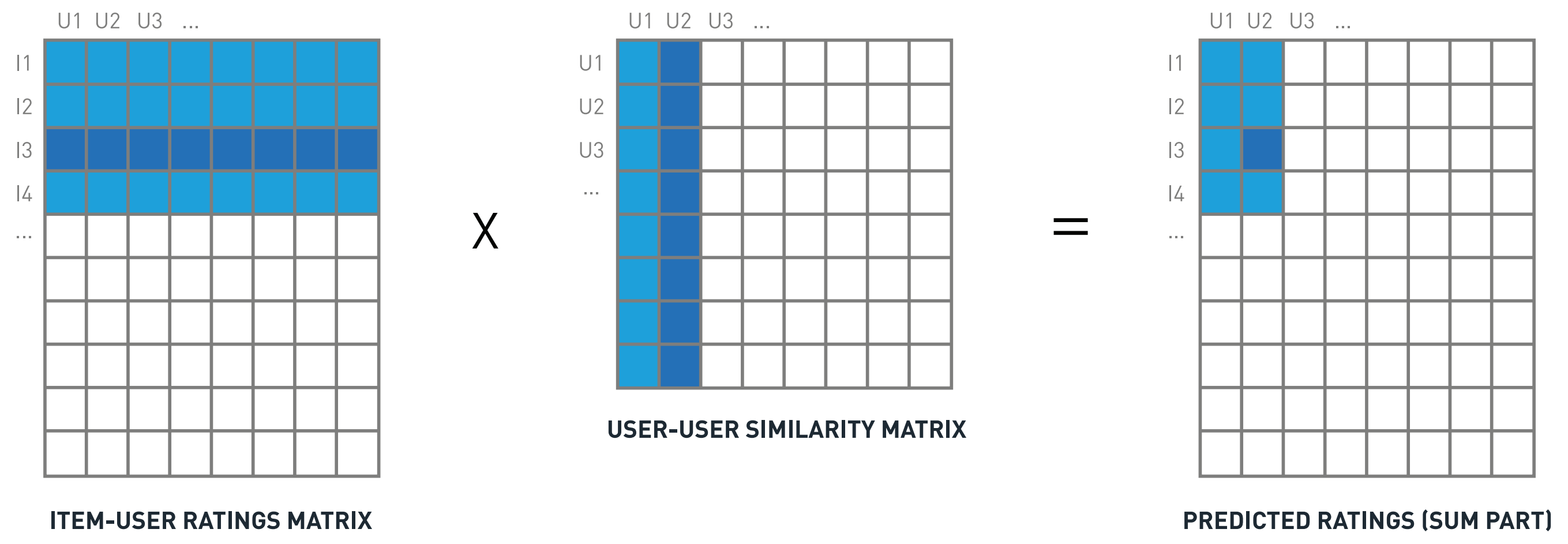
COMPARISON ON 1M MOVIELENS DATASET (~ 1,000,000 RATINGS)

20-50X FASTER

BUILD A RECOMMENDER ON LARGE DATASETS

All algorithms were ran on a single machine with 16 GB RAM, and evaluated using 10-fold cross validation. In such a setup, 'recommenderlab' implementation can not be used on MovieLens 10M dataset (at least for user-based CF, since it runs out of memory when similarities matrix needs to be calculated).

In our implementation, we tried to solve the problem of large matrices by calculating predictions block-by-block. The picture below shows the procedure for user-based CF. In each step we take N items from ratings matrix, M users from similarities matrix, and the product give as a block of predictions for them.



ITEM-BASED CF:

	rmse	exec time	block size - rows (users)	block size - columns (items)
k=100	0.839	10.53m	20 000	6 000
k=1000	0.831	23.33m	20 000	6 000

USER-BASED CF:

	rmse	exec time	block size - rows (users)	block size - columns (items)
k=100	0.851	178.43m	15 000	500
k=1000	0.832	191.86m	15 000	500

RESULTS ON MOVIELENS 10M DATASET (~10,000,000 RATINGS)

With this current implementation, when we need to find recommendations in real-time for one or several users, the calculation of similarities and predictions will be much faster since we will operate on a small number of users. The algorithm can be optimized further, by storing the similarity matrix as a model, rather than calculating it on-fly. An obvious advantage of this algorithm is that it can be parallelized, since we calculate predictions in each block independently.

FINAL NOTE

In this presentation we showed how you can optimize existing implementations of memory-based collaborative filtering, in order to achieve better performance and scalability.
In the near future we plan to work on this project further and extend it with new algorithms. The code is freely available on https://github.com/smartcat-labs/collaboratory.

smartcat.io

Feel free to comment and contribute!

www.SmartCat.io