Report from project 1

Magdalena Buszka & Martyna Firgolska

April 2022

Contents

1 Recommendation systems - motivation

Recommendation systems are frequently used as a way to effectively choose data presented to users - nowadays there are just too many options for an individual to consider them all. They can be used as a way to improve costumer satisfaction, make users stay on a platform longer (YouTube, Netflix), market products to users more likely to buy them (Amazon) or to reduce costs of logistic operations such as storing and shipping products.

1.1 Problem

We are solving a problem of predicting user rating of an item (movie). The data is very sparse, there are many movies, but users usually rate only few of them. Intuitively we would want to find similarities between users and movies and use them to infer rating for new pair user-movie. The resulting prediction can be later used to develop recommender system by finding items with highest rating among items not yet rated by this user. The prediction is not a recommender system by itself, but it is easier to measure its performance (for example using RMSE, see Evaluation), than to measure effectiveness of recommender system.

1.2 Dataset

We are using movielens data available at https://grouplens.org/datasets/movielens/ , but code used to compute the results can be used on any data that has the same format (provided that the size of data would not exceed pandas and numpy capabilities). For our project we are only using data in file ratings.csv which contains data about ratings of movies made by user. Each row of the file consists of userId, movieId, rating and timestamp. The methods described below use only first three columns of data and omit information about time.

1.2.1 Notation

By X we denote the matrix with columns representing score values for single movie and rows representing score values for single user. The entries which are not present in the table from which we create X are treated as missing data. We denote by J the set of indices for which we have missing data.

2 Evaluation

We train our models using train set and then evaluate it using test set. The sizes of train and test sets can be arbitrarily chosen, but in our project to evaluate different methods we chose train set with approximately 90% of ratings for each user and test set with approximately 10% of ratings for each user. Let X, $X^{(t)}$, $\tilde{Z}$ denote the train, test matrices and the output matrix with predictions given by the model respectively and let T denote the set of indices where the test matrix has non-missing data. We evaluate models based on their RMSE (root mean square error) defined as below:

$$RMSE = \sqrt{\frac{1}{|T|} \sum_{(j,k) \in T} \left(\tilde{Z}_{j,k} - X^{(t)}_{j,k} \right)^2} \tag{1}$$

The smaller the RMSE the better the model predicted the ratings of the test set given the information from the train set.

3 Methods

The methods described below are all matrix factorization methods - methods that approximate rating matrix $Z \in \mathbb{R}^{n \times d}$ as a product of 2 or more matrices of with lower dimensions.
We can think that matrix factorization methods attempt to 'learn' r latent factors which describe user and movie, such that the rating is the dot product of user and movie factors.

3.1 SVD

Let Z be a real $n \times d$ matrix. The matrix Z can be written as product of three matrices $Z = UDV^T$ where:

- U is ortonormal (that is $U^T U = I$) $n \times d$ matrix
- V is ortonormal $d \times d$ matrix
- D is diagonal matrix with non-negative entries

Moreover the entries from the diagonal of D, d_{ii} are square roots of the eigenvalues of ZZ^T, sorted descending (that is $d_{i,i} \geq d_{i+1,i+1}$), columns of U are normalised eigenvectors of ZZ^T and columns of V are normalised eigenvectors of $Z^T Z$.

Given such decomposition we can approximate Z by the product of truncated matrices U_r, D_r, V_r. Where U_r denotes the submatrix of matrix U consisting of r first columns of U, analogically for V and V_r, and D_r denotes the square $r \times r$ submatrix of D consisting of first r rows of first r columns od D. We have $Z \approx \tilde{Z}_r = U_r D_r V_r^T$, and $\|Z - \tilde{Z}_r\| = \sum_{i=r+1}^{d} \lambda_i$. Where λ_i is the i-th biggest eigenvalue of ZZ^T.

3.2 SVD1

We use SVD algorithm on filled matrix Z and predict the unknown values by the approximation from Z_r - the truncated SVD Z approximation.

3.3 SVD2

SVD2 is an iterative algorithm using SVD1. We start from matrix X, filled by some method to obtain full matrix Z. We then in each iteration perform SVD1 on the matrix which has original values of Z where we have data and the missing values are imputed with values returned by SVD1 performed in previous iteration. We do so until some stop conditions are met. The various stop conditions are discussed in section 3.3.1.

Algorithm 1 SVD2

1: $Z^{(0)} \leftarrow X$ filled by some method
2: $\tilde{Z}^{(0)} \leftarrow$ output of SVD 1 on $Z^{(0)}$
3: **while** stop condition not met **do**
4: $\quad Z^{(i)}_{j,k} \leftarrow Z^{(0)}_{j,k}$ for $(j,k) \notin J$
5: $\quad Z^{(i)}_{j,k} \leftarrow \tilde{Z}^{(i-1)}_{j,k}$ for $(j,k) \in J$
6: $\quad \tilde{Z}^{(i)} \leftarrow$ output of SVD 1 on $Z^{(i)}$
7: **end while**
8: **return** $\tilde{Z}^{(i)}$

By iteratively performing SVD1 we strive to minimise the importance of the initial fillings.

3.3.1 SVD2 stop conditions

We can consider various stop conditions that must be fulfilled for the algorithm to end

- **Small change**: We stop the algorithm when the change between the two next predictions is sufficiently small.
- **Rise of RMSE**: We split out train set into new train and validation (again with 90%-10%, stratified by user). We train only on new train and calculate the RMSE of the prediction taking the validation set as our test set. We stop the algorithm when the RMSE starts to rise instead of falling.
- **Set number of iterations**: We stop the algorithm under predetermined number of iterations.

3.4 NMF

Non-negative matrix factorization (NMF) is a matrix factorization method, which attempts to approximate non-negative matrix $Z \in \mathbb{R}_+^{n \times d}$ as a product of non-negative matrices $W \in \mathbb{R}_+^{n \times r}$ and $H \in \mathbb{R}_+^{r \times d}$.
$Z \approx WH$

3.5 Filling methods

We can consider various methods for initial filling of the matrix X, with missing data, to obtain full matrix Z. Since we impute values only for the missing data we have $X_{i,j} = Z_{i,j}$ for $(i, j) \in J$

- **Filling with constant**:

  The easiest imputation method would be to fill all the missing values with some constant chosen by us - f.e. 0.

- **Filling with a statistic of whole matrix**:

  Similarly to filling with constant we can fill all the missing values with one value based on all the non-missing data - for example the sample mean, median or mode.

- **Filling with statistic by row/column**:

  We can fill the missing values with different value depending on the row/column - such as the column/row mean, mode, median.

- **Filling with weighted statistic**:

  Next step after considering the columns/rows separately is filling the missing value with weighted mean of the value for imputation for its column and row - f.e. the weighted mean of column and row means, modes, medians. We fill the missing data at i-th row and j-th column with $z_{i,j} = (1 - w) \cdot m_{cj} + w \cdot m_{ri}$, where m_{cj}, m_{ri} are the statistics of j-th column and i-th row respectively and $w \in (0, 1)$ is some chosen weight.

- **Filling with samples from sample distribution by row/column**:

  Apart from filling with some predetermined value or a statistic based on the data, we can also fill the data by drawing random samples from some distribution. One idea is to draw the values to impute for each column/row from the sample distribution of all the non-missing values from this column/row.

Below we present the RMSE of predictions obtained just by filling the matrices, before any further matrix factorization is made as a baseline for further considerations. Comparing with this baseline will give us an idea of how good our methods are compared to just filling the matrix and making the prediction based on that. We obtain the best results for weighted mean with weight around 0.55. We can see that filling with zeroes gives us the worst result. We can also observe that mean is always the best of all the considered statistics and that row-wise statistics seem to fare better than column-wise ones. Of course we must remember that the fact that some filling method gave the best result prior to any matrix factorization is not a guarantee that it will give the best results in conjunction with other methods but we can suspect that it is worth to check it.

