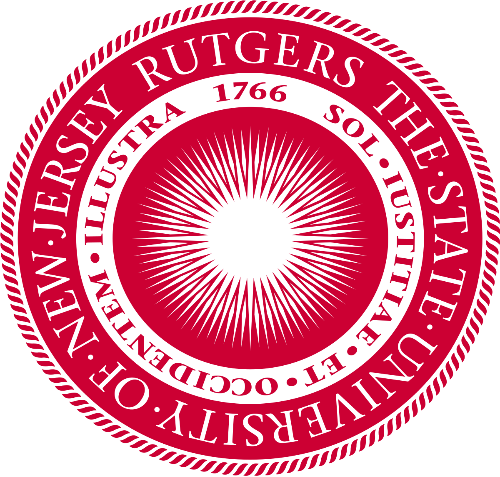
12/1/2019

**Movie Recommendation System**

By DIVYA NITIN NAIDU

MIT CAPSTONE PROJECT

Rutgers university



Contents

[Introduction 4](#_Toc26108849)

[Dataset 5](#_Toc26108851)

[Data Pre-processing 7](#_Toc26108852)

[List of genres 7](#_Toc26108853)

[Search Matrix 8](#_Toc26108854)

[Ratings Matrix 9](#_Toc26108855)

[Recommendation Algorithms 9](#_Toc26108856)

[Data Exploration 11](#_Toc26108857)

[Data exploration using Similarity function 12](#_Toc26108858)

[The most viewed Movies 13](#_Toc26108859)

[Average Movie rating 14](#_Toc26108860)

[Heatmap of the rating matrix 15](#_Toc26108861)

[Data Preparation 17](#_Toc26108862)

[ITEM-based Collaborative Filtering Model 19](#_Toc26108863)

[Building the recommendation model 19](#_Toc26108864)

[Applying recommender system on the dataset: 21](#_Toc26108865)

[USER-based Collaborative Filtering Model 23](#_Toc26108866)

[Building the recommendation system: 24](#_Toc26108867)

[Applying the recommender model on the test set 24](#_Toc26108868)

[Explore results 24](#_Toc26108869)

[Evaluating the Recommender Systems 26](#_Toc26108870)

[Preparing the data to evaluate models 26](#_Toc26108871)

[Splitting the data 26](#_Toc26108872)

[Bootstrapping the data 28](#_Toc26108873)

[k-Fold Cross-validation 29](#_Toc26108874)

[Evavluating the ratings 29](#_Toc26108875)

[Evaluating the recommendations 30](#_Toc26108876)

[Comparing models 33](#_Toc26108877)

[Identifying the most suitable model 34](#_Toc26108878)

[Optimizing a numeric parameter 35](#_Toc26108879)

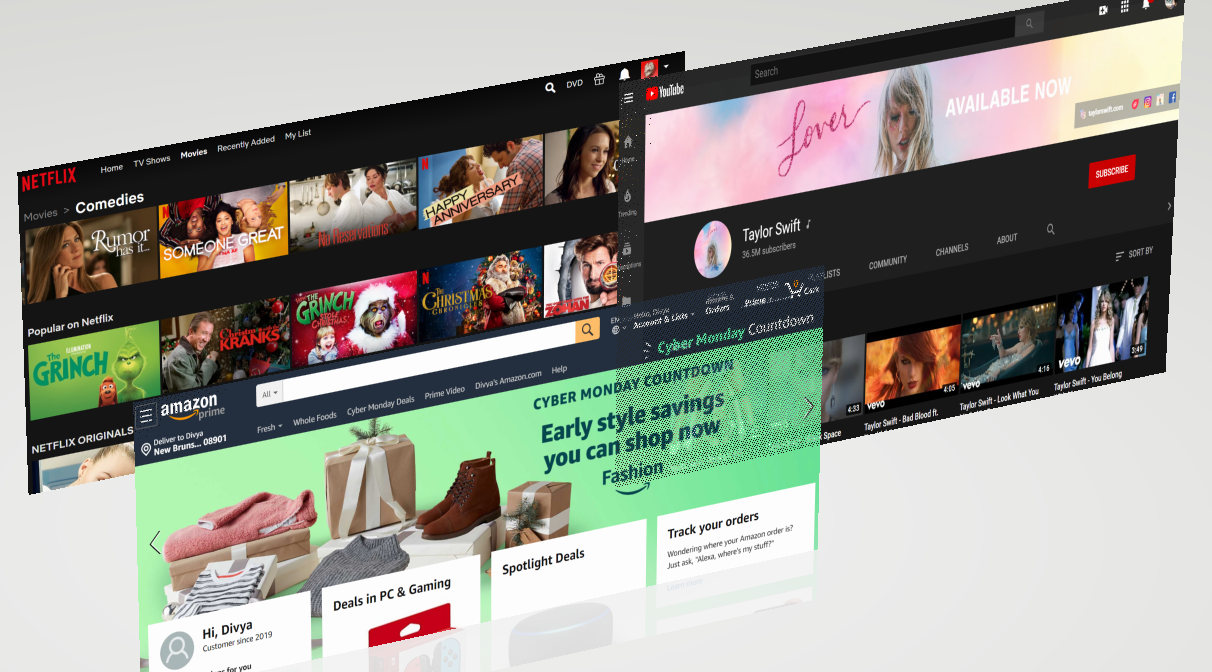
[Conclusion 37](#_Toc26108880)

[References 37](#_Toc26108881)

## Introduction

As the business needs are accelerating, there is an increased dependence on extracting meaningful information from humongous amount of raw data to drive business solutions. Also, with the increase in the amount of available information, new problems arise as people are finding it hard to select the items they want to see or use. This is where the recommender system comes in. They help us make decisions by learning our preferences or by learning the preferences of similar users.

Recommender systems are one of the most popular algorithms in data science today. They possess immense capability in various sectors ranging from entertainment to e-commerce. Recommender Systems have proven to be instrumental in pushing up company revenues and customer satisfaction with their implementation.



They are used by almost every major company in some form or the other. Netflix uses it to suggest movies to customers, YouTube uses it to decide which video to play next on autoplay, and Facebook uses it to recommend pages to like and people to follow.

In this project, a collaborative filtering recommender (CFR) system for recommending movies is developed.

The theory behind collaborative filtering to work with collaboration with user or movie id. For example, there are two user A and B, user A likes movie P,Q,R,S and user B like movies Q,R,S,T. Since movies Q, R and S are common for both users, therefore, movie P will be recommended to user B and movie T will be recommended to user A. The collaborative filtering approach considers only user preferences and does not consider the features or contents of the items (books or movies) being recommended.

## Libraries

The recommenderlab package is used to in this project and the same is followed from the r cran repository. The recommenderlab provides the infrastructure to develop and test recommender algorithms for rating data and 0-1 data in a unified framework. The Package provides basic algorithms and allows the user to develop and use his/her own algorithms in the framework via a simple registration procedure.

Collaborative filtering algorithms are typically divided into two groups, memory-based CF

and model-based CF algorithms. Memory-based CF use the whole (or at least a large sample of the) user database to create recommendations. Model-based algorithms use the user database to learn a more compact model (e.g, clusters with users of similar preferences) that is later used to create recommendations.

library(recommenderlab)  
library(ggplot2)  
library(data.table)  
library(reshape2)

## Dataset

The dataset used is from MovieLens, and is publicly available at <http://grouplens.org/datasets/movielens/latest>. This dataset (ml-latest-small) describes 5-star rating and free-text tagging activity from [MovieLens](http://movielens.org/), a movie recommendation service. These data were created by 610 users between March 29, 1996 and September 24, 2018. This dataset was generated on September 26, 2018.

The data are contained in the files links.csv, movies.csv, ratings.csv and tags.csv. I have used the files movies.csv and ratings.csv to build the recommendation system.

