Movie Recommendation System

Julia Kiczka

April 21, 2024

1 Primary objective

The primary objective of this project is to develop a movie recommendation system. We have access to a dataset from the MovieLens platform, particularly the ratings.csv file. This dataset contains around 100,000 ratings given by approximately 600 users for about 9,000 movies. The main idea is to create a matrix of ratings where rows represent users and columns represent movies. The goal is to predict the empty entries in this matrix, meaning to predict ratings for movies that the user has not yet rated. This could serve as a recommendation system for the user, suggesting movies that potentially have the highest likelihood of being rated positively. At first, we fill in the empty spaces in the matrix with chosen initial values, and then we attempt to apply various mathematical methods for approximating the matrix. These methods can help us find the most suitable predicted ratings. We're investigating techniques such as Non-negative Matrix Factorization (NMF), Singular Value Decomposition (SVD), iterated SVD, and Stochastic Gradient Descent (SGD) to handle matrix approximation and completion.

2 Methods description

2.1 NMF

Non-negative Matrix Factorization (NMF) approximates an $n \times d$ dimensional matrix Z with $\tilde{Z}$. In this method, $\tilde{Z}$ is constructed as $\tilde{Z}_r = W_r H_r$, where W_r and H_r are matrices with non-negative elements (W_r has r columns and H_r has r rows). We seek W_r and H_r such that the squared Frobenius norm $\|Z - W_r H_r\|^2$ is minimized.

2.2 SVD-1

This method aims to approximate an $n \times d$ dimensional matrix Z with another matrix $\tilde{Z}$. To capture only the essential information from Z, $\tilde{Z}$ is designed to have a lower rank than Z. We aim to find a rank-r matrix $\tilde{Z}_r$ ($r < \text{rank}(Z)$, where r is a parameter) that minimizes the Frobenius norm $\|Z - \tilde{Z}_r\|$.

Using Singular Value Decomposition (SVD) $Z = U\Lambda^{\frac{1}{2}}V^T$, we construct $\tilde{Z}$ as

$$\tilde{Z}_r = U_r \Lambda_r^{\frac{1}{2}} V_r^T,$$

where Λ_r contains the r largest eigenvalues of Z, and U_r and V_r contain only the corresponding columns.

2.3 SVD-2

This method iteratively applies SVD1 to matrix Z, repeating the process on the result of the previous iteration. In our approach, in each iteration, we only fill in the unknown fields of the matrix, while fixed ratings remain the same. The algorithm can stop after a fixed number of iterations or based on a predefined stopping criterion.

2.4 SGD

Stochastic Gradient Descent (SGD) aims to estimate matrix Z using matrices W and H. For a given Z of size $n \times d$, we want to find matrices W of size $n \times r$ and H of size $r \times d$ in the following way:

$$\arg\min_{W,H} \sum_{(i,j):z_{ij}\neq'?'} (z_{ij} - w_i^T h_j)^2$$

or

$$\arg\min_{W,H} \sum_{(i,j):z_{ij}\neq'?'} (z_{ij} - w_i^T h_j)^2 + \lambda(\|w_i\|^2 + \|h_j\|^2)$$

where $\lambda > 0$ is a parameter, h_j is the j-th column of H, and w_i^T is the i-th row of W. In the second version regularization term $\lambda(\|w_i\|^2 + \|h_j\|^2)$ penalizes large values in the matrices W and H, which should prevent overfitting, a common issue in machine learning where the model becomes overly tuned to the training data, capturing noise or irrelevant patterns. In general, regularization assists in controlling the complexity of the model. By discouraging overly complex models, it encourages simpler explanations for the data, which can lead to improved interpretability and generalization performance. Additionally, regularization contributes to the stability of the optimization process by preventing the parameters from diverging or reaching extreme values. This promotes faster convergence and more reliable solutions, particularly in iterative optimization algorithms.

