1 INTRODUCTION

1.1 INTRODUCTION TO RECOMMENDER SYSTEMS

Recommender systems are the software tools and techniques that provide suggestions, such as useful products on e-commerce websites, videos on YouTube, friends' recommendations on Facebook, book recommendations on Amazon, news recommendations on online news websites, and the list goes on. These recommendations are specific to you and differ from user to user.

The main goal of recommender systems is to provide suggestions to online users to make better decisions from many alternatives available over the Web. A better recommender system is directed more towards personalized recommendations by taking into consideration the available digital footprint of the user and information about a product, such as specifications, feedback from the users, comparison with other products, and so on, before making recommendations.

1.2 COLLABORATIVE FILTERING RECOMMENDER SYSTEMS

The basic idea of these systems is that, if two users share the same interests in the past, that is, they liked the same book, they will also have similar tastes in the future. If, for example, user A and user B have a similar purchase history and user A recently bought a book that user B has not yet seen, the basic idea is to propose this book to user B. The book recommendations on Amazon are one good example of this type of recommender system.

In this type of recommendation, filtering items from a large set of alternatives is done collaboratively between user’s preferences. Such systems are called collaborative filtering recommender systems.

While dealing with collaborative filtering recommender systems, we will learn about the following aspects:

- How to calculate the similarity between users
- How to calculate the similarity between items
- How do we deal with new items and new users whose data is not known

The collaborative filtering approach considers only user preferences and does not take into account the features or contents of the items being recommended. This approach requires a large set of user preferences for more accurate results.

1.3 ITEM-BASED RECOMMENDER SYSTEMS

This system recommends items to users by taking the similarity of items and user profiles into consideration. In simpler terms, the system recommends items similar to those that the user has liked in the past. The similarity of items is calculated based on the features associated with the other compared items and is matched with the user's historical preferences.

As an example, we can assume that, if a user has positively rated a movie that belongs to the action genre, then the system can learn to recommend other movies from the action genre.

While building a content-based recommendation system, we take into consideration the following questions:

- How do we create similarity between items?
- How do we create and update user profiles continuously?

This technique doesn't take into consideration the user's neighborhood preferences. Hence, it doesn't require a large user group's preference for items for better recommendation accuracy. It only considers the user's past preferences and the properties/features of the items.

1.4 KNOWLEDGE-BASED RECOMMENDER SYSTEMS

These types of recommender systems are employed in specific domains where the purchase history of the users is smaller. In such systems, the algorithm takes into consideration the knowledge about the items, such as features, user preferences asked explicitly, and recommendation criteria, before giving recommendations. The accuracy of the model is judged based on how useful the recommended item is to the user. Take, for example, a scenario in which you are building a recommender system that recommends household electronics, such as air conditioners, where most of the users will be first timers. In this case, the system considers features of the items, and user profiles are generated by obtaining additional information from the users, such as specifications, and then recommendations are made.

Before building these types of recommender systems, we take into consideration the following questions:

- What kind of information about the items is taken into the model?
- How are user preferences captured explicitly?

1.5 HYBRID SYSTEMS

We build hybrid recommender systems by combining various recommender systems to build a more robust system. By combining various recommender systems, we can eliminate the disadvantages of one system with the advantages of another system and thus build a more robust system. For example, by combining collaborative filtering methods, where the model fails when new items don't have ratings, with content-based systems, where feature information about the items is available, new items can be recommended more accurately and efficiently.

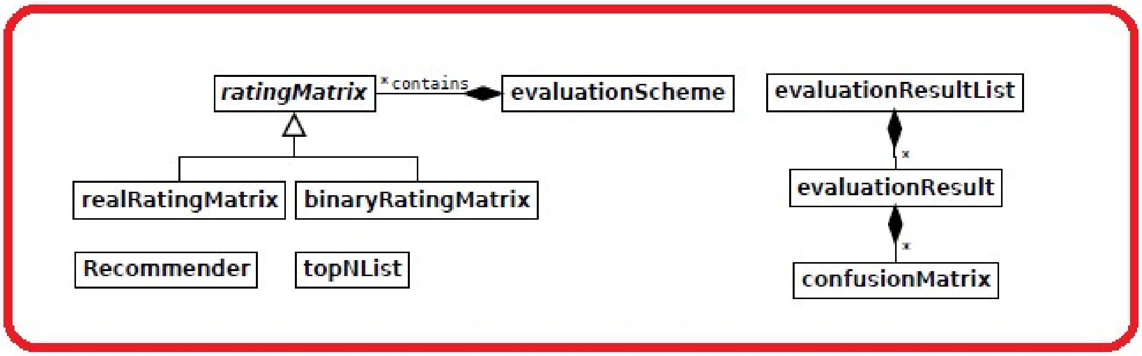
Before building a hybrid model, we consider the following questions:

- What techniques should be combined to achieve the business solution?
- How should we combine various techniques and their results for better predictions?

1.6 R PACKAGE FOR RECOMMENDATION – RECOMMENDERLAB

For this project, we will be using RecommenderLab package of R Statistical programming language. R is a language and environment for statistical computing and graphics.

RecommenderLab is a R package that provides a research infrastructure to test and develop recommender algorithms including UBCF, IBCF, FunkSVD and association rule-based algorithms.



UML diagram of Recommenderlab package

The package uses the abstract ratingMatrix to provide a common interface for rating data. ratingMatrix implements many methods typically available for matrix-like objects. For example, dim(), dimnames(), colCounts(), rowCounts(), colMeans(), rowMeans(), colSums() and rowSums(). Additionally sample() can be used to sample from users (rows) and image() produces an image plot.

For ratingMatrix we provide two concrete implementations realRatingMatrix and binaryRatingMatrix to represent different types of rating matrices.

- realRatingMatrix - Implements a rating matrix with real valued ratings stored in sparse format defined in package Matrix. Sparse matrices in Matrix typically do not store 0s explicitly, however for realRatingMatrix we use these sparse matrices such that instead of 0s, NAs are not explicitly stored.

- binaryRatingMatrix - implements a 0-1 rating matrix using the implementation of itemMatrix defined in package arules. itemMatrix stores only the ones and internally uses a sparse representation from package Matrix. With this class structure recommenderlab can be easily extended to other forms of rating matrices with different concepts for efficient storage in the future.

1.6.1 Exploring parameters of RecommenderLab

```
recommender_models <- recommenderRegistry$get_entries(dataType =
"realRatingMatrix")

names(recommender_models)

Output -

[1] "ALS_realRatingMatrix"          "ALS_implicit_realRatingMatrix"

[3] "IBCF_realRatingMatrix"         "POPULAR_realRatingMatrix"

[5] "RANDOM_realRatingMatrix"       "RERECOMMEND_realRatingMatrix"

[7] "SVD_realRatingMatrix"          "SVDF_realRatingMatrix"

[9] "UBCF_realRatingMatrix"
```

```
lapply(recommender_models, "[[", "description")

Output -

$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."


$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."

$UBCF_realRatingMatrix
[1] "Recommender based on user-based collaborative filtering."
```

The table above shows the Recommender algorithms that are supported by the RecommenderLab package. For our project, we will be using the User-based collaborative filtering and item-based collaborative filtering algorithms and compare the evaluation models for MovieLens dataset.

2 INTRODUCTION DATASET TO BE USED FOR MODEL – MOVIELENS DATASET

The datasets for this project can be downloaded from the following site: http://grouplens.org/datasets/movielens/latest.

There are two sets of data having different number of observations –

1. Small dataset - It contains 105339 ratings and 6138 tag applications across 10329 movies. These data were created by 668 users between April 03, 1996 and January 09, 2016.

2. Large dataset- It contains 22884377 ratings and 586994 tag applications across 34208 movies. These data were created by 247753 users between January 09, 1995 and January 29, 2016.

For initial model building and validation, the smaller dataset is used.

The data are contained in four files: links.csv, movies.csv, ratings.csv and tags.csv.

