MovieLens Dataset Analysis

E.K. RIHANI

07 mars, 2022

Contents

1 Overview

1.1 Goal of the study

Recommendation systems are algorithms that try to predict users' preferences in order to improve the suggestions one company can make to its customers. These algorithms are widely used in movies platforms, web stores and search engines in order to improve sales, but also the user experience and the accuracy of the suggested products, movies or links.

The aim of this study is to analyze a MovieLens dataset[1] and try to create a model that can predict users' future preferences based on the ratings they gave on the movies they previously saw, using data analysis and machine learning techniques.

The main goal is to predict users' probable future ratings with the lowest possible error. A secondary goal is to obtain these results in a reasonable amount of time, on a desktop computer.

1.2 The dataset

The MovieLens datasets were created by the GroupLens research lab, from the Department of Computer Science and Engineering at the University of Minnesota.

The dataset used for this study is the 10M dataset (https://grouplens.org/datasets/movielens/10m/). The 10M MovieLens dataset contains 10000054 ratings for 10677 movies, from 69878 users, which translates into a 10677×69878 matrix, with 746.1 millions cells, 98.7 % of which are empty.

In layman's terms, the goal of this project is to fill these empty cells as accurately as possible.

The provided information includes the user's unique ID (userId), the movie's unique ID (movieId), the rating given by the user (rating), the timestamp of the rating (timestamp), the title of the movie (title) and its genre (genres).

This dataset was split into a training set (*edx*, 90% of the ratings) that will be used for building, training and tuning the recommendation system and a validation set (*validation*, 10% of the ratings) that will exclusively be used for the final evaluation of the accuracy of the recommendation system.

2 Methods and Analysis

This project will be articulated around the use of the *recommenderlab* package, which is a very powerful library for building recommendation systems that was recommended by Pr. Rafael Irizarry in his data science course.

The configuration will be described, since it can have an impact on results and performance. Then recommenderlab library and its different recommendation systems will be briefly introduced. Their respective performances will be evaluated, both in computing time and accuracy, in order to select the most promising models. Their tuning parameters will then be tuned, and the best model will finally be used against the validation set.

Code extracts will be included to illustrate how to setup and use the *recommenderlab* package.

2.1 Computer and R configuration

The code, benchmarks and report were run on the following computer configuration :

- CPU : AMD Ryzen 5 5600G
- RAM : 2x16 GB DDR4-3200
- SSD : Crucial P5 M2 NVMe
- OS : Xubuntu Linux 20.04 LTS
- R : version 3.6.3
- Packages : tidyverse[2] (v1.3.1), caret[3] (v6.0-88), data.table[4] (v1.14.0), reshape2[5] (v1.4.4), , ggrepel[6] (v0.9.1), recommenderlab[7] (v0.2-7), ggpubr[8] (v0.4.0), rmarkdown[9] (v2.11), knitr[10] (v1.34).

The code was also tested on a Windows 7 x64 SP1 computer with 16 GB RAM and R 4.1.0. It was not tested on any Mac OS system. The provided code includes alternate settings for less powerful computers.

2.2 Preparing data for the recommenderlab package

The recommenderlab package uses several specific data formats, all based on a single sparse matrix, where the ratings (binary or numeric) are located at the intersection of the user (*userId*) row and the rated object (*movieId*) column. Because ratings are numeric, this study will use the *realRatingMatrix* format.

Since recommenderlab only uses ratings, one notable limitation of this study is that it will not use some potentially useful information such as the moment of the rating or the year or gender of the movie to improve the accuracy of the analysis.

The first step of the data preparation involves the transformation of the *edx* dataset into a $userId \times movieId$ matrix (*edx_rrm*) with *rating* at each user/movie intersection. The acast::reshape2 function can be used to perform this operation. The resulting matrix can then be easily converted in a *realRatingMatrix* format.

```
edx_rrm <- acast(edx, userId ~ movieId, value.var = "rating")
edx_rrm <- as(edx_rrm, "realRatingMatrix")
```

The same operation must be performed to validate the model. To make sure the *validation* isn't used until the final validation, it was explicitly chosen not to use it *at all* until the final steps.

This method has two limits. Firstly, it uses a considerable amount of memory on large datasets and thus requires to use the *memory.limit* function on Windows systems, or to have enough total memory (RAM+swap) on Linux systems. Otherwise, R may return errors or even crash.

Secondly, some recommenderlab models may encounter problems if the training set and the validation sets don't have the same items (movies). This can be the case if a list-splitting method is used, because some items can be present in the training set, but not in the validation set.

Two equivalent strategies can be used : either add the missing items with *NA* ratings in the validation set or remove the useless items in the training set. Since modifications of the validation set are explicitly forbidden, the only option is to remove useless items in the training set, i.e. exclude from the *edx* set all movies that don't exist in the *validation* set, before building the realRatingMatrix for the final validation.

2.3 Cross-validation

In order to compare the algorithms and tuning parameters, the cross-validation gold standard is probably the hold-out method. This method holds out a portion of the initial dataset only for the final evaluation, and splits the other part into training and validation subsets.

In this study, the *movielens* dataset was already split into an *validation* (final evaluation) and *edx* set that has to be split into a training and validation set. Several strategies can be used to perform this split, such as a basic split-train or a more elaborate K-fold cross validation.

Since the goal is to be able to compare most *recommenderlab* algorithms and tuning parameters in a reasonable amount of time, the split-train validation is a good choice, because this method is K times faster than a K-fold cross validation although it typically gives noisier and more biased results for algorithm selection and tuning than a K-fold cross-validation method.

However, in order to be able to compare the performance of nine algorithms (some of them being very slow) in a reasonnable time, and perform all the hyperparameter tunings, all initial benchmarks and validations were run on a smaller subset of the *edx* dataset. In other words, the strategy was to "undertrain" in all model selections and hyperparameter tuning steps. This strategy gives much faster results, but all preliminary performance metrics should of course be considered as rough estimates. The main caveat of this strategy is that the larger amount of bias and noise can lead to less-than-optimal model or hyperparameter choices, and a much larger final error.

The split-train method can be performed by splitting the *edx* (training/validation) dataset with a regular index sampling method and then building the corresponding realRatingMatrix, or more simply splitting the training realRatingMatrix (*edx_rrm*) into smaller matrices. For example, for a 90/10 training/prevalidation ratio :

```
test_index <- sample(x = seq(1, nrow(edx_rrm)), size = nrow(edx_rrm)*0.1, replace = FALSE)
edx_rrm_train <- edx_rrm[-test_index]
edx_rrm_test <- edx_rrm[test_index]
```

The *recommenderlab* package conveniently provides an *evaluationScheme* function that can perform this simple split-train method or build K-fold cross validation datasets. This packages also has an *evaluation* function that can be very useful for accuracy evaluation of models and hyperparameters, especially in association with K-fold cross validation. These function can prove to be very useful, notably on smaller datasets. However, the *evaluationScheme* function proved to be way too slow for the 10M movielens dataset on a regular desktop computer.