The main idea of SGD is to move towards the minimum of the function along the direction specified by the gradient. Algorithm operates iteratively and performs the following steps:

1. Initialize matrices W and H with some initial values W_0 and H_0 (referred to as the *starting point*).
2. For each iteration:
   (a) Sample a single pair (i, j) from the set of known entries of the matrix.
   (b) Update the rows of matrix W and the columns of matrix H as follows:

$$\tilde{w}_i^T = w_i^T - \eta \cdot \left(2(z_{ij} - w_i^T h_j)h_j^T + 2\lambda w_i^T\right)$$

$$\tilde{h}_j = h_j - \eta \cdot \left(2(z_{ij} - w_i^T h_j)w_i + 2\lambda h_j\right)$$

   where η is a predefined parameter representing the step size of the updates (learning rate), and λ is a regularization parameter.
   (c) Keep the remaining elements of matrices W and H unchanged, i.e., $\tilde{w}_k^T = w_k^T$ for $k \neq i$ and $\tilde{h}_l = h_l$ for $l \neq j$.
   (d) Set W to the updated $\tilde{W}$ and H to the updated $\tilde{H}$.
3. Repeat the process until a stopping condition is met, such as reaching a certain number of iterations or satisfying a convergence criterion.

3 Data description and system quality evaluation

Dataset contains information 610 users, 9724 movies and 100837 ratings. The columns are: unique userId (integer), unique movieId (float) and rating (integer). We keep this data in two-dimensional matrix of size $n \times d$ where n is the number of users and d is the number of movies. In each element (i, j), we record the rating given by user i for movie j. If user i has not rated movie j, we leave the element empty.

To assess the effectiveness of each method and determine the optimal one, we need to evaluate their performance. We partition our dataset into two subsets: the **training set** and the **test set**. The training set is utilized to develop our models, while the test set is reserved for assessing their performance. It's essential to ensure that our models generalize well across various data splits. Thus, I implement

k-fold cross-validation which involves splitting the dataset into $k = 10$ equal-sized subsets. Then, iteratively, each subset is used once as a test set while the remaining nine subsets are used for training, repeating the process 10 times. Finally, the performance metrics from each iteration are averaged to evaluate the model's performance reliably.The test set constitutes 10% of all the data, while the training set comprises the remaining 90%. All results presented by me will be the outcome of this process.

To assess how well our program performs, we use the root-mean-square error (RMSE) as a measure of how much our predictions differ from the actual values. We have two matrices: $\boldsymbol{Z}$, containing data from the training set, and $\boldsymbol{T}$, containing data from the test set. After our algorithm generates a matrix $\boldsymbol{Z'}$, we evaluate its effectiveness as:

$$\mathsf{RMSE} := \sqrt{\frac{1}{|\mathcal{T}|} \sum_{(u,m)\in\mathcal{T}} (\boldsymbol{Z'}[u,m] - \boldsymbol{T}[u,m])^2}$$

Here, $\mathcal{T}$ comprises pairs (u, m) from the test set. Consequently, the RMSE measures the distance between matrices $\boldsymbol{Z}$ and $\boldsymbol{Z'}$ based only on elements from $\mathcal{T}$.

4 Data Imputation

We will explore various techniques for handling missing values in the matrix, including zero substitution, filling with the row mean (user's average rating), column mean (average rating of movies), and weighted average of corresponding column and row.

After filling the empty fields in the matrix using various methods, RMSE between the prepared matrix and the original one at test indices presents as follows:

Method of Imputation	**RMSE Value**
Weighted Mean	0.904
User Mean	0.946
Movie Mean	0.988
Zero Filled	3.653

Table 1: RMSE values for different imputation methods

When considering the weighted mean, empty entries are filled according to the following formula:

$$Z[i,j] = (1-\alpha) \times \text{column\_means}[j] + \alpha \times \text{row\_means}[i]$$

If the mean of a column or a row is undefined, it is assumed to be 3.5, which corresponds to the mean rating from the entire dataset. The α parameter is the object of our search, the plots below (Figure 1) visualize the relationship between RMSE and α in the respective ranges [0.1, 0.9] and [0.5, 0.7].

