Methods of classification and dimension reduction
Project 1

Weronika Szota & Mateusz Szczepański

1. Problem

1.1 Problem description

Let us consider the *Recommender system* problem. From *ratings.csv* [1] we have 100837 ratings posted by 610 users for 9724 movies, its format is following

userId	movieId	rating
1	1	4.0
1	3	4.0
1	6	4.0
1	47	5.0
1	50	5.0
2	1	3.5

Figure 1: Data format of *ratings.csv*

where userId is a unique user id, movieId is a unique movie id and rating is a number from the set $\{0.5, 1, \ldots, 4.5, 5\}$. In the original file *ratings.csv* there is one more column, however it is not in the scope of the project.

Our goal is to predict "ratings", which are not provided by users.
In order to that we start with dividing our data in two sets - training set and test set, randomly, so that training set includes around 90% of ratings of each user.
After obtaining results using different procedures, we compare them based on *quality of the system* defined as follows.

Assume that our algorithm after training on $\mathbf{T}$ computes $\mathbf{Z}'$, a matrix containing elements $\mathbf{Z}'[u, m]$ for $(u, m) \in \tau$, where τ is defined as all indices of the test set (it is marked with red color in Section 1.2). **Quality of the system** is computed as **root-mean square error**

$$\mathrm{RMSE} = \sqrt{\frac{1}{|\tau|} \sum_{(u,m)\in\tau} (\mathbf{Z}'[u, m] - \mathbf{Z}[u, m])^2}.$$

1.2 Dummy example

The presented example showcases a sophisticated front-end solution to the problem at hand. The goal of this solution is to identify the mysterious '?' numbers. In this implementation, the training set is highlighted in green, while the test set is distinguished by a red color.
Let say we start with the following 4x4 matrix.

$$\mathbf{Z} = \begin{array}{c} \\ \text{user1} \\ \text{user2} \\ \text{user3} \\ \text{user4} \end{array} \begin{array}{c} \begin{array}{cccc} \text{movie1} & \text{movie2} & \text{movie3} & \text{movie4} \end{array} \\ \begin{pmatrix} 1.0 & 1.5 & 2.0 & ? \\ 5.0 & 2.0 & ? & 3.5 \\ 4.0 & ? & 3.5 & 2.5 \\ ? & 1.5 & 2.0 & 5.0 \end{pmatrix} \end{array} \longrightarrow \ldots$$

The subsequent step involves populating the matrix using a specified methodology (in this case, the values are replaced with 0's) for all occurrences of the mysterious '?' numbers in both the training and test sets. Following this, an algorithm is applied to the populated matrix, and the root mean square error (RMSE) is calculated based on the values from the test set. This process can be visually represented as follows

$$\dots \xrightarrow[\text{the matrix}]{\text{Fill up}} \begin{pmatrix} 0.0 & 1.5 & 2.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 3.5 \\ 4.0 & 0.0 & 0.0 & 2.5 \\ 0.0 & 1.5 & 2.0 & 0.0 \end{pmatrix} \xrightarrow[\text{algorithm}]{\text{Perform the}} \begin{pmatrix} 1.0 & 2.0 & 2.0 & 0.5 \\ 4.5 & 1.0 & 0.5 & 4.0 \\ 4.0 & 1.5 & 3.0 & 2.0 \\ 0.5 & 2.0 & 2.5 & 2.0 \end{pmatrix} \xrightarrow[\text{on the red test set}]{\text{Compute the RSME}} \text{RMSE} \approx 1.6 \, .$$

The final RMSE is approximately 1.6. It is important to note that the algorithm utilized may modify the training set to a certain degree. However, this is not a cause for concern as the ultimate objective is to accurately predict values in the test set.

1.3 Technical approach to the problem

The solutions for this project were implemented using the `Python3` programming language and can be found in the Github repository [4]. The repository contains information about the code, training and test sets used. In total, 10 training and test pairs were created using the script `generating_train_and_test_files.py`. Pairs are indexed as `train_`i, `test_`i, for $i = 0, 1, \dots, 9$. All algorithms were tested on all of these pairs to get the best possible results. The script `testing.py` was used to test and fit variables like truncation level r or number of iterations n in SVD2. Finally, the script `recom_system_310550_309068.py` was developed, which includes the most universally applicable variables and methods that were subjectively chosen by us. This script can be executed on any command line or prompt that supports `Python3`, as follows

```
python recom_system_310550_309068.py --train train_file --test test_file \\
--alg ALG --result result_file   .
```

2. Data imputation methods

We will explore various techniques for handling missing data in our training set, including those that are missing as well as those from the test set. For each method, an initial RMSE value has been provided to serve as a starting point. It is worth mentioning that the RMSE value is approximated due to the fact that it was checked on all the possible pairs of training and test sets.

- **Method 0**: Zeros
  Naive method - set missing values to 0.

  Initial RMSE ≈ 3.65. Oscilates from 3.645, up to 3.679.

- **Method 1**: Mean (row/column/matrix)
  We replace missing values with mean of known user's ratings.

  Initial RMSE ≈ 0.96. Oscilates from 0.953, up to 0.979.

- **Method 2**: Random variable from $N(\mu, \sigma^2)$ (row, limited)
  We propose replacing missing values with random variables from $N(\mu, \sigma)$. Based on the known ratings we estimate parameters of the distribution $\hat{\mu}$ and $\hat{\sigma}$ as sample mean and sample standard deviation, where sample contains of known ratings for particular user. After we sample from the distribution defined as above, we replace observations grater than 5 with 5 (thus this is the highest possible rating) and lower than 0 with 0 (thus this is the lowest possible rating).

  Initial RMSE ≈ 1.31. Oscilates from 1.287, up to 1.320.

- **Method 3**: Random variable from distribution on $\{0, 1, 2, 3, 4, 5\}$
  Based on the ratings granted from each user, for each rating $r \in \{0, 1, 2, 3, 4, 5\}$ we estimate probability

  $$p_r = P(\lfloor R \rfloor = r).$$

  Having that we sample value r with calculated probability p_r.

  Initial RMSE ≈ 1.32. Oscilates from 1.311, up to 1.351.

- **Method 4**: Most frequent value (row)
  We choose the most frequent value based on $\lfloor r \rfloor$ from each row and fill the gaps with the value obtained for each row independently.
  Initial RMSE ≈ 1.08. Oscilates from 1.074, up to 1.095.

- **Method 5**: Weighted average of means
  Missing value in $i-th$ row and $j-th$ column is replaced by weighted average:

$$0.66 \cdot mean(i-th\ row) + 0.34 \cdot mean(j-th\ column),$$

  the means are calculated based on the given ratings. This method is intuitive and represents that each rating is caused in 66% by user personal taste and in 34% by the general opinion about the movie.
  Initial RMSE ≈ 0.930. Oscilates from 0.923, up to 0.943.

