1 Low Rank Matrix Completion for Recommender Systems

In this article we will talk about the basics of recommender systems, the features of a problem to which low rank matrix completion (LRMC) are particularly suited and the corresponding advantages and disadvantages. In my next article I will go into explicit depth, provide an example algorithm and its implementation in Python along with a guide to implementing it on the GPU using CUPY. While that article will be for those mathematically and technically inclined, this one should be accessible to anyone with a high school background in mathematics.

1.1 The Problem

Imagine for a moment that you have a large collection of people frequently using a service that provides an overwhelming number of products. This can be people streaming music or videos from a platform, looking for interesting news articles, fashion, hardware, or even dates. Recomender systems are information filtering algorithms, they aim to take an overwhelming list of options and reduce it to a much smaller short list of options that are most likely to satisfy the user. Each of the examples I've given have their own peculiarities:

a) If someone watches a movie they really enjoy, they will often look to watch a sequel as soon as possible. If that same person bought a new drill, they probably don't want to buy a new drill so soon: even if it's from a company they really like and has better specs.

b) When people's tastes change quickly like in the case of seasonal fashion, naively assuming prior spending habits hold true can be very dangerous.

c) In the sphere of online dating or any other peer to peer matching service the users are also the products!

Clearly there cannot be a one size fits all solution. To that end let us narrow down the list of problems we are looking to tackle by making some simplifying assumptions.

1. Our users are not also our products

2. Our users' preferences are unlikely to change over the timeframe in which we collect data and make recommendations

3. Our products are not supplementary: a user that likes a product is expected to like similar products

The ultimate ideal is to be able to take any user and any product and predict how much that user likes or dislikes the product. We could for instance arrange this data in a large table (or matrix) with each user along the rows, the products along the columns and their satisfaction populating the enterior:

	LOTR	Mad Max	The Ring	Halloween
Alice	10	0	0	-10
Bob	0	-10	10	0
Charlie	0	10	-10	0
Daniel	-10	0	0	10

Table 1: Example Movie Preferences

It's clear from the data that Alice likes Lord of the Rings while Daniel hates it. If we have access to all this data then all we have to do to make a recommendation is find the highest scoring movie that hasn't already been watched for that particular user. In practise though we have only a small fraction of this data, imagine that most of this table is empty and it is our goal to somehow fill it in. Depending on the application we may not even have a short list of ratings as our starting point.

Example: Suppose our goal is to produce a recomender system for a music streaming platform. While users have the ability to rate songs they usually don't bother to. Instead they listen to play lists they've made, made by others, or skip between virtual radio stations. Instead of relying on just their ratings we can sort all the songs they have listened to in order of play time. If they skip a song the moment it starts we can assume they strongly dislike it while a song they have listened to for hours must be something they like. Using raw play time in seconds as a user score may cause issues depending the algorithm we wish to implement and so those scores may need to be renormalized.

Let us assume that we have gathered all the data we have and it looks something like this:

	LOTR	Mad Max	The Ring	Halloween
Alice	10	-	0 -	-10
Bob	0	-10	-	0
Charlie	0	10	-10	-
Daniel	-	0	0	10

Table 2: Incomplete Movie Preferences

Our goal is to fill in the missing entries as accurately as possible. It turns out that we can fill in all of our missing entries with a shocking amount of precision as long as the following assumptions hold:

a) how much each user likes or dislikes each product is determined predominantly by a small number of independent variables (e.g movie is scary, movie is high budget)

b) for each of those variables, each user has their own independent preference (user likes scary movies, user likes high budget movies)

c) for each of those variables, each product realizes those variables to different degrees (this movie is scary and low budget)

If the number of variables matches the number of movies then we are basically out of luck. In practise however users will often base their decision on only a few variables. There may be many other variables involved but our hope is that their combined contribution is negligible enough that we can make good predictions.

1.2 Algorithm Output

L-Table	Alice	Bob	Charlie	Daniel
var 1	5	5	-5	-5
var 2	-5	5	-5	5

R-Table	LOTR	Mad Max	The Ring	Halloween
var 1	1	-1	1	-1
var 2	-1	-1	1	1

Table 3: Movie Preferences LR Decomposition

We can recover the expected ratings by multiplying the user preferences by the product expressions fore each variable and summing the result

$$\text{Alice LOTR Rating} = \text{Alice.v1} \cdot \text{LOTR.v1} + \text{Alice.v2} \cdot \text{LOTR.v2} = 5 \cdot 1 + (-5) \cdot (-1) = 10$$

Treating these two tables as matrices L and R we can recover the original matrix X by simply computing $X = L^T R$ which amounts to computing the calculation above for each pair of user and product. In this grossly simplified example we had 16 total ratings which we were able to reincode using two tables containing 8 entries each. In practise however we might have 100,000 users and 10,000 products with an estimated 30 random variables. This way we would encode information for 1 billion possible ratings using just 3.03 million variables.