Filling method	RMSE
weighted mean with w = 0.55	0.894
row mean	0.941
column mean	0.967
row median	0.981
column median	0.991
whole matrix median	1.037
whole matrix mean	1.037
row mode	1.094
column mode	1.109
whole matrix mode	1.153
random sample from column distribution	1.297
random sample from row distribution	1.307
filling with 0	3.645

Table 1: Table of RMSE for matrix filled with various methods, before any matrix factorization

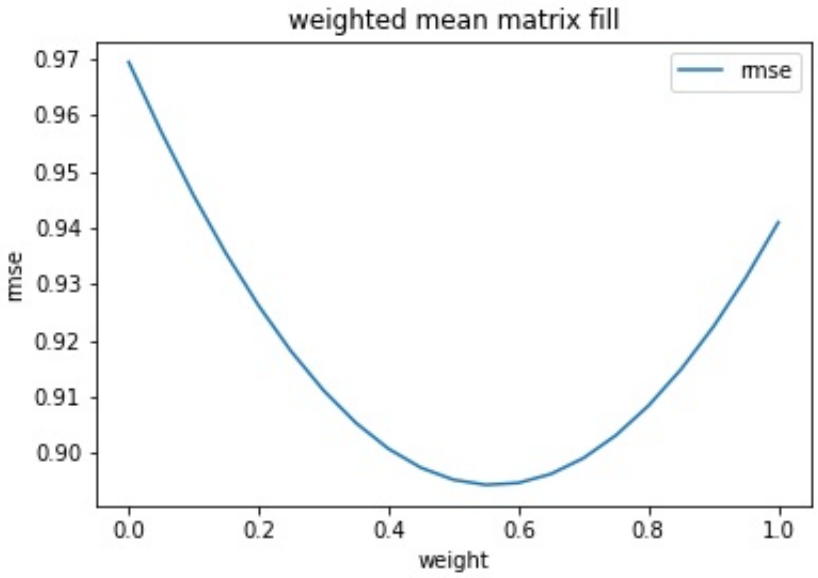

Figure 1: RMSE of weighted mean for different row weights

3.6 Choosing hyper-parameters

In each method we have some hyper-parameters which we have to choose. We will mainly decide by means of Cross-Validation and checking the mean RMSE.

3.6.1 SVD1

For each filling method we run 10-fold Cross-Validation to determine the best value of hyper-parameter r. We use the same seed for all the Cross-Validations which enables us to compare the results for different filling methods. For weighted mean we also consider 15 values of weights, equidistributed on $[0.1, 0.9]$.

We firstly run a few simulations to determine the rough interval to consider more closely and determine that we should consider $r \in \{2, 3, \ldots, 29, 30\}$. Filling with 0 gave the worst results with $RMSE > 2$ for all cases, filling with statistic other than mean also did not give us satisfactory results.

On figures 2, 3 and 4 we present the results of some of the methods of filling, including the best.

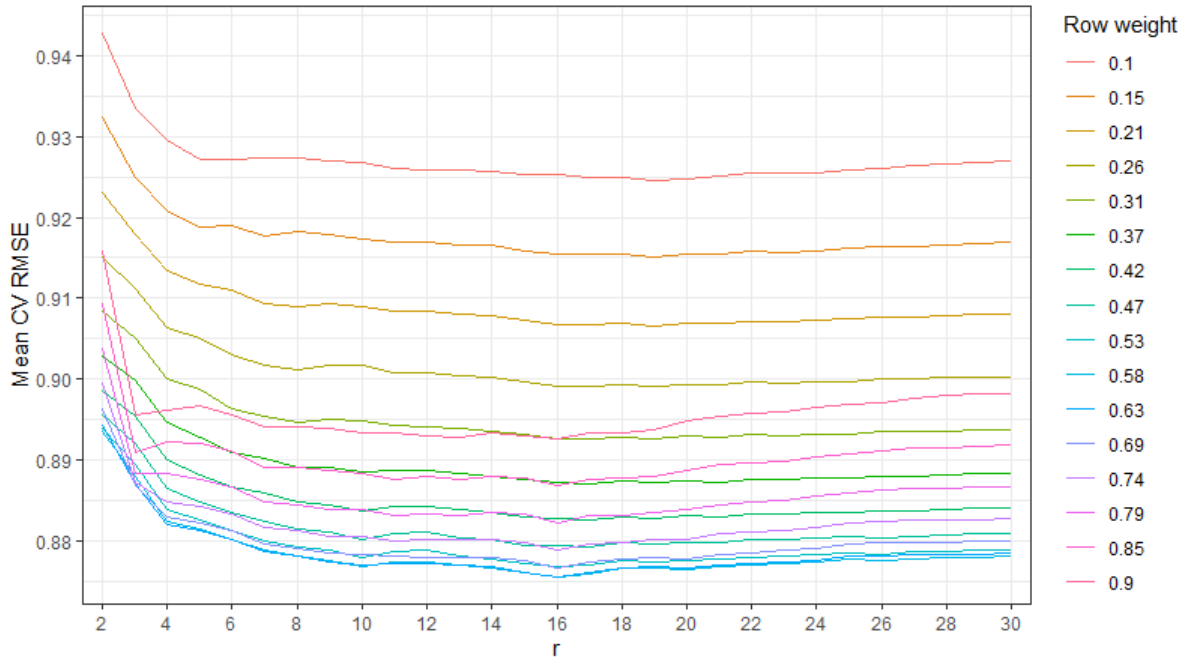

Figure 2: Mean RMSE for SVD1 various weights for filling with weighted mean of row and column means from 10-fold Cross Validation