A summary of *movies* is given below, together with several first rows of a dataframe:

## movieId title   
## Min. : 1 Confessions of a Dangerous Mind (2002): 2   
## 1st Qu.: 3248 Emma (1996) : 2   
## Median : 7300 Eros (2004) : 2   
## Mean : 42200 Saturn 3 (1980) : 2   
## 3rd Qu.: 76232 War of the Worlds (2005) : 2   
## Max. :193609 '71 (2014) : 1   
## (Other) :9731   
## genres   
## Drama :1053   
## Comedy : 946   
## Comedy|Drama : 435   
## Comedy|Romance: 363   
## Drama|Romance : 349   
## Documentary : 339   
## (Other) :6257

## movieId title  
## 1 1 Toy Story (1995)  
## 2 2 Jumanji (1995)  
## 3 3 Grumpier Old Men (1995)  
## 4 4 Waiting to Exhale (1995)  
## 5 5 Father of the Bride Part II (1995)  
## 6 6 Heat (1995)  
## genres  
## 1 Adventure|Animation|Children|Comedy|Fantasy  
## 2 Adventure|Children|Fantasy  
## 3 Comedy|Romance  
## 4 Comedy|Drama|Romance  
## 5 Comedy  
## 6 Action|Crime|Thriller

And here is a summary and a head of *ratings*:

## userId movieId rating timestamp   
## Min. : 1.0 Min. : 1 Min. :0.500 Min. :8.281e+08   
## 1st Qu.:177.0 1st Qu.: 1199 1st Qu.:3.000 1st Qu.:1.019e+09   
## Median :325.0 Median : 2991 Median :3.500 Median :1.186e+09   
## Mean :326.1 Mean : 19435 Mean :3.502 Mean :1.206e+09   
## 3rd Qu.:477.0 3rd Qu.: 8122 3rd Qu.:4.000 3rd Qu.:1.436e+09   
## Max. :610.0 Max. :193609 Max. :5.000 Max. :1.538e+09

## userId movieId rating timestamp  
## 1 1 1 4 964982703  
## 2 1 3 4 964981247  
## 3 1 6 4 964982224  
## 4 1 47 5 964983815  
## 5 1 50 5 964982931  
## 6 1 70 3 964982400

Both *usersId* and *movieId* are presented as integers and should be changed to factors. Genres of the movies are not easily usable because of their format, so we need to do the changes to get in the correct usable format.



Word cloud using tags.csv file.

## Data Pre-processing

Some pre-processing of the data available is required before creating the recommendation system.

The information of movie genres is re-organized in such a way that allows future users to search for the movies they like within specific genres. From the design perspective, this is much easier for the user compared to selecting a movie from a single very long list of all the available movies.

### List of genres

Here a matrix of corresponding genres for each movie is created.

## Action Adventure Animation Children Comedy Crime Documentary Drama  
## 1 0 1 1 1 1 0 0 0  
## 2 0 1 0 1 0 0 0 0  
## 3 0 0 0 0 1 0 0 0  
## 4 0 0 0 0 1 0 0 1  
## 5 0 0 0 0 1 0 0 0  
## 6 1 0 0 0 0 1 0 0  
## Fantasy Film-Noir Horror Musical Mystery Romance Sci-Fi Thriller War  
## 1 1 0 0 0 0 0 0 0 0  
## 2 1 0 0 0 0 0 0 0 0  
## 3 0 0 0 0 0 1 0 0 0  
## 4 0 0 0 0 0 1 0 0 0  
## 5 0 0 0 0 0 0 0 0 0  
## 6 0 0 0 0 0 0 0 1 0  
## Western  
## 1 0  
## 2 0  
## 3 0  
## 4 0  
## 5 0  
## 6 0

#### Word cloud of all the movie Genre

### Search Matrix

A *search matrix* is created for an easy search of a movie by its genre.

## movieId title Action Adventure Animation  
## 1 1 Toy Story (1995) 0 1 1  
## 2 2 Jumanji (1995) 0 1 0  
## 3 3 Grumpier Old Men (1995) 0 0 0  
## 4 4 Waiting to Exhale (1995) 0 0 0  
## 5 5 Father of the Bride Part II (1995) 0 0 0  
## 6 6 Heat (1995) 1 0 0  
## Children Comedy Crime Documentary Drama Fantasy Film-Noir Horror Musical  
## 1 1 1 0 0 0 1 0 0 0  
## 2 1 0 0 0 0 1 0 0 0  
## 3 0 1 0 0 0 0 0 0 0  
## 4 0 1 0 0 1 0 0 0 0  
## 5 0 1 0 0 0 0 0 0 0  
## 6 0 0 1 0 0 0 0 0 0  
## Mystery Romance Sci-Fi Thriller War Western  
## 1 0 0 0 0 0 0  
## 2 0 0 0 0 0 0  
## 3 0 1 0 0 0 0  
## 4 0 1 0 0 0 0  
## 5 0 0 0 0 0 0  
## 6 0 0 0 1 0 0

It is clearly seen that each movie can correspond to either one or more than one genre.

### Ratings Matrix

In order to use the ratings data for building a recommendation engine with *recommenderlab*, the rating matrix is converted into a sparse matrix of type *realRatingMatrix*.

## 610 x 9724 rating matrix of class 'realRatingMatrix' with 100836 ratings.

## Recommendation Algorithms

The *recommenderlab* package contains many options for the recommendation algorithm. I am using recommenderRegistry$get\_entries function to get the recommendation algorithms and details on its usage:

## [1] "ALS\_realRatingMatrix" "ALS\_implicit\_realRatingMatrix"  
## [3] "IBCF\_realRatingMatrix" "LIBMF\_realRatingMatrix"   
## [5] "POPULAR\_realRatingMatrix" "RANDOM\_realRatingMatrix"   
## [7] "RERECOMMEND\_realRatingMatrix" "SVD\_realRatingMatrix"   
## [9] "SVDF\_realRatingMatrix" "UBCF\_realRatingMatrix"

## $ALS\_realRatingMatrix  
## [1] "Recommender for explicit ratings based on latent factors, calculated by alternating least squares algorithm."  
##   
## $ALS\_implicit\_realRatingMatrix  
## [1] "Recommender for implicit data based on latent factors, calculated by alternating least squares algorithm."  
##   
## $IBCF\_realRatingMatrix  
## [1] "Recommender based on item-based collaborative filtering."  
##   
## $LIBMF\_realRatingMatrix  
## [1] "Matrix factorization with LIBMF via package recosystem (https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html)."  
##   
## $POPULAR\_realRatingMatrix  
## [1] "Recommender based on item popularity."  
##   
## $RANDOM\_realRatingMatrix  
## [1] "Produce random recommendations (real ratings)."  
##   
## $RERECOMMEND\_realRatingMatrix  
## [1] "Re-recommends highly rated items (real ratings)."  
##   
## $SVD\_realRatingMatrix  
## [1] "Recommender based on SVD approximation with column-mean imputation."  
##   
## $SVDF\_realRatingMatrix  
## [1] "Recommender based on Funk SVD with gradient descend (https://sifter.org/~simon/journal/20061211.html)."  
##   
## $UBCF\_realRatingMatrix  
## [1] "Recommender based on user-based collaborative filtering."

Every algorithm has a particular use. I am using IBCF(Item Based Collaborative Filtering) and UBCF(User Based Collaborative Filtering) models.

