PCML CS-433: Recommender System

Team: **Netflix and MaCHILL Learning**
Gael Lederrey, SCIPER 204874, gael.lederrey@epfl.ch
Stefano Savarè, SCIPER 260960, stefano.savare@epfl.ch
Joachim Muth, SCIPER 214757, joachim.muth@epfl.ch

School of Computer and Communication Sciences, EPF Lausanne, Switzerland

***Abstract*—Active Collaborative Filtering Recommender Systems for movie collection using blending of 12 different methods (8 direct scoring methods and 4 iterative ones) in order to achieve around 0.977 RMSE on Kaggle's EPFL ML Recommender System challenge.**

I. Introduction

Collaborative filtering is a set of techniques aiming at the creation of recommender systems. Usually, we define three types of collaborative filter: **active**, **passive** and **content-based** (the best one being obviously a mix of the three). In industry, recommenders are mainly used to suggest new item to users based on their taste: movies, music tracks, items to purchase, ...

The objective of the project is to develop a recommender system using **active collaborative filtering** (i.e. pseudonymised items [1] rated by pseudonymised users).

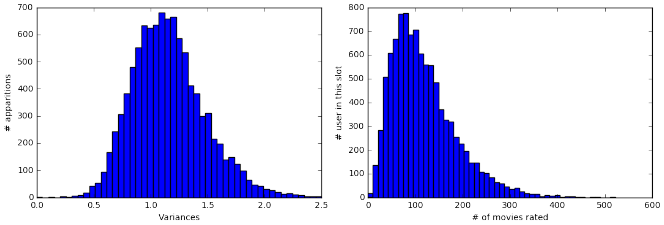
We will first go through a general **data analysis** in order to the quality of the data (spammers and participation of the users). Then we will test different models, starting from a basic mean given a prediction baseline and improving the score with more advanced algorithms.

According to *BellKor's Pragmatic Chaos* scientific paper [1] the best recommender is obtained through a clever blend of several models. Exploiting this approach, **we ultimately blended 12 different methods (8 direct scoring methods and 4 iterative ones)** eventually achieving around 0.977 RMSE. TO BE UPDATED WITH THE LATEST NUMBERS

The implemented model will be part of the **Kaggle's EPFL ML Recommender System** challenge in which predictions are rated by their RMSE compared with ground truth values. All external libraries are allowed as long as they are properly documented.

II. Data Description

The data represent ratings from $10'000$ users on $1'000$ movies in an integer scale from 1 to 5. Both of them are pseudonymized by an ID. This scale represents the number of *stars* given by the users, 1 being the lowest grade and 5 the best.



(a) Distribution of variances of ratings per user. No spammers.

(b) Number of movies rated per user. Good user participation.

Figure 1: Statistical description of data

The training set used to train our algorithm contains $1'176'952$ ratings which represent around 12% of filled ratings. Another $1'176'952$ ratings are hidden from us and must be predicted by our recommender algorithm in order to be scored on Kaggle platform.

III. Data Exploration

A. Search for spammers

One of the first step before starting learning from data is to ensure that they are real ones, and not produced by bots (spammers). As we know, spammers can act in different ways: **uniform spammers** constantly rate movies in the same way, while **random spammers** randomly rate movies. In order to check their existence, we analyzed the variances of user ratings: uniform spammers would be put in evidence by null variance, while random spammers present abnormally high variance. Figure (1a) shows the Gaussian distribution of the rating variances and ensure the data are free of spammers.

B. Participation of Users

Even free of spammers, data can still contain **inactive users**, i.e. users who subscribed to a platform but never use it or never rate movies. If they are in too big number compared with active users, they can disturb learning algorithms.

[1] Items anonymized by an ID, i.e. we do not have access to neither the name nor any content of it.

Figure (1b) shows histograms of number of movies rated by users and confirm us the good participation of the users.

C. User "Moods" (Deviation)

Because of mood/education//habits users having the same appreciation of a movie can rate it differently. Indeed, we show in the figure (2) that some users systematically rate lower/higher than others. It's important to take this effect into account in both evaluation of a movie and recommendation for the user and proceed to a normalization of the ratings.

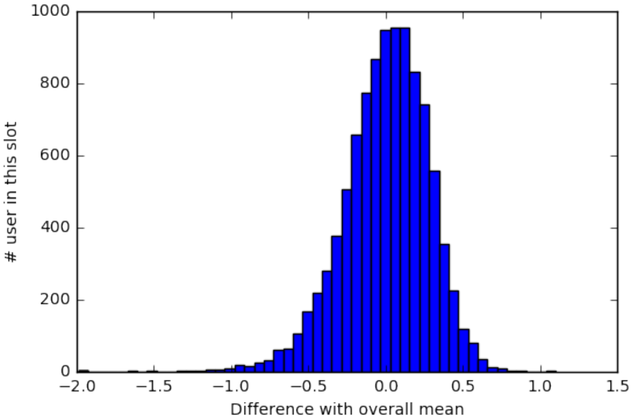


Figure 2: Difference of mean ratings per user compared with overall mean.