Apart from the possibility of sub-optimal model selection and tuning, all the choices made in these splitting steps for benchmarking/tuning purposes have absolutely no impact on the final RMSE, since the *edx* and *validation* datasets were already created using a list-based splitting method with a fixed seed.

2.4 Performance metrics

Several metrics can be used to evaluate the accuracy of the prediction, such as the Mean Average Error (MAE, consistent with the data units), the Mean Square Error (MSE, penalizes large errors) and the Root Mean Square Error (RMSE, penalizes large errors **and** is consistent with the data units). The metric used in this study to evaluate the accuracy of the prediction is the root-mean-squared error (RMSE), given by :

$$RMSE = \sqrt{\frac{1}{N}\sum_{m,u}\left(\hat{y}_{m,u} - y_{m,u}\right)^2}$$

With N the number of ratings, $\hat{y}_{m,u}$ the rating (for the movie m, by the user u) that is predicted by the model built with the training set and $y_{m,u}$ the actual rating of the testing set. The lower the RMSE, the more accurate the model.

Recommenderlab conveniently provides a *calcPredictionAccuracy* function that computes the MAE, MSE and RMSE. As an example (using a LIBMF method) the recommendation, prediction and accuracy evaluation can be run by :

```
recommendation <- Recommender(data = edx_rrm, method = "LIBMF")
prediction <- predict(recommendation, validation_rrm, type = "ratingMatrix")
accuracy <- calcPredictionAccuracy(validation_rrm, prediction)
```

Another very useful and easy to evaluate metric is the time needed to execute the recommendation, prediction and accuracy steps. The code used in this study is quite simple :

```
start_time <- Sys.time()
# Insert_code_here #
end_time <- Sys.time()
running_time <- difftime(end_time, start_time, units = "secs")
```

The *units* argument in the *difftime* function ensures all units are kept consistent. Mixing minutes and seconds can be problematic, especially if these results are used for plotting.

2.5 About the recommenderlab methods

The *Random* method, as its name suggests, assigns random ratings to all movies for each user. This method can be used to benchmark other models.

The *Popular* method is based on the most rated items, and will thus recommend the most viewed movies. The rating is evaluated using a distance method.

The *UBCF* and *IBCF* methods are respectively an user-based and an item-based collaborative filtering methods. These methods are the oldest recommender methods (1992) and are based on a "similar users like similar things" strategies. UBCF focuses on the users and is based on the hypothesis that users with comparable preferences will rate movies similarly, and thus tries to find for each user a k-neighborhood of the most comparable users (by using cosine similarity, by default) and aggregate their ratings to form a prediction. IBCF works similarly, but is focused on items, the core hypothesis being that users prefer movies that are similar to movies they already like. The IBCF method uses a comparable approach as the UBCF but tries to find similarities between movies by computing the k-neighborhood of movies instead of users.

The *ALS*, *ALS_implicit* and *LIBMF* methods are all based on the same mathematical concept : matrix factorization. Matrix factorization is a widely used method that tries to approximate the entire rating matrix $R_{u\times m}$ as the product of smaller-sized matrices $P_{k\times u}$ and $Q_{k\times m}$: $R \approx P'Q$. In other words, each user and each item are summarized by k dimensional vectors, where k is a small fixed number. The underlying mathematical problem is then to minimize the distance (least squares error) between each known $r_{u,m}$ rating and the corresponding intersection of the product of the two smaller-sized matrices $p_u^T.q_m$.

In ALS and ALS_implicit optimization strategies, p_u is fixed and q_m is optimized by minimizing the square error, then q_m is fixed and then p_u is optimized, hence the *Alternating Least Square* name. The ALS method is used for explicit data, which are strictly correlated to the values one is trying to predict. The ALS_implicit method is designed for implicit data that reflect the interactions between the user and the movie and is only indirectly linked to the rating. In this study, the ALS method would try to predict user ratings using user ratings, whereas the ALS_implicit method would try to predict user ratings with clicks, fast-forwards, number of times viewed… Since these kind of data isn't present in the MovieLens dataset, using the ALS_implicit method makes little sense.

LIMBF is an open-source library was created by Chin et al, from the Taiwan University, in 2014. This algorithm uses a gradient descent approach, which is an iterative algorithm that compute the local gradient of this distance and adjusts the p_u and q_m *against* the gradient in order to approach a better value. This specific library also aims to make full use of the computing power of modern processors (SSE and AMX instructions, multithreading) in Matrix Factorization.

The *SVD* and *SVDF* (Funk's SVD) methods are both Singular Value Decomposition methods. Singular Value Decomposition's goal is to decompose a $m \times n$ matrix A in three smaller matrices $A_{n\times m} = U_{m\times r} \times W_{r\times r} \times V^T_{r\times n}$ with U the matrix of the eigenvectors of $A \times A^T$, W a diagonal matrix of the singular values (square root of the eigenvalues of $A^T \times A$) and V matrix of the eigenvectors of $A^T \times A$. SVD can be used to efficiently minimize least square errors in order to build prediction algorithms. Funk's SVD is an improvement on the SVD model, that was specifically designed to address sparse matrices problems commonly met in recommendation systems.

2.6 Performance of the different models

Some models may be faster and/or more accurate than others and thus be more interesting for this study. In order to evaluate the model performance, smaller subsets will be used, which will allow quicker computation and comparison of the respective performance of all the recommenderlab methods.

The *edx* dataset was split into smaller subsets that will allow a quicker evaluation of the different models. These smaller subsets are then split into a training (90%) and validation (10%) sets.

Training and validation are then performed for each dataset. The computing time of the training, prediction and validation steps is measured. The accuracy of the prediction (RMSE) is also evaluated. It is then possible to create a scatterplot of the RMSE vs compute time of these different models, and plot lines for relevant RMSE levels (RMSE > 0.900 and RMSE < 0.865).

Table 1: Benchmark of the methods on a 0.5% subset

	RMSE	time
RANDOM	1.3773	0.08
POPULAR	0.9554	0.05
LIBMF	0.6425	0.10
SVD	0.9865	0.24
SVDF	0.9158	74.51
ALS	0.5834	49.26
ALS_implicit	2.7570	68.70
UBCF	0.8183	0.23

As one can expect, the random model is among the fastest, but performs quite poorly in terms of accuracy.