The parameters of these two models are displayed below:

recommender\_models$IBCF\_realRatingMatrix$parameters

## $k  
## [1] 30  
##   
## $method  
## [1] "Cosine"  
##   
## $normalize  
## [1] "center"  
##   
## $normalize\_sim\_matrix  
## [1] FALSE  
##   
## $alpha  
## [1] 0.5  
##   
## $na\_as\_zero  
## [1] FALSE

recommender\_models$UBCF\_realRatingMatrix$parameters

## $method  
## [1] "cosine"  
##   
## $nn  
## [1] 25  
##   
## $sample  
## [1] FALSE  
##   
## $normalize  
## [1] "center"

## Data Exploration

The ratings data is factorized to get the unique ratings. Also, number of unique rating is calculated.

unique(vector\_ratings)

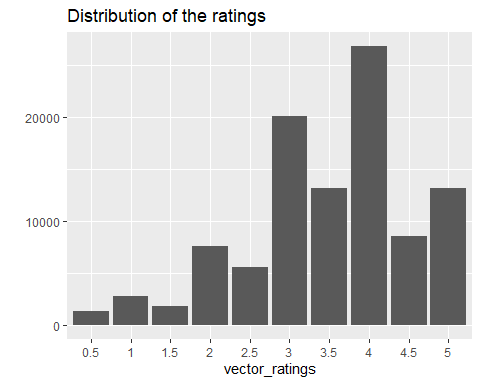
## [1] 4.0 0.0 4.5 2.5 3.5 3.0 5.0 0.5 2.0 1.5 1.0

table\_ratings

## vector\_ratings  
## 0 0.5 1 1.5 2 2.5 3 3.5 4   
## 5830804 1370 2811 1791 7551 5550 20047 13136 26818   
## 4.5 5   
## 8551 13211

As seen above, there are 11 unique score values. Most of the movies have zero ratings, it simply means no rating is available by users for movies.

According to the recommenderlab documentation, a rating equal to 0 represents a missing value or NULL value, so they are removed from the dataset before visualizing the results.



There are very few less than 3 rating scores, most movies are rated with a score of 3 or higher. The most common rating is 4.

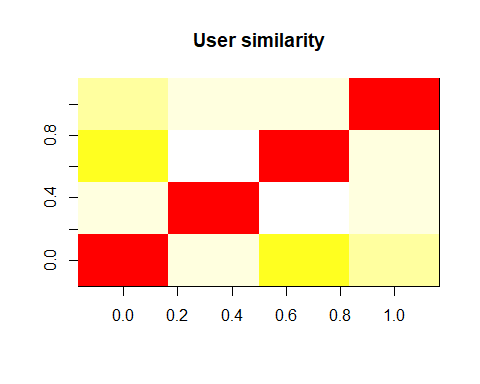
### Data exploration using Similarity function

Collaborative filtering algorithms are based on measuring the similarity between users or between items. For this purpose, *recommenderlab* contains the similarity function. The supported methods to compute similarities are *cosine, pearson*, and *jaccard*.

We start by getting the similarities between the first four users with each other. We create a similarity matrix using similarity function that uses a cosine distance.

In the given matrix, each row and each column correspond to a user, and each cell corresponds to the similarity between two users.

## 1 2 3 4  
## 1 0.0000000 1 0.7919033 0.9328096  
## 2 1.0000000 0 NA 1.0000000  
## 3 0.7919033 NA 0.0000000 1.0000000  
## 4 0.9328096 1 1.0000000 0.0000000

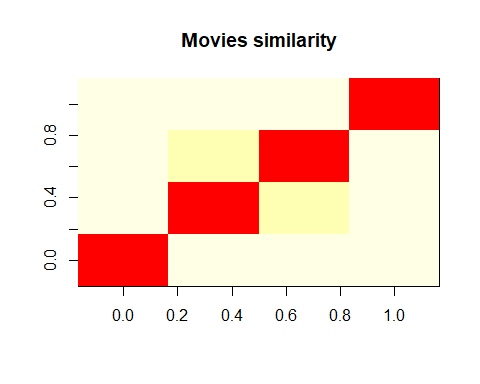


Here is heat map generated using the similarity matrix values.

The redder the cell is, the more similar two users are. Note that the diagonal is red, since it’s comparing each user with itself.

Using the same approach, we get the similarity between the first four movies.

## 1 2 3 4  
## 1 0.0000000 0.9644641 0.9715415 0.9838699  
## 2 0.9644641 0.0000000 0.9389013 0.9609877  
## 3 0.9715415 0.9389013 0.0000000 1.0000000  
## 4 0.9838699 0.9609877 1.0000000 0.0000000



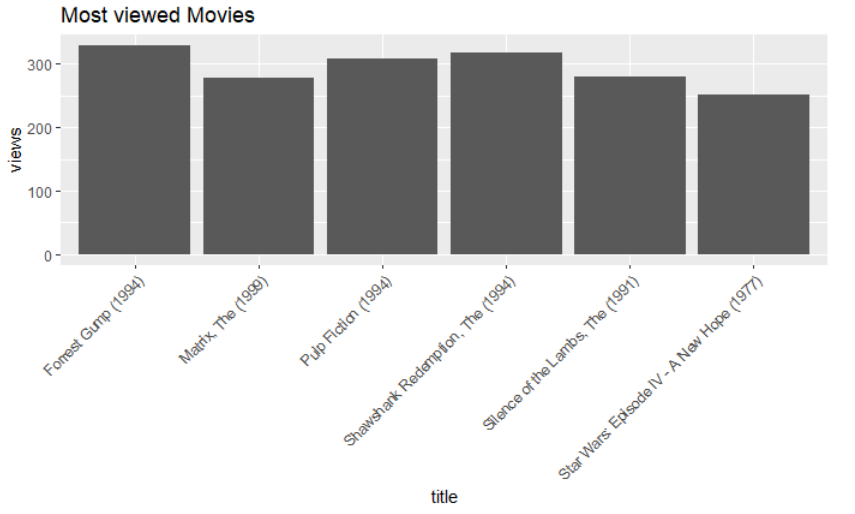
### The most viewed Movies

For getting the most viewed movies we are counting the views for each movie using the ratings matrix.

I have then created a table with parameters movie, views and title, where the title is acquired by subsetting the movies dataset.

## movie views title  
## 356 356 329 Forrest Gump (1994)  
## 318 318 317 Shawshank Redemption, The (1994)  
## 296 296 307 Pulp Fiction (1994)  
## 593 593 279 Silence of the Lambs, The (1991)  
## 2571 2571 278 Matrix, The (1999)  
## 260 260 251 Star Wars: Episode IV - A New Hope (1977)

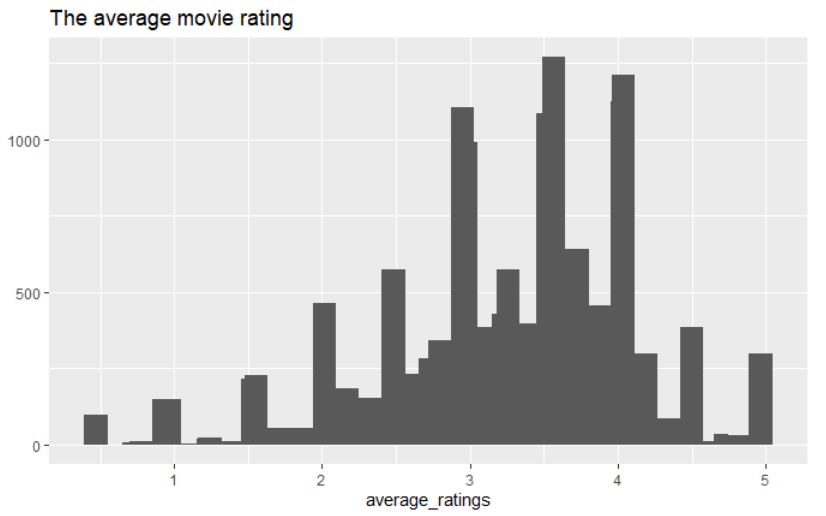
## [1] movie views title  
## <0 rows> (or 0-length row.names)



From the generated bar chart we can see that “Forrest Gump (1994)” is the most viewed movie and the second-most-viewed movie is “Shawshank Redemption,The (1994)” .

