Data Mining Project

Team: PieManuCat

Pietro Vigilanza (SN: 4930770, kaggle username: PieManu, kaggle display name: PieManu),
Cătălin Lupău (SN: 5042143, kaggle username: catalinlupau, kaggle display name: Catalin Lupau)

A report presented for the Bsc. Computer Science and Engineering
Data Mining Course

Electrical Engineering, Mathematics and Computer Science
TU Delft
the Netherlands

I. Introduction

The aim of this project was to create a movie recommendation system based on a dataset containing the ratings of 6040 netflix users on 3706 movies.

Our training data consisted of 3 datasets, containing data about **movies** , **users**, as well as user **ratings** for those movies. The movies dataset contained the **year** and the **title** for each movie; the **users** dataset contained the **gender**, **age**, and a number encoding their **profession**; most importantly, the **ratings** dataset contained the ratings some **users** gave to certain **movies**.

- ***Collaborative Filtering*** - Cătălin Lupău
- ***Latent Factor Decomposition*** - Pietro Vigilanza

II. Methodology

A. General Overview

As specified above, various algorithms were implemented and tested in order to obtain a good predictor. Our approach was based on the assumption that combining the predictions of different methods will improve the overall performance of our model. We designed our system in such a way that we can assign weights to each individual 'prediction strategy' (i.e. prediction algorithm). Mathematically this was implemented by taking a weighted average of the predictions of each algorithm. The algorithms return a '0' value when they cannot make a prediction. When computing the final rating, this is accounted for by just ignoring '0' values.

$$\frac{\sum_{predictedRating \neq 0} weight \cdot predictedRating}{\sum_{predictedRating \neq 0} weight} \tag{1}$$

An important concept among the algorithms is the ratings matrix, defined as: R[i, j] = 'the rating given by user i to movie j'.

B. Collaborative Filtering

This class of algorithms is based on the idea of finding the k most similar rows/columns to a 'target' row/column for which we want to make predictions. We can then use these 'k most similar neighbors' to approximate the missing values in our 'target' row/column by taking their weighted average. Neighbors with missing values on the column/row we are trying to predict are avoided. The following 2 formulas illustrate how this method works, in either the user-user case (equation 2) or 'item-item' case (equation 3).

$$R[i,j] = \frac{\sum_{n \in neighborRows} sim(R[i,:], R[n,:]) \cdot R[n,j]}{\sum_{n \in neighborRows} sim(R[i,:], R[n,:])} \tag{2}$$

$$R[i,j] = \frac{\sum_{n \in neighborCols} sim(R[:,j], R[:,n]) \cdot R[i,n]}{\sum_{n \in neighborCols} sim(R[:,j], R[:,n])} \tag{3}$$

Various algorithms following this general approach were implemented, as an attempt to balance 'prediction accuracy' and 'runtime'.

1) Locality Sensitive Hashing based on Cosine Distance: The main motivation behind this algorithm was that our first attempt of computing the similarity matrix for our dataset was taking way too much time. Hence, we looked into data structures that would enable us to find the most similar 'users' / 'movies' in a faster way through hashing. Our implementation is based on the referenced article [Bog].

In a nutshell, our hashing procedure consists of generating m random planes each splitting our 'user' / 'movie' space into two. If a certain 'user' / 'movie' happens to be above the plane, we assign it a '1', otherwise we assign it a '0'. After doing this for each plane, we end up with an m-bit signature for each of our 'users' / 'movies'. These signatures make up our hashes and are used as keys in a locality sensitive hash table, enabling us to find neighbors efficiently. Note that, even though the locality sensitive hash always approximates the 'cosine distance', our implementation can be configured to make use of 'pearson correlation' for the neighbor weights when computing the predicted rating.

2) Naive Collaborative Filtering using Similarity Matrix: This algorithm simply follows the classical collaborative filtering method. Unlike the LSH method above, this algorithm computes the actual 'Pearson correlation similarity matrix' for the 'users' / 'movies'. Having learned from our first attempt at computing it, this time we applied a series of optimizations to speed up the computation of the matrix.

