Data Science Lab Project 1 (IASD 2024-2025) : Collaborative Filtering

Alexandre Olech Lucas Henneçon Matthieu Neau

1. Summary of implementations

Here is a list of what we implemented for the DSLab first assignment. We kept track of all experiments with the help of the library MLflow.

- A matrix factorization model with gradient descent, with an inverse time decay schedule, built from scratch with Numpy.
- A replication of this implementation, using PyTorch for automatic differentiation.
- A method to build cross-validation splits in the context of matrix factorization, using an iterative sampling algorithm. We did not use it in our experiments as it led to unsatisfying results.
- Experiments on the importance and on the tuning of the initialization in matrix factorization.
- Experiments on the behavior and tuning of the latent dimension in matrix factorization.
- Experiments on the behavior and tuning of the regularization coefficients in matrix factorization.
- Implementation and experiments on a custom rounding method, based on user ratings behavior. Tuning of the best threshold for this method.
- Implementation of a parallel content-based method that predicts, for each user, the ratings of movies, based on one-hot-encoded movie features. This model was treated as a black box, its main purpose was to see if our collaborative filtering model could be enriched when combined with a content-based model. This proved to be the case.
- Our tuned matrix factorization model achieved 0.866 accuracy on the testing platform of the DSlab without rounding predictions, and it achieved 0.908 RMSE and 32.91 accuracy with our custom rounding method. Our ensemble model based on both content-based and collaborative filtering models, largely outperformed all our models on the validation set, even though it has not yet been tested on the DSlab platform.

2. Experiments

2.1. Collaborative filtering with matrix factorization

Our first experiments consisted in building a matrix factorization model for collaborative filtering (described by the cost function in equation 1), using gradient descent as optimization algorithm (the updates are described in equations 2 and 3). We wanted to start with a basic, yet highly customizable model. For this reason, we first implemented the model from scratch, using only Numpy and no external libraries. This first implementation was successful as the model was able to effectively learn, with performances better than the average of the DSLab, but the model was taking more than 20 seconds to train. Thus, we rewrote the code using PyTorch, which we used only for faster gradient computation, as we kept using our custom gradient descent algorithm. This also enabled us to reproduce our first results (results were identical for the PyTorch and Numpy implementation) and thus gain confidence in the validity of our computations.

$$C(U,V) = \|R - UV^T\|^2 + \lambda\|U\|^2 + \mu\|V\|^2 \tag{1}$$

$$U_{t+1} = U_t - \eta_t \frac{dC}{dU}(U_t, V_t) \tag{2}$$

$$V_{t+1} = V_t - \eta_t \frac{dC}{dV}(U_t, V_t) \tag{3}$$

2.2. Enabling Learning

When we built our gradient descent algorithm, we started with a simple implementation, with a fixed learning rate. We experimented with the values of the learning rate, and found values that made learning possible. But the learning curves were not very smooth: the training error as a function of the iterations was making bumps. Thus we implemented a more elaborate update formula, with a decay rate. The method we chose is known as the inverse time decay schedule, which is described in equation 4, where η_t is the learning rate at time t and γ_t is the decay rate.

$$\eta_t = \frac{\eta_{init}}{1 + \gamma t} \tag{4}$$

This decay schedule can be seen as a variant of the original Robbins-Monro schedule, described in equation 5. This modification allows for more flexible control over the decay rate through the parameter. With this learning rate update, the goal is that the algorithm starts with large steps to make rapid initial progress, and then gradually takes lower steps to fine-tune the solution and avoid overshooting the minima.

$$\eta_t = \frac{\eta_{init}}{t} \tag{5}$$

When experimenting with the learning and decay rates, we found that, with a sufficiently low value of the initial learning rate, having a value of the decay rate equal to the initial learning rate led to satisfying learning scenarios. In practice, we found $0.5 * 10^-3$

to be a good values for both coefficients. This led to satisfying smooth learning curves, and we choose a number of trials equal to 35 to not over-fit the training set.