A brief description of the data files is as below –

File Name	Description
ratings.csv	All ratings are contained in the file ratings.csv. Each line of this file after the header row represents one rating of one movie by one user, and has the following format: userId, movieId, rating, timestamp.
movies.csv	Movie information is contained in the file movies.csv. Each line of this file after the header row represents one movie, and has the following format: movieId, title, genres. Movie titles are entered manually or imported from https://www.themoviedb.org/, and include the year of release in parentheses.Errors and inconsistencies may exist in these titles.
links.csv	Identifiers that can be used to link to other sources of movie data are contained in the file links.csv. Each line of this file after the header row represents one movie, and has the following format: movieId, imdbId, tmdbId.
tags.csv	All tags are contained in the file tags.csv. Each line of this file after the header row represents one tag applied to one movie by one user, and has the following format: userId, movieId, tag, timestamp.

We only use the files movies.csv and ratings.csv to build a recommendation system. These files are used to build the user item ratings matrix, needed by the colloborative algorithms that are available with the Recommenderlab package.

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

```
summary(movies)
Output -
movieId          title              genres
 Min.   :     1   Length:10329       Length:10329
 1st Qu.:  3240   Class :character   Class :character
 Median :  7088   Mode  :character   Mode  :character
 Mean   : 31924
 3rd Qu.: 59900
 Max.   :149532
```

```
head(movies)
Output -
```

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

And here is a summary and a head of ratings:

```
summary(ratings)
Output -
userId          movieId              rating         timestamp
 Min.   :  1.0   Min.   :      1   Min.   :0.500   Min.   :8.286e+08
```

```
 1st Qu.:192.0   1st Qu.:  1073   1st Qu.:3.000   1st Qu.:9.711e+08
 Median :383.0   Median :  2497   Median :3.500   Median :1.115e+09
 Mean   :364.9   Mean   : 13381   Mean   :3.517   Mean   :1.130e+09
 3rd Qu.:557.0   3rd Qu.:  5991   3rd Qu.:4.000   3rd Qu.:1.275e+09
 Max.   :668.0   Max.   :149532   Max.   :5.000   Max.   :1.452e+09
```

```
head(ratings)
```

Output -

	userId <int>	movieId <int>	rating <dbl>
1	1	16	4.0
2	1	24	1.5
3	1	32	4.0
4	1	47	4.0
5	1	50	4.0
6	1	110	4.0

6 rows

3 INITIAL DATA WRANGLING

3.1 EXTRACT A LIST OF GENRES

Use one-hot encoding to create a matrix of corresponding genres for each movie. This will help us to generate the list of recommendations for movies based on genres preferred by the user. Even if we do not use the matrix to give genre based recommendation, we will be using it to find similarities between users for the rating matrix.

The summary of the genre matrix is as below.

	movieId	title	Action	Adventure	Animation	Children
	<int>	<chr>	<int>	<int>	<int>	<int>
1	1	Toy Story (1995)	0	1	1	1
2	2	Jumanji (1995)	0	1	0	0
3	3	Grumpier Old Men (1995)	0	0	0	0
4	4	Waiting to Exhale (1995)	0	0	0	0
5	5	Father of the Bride Part II (1995)	0	0	0	0
6	6	Heat (1995)	1	0	0	0

It is seen from the ratings matrix above that each movie can correspond to one or more genres.

3.1.1 Converting ratings matrix in a proper format

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

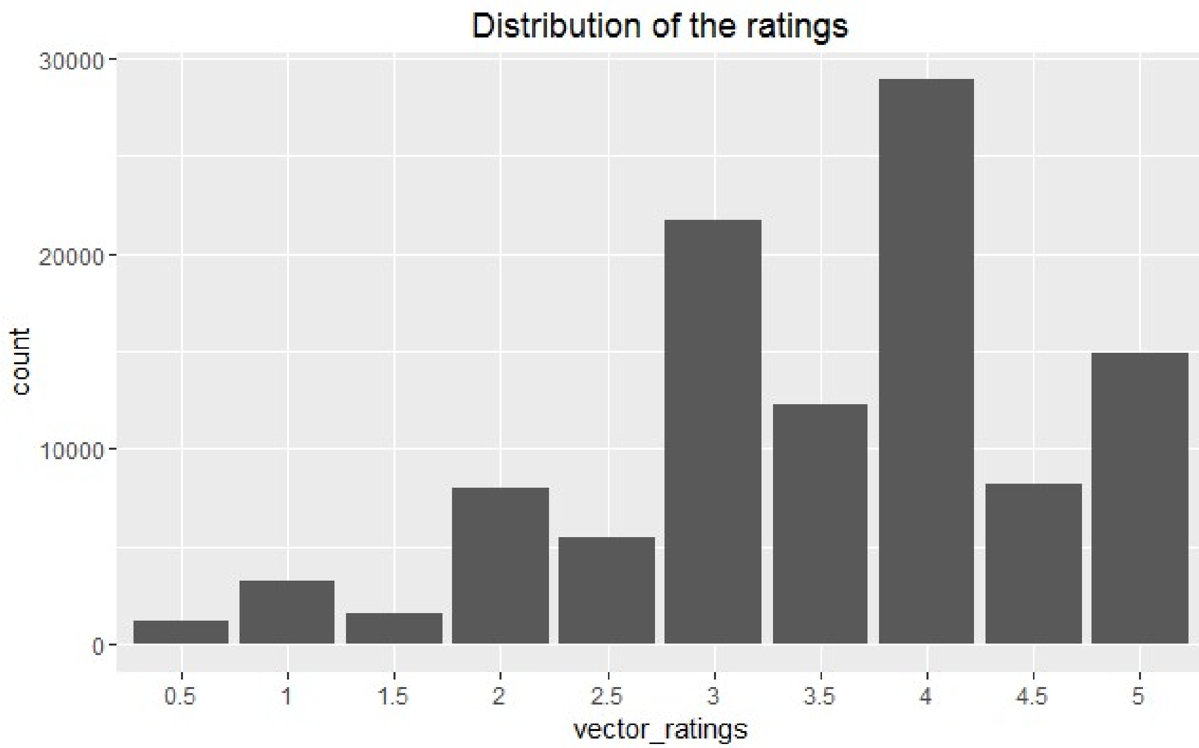
[1] 0.0 5.0 4.0 3.0 4.5 1.5 2.0 3.5 1.0 2.5 0.5

Unique values of movie ratings

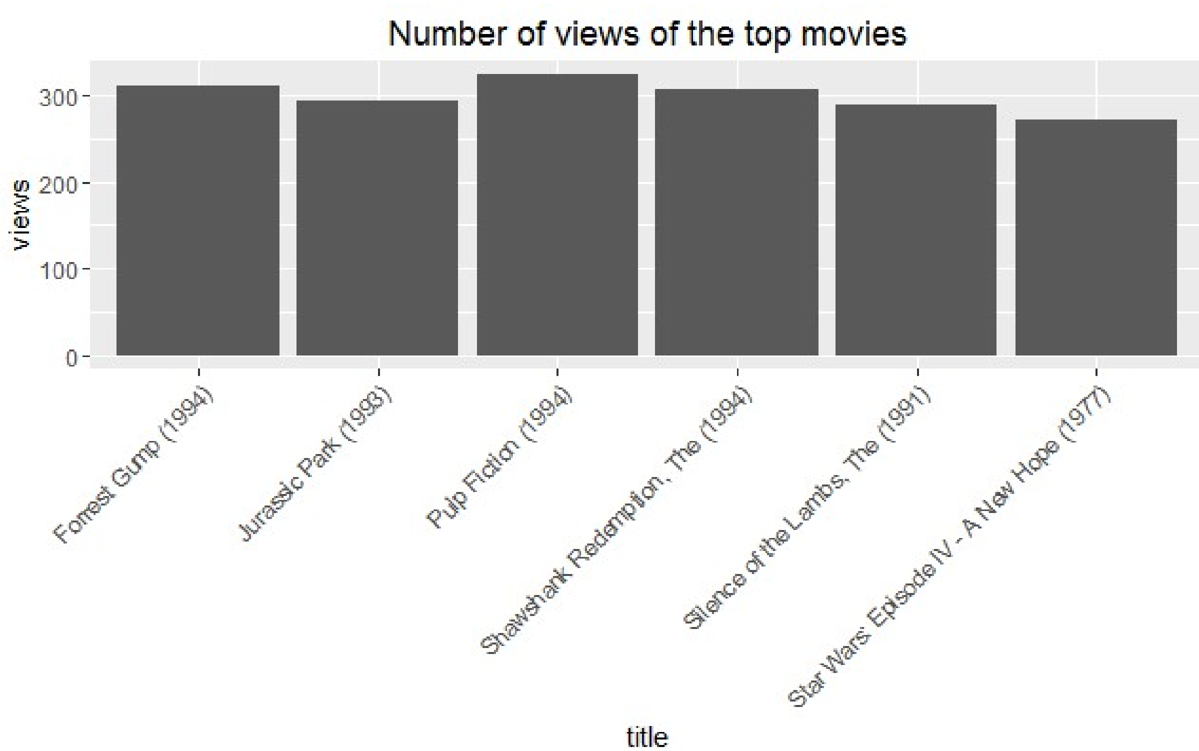
0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
6791761	1198	3258	1567	7943	5484	21729	12237	28880	8187	14856

