Movie Recommendation

Changelog and Feedback Response

- We expanded the discussion of neural network method
- We added the hyperparameter study section
- We put our results into the comparison section

Feedback Response

The main issue in the previous report was the lack of results and comparison. The reason why we didn't provide those is that we had not finished the implementation of all methods back then. Now, we can give results and comparison of the models.

Another thing was related to the dataset. The dataset was too large to get fast results in the kNN model. There are some tricks to speed up the model but we decided to use a subset of the original dataset as we were advised to do so in the demo.

Problem Definition

In movie recommendation systems, the system tries to suggest movies that users will probably enjoy. The input of the problem is a sparse R matrix where rows represent users and columns represent movies. Each nonzero entry of the matrix is a rating value given by the corresponding user to the corresponding movie. Since a regular user can only watch a small portion of all movies in the system, the input matrix R is quite sparse. That is, most of the entries in the matrix are missing. The movie recommendation problem is to estimate missing ratings in the input matrix so that if an entry r_{ij} is predicted as a high value, movie j can be recommended to user i.

Dataset

We are going to use the movielens-25m[1] dataset. Movielens-25m is a new dataset published by GroupLens research lab. It is a new version of their previously published dataset called Movielens-20m. **Update:** After the demo, we decided to use a small subset of this dataset. Row, column and nonzero statistics are given in Table 3.

Dataset includes links, movies, ratings and tags files:

- links.csv includes the mapping between movielens movie id, imdb movie id and tmdb movie id. This file is not used.
- movies.csv includes title, year, genre info. This file is not used.
- tags.csv includes user-generated tag information. This file is not used.
- ratings.csv includes the explicit ratings of users. It's the only data file we used in the project. Each line of ratings.csv file is a tuple of userId, movieId, rating and timestamp. Timestamp info is not used in our project.

Some statistics of the dataset are given in the table below:

Number of rows	162541
Number of columns	59047
Number of nonzeros	25000095
Density	0.00260
Minimum rating	0.5
Maximum rating	5.0
Minimum row index	1
Maximum row index	162541
Minimum column index	1
Maximum column index	209171

Table 1 - Basic dataset properties

Rating histogram is reported in Figure 1. One thing to notice from the histogram is that the number of whole number ratings (1, 2, 3, 4 and 5) is greater than the number of fractional ratings (0.5, 1.5, 2.5, 3.5 and 4.5). For example, users are more likely to give 3 stars than 2.5 stars or 3.5 stars. This figure shows the histogram calculated from all ratings. When we compute the average of rows and average of columns then show them in histograms, we get Figure 2 (a) and Figure 2 (b) respectively. In this figure, it is clear that rating distributions are close to a gaussian distribution.