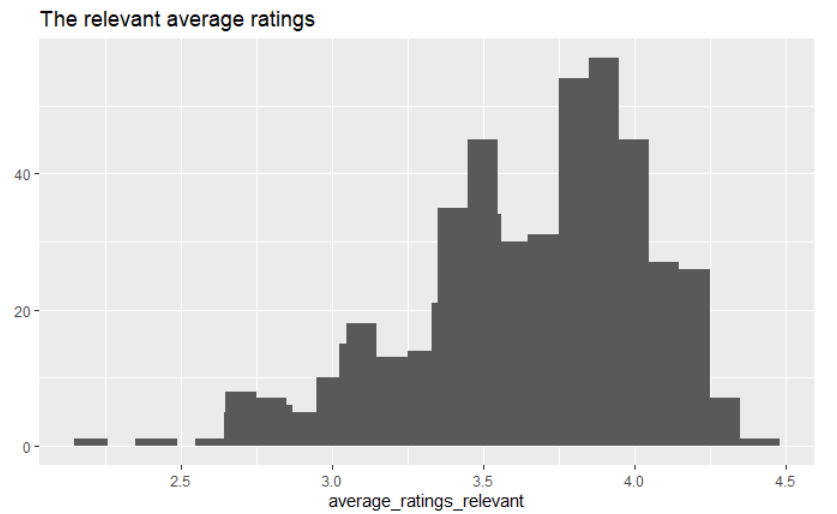
### Average Movie rating

The top-rated movies are now computed by the average rating for each of them.



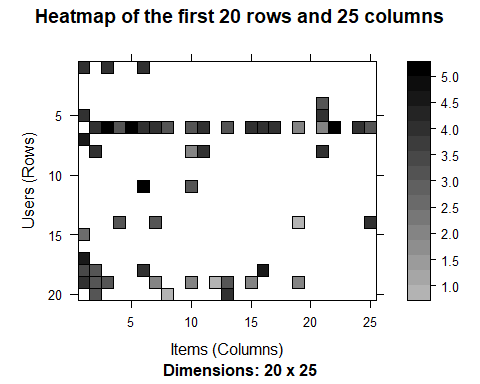
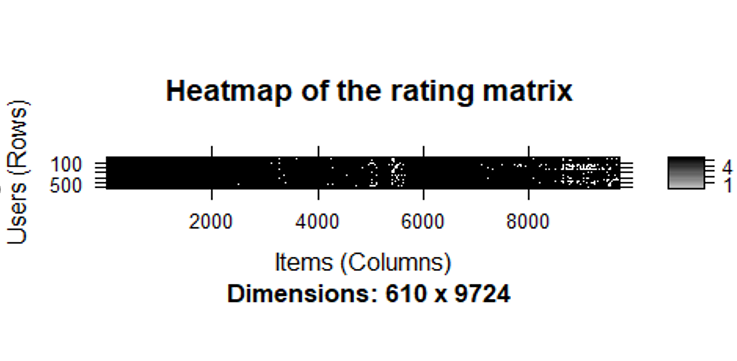
The highest value is around 3.5 in the first image showing the average movie rating, and also there are a few movies whose rating is either 1 or 5. Probably, the reason is that these movies received a rating from a few people only, so this is not considered.

Thus, the movies whose number of views is below a defined threshold of 50 are only considered. A subset of only relevant movies is created to get the relevant average movie ratings.



The second image above shows the relevant average ratings. All the rankings are between 2 and 4.5 approximately. The extremes were removed to get these relevant ratings. The highest value has now changed and is somewhere around 4.

### Heatmap of the rating matrix

The whole matrix of ratings is visualized using heat map whose colors represent the ratings. Each row of the matrix corresponds to a user, each column to a movie, and each cell to its rating. 

Since there are too many users and items, the first chart is hard to read. The second chart is built zooming in on the first rows and columns.

Some users saw more movies than the others. So, instead of displaying some random users and items, the most relevant users and items must be selected. Thus, only the users who have seen most of the movies and the movies that have been seen by most of users are considered.

To identify and select the most relevant users and movies:

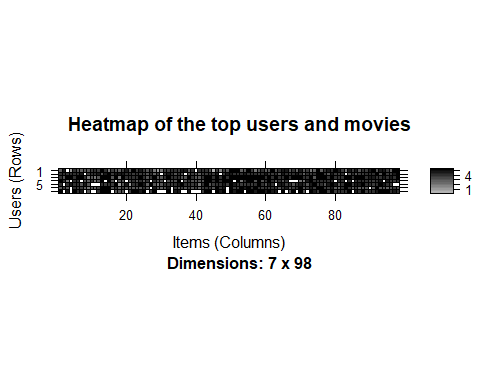
1. Determine the minimum number of movies per user.
2. Determine the minimum number of users per movie.
3. Select the users and movies matching these criteria.

## [1] "Minimum number of movies per user:"

## 99%   
## 1256.22

## [1] "Minimum number of users per movie:"

## 99%   
## 114.54



As seen from the above heat map, it’s pretty obvious that most of them have seen all the top movies. Some columns of the heatmap are darker than the others, meaning that these columns represent the highest-rated movies. The black rows represent users giving higher ratings and the white rows represent users fiving lower ratings.

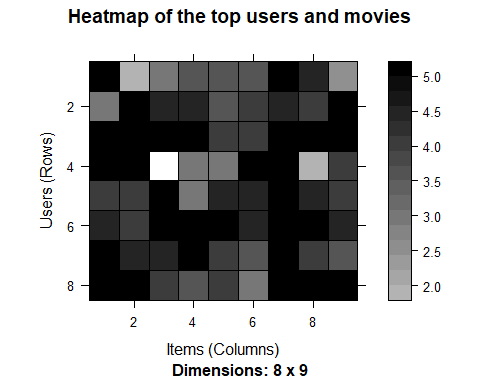
## Data Preparation

In order to select the most relevant data, I define the minimum number of users per rated movie as 50 and the minimum views number per movie as 50:

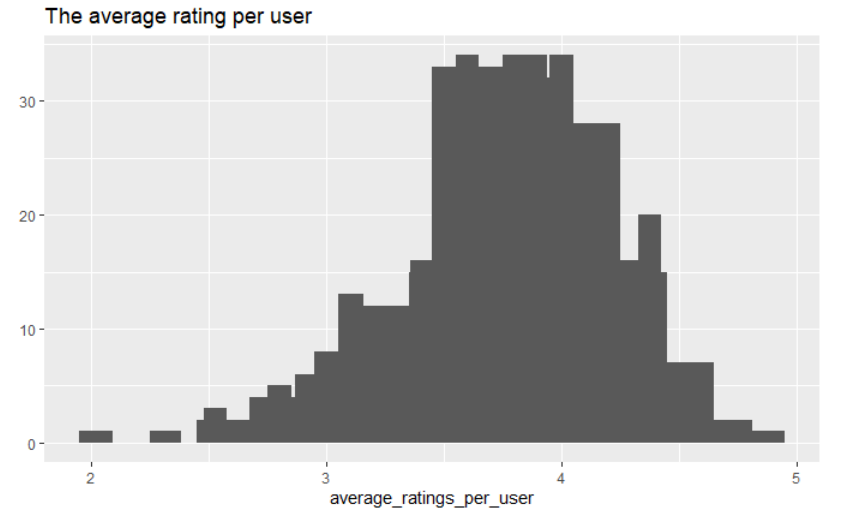
## 378 x 436 rating matrix of class 'realRatingMatrix' with 36214 ratings.

Such a selection of the most relevant data contains 378 users and 436 movies, compared to previous 610 users and 9724 movies.

