A Review and Benchmark of Recommender Systems

Advances in Data Mining - Assignment 2

Daniele Lain
LIACS, Leiden University
s1602403@umail.leidenuniv.nl

Giulio Lovisotto
LIACS, Leiden University
s1595687@umail.leidenuniv.nl

ABSTRACT

With the increasing size of available and recorded data, recommender systems are indispensable tools to help users navigate through such large amount of information. Recommendation systems are pervasively used to personalize the user experience on e-commerce websites and on review and aggregator websites. Major problems of such systems are scalability, because of the size of the datasets, and sparseness of the user data, because the observed part of such datasets is usually very small. There exist many approaches to solve such criticalities. We review and compare four of these algorithms, namely a baseline naive approach, Collaborative Filtering, Gradient Descent and Alternating Least Square, on the MovieLens dataset. We define a simple framework to compare recommendation algorithms with a k-fold-validation technique, and benchmark the accuracy, memory usage and running time of the algorithms. Some of our findings show that methods based on matrix factorization are more accurate, and that the parallelized version of Alternating Least Square algorithm currently grants the best trade-off between computational time and accuracy.

1. INTRODUCTION

Nowadays, it is a common perception to be submerged by choices. Users visiting an e-commerce site can be confused by the enormous amount of similar products, divided by categories and by features that maybe are not always enough to let users navigate such a big sea of information. Users visiting a movie website would like to look up quickly through the thousands of movies that can be the result of a filtered search, to find what they could like the most. From the users' point of view, this behavior is appreciated since it helps the usability of any platform. From a company who runs its business on such a website, such type of recommendation is appreciated because it could lead to a growth in sales.

This problem of guessing what a user could like from all the items on the system is what *recommender systems* try to solve. They use different approaches, which can be summarized as follows: *content filtering* approaches create profiles for each user or product to characterise its nature. *Collaborative filtering* approaches rely on past user behaviour. They analyze the relationships between users and the interdependencies among products to identify new user-item association. Collaborative filtering approaches suffer from the *cold start* problem, which means that for new products or users they do not have any information to rely on to recommend. We can use *neighborhood methods* or *latent factor models*. Neighborhood methods compute relationships between items or users, latent factor models try to characterize items and users on a predetermined number of features inferred from rating patterns.

When speaking about recommender systems we can discriminate between two possible behaviors [8]. The first one is called *prediction*. It consists of finding a numerical value which express the predicted likeliness of an item for a particular user, or vice-versa. The second type is called *recommendation*. It consists of finding a list of items that the considered user will like the most, or vice-versa.

Matrix factorisation methods are one of the most successful realization of the latent factor models. They are superior to the classic nearest-neighbour techniques under different aspects. They provide scalability, prediction accuracy, flexibility. They rely on different type of input data. The most convenient is explicit feedback, also called *ratings*. Usually ratings are presented in form of a sparse matrix. Another way is to use implicit feedback, which is obtained observing user behaviour, and usually forms a densely filled matrix.

Collaborative filtering is popular and widely deployed from Internet companies such as Amazon, Netflix, Google News [11]. In 2007, Netflix proposed a large-scale data mining competition called the *Netflix Prize*, with a big monetary prize. It challenged competitors in implementing a recommendation system that could reduce by 10% the error of the CineMatch system Netflix uses [11]. The competition saw lots of submissions, leading to improvements on different recommendation techniques [1, 10, 5, 2, 11, 9], proving that this is an active research fields that draws lots of attention both from researchers and companies.

Contribution. In this paper, we use the MovieLens 1M dataset to benchmark different recommender system algorithms. For every algorithm, we measure the execution time, the accuracy, and the memory requirements. Moreover, we give some insights on the bounds on the required time and memory of the implemented algorithms in general. We analyze how the time and memory requirements changes as a function of the size of the dataset: either users, movies, or ratings.

