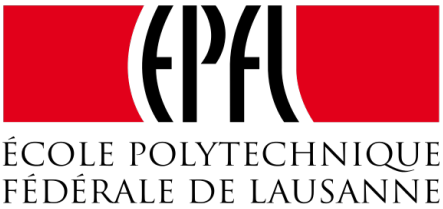
Machine Learning : Project 2
Recommender System

Boujdaria Omar, Ndoye Mohamed, Rinaldi Vincent
EPFL - Fall 2018

December 20, 2018



Contents

***Abstract*—A recommender system refers to a system that is capable of modeling preference of a set of items for a user, estimate a rating for an unrated item, and recommend the top items. One key reason why recommender system are booming in modern society is that people have too much options to choose from due to the prevalence of communication.**

I. Introduction

The dataset consists of tuples of *(user,item,rating)* : sparse ratings of 10.000 users for 1.000 different items. All ratings are integer values between 1 and 5 stars, and no additional information was given concerning either the items or users.

In order to establish a good recommender system, our goal is to build models and methods that predict accurately the rating a given user would give to an unseen movie. For the sake of interpretation, we will consider that the items are movies and we will use the terms item and movie interchangeably.

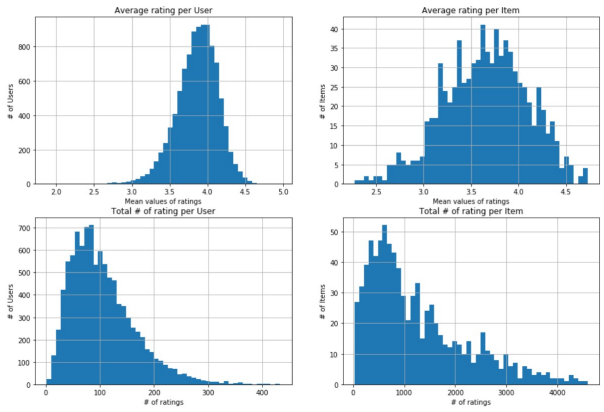


Figure 1. Distribution of the ratings.

Figure 1 shows the distribution of the ratings. We can observe that the average rating per item has a higher variance than the same average per user.

Figure 2 and Figure 3 shows the relationship between the total number of ratings and the average rating for both users and items. We can see a positive correlation between the average rating of an item and its number of ratings (*Pearson Correlation = 0.71*) : this indicates that the more ratings an item has, the higher its average rating will be. This relationship does not hold for users (*Pearson Correlation = 0.18*), hence, the frequency at which a user posts rating does not bias his ratings.

II. Models and Methods

A. Baseline Model

In this subsection we will consider a simple model : the baseline model.

Let N be the total number of ratings, and N_u,N_m be the number of ratings for user u and movie m respectively. We predict the rating of user u for movie m as :

$$\hat{r}_{um} = \mu + \alpha_u + \beta_m$$

$$\mu = \frac{1}{N}\sum_{u,m} r_{um} \quad - \quad global\ average\ rating$$

$$\alpha_u = \frac{1}{N_u}\sum_{m}(r_{um} - \mu) \quad - \quad user\ bias$$

$$\beta_m = \frac{1}{N_m}\sum_{u}(r_{um} - \alpha_u - \mu) \quad - \quad remaining\ item\ bias$$

$(\mu + \alpha_u)$ can be interpreted as the average opinion of user u, and $(\mu + \beta_m)$ the average quality of movie m.

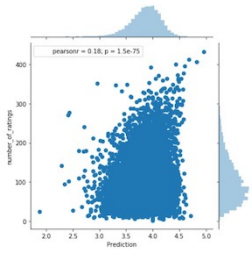


Figure 2. Correlation between the prediction and the number of ratings per user.

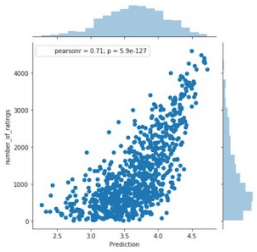


Figure 3. Correlation between the prediction and the number of ratings per item.

Given the size of the dataset, computing all these parameters (global mean, user and item biases) required a lot of time. The performance was very bad ($>$ 15 hours for a full run). Based on this observation, we decided to install Spark engine and to only use Resilient Distributed Dataset (RDD) transformations [1] and this led us to a complete run in less than 5 minutes.

B. Matrix-Factorization (ALS) and Baseline combined

One way to represent rating data is to use a user-item matrix X, where x_{ij} is the rating of the i^{th} user for the j^{th} item.

Matrix-Factorization algorithms work by decomposing the user-item matrix into the product of 2 matrices W and Z : low-dimension, tall matrices. The user-item matrix is the matrix where the value in the i^{th} row and the j^{th} column represents the rating of user i for item j.