  In order to obtain the optimal weights we consider initial RMSE based on filling missing values with

$$\alpha \cdot mean(i-th\ row) + (1-\alpha) \cdot mean(j-th\ column)$$

  as a function of α. As can be seen in Figures 2 and 3 the optimal value for α is 0.66.

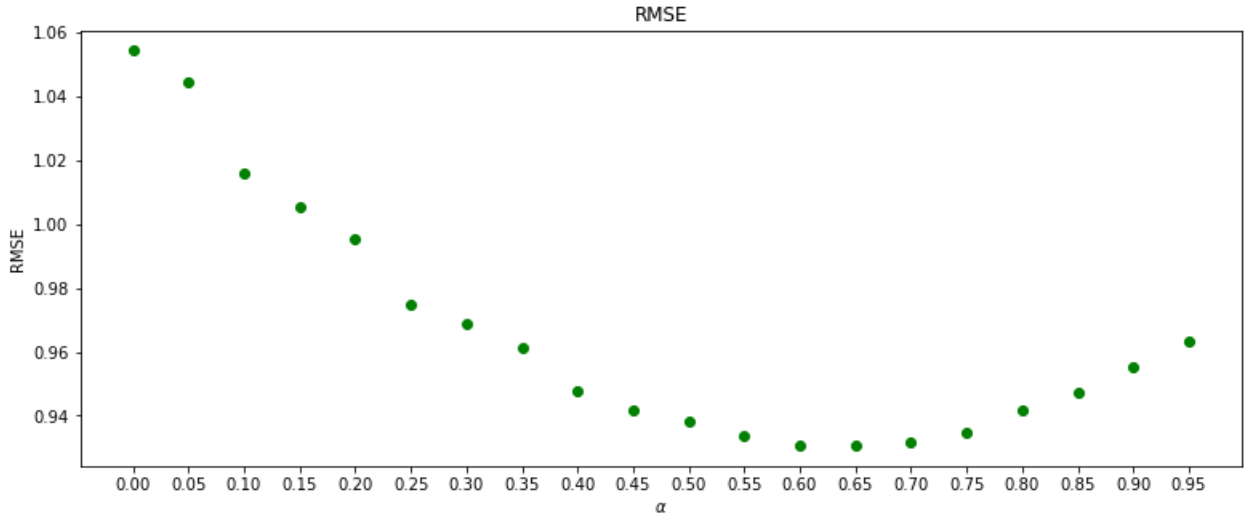

Figure 2: RMSE depending on the choice of weight

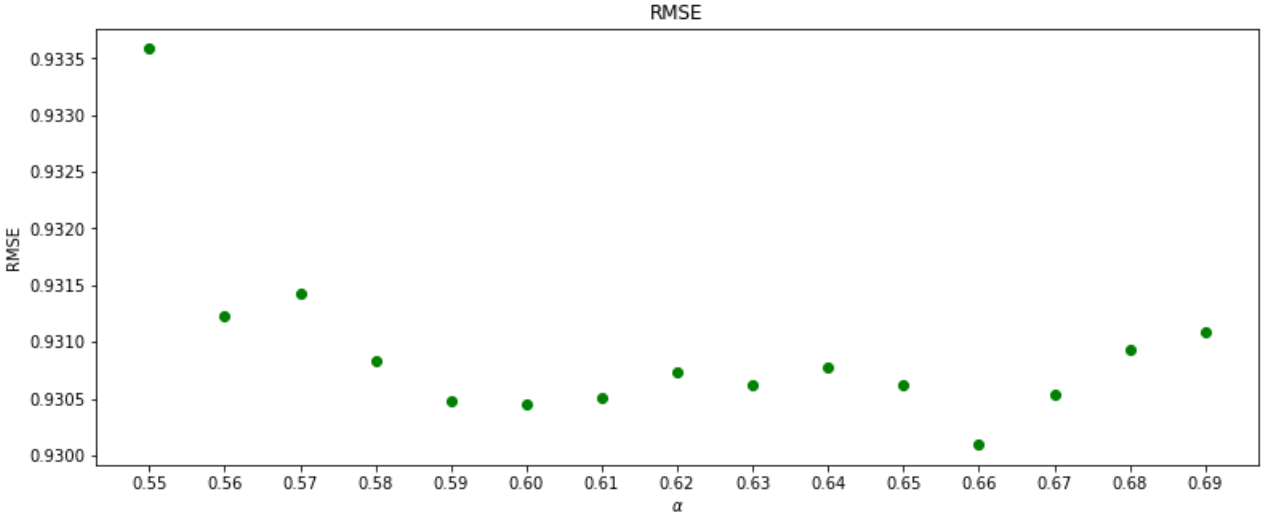

Figure 3: RMSE depending on the choice of weight

3. Algorithms

3.1 SVD1 (Singular Value Decomposition)

The singular value decomposition of a matrix $\mathbf{Z}$ is the factorization of matrix $\mathbf{Z}$ into the product of three matrices given as in the following theorem.

Theorem 1 *Let $\mathbf{Z}$ be a real $n \times d$ (assume $n \geq d$). Then it can be written as a product of three matrices*

$$\mathbf{Z} = \mathbf{U}\Lambda^{\frac{1}{2}}\mathbf{V}^T,$$

such that

- $\mathbf{U}$, $\mathbf{V}$ *are orthonormal matrices of size $n \times d$ and $d \times d$, respectively,*
- $\mathbf{\Lambda}$ *is a $d \times d$ diagonal matrix with non-negative entries.*

Theorem 1 does not provide anything other than matrix decomposition. The whole power of the SVD is in the approximation of the matrix which can be obtained thanks to the next theorem.

Theorem 2 *Let $\mathbf{Z}$ be a real $n \times d$ matrix, from SVD we have $\mathbf{Z} = \mathbf{U}\mathbf{\Lambda}^{\frac{1}{2}}\mathbf{V}^T$. Let $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n \geq 0$ be the eigenvalues of $\mathbf{Z}\mathbf{Z}^T$. Let $\mathbf{U}_r$ denote the matrix consisting of first r columns of $\mathbf{U}$ and similary let $\mathbf{V}_r$ denote the first r columns of $\mathbf{V}, r < d$. Additionally, let $\mathbf{\Lambda}_r$ be a square $r \times r$ matrix - truncated Λ to first r rows and columns. Define $\tilde{\mathbf{Z}}_r = \mathbf{U}_r\mathbf{\Lambda}_r^{\frac{1}{2}}\mathbf{V}_r^T$. We have*

$$||\mathbf{Z} - \tilde{\mathbf{Z}}_r||^2 = \sum_{i=r+1}^{d} \lambda_i.$$

Theorem 2 demonstrates that we can approximate a given matrix by using only a subset of its largest eigenvalues along with their corresponding eigenvectors. This approximation is obtained by using a truncated matrix, denoted by $\tilde{\mathbf{Z}}_r$, which is shown to be a good approximation (in the norm sense) of the original matrix.
In practical applications, the primary objective is to identify and extract a small number of significant eigenvalues from a matrix. These eigenvalues, along with their corresponding eigenvectors, can be used to construct a reduced matrix that captures the relevant features of the original dataset.

