Report on Matrix Factorization

Germain Bregeon – Emma Covili – Xavier Geffrier

Team Germain Emma Xavier

Introduction:

With the explosion of the internet in recent years, large amounts of data have appeared along with the need to sort them. In our case, the goal is to recommend the perfect movie to a user.

We are trying to build a recommender system using the matrix factorization method.
The matrix factorization method consists in approximating the ratings matrix R with lower rank user and item matrices, respectively U and I such that $R \approx IU^T$.

This problem can be stated as:

$$\min_{I,U} \| R - IU^T \|_{\mathcal{F}}^2$$

Where $U \in \mathbb{R}^{n,k}$, $I \in \mathbb{R}^{m,k}$ and $\| X \|_{\mathcal{F}}^2 = tr(X^T X)$.
We add regularization terms to avoid overfitting and improve generalization and then we have:

$$\min_{I,U} \| R - IU^T \|_{\mathcal{F}}^2 + \lambda \| I \|_{\mathcal{F}}^2 + \mu \| U \|_{\mathcal{F}}^2$$

With $C(I,U) = \| R - IU^T \|_{\mathcal{F}}^2 + \lambda \| I \|_{\mathcal{F}}^2 + \mu \| U \|_{\mathcal{F}}^2$

We compute the partial derivatives, and we get:

$$\frac{\partial C}{\partial U}(I,U) = -2R^T I + 2UI^T I + 2\mu U$$
$$\frac{\partial C}{\partial I}(I,U) = -2RU + 2IU^T U + 2\lambda I$$

Part 1: Methods

Stochastic Gradient Descent and Alternated Least Square on small dataset

Firstly, we implemented the two methods we saw during the class: the **Stochastic Gradient Descen**t (SGD) and the **Alternated Least Square** (ALS), using the numpy library on Python.
We tested them both on the small dataset (100k ratings) with the policy discussed in class, users with more than one rating, one in the test set and the others in the train set. To compare our methods, we used the Rooted Mean Square Error (RMSE).

We tried to determine the best hyper-parameters for our methods. So, we tested different values of k, the number of components, and different numbers of iterations to see which settings would lead our methods to converge as fast as possible.

Alternated least square and power factorization with robustness to missing entries on the 25M dataset

After getting some initial results on the smaller dataset, we decided to tackle the 25M dataset, with 60 000 movies and 160 000 users. After trying to run our initial code on this new matrix we quickly understood that we would need to change something for it to run without using

too much memory. To implement the alternated least square method on the big dataset, we decided to use the sparse matrix representation from "scipy" that also implements a lot of methods that are compatible with numpy methods. After implementing this method, we conducted several experiments trying to determine if there was a better k, and how many iterations we needed for our method to converge.

After getting results for this method, we quickly understood that considering the missing entries and setting them to 0 was very bad as all the values that we were predicting where 0. To solve this problem, we decided to find an algorithm that didn't consider these missing entries. In the book "Generalized Principal Component Analysis" from René Vidal, Yi Ma, S. Shankar Sastry, there is a part on PCA with robustness to missing entries, and we are trying to solve the same problem as them:

$$\min_{I,U} \| W \cdot (R - IU^T) \|_{\mathcal{F}}^2 + \lambda \| I \|_{\mathcal{F}}^2 + \mu \| U \|_{\mathcal{F}}^2 \text{ with } (W \cdot X)_{i,j} = w_{i,j} * x_{i,j} \text{ and}$$

$$W_{i,j} = \begin{cases} 1 \; if \; R_{i,j} \; is \; known \\ 0 \; if \; R_{i,j} \; is \; not \; known \end{cases}$$

We implemented the power factorization algorithm from this book that works by alternating minimization of I and U, at each iteration, we set for each line of I and U:

$$I_i = (\sum_{j=1}^{n} w_{i,j} u_j u_j^T + \lambda I_d)^{-1} \sum_{j=1}^{n} w_{i,j} R_{i,j} u_j \quad \text{for } i\text{=1,...m}$$
$$U_i = (\sum_{j=1}^{n} w_{i,j} i_j i_j^T + \mu I_d)^{-1} \sum_{j=1}^{n} w_{i,j} R_{i,j} i_j \quad \text{for } i\text{=1,...,n}$$

To implement this algorithm, we still used the "scipy" sparse matrix representation and tried to vectorize the operations as much as possible so that the algorithm didn't take too much time to compute. We conducted several experiments on the values of the different parameters to determine which parameters where the best for our model.

Deep Matrix Factorization and Constraint Programming.

We also implemented two different methods for solving the problem: one we found in literature, the **Deep Matrix Factorization** (DMF)[1] [Xue et al., 2017] and one we developed by ourselves, using constraint-programming (CP).