1.1 Organization

The paper is structured as follows: in Section 2 we present previous related work, in Section 3 we give a formal definition of the recommendation problem, in Section 4 we describe the algorithms we implemented and studied, in Section 5 we describe the dataset used, and we present our results and their interpretation. In Section 6 we give some conclusions and propose some future work.

2. RELATED WORK

In this section we present some of the related works in the recommender system literature.

Goldberg et al. were the first to develop a recommender system called Tapestry [3]. Tapestry used a collaborative filtering approach to filter e-mails, using explicit opinions of people in the research group, which knew each others interests.

Clustering techniques [6] aims to identify groups of users who have similar preferences. Once a cluster is identified, predictions can be obtained averaging the ratings of the users in that cluster. These approaches can be used as the first step for shrinking a candidate set of possible neighbours. Dividing the population in clusters hurts the overall accuracy of the recommendation system, but speeds up the prediction. Therefore, these techniques imply a trade-off between accuracy and throughput.

In an item-to-item similarity scenario, a fast approach is pre-computing all the item-to-item similarities. However, this has high requirements on space. Sarwar et al. studied the tradeoff between precision and memory consumption by saving only a number of similar items [8].

Recently, matrix factorization approaches have proved to be the most effective and accurate techniques to develop such a recommender system. They model both users and movies by giving them coordinates in a low dimensional feature space, and then they try to solve a low-rank approximation problem.

The Gravity Recommendation System (GRS) [9], is a solution that uses stochastic gradient descent to minimize the prediction error. It was developed by Takacs et al. in 2007, and it is based on Simon Funk's SVD [2], with the difference that each factor is updated simultaneously, and the whole matrix is randomly initialized.

Alternating Least Squares with Weighted λ Regularization (ALS-WR) [11], is a parallel algorithm proposed by Zhou et al in 2008. It is designed to be scalable, and the computational steps are totally parallelizable. *forse qualche info in più?*

3. PROBLEM STATEMENT

In a recommendation system there are 2 classes of entities, users and items. Users have preferences for certain items. We have a list of n users $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$ and a list of m items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$. Let $R = \{r_{ui}\}_{n \times m}$ denote the user-movie matrix where R_{ui} represent the rating score of movie i rated by user u, and the value is either a number, or it is missing. The aim of a recommender system is to estimate some of the missing values in R, based on the known values. The data is stored as a utility matrix which gives, for every pair user-item, a value that represents what is known about the degree of preference of that user for that item. The utility matrix is usually sparse; this means that most of the values are missing. Normally the goal is not to predict every blank entry, but rather to determine some entries in each row which is likely to be high. This means, we usually focus on predicting high-valued ratings. We refer to a prediction of a rating from user u of the item i with the hat superscript notation, that is, $\hat{R}_{ui}$ is the prediction of R_{ui}.

In its basic form, matrix factorization characterizes both items and users by vectors of factors inferred from item rating patterns. This vector of features has a dimensionality of n_f, and it models user-item interaction as inner products in that space.

Every item i and every user u are associated with a vector, $q_i, p_u \in \mathbb{R}^{n_f}$ respectively. These vectors measure the extent to which the item possesses those features. The resulting dot product captures the interaction between user u and item i, such that

$$\hat{R}_{ui} = {q_i}^T \cdot p_u.$$

The aim of a recommender system is to find an approximation of the rating matrix R, which we refer to as $\hat{R}$, such that the prediction error is minimized.

4. ALGORITHMS

In this section we formally describe the algorithms we implement and benchmark.

4.1 Naive Approaches

These naive approaches aim to be fast. They are based on the idea that the user will usually have an average behavior, and they return an average value as the predicted value.