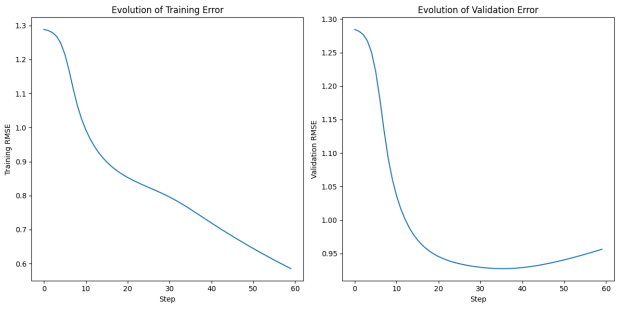

Figure 1. Learning curves of the unconstrained matrix factorization model with gradient descent. Left plot shows the evolution of training error as a function of the number of gradient descent iterates. Right plot shows the evolution of validation error as a function of the number of gradient iterates. Both plots use root mean squared error as metric.

2.2.1. First results on the test set

We then tested this model on the DSlab platform, without hyper-parameter tuning, with a latent dimension chosen arbitrarily as equal to 10 and no regularization. The results are displayed in table 1, under the row of the "UMF" model, which stands for "Unregularised Matrix Factorization". This model was able to achieve 0.906 RMSE and 24.58 accuracy, which was significantly better than performances on the validation set (by almost 0.04 points for the RMSE).

2.3. Building cross-validation splits with iterative sampling.

To improve our model, our goal was to do hyper-parameter tuning. To do this optimally, we aggregated the original train and test matrices that were given to us in the project. Then, we wanted to split the full matrix into five different training-validation splits, in order to perform cross-validation.

One difficulty is that, our machine learning problem was not to generalize to new users and movies, but rather to generalize to new interactions (ratings) for a given set of movie and users. In this setting, it seemed to us that an optimal cross-validation setting would be such that each user and each movie of the original rating matrix still have at least one interaction in every training and validation split (in other terms, the matrices built with the splitting method have the same number of columns and rows, and they correspond to the same users and movies).

To achieve this, we used an iterative splitting method to make sure that while splitting, we avoid the situation where there is a row-column cross with no interaction on the matrices built from the splitting.

Sadly, we were not able to do this without removing some movies or users with low interactions from the original full ratings matrix. Moreover, the validation performance when training the model on these splits was significantly lower than on the original splits given to us, which made us think that we were changing too much the properties of the matrices, hence creating matrices that were maybe too far from the matrices on which we want to predict ratings (for example the matrix in the DSlab platform), hence reducing the relevance of the method.

For these reasons, we decided to stick with the simple setup and use the "train" matrix for training and the "test" matrix for validation (which we called "validation" because it cannot be considered as test anymore once we do hyper-parameter tuning on it).

2.4. Assessing the importance of the distribution of initial embeddings

Once our model was able to learn effectively, we made the observation that the overall performance was sensitive to the initialization of the matrices U and V, as mentioned by Aggarwal [1]. Indeed, this is due to the fact that our cost function is non-convex in U and V, so it can have multiple local minima and the final solution can depend on where the optimization starts. We then conducted a review of various articles on initialization techniques and identified several methods for effective initialization in matrix factorization models.

As we were working in parallel, we already had an idea of the optimal latent dimension value. We then chose the value of K=101 to run our tests (we'll describe later how we found this value), and we used the custom rounding described later. To have more robust results, we chose not to fix the seed and to run several runs (5) for each type of initialization.

We first tried to initialize U and V with a small constant value of 1/100. Then we tried to initialize the matrices with probability distributions to inject diversity into the initial state of the optimization process. We used a normal distribution $\mathcal{N}(0, 1)$ where we divided each coefficient by $n_{users} + k$, as advised in [3] to normalize our matrices. We also initialized the matrices with a uniform distribution $\mathcal{U}(0, \frac{1}{100})$.

Then, we tested some techniques which use the coefficients of the ranking matrix R to initialise U and I. The first one is called randomA and is presented in [3]. In this method, we select randomly p columns of R, and compute the mean on the lines to form a column of U. We repeat this process to form the K(K is the latent dimension) columns of U. We do the same for V by selecting randomly p lines of R and computing the mean on the columns. We ran some tests to find the best parameter p, which was $p = 750$. We then tried a variant of this method called randomC which chooses p columns randomly among the longest (in norm 2) columns of R. The idea behind these two methods (randomA and randomC) is to initialize the matrices with some information of the given data and to obtain sparse matrices, which makes more sense and may be closer to the final solutions than dense matrices generated using probability distributions. We can see on figure 4 the RMSE of our different approaches and on figure 5 the accuracy.

We remark on the graphs that the best tradeoff between accuracy and RMSE is obtained with the randomA initialization, so this is the method we have been using.

