EPFL Machine Learning - Project 2

Pierre-Alexandre Lee, Thomas Garcia, Yves Lamonato
EPFL Lausanne, Switzerland

1 Introduction

For this project, we will put into practice some of the concepts seen during the lectures. Among the proposed projects, we choose the one that involves creating a recommender system from user's ratings of certain products. We can find such systems in big online stores like Amazon for example. Ours will be a much simplified version of it of course.

Just like in the first project, we will use the dataset provided on the Kaggle competition. We have used some libraries, the instruction for their installation can be found in the README.

2 The dataset

2.1 Raw data

The dataset on Kaggle is a .csv file, were each line consist of the matrix coordinates along with the value at those coordinates. Each value is the rating of a certain user (row) for an item (column). The coordinates were given in a String format, like for example *r36_c124*, which correspond to the 36th row of the 124th column. For some of the data processing we used functions from the lab 10 correction, with changes when needed.

2.2 Data pre-processing

1. *Importing data* : With almost all our algorithms, we needed the data to be in a matrix form. We used the sparse format from the Scipy library for efficiency. The loading algorithm reads the .csv file line by line, split the row and column index and subtracts 1 to them (since the indexes of the file begin at 1). We then use those coordinates to fill the sparse matrix with the corresponding ratings.

2. *Exploring the data* : Our data consist of 1'176'952 ratings given by 10'000 users for 1'000 items (almost 11.77% of the possible ratings). Using some statistics we got from the data importation part, we can plot the number of ratings per user, and also per item (number of ratings given to that item), as you can see in figure 1. From those stats, we found that each user in our data has rated at least 3 items, and that each items has at least 8 reviews. This means that we won't have to drop useless users or items.

3 Models

For the SGD and ALS training algorithms, we used the correction of the lab 10 as a template.

3.1 Models used

Those models were tested (the results are displayed in a table in the "Results" section):

- **Stochastic gradient descent (SGD):**

  This method, while quite simple, was too slow, and we didn't get very good results. The used a parameter $gamma$=0.02 (found via grid search) and a number of epochs=20 (with a higher number of epochs the error only improved marginally).

- **Alternating least squares with bias estimation (ALS):**

  For this model, we tested both the standard version from lab 10 and one where we also took into consideration the bias of the ratings. Here's a quick explanation on what the bias we are talking about is: Each user's and item's mean rating might differs a bit from the global mean, whether it be because an item is actually objectively good or because a user is prompt to give only good ratings (hence the term "bias"). As such, to take it into account, we computed the average of all ratings, the average rating per user and then the average rating per item. Those fixed values were included in the computation of the new features for users and items, and we adapted our error and results-exporting functions to take it into account.

  While one could think the best bias estimation wouldn't consider the empty entries of the ratings sparse matrix, we had better results by taking them for the global, user and item mean (thus our means were often small).

  We got our best result from the version with bias. For the two of them, we took for the stopping criterion the point where the training rmse change would be less than 1e-05. We observed from the error plots below (see figures 3 and 4) that lowering the criterion would only marginally

improve the result for a much greater computation time.

- **external library *Nimfa*:**

  The Nimfa library wasn't really good, because it isn't intended for recommendation systems, but more for pure matrix factorization. It even didn't let us specify a λ parameter.

- **external library *pyspark.mllib*:**

  This library gave us our best results, and it was quite fast (but to get our final result we had to let it run for 40 minutes on one average laptop). For our result, we set the number of iterations (a parameter of the ALS model from spark, which determine the number of passes through the dataset) to 250. Setting it to a higher value only increased the time for a marginally better result.

For each model, there was two or three basic parameters : The *rank* of the approximation (K in the lecture notes) and the λ of the regularization, with sometimes one for the users and one for the items.

Those parameters were tweaked using grid search. For each model, we noticed that the best *rank* was usually around 100 : More would lead to the training error decreasing, but the testing error would increase as the result of overfitting. Using a lower *rank* would result in both the training and testing error increasing.

For the regularization parameter λ, we noticed that a good value was 0.09 for this data. We also noticed that when a λ_{user} and a λ_{item} could be specified separately, we could have good result in setting $\lambda_{user} = 0.28$ and $\lambda_{item} = 0.028$, with the λ_{user} being 10 times higher than the λ_{item}. We think this can be explained by the fact that there are 10 times less users than item.

However, our best results were obtained using the same $\lambda = 0.09$ for both the users and the items, even though some of our libraries, for example *pyspark.mllib*, which only takes one λ parameter, says in its documentation that it also scales the parameter accordingly for the users and the items.

3.2 Cross-validation

We decided not to do any kind of cross-validation, since each run of most algorithms already takes at least 25 minutes, sometimes much longer. Also, the score we got in local for the test data (10% of the whole thing) is consistent with what we get on kaggle.