Global. In this technique, the prediction for the rating R_{ui}, for user u and item i is the mean of all the ratings:

$$\hat{R}^{(g)}_{ui} = \frac{\sum\limits_{(k,j)} R_{kj}}{|R_{kj}|} \text{ s.t. } R_{kj} \text{ non empty entry in } R.$$

Item. The prediction for the rating R_{ui} is the mean of all the ratings for the item i:

$$\hat{R}^{(it)}_{ui} = \frac{\sum\limits_{j} R_{ji}}{|R_{ji}|} \text{ s.t. } R_{ji} \text{ non empty rating for item } i.$$

User. The prediction for the rating R_{ui} is the mean of all the ratings given by user u:

$$\hat{R}^{(us)}_{ui} = \frac{\sum\limits_{j} R_{uj}}{|R_{uj}|} \text{ s.t. } R_{uj} \text{ non empty rating given by user } u.$$

User-Item. The prediction for the rating R_{ui} is given by the formula:

$$\hat{R}_{ui}^{(u-i)} = \alpha \cdot \hat{R}_{ui}^{(us)} + \beta \cdot \hat{R}_{ui}^{(it)} + \gamma,$$

where the parameter α, β and γ are estimated using linear regression.

4.2 Collaborative Filtering

Collaborative filtering approaches can be divided into user-based and item-based.

User-based. The method consists on recommending items to the user based on what other like-minded users liked. First, it finds similar users using a similarity measure. These similar users are called *neighbors*. Then it considers the first l more similar neighbours N, and computes the prediction value $\hat{R}_{ui}$ with a weighted sum:

$$\hat{R}_{ui} = \frac{\sum_{j \in N} R_{ji} \cdot s_{uj}}{\sum_{j \in N} s_{uj}},$$

where j's are the members of N, R_{ji} is the rating given by user j to the item i, and s_{uj} is the similarity between user u and j.

Item-based. This method works in the same way as the user-based method, but it considers the similar items instead of the similar users as neighbours N. Then:

$$\hat{R}_{ui} = \frac{\sum_{j \in N} R_{uj} \cdot s_{ij}}{\sum_{j \in N} s_{ij}},$$

where j's are the members of N, R_{ui} is the rating given by the user u to the item i, and s_{ij} is the similarity between items i, j.

As distance similarity we test both *cosine similarity* and *Pearson correlation*. When computing the similarity, we consider only the elements rated by both the users. To choose the neighbours for an user u to predict the rating R_{ui}, we only pick users that have already rated the movie i, and vice-versa for the items.

4.3 Gradient Descent

In the stochastic gradient descent, we iterate over all the ratings in the training set trying to minimize the prediction error. For each given training case the system computes $\hat{R}_{ui} = U_u^T I_i$, and its associated prediction error:

$$e_{ui} = R_{ui} - \hat{R}_{ui}.$$

Then it modifies the parameters proportionally to η in the opposite direction of the gradient, and applies a regularization with factor λ to prevent large weights, yielding:

$$U_u \leftarrow U_u + \eta \cdot (e_{ui} \cdot I_m - \lambda \cdot U_u)$$

$$I_i \leftarrow I_i + \eta \cdot (e_{ui} \cdot U_u - \lambda \cdot I_i)$$

This approach tries to minimize the error on the prediction, thus better approximating R_{ui}.

We implement the basic algorithm described in [9]. We do not consider any of the improvement described in the paper: dates, constant values in matrices, and values rounding.

4.4 Alternating Least Squares

The ALS algorithm is another matrix factorisation approach. It tries to minimize the error by alternating two steps: fixing U_u and optimizing I_i, and fixing I_i and optimizing U_u. When all the U_u are fixed, the algorithm recomputes the I_i by solving a least-squares problem, and vice-versa.

ALS is convenient when the system can use parallelization. In fact every computation for U_u and for I_i is independent from the other user and item factors, respectively.

We implement the Alternating Least Squares algorithm with the Weighted-*lambda*-Regularization (ALS-WR) optimization, which is an approach that empirically prevented overfitting on the 100M MovieLens dataset in the work of Zhou et al. [11].

