Machine Learning - Project 2: Recommender Systems

Olivier Couque (246212), Emma Lejal Glaude (250692), Matthieu Sauve (246243)
Data Science, EPFL, Switzerland

***Abstract*—Recommender systems have become an important feature on most e-commerce platforms such as Amazon, Ebay and other web-based applications such as Facebook or IMDB. They are used in predicting users potential interests based on their preferences and past behavior on the platform. This paper explores different algorithms used in the recommender field, analyzes and compares the accuracy of their predictions.**

I. Introduction

With more than a decade of research on recommender systems, a vast variety of algorithms exist in literature and are being used in different domains. From the engineering point of view, each type of system has its own strengths and weaknesses. The two engineering dimensions of recommender systems (like most other intelligent systems) are speed and accuracy. The winning ones are usually those that follow the principle of system and algorithm co-design.

In this project, we designed some recommender systems in Python, based on different techniques: especially user-based collaborative filtering and matrix factorization, with the main goal of predicting accurately the ratings each user would give to each item.

The report is organized as follows: in Section II, we describe the characteristics of the dataset we used. In Section III we will present the recommender system based on the collaborative filtering technique whereas Section IV will present the versions based on matrix factorization. Section V will present the results obtained for the different methods we implemented and Section VI will discuss on what could have been done or wasn't done through this project that could have given better results. Finally Section VII will conclude.

II. Dataset Characteristics

We used the provided dataset of 1 million ratings available at the link [1]. The dataset contains $1,176,952$ ratings of $1,000$ items, considered as movies through the whole project, made by $10,000$ users. The movies are rated with integer values from '1' to '5'. The data is contained in a single CSV file:

data_train.csv: Each line contains a user id, a movie id, and the rating; in the following format:

movieID_userID,rating (e.g. r15_c23,4 meaning user 15 gave movie 23 a rating of 4)

We are also given the **sample_submission.csv** file that gives us the specific predictions we are supposed to deliver for the Kaggle competition. In order to work easily on the data, we analyzed the CSV files and converted the data in a sparse rating matrix $\mathbf{R}$ of dimensions $Movies$ $\mathbf{I} \times Users$ $\mathbf{U}$.

Number of users	10000
Number of movies	1000
Total number of ratings	$1,176,952$
Average of all ratings	3.857
Sparsity	88.23%
Median of ratings per user	104
Median of ratings per movie	880

TABLE I
Statistics of the dataset

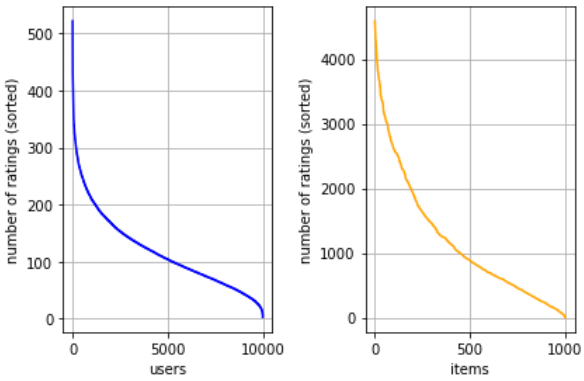

Fig. 1. Number of ratings per user (left) and movie (right). From the left graph, most of the users have an important number of ratings, thus just a few users have rated few movies. We observe the same tendencies for movies, on the left graph

III. User - User Collaborative Filtering

Collaborative filtering (CF) is a popular recommendation algorithm that bases the predictions and recommendations on the past ratings of users in the system. Thus, User-user CF consists of finding other users whose past rating behavior is similar to the one of the current user and use their ratings on other items to predict what the current user will like.

To compute the predictions for each user in the algorithm, we need a similarity function $s(u, v)$ computing the similarity between two users: u and v. The similarity function is used to compute a neighborhood $\mathbf{K} \subseteq U$ for each user u that consists of the K users that are most similar to u. Finally, the predicted rating for user u and movie i will be computed using the set of neighbors $\mathbf{K}$, as follows:

$$p_{u,i} = \frac{\sum_{v \in \mathbf{K}} s(u,v).r_{v,i}}{\sum_{v \in \mathbf{K}} |s(u,v)|}$$

where $r_{v,i}$ is the rating user v gave to movie i. There are several choices for the similarity metrics and we decided to use the Cosine similarity and Pearson correlation (link [2] for definitions) as they are widely used similarity measures in the recommender field.