IV. Models

A. Global mean/median (2 models)

The simplest model is to take all the ratings in the train set and apply the mean or the median value. We return this value as the prediction. This give a baseline score from which we can compare further models.

B. User/Movie mean/median (4 models)

Another simple model is to compute the mean or median value for the users or the movies.

C. Movie mean/median with normalized user moods (2 models)

The third set of model uses the mean or median value for each movie. We also compute the "mood" of the users this way:

$$d_u = \overline{U} - \overline{u} \quad \forall u \in U \tag{1}$$

where $\overline{U} = \frac{1}{\#U} \sum_{u \in U} \overline{u}$ and $\overline{u}$ being the average rating of the user u.

Then, we return the prediction of a user u on a movie m:

$$p_{m,u} = \overline{m} + d_u \tag{2}$$

where $\overline{m}$ is either the mean or the median of the ratings on the movie m.

D. Matrix Factorization using Stochastic Gradient Descent (MF-SGD)

Matrix factorization techniques proved to be one of the most effective strategies to implement recommender systems. Given D items, N users and the corresponding rating matrix $X \in \mathbb{R}^{D \times N}$, we aim to find two matrices $W \in \mathbb{R}^{D \times K}$ and $Z \in \mathbb{R}^{N \times K}$ such that the quantity

$$E = \frac{1}{2} \sum_{\substack{d=1,\ldots,D \\ n=1,\ldots,N}} \left(x_{dn} - \left(WZ^T \right)_{dn} \right)^2 + \frac{\lambda}{2} \|W\|^2 + \frac{\lambda}{2} \|Z\|^2 \tag{3}$$

is minimized. K is a parameter, corresponding to the number of the *latent factors*; λ is a scalar that weight the regularization terms.

Different techniques have been deployed to solve this minimization problem. In this Subsection we will present the Stochastic Gradient Descent method, while in the next one we explain the Alternating Least Square optimization. The Stochastic Gradient Descent method is a faster variant of the standard gradient descent optimization. The gradient of the functional 3 is computed only on a single element of the summation, randomly chosen. The update process then follows the same rules of the batch gradient descent. An almost certain convergence to a local minimum is guaranteed under not restrictive hypothesis.

Using the `scipy.sparse` matrix library, we implemented the SGD method from scratch. Through a cross validation process we chose the best values for the parameters K, λ and the number of iterations. The results that we obtained will be presented in Section VI-B.

E. Matrix Factorization using Alternating Least Square (ALS)

ALS is one of the main alternatives to the SGD to solve the problem 3. It is an iterative algorithm consisting in alternately fixing one of the two matrices W or Z, while optimizing the problem 3 with respect to the other matrix. The optimization problem at each iteration is much easier to solve compared to the one solved by the SGD. A simple least squares technique can be exploited.

One of the most advanced open source frameworks available to solve this problem is Apache Spark. It is a cluster computing framework that provides to the programmers an application programming interface to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets. The Spark package `mllib` contains several implementations of different machine learning algorithms, including the ALS for collaborative filtering. Therefore we decided to exploit this library for the implementation. Spark also allows to specify the parameters K, the number of iterations and λ. We perform a cross validation to choose the best parameters, whose results will be discussed in Section VI-B.

F. *PyFM*

PyFM is a python implementation of Factorization Machines. This library is a wrapper of the C++ library libFM [2], one of the most advanced matrix factorization libraries. It can be found on Github [3]. The idea behind the algorithm is similar to the MF-SGD IV-D.

The usage of this library is really simple. It uses a few parameters:

- Number of factors corresponding to the scalar K in the MF-SGD.
- Number of iterations
- Initial learning rate

The number of iterations was fixed. The two other parameters were chosen with the use of a simple cross validation process.

G. *Matrix Factorization using Ridge Regression (MF-RR)*

Small description

V. Blending

The *Bellkor's Pragmatic Chaos* team, winner of 2009 *Netflix Prize*, explain in its paper that its solution was obtained by blending a hundred different models. [1] Without having the same number of models, we proceed the same to obtain our final solution. We perform a weighted sum that we optimize using **Sequential Least Squares Programming** (SLSQP) method provided by `scipy.optimize.minimize` library. Initial weights are set to $^{1}/_{n}$ for each model (n being the number of models).

A. *SLSQP method*

Sequential Least Squares Programming method is a **Quasi-Newton method**. Unlike Newton method, it does not compute the Hessian matrix but estimates it by successive gradient vector analyze [4] using **Broyden-Fletcher-Goldfarb-Shanno** algorithm (BFGS). This method allows optimization for function without knowing Hessian matrix, in a short computation time.