3.2 SVD2 (Iterative SVD)

The method proposed in this section presents an extension of the singular value decomposition algorithm, referred to as SVD1, by performing a few additional iterations. To ensure the convergence of the SVD2 algorithm, it is necessary to specify a stopping condition. This can be accomplished by either selecting a fixed number of iterations beforehand or by setting a condition based on the properties of the training set. Specifically, the method involves the following steps

1. There is given matrix $\mathbf{X}$.
2. Perform SVD1 algorithm on $\mathbf{X}$ with the given level of reduction, denoted as r.
3. Now there is given a new matrix $\tilde{\mathbf{X}}_r$.
4. Create a new matrix $\mathbf{X}_2$ by first using the original matrix $\mathbf{X}$ to populate the training data, and then replacing the remaining entries with the corresponding values from the truncated matrix $\tilde{\mathbf{X}}_r$.
5. With the matrix $\mathbf{X}_2$ go back to step 1 and perform again as long as stop condition does not hold.

3.3 NMF (Non-negative matrix factorization)

Non-negative matrix factorization aims to approximate $n \times d$ non-negative matrix $\mathbf{Z}$ as a product of two matrices, say $\mathbf{W}$ (of size $n \times r$) and $\mathbf{H}$ (of size $r \times d$) with the restriction that $\mathbf{W}$ and $\mathbf{H}$ are non-negative matrices. There are no other restrictions on the matrices. Actually, NMF is a set of algorithms, one of the parameter id a distance dist function between matrices and NMF's goal is to minimize the distance between $\mathbf{Z}$ an $\mathbf{WH}$. Summing up, NMF aims to solve the following problem

Given $\mathbf{Z}$ of size $n \times d$ and $r \leq d$ find

$$\arg \min_{W,H} \text{dist}(\mathbf{Z}, \mathbf{WH}),$$

such that $\mathbf{W}, \mathbf{H} \geq 0$.

Often the Frobenius norm is considered

$$\text{dist}_F(\mathbf{Z}, \mathbf{WH}) = \sum_{i=1}^{n} \sum_{j=1}^{d} (\mathbf{Z}(i,j) - (\mathbf{WH}(i,j)))^2.$$

There are several reasons for this norm being popular. It implicitly assumes that $\mathbf{Z}$ is of form $\mathbf{WH} + \mathbf{N}$,where $\mathbf{N}$ represents a Gaussian noise.

3.4 SGD (Stochastic Gradient Descent)

The general Gradient Descent is an iterative algorithm that numerically estimates where a function outputs its lowest values. It finds local minima by approximating solutions with number, we follow the negative of the gradient. More details can be found in [6], [7] and [8].

Let us consider minimizing the function f. We start with the point x_0 and move a positive distance α in the direction of the negative gradient, then our new and improved x_1 will look like this

$$x_1 = x_0 - \alpha \nabla f(x_0),$$

where $\nabla f(x)$ is a gradient of function f. More generally we have

$$x_{n+1} = x_n - \alpha \nabla f(x_n).$$

In Gradient Descent technique, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset.

In Stochastic Gradient Descent a few samples are selected randomly instead of the whole data set for each iteration. In this variant, only one random training example is used to calculate the gradient and update the parameters at each iteration.

In previous methods we needed first to impute missing values in matrix $\mathbf{Z}$. Now we can refolmulate our problem. For a given $\mathbf{Z}$ of size $n \times d$ we want to find matrices $\mathbf{W}$ of size $n \times r$ and $\mathbf{H}$ of size $r \times d$ in the following way

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

or for parameter $\lambda > 0$

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

where h_j is j-th column of h, whereas w_i^T is $i-$th row of $\mathbf{W}$.

4. Solutions

4.1 SVD1

In the following solution there has been used an implementation of the SVD from `sklearn` package in Python. More details can be found in the original documentation [2].
Before applying the truncation, we need to determine the appropriate level of truncation that will yield the best results. Our goal is to choose the lowest possible value of r that still produces an accurate approximation of the original matrix. Final RMSE is also dependent on the data imputation method described in Section 2. For example for **Method 0**, to illustrate the impact of different values of r, we computed the root mean square error (RMSE)

for values of r ranging from 0 to 30, where 0 means RMSE without performing SVD - only applying the inputation method. The visualization is done only for pair `train_0` and `test_0`, however similar behavior is noticable for all splits.

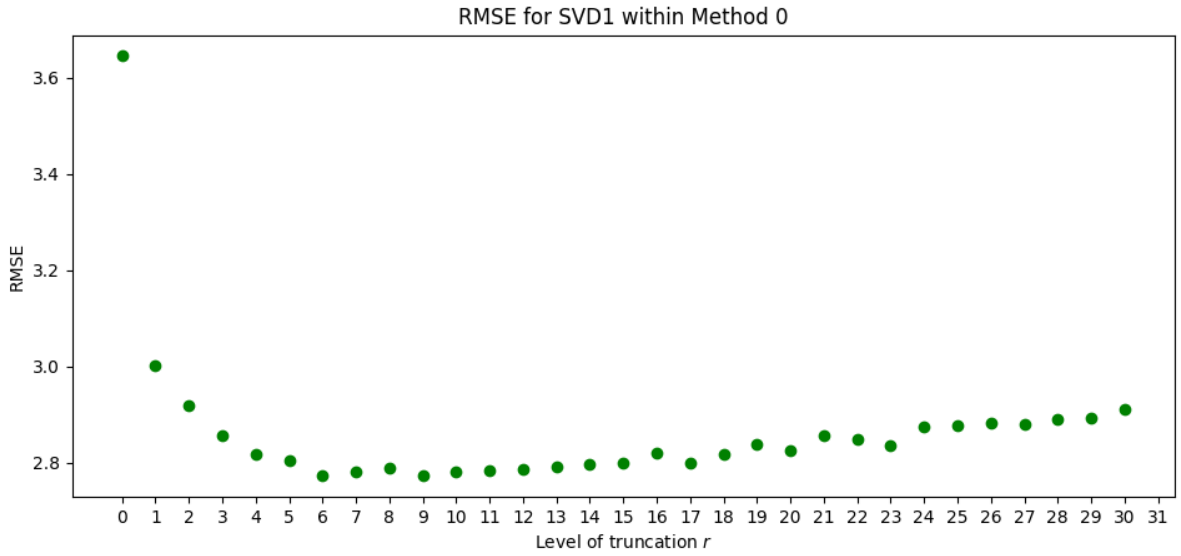

Figure 4: RMSE, SVD1, **Method 0**, $1 \leq r \leq 30$, for `train_0` and `test_0`