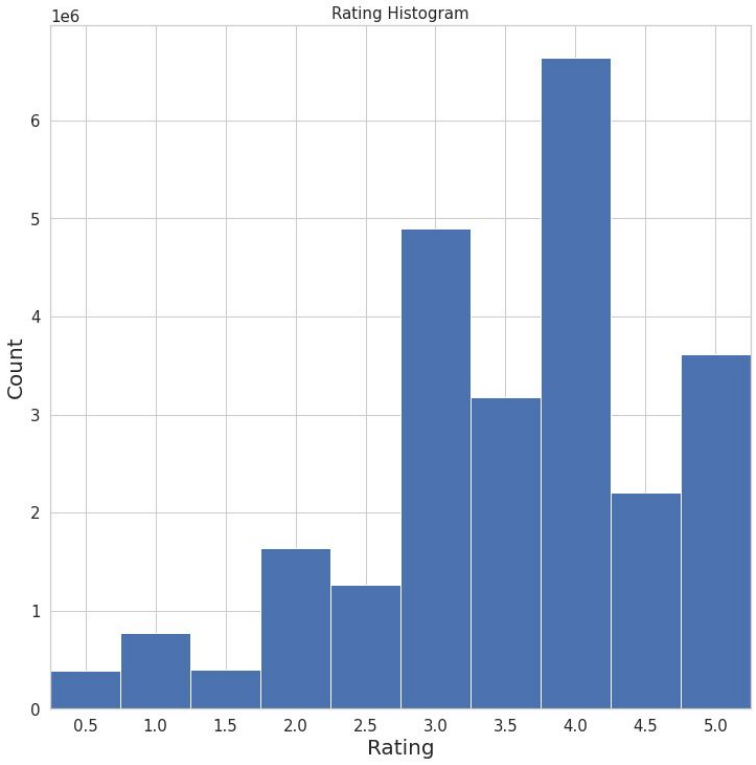

Figure 1 - Rating histogram of the dataset

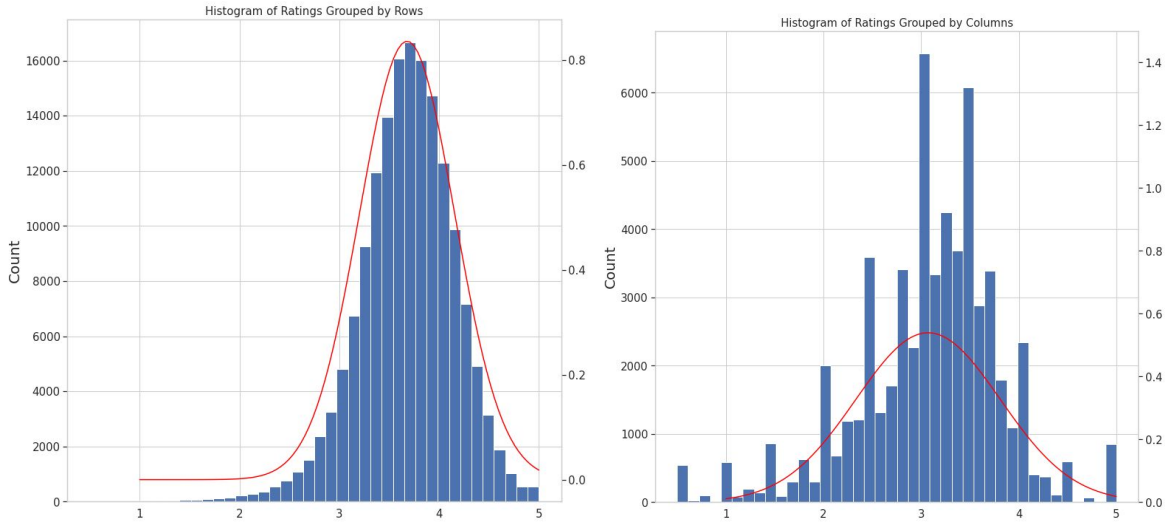

Figure 2 - (a) The histogram of average values of rows. (b) The histogram of average values of columns. Red curve shows the fitted normal distribution probability density function

Another important aspect of the dataset is row and column degrees. The number of rated movies may differ from user to user and likewise the number of users rated the movie depends on the movie. Minimum, maximum and average degrees of rows and columns are given in Table 2. Although an average user has rated around 150 movies, there is at least one user who rated more than half of the movies in the system.

Minimum row degree	20
Maximum row degree	32202
Average row degree	153.80
Minimum column degree	1
Maximum column degree	81491
Average column degree	423.39

Table 2 - Dataset degree properties

Number of rows	610
Number of columns	9724
Number of nonzeros	100836

Table 3 - Properties of the subsampled dataset. It is the dataset we use in our experiments

Methods

0. Baseline Model

A very simple model is constructed to form a baseline for our methods. In this model, we find the mean of the whole training matrix. Row deviation array and column deviation array is computed by finding deviation of rows from mean value and deviation of columns from mean respectively. According to these values, the following RMSE values are found.

Mean prediction RMSE: 1.04 (Each missing value is predicted as the mean value of the matrix)
Row-average adjusted mean prediction RMSE: 0.94 (Each missing value is predicted as corresponding row's average)
Col-average adjusted mean prediction RMSE: 0.97 (Each missing value is predicted as corresponding columns's average)
Row-average & Column-average adjusted mean prediction RMSE: 0.92 (Both row deviation and column deviation are considered)

Apart from benchmark purposes, we used them to test our code. That is, when our proposed models give RMSE values a loot higher than baseline error, we said that something is probably wrong in our implementation and then we debugged our code.

