MovieLens Recommender: A Comparative Study of Recommendation Algorithms

Abstract

This report details the process and results of building my predictive model on the MovieLens 1M dataset. I tried a variety of recommender system models, including Content-Based, Collaborative Filtering, Singular Value Decomposition (SVD), and Deep Learning models. Among all the models, the Softmax Deep Neural Networks performed the best with the smallest Mean Square Error (MSE) on the test set. I did not use pre-written packages to fit the data to the models in this project. All the models were customized by myself based on my understanding of the principles of the algorithms.

Table of Contents

1. Introduction

Recommender systems are a subclass of information filtering systems that predict or recommend items that may be of interest to users. These systems are ubiquitous in our digital lives, with applications ranging from e-commerce platforms and streaming services to social media and content aggregation sites. The main goal of recommender systems is to help users discover relevant items from a large number of choices, thereby improving their user experience and increasing user engagement.

In this report, I will detail the process of how I built a predictive model on the MovieLens 1M dataset. This dataset contains about 1M pieces of movie rating data, as well as additional information about users and movies. I first divided the data into an 80% training dataset and a 20% testing dataset. I then proposed some models to fit the data for this problem and implemented these models myself by writing algorithmic code. I will explain why I chose to use my proposed models and use what I learned in class to guide my analysis.

2. Dataset

MovieLens 1M dataset has been used for this project. This dataset contains 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

MovieLens data sets were collected by the GroupLens Research Project at the University of Minnesota. This data set consists of:

- 1,000,209 ratings (1-5) from 6,040 users on 3,900 movies.
- Basic information about each movie including title and genre (Action, Adventure, Animation, Children's, Comedy, etc.)
- Simple demographic information for the users (age, gender, occupation, zip)

3. Data Visualization

I analysed and visualised the data using "Matplotlib" and "Seaborn" in Python to derive some key insights before proceeding further with the model analysis.

I firstly plotted bar plot and pie chart of rating frequency to understand the contribution. From figure 1 and 2 it can be observed that most of the users gave 4 star rating to the movies they watched followed by 3 and 5 stars. Next, I plotted the number of movies based on genre and by looking at figure 3 it can be seen that most of the movies belong to movie genre: Drama followed by Comedy then Action, Thriller and Romance.