Heat map is generated with the data of users and movies in the new matrix of the most relevant data:



In the heatmap, some rows are darker than the others. This might mean that some users give higher ratings to all the movies.



The average rating per user across all the users varies a lot, as the second chart above shows.

## ITEM-based Collaborative Filtering Model

Item-based CF is a model-based approach which produces recommendations based on the relationship between items inferred from the rating matrix. The assumption behind this approach is that users will prefer items that are similar to other items they like.

Collaborative filtering is a branch of recommendation that takes account of the information about different users. The word “collaborative” refers to the fact that users collaborate with each other to recommend items. In fact, the algorithms take account of user ratings and preferences.

The starting point is a rating matrix in which rows correspond to users and columns correspond to items. The core algorithm is based on these steps:

1. For each two items, measure how similar they are in terms of having received similar ratings by similar users
2. For each item, identify the k most similar items
3. For each user, identify the items that are most similar to the user’s purchases

This model is build using 80% of the dataset as a training set and remaining 20% as a test set.

### Building the recommendation model

The model-building step consists of calculating a similarity matrix containing all item-to-item similarities using a given similarity measure. Popular are again Pearson correlation and Cosine similarity. All pairwise similarities are stored in a nxn similarity matrix S.

Let’s have a look at the default parameters of IBCF model. Here, *k* is the number of items to compute the similarities among them in the first step. Aor each item, the algorithm identifies its *k* most similar items and stores the number. *method* is a similarity funtion, which is *Cosine* by default, may also be *pearson*. I create the model using the default parameters of method = Cosine and k=30.

## $k  
## [1] 30  
##   
## $method  
## [1] "Cosine"  
##   
## $normalize  
## [1] "center"  
##   
## $normalize\_sim\_matrix  
## [1] FALSE  
##   
## $alpha  
## [1] 0.5  
##   
## $na\_as\_zero  
## [1] FALSE

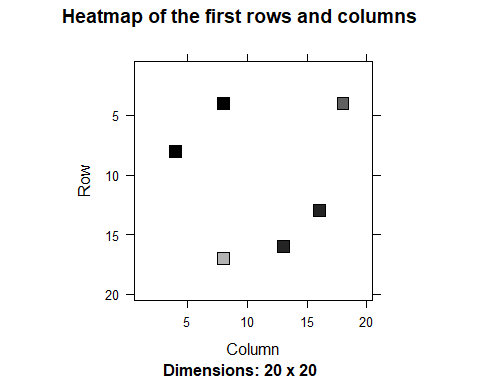
## Recommender of type 'IBCF' for 'realRatingMatrix'   
## learned using 316 users.

## [1] "Recommender"  
## attr(,"package")  
## [1] "recommenderlab"

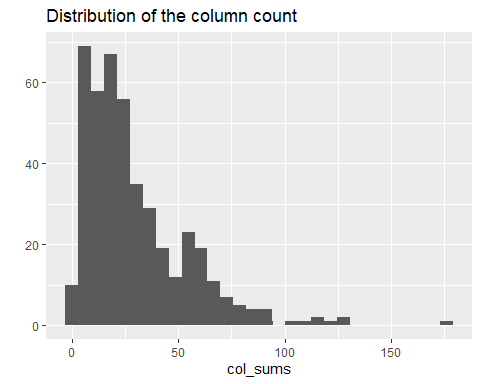
Exploring the recommender model:

## [1] "dgCMatrix"  
## attr(,"package")  
## [1] "Matrix"

## [1] 436 436



## row\_sums  
## 30   
## 436



*dgCMatrix* is a similarity matrix created by the model. Its dimensions are 436 x 436, which is equal to the number of items. The heatmap of 20 first items show that many values are equal to 0. The reason is that each row contains only k (30) elements that are greater than 0. The number of non-null elements for each column depends on how many times the corresponding movie was included in the top k of another movie. Thus, the matrix is not neccessarily simmetric, which is also the case in our model.

The chart of the distribution of the number of elements by column shows there are a few movies that are similar to many others.

### Applying recommender system on the dataset:

Applying the model on the test set to recommend movies to the users. Defining *n\_recommended* equal to 10 that specifies the number of movies to recommend to each user.

For each user, the algorithm extracts its rated movies. For each movie, it identifies all its similar items, starting from the similarity matrix. Then, the algorithm ranks each similar item in this way:

* Extract the user rating of each purchase associated with this item. The rating is used as a weight.
* Extract the similarity of the item with each purchase associated with this item.
* Multiply each weight with the related similarity.
* Sum everything up.

Then, the algorithm identifies the top 10 recommendations:

## Recommendations as 'topNList' with n = 10 for 62 users.

Let’s explore the results of the recommendations for the first user:

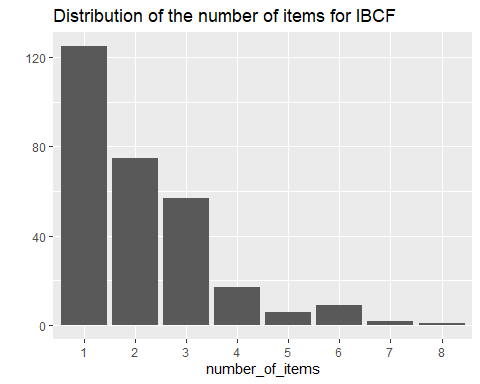
## [1] "Austin Powers: International Man of Mystery (1997)"  
## [2] "Royal Tenenbaums, The (2001)"   
## [3] "Italian Job, The (2003)"   
## [4] "Love Actually (2003)"   
## [5] "Charlie and the Chocolate Factory (2005)"   
## [6] "Up (2009)"   
## [7] "Mission: Impossible II (2000)"   
## [8] "Crimson Tide (1995)"   
## [9] "O Brother, Where Art Thou? (2000)"   
## [10] "Beauty and the Beast (1991)"

It’s also possible to define a matrix with the recommendations for each user. I visualize the recommendations for the first four users:

## [,1] [,2] [,3] [,4]  
## [1,] 1517 4239 3 1  
## [2,] 4979 33679 288 11  
## [3,] 6378 79132 349 21  
## [4,] 6942 2167 485 25  
## [5,] 30793 48 953 47  
## [6,] 68954 420 1035 95  
## [7,] 3623 553 1097 158  
## [8,] 161 589 2000 160  
## [9,] 4027 1573 5378 163  
## [10,] 595 1610 8665 231

Here, the columns represent the first 4 users, and the rows are the *movieId* values of recommended 10 movies.

We can now identify the most recommended movies. The following image shows the distribution of the number of items for Item-based CF:



## Movie title No of items  
## 440 Dave (1993) 8  
## 3 Grumpier Old Men (1995) 7  
## 48 Pocahontas (1995) 7  
## 17 Sense and Sensibility (1995) 6  
## 19 19 6  
## 160 160 6

Most of the movies have been recommended only a few times, and a few movies have been recommended more than 5 times.

Item-based CF recommends items on the basis of the similarity matrix. It’s an eager-learning model, that is, once it’s built, it doesn’t need to access the initial data. For each item, the model stores the k-most similar, so the amount of information is small once the model is built. This is an advantage in the presence of lots of data.

In addition, this algorithm is efficient and scalable, so it works well with big rating matrices. Furthermore, item-based CF is successfully applied in large scale recommender systems (e.g., by Amazon.com).

## USER-based Collaborative Filtering Model

User-based CF is a memory-based algorithm which tries to mimics word-of-mouth by analyzing rating data from many individuals. The assumption is that users with similar preferences will rate items similarly. Thus, missing ratings for a user can be predicted by first finding a neighborhood of similar users and then aggregate the ratings of these users to form a prediction.

The neighborhood is defined in terms of similarity between users, either by taking a given number of most similar users (k nearest neighbors) or all users within a given similarity threshold. Popular similarity measures for CF are the Pearson correlation coefficient and the Cosine similarity.