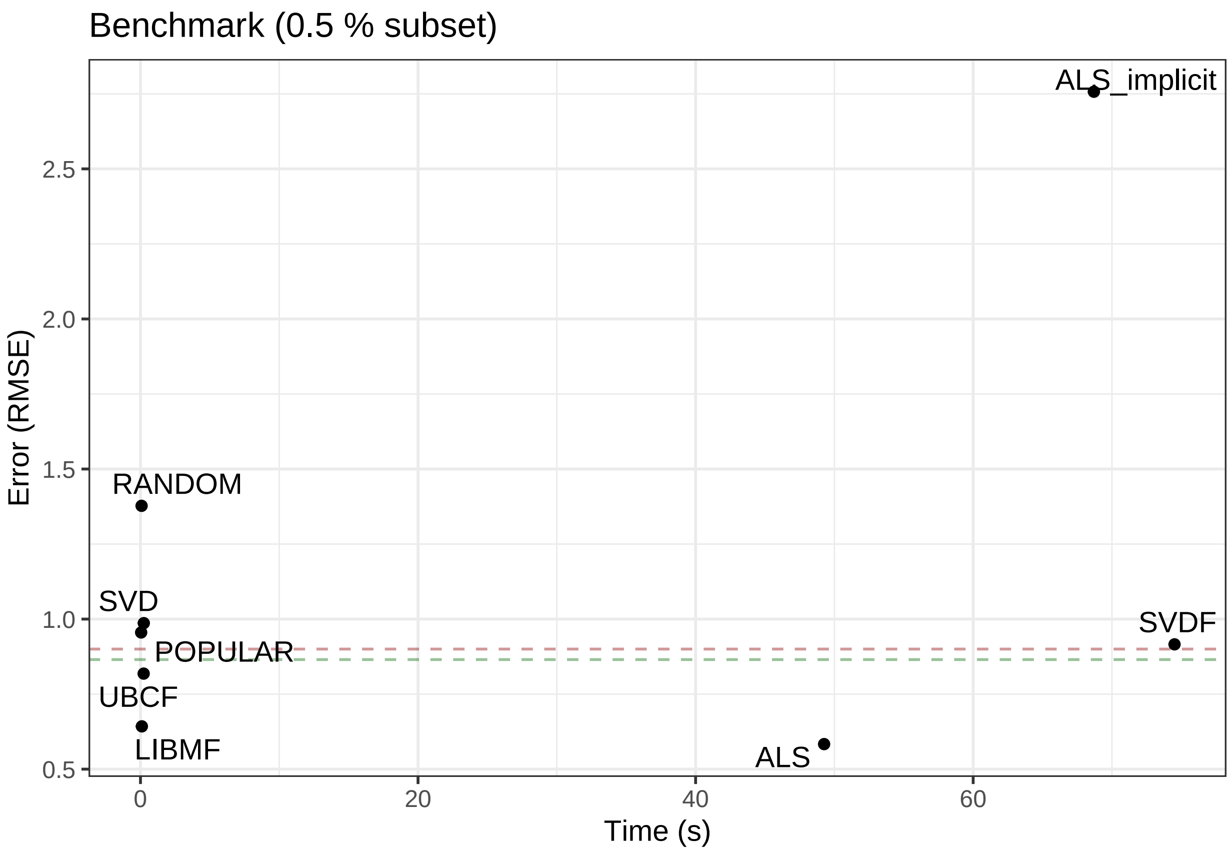


Figure 1: Benchmark of the recommenderlab methods with a 0.5% subset size

The Popular, SVD, UBCF and LIBMF are clear winners, being all in the "fast and accurate" quadrant of the RMSE vs time plot. The accuracy of the SVDF method is comparable with the accuracy of SVD, but SVDF seems to be much slower.

The ALS method seems to be quite slow, but accurate. As expected, a large difference can be seen between explicit and implicit ALS methods. The data are explicit : (known) user ratings are used to predict (unknown) user ratings. Force-feeding these explicit data in an method that was designed around implicit data gives results that are both very slow and inaccurate ($RMSE_{ALSimplicit} > RMSE_{random}$).

As said before, the UBCF method performed quite well. However, the IBCF was so slow it wasn't included in the plot. This can look quite surprising at first. However, the UBCF is based on users, while the IBCF is based on items (movies). In other words, UBCF is row-based, while IBCF is column-based : since the smaller datasets are defined as a smaller matrix row-wise, the UBCF worked on 0.5% of the rows, while the IBCF method had to work on 100% of the columns, hence the poor performance, both on time and accuracy criteria.

The ALS_implicit and IBCF results can look quite surprising at first, but were predictable in the end. This underline the importance of understanding how these methods work, and not just treat them as "black boxes" or "magic bullets."

These preliminary results can be used to exclude the slowest and most inaccurate models (SVDF, ALS, ALS_implicit, random) for the next benchmarks with larger datasets.