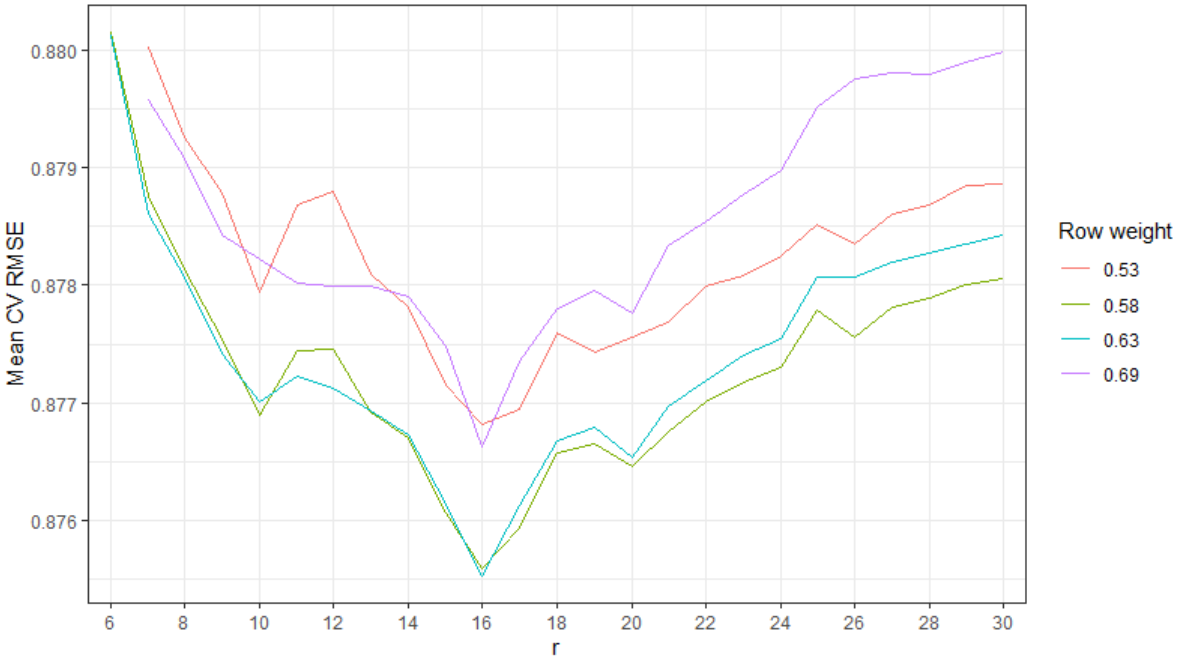

Figure 3: Closeup of figure 2

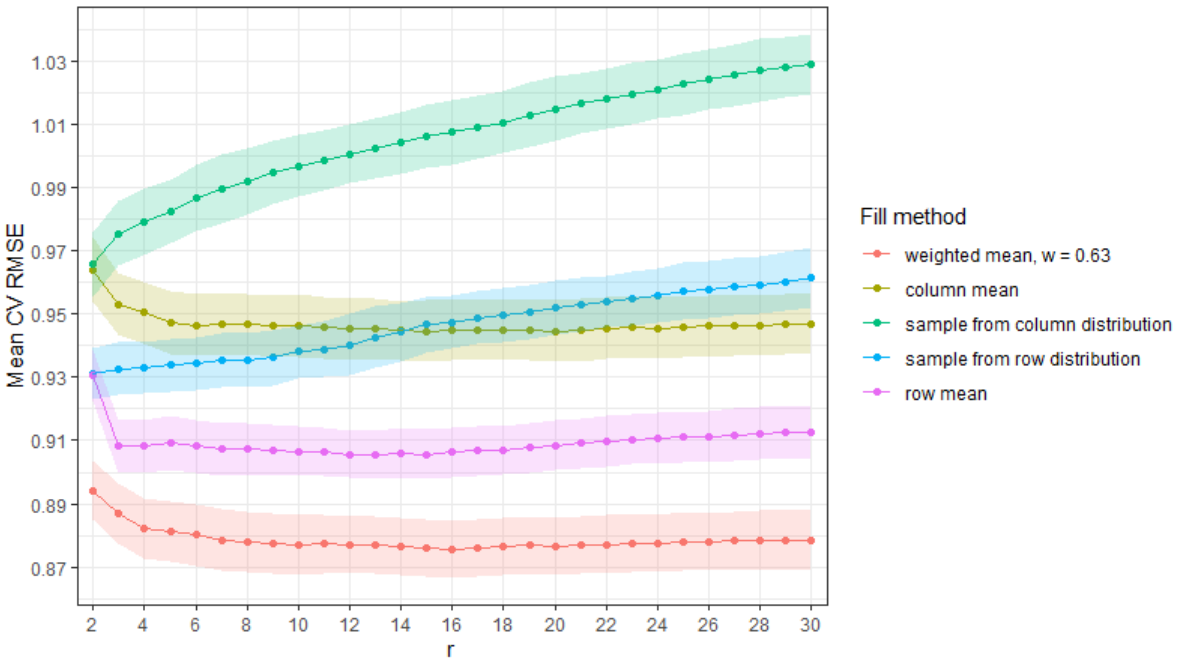

Figure 4: Mean RMSE for SVD1 of top methods of filling matrices for 10-fold Cross Validation with 2 standard deviation width confidence bands

As can be seen from figure 4 the best method for filling is weighted mean with row mean weight 0.63, column mean weight 0.37 and $r = 16$. The mean RMSE from Cross Validation is then approximately 0.875.

As we can see filling with values determined by row yields significantly better results than the same method applied on columns. From this we can suspect that the fact which user is giving the score is more important than which film is scored. But since the best weight for weighted mean is 0.63 we see that adding the information from the column increases our precision of prediction.

3.6.2 SVD2

After few initial runs on random splits and logging the RMSE values at each step we determine that using fixed number of steps or stopping when the results from consecutive iterations are close often leads to rise of RMSE after few initial steps. Optimization of both the step number/ margin of change and r on the same time would require vast amount of computations. We decide on using the stopping method which relies on computing the RMSE on validation set and stops when the RMSE starts to rise. By checking whether the next step still improves our result on the validation set we do not have to manually tune the number of steps optimal for each r or depend on the method convergence.

We use 5-fold cross-validation, with fixed random seed, for each of the filling methods to determine the best one and the best value of r. Again we first determine the sensible interval of r to consider - for SVD2 it is from 2 to about 30, after that the RMSE either plateaus or rises. For weighted mean we try 10 equidistributed weights from $[0.3, 0.75]$. As we can see from figures 5 and 6 the best weight is $w = 0.75$. From figure 7 we can see that the best filling method is weighted mean, just as it was in case of SVD1 . This time however the best value for r is 2.

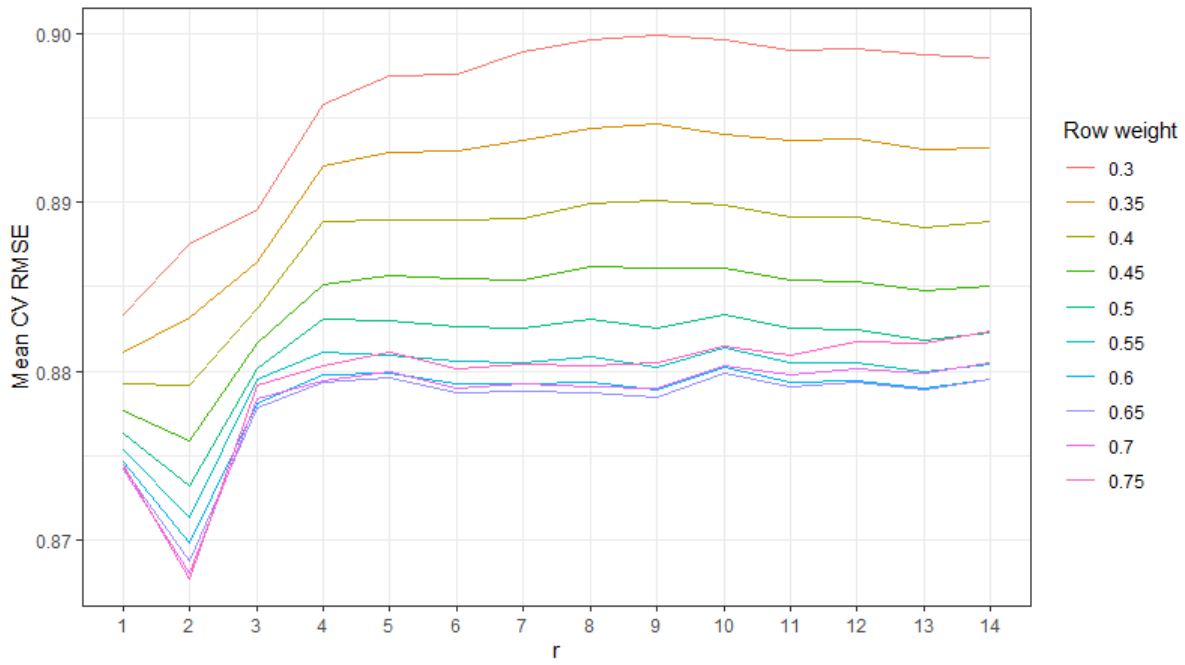

Figure 5: Mean RMSE for SVD2 various weights for filling with weighted mean of row and column means from 5-fold Cross Validation