Computing a value inside the similarity matrix would normally consist of performing the following computation (again '0' values are simply ignored):

$$S[i,j] = \frac{\sum_k (x_k - \bar{x})(y_k, -\bar{y})}{\sqrt{\sum_k (x_k - \bar{x})^2}\sqrt{\sum_k (y_k - \bar{y})^2}} \tag{4}$$

where x is the ratings vector corresponding to the i-th 'user' / 'movie', and y the ratings vector for the j-th 'user' / 'movie'. the ratings vector corresponding to the i-th 'user' / 'movie'.

Our optimization consists of pre-computing the following quantities separately for each vector and re-using them when computing the actual similarities:

$$\sum_k (v_k - \bar{v}) \quad \sqrt{\sum_k (v_k - \bar{v})^2} \tag{5}$$

Once the 'user' / 'movie' similarity matrix is computed it is saved on disk and simply re-used whenever our program makes a prediction.

3) Collaborative Filtering with Global Baseline Estimate: This algorithm is essentially an improvement over the previous one. It still uses a precomputed user-user / item-item similarity matrix. The big difference, however, is that when making predictions it also includes global baseline estimates. In essence, the algorithm first makes a rough approximation of the rating by taking into account the global average rating adjusted by the user and the movie biases. This rough approximation forms the baseline of the prediction. The equation below represents the baseline estimate for user 'i' and movie 'j'.

$$baseline(i,j) = \mu + b_i + b_j \tag{6}$$

where each symbol represents:

$$\mu = \frac{1}{N}\sum_{i,j} R[i,j] \quad b_i = \overline{R[i,:]} - \mu \quad b_j = \overline{R[:,j]} - \mu \tag{7}$$

Collaborative filtering is then used to adjust the baseline with how much the closest k neighbors deviate from their own baselines. This lead to the following formulas (user-user and movie-movie) for the average deviation of the neighbors:

$$deviation(i,j) = \frac{\sum_{n \in neighborRows} sim(R[i,:], R[n,:]) \cdot (R[n,j] - baseline(n,j))}{\sum_{n \in neighborRows} sim(R[i,:], R[n,:])} \tag{8}$$

$$deviation(i,j) = \frac{\sum_{n \in neighborCols} sim(R[:,j], R[:,n]) \cdot (R[i,n] - baseline(i,n))}{\sum_{n \in neighborCols} sim(R[:,j], R[:,n])} \tag{9}$$

Regardless of whether we choose to compute the deviation using user-user or item-item collaborative filtering, the final prediction for the rating becomes:

$$R[i,j] = baseline(i,j) + deviation(i,j) \tag{10}$$

4) Collaborative Filtering with Agglomerative Clustering: This algorithms is partly based on the one described in chapter 9.3.3 of the book 'Mining of Massive Dataset' [Les]. Having said that, we have tried to speed it up as much as possible. In a nutshell, our algorithms performs the following 3 steps:

- makes use of agglomerative clustering to cluster the users based on their pearson correlation. The ratings of the users in each cluster are averaged out, hence obtaining a new ratings matrix R' with the same number of rows as formed user clusters. The number of columns remains intact for now.
- makes use of the same agglomerative clustering algorithm to cluster the movie columns in the newly defined ratings matrix R'. The ratings of the movies in each cluster are again averaged out, hence obtaining a second ratings matrix R'' with the same number of rows as R' and the same number of columns as the formed movie clusters.
- makes use of the reduced matrix R'' to make predictions. For example, if we want to predict the rating $R[i,j]$ of user 'i' and movie 'j', the algorithm finds the cluster C_i of user i and the cluster C_j of movie j. The predicted value is then $R''[C_i][C_j]$. In case $R''[C_i][C_j]$ is 0, user-user collaborative filtering is applied to the matrix R''

However, what is unique about our implementation of the algorithm are the speed optimizations we applied to make it usable for our problem. The most essential optimization is represented by the fact that we used a 'randomized agglomerative clustering algorithm'. Finding the 2 closest clusters is a very expensive operation, hence we chose to select a random sample of the clusters and merge the 2 closest clusters in the sample. Further optimizations refer to the steps we took to avoid re-computing similarity matrices, such as assumming that the similarity between the columns of the ratings matrix remains roughly the same after clustering the rows.

To summarize, this method can be seen as applying agglomerative clustering to compress the original ratings matrix R into a smaller matrix R'' which is then used to make predictions either directly or through collaborative filtering in case of missing values.