To reduce computation time, and as the authors suggest, we implement a parallel version of the ALS algorithm. This computes every I_i and U_u in parallel for every different movie and user. Given the independence property stated above, this does not change the soundness or the result of the algorithm.

5. EXPERIMENTS

In this section we describe the training and test dataset, the methodology we use for the experiments, and our findings and their interpretation.

5.1 Dataset

We use the *MovieLens* (1M) dataset [7], collected by GroupLens Research from the MovieLens web site. The dataset contains 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users. We pre-process and clean up the dataset, normalizing movie IDs such that there are no "missing" movies, that is, their IDs are incremental. Moreover, we subtract 1 from the ID every user/movie such that the IDs starts from 0. This allows us to quickly load the information in a matrix and grants a direct relation between IDs and indexes in our matrices. Such mapping operation does not alter any of the results, and it is easily reversible.

5.2 Methodology

Our goal is to compute the accuracy, running time and memory consumption of every algorithm. To compute the accuracy, we consider 2 error measures. The first is the Root Mean Square Error (RMSE), which represents the sample standard deviation of the differences between predicted values and observed values. The second is the Mean Absolute Error (MAE), which also measures how close forecasts or predictions are to the eventual outcomes, but with an average of the absolute errors.

To test the performances of the algorithms we design a specific interface *recommend* as a simple framework of recommender algorithms benchmarking. It takes as input the tuple

$$recommend(users, movies, train, test, kwargs)$$

where $users$ is the list of users, $movies$ is the list of movies, $train$ is a $n_{train} \times 3$ matrix containing the train ratings, $test$ is a $n_{test} \times 3$ matrix containing the test ratings, and $kwargs$ is a serie of keyword arguments in Python style. We obtain as output the serie of elements:

$$(\hat{R}, time, memory, rmse, mae[, U, M, rmse_s, mae_s])$$

where $hatR$ is the predicted matrix, $time$ is the total time used for the computation, $memory$ is the maximum memory occupation of the process, $rmse$ is the root mean square error of the final $\hat{R}$, mae is the mean absolute error of the final $\hat{R}$. Additionally if the recommender uses matrix factorization, U and M are the two low rank matrices s.t. $\hat{R} = U^T \times M$, and $rmse_s, mae_s$ are arrays of size $steps$ (number of steps) containing the errors for each step, for both the training and the test set.

We implement the *k-fold-validation* technique for a better estimate of the accuracy of the different algorithms. Thus, we divide the ratings into $k = 5$ similar sized sets, and we use 4 of them as training set and the other 1 as test set. We do this for every possible combination and average the results.

Only for the RMSE and the MAE computation, we normalize the predicted values $\hat{R}_{ui}$ that do not lie in the range (1, 5) to 1 and 5 for the values lesser than 1 and greater than 5, respectively.

We run our experiments using Python/Numpy suite on a dual hyper-threaded octa-core Intel Xeon CPU's type E5-2667, running at 3.30GHz, with 48GB of RAM and a total of 32 cores. We run the ALS parallel algorithm on 16 cores.

5.3 Results

In this section we report the results of our experiments on the dataset, and we analyze our findings. Every reported measure is the average of the 5 runs of the 5-fold-validation technique.

We start from some well-known good parameters [4] and test different configuration setups for GD and ALS. We also test two different similarity measures, cosine similarity and Pearson correlation, for the collaborative filtering, and different sizes of the neighbours set N.

For the GD algorithm we find the best results in terms of errors on the test set with $n_f = 6$, $\lambda = 0.07$ and $\nu = 0.005$. For the ALS algorithm the best setup is $n_f = 4$ and $\lambda = 0.05$. For the Collaborative Filtering the best setup was obtained using the cosine similarity with $|N| = 5$ neighbors.

If not specified otherwise, the reported results are relative to the best configuration setup for the algorithm. In Table 1 we report the best performing results obtained for every algorithm.