2.5. Model complexity

In such Matrix Factorization models, the complexity of the model is determined by the chosen latent dimension (which is the number of columns of the matrices U and V), and the magnitude of the regularization coefficients. Increasing the latent dimension increases

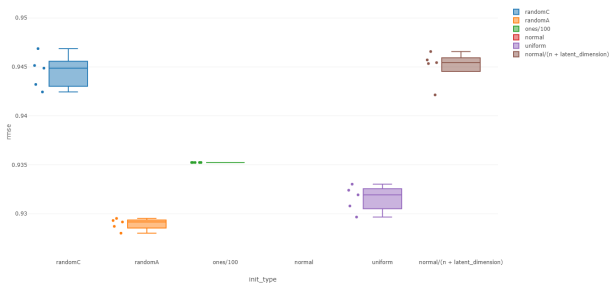

Figure 2. Root mean squared error associated to different initialization strategies (zoomed).

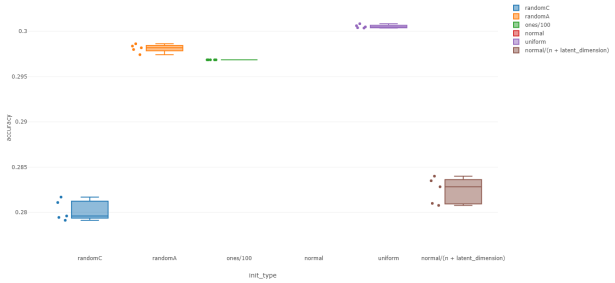

Figure 3. Accuracy associated to different initialization strategies (zoomed).

the space of potential parameters values the model can take, thus it increases model complexity. On the contrary, increasing the regularization coefficients constrains the matrices to have a small norm, hence reducing the space of possible parameters.

Since the latent dimension and the regularization coefficients both affect the same thing (model complexity), they should ideally be considered as interrelated, and each should be tuned as a function of the other.

We came to this thinking when we observed how hard it is to naively find good values of the regularization coefficients and the latent dimension, for example with a grid search, for example because the scale of the value of good regularization coefficients changes for each value of the latent dimension, since the norm of matrices increase when their size increase. Our first attempts at performing a grid search on the 2 types of hyper-parameters at the same time proved unfruitful, and we think this interpretation may explain why this happens.

For unclear reasons, which might have something to do with the randomness of the initial weights, it seemed like the best values of both regularization and latent dimension were very low values (but this later proved to be false with a more elaborated experimental setup). This observation has also been relayed by other teams of the DSlab. It might be due to the fact that we tackle model complexity with two levers at the same time.

Unsatisfied with this first attempt, we changed our approach and started to tackle model complexity with one hyper-parameter at a time. We started by finding the best latent dimension, without regularization, and then we added some amount of regularization that proved to be beneficial. With this approach, we may have not found the optimal latent dimension - regularization couple, but we managed to significantly reduce over-fitting, using the two types of hyper-parameters, which is already satisfying.

2.5.1. Effect of the latent dimension

We thus started searching for the best value of the latent dimension, without regularization (that is, $\lambda = 0$ and $\mu = 0$). In our experiments, we chose to not use a seed nor a fixed value for the initialization of matrices U and V. While this is unusual for gradient descent, we did it because we thought that it would reduce our chances of over-fitting the predictions at some point, and that it would enable us to not be too dependent on a specific initialization when searching for the best hyper-parameters. For this reason, in our experimental setup, we often performed multiple runs for each combination of hyper-parameter values.

As expected in typical learning scenarios, we found the training error to be a decreasing function of the latent dimension. On average, across samples, the validation error starts by decreasing as the latent dimension increases, and then starts increasing (figure 4). This is coherent with the principles of learning theory (as shown in the second chapter of Bach [2]). We found out that the values of the latent dimension that lead to the lowest error on the validation set were found around 100. Looking closer, we found that the value leading to the best results on the validation set was K = 101.

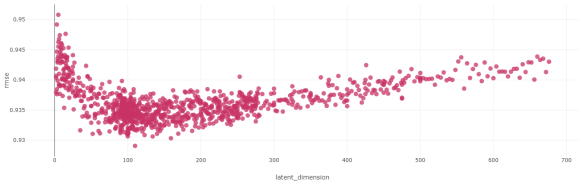

Figure 4. Evolution of the validation error (RMSE) versus the latent dimension. Each point corresponds to a run with the given value of the latent dimension. Multiple samples were taken into consideration, in order to account for the randomness of the initialization.

