**Abstract**

This short report will focus on describing the algorithms and solutions proposed by us in order to tackle the Politecnico di Milano Recommender System challenge for A.Y. 2016/17 held on the Kaggle platform. The challenge consists of recommending five items to each of ten thousand users from a dataset of 165 thousand items and almost 520 thousand. In this report we are going to talk in a sequential order about the algorithm we selected to tackle the problem, explaining why we used these approaches, how we implemented it in Python and which difficulties we found during the process. Our general strategy was to first find a theoretic possible solution by studying the literature, and then improve that using our own intuition and experience.

# 1 Approaches

## 1.1 Pre-processing

Before implementing the recommender we decided to create a local implementation of MAP evaluation. The first step to do it was to split the provided data set into independent train and test data sets. This is done using a simple script *split.py*, randomly assigning samples to either train or test data set, with specified probability. Obviously, our local evaluation does not give the exact same output as Kaggle evaluation, but we noticed that the magnitude of an improvement in our local score corresponded to a similar amount of improvement in the Kaggle evaluation, thus we consider our local evaluation a good approximation of the real evaluation done by Kaggle. So the set of parameters maximizing locally calculated score of a given model, should be close to parameters maximizing Kaggle evaluation of the same model. Local evaluation is performed by *evaluate.py* script, which can be used for assessment of any model, making it possible to compare performances of different models.

## 1.2 User-based Collaborative Filtering

The first implemented approach was User - Based Collaborative Filtering (CF) recommender. This approach was more simple to implement than Item - Based CF, because of the size of User Similarity Matrix. For 40,000 users in our data the size of this matrix was $40,000^2$ cells which is far less than $150000^2$ for Item Similarity Matrix, computed for Item - Based approach. The User - Based implementation can be seen in *user_simple.py* file. The current version of the file, unlike the first one, is already well optimized with proper scipy data structures used.

## 1.3 Item-based Collaborative Filtering

The next approach implemented was naturally Item - Based Collaborative Filtering. It can be seen in *item_simple.py* file, and it's easy to notice that it's

almost identical to User - Based approach. The difference is calculating Item Similarity Matrix instead of User Similarity Matrix, and applying shrinkage in slightly different way. Despite the fact we spent a lot of time tuning the solution (by adjusting K and shrink term), we didn't notice big improvements compared to User - Based approach. The reason for that might be the sparsity of User Rating Matrix. For 40,000 users and 150,000 items (which gives 6 billion cells) we have only 500,000 ratings, which gives only one rating per 12,000 cells. In fact many ratings are duplicated (same user - item pair), so the User Rating Matrix is even more sparse than that. We were able to deal with this amount of data in an efficient way using Compressed Sparse Row matrices (CSR) from Scipy *scipy.sparse.csr_matrix*[1], which are well suited for doing efficient arithmetic operations, efficient row slicing and fast matrix vector products.

### 1.4 Matrix factorization approach

After these Collaborative filtering algorithms we decided to try matrix decomposition, in particular FunkSVD algorithm, which is one of the simplest implementations of Singular Value Decomposition. This algorithm is based on Stochastic Gradient Descent, which is an iterative method to optimize an objective function. In our case SGD has been used for approximating the rating matrix R with the dot product between two so-called user factors (U) and item factors (V) matrices. Due to its iterative nature, many computations on the matrices U and V were required and the best way to perform this was using cython. We used cython implementation of SGD provided by Massimo Quadrana[2]. The biggest challenge connected with this approach was tuning of algorithm's parameters, which included:

- number of latent factors
- initial learning rate used in SGD
- regularization term
- number of iterations in training the model with SGD
- mean value used to initialize the latent factors
- standard deviation value used to initialize the latent factors
- learning rate decay

Tuning of these parameters was difficult because of the computational intensity of FunkSVD solution. On our machines it took more than 30 minutes to evaluate single parameter selection. Interestingly, the main reason for such a long running time of our algorithm was not SGD algorithm itself, but the fact that the outcome U, V matrices were not sparse. This means that estimated ratings had to be computed for each user independently, in order not to exceed memory limits. After several days of tuning we were able to almost match the score of Item - Based CF approach on Kaggle, but we didn't manage to improve.

### 1.5 Hybrid solutions

Finally, we ended up having 3 different algorithms (User - Based CF, Item - Based CF, and FunkSVD) scoring very similar score on Kaggle (less than 10% of difference). It seemed natural to try combining them into one hybrid solution. We noticed that each of 3 algorithms computes Estimated User Rating Matrix first, and makes recommendations based on this matrix. The idea was to calculate weighted sum of these matrices, and make recommendations based on this final user rating matrix. The solution can be seen in *hybrid.py* file (it makes use of outcome matrices from 3 partial algorithms previously saved to files). The hybrid solution, after adjusting weights assigned to different 3 matrices, gave us 5,58% of improvement compared to our second best algorithm (namely, Item - Based CF).

### 1.6 How to reproduce the algorithm

In order to fully reproduce the algorithm, Python 3.x is needed and a C compiler for cython. First of all execute *setup.py* to properly compile the cython code located in the folder *_cython*. After this, execute the function *split.py* to divide the original dataset into test set and train set. Then, execute the following three algorithms: *item_simple.py*, *user_simple.py*, *mf.py*. Then execute *hybrid.py*, which outputs the file *result.csv*.

## References


[1] Scipy reference for CSR matrices `https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.sparse.csr_matrix.html`

[2] Massimo Quadrana's GitHub repository for the course `https://github.com/mquad/recsys-course-2016-pub`