Rate the movies rated by the most similar users. The rating is the average rating among similar users and the approaches are:

* Average rating
* Weighted average rating, using the similarities as weights

### Building the recommendation system:

Checking the default parameters of USER-BASED CF model. Here, *nn* is a number of similar users, and *method* is a similarity function, which is *cosine* by default. This recommender model is built using the default parameters on the training set.

## $method  
## [1] "cosine"  
##   
## $nn  
## [1] 25  
##   
## $sample  
## [1] FALSE  
##   
## $normalize  
## [1] "center"

## Recommender of type 'UBCF' for 'realRatingMatrix'   
## learned using 316 users.

## 316 x 436 rating matrix of class 'realRatingMatrix' with 30402 ratings.  
## Normalized using center on rows.

### Applying the recommender model on the test set

In the same way as the ITEM-BASED CF, determining the top ten recommendations for each new user in the test set.

## Recommendations as 'topNList' with n = 10 for 62 users.

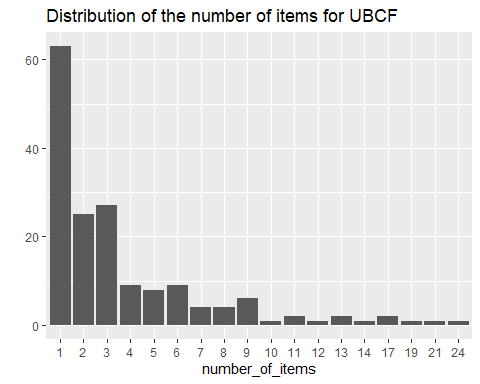
### Explore results

Let’s take a look at the first four users:

## [,1] [,2] [,3] [,4]  
## [1,] 588 457 5952 457  
## [2,] 1097 356 49272 260  
## [3,] 553 349 7438 608  
## [4,] 1200 50 2959 364  
## [5,] 223 260 4963 50  
## [6,] 912 4226 8961 223  
## [7,] 25 34 356 47  
## [8,] 1035 588 4226 1  
## [9,] 62 364 6874 337  
## [10,] 2804 225 6711 1198

The above matrix contain *movieId* of each recommended movie (rows) for the first four users (columns) in our test dataset.

I also compute how many times each movie got recommended and build the related frequency histogram:



Compared with the ITEM-BASED CF, the distribution has a longer tail. This means that there are some movies that are recommended much more often than the others. The maximum is more than 20, compared to 10-ish for ITEM-BASED CF.

Let’s take a look at the top titles:

## Movie title No of items  
## 457 Fugitive, The (1993) 24  
## 318 Shawshank Redemption, The (1994) 21  
## 223 Clerks (1994) 19  
## 110 Braveheart (1995) 17

Comparing the results of USER-BASED CF with ITEM-BASED CF helps find some useful insight on different algorithms. USER-BASED CF needs to access the initial data. Since it needs to keep the entire database in memory, it doesn’t work well in the presence of a big rating matrix. Also, building the similarity matrix requires a lot of computing power and time.

However, USER-BASED CF’s accuracy is proven to be slightly more accurate than ITEM-BASED CF (I will also discuss it in the next section), so it’s a good option if the dataset is not too big.

## Evaluating the Recommender Systems

Given a rating matrix R, recommender algorithms are evaluated by first partitioning the users(rows) in R into two sets training and test set. The rows of R corresponding to the training users are used to learn the recommender model.

In order to compare their performances and choose the most appropriate model:

* Prepare the data to evaluate performance
* Evaluate the performance of some models
* Choose the best performing models
* Optimize model parameters

### Preparing the data to evaluate models

To determine how to split the users in training and test sets, there are several approaches:

* splitting the data
* bootstrapping the data
* k-fold cross-validation

### Splitting the data

In this method, we can randomly assign a predefined proportion of the users to the training set and all others to the test set. Splitting the data into training and test sets is often done using a 80/20 proportion. For each user in the test set, we need to define how many items to use to generate recommendations. For this, the minimum number of items rated by users as checked to be sure there will be no users with no items to test.

min(rowCounts(ratings\_movies))

## [1] 11

items\_to\_keep <- 5 #number of items to generate recommendations  
rating\_threshold <- 3 # threshold with the minimum rating that is considered good  
n\_eval <- 1 #number of times to run evaluation  
  
eval\_sets <- evaluationScheme(data = ratings\_movies,   
 method = "split",  
 train = percentage\_training,   
 given = items\_to\_keep,   
 goodRating = rating\_threshold,   
 k = n\_eval)   
eval\_sets

## Evaluation scheme with 5 items given  
## Method: 'split' with 1 run(s).  
## Training set proportion: 0.800  
## Good ratings: >=3.000000  
## Data set: 378 x 436 rating matrix of class 'realRatingMatrix' with 36214 ratings.

getData(eval\_sets, "train") # training set

## 302 x 436 rating matrix of class 'realRatingMatrix' with 28131 ratings.

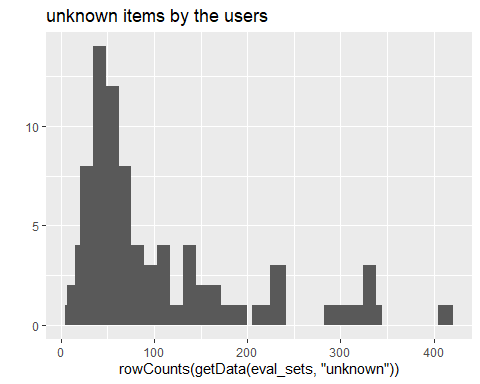
getData(eval\_sets, "known") # set with the items used to build the recommendations

## 76 x 436 rating matrix of class 'realRatingMatrix' with 380 ratings.

getData(eval\_sets, "unknown") # set with the items used to test the recommendations

## 76 x 436 rating matrix of class 'realRatingMatrix' with 7703 ratings.

qplot(rowCounts(getData(eval\_sets, "unknown"))) +   
 geom\_histogram(binwidth = 10) +   
 ggtitle("unknown items by the users")



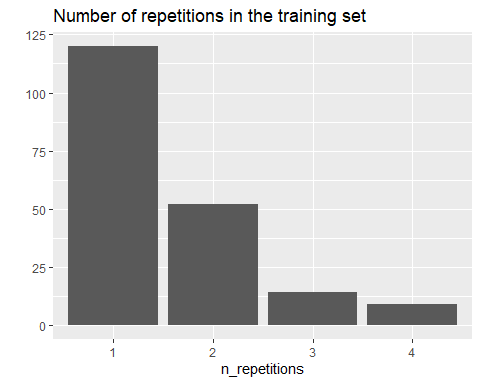
The above image displays the unknown items by the users, which varies a lot.

### Bootstrapping the data

In bootstrapping, we can sample from test set with replacement to create the training set and then use the users not present in the training set as the test set.

This procedure has the advantage that for smaller data sets we can create larger training sets and still have users left for testing. The same user can be sampled more than once and, if the training set has the same size as it did earlier, there will be more users in the test set.

eval\_sets <- evaluationScheme(data = ratings\_movies,   
 method = "bootstrap",   
 train = percentage\_training,   
 given = items\_to\_keep,  
 goodRating = rating\_threshold,   
 k = n\_eval)  
  
table\_train <- table(eval\_sets@runsTrain[[1]])  
n\_repetitions <- factor(as.vector(table\_train))  
qplot(n\_repetitions) +   
 ggtitle("Number of repetitions in the training set")



The above chart shows that most of the users have been sampled fewer than four times.