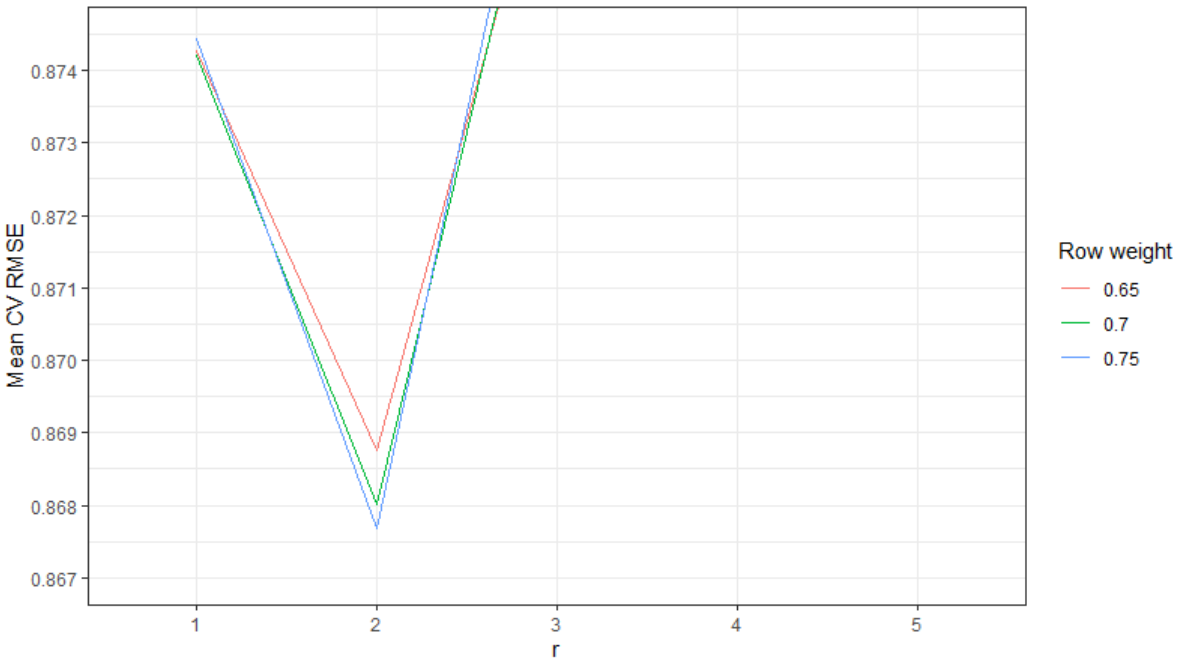

Figure 6: Closeup for 3 best weights from figure 5

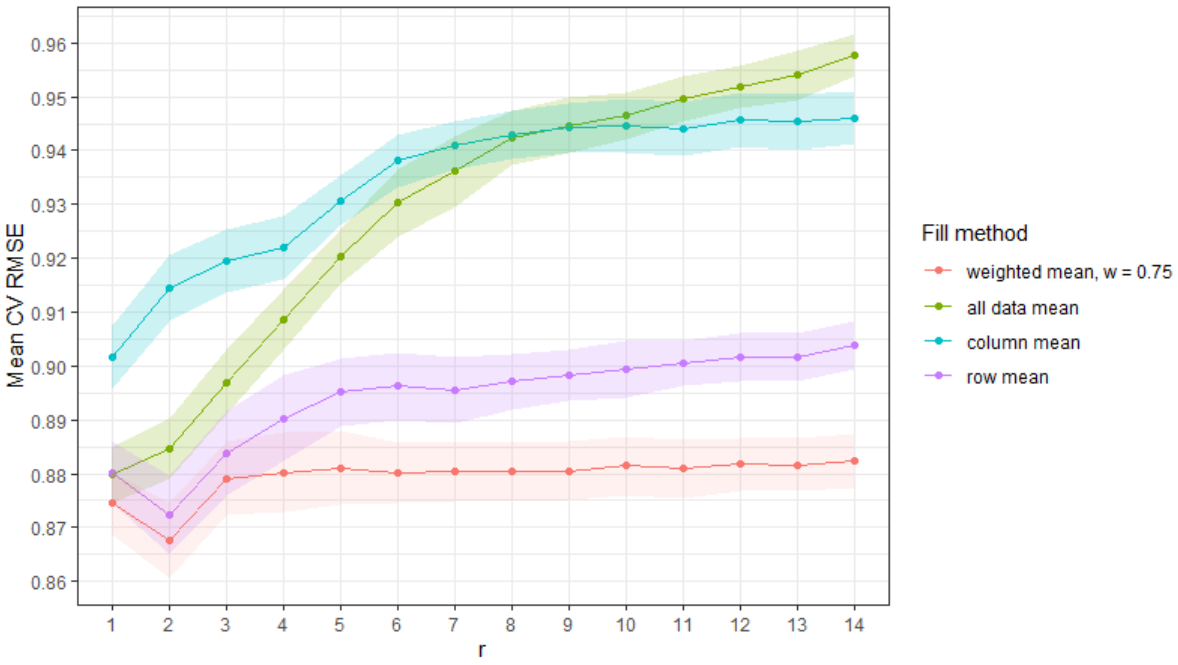

Figure 7: Mean RMSE for SVD2 of top methods of filling matrices for 5-fold Cross Validation with 2 standard deviation width confidence bands

3.6.3 NMF

We use Cross-Validation to determine the best value of hyper-parameter r. We use 5-fold CV as the algorithm takes significantly longer to run than SVD1. Again we use the same seed to be able to compare the mean RMSE between different filling methods.

This time we consider $r \in \{10, 11, \ldots, 29, 30\}$ after the initial check. The results differ from SVD1 but the weighted mean is again the best filling method, with row means having bigger weight than column means. This time however the best r is quite bigger. Since the best values of mean RMSE are for weight $w = 0.55$ at $r = 25$ and $r = 31$ we choose $r = 25$ hoping that using lower r will gives greater chance at generalisation.

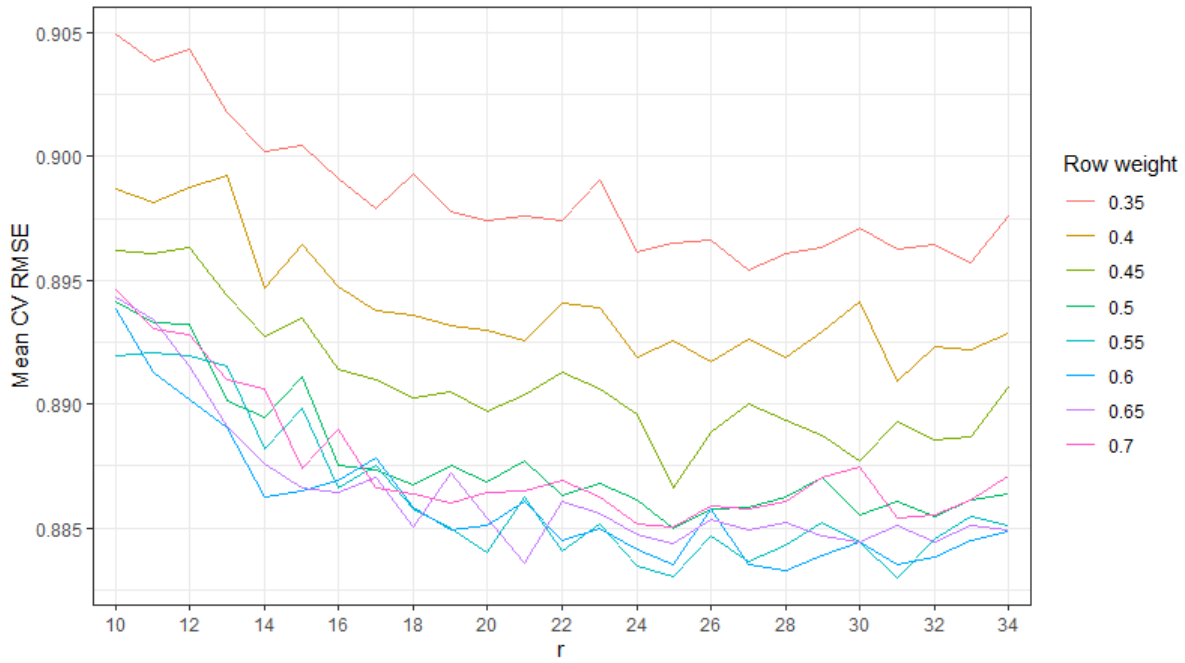

Figure 8: Mean RMSE for NMF various weights for filling with weighted mean of row and column means from 5-fold Cross Validation