To achieve the best approximation (universally), every method has been tested on all train/test sets to determine which r value is best (on average) for the particular method.
We have found that for **Method 0**, truncation levels $r = 6$ and $r = 9$ produce similar results. However, in order to choose the most efficient truncation level, we aim to select the lowest possible truncation level that still yields acceptable accuracy. Therefore, we will select $r = 6$ for our table. Let us examine and compare the outcomes of various methods.
The final result for SVD1 is shown in the table below.

Method	$\approx r$	$\approx$ RMSE
Method 0 - Zeros	6	2.774
Method 1 - Mean of row	2	0.920
Method 2 - Normal dist.	1	0.943
Method 3 - Based on data	1	0.950
Method 4 - Frequencies	10	1.029
Method 5 - Weighted mean	5	0.899

A few interesting and noteworthy corollaries:

- For **Method 0**, it was observed that the RMSE for $r \geq 7$ was consistently worse than for $r = 6$ across almost every pair of training and test sets. This suggests that $r = 6$ represents some kind of a local minimum in this case.
- In the case of **Method 1**, it was found that $r = 2$ consistently produced the best results across almost all training and test sets, with RMSE increasing for larger values of r. Notably, using only one feature/component ($r = 1$) generally led to poor results.
- The most intriguing results were obtained with **Method 2**, which showed that using normally distributed missing data made it possible to approximate the matrix very well with just one leading component/feature. Increasing the number of components/features (r) generally led to higher RMSE.
- Similarly to the previous method, **Method 3** yielded the best results for $r = 1$. However, the resulting RMSE was worse than in the previous method.

- **Method 4** did not produce very good results. The RMSE was high, and it required a truncation level of at least 8, up to 10.
- On average, **Method 5** produces the best results for $r = 5$, with RMSE values below 0.9. It should be noted that for certain test and train pairs, the optimal value for r can be as high as 15 or 23. However the method is the best in comparison to all others.

Furthermore, we present a boxplot to showcase the selection process of the optimal truncation level, r, for **Method 5** within all prepared splits. We can notice that the best result is yielded by $r = 5$.

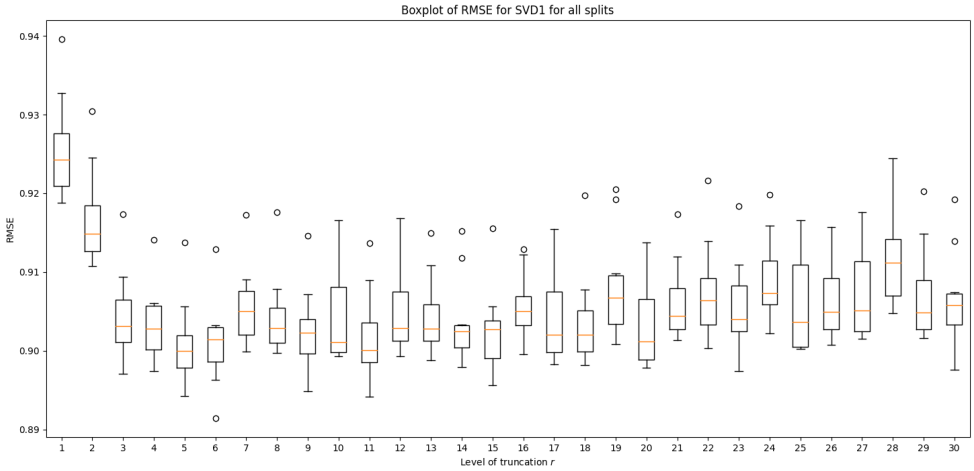

Figure 5: RMSE for SVD1 as a function of truncation level r across all splits, **Method 5**

So for the final implementation of SVD1 we have chosen **Method 5** and truncation level $r = 5$. It is good to note that $r = 0$ is skipped in the above plot.

4.2 SVD2

In the following solution we use iteration of SVD1 algorithm as can be seen in the description above. This method aims to improve the SVD1 result throught iterations. Same as for SVD1 we have to set number of components (truncation level). We will take the optimal setting derived from the analysis for SVD1 which is method 5 and $r = 5$. Another parameter is the maximum numbers of iterations that should be performed to correct our result. Initially we set this value to 15, but the optimal result (in sense of minimizing RMSE) can be found after nine iterations. We can see that the average RMSE is obtained for five iterations, however the size of the box which coresponds to IQR is too wide. As can be seen in the Figure 6 average RMSE is lowest for nine iterations. Also box characterization for nine iterations is quiet good - the IQR is relatively small compering to other boxes, so the RMSE values don't vary so much.

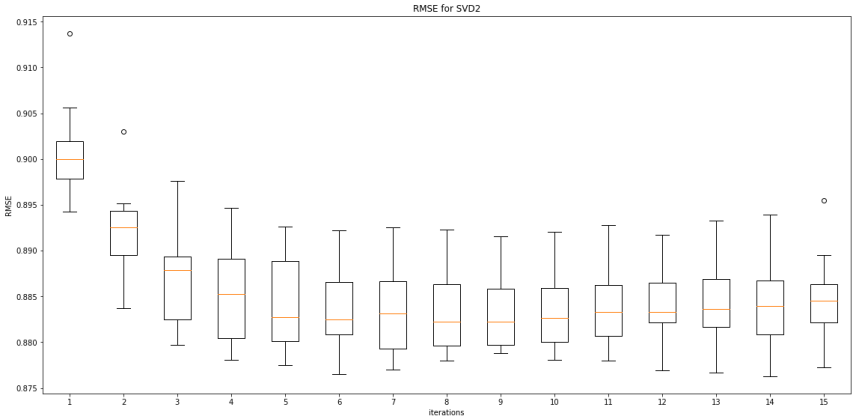

Figure 6: RMSE for SVD2 as a function of number of iterations across all splits

In addition to determining number of steps we tried to define stop condition based on matrix norm. As we are operating on matrices and the goal is to minimizing RMSE we can think about the process of tunning the results as of the way to lower the distance between the result and test matrices (the norm of the difference of the test and result matrices). Therefore we considered Frobenius norm < 0.01. It occured that it was obtained after huge number of steps and didn't lead us to better results than obtained in nine iterations. That is why we decided to ommit this stop condition and define just the one based on number of iterations.