Count of each ratings value

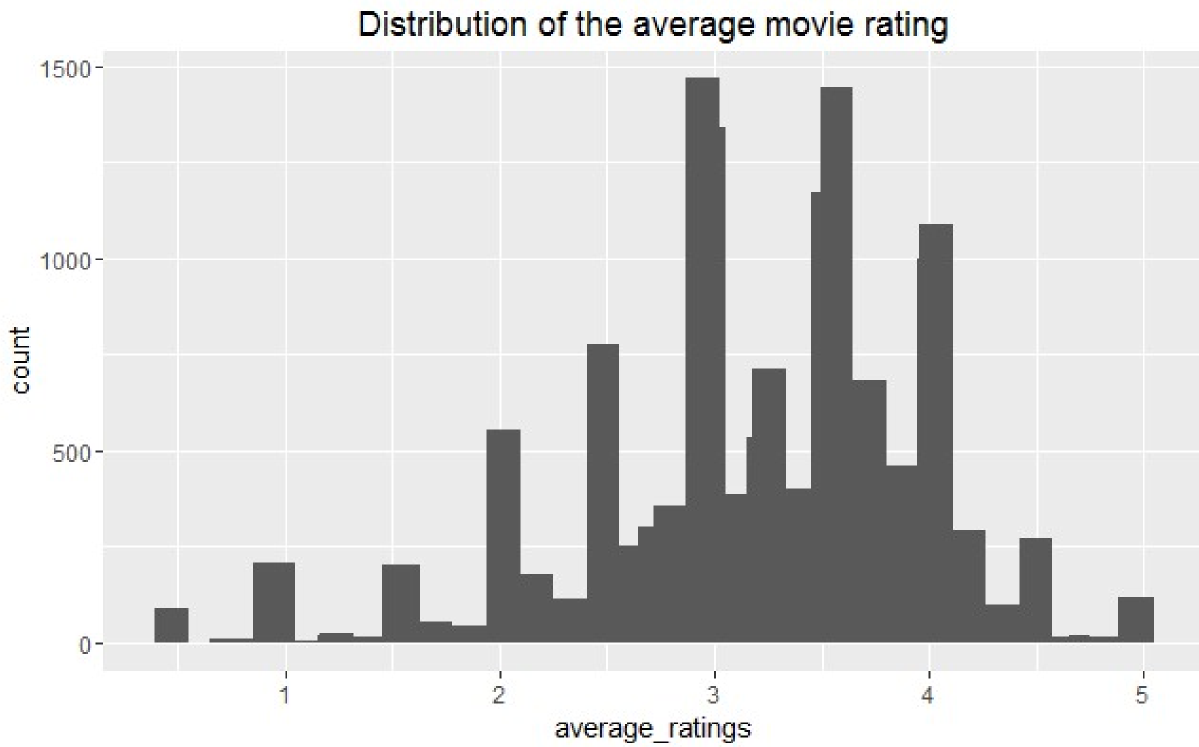
There are 11 unique score values. The lower values mean lower ratings and vice versa. According to the MovieLens dataset documentation, a rating equal to 0 represents a missing value, hence it is removed from the dataset before visualizing the results.



There are less low (less than 3) rating scores, the majority of movies are rated with a score of 3 or higher. The most common rating is 4.

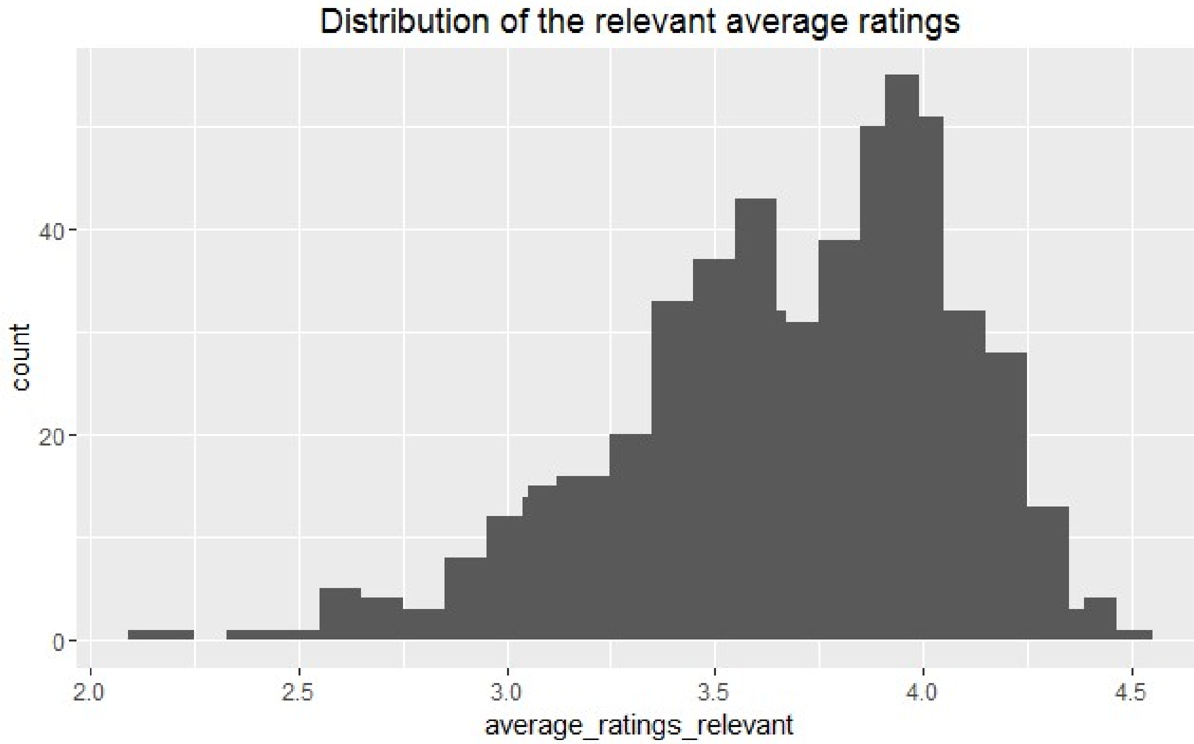


From the plot, it can be seen that "Pulp Fiction (1994)" is the most viewed movie, exceeding the second-most-viewed "Forrest Gump (1994)" by 14 views.



The distribution above shows the distribution of the average movie rating. The highest value is around 3, and 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 shouldn't take them into account.

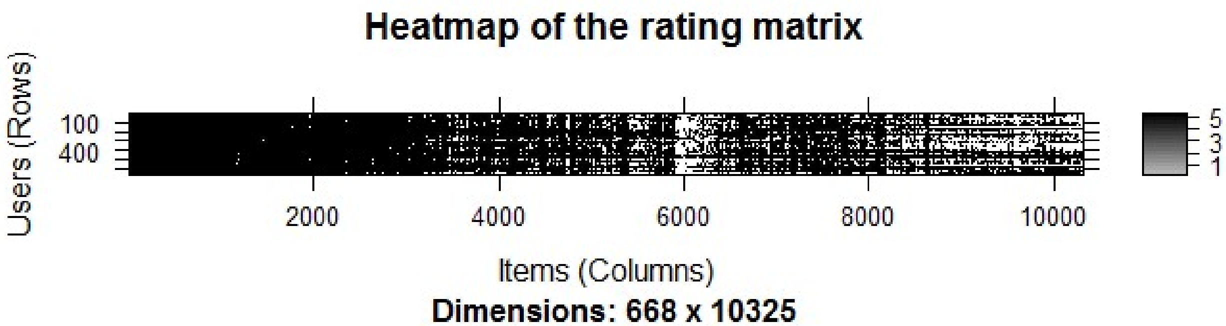
Assigning a threshold value of minimum of 50 views per user, create a subset of only relevant movies.