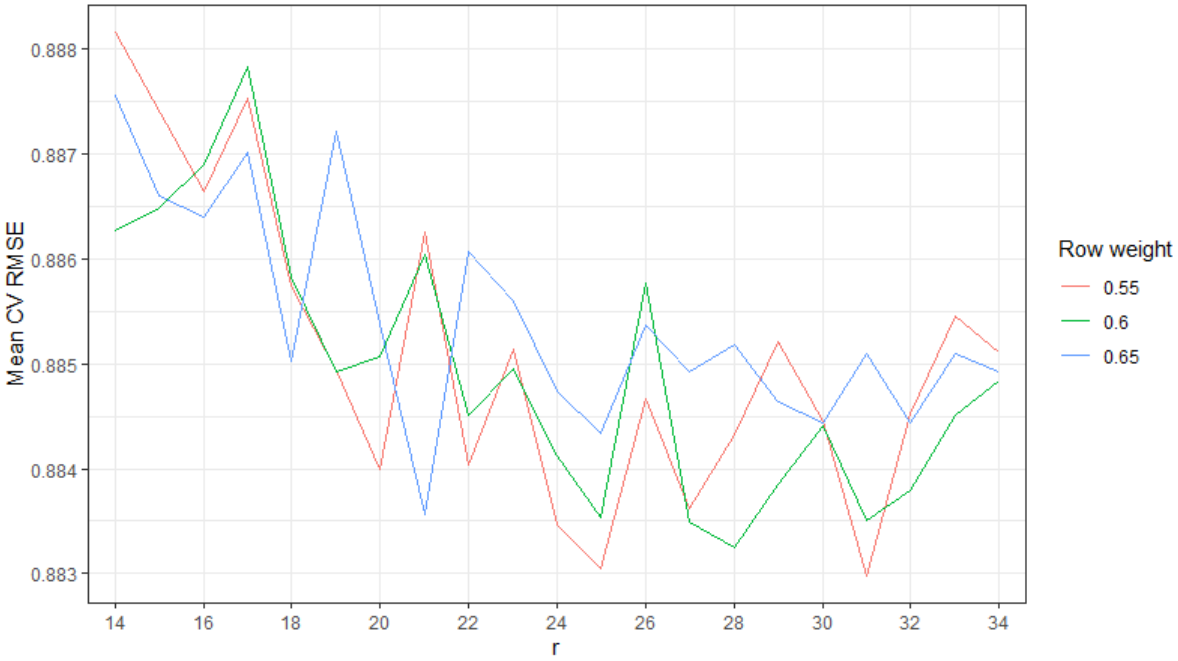

Figure 9: Closeup for 3 best weights from figure 8

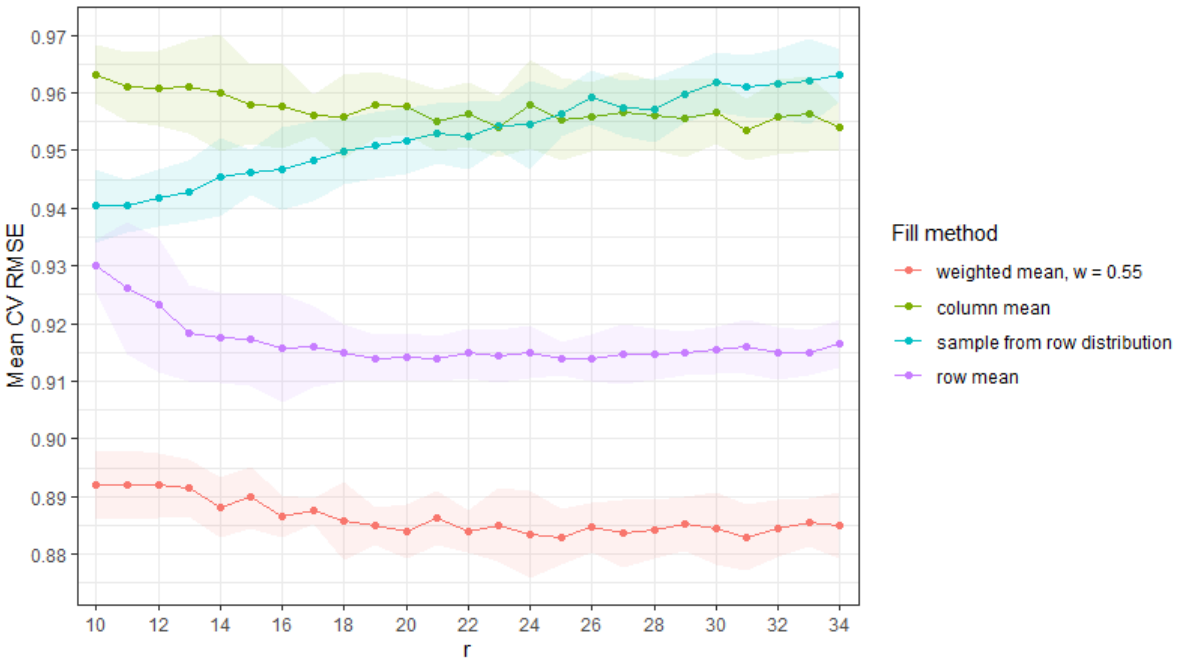

Figure 10: Mean RMSE for NMF of top methods of filling matrices for 5-fold Cross Validation with 2 standard deviation width confidence bands

3.7 General conclusions

After examining the mean RMSE results from our experiments performed to determine the best hyper-parameters for our methods we can come to some general conclusions:

- Looking at information from the same row gives us more precise predictions than the same information for column, but connecting information from both of them gives us better results, than only looking at rows
- Initial filling is a crucial step of the prediction - poor choice of initial filling method yields poor results even when we choose the best possible, for the parameters and method chosen, value of parameter r
- Considered methods reduce the RMSE in comparison to just initial filling, given that we choose the parameter r appropriately.

3.8 Stochastic Gradient Descend

Previous methods all needed full matrix Z to work, which left us with problem of filling unknown data. In this method we once again want to find matrices $W \in \mathbb{R}^{n \times r}$ and $H \in \mathbb{R}^{r \times d}$ such that WH is similar to Z on known coordinates. Formally we want to find W, H such that:

$$argmin_{W,H} \sum_{(i,j):z_{i,j} \neq nan} (z_{ij} - W_i^T H_j)^2 + \lambda(||W_i^T||^2 + ||H_j||^2)$$

Notice how this function is just MSE computed over known data. The λ parameter is the regularization constant, we use it in order to limit values of W and H matrices. W_i^T is the i-th row of W and H_j is j-th column of H matrix. For our problem we can think that they are respectively user and movie latent features. To find minimum of this function we will use stochastic gradient descend.