Having examined various methods for filling missing values, we conclude that the weighted mean imputation with a parameter value of $\boldsymbol{\alpha}$**=0.59** yields the lowest RMSE of **0.9043**. It is clear that filling missing values with zeroes does not lead to favorable results, as it would presuppose that films not yet watched will be rated very low. It has been observed that the user's rating mean provides a lower RMSE compared to the movie rating mean. This may be attributed to the subjective nature of individual users' perceptions of the movie. However, there are objectively better and worse movies, so the average rating of a film should also have significance. Based on our observations, we have determined that employing a linear combination with a higher weight assigned to user preferences compared to the weight assigned to the average movie rating yields the optimal results. We could also explore how to incorporate the fact that individuals with similar past preferences are likely to rate subsequent movies similarly. In such cases, more sophisticated methods that consider the similarity between two users could be employed - for instance, we might choose the k most similar users and base predictions solely on their ratings. To measure similarity between users, we could use metrics such as cosine similarity or Pearson correlation.

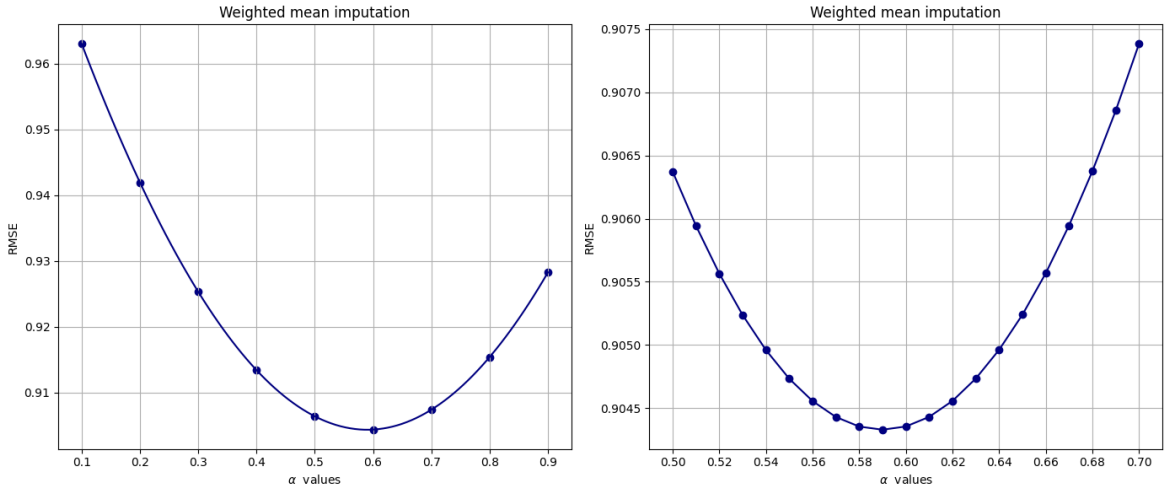

Figure 1: Weighted mean imputation: RMSE vs α

5 Parameter tuning

5.1 NMF

All presented results concern NMF with default settings in the Python implementation, with the number of iterations set to 200. Increasing the number of iterations could potentially improve performance, but in our current investigation, we are primarily focusing on selecting the number of components. Applying NMF and exploring the optimal number of components indicates that the best **number of components = 36**, achieving the lowest RMSE of **0.891**. Below, in Figure 2, we can observe the relationship between the number of components and RMSE.

5.2 SVD-1

Below, in Figure 3, we can observe the relationship between the number of components and RMSE for SVD. The minimum RMSE achieved with SVD is **0.889**, occurring at **number of components = 7**.