C. Latent Factor Decomposition

This class of algorithms took a substantially different approach from the collaborative filtering design discussed above. With this strategy, rather than finding the 'k most similar neighbors' for a given user, we try to predict the user's rating by revealing the 'latent factors' of the data. To do so, these algorithms aimed to factor out two matrices U and V from the original rating matrix R. The prediction matrix P for any user i and movie j is then generated from the result of multiplying U and V together. The prediction for a missing value $R[r,t]$ is then located in the corresponding prediction matrix entry $P[r,t]$. In other words we are trying to approximate this equality as much as possible:

$$R = UV = P \tag{11}$$

For this method, there isn't an algebraic approach to finding the optimal UV matrix, but rather we try to get as close as possible to an optimal solution by

the gradient descent technique. To do so we defined a cost function based on minimizing the *Root-Mean Squared Error* of the resulting $R - UV$ matrix.

For this subset of algorithms we implemented three variations with varying levels of success. There are five consistent aspects within the design of each model. These constants within the three implementations are detailed below:

- All models rely on the fact that the rating matrix R exists. This rating matrix is generated by lining up all users as the rows and the movies as the columns of the R matrix. Each value greater than 0 in this matrix is considered as an existing rating
- If the rating matrix R is of $n \times m$ dimensions, then matrix U is of $n \times d$ dimensions while matrix V is of $d \times m$ dimension. In this case, d becomes a parameter that we try to estimate in all three models.
- All models have a learning rate μ which specifies how fast the model moves towards a local minimum after calculating the gradient of the cost function.
- For the gradient descent process, we rely on *Stochastic Gradient Descent*. That means that instead of optimizing our UV matrix as a whole, we take small steps by slowly optimizing one row if U and one column of V at a time.
- Initially, U and V are randomly generate matrices with values ranging from 0 to 1 taken from a uniform distribution.

1) Simple UV Decomposer: The initial implementation consisted of simply reducing a simple cost function with gradient descent. Our cost function was the following:

$$C = \sum_{training} (r_{xi} - v_i u_x)^2 \quad \nabla \frac{\partial C}{\partial v_j} = 2(r_{ij} - v_j u_i)u_i \quad \nabla \frac{\partial C}{\partial u_i} = 2(r_{ij} - v_j u_i)v_i \tag{12}$$

To do this, we simply iterated over each defined r_{ij} term in the matrix and calculated the gradient of function.

This original RMSE implementation was decent, but it performance was rather low compared to the other two models. Due to the lack of regularizing parameters the model was affected by over fitting the data. This is because without regularization terms, the values of the U and V matrices have a lot of freedom to take risky choices for the sake of reducing the cost function given.

2) Regularized UV Decomposer : Regularized UV composition turned out to be the a huge step into the right direction. With some simple parameter tuning we were able to easily beat the results of our both the simple UV decomposition and the collaborative filtering algorithms.

The main difference with this approach is that we added regularization terms, which makes gradeint adjustments be more conservative. Hence this model contained two extra parameters - namely delta_1 and delta_2. delta_1 corresponded to the regularization term for the rows of the original matrix (the users), while delta_2 consisted of the regularization term for the columns of the original matrix (the movies). The equation used can be seen below:

$$C = \sum_{training} (r_{xi} - v_i u_i)^2 + \lambda_1 ||u_i||^2 + \lambda_2 ||v_j||^2 \tag{13}$$

$$\nabla \frac{\partial C}{\partial v_j} = 2(r_{ij} - v_j u_i)u_i + \lambda_2 v_j \quad \nabla \frac{\partial C}{\partial u_i} = 2(r_{ij} - v_j u_i)v_i + \lambda_1 u_i \tag{14}$$

The regularization terms were crucial and lead towards investigating new approaches to improve our score. Intuitively, the regularization parameters reduced overfitting since it punishes the cost function if either the U and the V matrix fit the data. This is because it favors smaller, and conservative predictions, rather than 'stretching' its values to fit data.

3) Biased-Regularized Decomposer: The Biased-Regularized UV decomposer is our most complex model since it included the most amount of parameters to tune. For this example we added a bias for the user and the movie and the mean of the matrix. This meant our model was able to spot certain biases - like a movie that is generally liked by all audiences.