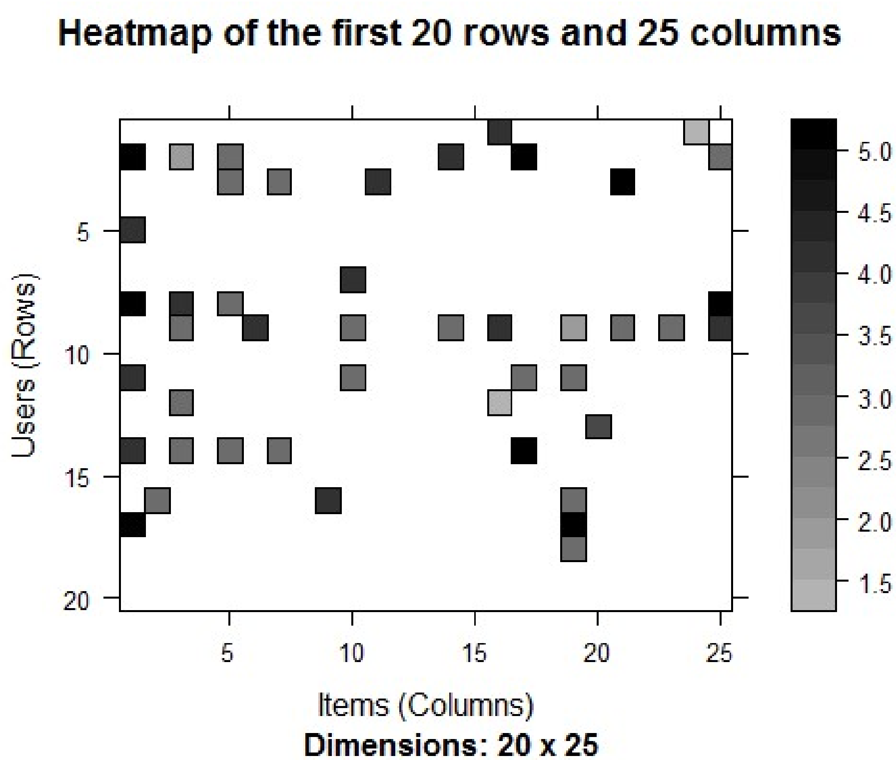
The second image above shows the distribution of the relevant average ratings. All the rankings are between 2.16 and 4.45. As expected, the extremes were removed. The highest value changes, and now it is around 4.

3.1.2 Heatmap of the rating matrix

Visualizing the whole matrix of ratings by building a 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 heatmap chart is hard to read hard as it has too many dimensions. To aid in visualizing, we will plot the heatmap of the first 20 rows and 25 columns.

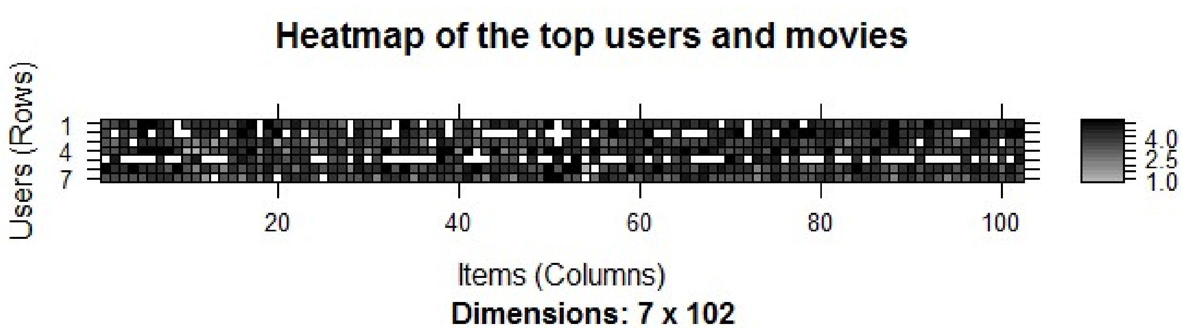


Zooming in on the first rows and columns, it can be observed that the some users saw more movies than the others.

To avoid this user bias, an efficient recommendation algorithm should select the most relevant users (the users who have seen many movies) and movies (the movies that have been seen by many users). To select the most relevant users and movies, the following steps are followed -