We fix the neighborhood size of each user u to a constant $K \in \mathbb{N}$ in order to find for which value of K the predictions are the more accurate. The neighborhood for each user *u* can be chosen using different approaches. The one used in the project works as follows:

We discard all the users that haven't seen the movie *i* that we want to predict and then choose the K users *v*, distinct from *u*, with best similarities $s(u,v) \Longrightarrow$ many neighborhood per user (one for each movie he hasn't rated).

This approach allows to give predictions for movies with a few ratings, meaning they are not popular but might be good. If we want a user's prediction for this kind of movie but the users he is most similar with haven't rated this movie, we would get a 0 as prediction, which we don't want. Thus we first match the user to the few users that have seen the particular movie and then compute his best neighbors from this set of users.

A fundamental limitation of the CF algorithm is its inability to scale to large number of users because of the expensive pairwise similarity computations involved. Moreover, the algorithm suffers from what is called the cold start problem, due to its inability to address the system's new products and users.

IV. Matrix Factorization Model

A. Basic Matrix Factorization

Matrix factorization models map both users and items to a joint latent factor space of dimensionality K, such that user-item interactions are modeled as inner products in that space. Thus, an advantage of Matrix Factorization compared to Collaborative Filtering is the flexibility to deal with various data aspects. Our goal is to approximate as close as we can our ratings matrix $\mathbf{R}$ ($I \times U$) with the matrices $\mathbf{W}$ of dimension $I \times K$ and $\mathbf{Z}$ of dimension $U \times K$, the matrices of movie features and user features respectively, such that:

$$\mathbf{X} \approx \mathbf{W}\mathbf{Z}^T$$

The resulting dot product, $\mathbf{Z}_u^T \mathbf{W}_i$ captures user u's overall interest in the item i's characteristics: this approximates user u's rating of item i $p_{u,i}$.

In order to learn the feature vectors, $\mathbf{w}_i$ and $\mathbf{z}_u$, the system minimizes the regularized squared error on the set of known ratings:

$$min_{W,Z} \sum_{(u,i) \in \Omega} (r_{u,i} - (\mathbf{W}\mathbf{Z}^T)_{u,i})^2 + \lambda_u ||Z||_F^2 + \lambda_i ||W||_F^2 \tag{1}$$

where Ω is the set of pairs (u, i) with known ratings and where λ_u and λ_i are regularizer terms (> 0) to avoid overfitting the data.

In order to minimize the loss given by Equation 1, we used the Stochastic Gradient Descent (SGD) in the following way:

Features Initialization: we initialize the user features $\mathbf{Z}$ and item features $\mathbf{W}$ matrices randomly with each element in the interval $[0, \frac{1}{K}]$, K being the number of features. This way, we obtain first predictions which are not too high, when computing $\mathbf{W}\mathbf{Z}^T$, that will be modified in the next steps of SGD.

Then, for each epoch of SGD we iterate through all the known ratings $r_{u,i}$, compute the error between the rating and the corresponding prediction: $error_{u,i} = r_{u,i} - \mathbf{z}_u^T \mathbf{w}_i$ and update the corresponding feature vectors using the gradient of the loss:

$$\begin{aligned} \mathbf{z_u} &= \mathbf{z_u} + \gamma(error_{u,i}\mathbf{w_i} - \lambda_u \mathbf{z_u}) \\ \mathbf{w_i} &= \mathbf{w_i} + \gamma(error_{u,i}\mathbf{z_u} - \lambda_i \mathbf{w_i}) \end{aligned} \tag{2}$$

where $\gamma > 0$ is the step size (learning rate). Typically, the number of epochs we had for SGD was between 20 and 50: it was sufficiently precise and avoided the entire SGD algorithm to run for too long (especially for the grid search).