For the implementation, we used PySpark Machine Learning library [2]. more precisely, we used regularized alternating least-squares (pyspark.mllib.recommendation.ALS).

For this model to work correctly, there should be no new user or item in the ratings that we want to predict. However, the *sample_submission.csv* file has new items (item 'c835' to 'c1000'). Based on that, we decided to predict the pairs containing those new items with the Baseline Model that will give them a default item bias (computed as the mean of the item biases) and the other pairs are predicted with the Matrix-Factorization model. At the end, the predictions are combined and the submission file is created accordingly.

C. Matrix Factorization with Stochastic Gradient Descent

There exists an alternative training algorithm for the Matrix Factorization purpose which is Stochastic Gradient Descend (SGD). Matrix factorization is often related to low-rank matrices and singular value decomposition (SVD). For the Netflix prize competition, Simon Funk popularized a regularized iterative SVD technique starting from a sparse matrix (with users as rows, movies as columns, and ratings given by a user to a movie as values) to predict the missing movie ratings from users. To implement Matrix Factorization with Stochastic Gradient Descent, we got our inspiration from Simon Funk's technique, which then also make use of a Stochastic Gradient Descent.

Matrix factorization assumes that [3] :

- Each user can be described by k attributes (or features).
- Each item (movie in our case) can be described by an analogous set of k attributes (or features).
- If we multiply each feature of the user by the corresponding feature of the movie and add everything together (like a dot product between two feature vectors of the same size), this will be a good approximation for the rating the user would give to the movie.

We then assume for this model that a user u rating for item i can be described simply by the dot product of the user and item latent vectors :

$$\hat{r}_{ui} = q_i^T * p_u$$

where p_u represents the affinity of user u for each of the latent factors, and similarly with the vector q_i for item i. Concerning the notation, we will now write the estimated rating $\hat{r}_{ui}$ as the result of $p_u * q_i$ (dot product) assuming that q_i has been transposed.

In order to apply SGD for our technique, what we are looking for is the value of the derivative of :

$$f_{ui} = (r_{ui} - \hat{r}_{ui})^2 = (r_{ui} - p_u * q_i)^2$$

with respect to any p_u and q_i, such that the loss function we want to minimize is :

$$f = \sum_{r_{ui} \epsilon R} (r_{ui} - \hat{r}_{ui})^2$$

The derivative of f_{ui} with respect to a given vector p_u is given by :

$$\frac{\partial f_{ui}}{\partial p_u} = \frac{\partial}{\partial p_u} * (r_{ui} - p_u * q_i)^2 = -2 * q_i * (r_{ui} - p_u * q_i)$$

Symmetrically with respect to a given vector q_i :

$$\frac{\partial f_{ui}}{\partial q_i} = \frac{\partial}{\partial q_i} * (r_{ui} - p_u * q_i)^2 = -2 * p_u * (r_{ui} - p_u * q_i)$$

The SGD whole procedure is then [4] :

- Initialize every vectors p_u and q_i (randomly)
- For a chosen number of time :
  - For all existing ratings r_{ui} :
    - compute $\frac{\partial f_{ui}}{\partial p_u}$ and $\frac{\partial f_{ui}}{\partial q_i}$
    - update p_u and q_i with the following formulas :
      $p_u \leftarrow p_u + \gamma * ((r_{ui} - p_u * q_i) * q_i - \lambda * p_u)$
      $q_i \leftarrow q_i + \gamma * ((r_{ui} - p_u * q_i) * p_u - \lambda * q_i)$

Where γ is the *Learning Rate* into which we merged the constant factor -2 and λ is the *Regularization Term*.

At the very beginning, we tried Funks original algorithm, which was a little bit different : instead of computing every factors in once, his algorithm trains the first factor first, then the second, then the third, until the last factor. In addition to that, we were initializing our features to a value that revealed to be efficient for the rest of the computation, which is $\sqrt{\sum_{r_{ui} \epsilon R}(r_{ui})/n}$ where n is the total number of existing ratings) since the initialization of the features has an impact on the fit and the performance of the model. Also, when training a column, instead of always iterating the same number of times, we were prematurely stopping the column training when the train RMSE was not improving a lot after one iteration (if the improvement was less than 0.001) so we could accelerate the whole computation. This implementation was pretty interesting. However, iterating for every column was costing a huge amount of time, and we couldn't perform an efficient Local Cross-Validation on this version of algorithm to find the optimal hyperparameters in a reasonable amount of time. We only computed a very overfitted model (which computation took 17 hours) with parameters recommended by some researchers that already worked on this problem, that gave us an interesting RMSE of 1,056 on the *CrowdAI* server, but since we cannot justify our hyperparameters choice, we decided to focus on a faster algorithm that computes the SGD procedure very efficiently by using *Scipy 'sparse'* matrices from the *Scipy* library.