2.5.2. Regularising with identical coefficients

Then, we wanted to investigate if some regularization was beneficial for this value of K = 101. It did not seem obvious that regularization was necessarily beneficial at this point, since we had already adjusted complexity to maximize generalization performance.

We started from a simplified setting, where we constrained the two regularization coefficients to be equal. We then observed that the validation error was a decreasing at first and then increasing function of lambda (figure 5). We found that the best values of regularization coefficients should lie around 5.5.

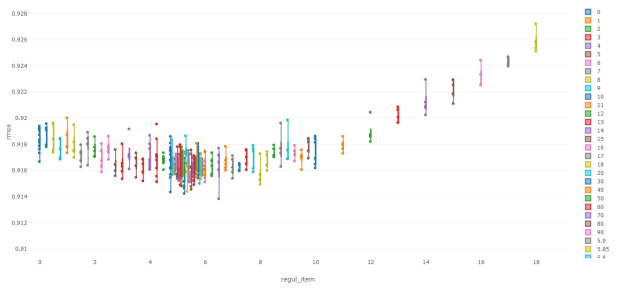

Figure 5. Evolution of the validation error (RMSE) versus the size of the regularization coefficients, in the simplified setting where both coefficients are equal, with high zoom and box plots.

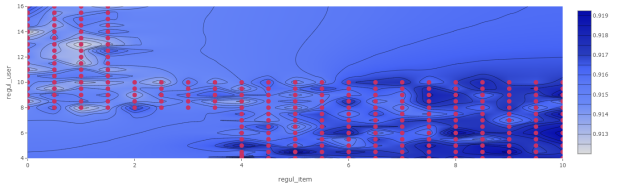

Figure 6. Evolution of the training error (indicated by color intensity) versus both regularization coefficients with values that are allowed to differ.

2.5.3. Regularising with differing coefficients

Then we asked ourselves : would it be beneficial to have different regularization values for the items and for the users ? We did some research on the subject and found at that, for example, Aggarwal [1], in his book about recommender systems, says that using different values for the two usually leads to better results.

Thus, we performed a computationally expensive grid search, in order to study look at the average performance of the model, for different couple of regularization coefficients, around the value of 5.5 that was found in the simplified setting, with at least 5 samples by coefficients combination.

Our findings seemed to indicate that, around the chosen value, it is indeed beneficial to have a significant imbalance between the user regularization coefficient and the item regularization coefficient (figure 6), especially with a higher user regularization coefficient and a lower item regularization coefficient.

This observation led us to the following hypothesis : because the number of movies (4980) is largely superior to the number of users (610), it is possible that the amount of over-fitting with respect to the user embeddings (represented by the first matrix) is much higher than the amount of over-fitting with respect to the item embeddings (represented by the second matrix). This would be due to the fact that it is easier to represent the embeddings of 610 users with a dimension of 101 than it is to represent the embeddings of 4980 movies with a dimension of 101.

One difficulty is that it can be expensive to evaluate all the possibilities of coefficients, as the function (error associated to lambda and mu) might be complex. We did some research on the subject and found out that, in their article on Bayesian probabilistic matrix factorization using Markov chain Monte Carlo, Salakhutdinov et al. [5] propose a Gibbs sampling method to jointly learn both hyper-parameters with a probabilistic approach.

Algorithm 1 Preference-based Rounding

Input: initial ratings matrix R of size (n,d), predictions matrix $\hat{R}$ of size(n,d), threshold τ
Initialize p as a vector with n elements
for $i = 1$ to n **do**
  $P[i] \leftarrow$ Proportion of half-ratings for row i of R
end for
for $i = 1$ to n **do**
  if $P[i] > \tau$ **then**
    round values of $\hat{R}[i]$ to half-ratings
  else
    round values of $\hat{R}[i]$ to integer values
  end if
end for

2.6. Custom Rounding method based on user rating behavior

Aside from building and tuning our initial model, we made some observations that lead us to believe the problem would benefit from integrating more diverse approaches and types of information.

In our exploratory analysis of the data, we found out that the users have a wide variety of rating behavior. In particular, some users tend to never use half-ratings (0.5, 1.5, 2.5, 3.5, 4.5) while other users use them as frequently as other ratings. We used this fact to better reproduce the rating behavior of the users. We added a rounding type parameter in our model, so that different rounding methods are applied, depending on the general tendency of the user to use half-ratings. Our decision method is described in algorithm 1. The main idea is to define a threshold which is a real number between 0 and 1, and to compute, for each user, the average proportion of half-ratings. Then, for each user, if the proportion is superior to the threshold, half-rating rounding is performed, and integer rounding is performed otherwise. We optimised this threshold on the validation set, and found an increasing relation between this threshold and accuracy, from 0 to 0.50 (figure 7). We chose a value of 0.48 as it led to the best accuracy on average.