3.3 Model accuracy

For the computation of the error, we used the *RMSE* over the given entries. This is also what is used in the Kaggle competition.

$$RMSE(W, Z) := \sqrt{\frac{1}{|\Omega|} \sum_{(d,n)\in\Omega} [x_{dn} - (WZ^\top)_{dn}]^2}$$

Where Ω is the set of all our given entries in the ratings matrix, W is the feature matrix of the users (dimension 10000 x K) and Z the feature matrix of the items (dimension 1000 x K).

For the kaggle submission, we tried to submit rounded (to the nearest int) and unrounded submission. We quickly found out that unrounded submissions yield the best score.

4 Results

We give here our best results for each model. While in local the ALS with bias is the best, on the kaggle competition, Spark is a little bit better. We also couldn't get any graph for Spark since we can't access the error during the computation.

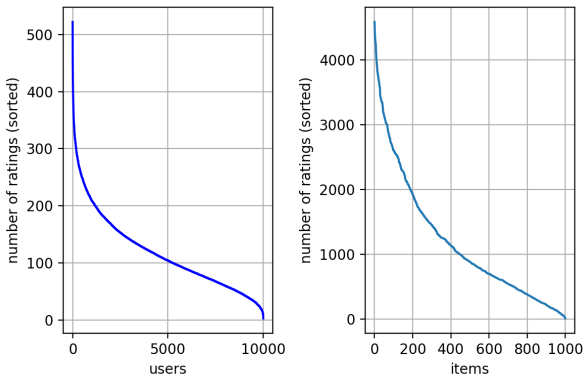

Figure 1: (Left) Sorted number of ratings per user. (Right) Sorted number of ratings for each item.

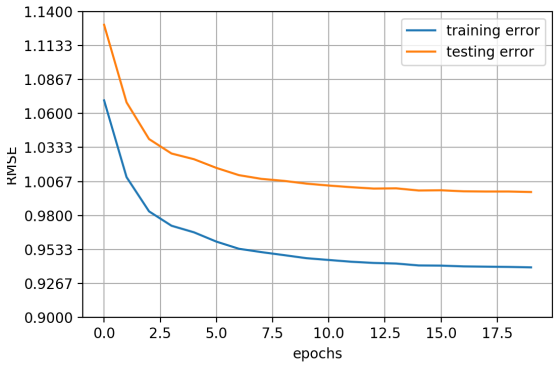

Figure 2: RMSE for the SGD model

Model	**Testing accuracy**	***rank***	λ_{user}	λ_{item}
SGD	0.99839302	100	0.09	0.09
ALS (without bias)	0.98380363	110	0.09	0.09
ALS (with bias)	0.98358184	110	0.09	0.09
Nimfa	3.48711704	100	/	/
pyspark.mllib	0.987689412983	100	0.09	0.09

Table 1: RMSE and best parameters for our models

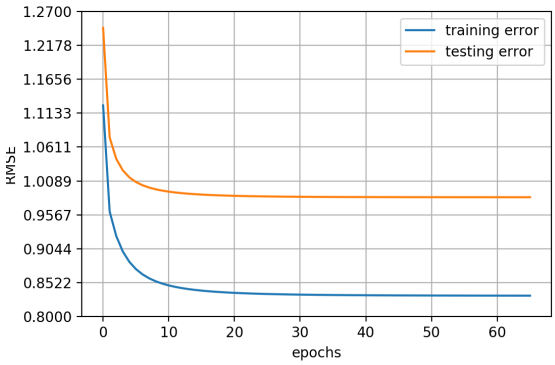

Figure 3: (Left) RMSE for the ALS without bias model.

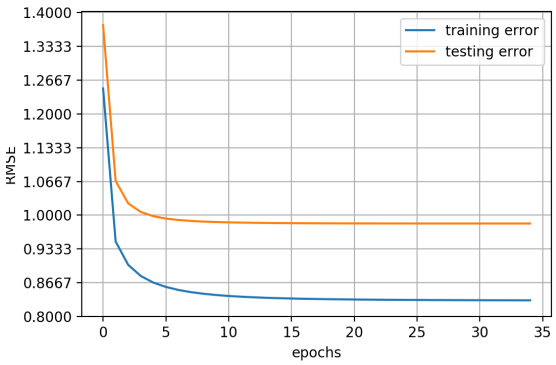

Figure 4: (Right) RMSE for the ALS with bias model.

5 Conclusion

Since most of the work is parameter tuning, we tried many methods and libraries. In the end the Spark library was the best one (our self-implemented ALS with bias implementation being the close second), and quite fast (the spark library is very quick, since it uses the whole CPU with great parallelization of the task).