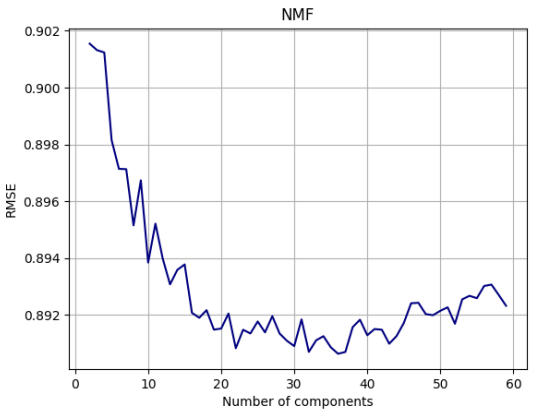

Figure 2: NMF - RMSE and components number

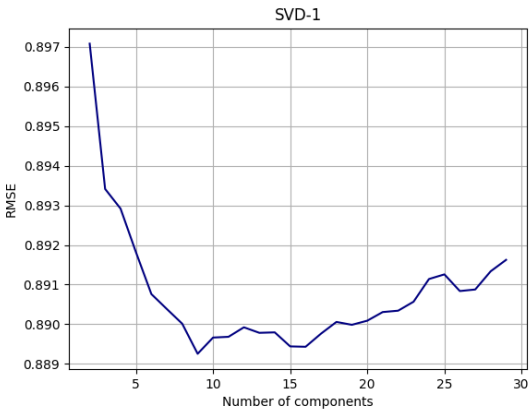

Figure 3: SVD1 - RMSE and components number

5.3 SVD-2

As I mentioned before, in this method, we apply SVD multiple times. We expect that the obtained results will be slightly better than for SVD-1, and the experiments confirm this. In addition to

determining the number of components, our task in SVD-2 also involves selecting a reasonable stopping criterion. I will consider the number of iterations. In the Figure 4 below, we can observe that the minimum is reached for the **number of components = 2** and the number of **iterations = 33**, resulting in an RMSE of **0.875**.

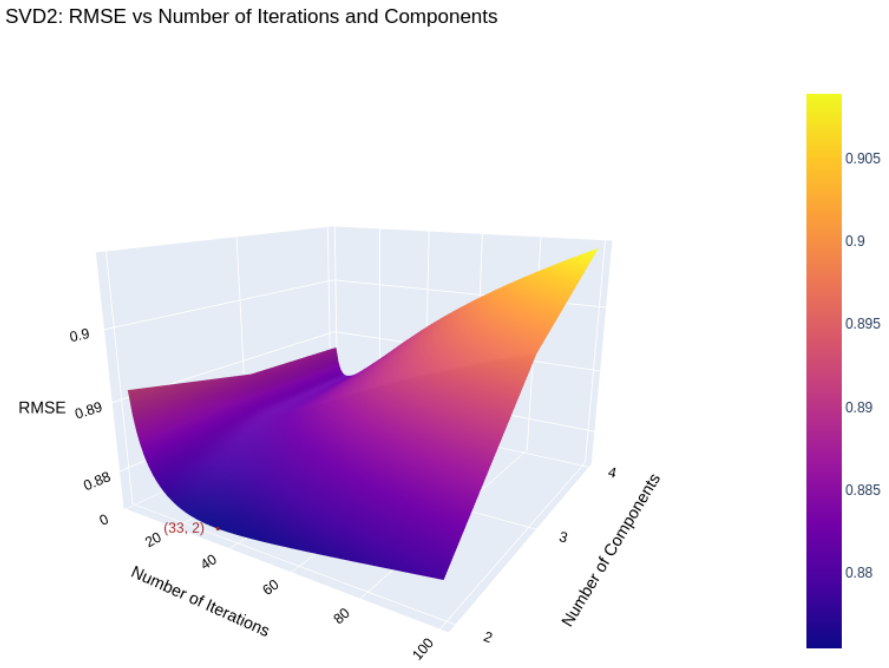

Figure 4: SVD2 - RMSE vs number of components and number of iterations

5.4 SGD

For a given Z of size $n \times d$, we want to find matrices W of size $n \times r$ and H of size $r \times d$ and in every step we update W and H the following way:

$$\tilde{w}_i^T = w_i^T - \eta \cdot \left(2(z_{ij} - w_i^T h_j)h_j^T + 2\lambda w_i^T\right)$$

$$\tilde{h}_j = h_j - \eta \cdot \left(2(z_{ij} - w_i^T h_j)w_i + 2\lambda h_j\right)$$

where η is considered as learning rate and r as number of components. The parameters we need to determine in this algorithm are the number of epochs, learning rate, number of components, and the regularization parameter λ. Initially, all elements of matrices W and H are set to $\sqrt{\frac{m}{r}}$, where m denotes the global mean of ratings = 3.5. This initialization ensures that the inner product $w_i^T h_j = m$, thereby establishing an equitable distribution of ratings at the beginning.