2.7. New results on the DSlab platform

With the mentioned tuning of matrix initial values, latent dimension and regularization coefficients, we tested the new model on the platform (see table 1 for all results) and obtained a RMSE of 0.866. In another attempt, we added our custom preference-based rounding, and we reached an accuracy of 32.91, which to our knowledge is the best accuracy ever achieved in the DSlab so far, for this project. However, it is important to note that this run lead to a RMSE of 0.908, which indicates that our rounding method, in its current state, has a non-negligible cost of approximately 0.04 in RMSE.

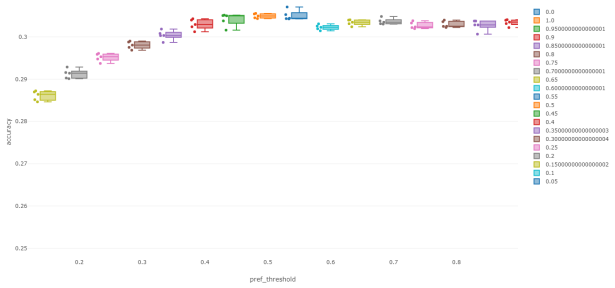

Figure 7. Accuracy associated to different preference thresholds, in the context of our custom rounding method, based on user rating behavior.

2.8. Building a content-based model with movie features

Aside from building and tuning our initial model, we made some observations that lead us to believe the problem would benefit from integrating more diverse approaches and types of information. Taking user ratings behavior into account in our rounding method was a first step. Another step was to take the movie features into account. We thought that it would be interesting because these features provide a very different type of information compared to the ratings, and we wanted to combine the two types of information to get a richer model.

Thus we started to build a content-based model with the movie features. We first created a simple model, where, for each user, a linear regression is trained, using one-hot-encoded movie features as features and ratings as labels. For each user, we then predict the missing rating with the trained model. Since the model of each user is independent, we were able to use parallel processing, so that it only takes 10 seconds to train the models and get all the predictions.

With this simple linear regression, we were able to reach 1.092 RMSE on our validation set (see table 1). We then replaced linear regression by gradient boosting, and we achieved 0.96 RMSE, which beats the baseline of our linear regression model. In our approach, we treated this gradient boosting as a black box model and for this reason we do not present the equations behind it. In this project, we were not directly interested in classical machine learning models, but rather in understanding the properties of collaborative filtering models, and the only reason that made us experiment with content-based models was to evaluate how valuable it is to combine a collaborative-filtering method with a content-based method. For this reason, the gradient boosting model was treated as a black box model.

We thus selected gradient boosting for our model, and we tuned the number of leaves, the maximum depth and the learning rate, without significant changes in RMSE (default values achieved comparable performance).

2.9. Combining the strengths of content-based and collaborative-filtering methods

We then built an aggregation of our optimised collaborative filtering model (unconstrained matrix factorization) and content-based model (gradient boosting), and we computed the final predictions as a weighted sum of both predictions. This weighted sum is described in equation 6, where $\hat{R}$ are predicted ratings, and CF and CB refer to our content-based and collaborative-filtering models, that is, regularised matrix factorization and feature-based parallel gradient boosting.

$$\hat{R} = w\hat{R}_{CF} + (1-w)\hat{R}_{CB} \tag{6}$$

Then, we searched for the best value of the weight, and we found our best values between 0.36 and 0.45 (see figure 8), without significant change in the range [0.36, 0.45]. We thus chose a weight of 0.4, which means that our predictions are determined by movie features at a degree of 40% and by collaborative filtering at a degree of 60%.

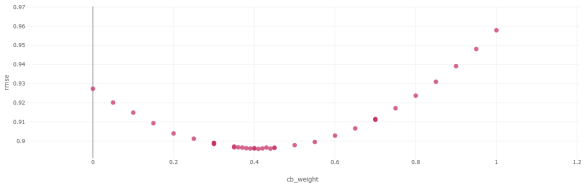

Figure 8. Validation error (RMSE) as a function of the relative weight of the content-based model (gradient boosting with movie categories as features) versus the weight of the collaborative filtering model (regularized unconstrained matrix factorization).