To have the most possible optimal SGD computation that leads to the best latent vectors for each user and item, and get the most accurate predictions for a totally unknown dataset, we have to select precisely and specifically our parameters

according to some verification tests. We then ran a 5-fold Local Cross-Validation for each hyperparameter such as the learning rate, the regularization term, and the number of features. The Cross-Validation steps are explained in more details in the next section.

According to the results of the Cross-Validation we performed to select our hyperparameters, we chose to set *num_features* = 10, *lrate* = 0.002 and *regularizer* = 0.02. We also decided to set *num_epoch* to 20, which is a pretty good value to train sufficiently our model without taking too much risk of overfitting.

Then, for each user-movie pairs that exists in the train set (we iterate over the list of user-movie pair indices we computed before, which accelerates the process when using *Scipy 'sparse'* matrices and *Scipy* functions), we update their corresponding feature vectors *num_epoch* times following the SGD steps we described previously. Note that we first should save the current value of p_u and q_i before updating them to make the computation of their next value correct, according to the update formula. After each *'epoch'* iteration, we calculate the RMSE (Root Mean Square Error) between the train set ratings and our actual prediction ratings to keep an update of the evolution in the case we still would overfit the model.

$$RMSE = \sqrt{\frac{\sum_{r_{ui} \epsilon R} (r_{ui} - \hat{r}_{ui})^2}{n}}$$

Where n is the number of ratings in the train set.

Finally we can simply retrieve our predicted ratings by performing a dot product between the associated latent factors of each user-movie pair that belongs to the dataset we have to predict.

III. Results

A. Baseline Model result

Using this method, the RMSE we obtained was 1.057. The parameters that we can tune are the default biases.

For the default bias of the users, we chose it to be the mean of the biases of the users that we already found and for the default bias of the items we chose it to be the mean of the item biases we already computed.

B. Result of Baseline and Matrix-Factorization with ALS combined

For this method, the best RMSE that we obtained using the parameters described below was 1.177, which is worst than the baseline model.

The Matrix-Factorization model uses several parameters and the two most important are the rank and the regularization parameter. For the first one, we chose it in accordance with what is generally done, i.e. between 5 and 25, so we chose 20. For the regularization parameter we chose the value that minimizes the error, i.e. we tried between several values and finally we found out that $\lambda = 10^{-5}$ was good (see Figure 4).

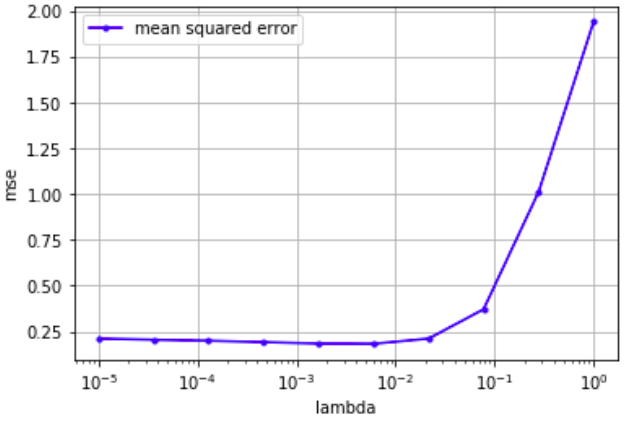


Figure 4. Mean Squared Error vs Regularization paramater

C. Result of Matrix-Factorization using SGD

The performance of our model will be based on the value with assign to our hyperparameters. The main purpose is to avoid overfitting or underfitting the model. A well-known great local validation technique to try finding the best adapted paramters for a specific model is Cross-Validation. We had to tune the following parameters : *num_features* representing the number of feature each latent vector must contain, *lrate* which is the learning rate, and *regularizer*, the regularization term. To tune these parameters, we performed on each of them a 5-Folds Cross-Validation on five different possible value they can take, while the two other parameters were set at a chosen reasonable constant value. We decided to set the constant number of features to 25, the constant regularization term to 0.01, and the constant learning rate to 0.0025. We apply a 5-Folds Cross Validation on *num_features* for values [10; 20; 30; 40; 50], on *lrate* for values [0.001; 0.002; 0.003; 0.004; 0.005], and on *regularizer* for [0.001; 0.005; 0.01; 0.015; 0.02].

Figure 5 shows the evolution of the Root Mean Square Error (RMSE) on the train set and the test set. It's obvious to see that an overfit is apparent if we have more than 10 features in each latent vector. We then decided to set 10 as the number of features for this model.

Figure 6 presents the effect of the learning rate on the model. As we can see in the plot, increasing learning rate progressively leads to overfitting, even if we employ heavy regularization (we recall that the regularization term is set to 0.01). It is then better to choose a small learning rate. According to the plot, it seems a good idea to choose a learning rate equals to 0.002, since it is at this point that the two lines are diverging, and we don't have any information for a smaller learning rate.