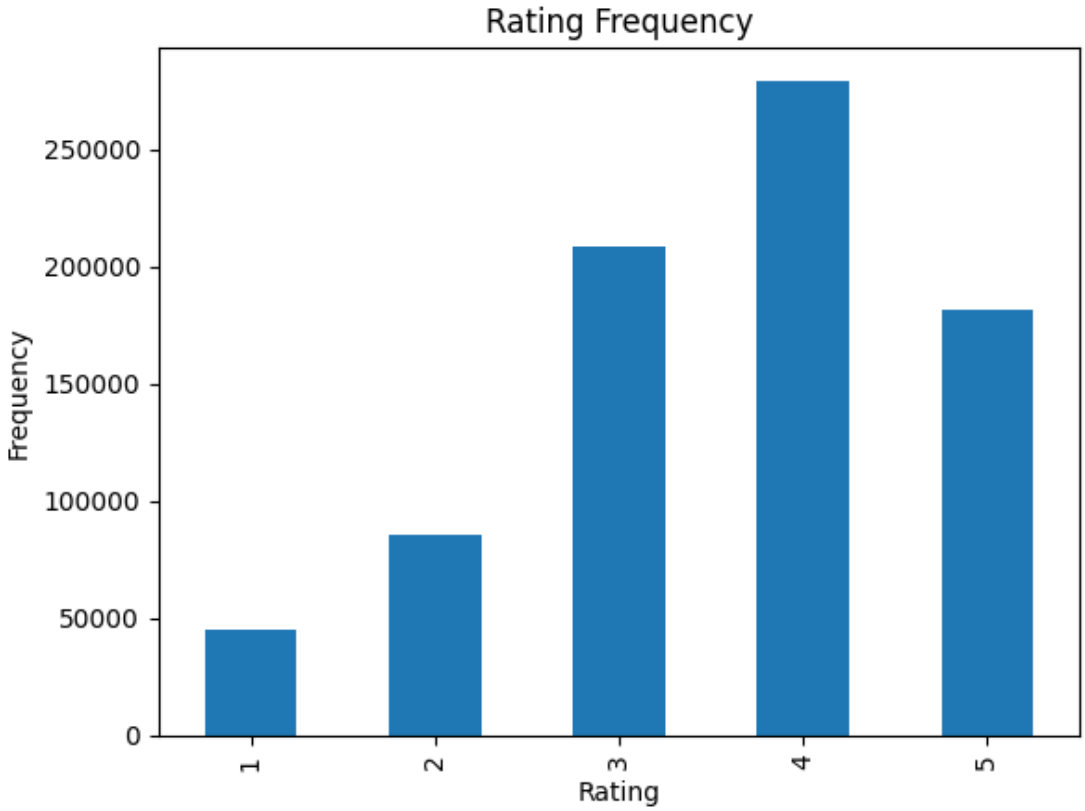

Fig 1. Bar plot of rating frequency

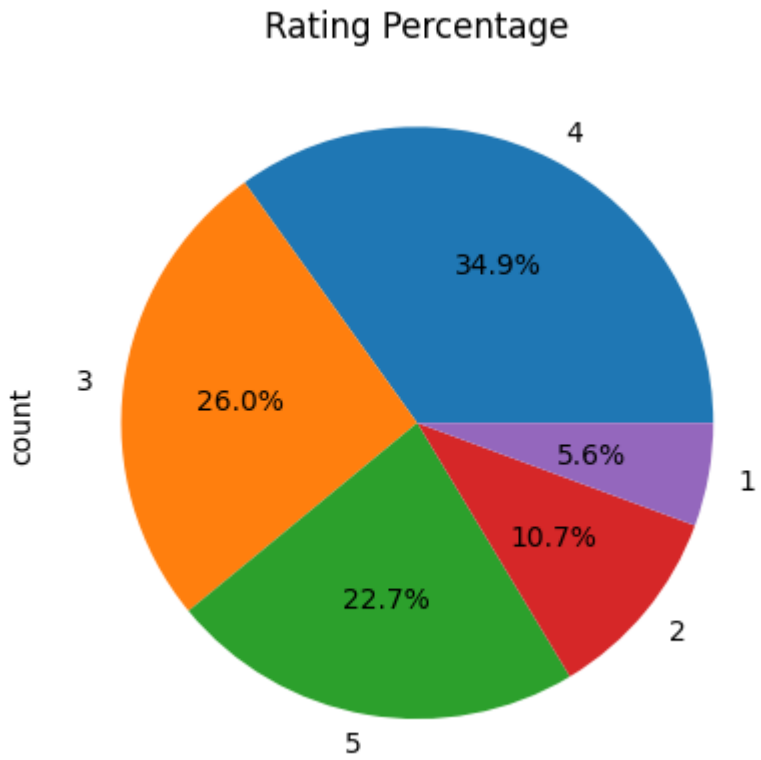

Fig 2. Pie chart of rating frequency

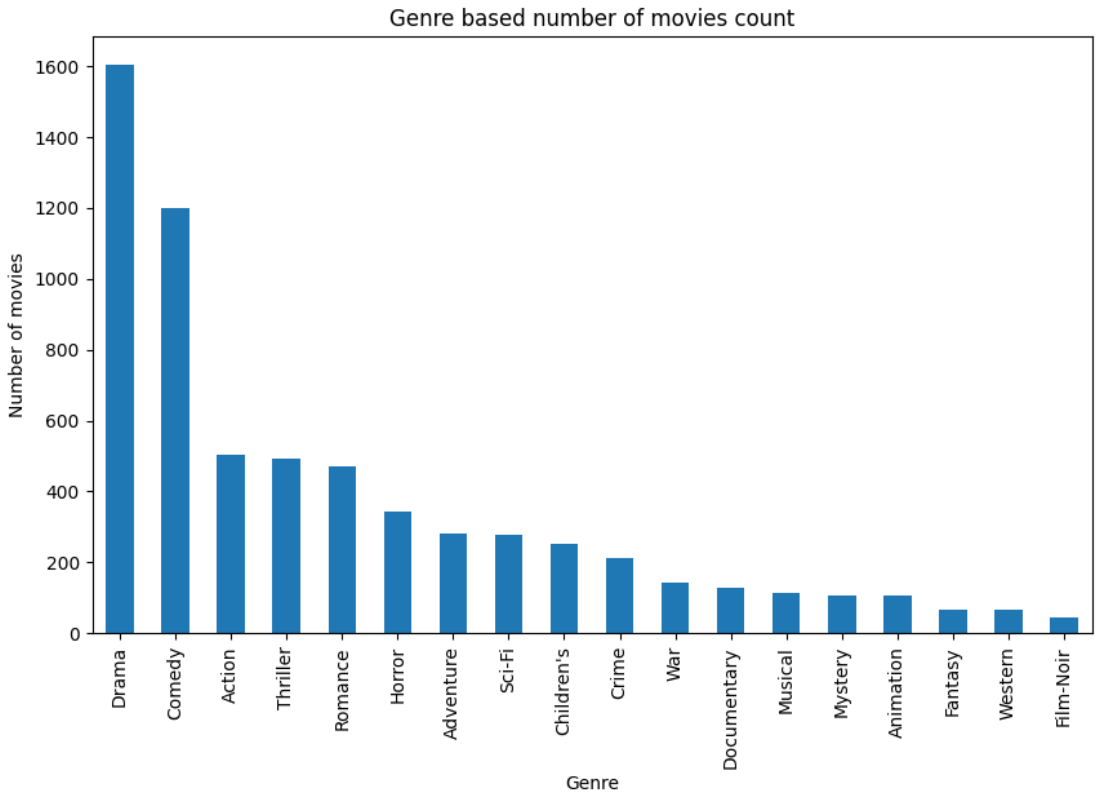

Fig 3. Genre based number of movies count

4. Recommender System Models

4.1 Content-Based

The first model I tried was Content-based Filtering. because I learnt that there are features in the dataset about the genre of movies. The content-based recommendation model can use these features to recommend movies that have similar characteristics to the movies that the user has liked before. This means that if the user has previously liked a certain genre of movie, then the system will recommend that genre of movie. In addition, in contrast to collaborative filtering models, content-based recommendation models are not plagued by the cold-start problem, i.e., the system can still make recommendations when new users or new movies are added.

Firstly, I carried out data processing. I used “TfidfVectorizer” to convert the film's genre and title into TF-IDF features to measure their importance. Then I used “scipy.sparse.hstack” to combine the title and genre features of the movies to form a movie content matrix. Finally, I calculated the similarity matrix of the film content matrix using “cosine_similarity”. Cosine similarity is a measure of the angle between two non-zero vectors and is used to calculate the similarity between them.

In the model building section, I defined a class "ContentBasedFiltering". The “fit(self, ratings)” function accepts ratings from the user and stores them in an instance variable

of the class. This function is called during the training phase of the model. The "predict(self, user, item)" function predicts the ratings based on the given user ID and movie ID. Here are the detailed steps of this function: first, it gets all the ratings of the given user from the rating data. Then, it checks if the given movie ID