Interestingly, this gives an RMSE of 0.89, while the performances of our CF and CB models are respectively 0.93 and 0.96 on average in this setting. The combination of the two models leads to better predictions than each model's individual predictions, even though one model is better than the other in terms of performance. This improved score shows the strength of combining different types of information into a predictive algorithm.

2.10. General performance comparison

We have not yet tested the content-based model and the hybrid approaches on the DSlab platform, but judging from performances on the validation set, our hybrid model with matrix factorization and feature-based parallel gradient boosting seems to be the most promising in terms of RMSE performance, while the preference-based rounding version of this same model seems to be the most promising in terms of accuracy.

While we did not compare our different models in terms of time performance, we observed that all the models we tested on the DSlab platform so far run in less than 6 seconds.

While simple in their design, the models we have built have proved to be very efficient in terms of time and highly customizable. They achieve the best accuracy, and they seem to reach levels of RMSE on par with deep learning implementations on the DSlab.

Of course, the comparison is biased, because deep learning models need more tuning, more training time, and more data to achieve their best performance. And in real life scenario, they are likely to be superior to such simple implementations.

Table 1. Performance comparison of the different methods implemented. The validation ratings are the ratings that have been used to tune hyper-parameters. The test ratings are the hidden ratings on the DS Lab platform. UMF corresponds to unregularized matrix factorization, with basic half-ratings rounding and without hyper-parameter tuning. MF+ corresponds to matrix factorization with differing regularization coefficients and hyper-parameter tuning, without rounding. PB-MF+ corresponds to the same model as MF+, but with custom preference-based rounding based on user rating behavior. GB corresponds to the parallel gradient boosting models that predict user ratings based on movie features. MF+GB is the model obtained by weighting the predictions of MF+ and GB, with a weight of 0.4 for GB and 0.6 for MF+. PB-MF+ GB corresponds to the same model as MF+GB, but with preference-based rounding at the end.

MODEL	VAL. RMSE	VAL. ACC.	TEST RMSE	TEST ACC.
UMF	0.943	23.15	0.906	24.58
MF+	0.913	0.00	0.866	0.19
PB-MF+	0.934	30.01	0.908	32.91
LINREG	1.092	20.65	-	-
GB	0.960	21.74	-	-
MF+GB	0.896	0.00	-	-
PB-MF+GB	0.937	30.55	-	-

3. Key takeaways

Here is a list of things we have learned from our experiments:

- In some cases, like collaborative filtering, implementing a model from scratch without complex frameworks ensures good understanding and control over the multiple aspects of a model.
- In our learning rate method with decay, using a decay rate equal to the initial learning rate often leads to satisfying learning scenarios.
- Building cross-validation splits for a fixed set of movies and users can be challenging and is not always possible in the context of collaborative filtering.
- The distribution of the initial values has a significant influence on performance in the context of collaborative filtering with gradient descent.
- The best latent dimension and the best regularization coefficients follow an interrelated relation, as they both control model complexity. It is highly valuable to tune both in this type of problems.
- In the context of matrix factorization, model complexity can be thought of as comprising both the complexity relatively to the user representation and to the item representation. It is highly valuable to use differing values for the regularization coefficients, to tackle the two different types of complexity separately, which cannot be done just by tuning the latent dimension alone.
- In the context of ratings prediction, considering the rating behavior of the users in the pre-processing step leads to significant increases in accuracy.
- Combining the predictions of collaborative filtering and content-based methods can lead to a model that significantly outperforms each model individually.
- With short available time (three weeks in our case), simple, yet highly tuned, customized and diverse models achieve comparable performance to neural network methods (with faster running times, higher accuracy and close RMSE in our case).

4. References

[1] C. C. Aggarwal, Recommender Systems: The Textbook. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-29659-3

[2] F. Bach, Learning Theory from First Principles. MIT Press, 2024.

[3] A. N. Langville, "Algorithms for the Nonnegative Matrix Factorization in Text Mining," Slides from SAS Meeting, 2005.

[4] A. N. Langville, C. D. Meyer, R. Albright, et al., "Initializations for the Nonnegative Matrix Factorization," in Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Citeseer, 2006.

[5] R. Salakhutdinov and A. Mnih, "Bayesian Probabilistic Matrix Factorization Using Markov Chain Monte Carlo," in International Conference on Machine Learning, pp. 880–887, 2008.