The function we minimized was the following:

$$C = \sum_{training} (r_{xi} - (R_{avg} + b_i + b_j + v_j u_i))^2 + \lambda_1 ||u_x||^2 + \lambda_2 ||v_x||^2 + \lambda_3 ||b_i||^2 + \lambda_4 ||b_j||^2 \tag{15}$$

$$\nabla \frac{\partial C}{\partial b_i} = 2(r_{ij} - (R_{avg} + b_i + b_j + v_j u_i)) + \lambda_3 b_i \quad \nabla \frac{\partial C}{\partial b_j} = 2(r_{ij} - (R_{avg} + b_i + b_j + v_j u_i)) + \lambda_4 b_j \tag{16}$$

The gradient for u_i and v_j are identical, but with the changes made to adjust for biases and mean. This approach worked better since it leberaged more information from the data.

III. Results

A. Performance Evaluation

The metric we chose to evaluate our predictors with was the 'Root Mean Squared Error' (RMSE). To evaluate our models with different hyper-parameters and test our results we took a random sample of our data and randomly split it into a **training set** and a **validation set**. The split ratio was usually 80% training and 20% validation.

B. Collaborative Filtering

Due to the potentially large number of experiments, we chose to treat Item-Item and User-User collaborative filterings of the same types (i.e. LSH, Naive, etc.) as single predictors. This was achieved by considering the weights of Item-Item and User-User as a distinct parameter to be optimized. The optimization strategy consisted of trying different values for a certain parameter, while keeping the other parameters constant. The best parameter from a previous iteration would be kept constant, while optimizing the next parameter.

Across the different collaborative filtering models, the following hyperparameters were subject to optimization:

- ***weights*** - used in LSH, Naive and Global Baseline collaborative filtering algorithms to balance between Item-Item and User-User collaborative filtering
- ***k-neighbors*** - used in all collaborative filtering predictors to define the number of neighbors to be averaged when making a prediction
- ***similarity-measure*** - used in LSH collaborative filtering to specify which similarity measure should be used as neighbor weight (does not affect the hashing function)
- ***dimension-ratio*** - used in collaborative filtering with agglomerative clustering to define the fraction of the original size to which the ratings matrix should be reduced through clustering
- ***sample-size*** - used in collaborative filtering with agglomerative clustering to specify the number of random samples to be analysed when deciding which clusters to merge

Graphs presenting the RMSE scores of the different collaborative filtering algorithms for different parameter values can be found at the end of the report. An important aspect to point out is that the hyperparameter optimizations for the different collaborative filtering algorithms were performed on 40% of the available data. Hence the RMSE scores in these figures should only be interpreted relative to each other, but not as absolute measures of the performance of the predictors.

C. Latent Factor Decomposition

The following hyper-parameters were subject to optimization:

- ***d*** - the amount of "concepts" which determines how many columns U has and how many rows V has
- ***mu*** - the learning rate for gradient descent
- ***Iterations*** - max number of iterations
- ***lambda1*** - regularization parameter for rows of U
- ***lambda2*** - regularization parameter for columns of V
- ***lambda3*** - regularization parameter for user bias
- ***lambda4*** - regularization parameter for movie bias

D. Final Model

Our best individual models proved to be quite proficient at recommending user rating. Individually, our *Global User-User Collaborative Filtering* algorithm reached an RMSE of 0.91. The *Global Item-Item* improved this core substantially by reaching an RMSE of 0.88 in our tests. The best performing algorithm was the *Biased-Regularized UV Decomposer* which achieved its lowest RMSE score at 0.86 in our test.

Regardless of individual performances of algorithms, the best predictors were achieved by doing a "kitchen sink" approach with additional weight adjustment depending on the "quality" of the predictor. To achieve this we combined one instance from our best three models: *Global User-User CF*, *Global Item-Item CF*, and the *Biased-Regularized UV Decomposer*. The weights that gave us the best performance were as follows [User-User CF: 0.1, Item-Item CF: 0.4, UV: 0.5]. With the three well trained predictors and this set of weights our test were able to reach RMSE scores of 0.81.

IV. Final Discussion and Conclusion

Overall we can conclude that the algorithms implemented achieved very respectable results. Compared to the record result set by Belkor's Pragmatic Chaos team in 2008 of 0.856, our three simple models together achieved an RMSE of 0.843 when tested against the Kaggle dataset.