When determining the duration of our algorithm, the number of epochs becomes crucial. This largely depends on the chosen learning rate. Assuming we have an optimal learning rate – small enough to find the minimum accurately without overshooting it, yet large enough for the algorithm to converge relatively quickly – we need to consider the number of epochs to perform. Additionally, we can assess if the steps taken are sufficiently small by examining the stopping condition. For instance, if the changes made in further iterations are small enough, we can stop the algorithm. I attempted to evaluate this by looking at the improvement $\epsilon := \|W_{\text{previous}} - W\| + \|H_{\text{previous}} - H\|$. It turned out that setting the stopping condition as $\epsilon < 10^{-46}$ resulted in the algorithm finishing within < 10 iterations in most cases. However, this value was too large to get satisfying results, so I chose $\epsilon < 10^{-47}$ and a maximum number of iterations of $6,000,000$ just in case the condition is not met. In practice, we rarely

get $\epsilon < 10^{-47}$. It seems that the stopping condition doesn't help a lot in this situation, so maybe more subtle method of measuring changes should be used. Nevertheless, at this point sticking to number of iterations seems easier. Chosen number of iterations is quite large, but setting it smaller eg. 1,000,000 makes significant difference to the results (For optimally chosen parameters, we have **RMSE = 0.892** for 1,000,000 vs **RMSE = 0.879** for 6,000,000). Figure 5 below shows RMSE computed on the test set after training algorithm with different learning rate and number of components, for now setting λ**=0**, so without regularization. We can see that the smallest value of RMSE **= 0.879** is given for **learning rate=0.0005** and **number components = 4**.

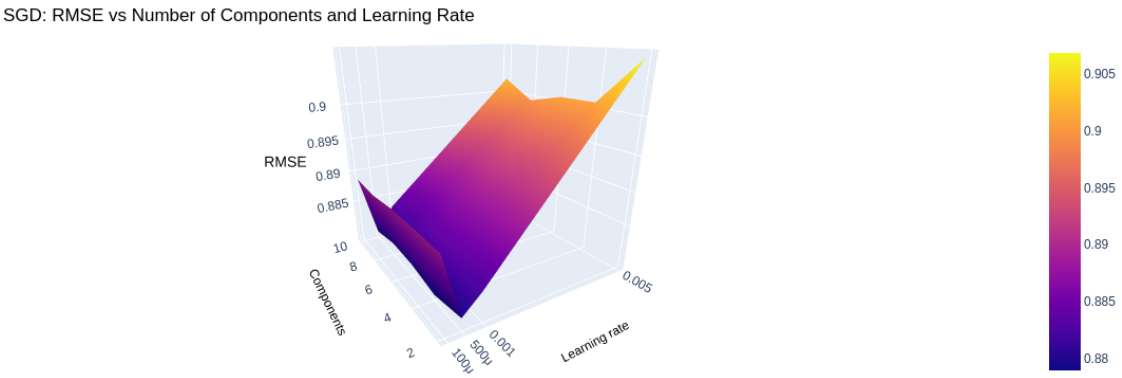

Figure 5: SGD - RMSE vs number of components and learning rate

In the code, I implemented SGD exactly as described, where the step in the direction of the gradient is computed based on calculations for a single specific pair of randomly chosen indices. However, one could also consider computing the step based on a subset of training indices, not just one pair (it would be **minibatch SGD**). Other enhancements include techniques like momentum and decay, which often helps in similar problems, but may not necessarily apply here.

- **Momentum**: Momentum is a technique that helps alleviate issues related to getting stuck in local minima or ravines during model training. It involves adding a fraction of the previous step's weight update to the current step. This way, if the gradients in successive steps point in a similar direction, momentum accelerates learning.