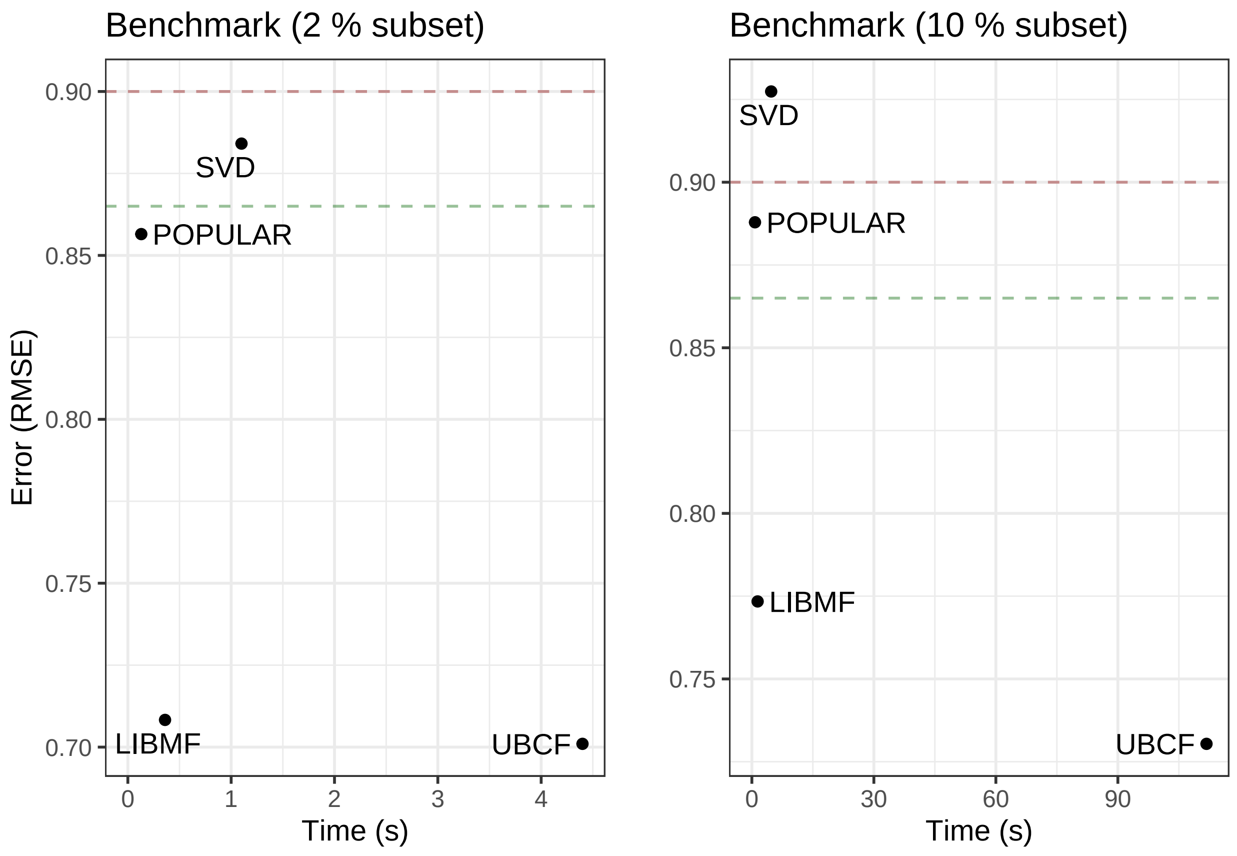
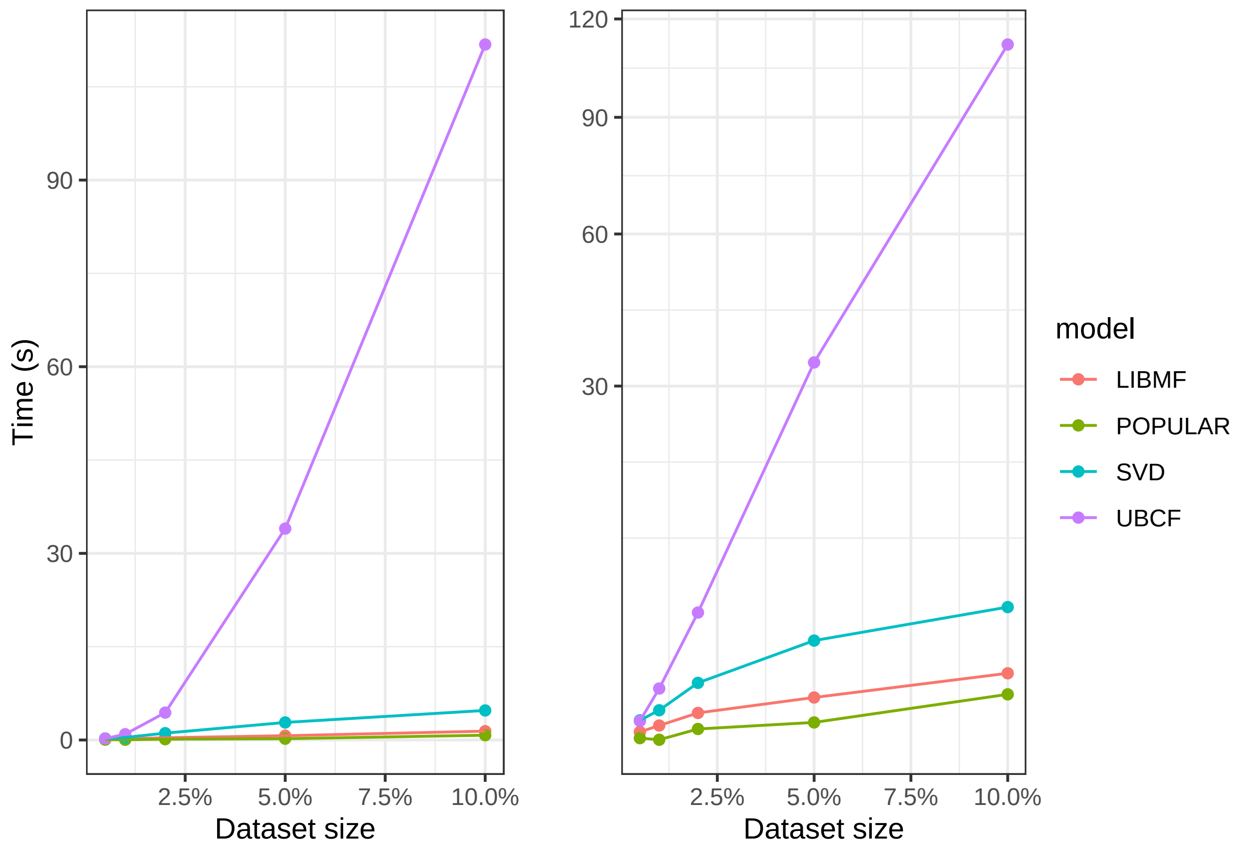


Figure 2: Benchmark of the recommenderlab models with 2% and 10% subset sizes

These larger datasets seem to confirm the respective places of the SVD, UBCF, LIBMF and Popular methods : SVD and Popular are fast but not very accurate, UBCF is quite accurate but slow. LIBMF seems to be the most interesting model, being both very accurate and fast.

However, one should keep in mind that these results – both in time and accuracy – are for very small datasets. It is therefore necessary to carry on and try to predict the performance of these models on larger datasets.



The computing time vs dataset size scatterplot seems to show a lower computing time for SVD, Popular and LIBMF models, while the UBCF model appears to be considerably slower. Furthermore, the computing time of this model seems to exhibit a quadratic behavior (right plot), whereas the other methods seem to be linear.

It is possible to roughly predict the computing time for a full-sized dataset :

Table 2: Modelling of time vs size behavior of the 4 best models

Method	Model	Intercept	Slope	Pred. Time (s)	R^2
Popular	Linear	-0.038	7.50	7.5	0.940
LIBMF	Linear	0.042	13.68	13.7	0.997
SVD	Linear	0.077	48.29	48.4	0.989
UBCF	Quadratic	0.008	107.58	11575.1	0.996

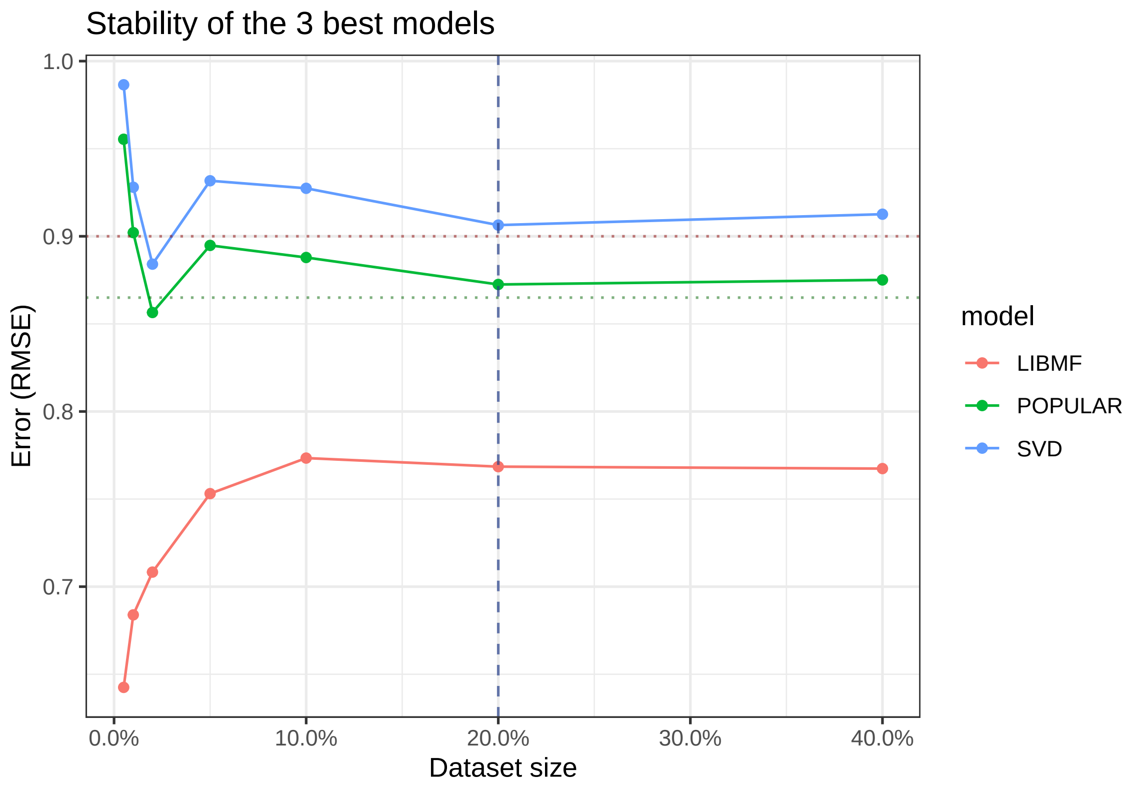
The UBCF model, while quite accurate, will probably be too slow for this study.