5.3.1 Error measures

Figure 1 reports the two error measures, Root-Mean-Square Error and Mean Absolute Error, for every algorithm. In general the matrix factorization techniques grant us a smaller error on the recommendation. Naive approaches still grant a decent approximation, while being very simple to implement.

Algorithm	RMSE	MAE	memory (MB)	time (sec)
$global_{avg}$	1.117	0.9339	748	0.14
$user_{avg}$	1.0355	0.8290	748	0.23
$item_{avg}$	0.9793	0.7822	748	0.24
$useritem_{avg}$	0.9527	0.7633	749	19
CF_{item}	1.0875	0.8904	1049	79
CF_{user}	1.0586	0.8307	1501	158
ALS	0.8764	0.6850	1256	8626
GD	0.8806	0.6894	748	5162

Table 1: Results for the implemented algorithms

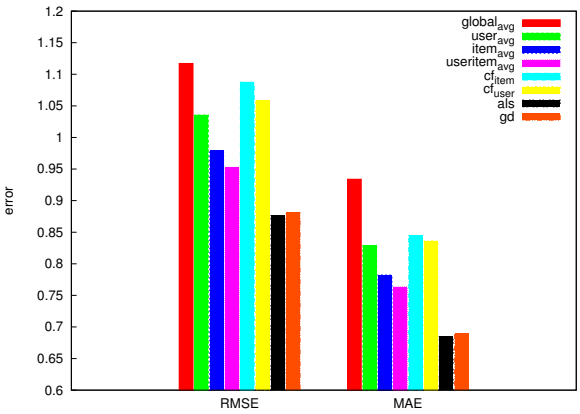


Figure 1: Average RMSE and MAE on the test set

Figure 2 reports the average error measures on the test set for the ALS and GD algorithms for varying values of lambda λ, with fixed number of features, $n_f = 4$ and $n_f = 6$ for ALS and GD, respectively. The best combination is $\lambda = 0.05$ and $\lambda = 0.07$ for ALS and GD, respectively. As we can see, the lambda value does not affect the final error, since it's only a regularization factor, used to prevent big values from influencing too much the convergence process.

5.3.2 Execution time

Figure 4b reports the execution time in minutes for every algorithm, in log scale, for a single step of the 5-fold-validation. Naive approaches are the fastest, and also collaborative filtering performs relatively well. Matrix factorization techniques are slower, with a running time on the order of magnitude of hours. The ALS execution time is relative to the parallelized version of the algorithm over 16 cores. In general, the GD algorithm execution time grows linearly with the number of ratings, since in every step we are performing a gradient descent optimization for every given rating in R, n_r. So, if we consider the update of a single value in U or M as the basic operation for the complexity, we have that the time required for a single step in GD is: $O(n_r n_f)$. This means that the time required for GD to complete a step grows linearly with the number of ratings and with the number of features.

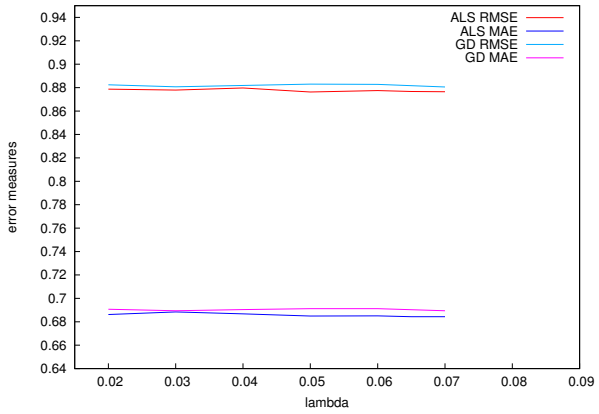


Figure 2: Average RMSE and MAE for the ALS and GD algorithm for varying λ values

The ALS algorithm execution time grows with the number of movies or users, since in that case we need to solve more systems of equations in every step. We know that solving a system of k linear equation is at least $O(k^2)$. In our case we first solve $|\mathcal{U}| = n$ systems of n_f (number of features) equations and later $|\mathcal{I}| = m$ systems of n_f equations. So, the step complexity in ALS grows linearly with the number of users and movies, and with an exponent of 2 relatively to the number of features n_f, $O(n_f^2(n+m))$.