3.8.1 Gradient Descend

Gradient descend method is based on assumption that loss function is continuous. Let's assume that function $f(x)$ is continuous loss function, and we are trying to find its minimum. We can choose some point x and compute derivative (gradient in case of x being a vector) in this point. Gradient is a direction where function grows fastest, so if we want to find a minimum we need to move in the opposite direction.

$$x_{new} = x - \alpha \frac{\partial f(x)}{\partial x}$$

$\alpha > 0$ is a parameter called learning rate, which decides how big are the steps we are taking. We iteratively follow this procedure for the new x until we are satisfied with result (usually when gradient approaches 0, which means we are in local minimum).
This method is prone to get stuck in local minima. To avoid this it is important to carefully tune learning rate. We can also run the algorithm several times with different starting points and choose the best result. For some applications we are satisfied with good enough local minimum.

3.8.2 SGD

When the loss function depends on large training data then computational cost of computing it is large, and grows with size of dataset. It is not efficient to compute gradient on entire dataset for each step. Instead we will compute gradient for one sample and update x accordingly. Then we will choose different sample and repeat. After many steps we will still move in overall direction of negative gradient. This way we can make more steps in less time and the speed of the algorithm is not affected by dataset size so much.

3.8.3 Batches

Instead of using one sample to compute gradient we can use several samples per step. The subset of data used to compute single step of the algoritm is called a batch. Usually the trainset is first permuted and split into batches of identical sizes. Then the algorithm computes steps for batches, effectively going over the whole trainset. A cycle over all batches of trainset is called epoch. This approach lets us to take steps in more accurate direction while still making the computation faster than GD.

3.8.4 Gradients

In order to use SGD we need to compute the gradient of loss function. We can compute gradient for a single sample z_{ij}. If we are using batches bigger than one sample the gradient will be a sum of gradients for all samples in batch. Recall that loss function for sample z_{ij} is:

$$Loss(W, H) = (z_{ij} - W_i^T H_j)^2 + \lambda(||W_i^T||^2 + ||H_j||^2)$$

We need to compute the gradient with respect to W and H. Notice how the function depends only on i-th row of W and j-th row of H. This means that gradient for other rows and columns is zero.

$$\frac{\partial Loss(W, H)}{\partial W_i^T} = 2 * (z_{ij} - W_i^T H_j) * (-H_j) + \lambda W_i^T$$

$$\frac{\partial Loss(W, H)}{\partial H_j} = 2 * (z_{ij} - W_i^T H_j) * (-W_i^T) + \lambda H_j$$

Having computed gradients the algorithm is as follows:

Algorithm 2 SGD

$W, H \leftarrow$ initialized with random values
while stop condition not met **do**
  for Sample in PermutedTrainset **do**
    $User, Item, Rating = Sample$
    $Error \leftarrow Rating - W_{User}^T * H_{Item}$
    $W_{User}^T \leftarrow W_{User}^T - \alpha(-Error * H_{Item} + \lambda W_{User}^T)$
    $H_{Item} \leftarrow H_{Item} - \alpha(-Error * W_{User}^T + \lambda H_{Item})$
  end for
end while
return WH

In the algorithm we do not need to create matrix Z or fill it in any way, we can get rating data directly from train set. Notice that we are omitting constant 2 in gradient computation - it is 'hidden' in α parameter.

3.8.5 Learning Rate

It is crucial to choose a good learning rate. Too small will lead to slow convergence or getting stuck in a local minimum. Too large and algorithm will be unable to converge instead jumping over all minimums. To better tune this parameters sometimes learning rate changes during the duration of algorithm instead of it being a constant. One of the basic learning rate schedule is to start with bigger alpha and reduce it after every x batches/epochs. This way we can converge faster at the beginning and explore search space and refine our answer better when we are nearing the end.

3.8.6 Velocity/Momentum

Consider quadratic function such as $x^2 + 30y^2$ Gradient is $(2x, 60y)$. For point $(x, y) = (4, 0.5)$ we would like to make steps in direction of point $(0, 0)$ - a bigger step in x coordinate and small in y coordinate. The gradient has y coordinate much bigger than x coordinate. The effect is that we are not going in straight line toward minimum but we are oscilating a lot.

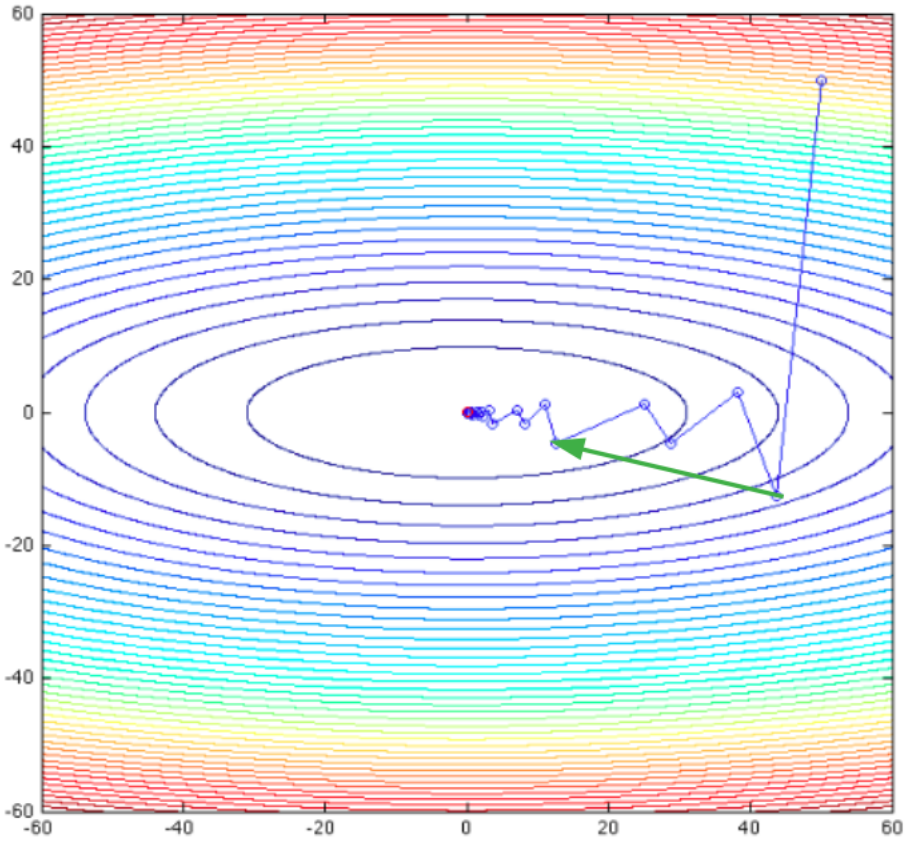

Figure 11: Graphical intuition as to way the momentum may help us

The way to help with this problem is to introduce momentum. Instead of using gradient of current point to update it we are using momentum - weighted exponentially average of past gradients. To implement it we just compute gradient g and keep momentum variable m(initialized with 0). Then we update $m \leftarrow -g + m/2$, $x \leftarrow x + m$.