VI. Result

In order to create our recommender algorithm, 2 steps are required.

1) Find the best parameters for each model. A cross validation process was performed on the models MF-SGD, ALS, Collab. Filtering and PyFM.
2) Apply the blending between all the models with another cross validation process.

In this section, we first present the blending of the models as well as their parameters found after cross validation. Then, we present the benchmarks of the models and the blending.

A. *Blending*

Table I provides the weights after minimizing on the average RMSE. It also provides the parameters used for each model.

Model	weight	parameters
Global mean	2.87634	-
Global median	0.88256	-
User mean	-3.81181	-
User median	0.00362	-
Movie mean	-1.57271	-
Movie mean (mood norm.)	1.65276	-
Movie median	-2.27553	-
Movie median (mood norm.)	2.27933	-

MF-SGD (mood normalization)	-0.16857	$\lambda = 0.004$
		features = 20
		iterations = 20

ALS	0.75256	$\lambda = 0.081$
		rank = 8
		iterations = 24

Collab. Filtering	0.04356	$\alpha = 19$ features = 20

PyFM	0.30050	factors = 20
		iterations = 100
		learning rate = 0.001

Table I: Blending of models.

B. *Benchmark*

Table II presents the average RMSE of each model applied on the validation sets for the blending cross validation process. Last line is the result of the blending on the same validation sets.

Model	RMSE
Global mean	1.11906
Global median	1.12812
User mean	1.09531
User median	1.15200
Movie mean	1.03050
Movie mean (mood norm.)	0.99659
Movie median	1.09957
Movie median (mood norm.)	1.05784
MF-SGD (mood normalization)	0.99994
ALS	0.98875
Collab. Filtering	1.02776
PyFM	0.99178
blending	0.96191

Table II: Benchmark of models.

The blending gives a RMSE of 0.97788 on Kaggle's platform.

VII. Discussion of the results

Excluding the trivial models based on the user/movie mean/median, we mainly focused on **Matrix Factorization algorithms**, exploiting different techniques to achieve the best factorization possible. Content-based filtering algorithms are not suited for this problem since the users and the movies are fully anonymized and without additional information.

The blending plays an important role in our project. For the large number of studied models, finding a good blend between them proved to be a complex task. The solution that we present pushes as much as possible the RMSE, while at the same time introducing a noticeable overfitting of the data. In this Section we will discuss about these aspects.

1) Choice of the models:

2) Why keep models with and without normalization?: It should be legitimate to ask why we are keeping both normalized and unnormalized model for median/mean. Looking at the coefficients gives a partial answer. As we see in table I, normalized and unnormalized models oppose themselves almost exactly, with a little advantage for normalized models. The effect is that the method is more taken into account for users in the tail of the mood curve than for the central ones. Then, it allows the optimizer to have finer control on the blending.

3) Blending choices: The role of the blending part is extremely important in obtaining the best score possible. The large number of models available would be pointless without a good blend. Moreover, we mainly consider simple models, leaving to the blender the task of building a more complex one.

As mentioned in Section V we used the Sequential Least Square Programming method to optimize the weighted sum of models. Simpler methods, like grid search, are computationally too expensive and less accurate than the algorithm used. We spent some time in finding the best optimization method for our purposes.

We decided to also allow negative weight. This choice was made for two reasons:

- We obtained better results.
- Negative weights may help in better fitting the distribution of the training dataset. This can lead to overfitting, as we discuss in the next subsection, but, at the same time, it also produces better results.

We also did not put an contraint on the sum of the weight (i.e. the sum is not fixed to 1) for the same reasons.

4) Overfitting: We apply several techniques to reduce as much as possible the overfitting of the models we used. In particular we used a 4-folds cross-validation both to determine the best parameters for each model, both to choose the best weights in the blending. Despite this, the model slightly overfits the data. For the Kaggle submission We obtained a score of ADD SCORE in the training dataset and ADD SCORE in the validation dataset. In the cross validation we obtained a score of ADD SCORE. There are many possible reasons underlying this behaviour. Probably the blending process that we used, although the proven accuracy, introcudes too much complexity in the model, thus overfitting the training database. Using simpler models would solve this problem, but would worsen the prediction. We therefore decided to keep the more complex one.

References

[1] Y. Koren, “The BellKor Solution to the Netflix Grand Prize,” 2009.

[2] S. Rendle, “Factorization machines with libFM,” *ACM Trans. Intell. Syst. Technol.*, vol. 3, no. 3, pp. 57:1–57:22, 2012.

[3] CoreyLynch, “pyfm: Factorization machines in python,” https://github.com/coreylynch/pyFM, 2016.

[4] Wikipedia, “Quasi-Newton Method — Wikipedia, The Free Encyclopedia,” 2016, [Online; accessed 20-December-2016]. [Online]. Available: https://en.wikipedia.org/wiki/Quasi-Newton_method