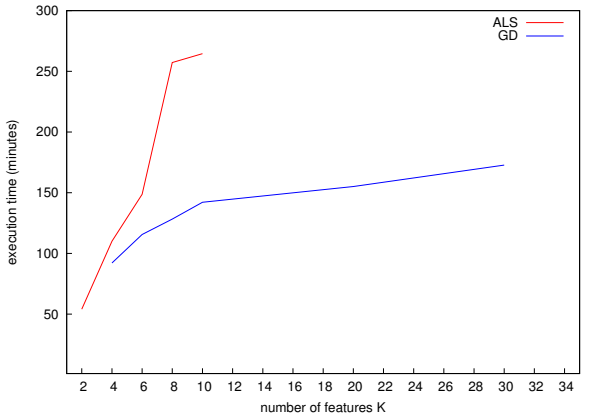


Figure 3: Average execution time for the ALS and GD algorithm for varying n_f values

Figure 3 reports the average execution time for the ALS and GD algorithms when varying number of features n_f, with a fixed lambda of $\lambda = 0.065$ and $\lambda = 0.02$, respectively. We can see that, as expected, in the ALS algorithm the execution time grows quadratically with the increment of the number of features, meanwhile in the GD algorithm it grows linearly.

We can speed up the Collaborative Filtering by doing precomputations of the similarity matrix. This might become impractical if the status of the system is continuously changing over time, e.g., if a new user enters the system, or if an old user rates another movie. The computation of the similarity matrix is the slowest part of this approach. For every movie or user, the recommender has to compute the similarity of that entity with all the other entities. This can be done with a complexity of $O\left(\frac{k(k-1)}{2}\right)$, where k is the number of entities (either movies m or users n, depending on whether the approach is item-based or user-based). Thus, also the Collaborative Filtering approach has a quadratic complexity in terms of the number of entities.

5.3.3 *Maximum memory occupation*

Figure 4a reports the maximum memory occupation, measured in Megabytes, for every algorithm. We can see that all the naive approaches have similar values. The collaborative filtering algorithm requires some more memory space due to the similarity matrix, which is either $n \times n$ or $m \times m$. The ALS algorithm has some memory overhead due to the parallelization, which requires some additional data structures. The GD algorithm has the same memory requirement as the naive approaches, since the fundamental structure is again the sparse rating matrix. In general, the memory required by the algorithms grows with the number of users and movies, for every algorithm. Collaborative filtering is the most memory consuming approach, since it has to keep a full similarity matrix for the users (or movies), of size $n \times n$ or $m \times m$, along with the ratings matrix R.

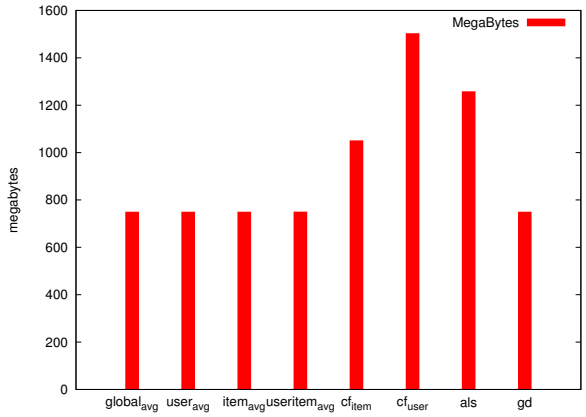
5.3.4 *Convergence*

Figure 5 reports the RMSE and the MAE error measures for the Alternating Least Squares algorithm over the iteration steps, on both the train and the test set. We can see that at the first iteration we already have a relatively small error, and that it has a fast decrease. Moreover, the error stabilizes after 15 iterations, on both the train and the test set. Notably, on the train set the error keeps decreasing (very slowly), and on the test set it alternatively makes small increments or decrements, probably due to minor overfitting.