1. User-user & Item-item Collaborative Filtering (kNN)

Here we'll explain user-user collaborative filtering only. The same explanations apply for item-item collaborative filtering as well when we applied the same technique for R^T instead of R.

User-user collaborative filtering is basically a k-nearest neighbor model. To estimate a rating r_{ij}, we find k most similar users to user i, who has rated movie j. Jaccard distance, cosine distance and euclidean distance are common similarity functions used to find neighbors. Considering that our vectors are sparse and contain non-binary values, we thought the pearson correlation coefficient can be used as the similarity function in our project. So, we used the pearson correlation coefficient given as:

$$sim(i,j) = \frac{\sum_x (r_{ix} - \bar{r_i})(r_{jx} - \bar{r_j})}{\sqrt{\sum_x (r_{ix} - \bar{r_i})^2} \sqrt{\sum_x (r_{jx} - \bar{r_j})^2}}$$

After finding k-neighbors, the rating can be predicted with the average of ratings given by those k similar users to movie j. Instead of simple averaging, The weighted average or other suitable aggregation methods can be used. The aggregation methods that we've tried are given below. $nn(i)$ denotes the nearest neighbor set of user i, $s(i, j)$ denotes the similarity of user i and user j and b_{ij} denotes the baseline rating prediction of user i to item j.

Simple Average:

$$\hat{r_{ij}} = \frac{\sum_{k \in nn(i)} r_{kj}}{K}$$

Weighted Average:

$$\hat{r_{ij}} = \frac{\sum_{k \in nn(i)} s(i,k) \cdot r_{kj}}{\sum_{k \in nn(i)} s(i,k)}$$

Baseline Considered Weighted Average:

$$\hat{r_{ij}} = b_{ij} + \frac{\sum_{k \in nn(i)} s(i,k) \cdot (r_{kj} - b_{kj})}{\sum_{k \in nn(i)} s(i,k)}$$

In this model, k and the selection of the aggregation method (simple average, weighted average and baseline considered weighted average) are the hyperparameters of the model.

We implemented the kNN method. However, it is quite slow because of $O(N^2)$ time complexity of the model. Quadratic time complexity comes from the fact that we need to calculate the similarity of every pair. To overcome the problem, locality sensitive hashing (LSH) methods can be used to find similar users with high probability in sub quadratic times. This improvement is not coded, we decided to use a smaller dataset instead.

2. Funk-SVD

Matrix factorization/completion methods are popularized in the context of recommendation problems with Netflix prize challenge[2]. One of the factorization methods called Funk-SVD[3] tries to find two dense matrices called $P_{n\,x\,f}$ (user latent-factor matrix) and $Q_{f\,x\,m}$ (item latent-factor matrix) such that their matrix multiplication approximates the rating matrix $R_{n\,x\,m}$. In this case, the cost function becomes differentiable so it can be minimized with gradient descent algorithms. Funk-SVD is similar to PCA concept which is discussed in the course lectures. We used stochastic gradient descent (SGD) to minimize the cost function, SGD update rules for a single training rating r_{ij} are as follows:

$$p_i \leftarrow p_i + \lambda(e_{ij} \cdot q_j + \beta \cdot p_i)$$

$$q_j \leftarrow q_j + \lambda(e_{ij} \cdot p_i + \beta \cdot q_j)$$

Where e_{ij} is error term calculated from r_{ij}, λ denotes the learning rate and β denotes the regularization parameter.

In this model, factor size f, learning rate λ and regularization parameter β are the hyperparameters of the model. Figure 3 shows RMSE values for training and validation set with respect to iteration number for a selection of parameters. How we select the model's hyperparameters will be explained in the Hyperparameter Study section.