Even with this good score, we beleive that there was definitely room for improvement. For example, we didn't use all the data given, since we didn't take the year of the movie into consideration. With more time we could have also tried to expand the dataset with external, independent data such as adding movie genres for example.

V. Figures and Charts

See next page.

References

[Bog] BogoToBogo. *Locality-sensitive hashing using Cosine Distance*. URL: https://bogotobogo.com/Algorithms/Locality_Sensitive_Hashing_LSH_using_Cosine_Distance_Similarity.php.

[Les] Ullman Leskovec Rajaraman. *Mining of Massive Datasets*.

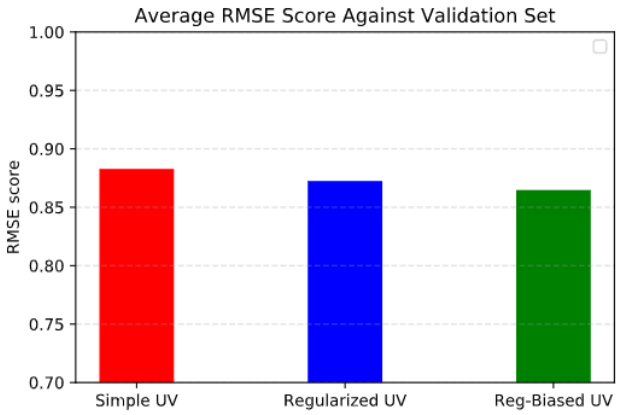

Fig. 1: RMSE score against validation set. Same learnig rate mu = 0.003. Regularization terms set at 0.009 for U and 0.011 for V. Bias terms are set at b_1 = 0.009 b_2 = 0.012. Bias for movies is higher since our experiments shows that this bias is more important. People are harder to predict.

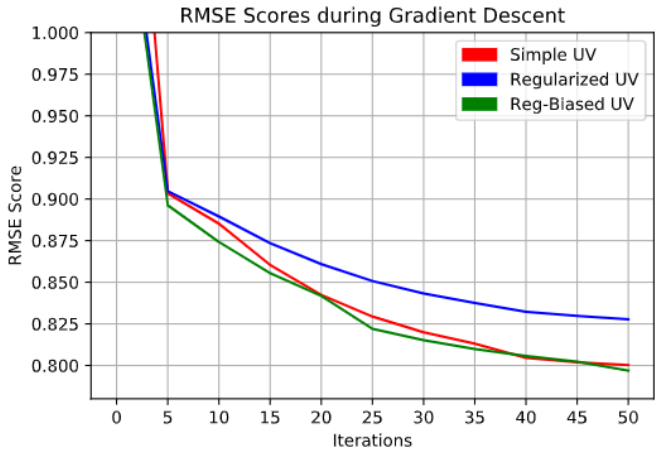

Fig. 2: Gradient Descent on three different models with mu = 0.003. Greater steps size resulted is a less optimal result. We see that simple UV achieves really god RMSE score, but this is due to greater overfitting. On graph to the left we see simple UV has worst performance. 50 iterations also showed to be a good enough for great RMSE scores

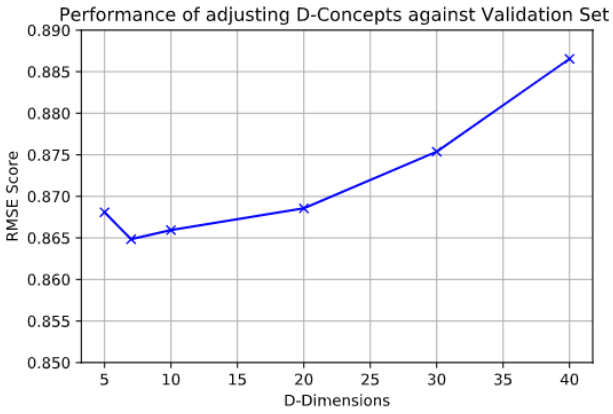

Fig. 3: Adjusting D-number of Concepts on Reg-Biased UV Model. All models seemed to be optimal when the number of D-dimensions was between 6 and 9. This showed that our data has around 6 to 9 big concepts. This aligns with the different movie genre types which are also between 6 and 10 big genres.