To estimate how well our system is learning, we divide our ratings matrix $\mathbf{R}$ in a training (typically 90% of ratings) and testing ratings (10%) matrices, randomly splitted. Our system learns on the training ratings and we verify its accuracy by comparing the test ratings with the corresponding trained predictions, using root mean squared error (RMSE). Even if it would have been a great idea, we did not use cross-validation as it would have taken way too long to compute all the SGD with each different training matrices.

In order to find the best parameters: step size γ, regularizers λ_{user}, λ_{item} and the number of features K we proceed as follows:

First we find the step size γ by setting the other parameters and running SGD with different values of γ: with $\lambda_{user} = \lambda_{item} = 0.01$, $K = 50$, we obtained the minimal test loss for $\gamma = 0.025$ ($RMSE = 0.9925$). This γ value may seem big at a first glance (in comparison, $\gamma = 0.005$ in the paper [3]) but we are decreasing its value at each epoch of SGD as follows: $\gamma^{t+1} = 0.8\gamma^t$ in order to have a fast convergence to the minimal loss and avoiding the divergence at the same time. In the end, it is more less equivalent to using a small and constant rate such as $\gamma = 0.005$ but this will converge slowly and will require more epochs and more computation time.

Once we find the best step size, we implement a grid search to find the combination of λ_{user}, λ_{item} and K that gives us the minimal test loss when running SGD with these parameters. A particularity of our grid search consists in the use of warm start.

Warm Start: At first iteration, we initialize randomly the matrices of features $\mathbf{W}$ and $\mathbf{Z}$, as explained before, but for the next iterations we run SGD with the features matrices that gave us the previous minimal test loss instead of always starting again with random features matrices, because we know that the ones found previously are closer to the ideal features. Obviously, warm start is reinitialized at each new iteration on the number of features K as the features matrices change of dimensions.

B. Adding Bias

A lot of observed variations in rating values is due to effects associated to either users or items, known as biases: some users tend to give higher ratings than others and some items tend to receive higher ratings than others.

Thus, we want our system to consider the bias associated to each user and item. To do so, we associate a bias, denoted as $\beta_{u,i}$, involved in each rating as:

$$\beta_{u,i} = \beta_u + \beta_i + \mu$$

where μ is the average of all the ratings, β_u and β_i are the observed deviation of user u and item i respectively, from the average. For example, suppose that you want a first-order estimate for user Bob's rating of the movie Pulp Fiction. Let's say that the average rating over all movies, μ, is 3.8 stars. Furthermore, Pulp Fiction is better than an average movie, so it tends to be rated 0.6 stars above the average. On the other hand, Bob is a critical user, who tends to rate 0.7 stars lower than the average. Thus, the bias for Pulp Fiction's rating by Bob would be $\beta_{Bob,PulpFiction} = (3.8 + 0.6 - 0.7) = 3.7$.

Biases extend the prediction as follows: $p_{u,i} = \beta_{u,i} + \mathbf{z}_u^T \mathbf{w}_i$ which is composed of global average, item bias, user bias, and user-item interaction. The system learns the user features $\mathbf{Z}$ and item features $\mathbf{W}$ by minimizing the extended loss function from Equation 1 as follows:

$$\begin{aligned} min_{W,Z} \sum_{(u,i)\in\Omega} (r_{u,i} - \beta_{u,i} - \mathbf{W_u}\mathbf{Z}_i^T) \\ +\lambda_i(||W||_F^2 + \beta_i^2) + \lambda_u(||Z||_F^2 + \beta_u^2) \end{aligned} \tag{3}$$

Thus, the update of the feature vectors from Equation 2 at each epoch of the SGD slightly changes by taking in consideration the bias error : this time, the error given by the update is computed as $error_{u,i} = r_{u,i} - \beta_{u,i} - \mathbf{z}_u^T \mathbf{w}_i$. This result is computed from the gradient given by the new loss function of Equation 3. In order to find the best parameters for K, λ_i and λ_u we run a grid search as we did for Basic Matrix Factorization (subsection IV-A).