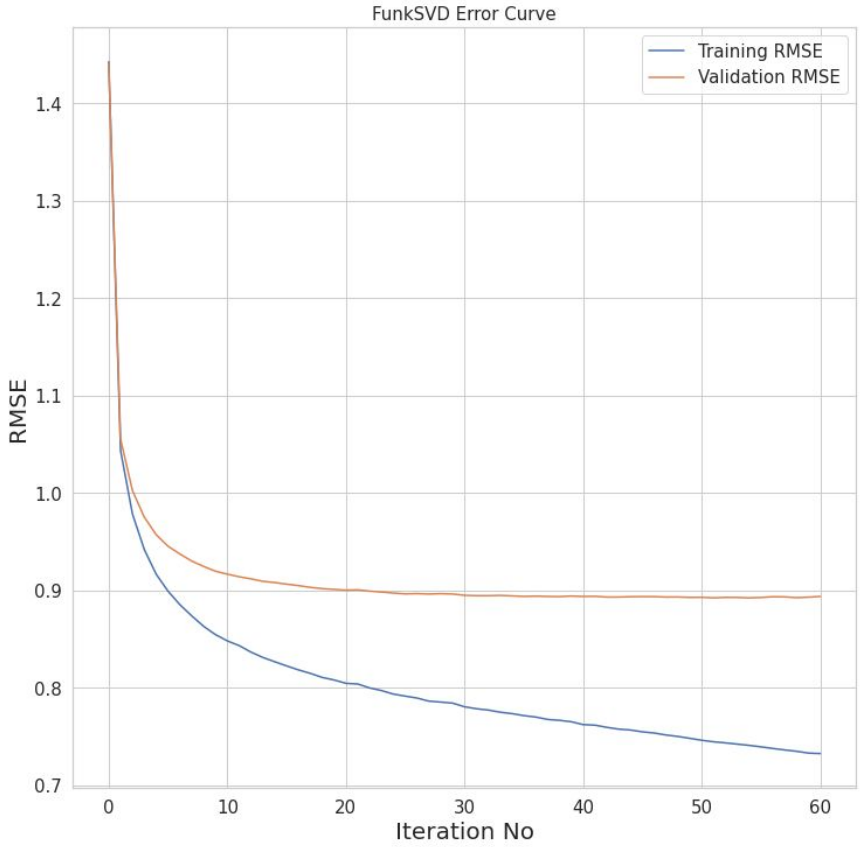

Figure 3 - The curve of training and validation root mean squared error with respect to iteration numbers in the Funk-SVD method

3. Neural Collaborative Filtering

Neural networks are a significant part of the course so we would like to apply a neural network approach to predict ratings in our project. Recommendation with neural networks is a bit tricky because input layer representation and network architecture are not straightforward. After doing a literature research, we decided to use neural collaborative filtering[4] (NCF) architecture depicted in Figure 4.

We like to consider this architecture as two parts. The first part is the embedding layer. The embedding layer is similar to the embeddings in the FunkSVD model. Again, we represent rows with $P_{n\,x\,f}$ (user latent-factor matrix) and represent columns with $Q_{f\,x\,m}$ (item latent-factor matrix). So, when a user index and item index are fed into the embedding layer, corresponding P and Q rows are extracted, concatenated and forwarded into the next layer.

The second part of the architecture is a simple multilayer perceptron. The input of this part is the concatenated user and item embedding vector. The output layer is a single output node computing the rating estimate. The layers between them are fully connected hidden layers. The count of hidden layer and the size of each hidden layer are hyperparameters of the model. Other hyperparameters include the latent factor size and learning rate. In our implementation, we used stochastic gradient descent to optimize both weights in the embedding layer and weights in the multilayer perceptron layers.