The LIBMF, POPULAR and SVD models will thus be selected. Since all the previous results were obtained with default settings, the next step will be to tune them in order to improve their performance.

2.7 Tuning the models

2.7.1 Selecting the best dataset size

In order to tune the models, it is first necessary to chose a relevant-sized training set.



According to the RMSE vs size scatterplot, the 3 selected models seem to show stability for $size \geq 0.20$. Two models already have adequate performance out of the box, with $RMSE_{pop} = 0.8667$ in $t_{pop} = 5.84$ s for the Popular model and $RMSE_{lib} = 0.7575$ in $t_{lib} = 13.91$ s for LIBMF. Before tuning, SVD seems to lag behind, both in RMSE (0.9014) and time (81.13 s) performance.

2.7.2 Tuning the POPULAR model

The POPULAR method only has one parameter : *normalize*, which defines the normalization method used to counter the user biases. This parameter can be set on *center* or *Z-score*. The *center* normalization uses the mean of the ratings, whereas the *Z-score* normalization goes further by dividing by the standard deviation, which allows this normalization method to better handle outliers.

Table 3: Popular model, *normalize* parameter tuning

value	rmse	time
'center'	0.8926	5.31 secs
'Z-score'	0.8935	3.98 secs

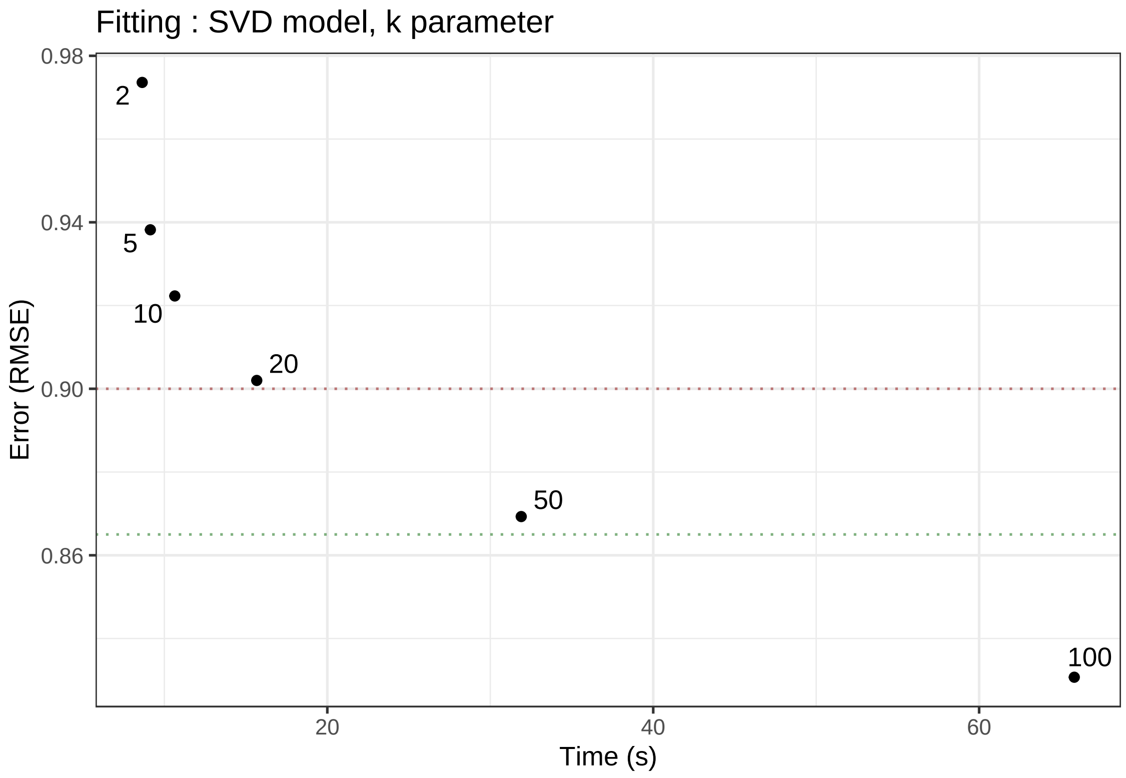
Quite surprisingly, the results show that the Z-score normalization, albeit more refined, produces slightly less accurate predictions than the center normalization, but is sensibly faster.

In conclusion, the Popular method, while quite simple to understand and to use, is not accurate enough (RMSE = 0.8926) for this study.