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%
1198.17
[1] "Minimum number of users per movie:"
99%
115
```



From above heatmap the following points can be observed -

- Of the users having watched more movies, 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.
- Conversely, darker rows represent users giving higher ratings.

Because of the above factors, it would be a good to normalize the data before building the model.

4 Data Preparation

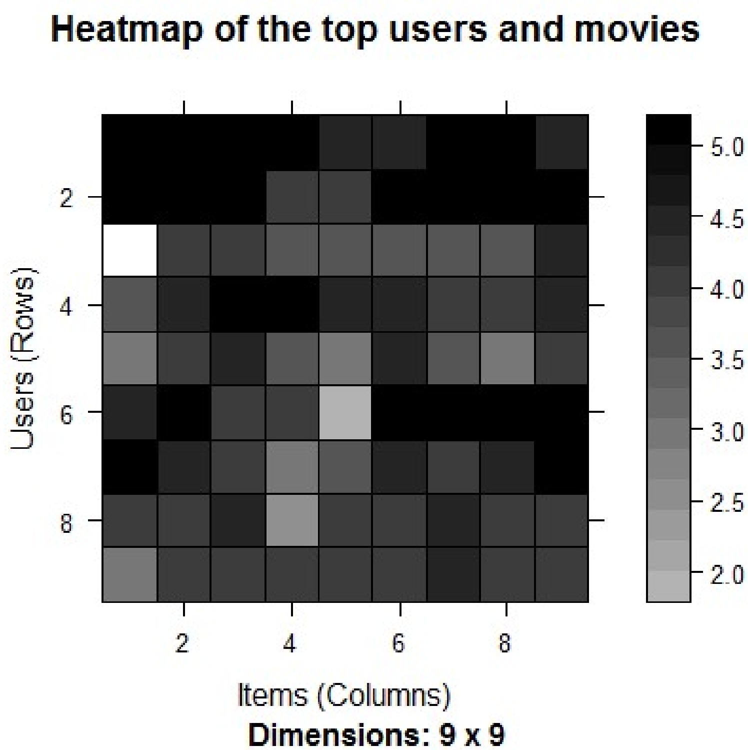
The data preparation process consists of the following steps:

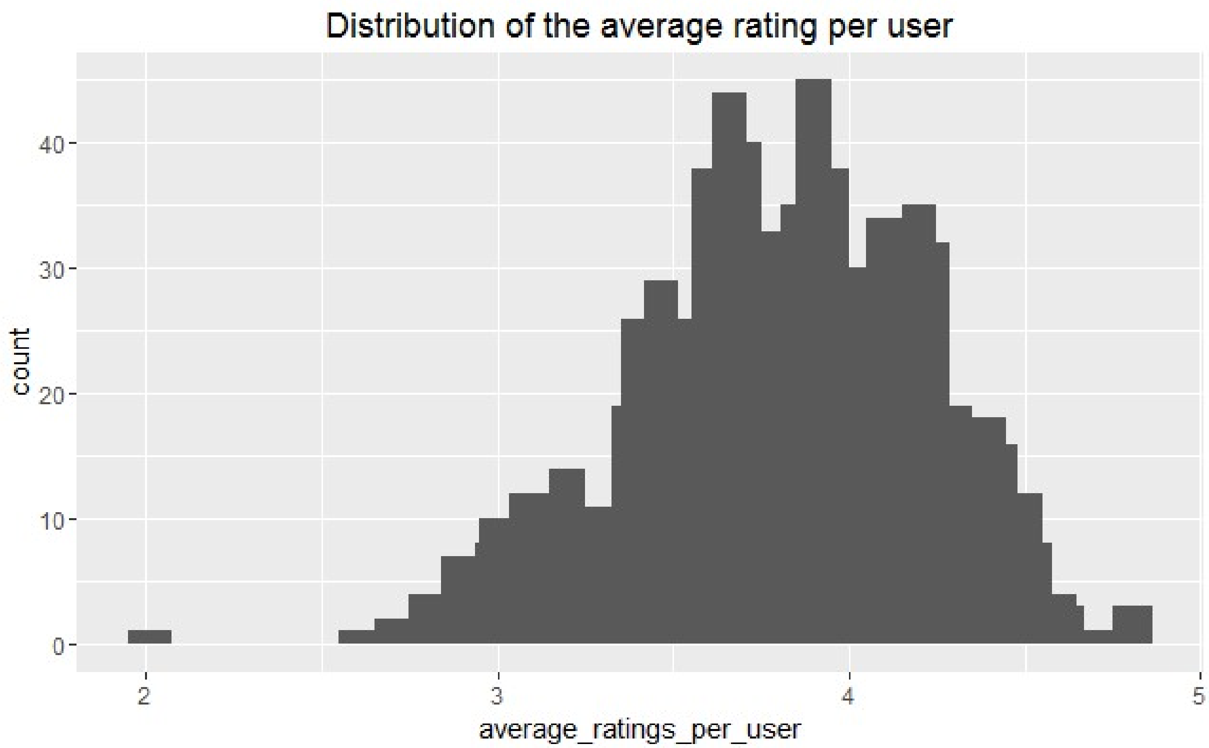
1. Select the relevant data.
2. Normalize the data.
3. Binarize the data.

4.1 Select the relevant data

In order to select the most relevant data, the minimum number of users per rated movie is defined as 50 and the minimum views number per movie as 50. Such a selection of the most relevant data contains 420 users and 447 movies, compared to previous 668 users and 10325 movies in the total dataset.

Using the same approach as previously, visualize the top 2 percent of users and movies in the new matrix of the most relevant data:



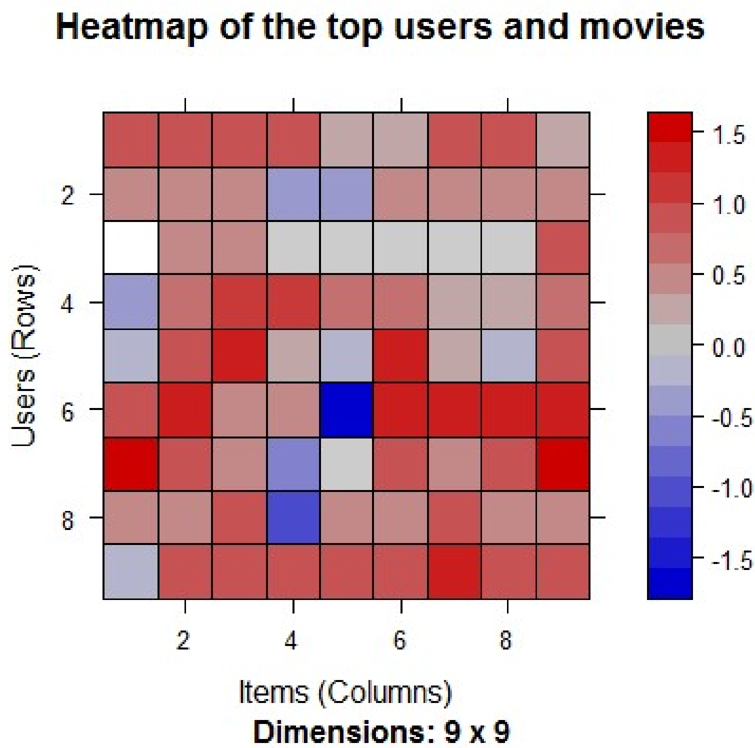


In the heatmap, some rows are darker than the others. This might mean that some users give higher ratings to all the movies. The distribution of the average rating per user across all the users varies a lot, as the second chart above shows.

4.2 NORMALIZING THE DATA

Having users who give high (or low) ratings to all their movies might bias the results. In order to remove this effect, we normalize the data in such a way that the average rating of each user is 0.

Visualizing the normalized matrix for the top movies. It is colored now because the data is continuous:



There are still some lines that seem to be more blue or more red. The reason is that we are visualizing only the top movies. We have already checked that the average rating is 0 for each user, so there is no bias in the user ratings.

4.2 Binarizing the data

Some recommendation models work on binary data, so it might be useful to binarize the data, that is, define a table containing only 0s and 1s. The 0s will be either treated as missing values or as bad ratings.

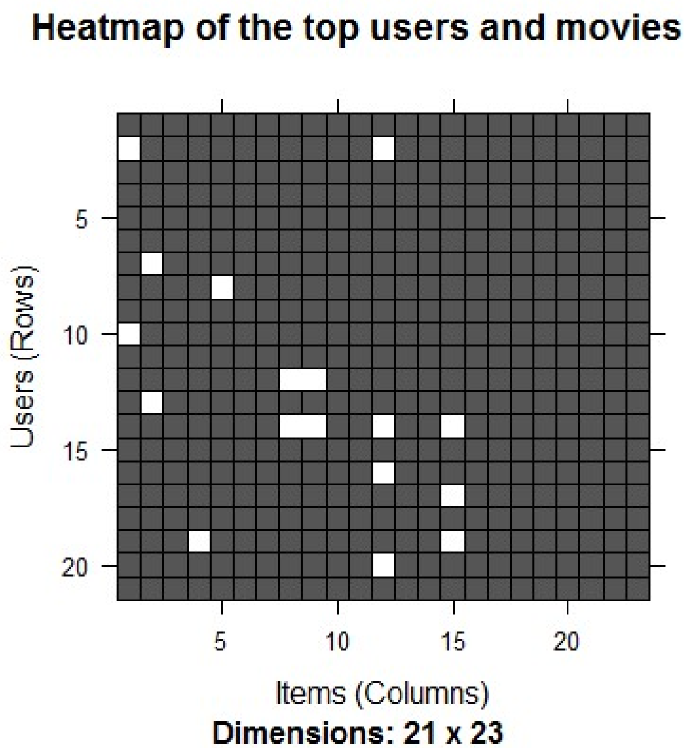
We can either:

- Define a matrix having 1 if the user rated the movie, and 0 otherwise. In this case, the information about the rating is lost.
- Define a matrix having 1 if the rating is above or equal to a definite threshold (for example, 3), and 0 otherwise. In this case, giving a bad rating to a movie is equivalent to not having rated it.

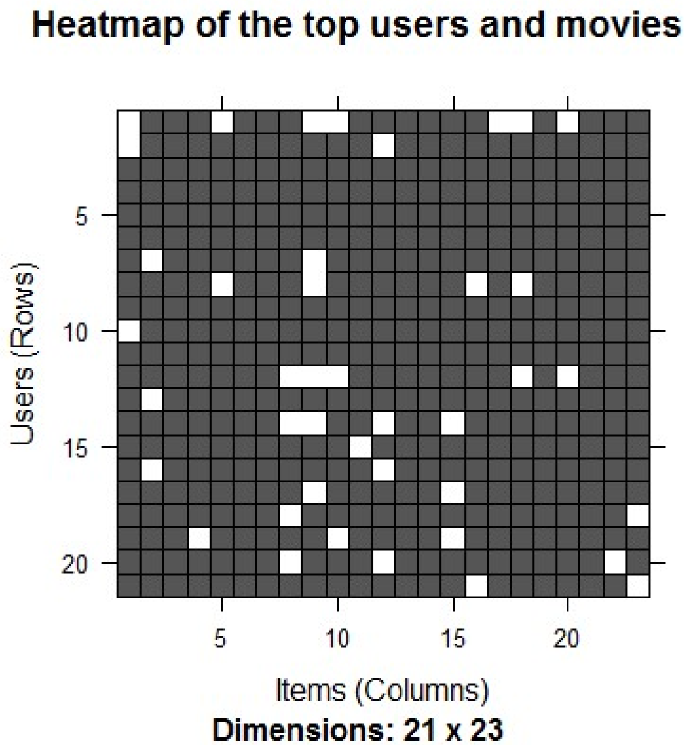
Depending on the context, one choice may be more appropriate than the other.

As a next step, two matrices following the two different approaches are defined. Visualize a 5 percent portion of each of binarized matrices.

1. 1st option - Define a matrix equal to 1 if the movie has been watched



2. 2nd option - Define a matrix equal to 1 if the cell has a rating above the threshold



There are more white cells in the second heatmap, which shows that there are more movies with no or bad ratings than those that were not watched by raters.

5 ITEM-BASED COLLOBORATIVE FILTERING

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.

5.1 BUILDING THE RECOMMENDATION MODEL

We build the model using 80% of the whole dataset as a training set, and 20% - as a test set.

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. After, for each item, the algorithm identifies its k most similar items and stores the number.

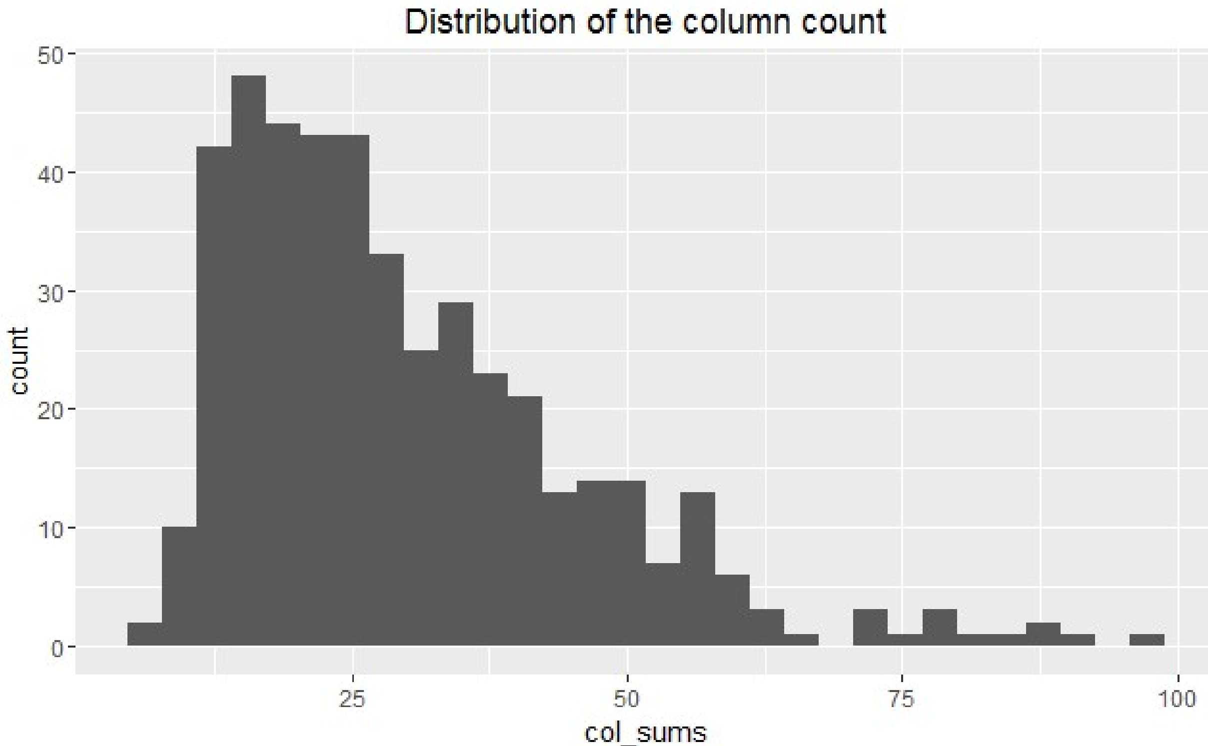
method is a similarity function, which is Cosine by default, may also be Pearson. The recommender model is developed using the default parameters of method = Cosine and k=30.