For the final implementation of SVD2 we have chosen **Method 5**, truncation level $r = 5$ and 9 iterations.

4.3 NMF

In the following solution there has been used an implementation of the NMF from `sklearn` package in Python. More details can be found in the original documentation [3].
There are a few technical issues that must be addressed. Firstly, the given implementation allows us to choose the initial matrices $\mathbf{W}$ and $\mathbf{H}$, as well as a maximum number of iterations for the algorithm to perform. We did not observe any crucial differences between the possible inputs for `initial`, whether it be `'random'`, `'nndsvda'`, or `'nndsvdar'`. However, we ultimately chose `'nndsvda'` as the initial input. The choice of the number of iterations will be explained at the end of this subsection.
Similarly to SVD1, let see for **Method 0**, how different values of r impacts on RMSE. In this showcase example maximum number of iterations is set to `max_iter=200`.

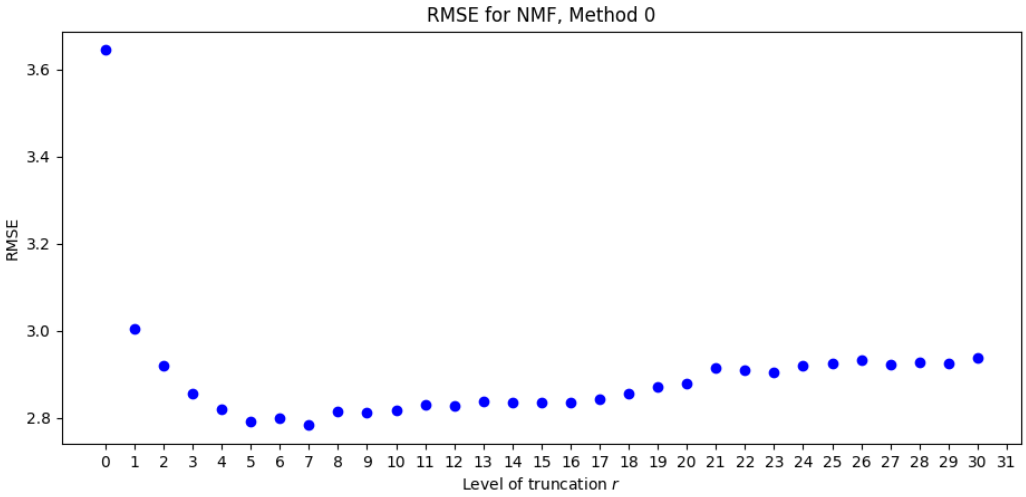

Figure 7: RMSE, NMF, **Method 0**, $1 \leq r \leq 30$, for `train_0` and `test_0`

In this case, we observe that the optimal values of RMSE are obtained for $r \in \{5, 7\}$. It should be noted that a higher value of r results in a significant increase in the computation time. Thus, we would prefer to select $r = 5$, which yields RMSE = 2.791.
Let see, how the algorithm deals with different imputation methods.

Method	$\approx r$	$\approx$ RMSE
Method 0 - Zeros	5	2.791
Method 1 - Mean of row	17	0.920
Method 2 - Normal dist.	1	0.945
Method 3 - Based on data	1	0.950
Method 4 - Frequencies	13	1.033
Method 5 - Weighted mean	27	0.893

A few facts and corollaries:

- For **Method 0**, there was a visible pattern that RMSE for $r = 7$ was worse than for $r = 6$, so we can conclude that $r = 6$ is some kind of local minimum in this case.
- For **Method 1**, almost always $r = 2$ provided the best results - then RMSE was increasing. Only one feature/component ($r = 1$) was providing pretty bad results.
- **Method 2** produced the most interesting result. It turned out that normally distributed missing data allows to use only 1 leading component/feature to approximate the matrix very well. Bigger r was giving higher RMSE.
- In the case of **Method 3**, $r = 1$ gave the best results, but the overall RMSE was worse than in the other methods.
- **Method 4** produced not too good results. RMSE is high and it requires level of truncation at least $8 - 10$. However, there is a special threshold around $10 \leq r \leq 20$ which results in a significant reduction of RMSE.
- While **Method 5** yielded the best results on average, it exhibited lower reliability than other methods. The optimal truncation level for all data splits oscillated between 24 and 28. This can be observed in Figure **??**, where selecting $r = 27$ leads to a relatively high level of RMSE for the `train_0` and `test_0` sets, despite producing the best results for 6 out of 10 sets.
- Some may say that there is a visible periodicity in **Method 1**. However, after analyzing $r > 50$, it turns out that RMSE is increasing without any special periodicity.
- The parameter `max_iter` didn't seem to make a significant difference for most values. Only when we compared levels like 10 and 200 was there a noticeable difference, but this difference was negligible between 200 and 1000. After some testing, we chose to set `max_iter=200`, as it provided good performance and reliable results.

Additionally, we provide a boxplot for **Method 5** across all data splits, showcasing the process of selecting the optimal truncation level, r, for each method.