is in the valid range of the similarity matrix. If not, then it returns the average rating of the user as a prediction. Next, it filters out the movie IDs that are not present in the similarity matrix. Then, it obtains the similarity scores between the given movie and the movies that the user has already watched from the similarity matrix. Only similarity scores greater than 0 are retained. If there is not any valid similarity score then it returns the average rating of all the users as a prediction. Finally, it calculates the weighted score, which is the similarity score multiplied by the user's rating for the corresponding film and then divided by the sum of all similarity scores. This weighted score is the predicted score. The results show that this model has good performance with an RMSE value of 1.127676712728088 (**MSE = 1.2716547684292268**) on the test set.

4.2 Collaborative Filtering

In this section I tried a neighbourhood-based collaborative filtering model, including user-based and item-based. Two different similarity measures were used: cosine similarity and Pearson similarity. Cosine similarity is commonly used to calculate the cosine of the angle between two vectors, and in this case it can be used to measure the degree of similarity between the rating vectors, and the mathematical expression is shown in Fig. 4. Pearson correlation coefficient measures the linear correlation between the two variables, and it can be used for the rating data which has a magnitude relationship, and the mathematical expression is shown in Fig. 5.

$$sim(\mu, v) = \frac{\sum_{i \in I_{\mu v}} r_{\mu i} r_{vi}}{\sqrt{\sum_{i \in I_{\mu v}} r_{\mu i}^2} \sqrt{\sum_{i \in I_{\mu v}} r_{vi}^2}}$$

Fig. 4 Cosine similarity

$$sim(\mu, v) = \frac{\sum_{i \in I_{\mu v}} (r_{\mu i} - \bar{r}_{\mu})(r_{vi} - \bar{r}_{v})}{\sqrt{\sum_{i \in I_{\mu v}} (r_{\mu i} - \bar{r}_{\mu})^2} \sqrt{\sum_{i \in I_{\mu v}} (r_{vi} - \bar{r}_{v})^2}}$$