V. Results

For each of the following subsections, we divided randomly the ratings in a training set (90% of ratings) and a testing set (10%). We trained our recommender system on the training set, and tested our system by computing the RMSE between the predictions and actual ratings of the testing set.

Because the real ratings are integer values between 1 and 5 and that some of the predictions can go below or above that interval, we make sure to set the predictions above 5 and below 1 to 5 and 1 respectively before submitting on Kaggle.

A. User - User Collaborative Filtering

We computed the RMSE on the testing set with different values for the number of best neighbors K to keep for each user when computing the predictions. We also used two different similarity metrics: cosine similarity and Pearson similarity between the users.

K	Cosine similarity	Pearson correlation
10	1.05883	1.06188
30	1.03010	1.03032
50	1.02562	1.02428
75	1.02320	1.02192
100	1.02272	1.02032
150	1.02237	1.01975
200	**1.02232**	**1.01971**
300	1.02289	1.02003
500	1.02406	1.02122
750	1.02514	1.02247
1000	1.02581	1.02345

TABLE II

RMSE ON THE TESTING SET DEPENDING ON NUMBER OF BEST NEIGHBORS K PER USER, AND SIMILARITIES BETWEEN USERS: COSINE OR PEARSON. WE OBSERVE THAT THE MINIMAL LOSS IS OBTAINED WHEN $K = 200$ AND USING PEARSON CORRELATION (LOSS WITH COSINE SIMILARITY IS SLIGHTLY SUPERIOR).

From the results of Table II, we find that the minimal RMSE is obtained when the number of best neighbors to consider for each user is $K = 200$ (2% of the total number of possible neighbors). It seems logical as we need a certain amount of best neighbors to find a particular user's (call it *u*) tastes but we don't want too much neighbors as the more we increase the amount of neighbors, the more they will have different tendencies compared to *u*'s. The most similar users are already chosen and by increasing K we would only add more of the remaining users which are less similar to *u*. Thus the predictions will be less accurate and the RMSE will tend to grow. By training on the full ratings matrix using these best parameters, we obtain a loss of $RMSE = 1.01899$ on Kaggle.

B. Basic Matrix Factorization

As mentioned in the corresponding Section IV-A, we compute a grid search to find our optimal parameters giving the minimal test loss. The results are showed in Figure 2.

From the grid search, we found the minimal test loss to be: $RMSE = 0.9817$ when $K = 20$, $\lambda_{user} = 0.1$ and $\lambda_{item} = 0.01$. We computed the SGD and

compared loss between all the possible combinations of $K \in [20, 45, 75, 100, 150]$, $\lambda_u \in [1.0, 0.1, 0.01, 0.001]$ and $\lambda_i \in [1.0, 0.1, 0.01, 0.001]$.

We used this best parameters to train on the full ratings (no testing) such that our system is more accurate and submitted the wanted predictions on Kaggle that allowed us to reach a loss of $RMSE = 0.98399$.

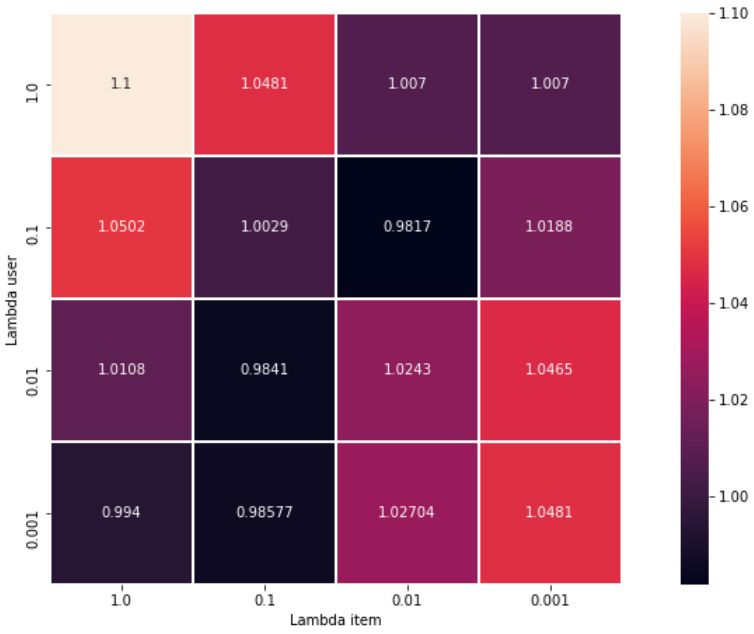

Fig. 2. Loss results of the grid search on λ_i for the Basic Matrix Factorization technique with $K = 20$ number of features. We observe that the loss is minimal (darkest square) when $\lambda_{user} = 0.1$ and $\lambda_{item} = 0.01$