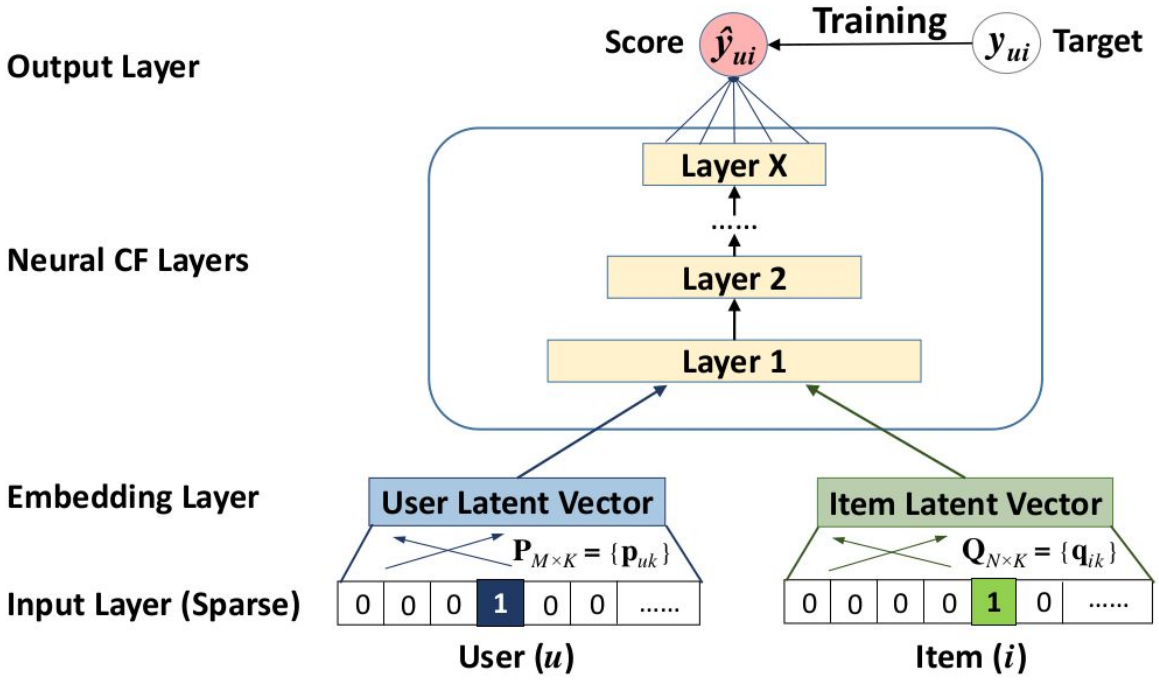

Figure 4 - Network architecture of NCF model[4]

Challenges

- The matrix is sparse. To overcome memory issues, sparse matrices should be stored in specialized sparse formats. We need to use COO, DOK, CSR and CSC sparse matrix formats carefully to solve this issue.
- Rating distribution: Ratings are not distributed on the matrix uniformly. The degree of rows and the degree of columns follows a pareto distribution. This phenomenon is generally known as the 20-80 rule. In this context, it means that 20 percent of users have 80 percent of all ratings and the other 80 percent majority of users have 20 percent of ratings. A similar pattern can be seen in column degrees as well. So, the matrix is quite skewed in terms of the number ratings. It is a major problem in statistical learning since the model does not have enough data for a large part of its parameters. The degrees of rows and columns are plotted with respect to their frequency in Figure 5.

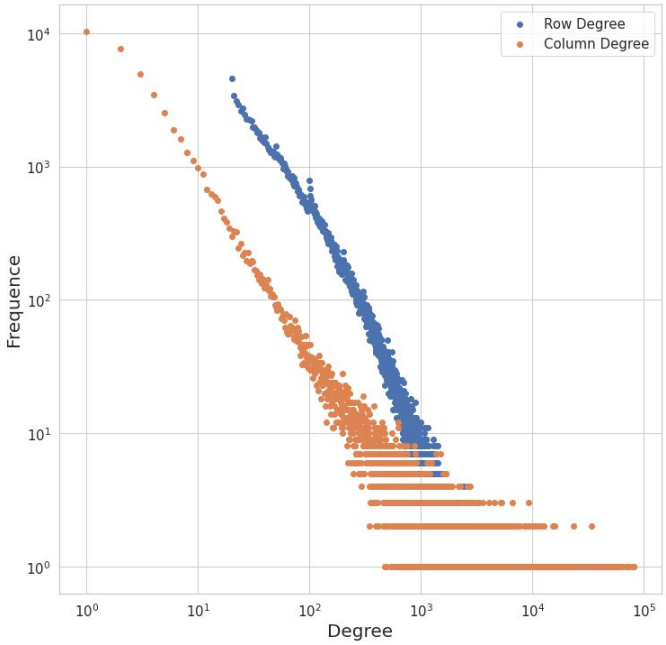

Figure 5 - Degree-Frequence plot of rows and columns in log scale

Simulation Setup

We follow the following steps to compare three suggested methods:

1. Preprocessing

We filter out unrelated columns in our dataset. As shown in the Problem Definition section, maximum column index is not equal to the number columns. That is, there are gaps between column indices. For the sake of easiness of the implementations, we mapped row and column indices so that they start from 0 and they are not gaps between indices. We use this mapped dataset as if it is our original dataset.