Fig. 5 Pearson similarity

4.2.1 User Based with Cosine Similarity

The basic idea of user-based modelling is to recommend items based on users who are

similar to you. We first need to define what a "similar" user is. For a given two users u and v, let Iu and Iv be the set of items they have evaluated. We can compare the similarity of Iu ∩ Iv, i.e., the items that both u and v have evaluated. Different users have different baselines for evaluating items, so we should normalise by subtracting their average scores. Then, we can calculate the cosine similarity between u and v. Given the evaluation of item j by the k users closest to user u, Pu(j), the predicted evaluation is the weighted average of item j by neighbour v.

In the code part of the implementation of this model, I first defined a function "cos_sim" to calculate the cosine similarity between two vectors. I added a very small constant (1e-5) to avoid division by zero. Next, I defined a class "UserBasedCF" to implement a user-based collaborative filtering model. I set a hyperparameter of k, which represents the number of most similar users to be considered in the prediction. In the "fit (self, trainset)" function, for each pair of users i and j, I compute the cosine similarity between the rating vectors of their rated items and store the results in user_similarity[i, j] and user_similarity[j, i]. In the "predict (self, user, item)" function, the following steps are used to predict the ratings of a given user and item:

- First, get the similarities between the given user and all other users.
- Find the index “top_k_users” of the top k most similar users to the given user from “user_similarities”.
- Get the ratings “item_ratings” of these k most similar users towards the predicted item.
- If none of the k users rated the item, return the average rating of the item as the predicted value.
- Otherwise, a weighted average of the ratings and similarities of similar users is used to obtain the predicted ratings. Specifically, each similar user's rating is multiplied by its similarity to the target user, summed and divided by the sum of all similarities.

The results show that this model has a mediocre performance with an RMSE value of 3.494451056783176 (**MSE = 12.211188188253056**) on the test set.

4.2.2 User Based with Pearson Similarity

In this section, I first define a "pear_sim" function to compute the Pearson correlation coefficient between two vectors. Before calculating the correlation coefficient, the function centres each vector to remove the effect of user rating bias on the correlation coefficient calculation. Next, I use the Pearson correlation to calculate the similarity between users. The rest of the code is along the same lines as the above section. The results show that this model has a mediocre performance with an RMSE value of 3.5479099644304 (**MSE = 12.587665115704523**) on the test set.

4.2.3 Item Based with Cosine Similarity

In contrast to user-based filtering, we can use a similar approach for item-based filtering. We can use the intersection Ui ∩ Uj of users who have evaluated items i and j to define a similarity measure between items. Similarly, we can use user u's prediction of item t as the weighted average of the k neighbours of item t, Qt(u).

In the code part of implementing this model, I first define a function "cos_sim" to calculate the cosine similarity between two vectors. Next, I define a class "ItemBasedCF" to implement the item-based collaborative filtering model. I set a hyperparameter of k, which represents the number of most similar items to be considered in the prediction. In the "fit (self, trainset)" function, unlike the user-based model, I first transpose the input training dataset (self.trainset = trainset.T). This is because in the item-based model, we need to compare the similarity between items, not between users. Then, for each pair of items i and j, I compute the cosine similarity between the rating vectors of their rated users and store the results in item_similarity[i, j] and item_similarity[j, i]. In the "predict (self, user, item)" function, I implement the following steps to predict the ratings of a given user and item:

- First, get the similarity between the given item and all other items.
- Find the index "top_k_items" of the top k items from "item_similarities" that are most similar to the given item.
- Get the ratings of the target user for the k most similar items "user_ratings".
- If the user has not rated the k items, return the average rating of the user as a prediction.
- Otherwise, a weighted average of the ratings and similarity of similar items is used to obtain the predicted ratings. Specifically, the rating of each similar item is multiplied by its similarity to the target item, summed and divided by the sum of all similarities.

The results show that this model has a mediocre performance with an RMSE value of 3.4646989334391756 (**MSE = 12.004138699374561**) on the test set.