### k-Fold Cross-validation

In k-fold cross-validation, we split Users into k sets (called folds) of approximately the same size. Then we evaluate k times, always using one fold for testing and all other folds for leaning. The k results can be averaged. This approach makes sure that each user is at least once in the test set and the averaging produces more robust results and error estimates. It is the most accurate approach, although it’s computationally heavier.

n\_fold <- 4  
eval\_sets <- evaluationScheme(data = ratings\_movies,   
 method = "cross-validation",  
 k = n\_fold,   
 given = items\_to\_keep,   
 goodRating = rating\_threshold)  
size\_sets <- sapply(eval\_sets@runsTrain, length)  
size\_sets

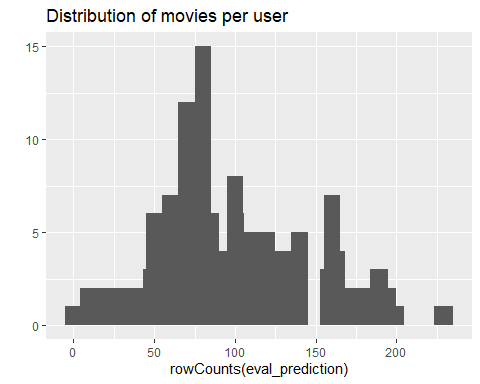
## [1] 282 282 282 282

Using 4-fold approach, we get four sets of the same size 282.

### Evavluating the ratings

In this project k-fold approach is used for evaluation.

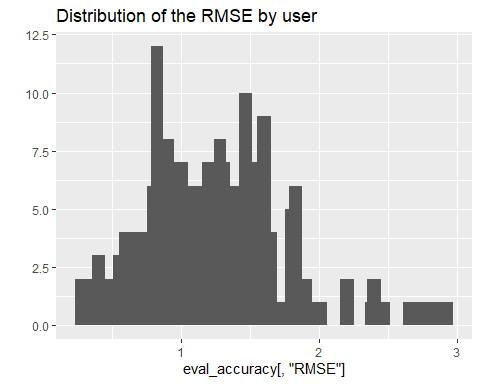
Re-defining the evaluation sets to build ITEM-BASED CF model and creating a matrix with predicted ratings.



The above image displays the distribution of movies per user in the matrix of predicted ratings.

Now, computing the accuracy measures for each user. Most of the RMSEs (Root mean square errors) are in the range of 0.5 to 1.8:

## RMSE MSE MAE  
## [1,] 1.9148542 3.6666667 1.3333333  
## [2,] 1.5205814 2.3121679 1.2069902  
## [3,] 0.7071068 0.5000000 0.5000000  
## [4,] 1.8187062 3.3076923 1.3076923  
## [5,] 1.2916013 1.6682340 1.0131967  
## [6,] 0.6944064 0.4822003 0.5405519



In order to have a performance index for the whole model, I specify *byUser* as FALSE and compute the average indices:

## RMSE MSE MAE   
## 1.392599 1.939332 1.039523

The measures of accuracy are useful to compare the performance of different models on the same data.

### Evaluating the recommendations

Another way to measure accuracies is by comparing the recommendations with the purchases having a positive rating. For this, I have used a prebuilt *evaluate* function in *recommenderlab* library. The function evaluates the recommender performance depending on the number *n* of items to recommend to each user. I use *n* as a sequence n = seq(10, 100, 10). The first rows of the resulting performance matrix is presented below:

## ITEM-BASED CF run fold/sample [model time/prediction time]  
## 1 [0.39sec/0.02sec]   
## 2 [0.37sec/0.04sec]   
## 3 [0.32sec/0.03sec]   
## 4 [0.45sec/0.02sec]

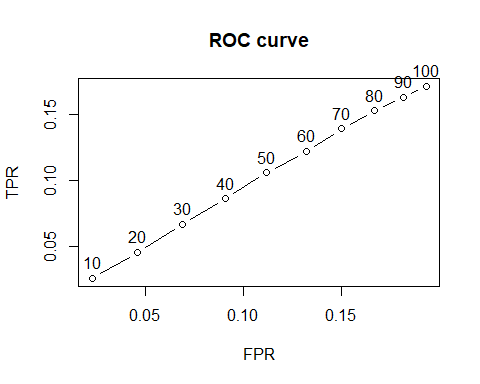
## TP FP FN TN precision recall TPR  
## 10 1.770833 8.083333 69.32292 351.8229 0.1803509 0.02688444 0.02688444  
## 20 3.156250 16.395833 67.93750 343.5104 0.1627911 0.04493197 0.04493197  
## 30 4.625000 24.416667 66.46875 335.4896 0.1618227 0.06879730 0.06879730  
## 40 5.906250 32.406250 65.18750 327.5000 0.1592789 0.08909887 0.08909887  
## 50 7.395833 39.906250 63.69792 320.0000 0.1625679 0.11175631 0.11175631  
## 60 8.479167 47.281250 62.61458 312.6250 0.1595417 0.12617534 0.12617534  
## FPR  
## 10 0.02262760  
## 20 0.04588307  
## 30 0.06820589  
## 40 0.09032537  
## 50 0.11106022  
## 60 0.13148483

In order to have a look at all the splits at the same time, I sum up the indices of columns TP, FP, FN and TN:

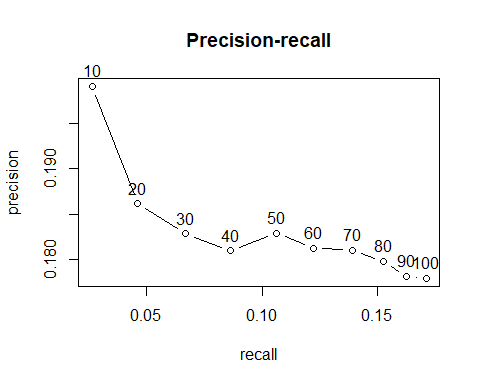
## TP FP FN TN  
## 10 7.822917 31.51042 317.1771 1367.490  
## 20 14.583333 63.80208 310.4167 1335.198  
## 30 21.302083 95.25000 303.6979 1303.750  
## 40 27.760417 126.04167 297.2396 1272.958  
## 50 34.645833 155.13542 290.3542 1243.865  
## 60 40.364583 183.02083 284.6354 1215.979

Finally, I plot the ROC and the precision/recall curves:

plot(results, annotate = TRUE, main = "ROC curve")



plot(results, "prec/rec", annotate = TRUE, main = "Precision-recall")



If a small percentage of rated movies is recommended, the precision decreases. On the other hand, the higher percentage of rated movies is recommended the higher is the recall.

## Comparing models

In order to compare different models, I define them as a following list:

* Item-based collaborative filtering, using the Cosine as the distance function
* Item-based collaborative filtering, using the Pearson correlation as the distance function
* User-based collaborative filtering, using the Cosine as the distance function
* User-based collaborative filtering, using the Pearson correlation as the distance function
* Random recommendations to have a base line

Then, I define a different set of numbers for recommended movies (n\_recommendations <- c(1, 5, seq(10, 100, 10))), run and evaluate the models:

## IBCF run fold/sample [model time/prediction time]  
## 1 [0.34sec/0.02sec]   
## 2 [0.34sec/0.02sec]   
## 3 [0.37sec/0.02sec]   
## 4 [0.53sec/0.04sec]   
## IBCF run fold/sample [model time/prediction time]  
## 1 [0.47sec/0.01sec]   
## 2 [0.46sec/0.03sec]   
## 3 [0.45sec/0.02sec]   
## 4 [0.38sec/0.03sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/0.11sec]   
## 2 [0sec/0.09sec]   
## 3 [0.01sec/0.1sec]   
## 4 [0.01sec/0.1sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0.01sec/0.11sec]   
## 2 [0sec/0.11sec]   
## 3 [0sec/0.1sec]   
## 4 [0sec/0.11sec]   
## RANDOM run fold/sample [model time/prediction time]  
## 1 [0sec/0.03sec]   
## 2 [0sec/0.04sec]   
## 3 [0sec/0.03sec]   
## 4 [0sec/0.03sec]