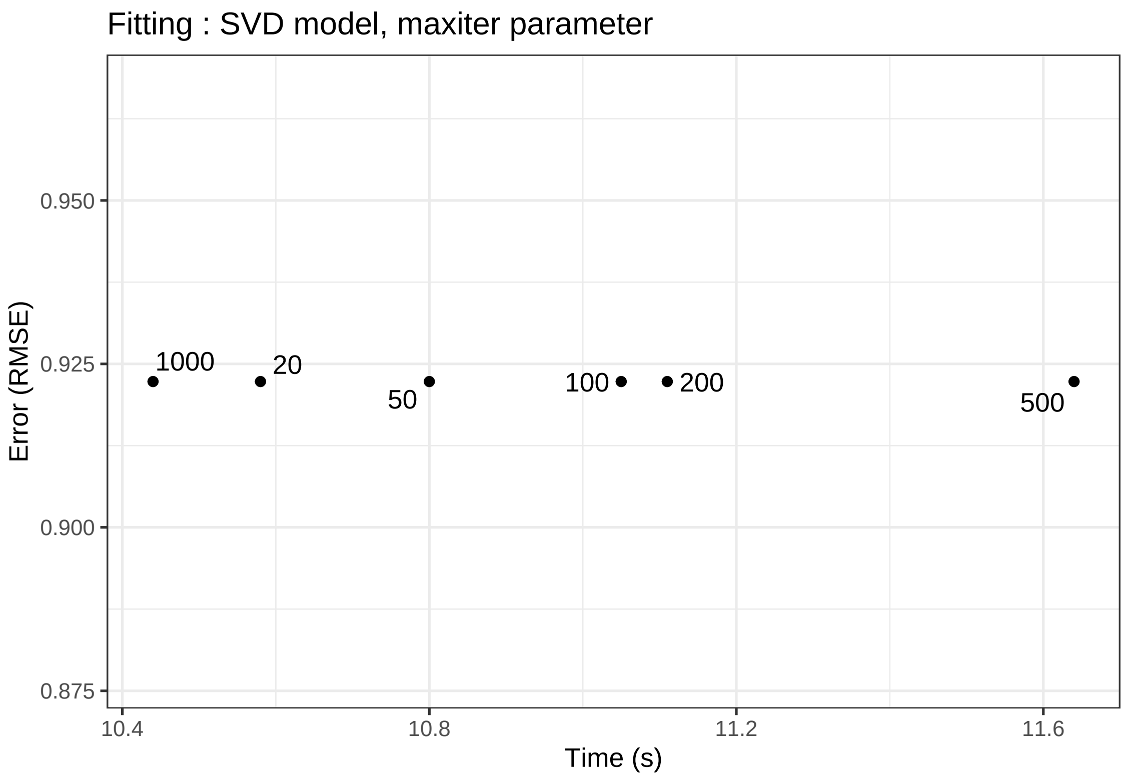
2.7.3 Tuning the SVD model

The SVD method has three parameters :

- k : the rank of the SVD approximation (default : 10)
- *maxiter* : the maximum number of iterations (default : 100)
- *normalize* : the normalization method, *center* or *Z-score* (default : center)



Raising the k parameter vastly improves the accuracy of the prediction. However, this has a considerable cost in computing time. Getting RMSE below the optimal value (RMSE ≤ 0.865) requires to set $k > 50$ (RMSE = 0.8307 in 65.84 s for $k = 100$).



The *maxiter* parameter doesn't seem to have any meaningful impact on this study : all RMSE are identical, and all computing times seem randomly distributed in a 1.2 seconds time span.

Table 4: SVD model, *normalize* parameter tuning

value	rmse	time
'center'	0.9223	10.84 secs
'Z-Score'	0.9250	10.58 secs

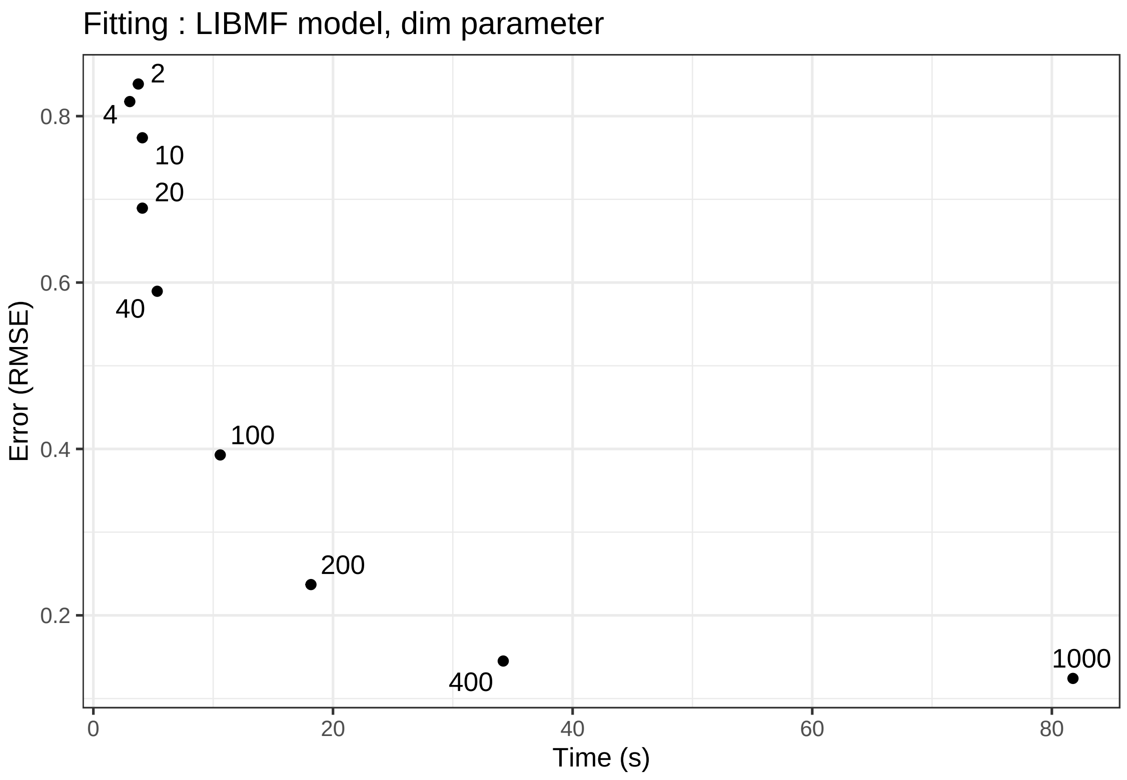
The *normalize* parameter does not seem to have a sensible impact on RMSE or computing time.

In conclusion, the SVD model can be used to reach RMSE ≤ 0.865. Better accuracy (lower RMSE) can be obtained with higher ranks of SVD approximations ($k > 50$), but this accuracy has a strong cost in computing time. The *maxiter* and *normalize* factors don't seem to have any meaningful impact on accuracy or computing time.

2.7.4 Tuning the LIBMF model

The LIBMF method has four parameters :