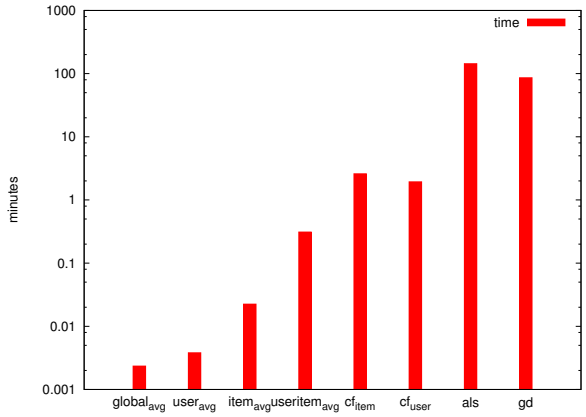
Figure 6 contains the RMSE and the MAE error measures for the Gradient Descent algorithm over the iteration steps, on both the train and the test set. We can see that at the first iteration we find a high error (> 1) that, however, has a fast decrease. The convergence is slower than in the ALS algorithm. The error on the test set stabilizes after about 40 steps, and the error on the train set keeps decreasing until the last step.

6. CONCLUSIONS

In this paper, we used the MovieLens 1M dataset to benchmark four different recommender systems. We measure the execution time, the accuracy, and the memory requirements, understanding the main characteristics of every approach. We also gave some insights on asymptotic bounds on runtime and memory occupation. Our findings show that there is a consistent ratio between runtime and error measures, meaning, the lower the runtime the higher the error. The tradeoff between such metrics is to be investigated thoroughly and should be deducted with respect to the specific applications that need to implement a recommendation system. Regarding the scalability, we believe that our results show that ALS is the most promising scalable algorithm, because of the property of computational independence. We believe