C. Biased Matrix Factorization

With the grid search on the same combinations of parameters as Section V-B, we found that the best parameters giving the minimal test loss of $RMSE = 0.9453$ were $K = 20$, $\lambda_{user} = 0.1$ and $\lambda_{item} = 0.01$

For the final submission on Kaggle, we did not train on the whole ratings because the system using the best parameters found in the grid search was overfitting. We had no way to control it as we had no test set to check the test loss augmentation at each epoch of SGD (meaning overfitting).

However, by just training on 90% of the ratings to make sure there were no overfitting (check test loss at each epoch of SGD) and by computing the predictions with the features matrix giving the minimal test loss through the grid search (using warm start explained in Section IV-A), we obtained a loss of $RMSE = 0.98308$ (best submission) with the submission of these predictions, which is better than the loss obtained with Basic Matrix Factorization.

VI. Discussion

As showed in Section V, the biased matrix factorization is the technique that gave us the minimal testing loss. This is no surprise compared to the collaborative filtering method which is known to be less accurate compared to matrix factorization (Netflix Prize winners used this last technique). Moreover, as showed in the paper [5], taking the users and items biases in consideration, like we did in the biased matrix factorization method, yields better predictions.

We mainly implemented the recommender system on our own. We also used the Surprise library (link [4]) which is a Python scikit useful library for building and analyzing recommender systems. This library was faster for running SGD algorithm and allowed us to use k-fold cross-validation (with k = 5) which could be possible using our own developed recommender system but would imply a too long running time. However the predictions were less accurate using the library: typically, the minimal RMSE with the biased matrix factorization technique using Surprise was above 1.00 on Kaggle. Thus we decided to neglect the library in the end.

Alternating Least Squares (ALS) is also a popular algorithm used to compute the features matrices. However we decided to stick with the SGD algorithm because of the relative fast running time compared to ALS.

The recommender system could have been more efficient with additional data, generally provided, such as implicit feedback of the users, timestamps on the user's ratings, or movie genres. Some techniques like temporal dynamics, mentioned in the paper [5] showed a better accuracy of predictions in addition to the biases.

We realized towards the end of the project that using neural networks could have improved the accuracy of the system. However, the documentations we used for the project mostly described collaborative filtering and matrix factorization techniques and we sticked to these methods.

VII. Conclusion

Hence, we have seen different techniques to implement a recommender system: collaborative filtering and matrix factorization. However, the system designed through this project can still be improved, notably by using different techniques (neural networks) or by having a more precise grid search over the parameters by using cross-validation. Moreover, even if our own implemented recommender system is more efficient than those provided by some libraries (Surprise library for example), the overall computation time is also more important and this needs to be considered depending on the wanted accuracy.

References

[1] Kaggle competition, datasets [Online]: https://www.kaggle.com/c/epfml17-rec-sys

[2] Cosine Similarity and Pearson Correlation definitions and formulas [Online]:
https://en.wikipedia.org/wiki/Cosine_similarity#Definition
https://en.wikipedia.org/wiki/Pearson_correlation_coefficient#For_a_sample

[3] MNIH, Andriy et SALAKHUTDINOV, Ruslan R. Probabilistic matrix factorization. In : Advances in neural information processing systems. 2008. p. 1257-1264.

[4] Surprise library for recommender systems [Online]:
http://surpriselib.com/

[5] KOREN, Yehuda, BELL, Robert, et VOLINSKY, Chris. Matrix factorization techniques for recommender systems. Computer, 2009, vol. 42, no 8.