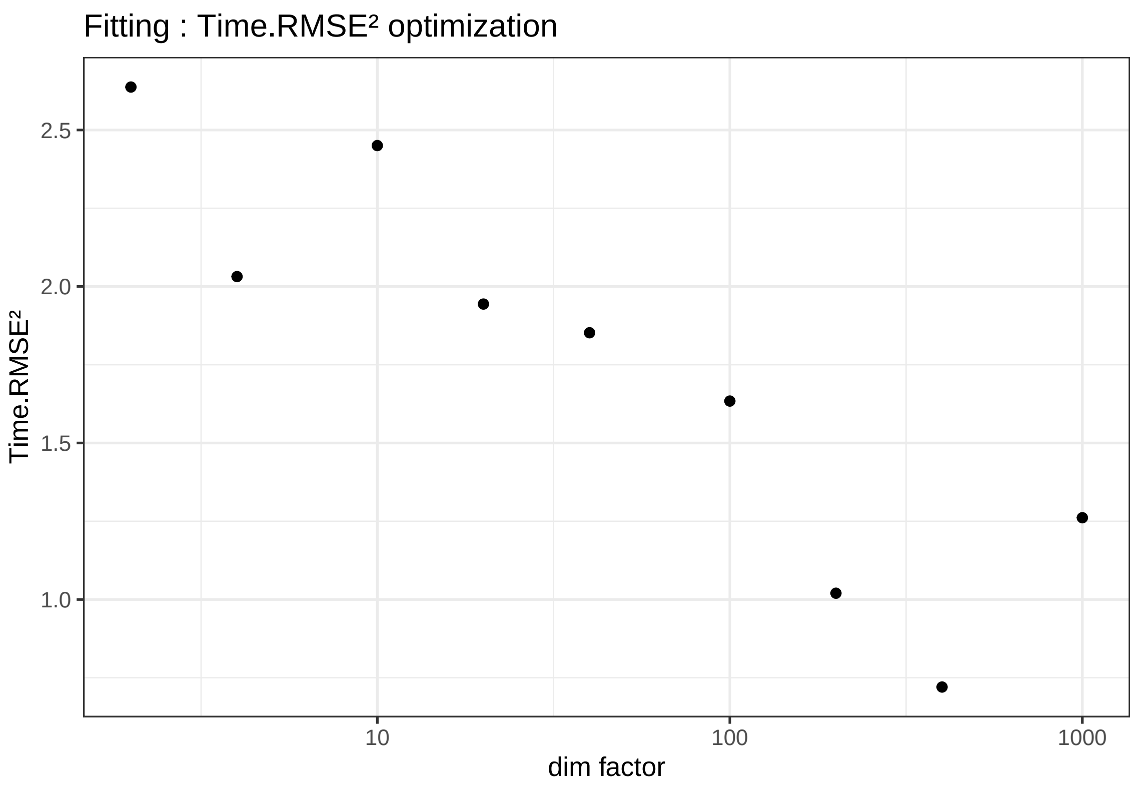
- *dim* : the number of latent features (default : 10)
- *costp_l2* : the regularization parameter for the user factor (default : 0.01)
- *costq_l2* : the regularization parameter for the item factor (default : 0.01)
- *nthread* : the number of threads (default : 1)



The *dim* parameter has a major effect on accuracy and computing time. Higher values (*dim* = 1000) yield a better accuracy, with RMSE = 0.1242, at the cost of a higher computing time (t = 81.76 s).

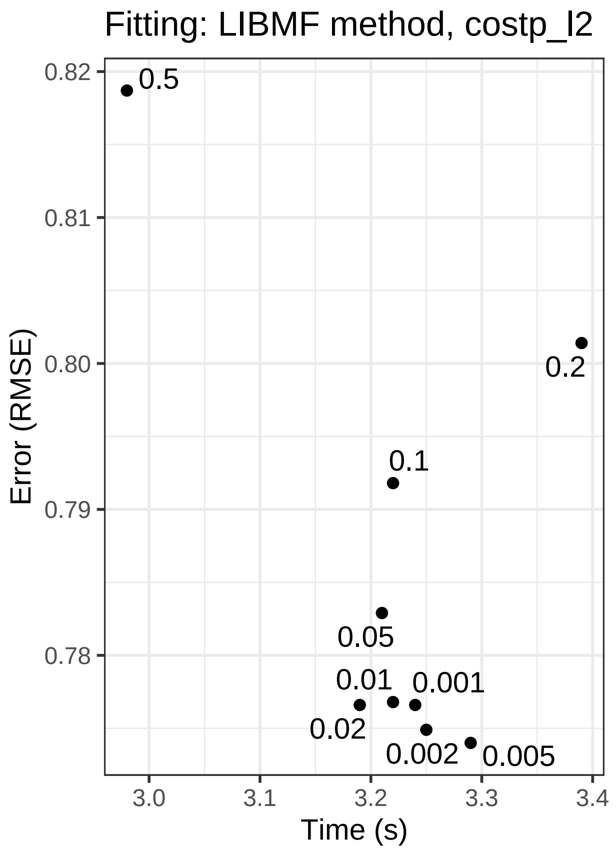
Very high or very low *dim* values seem to show diminishing returns in time or accuracy. It is thus possible to design time/RMSE optimization strategies by computing a composite indicator (such as $time \times RMSE^n$, with n a fixed weight factor), then finding its minimum.

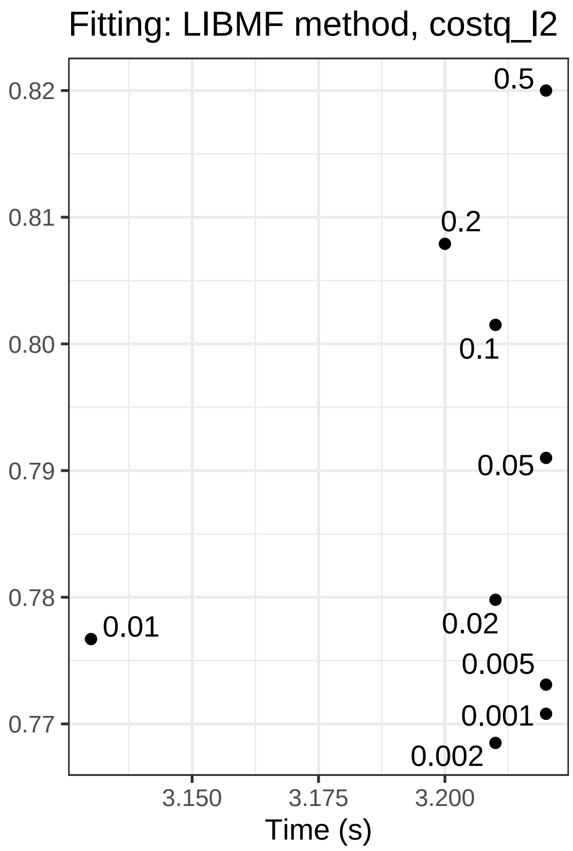
One can for example choose to put a 2:1 weight on RMSE vs time ($n = 2$), to reflect the higher priority given to RMSE – which is the primary goal of this project – while also taking computing time into account.



With this (arbitrary) 2:1 ratio given on the RMSE/time composite factor ($n = 2$), an optimum seems to be reached for a dimension value of $dim \approx 400$.

That dim value gives RMSE = 0.1451 in 34.21 seconds.