- **Decay**: The decay parameter is used to reduce the learning rate over time. It works by decreasing the value of the learning rate at each learning step. Decay is applied to prevent excessive oscillation around the minimum of the cost function and to allow the model to achieve more precise adjustments by gradually adjusting the learning rate as learning progresses.

Regularization

Maximal number of iterations (6,000,000) I discussed before seems quite large, but in fact setting it equal to smaller number (1,000,000) makes significant contribution to the results. With 1,000,000 iterations **RMSE is 0.892**, with 6,000,000 it's **0.879**. On the other hand, keeping it high makes computations even more time consuming (now we have also parameter λ to optimize), so for now, for regularization problem, let's set it to 1000,000 and look at the results. In Figure 6 we can see comparison of RMSE, learning rate and λ. It turned out that for every component number $\in [2, 4, 6, 8]$ plot looks similar and smallest value of **RMSE=0.885** on the test set is obtained for number of **components = 6**, **learning rate = 0.001** and λ**=0.001**.

We can see that RMSE is smaller than for SGD without regularization with the same number of iterations (1,000,000), but it is bigger than RMSE for SGD with 6,000,000 iterations. Intuitively, penalizing matrices with large coefficients should prevent overfitting and improve the final result on the test set. On the other hand, we impose additional conditions on the W and H, which makes it harder to find the optimal ones. It is also important to note that SGD is an algorithm that involves a random factor - selecting a pair of indexes for gradient update. The difference in RMSE is certainly influenced by considering the different number of iterations in the two problems, so it does not necessarily make sense to compare the numerical results exactly. It is worth simply looking at the

results of the version with regularization and noting that the parameters found (learning rate, number of components) are different than for the problem without regularization and noticing that the version with regularization and fewer iterations still works better than, for example, SVD or NMF. However, the comparison with other methods is also not entirely reliable, because the number of iterations in SGD is significant compared to the number of iterations in NMF. We can simply look at it as an experiment to see how much we can achieve using SGD. To perform an extremely detailed search that would result in finding the optimally best parameters, greater computational resources would be required.

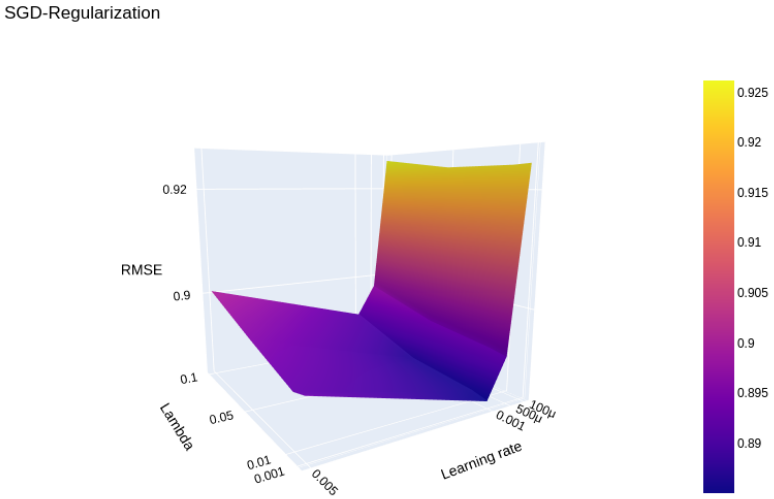

Figure 6: SGD + Regularization with number of components = 6

6 Conclusions

We can observe that the iterated SVD-2 performs the best, followed by SGD, then SVD-1, and finally NMF. A very important element of the considered problem was the selection of appropriate initial values for the matrix. The selection of the number of components typically involves finding a good balance between overfitting on the training data and insufficient approximation of the matrix. We aimed to determine parameters that enable us to reconstruct the given matrix adequately but not to the extent that it fits the test data well, not just the training data. Below (Table 2) we can see a comparison of the results:

Method	RMSE
Iterated SVD	0.875
SGD	0.879
SVD-1	0.889
NMF	0.891

Table 2: Summary of RMSE for different methods