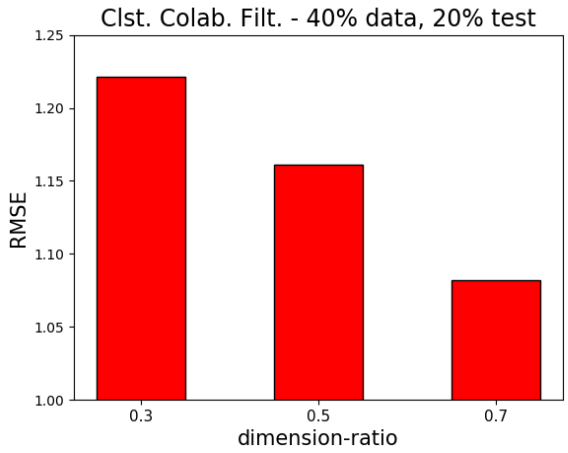

Fig. 4: Constant parameters: k-neighbors: 30, sample-size: 100. When applying a clustering based collaborative filtering algorithm lower compression rates correspond to higher accuracy. With shrunk matrix size, the algorithms becomes closer and closer to simply predicting the global average rating.

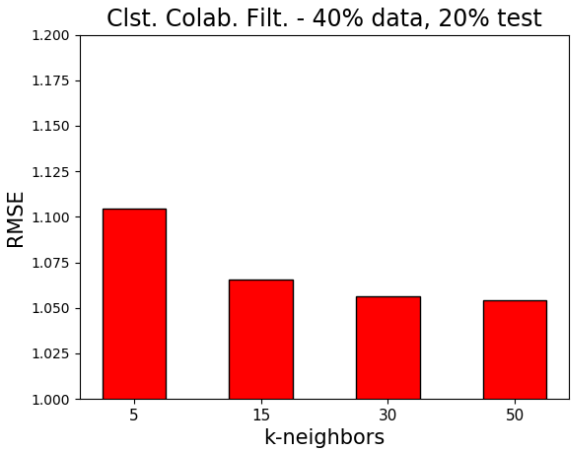

Fig. 5: Constant parameters: dimension-ratio: 0.7, sample-size: 100. In this figure we can see how the number of neighbors influences the clustering based collaborative filtering algorithm. Not using enough neighbors can lead to poor results.

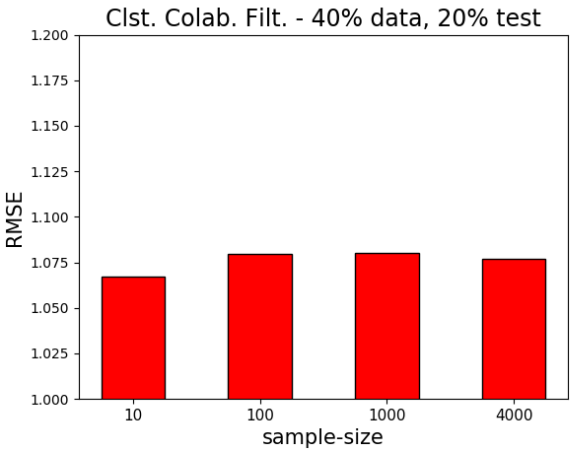

Fig. 6: Constant parameters: dimension-ratio: 0.7, k-neighbors: 50. This figure shows that the impact of the sample-size on the prediction capabilities of the clustering based collaborative filtering algorithm is quite small.

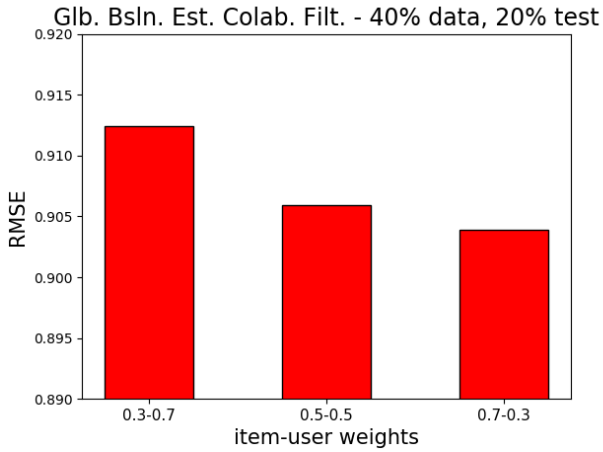

Fig. 7: Constant parameters: k-neighbors: 30. This plot shows the influence of using different weights when computing the average of the ratings predicted by item-item and user-user global baseline collaborative filtering algorithms. From the data, it becomes apparent that item-item collaborative filtering tends to perform better. This can be explained by the fact that items are less complex than users.