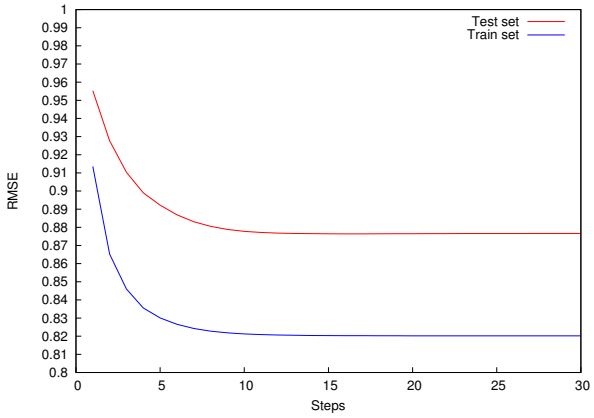


(a) Memory occupation

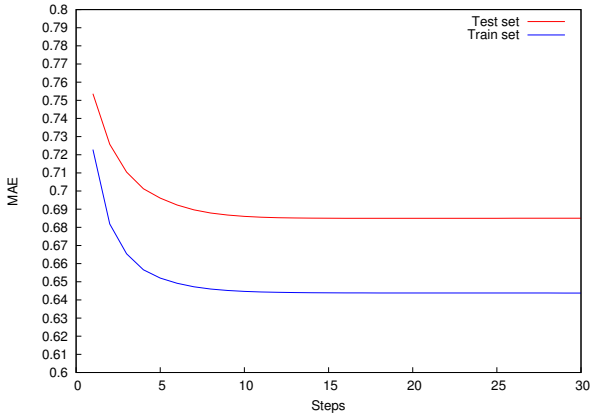


(b) Execution time

Figure 4: Average memory occupation and running time for every algorithm

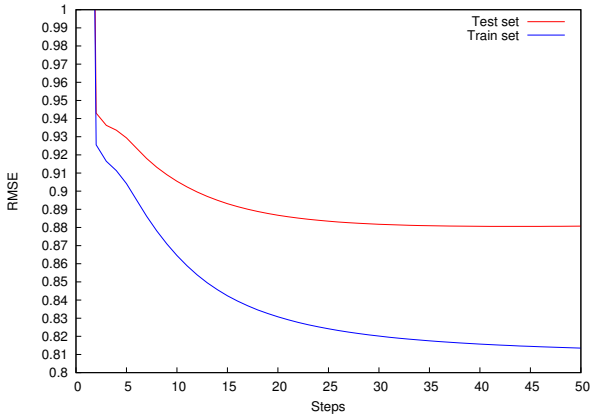


(a) RMSE for the ALS algorithm

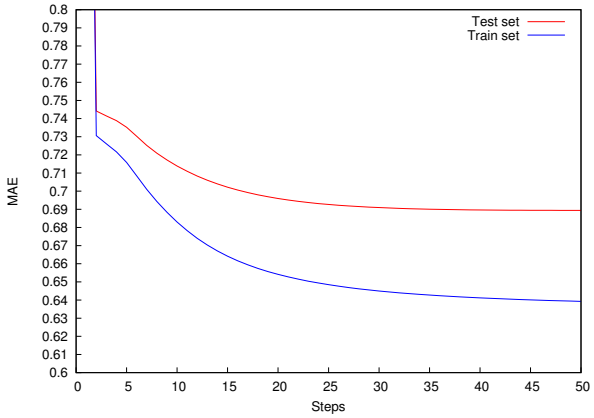


(b) MAE for the ALS algorithm

Figure 5: Average RMSE and MAE errors for the ALS algorithm over the steps



(a) Average RMSE for the GD algorithm over the steps



(b) Average MAE for the GD algorithm over the steps

Figure 6: Average RMSE and MAE errors for the GD algorithm over the steps

that such systems scale with horizontal distribution, therefore algorithms that do not need the whole global state, the whole dataset, or to perform sequential steps are the algorithms of choice to distribute the computation.

From this work, we had an in-depth understanding of the algorithms and of practical issues related to their deployment on real, big data. We had also lots of insight on working with Python/Numpy, in particular the practical issues in using sparse matrices and linear algebra functions. We conducted a formal approach to a systematic evaluation of the accuracy of the algorithms with the 5-fold cross validation technique, and using standard error measures. Working with big datasets and with complex approaches which require lots of computation allowed us to gain insights of theoretical and effective computational costs.

Future work could include implementing and benchmarking more recommendation algorithms within our simple framework. Other directions could be implementing other different factorization algorithms, and running more tests to get more consistent averages over the folds of the k-fold-validation technique.

7. REFERENCES

[1] R. M. Bell, Y. Koren, and C. Volinsky. The bellkor solution to the netflix prize, 2007.

[2] S. Funk. Singular value decomposition for the netflix prize. `http://sifter.org/~simon/journal/20061211.html`, 2014.

[3] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[4] MyMediaLite. Mymedialite recommender system library. `http://mymedialite.net/examples/datasets.html`, 2014.

[5] A. Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD cup and workshop*, volume 2007, pages 5–8, 2007.

[6] A. Rajaraman and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[7] G. Research. Movielens dataset. `http://grouplens.org/datasets/movielens/`, 2014.

[8] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[9] G. Takacs, I. Pilaszy, B. Nemeth, and D. Tikk. On the gravity recommendation system. In *Proceedings of KDD cup and workshop*, volume 2007, 2007.

[10] G. Takács, I. Pilászy, B. Németh, and D. Tikk. Matrix factorization and neighbor based algorithms for the netflix prize problem. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 267–274. ACM, 2008.

[11] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.