2. Splitting dataset

We split 60% of data as the training set, 20% of data as the validation set and 20% of data as the test set. We shuffled the dataset before splitting. In the hyperparameter tuning step, models will be trained on the training set and hyperparameters will be tuned according to the error values computed on the validation set. The test set will be used to report values of tuned models. With this setup, models won't be able to see test data in order to simulate real-life machine learning problems.

3. Tuning the models

Each of the suggested models has some hyperparameters that should be configured before training such as k parameter of kNN models. To find the optimal hyperparameters, we search over a range of values. We train the model using the training set with various different hyperparameters, the model that gives the minimum error on the validation set will be selected.

4. Evaluation

Root mean square error will be used as the evaluation metric. Here r_{ij} denotes the true rating value, $\widehat{r_{ij}}$ denotes the predicted rating value and $|R|$ denotes the number ratings.

$$RMSE = \sqrt{\frac{1}{|R|}\sum_{r_{ij}\in R}(r_{ij} - \hat{r}_{ij})^2}$$

RMSE values calculated on the test set for our tuned models will be compared with each other to show which of the three models is better.

Hyperparameter Study

We tune the hyperparameters of the models on a validation set as it is described in the Simulation Setup section. The following tables show some of the parameters we've tried and their corresponding RMSE values.

1. K-Nearest Neighbor

	RMSE of Different Aggregation Approaches		
K	**Simple Average**	**Weighted Average**	**Baseline Average**
4	1.00	1.02	0.93
8	0.98	0.99	0.915
16	0.97	0.98	0.907
32	0.97	0.98	**0.906**
64	0.97	0.98	0.907

Table 4 - kNN model hyperparameter tuning results

Baseline average aggregation method is clearly superior. It is expected due to the reasons mentioned in the kNN section. About the selection of K, K=32 seems to be the best. The high value of K increases the generalizability of the model but it may cause underfitting. K=32 selection seems to be the best tradeoff.

2. FunkSVD

Factor Size	Learning Rate	Regularization	RMSE
8	0.003	0.2	0.91
16	0.003	0.5	1.00
16	**0.003**	**0.1**	**0.897**
128	0.003	0.2	1.04

Table 5 - FunkSVD model hyperparameter tuning results

Small factor size values may cause underfitting and large factor size values may cause overfitting. The regularization parameter should also be chosen carefully to balance this tradeoff. According to our experiments, 16 factor size with 0.1 regularization gives the best result.

3. Neural Collaborative Filtering

Factor Size	1st Layer Node Count	2nd Layer Node Count	3rd Layer Node Count	RMSE
8	16	-	-	0.893
16	**16**	**-**	**-**	**0.890**
32	64	64	-	0.892
16	32	16	16	0.908
128	64	-	-	0.902

Table 6 - NCF model hyperparameter tuning results

About the factor size selection, it is similar to factor size in the FunkSVD model. Large values may overfit and small values may not be strong enough to capture the patterns. However it is not easy to interpret why some hidden layer architecture performs better than others. Intuitively, we know that at least one hidden layer should be used. Other than that, this selection was mainly on trying different parameters. The best parameters are highlighted in the table above. Learning rate selection is not shown on the table for the sake of clarity of the table.

Comparison

For each model, we tried only one set of parameters. This parameter selection is explained in the Hyperparameter Study section. The results are given below and they are compared in Figure X.

KNN Test Set RMSE : 0.886 (in 723.1 seconds)
FunkSVD Test Set RMSE : 0.893 (in 86.6 seconds)
NCF Test Set RMSE : 0.884 (in 244.2 seconds)