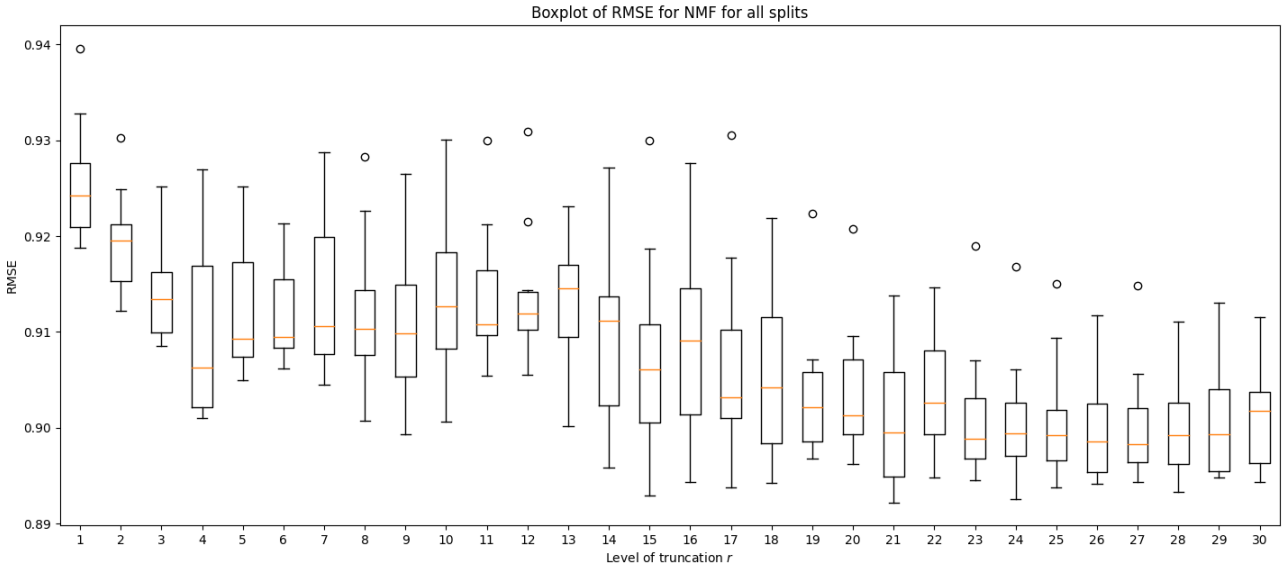

Figure 8: RMSE for NMF as a function of truncation level r across all splits, **Method 5**

Observing the results obtained for truncation levels within the range of 23 to 27, we note that they produce comparable outcomes in terms of accuracy. However, upon comparison, the optimal truncation level was found to be $r = 27$, yielding the highest level of accuracy among all alternatives. Therefore, $r = 27$ is selected as the optimal truncation level for all the splits and the best method.
The final implementation uses **Method 5** with `max_iter=200`, `initial='nndsvda'`, and truncation level $r = 27$.

4.4 SGD

The following solution requires an implementation of the algorithm made by us. We should predifined parameters: r (number of components), *steps* (number of steps to be performed in order to find the minimum), *iterations* (number of iterations for each step), *learning rate* (how fast our solution will move in the direction of the negative gradient), λ and matrices $\mathbf{W}, \mathbf{H}$ defined in the description of SGD method.

For a patricular configuration one step iteration includes: sampling a pair $(i, j) : z_{i,j} \neq '?'$, updating $i - th$ row of matrix $\mathbf{W}$ and $j - th$ column of matrix $\mathbf{H}$ with respect to negative gradient of the function and learning rate α. We repeat this *iterations* times. Before starting performing the algorithm for next *step* we update value of *learning rate* so we will slower move toward the minimum. Thus the value of *learning rate* will constantly decrease.

The main problem we face when working with this algorithm is a bunch of parameters that have to be set. Most of them don't have the resonable explanation, thus we base our choices on ituitions and emipirical tests to lower RMSE value. The initial matrices $\mathbf{W}$ and $\mathbf{H}$ consists of random variables from uniform distribution such that elements of initial matrix $\mathbf{Z} = \mathbf{WH}$ were mostly between 0 and 5. We use random variables from uniform distribution $U(0, 0.2)$ as the elements of matrix $\mathbf{W}$ and random variables from uniform distribution $U(0.9, 2.8)$ as elements of matrix $\mathbf{H}$.

Then we consider optimal value for λ. Our therotical introduction shows that λ appear in equation only with $(||w_i||^2 + ||h_j||^2)$. Since we are minimizing our function, very small values of λ make the component $(||w_i||^2 + ||h_j||^2)$ less important (value tends to 0) and thus the problem is mostly about minimizing $(z_{ij} - w_i^T h_j)^2$ so it is easier for the algorithm to find the minimal value. It is consistent with simulations we perform. As can be seen $\lambda = 0.00001$ gives the best results (first point in the Figure 9).

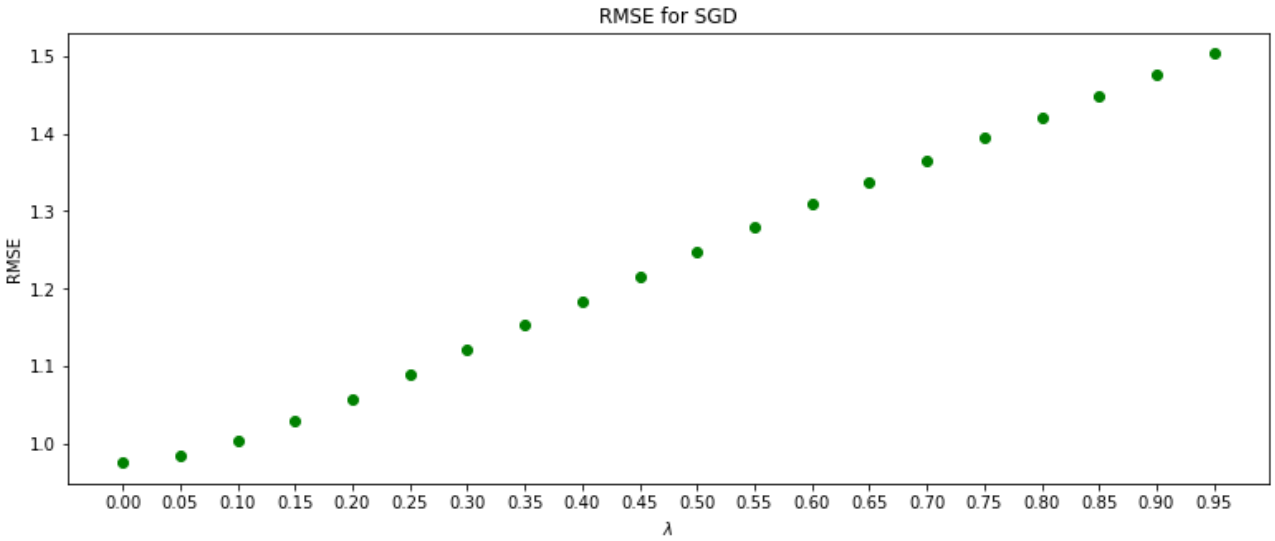

Figure 9: Average RMSE across all splits as a function of λ, *iterations* $= 10000$, $r = 7$

Having $\lambda = 0.00001$ we move on to set the next parameter which is number of *iterations*. We investigate numbers from 10000 to 100000 with step 10000. It occured that higher repetitions lead to better results as can be seen in the Figure 10. However higher number of repetition demands more computational time. That is why we will limit our analysis to 50000 repetitions, since it gives relatively good results and does not require so much time.