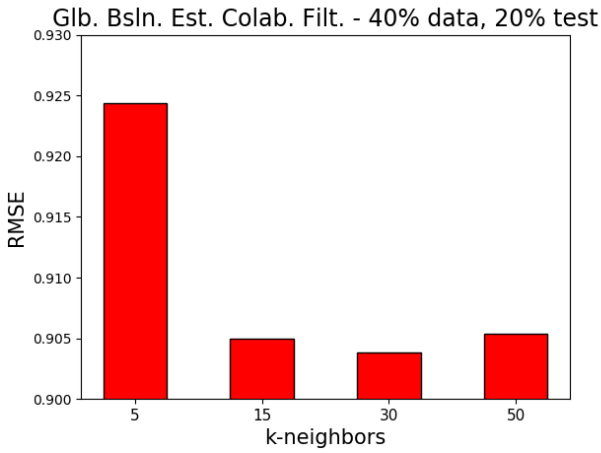

Fig. 8: Constant parameters: weights: 0.7-0.3. This plot shows how the predictive performance of global baseline collaborative filtering is influenced by the number of neighbors. Low values of the parameter correspond to bad performance, since good predictions rely on decent data sizes. However, we can see that as the number of neighbors grows, the performance starts to drop. This is a kind of underfitting, since making predictions on too many neighbors is roughly the same as taking the global average rating.

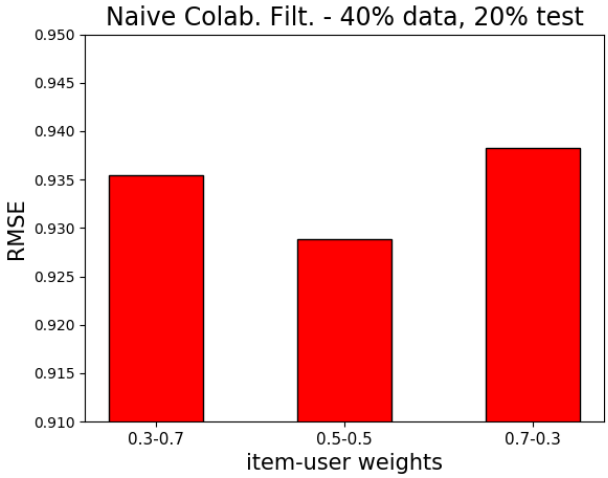

Fig. 9: Constant parameters: k-neighbors: 3. This plot shows the influence of the weights applied to item-item and user-user naive collaborative filtering. The best result seems to be obtained when combining the 2 ratings in an equal fashion.

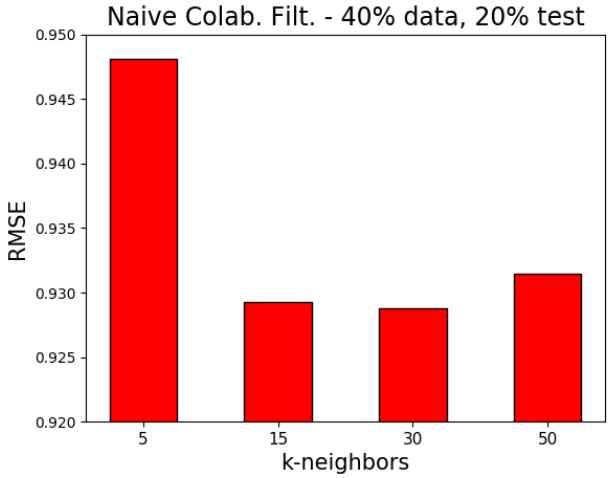

Fig. 10: Constant parameters: weights: 0.5 - 0.5. In this barchart, the influence of the number of neighbors on the RMSE score of the naive collaborative filtering algorithm can be observed. The best value for the parameter seems to be around 30. It looks like very small values, such as 5, are overfitting, while larger values, in the realm of 50, are underfitting.