3.9 SGD hyper-parameters selection

3.9.1 Preliminary tests

To select best hyper-parameters for the algorithm we run preliminary test on a single split of data in order to gain some intuition about them.
For r values the best results were for r between 10-20 with RMSE on testset around 0.88 (best for r=11 with RMSE=0.876)

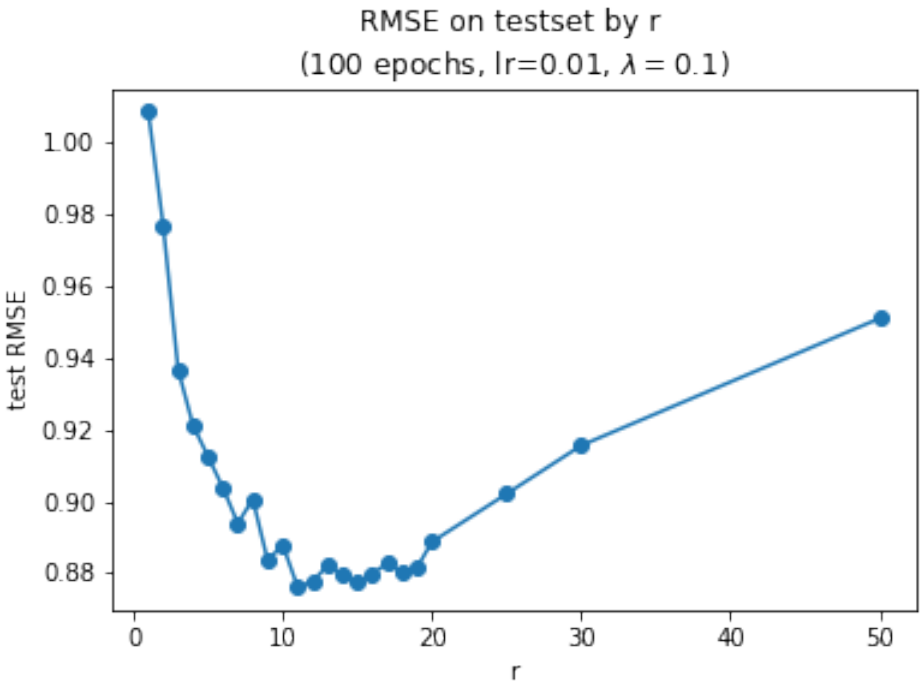

Figure 12: Preliminary testing of the SGD parameters - r values

Learning rate controls how big are the steps we are taking. Learning rates above 0.1 resulted in overflow error at the beginning of the algorithm, due to high value of gradient at this stage. The best values are around 0.005 - 0.01, with min RMSE = 0.872 in lr = 0.01. It seems that lower values would not give much advantages, and take longer to converge.

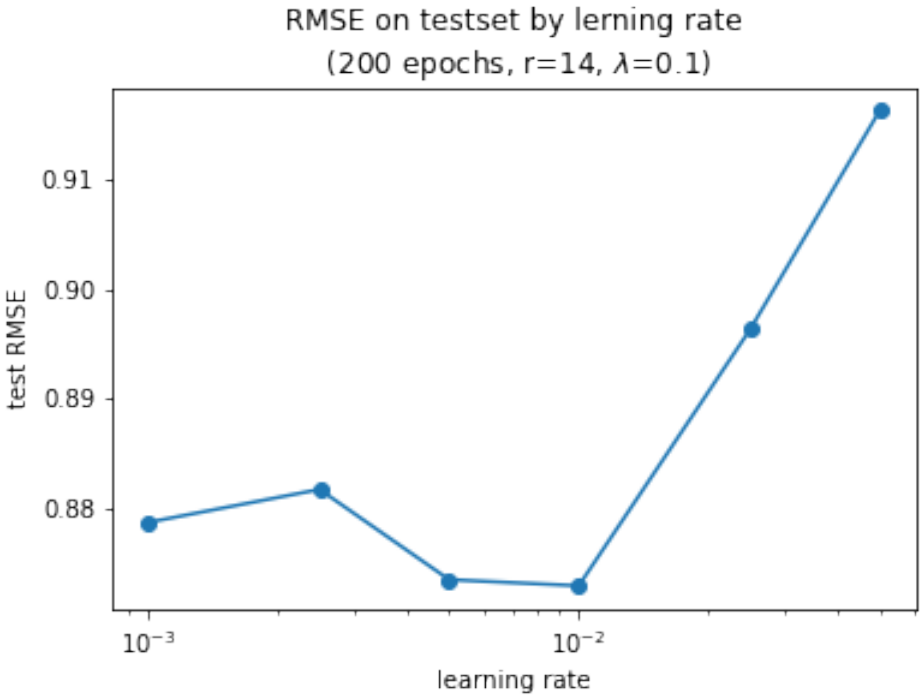

Figure 13: Preliminary testing of the SGD parameters - learning rate values

The lambda parameter is used to regularize W and H matrices. On a plot below we can see that higher values of λ lead to smaller differences between test and train RMSE, however larger values may prevent sgd from computing better results or take longer to converge. Lambdas between 0.1 to 0.2 gave best results. The best was $\lambda = 0.15$ with $RMSE = 0.874$.

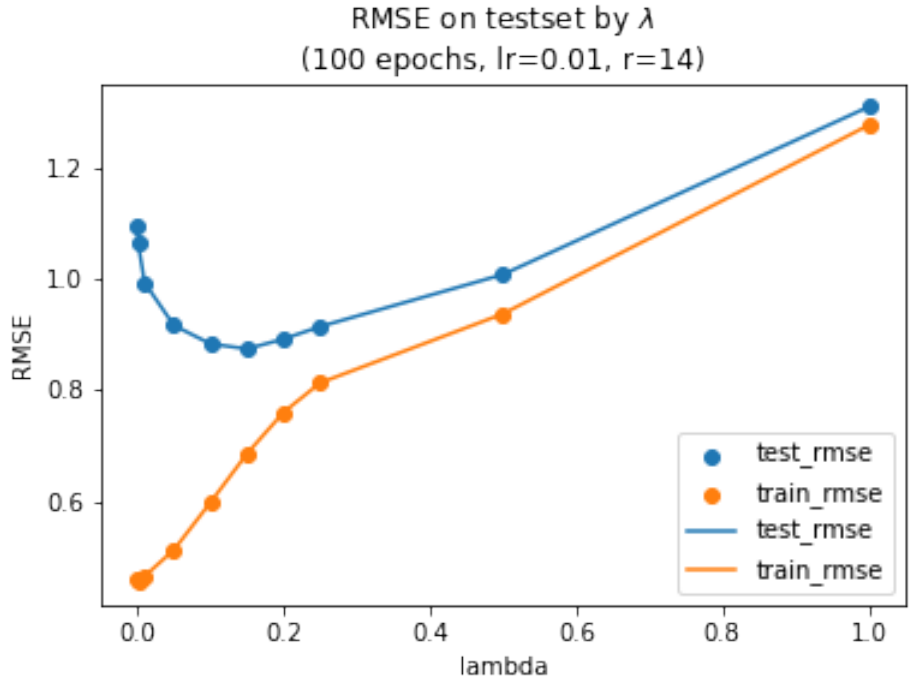

Figure 14: Preliminary testing of the SGD parameters - λ values

3.9.2 Repeated Tests

In order to get more reliable results we repeated computation on 5 different data splits with hyperparameters in narrower ranges (best ranges from preliminary tests).
The best lambda is 0.1 with 0.15 and 0.075 close behind, as is shown on figure 10.