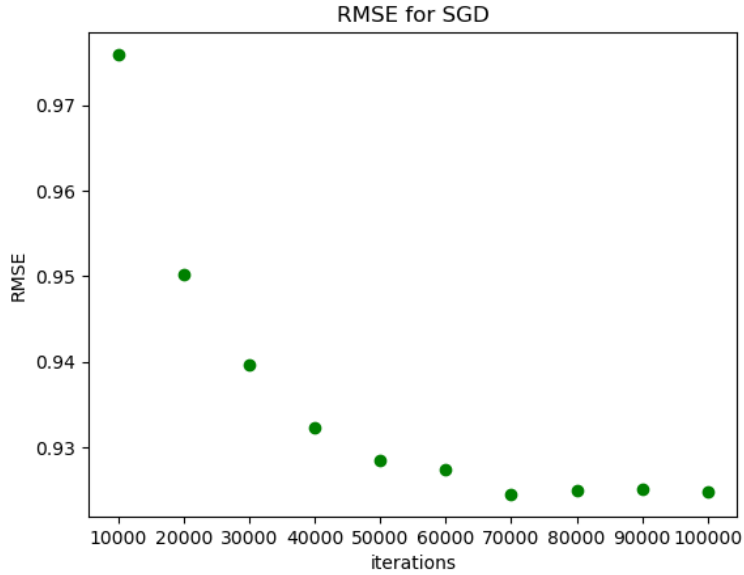

Figure 10: Average RMSE across all splits as a function of *iterations*, $r = 7$, $\lambda = 0.00001$

For *iterations* set to 50000 we will examine RMSE as a function of *steps*. Results can be seen in the Figure 11. We can see that the best results can be obtained for more than 35 steps.

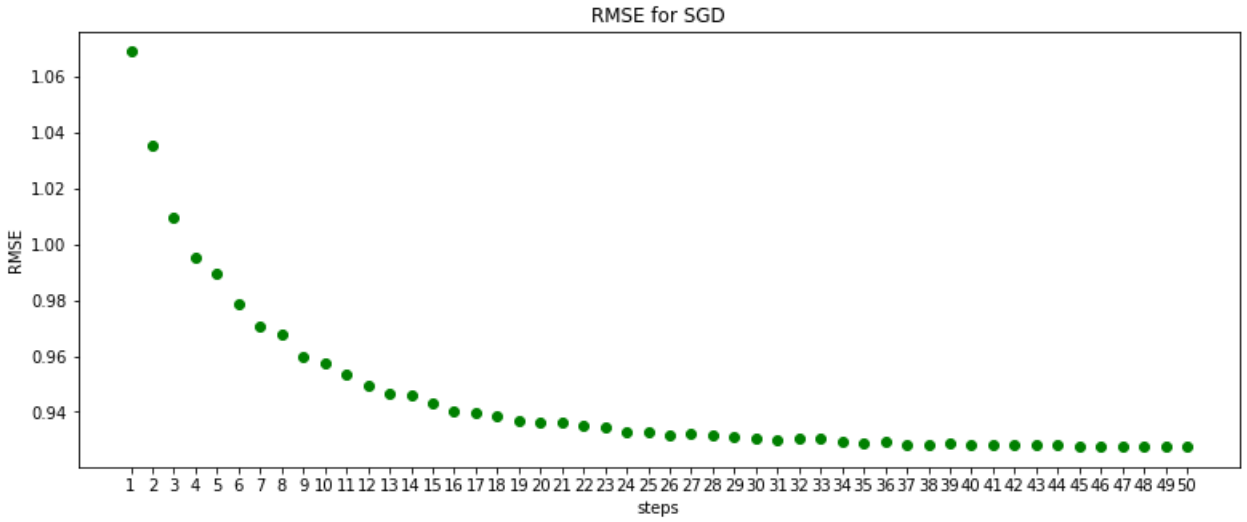

Figure 11: Average RMSE across all splits as a function of *steps*, $r = 7$, $\lambda = 0.00001$, $iterations = 50000$

Now we consider the optimal value for *learning rate*. We obtain the lowest average $RMSE$ across all splits for $learning\ rate = 0.007$.

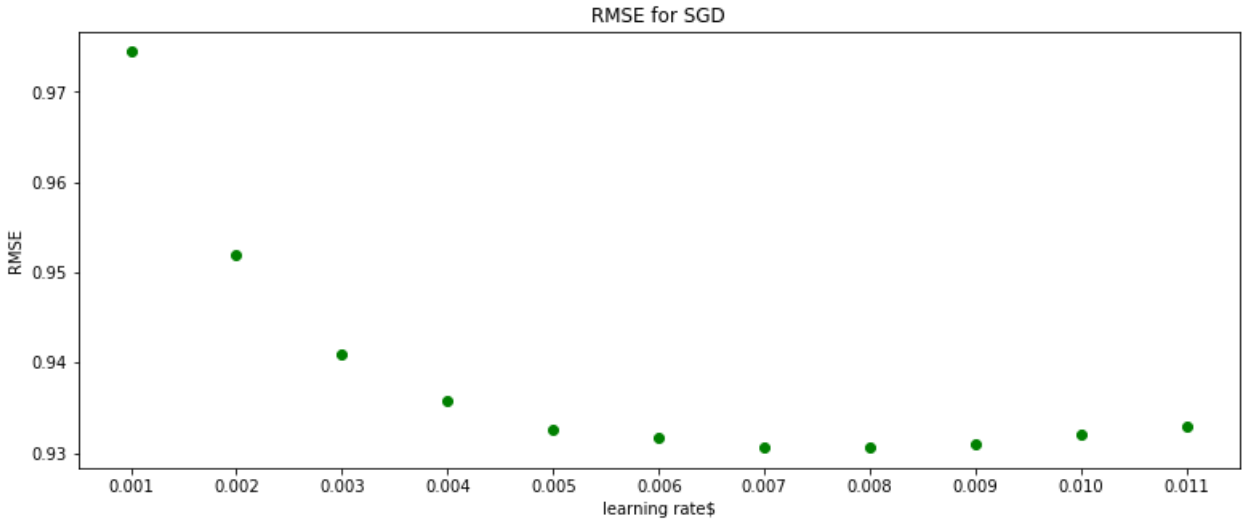

Figure 12: Average RMSE across all splits as a function of *learning rate*, $iterations = 10000$, $r = 7$, $\lambda = 0.00001$, $steps = 35$

The initial value of *learning rate* is the one with start with. As mentioned at the beggining of this description we will modify *learning rate* after performing each step. The previous value will be mulitiplied by $factor$. Now we search for optimal $factor$ value.