```
$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 â€˜IBCFâ€™ for â€˜realRatingMatrixâ€™
learned using 319 users.
[1] "Recommender"
attr(,"package")
[1] "recommenderlab"
```

Exploring the recommender model:

```
[1] "dgCMatrix"
attr(,"package")
[1] "Matrix"
[1] 447 447
row_sums
 30
447
```



dgCMatrix is a similarity matrix created by the model. Its dimensions are 447 x 447, 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 symmetric, 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.

5.2 APPLYING RECOMMENDER SYSTEM ON THE MOVIELENS DATASET

Now, it is possible to recommend movies to the users in the test set. I define 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:

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

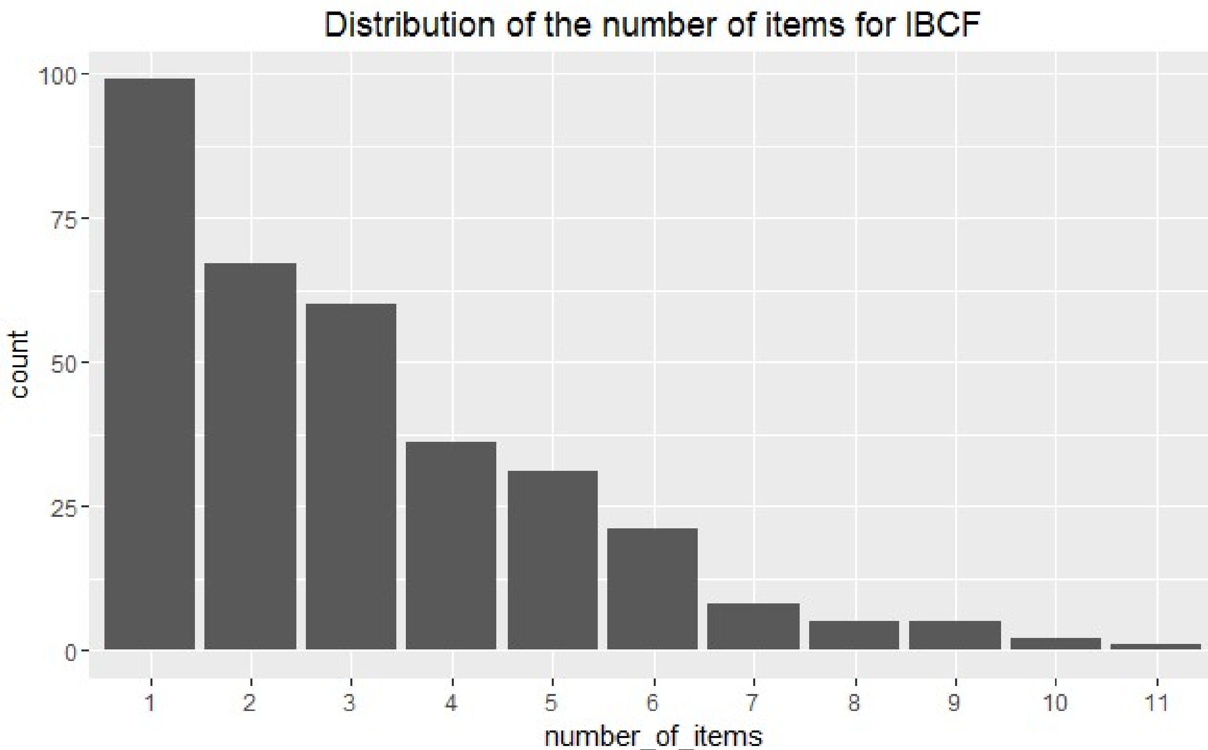
```
[1] "Lawrence of Arabia (1962)"
 [2] "Chinatown (1974)"
 [3] "Shining, The (1980)"
 [4] "Almost Famous (2000)"
 [5] "Pan's Labyrinth (Laberinto del fauno, El) (2006)"
 [6] "Casino Royale (2006)"
 [7] "WALLÃ‚Â·E (2008)"
 [8] "Avatar (2009)"
 [9] "Monty Python and the Holy Grail (1975)"
[10] "Blues Brothers, The (1980)"
```

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