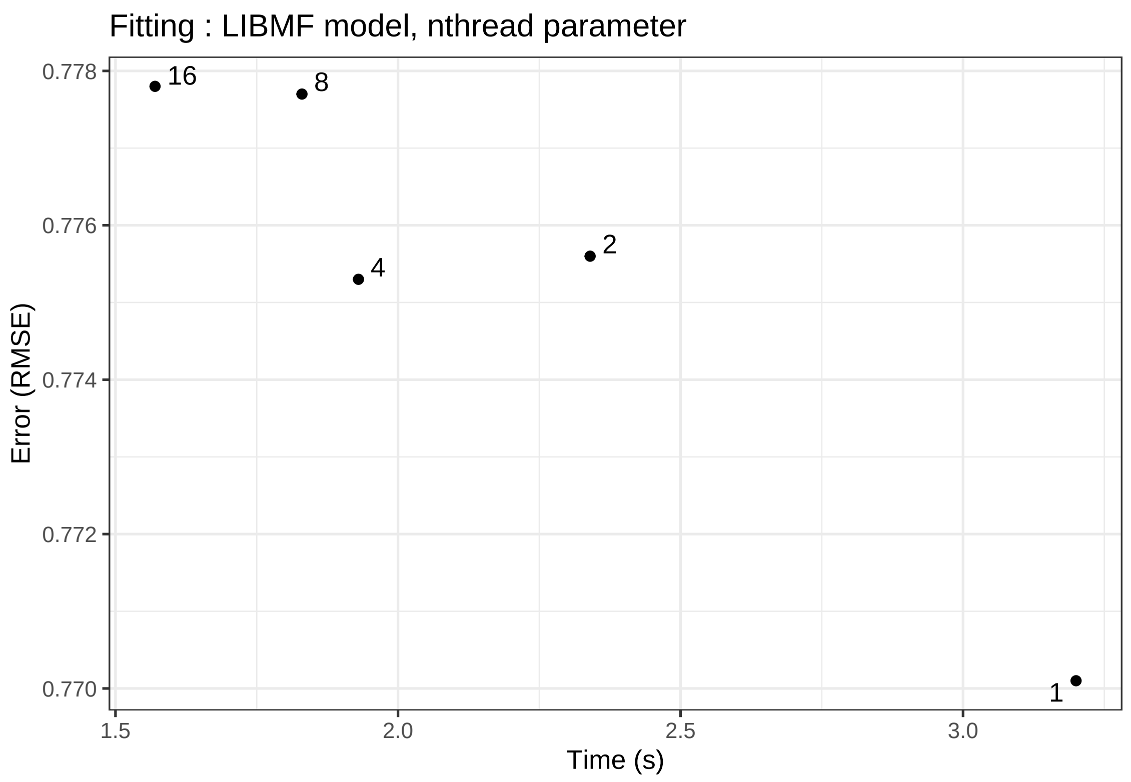




The user (*costp_l2*) and item (*costq_l2*) regularization factors seem to have an important effect on the accuracy and a minor effect on computing time.

The accuracy seems to considerably decrease for higher values of *costp_l2* and *costq_l2*. However, for values below 0.01 (default value), the potential accuracy gains seem to be quite marginal.

Apart from some outliers, most of the *costp_l2* and *costq_l2* values seem to have little effect on computing time, with 95% confidence intervals widths of 0.14 seconds for *costp_l2* and 0.04 seconds for *costq_l2*.



The number of threads (*nthread*) seem to have some meaningful impact in computing time. The RMSE varies very slightly from one *ntrehad* setting to another, but the difference is very small ($\Delta_{RMSE} \approx 0.008$) This is an interesting result that shows that the LIMBF multithreading works out of the box, without having to install additional libraries or set the parallelization manually.

However, the improvement in computing time can seem disappointing, since the impact is pretty minor, especially for $nthread \geq 4$.

This can probably be explained by the small size of the sample used for fitting purposes and the resulting short computation time : the gains obtained by multithreading thus cannot overcome the time and memory used to setup the parallelization. This phenomenon is usually known as *parallelism overhead.*

A more significant performance boost with higher values of *nthread* can probably be expected on larger datasets.

In conclusion, LIBMF seems to be, by far, the most interesting model, with both excellent accuracy and computing times, even with default parameters.

The most impactful way to improve accuracy of the LIMBF model seems to increase the number of latent features (*dim*). However, this factor has a strong impact on computing time. The results are perfectly consistent with those obtained by the creators of the LIBMF library in their initial tests. The excellent performance of the LIBMF model allows the user to choose between good accuracy and excellent computing time, or outstanding accuracy in a reasonable time.

The user factor (*costp_l2*) and item factor (*costq_l2*) regularization parameters seem to have some minor impact, with lower values giving slightly better accuracy, with little effect in computing time. However, since the LIMBF model uses these factors to penalize potential overfitting, it may be preferable to keep the default values.

The number of threads has some impact in computing time, but this impact looks minor, and the expected gains on this small subset are unfortunately counterweighted by the parallelism overhead.

3 Results

3.1 Evaluation against the *validation* dataset

Given the previous benchmarks, the following parameters were chosen for the final evaluation, against the *validation* dataset :

- Method : LIMBF
- Number of latent features (*dim*) : 400
- Regularization parameter, user factor (*costp_l2*) : 0.01
- Regularization parameter, item factor (*costq_l2*) : 0.01
- Number of threads (*nthread*) : 16

With these settings, this model managed to obtain RMSE = 0.0612 in 308.49 s.

As suggested by the previous selection and tuning processes, the LIBMF model provides excellent accuracy in a very reasonable amount of time.

4 Conclusion

The aim of this study was to build a recommendation system based on the recommenderlab package and use it to predict user preferences on the Movielens dataset. After splitting the data into a training and validation sets, some data preparation of the training set was performed, notably by converting the sparse dataset into a suitable matrix format. Training and evaluation of nine recommendation algorithms were then performed, using cross-validation on a smaller subset of the original training set. The three best algorithms were then fine-tuned, and the best one (LIBMF) was used against the validation set.

The LIBMF method proved to give outstanding performance, with excellent accuracy (RMSE = 0.0612) in a reasonable amount of time (t = 308.49 s).

One of the limitations of this study is the inability to test the code on any MacOS system, although it was run and tested on two different systems (R 4.1.0 on Windows, R 3.6.3 on Linux).

Furthermore, despite being oriented toward both accuracy and performance in terms of computing time of the recommendation algorithm, there is still a considerable room for improvement in the performance field of the selection and tuning processes themselves. Future work will be dedicated to parallelization and further code optimization.

5 References

1. F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets: History and Context. 2016; Available from: http://dx.doi.org/10.1145/2827872
2. Wickham H. Tidyverse: Easily install and load the tidyverse [Internet]. 2021. Available from: https://CRAN.R-project.org/package=tidyverse
3. Kuhn M. Caret: Classification and regression training [Internet]. 2021. Available from: https://github.com/topepo/caret/
4. Dowle M, Srinivasan A. Data.table: Extension of ‘data.frame‘ [Internet]. 2021. Available from: https://CRAN.R-project.org/package=data.table
5. Wickham H. reshape2: Flexibly reshape data: A reboot of the reshape package [Internet]. 2020. Available from: https://github.com/hadley/reshape
6. Slowikowski K. Ggrepel: Automatically position non-overlapping text labels with ggplot2 [Internet]. 2021. Available from: https://github.com/slowkow/ggrepel
7. Hahsler M. Recommenderlab: Lab for developing and testing recommender algorithms [Internet]. 2021. Available from: https://github.com/mhahsler/recommenderlab
8. Kassambara A. Ggpubr: ggplot2 based publication ready plots [Internet]. 2020. Available from: https://rpkgs.datanovia.com/ggpubr/
9. Allaire J, Xie Y, McPherson J, Luraschi J, Ushey K, Atkins A, et al. Rmarkdown: Dynamic documents for r [Internet]. 2022. Available from: https://CRAN.R-project.org/package=rmarkdown
10. Xie Y. Knitr: A general-purpose package for dynamic report generation in r [Internet]. 2021. Available from: https://yihui.org/knitr/