4.2.4 Item Based with Pearson Similarity

In this section, I first define a "pear_sim" function that calculates the Pearson correlation coefficient between two vectors. Before calculating the correlation coefficient, the function first centred each vector (subtracting the vector mean), in order to eliminate the effect of user bias in the ratings on the calculation of the correlation coefficient. Next, I use the Pearson correlation to calculate the similarity between items. The rest of the code is along the same lines as the above section. The results show that this model has a mediocre performance with an RMSE value of 3.5296624277273656 (**MSE = 12.45851685371024**) on the test set.

4.3 SVD Model

One challenge that comes up when building recommender systems - the long tail problem - was brought up in lecture. By looking at Figure 6 we can see that only a small percentage of items are rated frequently. Such items are known as popular items. The vast majority of items are rarely rated. The solution to the long tail can be to use matrix decomposition techniques to train models to learn user-item interactions by capturing user information in user latents and item information in item latents. At the same time, matrix decomposition techniques can significantly reduce dimensionality and sparsity, reducing the large memory footprint and making our system more scalable. In this section I will try to use singular value decomposition to build recommender systems.

Singular value decomposition is done on utility matrix and latent features of rows and columns (movies and users in this project). In SVD decomposition, Where A is a m * n utility matrix, U is a m * r orthogonal left singular matrix, which represents the relationship between users and latent factors, S is a r * r diagonal matrix, which describes the strength of each latent factor and V is a r * n diagonal right singular matrix, which indicates the similarity between items and latent factors. The latent factors here are the characteristics of the items, for example, the genre of the movie. The SVD decreases the dimension of the utility matrix A by extracting its latent factors. It maps each user and each item into a r-dimensional latent space. This mapping facilitates a clear representation of relationships between users and items.

The SVD reduces the dimensionality and provides us with important latent features that approximate almost all values in the utility matrix (including the null). The values already in the utility matrix can be used to evaluate the predictions and adjust the parameters/weights in the latent features or help us select the number of latent features from the SVD decomposition.

I wrote a class "SVD" to build a recommendation system based on matrix decomposition. First, I defined four hyperparameters: n_factors (number of hidden factors), n_epochs (number of training iterations), lr (learning rate), reg (regularisation parameter). The "fit(self, ratings)" function accepts the user's ratings data ratings and trains the model. First, it initialises the user factor matrix and item factor matrix, then it performs several iterations, in each iteration, it iterates through each non-zero rating, calculates the prediction error, and updates the corresponding user factor and item factor. The "predict(self, user, item)" function predicts the rating based on the given user ID and item ID. The predicted rating is the dot product of the user factor and item factor. In addition, I used a grid search strategy to tune the hyperparameters in this model. Finally, due to time consuming issues, I selected the parameters that performed the best before I interrupted the procedure. They are n_factors=40, n_epochs=40, lr=0.005, and reg=0.05. The results show that this model has good performance with an RMSE value of 1.3213269988150962 (**MSE = 1.7459050377977092**) on the test set.