Our constraint programming method uses the solver provided by Google in its or-tools library. Our goal is to find entries of matrices I and U, which are the variables in our problem. Given that, for each entry $r_{i,j}$ of matrix R, we want $r_{i,j} \approx I_{i,*}^T \cdot U_{j,*}$ we define two inequality constraints, according to a tolerance value t:

$$r_{i,j} \geq \left(i_{i,1} u_{j,1} + \cdots + i_{i,k} u_{j,k}\right) \times (1 - t)$$
$$r_{i,j} \leq \left(i_{i,1} u_{j,1} + \cdots + i_{i,k} u_{j,k}\right) \times (1 + t)$$

We then use the solver to find existence of a solution and test error on it. For this implementation, we did not use the MovieLens dataset, but more smaller matrices we made ourselves.

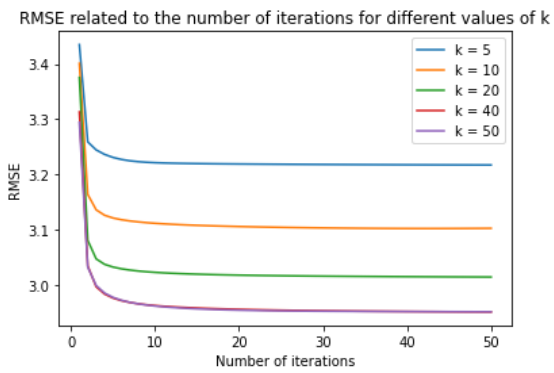
[1] https://www.ijcai.org/Proceedings/2017/0447.pdf

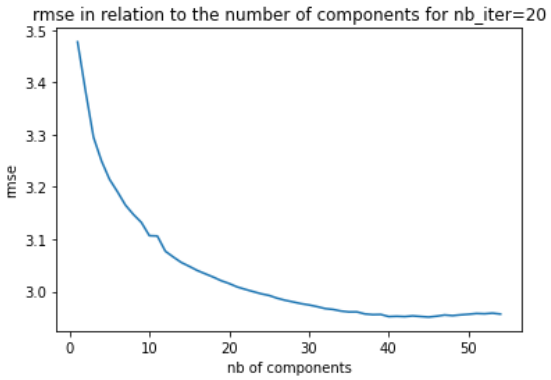
Our DMF implementation has been made using the Keras Functional API, and a small portion of code has been inspired from a previous implementation[2]. The idea is to train a deep neural network to predict missing values in matrix R, by training it to represent users (matrix rows) and items (matrix columns) as vectors in a latent space, and then computing cosine similarity between latent representations of an item i and a user u to get $\hat{R}_{u,i}$.
We conducted several experiments to evaluate the global quality of this approach, and to try the sensibility to hyperparameters or other architectures. We tried to change: the negative sampling ratio, the learning rate, the number of hidden layers, the number of factors in the final latent space. Everything was trained using the 1M MovieLens dataset and rated with RMSE. Only one training used the 25M dataset, much slower to read.

Part 2: Results

Alternated least square for sparce matrices on 25M dataset

The ALS method for sparse matrices gives relatively good results according to the fact that it doesn't take the zeros of the matrix into account. We tried to determine the best k, number of components, according to the number of iterations.
As we can see on the following graphs, the best k is 40 as it gives the lowest RMSE. On the other one, we can see that in 20 iterations, the RMSE drops to 2.8-2.9 for k equal to 40.





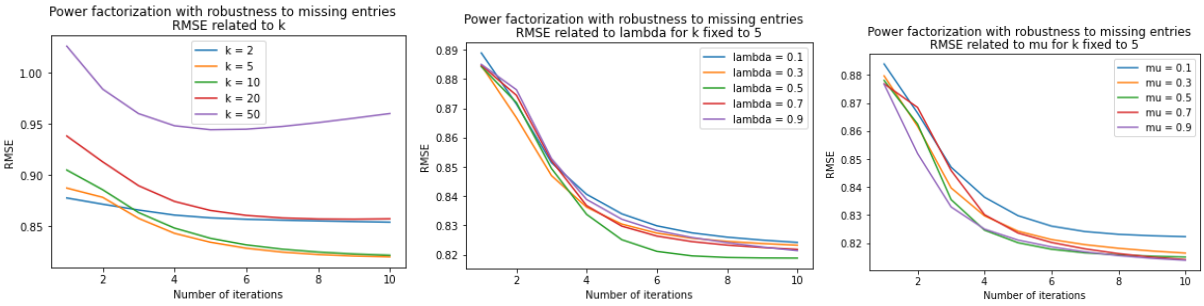
Although this method gives better results than the SGD or the usual ALS on the smaller dataset, we were not quite satisfied with it. Indeed, for ratings between 0 and 5 having a RMSE of 2.8 seemed a bit too high. This is the reason why we tried the next method.