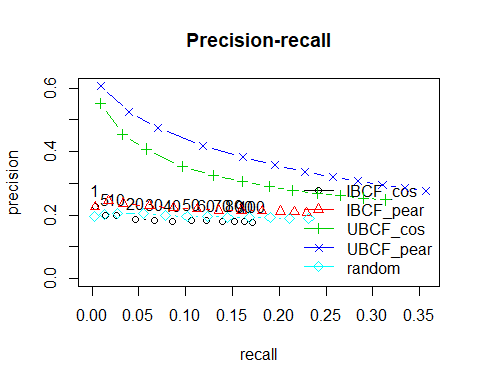
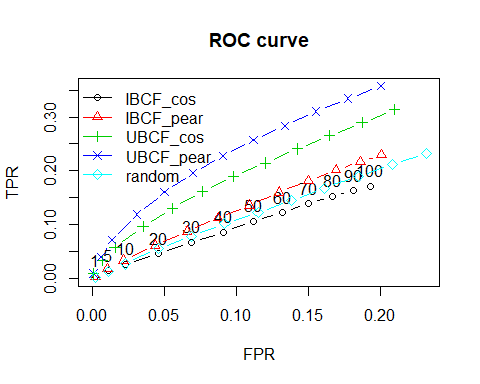
## IBCF\_cos IBCF\_pear UBCF\_cos UBCF\_pear random   
## TRUE TRUE TRUE TRUE TRUE

The following table presents as an example the first rows of the performance evaluation matrix for the ITEM-BASED CF with Cosine distance:

## precision recall TPR FPR  
## 1 0.2275476 0.003550591 0.003550591 0.002208282  
## 5 0.2000056 0.014036587 0.014036587 0.011609759  
## 10 0.1990743 0.026207999 0.026207999 0.023009845  
## 20 0.1861723 0.045911542 0.045911542 0.046173680  
## 30 0.1828426 0.066904446 0.066904446 0.068802281  
## 40 0.1809724 0.086156842 0.086156842 0.090994862

### Identifying the most suitable model

I compare the models by building a chart displaying their ROC curves and Precision/recall curves.



A good performance index is the area under the curve (AUC), that is, the area under the ROC curve. Even without computing it, the chart shows that the highest is USER-BASED CF with Pearson distance, so it’s the best-performing technique.

The UBCF with pearson distance is still the top model. Depending on what is the main purpose of the system, an appropriate number of items to recommend should be defined.

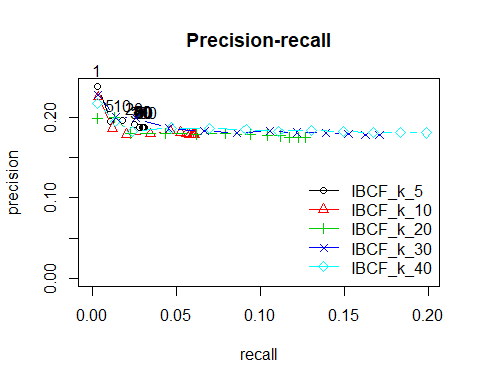
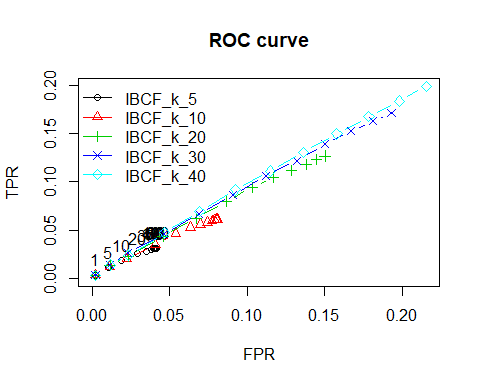
## Optimizing a numeric parameter

ITEM-BASED CF takes account of the k-closest items. I will explore more values, ranging between 5 and 40, in order to tune this parameter:

vector\_k <- c(5, 10, 20, 30, 40)  
models\_to\_evaluate <- lapply(vector\_k, function(k){  
 list(name = "IBCF",  
 param = list(method = "cosine", k = k))  
})  
names(models\_to\_evaluate) <- paste0("IBCF\_k\_", vector\_k)

Now I build and evaluate the same IBCF/cosine models with different values of the k-closest items:

## IBCF run fold/sample [model time/prediction time]  
## 1 [0.38sec/0.03sec]   
## 2 [0.53sec/0.02sec]   
## 3 [0.48sec/0.02sec]   
## 4 [0.41sec/0.02sec]   
## IBCF run fold/sample [model time/prediction time]  
## 1 [0.4sec/0.02sec]   
## 2 [0.41sec/0.03sec]   
## 3 [0.37sec/0.02sec]   
## 4 [0.35sec/0.04sec]   
## IBCF run fold/sample [model time/prediction time]  
## 1 [0.55sec/0.03sec]   
## 2 [0.41sec/0.03sec]   
## 3 [0.48sec/0.05sec]   
## 4 [0.36sec/0.01sec]   
## IBCF run fold/sample [model time/prediction time]  
## 1 [0.37sec/0.02sec]   
## 2 [0.4sec/0.01sec]   
## 3 [0.4sec/0.02sec]   
## 4 [0.35sec/0.01sec]   
## IBCF run fold/sample [model time/prediction time]  
## 1 [0.37sec/0.02sec]   
## 2 [0.36sec/0.01sec]   
## 3 [0.32sec/0.03sec]   
## 4 [0.33sec/0.03sec]



Based on the ROC curve’s plot, the k having the biggest AUC is 10. Another good candidate is 5, but it can never have a high TPR. This means that, even if we set a very high n value, the algorithm won’t be able to recommend a big percentage of items that the user liked. The ITEM-BASED CF with k = 5 recommends only a few items similar to the purchases. Therefore, it can’t be used to recommend many items.

Based on the precision/recall plot, k should be set to 10 to achieve the highest recall. If we are more interested in the precision, we set k to 5.

## Conclusion

The R extension package recommenderlab which is especially geared towards developing and testing recommender algorithms is the core of this project. The package allows to create evaluation schemes following accepted methods and then use them to evaluate and compare recommender algorithms.

Using the recommenderlab library we just created a movie recommender system based on the collaborative filtering algorithm. We have successfully recommended 10 movies that the user is likely to prefer. The recommenderlab library could be used to create recommendations using other datasets apart from the MovieLens dataset.

With enough data, collaborative filtering provides a powerful way for data scientists to recommend new products or items to users. If you have well-detailed metadata about your products, you could also use a content-based approach to recommendations.

Recommendation Engine is your companion and advisor to help you make the right choices by providing you tailored options and creating a personalized experience for you. It is beyond any doubt that recommendation engines are getting popular and critical in the new age of things.

## References

* 1. "recommenderlab: A Framework for Developing and Testing Recommendation Algorithms", Michael Hahsler (2019), SMU - Cran-r-vignettes
  2. <https://hackernoon.com/introduction-to-recommender-system-part-1-collaborative-filtering-singular-value-decomposition-44c9659c5e75>
  3. <https://blog.datasciencedojo.com/movie-recommender-systems/>
  4. <https://medium.com/@james_aka_yale/the-4-recommendation-engines-that-can-predict-your-movie-tastes-bbec857b8223>
  5. Deshpande M, Karypis G (2004). \Item-based top-N recommendation algorithms." ACM Translations on Information Systems, 22(1), 143{177. ISSN 1046-8188.
  6. <https://github.com/kphaser/movieRecommender>
  7. <https://www.sas.com/content/dam/SAS/en_us/doc/conclusionpaper1/recommendation-systems-107451.pdf>