```
     [,1]  [,2] [,3] [,4]
 [1,]  1204  1206    6 2080
 [2,]  1252  2700   16  300
 [3,]  1258  2997   17 2321
 [4,]  3897 48774   21 3897
 [5,] 48394  7438   25 6502
 [6,] 49272  1263   48 1219
 [7,] 60069  6874   70 1230
 [8,] 72998  4034  111 2302
 [9,]  1136   111  141 3081
[10,]  1220  1923  161 6016
```

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

Now, let's identify the most recommended movies. The following image shows the distribution of the number of items for IBCF:



Movie title <chr>	No of items <fctr>
903	Vertigo (1958)
111	Taxi Driver (1976)
923	Citizen Kane (1941)
17	Sense and Sensibility (1995)

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

IBCF 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.

6 USER-BASED COLLOBORATIVE FILTERING

Now, we will use the user-based approach to develop a recommender engine. According to this approach, given a new user, its similar users are first identified. Then, the top-rated items rated by similar users are recommended.

For each new user, these are the steps:

1. Measure how similar each user is to the new one. Like IBCF, popular similarity measures are correlation and cosine.

2. Identify the most similar users. The options are:
   - Take account of the top k users (k-nearest_neighbors)
   - Take account of the users whose similarity is above a defined threshold

3. 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

4. Pick the top-rated movies.

6.1 BUILDING THE RECOMMENDATION SYSTEM

First check the default parameters of UBCF model. Here, nn is a number of similar users, and method is a similarity function, which is cosine by default. We will build a recommender model leaving the parameters to their defaults and using the training set.

```
$method
[1] "cosine"


$nn
[1] 25


$sample
[1] FALSE


$normalize
[1] "center"


Recommender of type â€˜UBCFâ€™ for â€˜realRatingMatrixâ€™
```

```
learned using 319 users.
319 x 447 rating matrix of class â€˜realRatingMatrixâ€™ with 29679 ratings.
Normalized using center on rows.
```

Recommender of type ‘UBCF’ for ‘realRatingMatrix’ learned using 332 users. 332 x 447 rating matrix of class ‘realRatingMatrix’ with 29519 ratings. Normalized using center on rows.

6.2 Building the Recommendation System

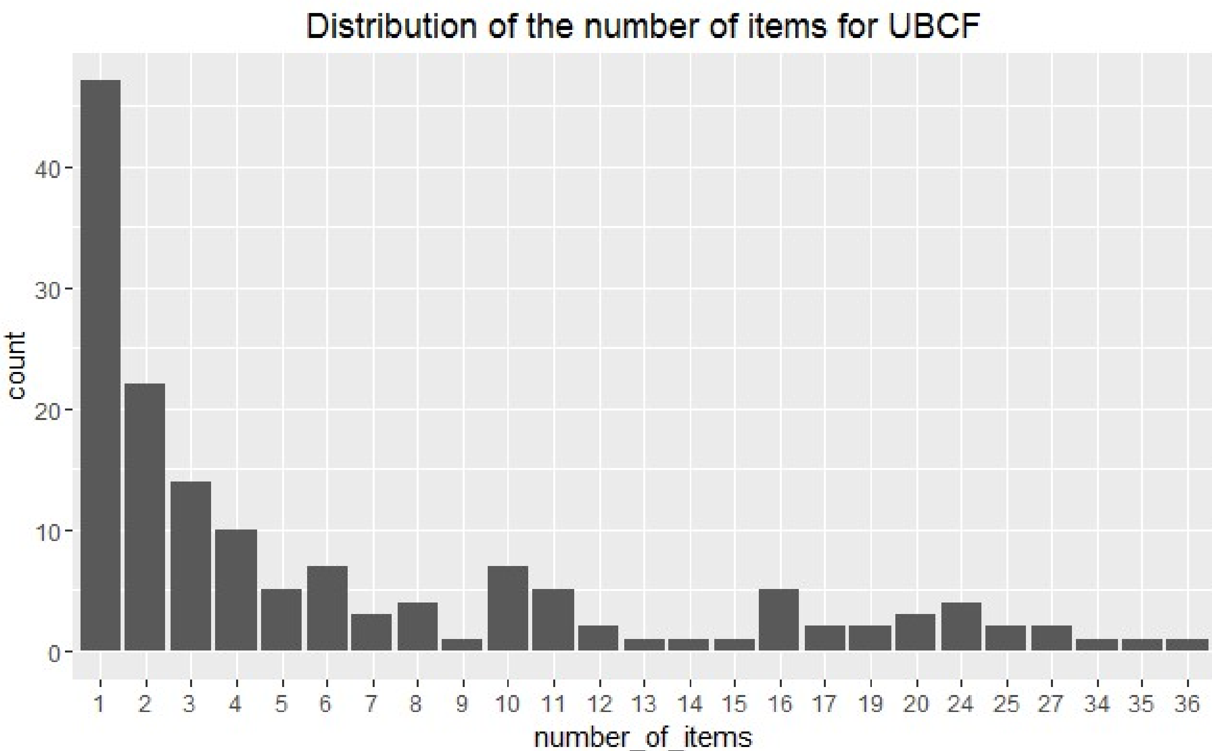
In the same way as the IBCF, we now determine the top ten recommendations for each new user in the test set.

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

```
      [,1] [,2]  [,3] [,4]
 [1,] 1197  608   318 2762
 [2,]  527  541  2571  858
 [3,] 4993 1213  4993 2028
 [4,] 1136 2997 58559 1221
 [5,]  296 1206   110   50
 [6,]  318 1089  1200 1704
 [7,] 1196 1196  7153  908
 [8,] 5618 1200  5952 1183
 [9,] 2858 2858  2858  923
[10,] 1079  858   527 1641
```

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

We now compute how many times each movie got recommended and build the related frequency histogram:



Compared with the IBCF, 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 30, compared to 10-ish for IBCF.

Movie title <chr>	No of items <fctr>
527	Schindler's List (1993)
50	Usual Suspects, The (1995)
318	Shawshank Redemption, The (1994)
32	Twelve Monkeys (a.k.a. 12 Monkeys) (1995)

4 rows

Comparing the results of UBCF with IBCF helps find some useful insight on different algorithms. UBCF 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, UBCF's accuracy is proven to be slightly more accurate than IBCF (refer to next section), so it's a good option if the dataset is not too big.

7 EVALUATING THE RECOMMENDER SYSTEMS

There are a few options to choose from when deciding to create a recommendation engine. In order to compare their performances and choose the most appropriate model, we follow these steps:

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

7.1 PREPARING THE DATA TO EVALUATE MODELS

We need two training and testing data to evaluate the model. There are several methods to create them: 1) splitting the data into training and test sets, 2) bootstrapping, 3) using k-fold.

7.1.1 Splitting the data

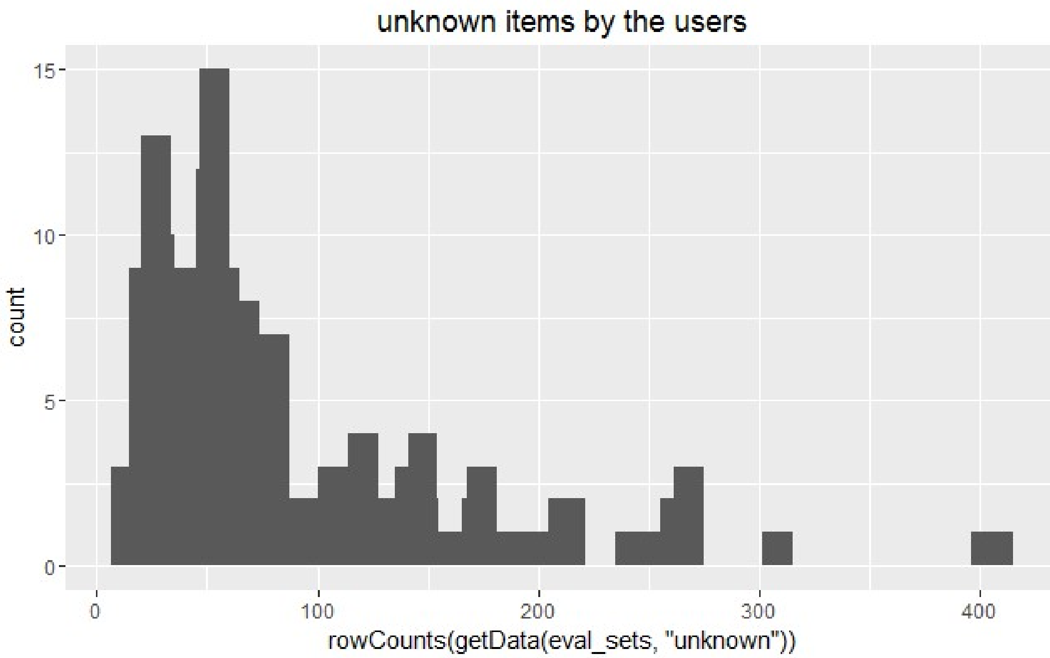
Splitting the data into training and test sets is often done using a 80/20 proportion.

For each user in the test set, we need to define how many items to use to generate recommendations. For this, first check the minimum number of items rated by users to be sure there will be no users with no items to test.