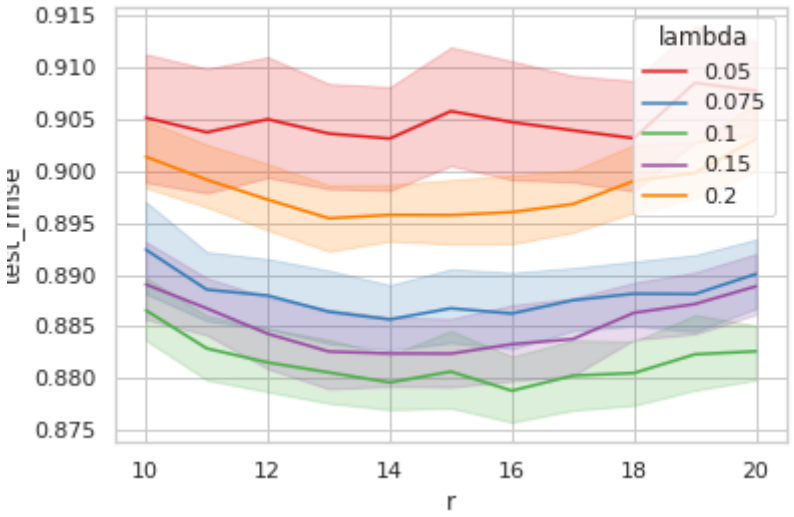

Figure 15: Mean RMSE for different λ with respect to r taken over all tested learning rates and data splits

It is less clear which learning rate is best just from plotting mean for each learning rate, but it should be between 0.005 and 0.0025.

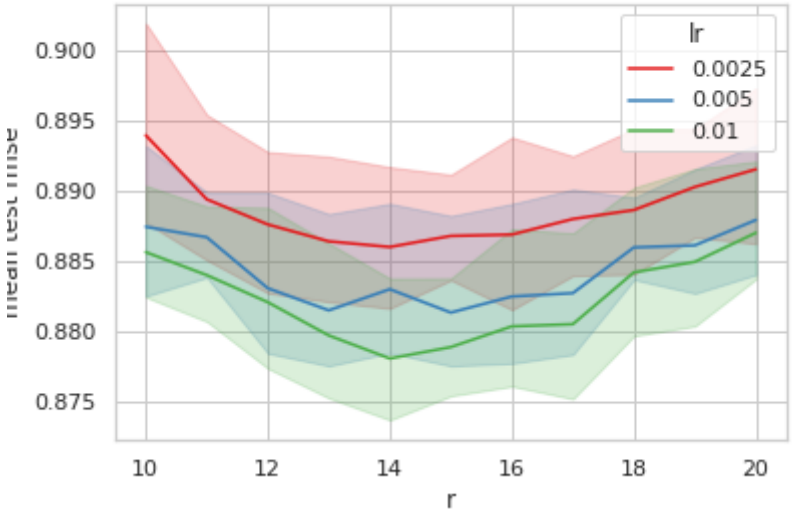

Figure 16: Mean RMSE for different learning rates (lr) with respect to r taken over all tested λ and data splits

But if we consider learning rate for a specific λ the winner is 0.005 along with $\lambda = 0.1$.
The r values from 14 to 16 do not give much difference in RMSE. For $\lambda = 0.1$ and learning rate $= 0.005$ we chose r=15. This combination results in mean test RMSE around 0.877.

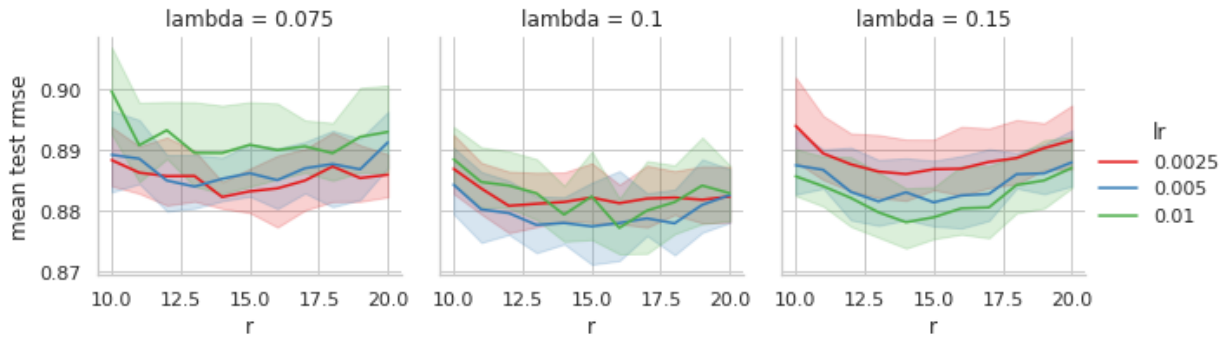

Figure 17: Mean RMSE for different lambdas and learning rates with respect to r taken over all tested data splits

4 Results

Below we present box plots for RMSE calculated on 100 random splits of data into 90%-10% train-test for each method. The hyper-parameters chosen are the ones which gave the best results as discussed in 3.6. All of the methods improve predictions in comparison to just filling the matrix with weighted mean (the best result for just filling the matrix). We can see that SVD2 gives us the best results - 0.863 on average. All of the methods have similar interquartile range - about 0.0075.

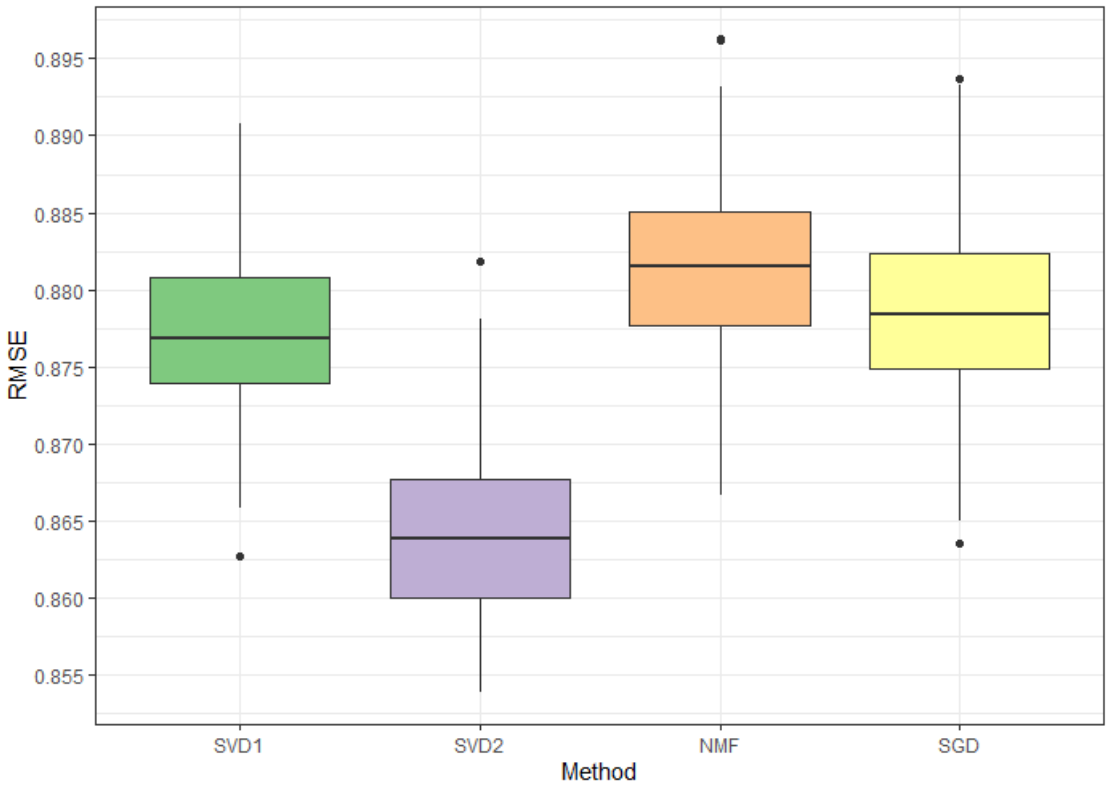

Figure 18: Boxplot of RMSE for different methods for 100 random splits