1.3 Interpretting The Results

The LR decomposition described above is not unique. Notice how the ratings don't change if we multiply the first row of L and divide the first row of R by the same number. This makes sense, if we scale up the preference and scale down the expression by the same amount the net result should not change. If we want the results to be unique we need to add some more constraints. We can require that the rows of each component matrix are as uncorellated as possible, that way each variable can be thought of as being independent. Mathematically we would require that each row is orthogonal to every other. There is still a matter of fixing the scale, to that end we can require that the rows of both L and R matrices have the same norm.

After normalizing we can measure the norm of each row of L, this process assigns to each hidden variable a quantity called the singular value. In our case both singular values will end up being 10. The singular value has a very simple interpretation: the larger the singular value, the larger the impact the variable has on the final score. Keep in mind though that this is only on average. If after running our algorithm using an assumption of 10 hidden variables, our singular values ended up being in order: $(100, 93, 92, 81, 1, 0.2, 0.04, 0.001, 0.0007, 0.0003)$, that would suggest that we could use 7 variables instead and get essentially the same result.

In the example above I had chosen var 1 to represent "this movie has a high budget" and var 2 to represent "this movie is scary". We can see that we can use each variable to arrange our movies along an axis. In practise however our variables will rarely lend themselves well to obvious interpretation. The main problem is our requirement that each variable is as independent as possible. The algorithm is much more likely to have var1 be something like "this movie is high budget or very slightly scary" while var2 is "this movie is scary or has as slightly higher budget". In order to make the output independent mathematically we end up making them very mixed to our human understanding. For the example above there are a lot more low budget horror movies so those two concepts are not independent.

Suppose we did have a metric available to sort our products, for example movie budget. We can instantly use the rows of our R table to determine how large of an effect that particular metric has on average user ratings. If w is our normalized product metric (an assignment of one number for each movie), λ_i are our singular values, and v_i are the corrresponding rows of our R matrix (the expressions), then the expected impact is given by

$$I = \sum_i \lambda_i \langle v_i, w \rangle$$

Remember that we are measuring impact rather than goodness, high impact means that it could swing ratings up OR down. The key take away is that we cannot determine what makes a product more or less attractive in general but we can quickly test any such hypothesis.

Due to the symmetry of the problem, we can also interpret the rows of the user matrix L as independent user features and see how much of an impact user traits (e.g age and gender) have on expected preference.

1.4 Key Advantages

Easy Implementation and Scaling: In my next post I will outline an effecient algorithm that can be run on consumer grade GPUs for very large data sets. It is also possible to break up the computation to run on even larger data sets than a GPU would normally allow at the expense of time.

Data requirements grow slower than data size: In order to run the algorithm we need a sufficient number of existing ratings. If the number of users and products are both proportional to N and there are r hidden variables, the amount of existing data we need is proportional to $Nr\log(N)$, if we double the number of users and products we only need on average one more rating per user.

Easy data segmentation: Even if our data happens to be too sparse, we can still run the algorithm on a subset of users and products that do have a higher rating density. We can then leverage the data gathered to better recommend any product to our most prolific users or to better recommend our most popular products to any other user (as long as those users and products have at least r ratings).

Real time recommendations: Once we have a good working model, we can estimate preference for new users and new products once they've gained at least r ratings without having to rerun the algorithm. This computation is instant on 20 year old hardware.

Fast iterations: Over time we will get more rating data (along with new users and products), we can use our old model as a starting point in the algorithm to save a lot of computation time when updating it.

Data agnostic: The algorithm does not require any meta data about the users or the products to provide solid recommendations.

Provides metadata for user and product clustering: The r values assigned to each product and user can be passed to any clustering algorithm. This clustering information can be used for targeted ad campaigns for users or to provide more useful product groupings during organic browsing.

Possible to provide relative confidence indicators: The output of the algorithm can produce estimated ratings that do not match existing ratings given to it. The gap between the two can be used to estimate the relative confidence of predictions.

1.5 Key Disadvantages

Cannot be naturally augmented with metadata: There is no way to input metadata into the algorithms in order to make the output more accurate.

Hidden variables require work to interpret: As mentioned earlier we can see how compatible metadata is with the rating model in so far as it informs preference, however that requires analysis and some guess work. It is, to be fair, much simpler and transparent than other machine learning models.

All or nothing: With sufficient data the algorithm produces a rating for every possible user product pair. If the existing data is too sparse though the algorithm will simply produce garbage. It isn't possible to take a tradeoff of fewer recommendations in exchange for lower data requirements. In that case more complicated models are better suited.

Non Linear response: The model is purely linear and does not account for non linear response very well. As an example someone might like scary movies but not TOO scary. The model accounts for these types of effects by requiring more hidden variables and therefore higher existing data requirements.