Finally, Figure 7 shows the RMSE on the train and test set depending of the given regularization parameter value. The results lets us understand that a low regularization value term will result in overfitting the training data (for a regularization rate equal to 0.001, the RMSE on the train set is equal to 0, whereas, for the test set acting as unknown values, the RMSE is very high). The error will go down as more regularization is enforced. Then we conclude that it is better to choose a high regularization rate : we choose 0.02 as the value of our regularization term.

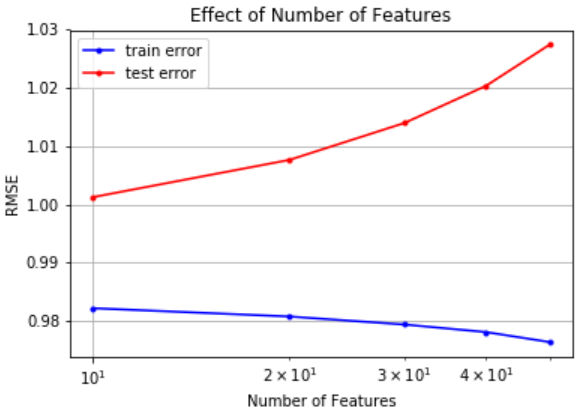


Figure 5.

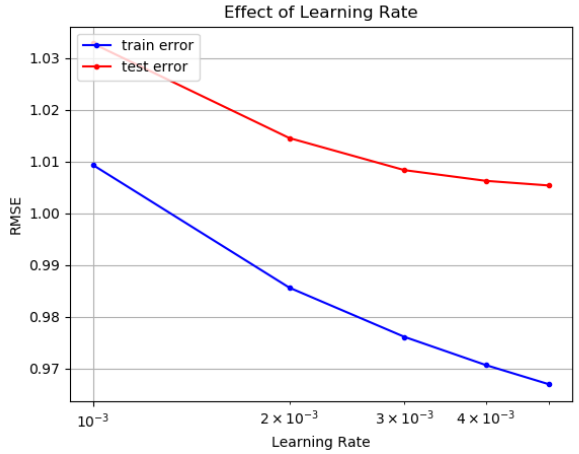


Figure 6.

The script takes about 10 minutes to execute, which is way more acceptable than the SVD Regularization method we implemented at first. However, on the *CrowdAI* server, we get an RMSE equal to 1.137, which is pretty bad, comparing to the case of the SVD Regularization (1.056) on which we set a very high number of features (50), a high learning rate (0.035) and pretty much the same regularization parameter (0.01).

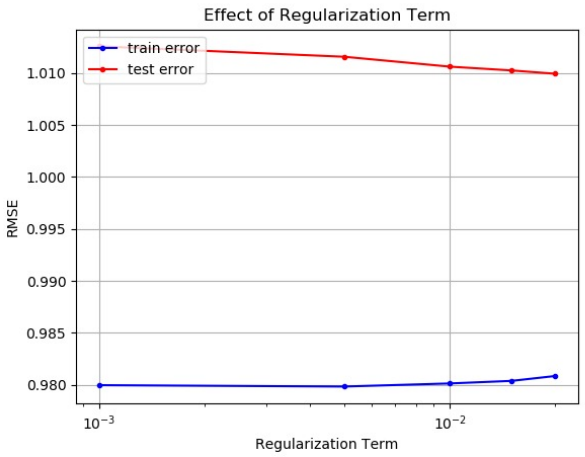


Figure 7.

IV. Discussion and Summary

From the results, we can surprisingly observe that combining the Baseline and Matrix-Factorization models we obtain a less accurate prediction than Baseline Model alone.
Indeed the RMSE of the baseline model is smaller than the one of the Matrix-Factorization.

In addition, even if we could perform an interesting in-deep research of the hyperparameters for the Matrix Factorization method using SGD, in contrary to the SVD Regularization method inspired from Simon Funk on which we had troubles to efficiently investigate on the most optimal parameters to use, since the computation time of this algorithm was too high, our SGD Matrix Factorization model didn't seem to be enough performant, whereas we could obtain a nice error rate with the SVD method using exhaustive parameters selection (the selection that led to our best RMSE was even really overfitting our model). Some tasks in the future need to be perform, such as trying to find a way to improve the SGD computation on more than 1 million elements for the SVD method to be able to research efficiently the best hyperparameters since this method seems to perform well. Also, we can keep focusing on Matrix Factorization using SGD, and improve it, by for example using bias parameters to have more precision.

References

[1] RDD Programming Guide
`spark.apache.org/.../rdd-programming-guide`

[2] Pyspark
`spark.apache.org/.../mllib.recommendation.ALS`

[3] Matrix Factorization
`blog.insightdatascience.com/.../all-that-jazz`

[4] SGD Procedure
`nicolas-hug.com/.../matrix_facto_3`