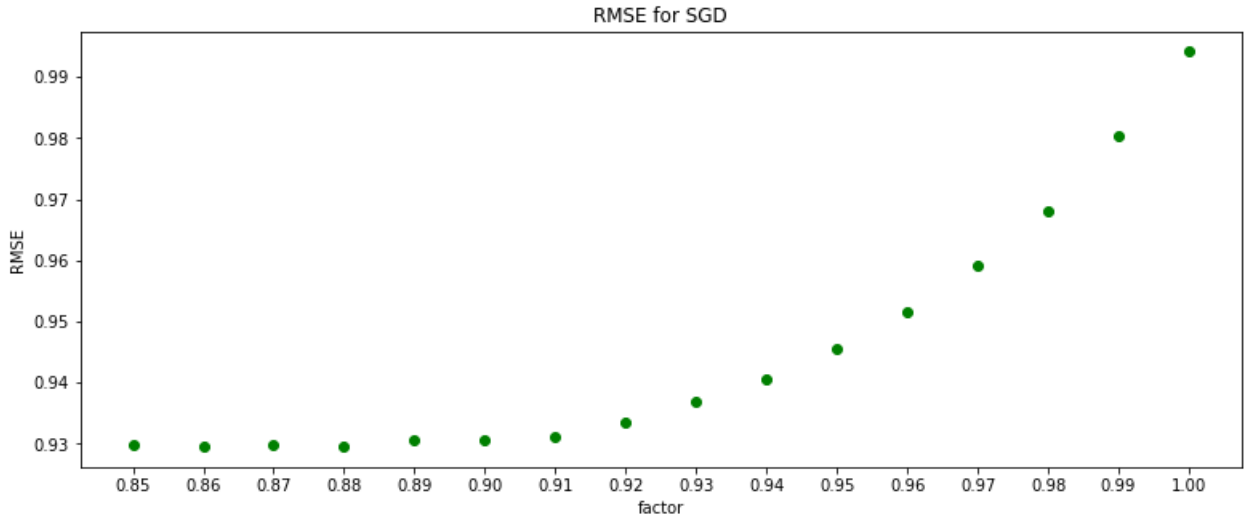

Figure 13: Average RMSE across all splits as a function of $factor$, $iterations = 10000$, $r = 7$, $\lambda = 0.00001$, $learning\ rate = 0.007$, $steps = 35$

As can be seen in the Figure 13 the best results are obtained for $factor = 0.88$. The last parameter we set is the number of components r. As the Figure 14 shows the optimal r is equal to 1.

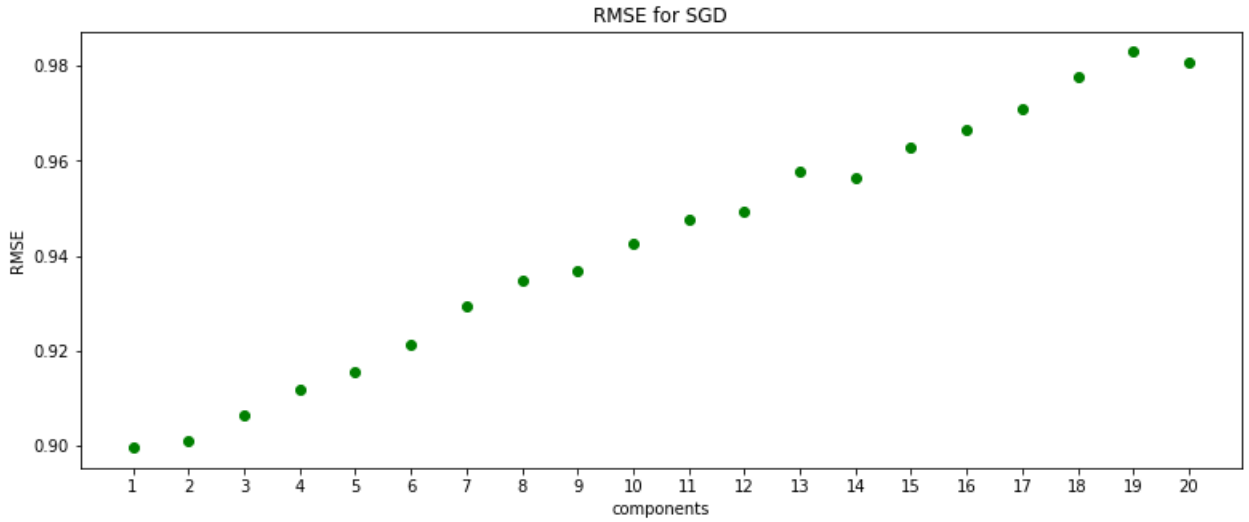

Figure 14: Average RMSE across all splits as a function of $components$, $iterations = 10000$, $r = 7$, $\lambda = 0.00001$, $learning\ rate = 0.007$, $factor = 0.88$

Summing up, our simulations led us to the following setting that should result into optimal outcome. We take:

- $r = 1$,
- $iterations = 50000$,
- $steps = 35$,
- $\lambda = 0.00001$,
- $learning\ rate = 0.007$,
- $factor = 0.88$.

With parameters defined as above we obtain average RMSE across all splits: 0.897860847. The best one was 0.891011696.

There are couple extensions of SGD that are worth mentioning, however in our problem they won't improve our results much:

- SGD with momentum
  Stochastic gradient descent with momentum is an SGD alternative which remembers the update Δw at each iteration and determines the next update as a linear combination of the gradient and the previous update:

  $$\Delta w := \alpha \Delta w - \nu \nabla Q_i(w)$$

  $$w := w + \Delta w$$

  It leads to general equation:

  $$w := w - \nu \nabla Q_i(w) + \alpha \Delta w,$$

  where the parameter w which minimizes $Q(w)$ is to be estimated, ν is a learning rate and α is the exponential decay factor between 0 and 1 that determines the relative contribution of the current gradient and earlier gradients to the weight change.

- Averaged stochastic gradient descent (ASGD)
  In ASGD, in each iteration of the algorithm, the parameter updates are aggregated with previous updates, creating a "moving average". As a result, the model parameters are updated based on more stable and representative data, allowing for faster convergence to an optimal solution.

  The one iteration can be expressed as:

  $$w := w - \frac{\alpha}{n} \sum_{i=1}^{n} v_i^k,$$

  where on each step we set $v_i^k = \nabla f_i(w^k)$ for all i.

- Implicit updates (ISGD), AdaGrad, RMSProp, Adam, Backtracking line search, Second-order methods.

References

[1] https://files.grouplens.org/datasets/movielens/ml-latest-small.zip

[2] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html

[4] https://github.com/mszczepanskigit/Methods-of-classification-and-dimension-reduction

[5] https://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/book-chapter-4.pdf

[6] https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/

[7] https://www.ibm.com/topics/gradient-descent

[8] https://realpython.com/gradient-descent-algorithm-python/#stochastic-gradient-descent-algorithms

[9] https://tinyurl.com/4xyj7cm5

[10] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[11] https://www.cs.ubc.ca/~schmidtm/Courses/540-W18/L11.pdf