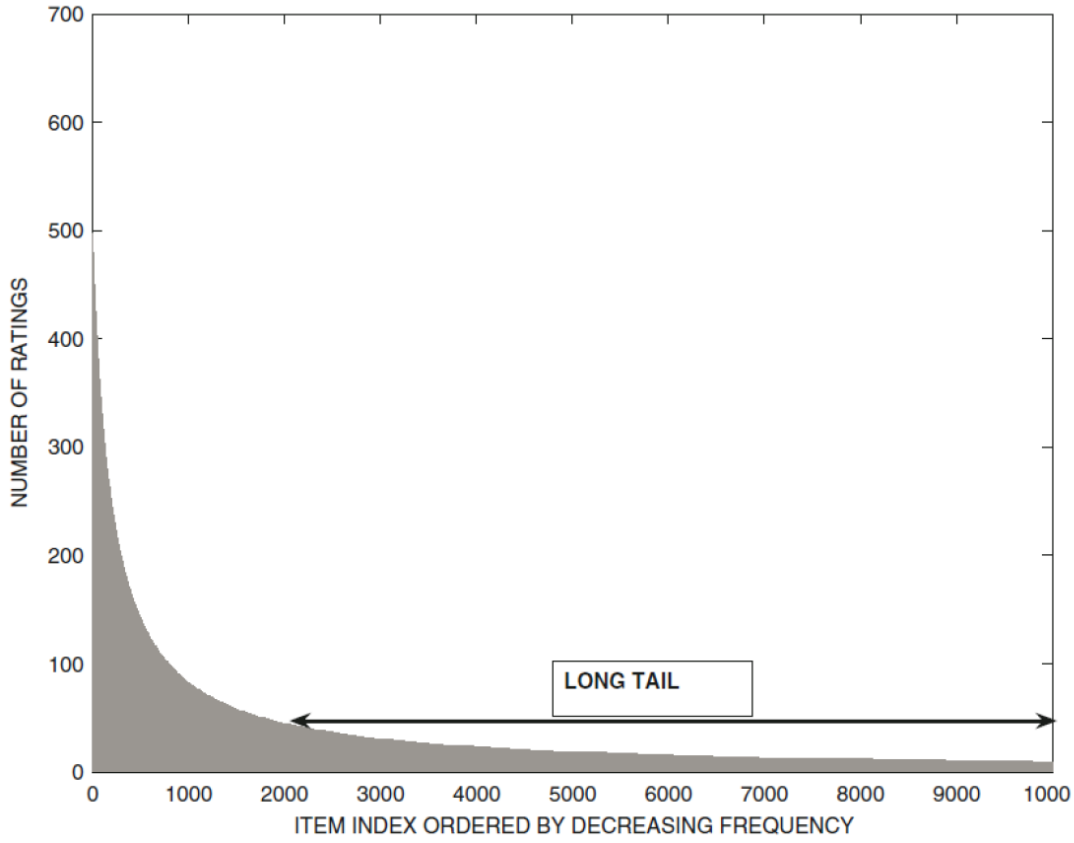

Fig. 6 Distribution of ratings frequency is “long-tailed” (screenshot from Lecture 8)

4.4 Deep Learning Model

The limitations of matrix decomposition can be addressed by Deep Neural Network (DNN) models. Due to the flexibility of the network's input layers, DNNs can easily merge user and movie features, which helps to capture user-specific interests and improve the relevance of recommendations.

In this project, I use a deep neural network with a Softmax layer to recommend movies. Users and movies are converted into indexes and fed into the deep neural network, and the output of the network is a dense layer of 9 neurons (corresponding to possible ratings from 1 to 5) with an added Softmax activation function.

The deep neural network model is constructed by extracting latent features of the user and the movie with the help of an embedding layer. These features are then connected together and processed through two dense layers with ReLU activation functions. Finally, the predicted ratings are output through a dense layer with a Softmax activation function.

The hyperparameters of the model include the size of the embedding layer (50), the number of neurons in the dense layers (256 and 128), and the number of training iterations (10). The model is trained using a stochastic gradient descent (SGD)

optimiser and a sparse categorical cross entropy loss function.

During training, the model's weights are updated to minimise validation loss. Finally, the model is predicted on the test set and the root mean square error (RMSE) of the prediction is calculated. The results show that this model has good performance with an RMSE value of 1.0151446158532635 (**MSE = 1.03051859109587**) on the test set.

5. Conclusion

In this project, I first performed exploratory data analysis (EDA) on the original dataset, including data visualisation, which is an important step in building a suitable recommender system model. I noticed that the "movies" data in the original dataset contained a rich set of movie features, so I first tried Content-based filtering with satisfactory results.

Then, I turned my attention to the matrix of user-movie interaction "ratings" and tried user-based collaborative filtering and item-based collaborative filtering. I used both cosine similarity and Pearson similarity as similarity measures. However, the results of collaborative filtering were not satisfactory. I analysed the problem and concluded that it could be due to data sparsity caused by the long-tail problem, which made it difficult for collaborative filtering to find enough similar users or items to make accurate recommendations.

Therefore, I next tried a matrix decomposition method based on singular value decomposition (SVD). After tuning the hyperparameters, the method also showed good performance. Finally, I experimented with deep neural networks (DNNs). Although I did not perform hyperparameter tuning, this model outperformed all previous models on the test set.

By comparing the mean square error (MSE) values of each model on the test set, I concluded that the recommender system using the Softmax deep neural network was optimal. The experience of this project suggests that while the various models have their merits and applicability scenarios, the deep neural network provided the best performance on this particular problem. It also reminds us that in real-world problems, we need to try and compare different models to find the most appropriate solution based on the specific data and requirements.

Tab. 1

Model	Content Based	Collaborative Filtering				SVD	DNN
		User Based (Cosine)	User Based (Pearson)	Item Based (Cosine)	Item Based (Pearson)		
RMSE	1.12768	3.49445	3.54791	3.46470	3.52966	1.32133	1.01514
MSE	1.27165	12.21119	12.58767	12.00414	12.45852	1.74591	1.03052