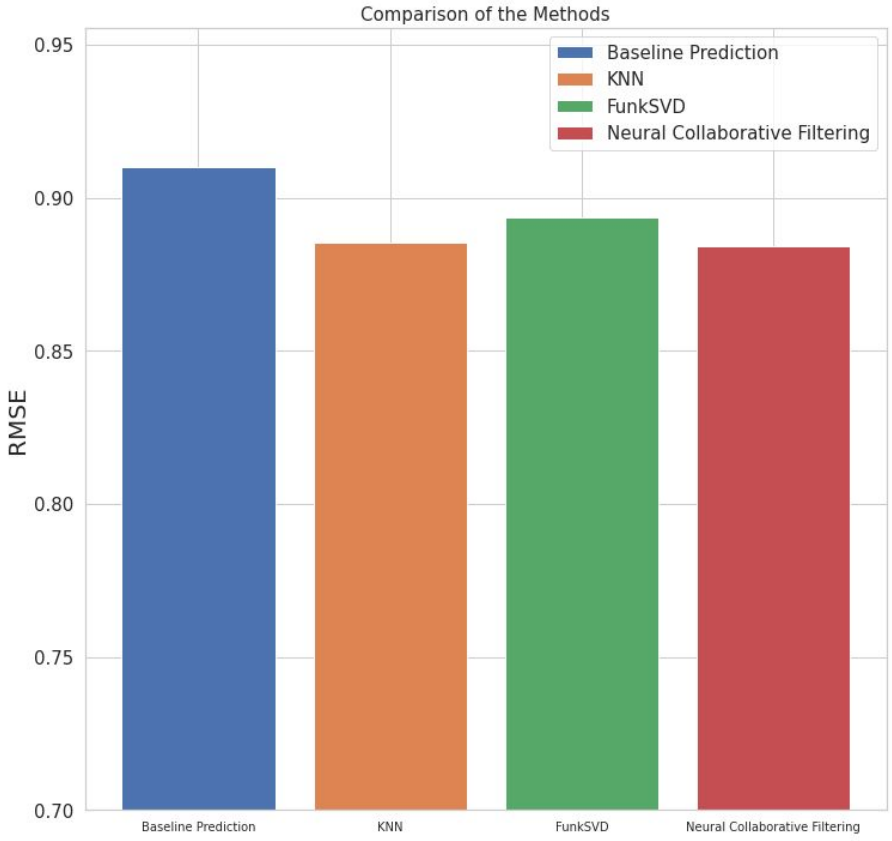

Figure 6 - RMSE results of different methods along with baseline RMSE errors

Although every model performs better than baseline estimates, it can be said that NCF gives the best result and FunkSVD gives the worst result. The difference between the models is not much. Since FunkSVD computes the rating by just looking at the dot product of embedded vectors, the relationships it can model are limited. On the other hand, NCF can model more than the dot product relationship users and movies. In theory, NCF with the correct setup can learn anything FunkSVD can learn because of user-item embeddings. However, FunkSVD is faster than other models so it can be also considered as a valid method in practice especially when we consider how close the RMSE values are.

Conclusion

We implemented user-user collaborative filtering with k nearest neighbor algorithm, FunkSVD matrix completion algorithm and neural collaborative algorithm to predict the missing entries from a rating matrix generated from Movielens-25m dataset. After splitting the dataset into training, validation and test sets, we tuned parameters of each model using training and validation sets. Once best hyperparameters are found, we tried a single model for each algorithm to compare them with themselves and the baseline model. According to the experiments, the neural collaborative filtering algorithm with 16 latent factors and a single hidden layer of 16 nodes gives the best result evaluated with root mean square metric.

Libraries We Used

All essential parts have been coded by us. The libraries we used in the project are listed below with brief explanations regarding why we used those.

- numpy : For fast numerical computing.
- scipy : For efficient sparse matrix data structures.
- jupyter : It lets us to code the project using a browser in a more interactive way.
- matplotlib : For plotting purposes.
- seaborn : For plotting purposes.
- tqdm : For progress bars since training may take a very long time.
- pytest : For testing purposes.

References

1. Harper, F. Maxwell, and Joseph A. Konstan. "The movielens datasets: History and context." Acm transactions on interactive intelligent systems (tiis) 5.4 (2015): 1-19.

2. Bennett, James, and Stan Lanning. "The netflix prize." Proceedings of KDD cup and workshop. Vol. 2007. 2007.

3. Bell, Robert M., and Yehuda Koren. "Lessons from the Netflix prize challenge." Acm Sigkdd Explorations Newsletter 9.2 (2007): 75-79.

4. He, Xiangnan, et al. "Neural collaborative filtering." Proceedings of the 26th international conference on world wide web. 2017.