Power factorization with robustness to missing entries on the 25M dataset

For this method our RMSE seems to be far more acceptable as it is of order 0.82. As it were our best method so far, we tried to determine the best hyper-parameters as possible, so we ran the method for different values of k, λ, and μ.

In the end, we obtain that the best parameters are $k = 5$, $\lambda = 0.5$ and $\mu = 0.9$.

[2] https://github.com/hegongshan/deep_matrix_factorization



Constraint programming

Constraint programming worked; in a sense it was able in most cases to find the factorization of a matrix if tolerance is high enough. However, it has two big issues making it a very bad solution to this problem.

- First, the way our constraints are defined requires every entry in R to be correctly approximated, which in real life would be very unlikely with a small k. It does not allow any prediction error, we can only set a higher tolerance, with impacts on all the learning. It could be improved in two ways. We could give a different tolerance for each entry in R, but it would imply a huge amount of "die and retry" to find the best possible precision. We could replace our set of constraints with just one constraint computing the error of the whole matrix, but this would take a very long time to learn, since all variables would have to be set before the constraint can be evaluated, killing the efficiency of constraint programming.
- Then, and more importantly, complexity and time execution are really high. This is partially because constraints are non-linear: they are expressed with multiplication between variables, and solvers are very less performant in this case. But the main issue is that CP is designed to explore discrete spaces, and we are here looking for floating values. We can "discretize" the problem by fixing a digit precision and look for integers, but it creates a too large space, which cardinality is given by $k \times (m + n) \times 10^d$, with d the digit precision. It grows exponentially as d grows.

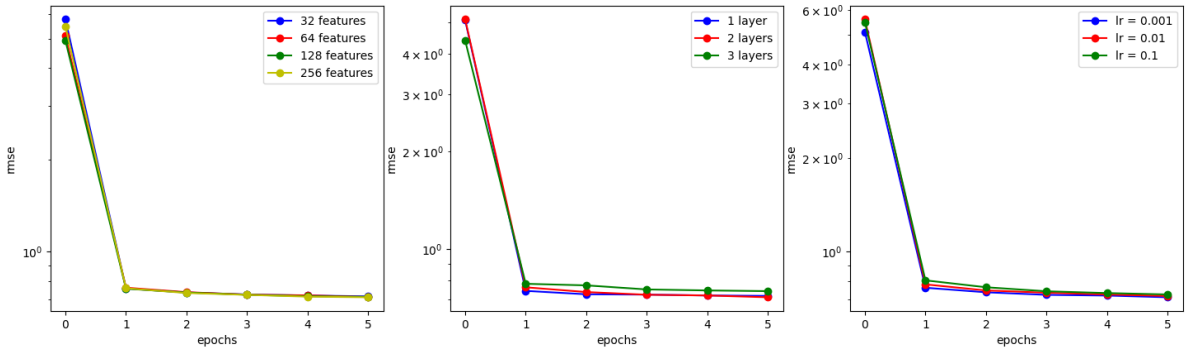
Deep Matrix Factorization results

DMF gives good results. With a simple architecture, we got very quickly a RMSE around 0.71 on our test dataset, between the predicted and actual ratings (0-5). Our base parameters were: 2 hidden layers, 64 features in last latent space for each vector (user and item), a 0.001 learning rate, and no negative sampling.

The model is almost non-sensitive to hyper parameters. Even this was expected reading the Xue paper, it looks like their variation is not distinguishable of noise in the resulting error. This may be explained by the fact that even a very simple network, with one small layer, is already enough to provide good predictions, so tweaking our quite complex network had very marginal effects. However, some hyper parameters variations caused in practice a very huge disparity in learning time, because they changed the number of total parameters to train, or the.

The only parameter with high influence on result was the negative sample ratio, defined as the number of 0-rating added in training dataset for each non-zero rating. Xue predicted a

good ratio at 5, where we found than it was better to not use negative sampling at all to not disturb the model.



Comparison of RMSE evolution when changing number of features in final latent space, number of layers in model, and learning rate

Conclusion:

In the end, our takeaway is that constraint programming was a fun method to try and implement but the results weren't there, and we demonstrated why it was a bad idea to do constraint programming on a problem like this one. The best model that we got in the end was the model using power factorization with robustness to missing entries, we tried to develop this model by removing user/item biases on each user/item, but we didn't have time to debug this method and Jupyter Notebook didn't like it as our Kernel was crashing a lot while computing. Deep Matrix factorization also seems promising, but due to a lack of time, we also didn't have time to explore it fully on bigger datasets.