```
Evaluation scheme with 5 items given

Method: â€˜splitâ€™ with 1 run(s).

Training set proportion: 0.800

Good ratings: >=3.000000

Data set: 420 x 447 rating matrix of class â€˜realRatingMatrixâ€™ with 38341 ratings.
```

```
336 x 447 rating matrix of class â€˜realRatingMatrixâ€™ with 30253 ratings.
```

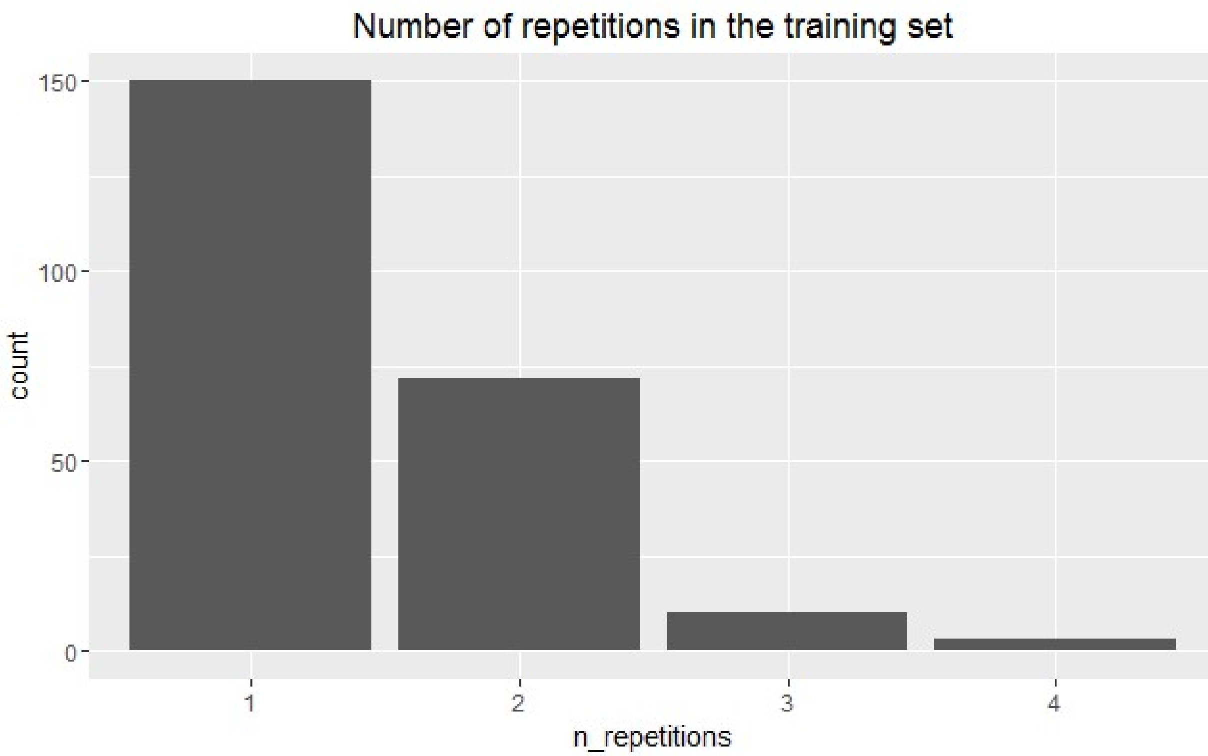
Training Data



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

7.1.2 Bootstrapping the data

Bootstrapping is another approach to split the data. 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.



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

7.1.3 Using cross-validation to validate models

The k-fold cross-validation approach is the most accurate one, although it's computationally heavier.

Using this approach, we split the data into some chunks, take a chunk out as the test set, and evaluate the accuracy. Then, we can do the same with each other chunk and compute the average accuracy.

```
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
```

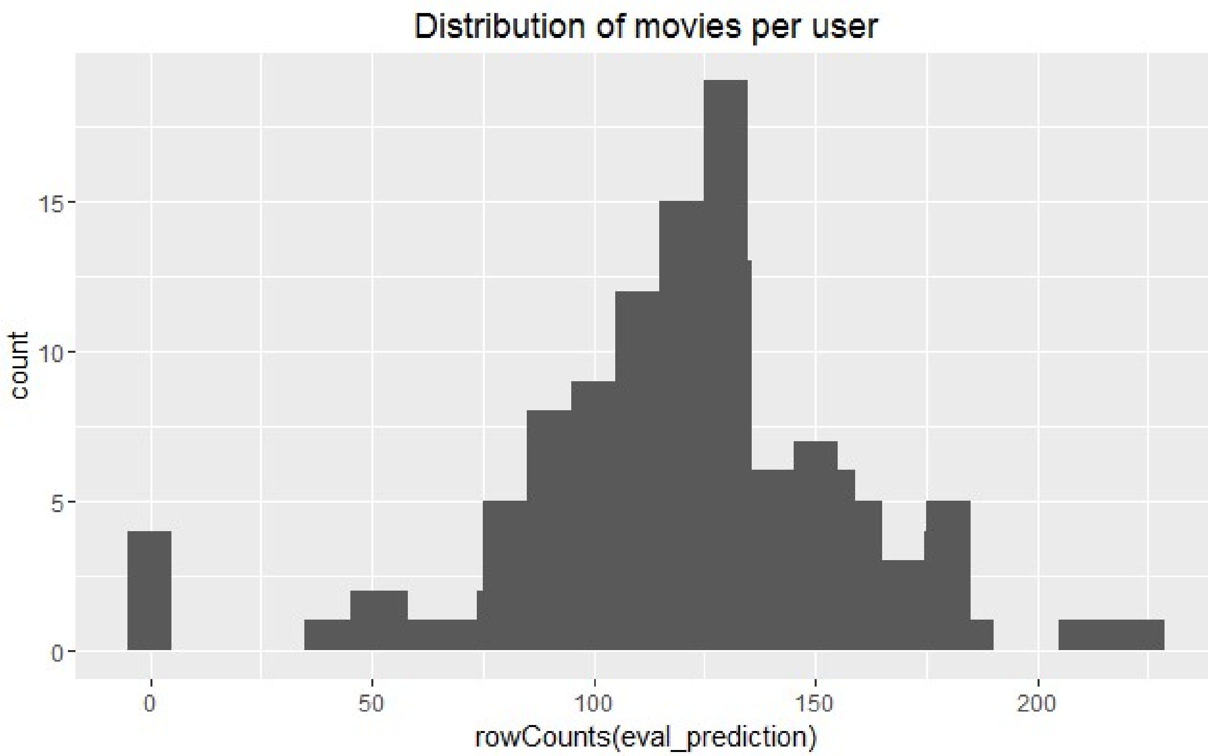
Output –

```
[1] 315 315 315 315
```

Using 4-fold approach, we get four sets of the same size 315.

7.2 EVALUATING THE RATINGS

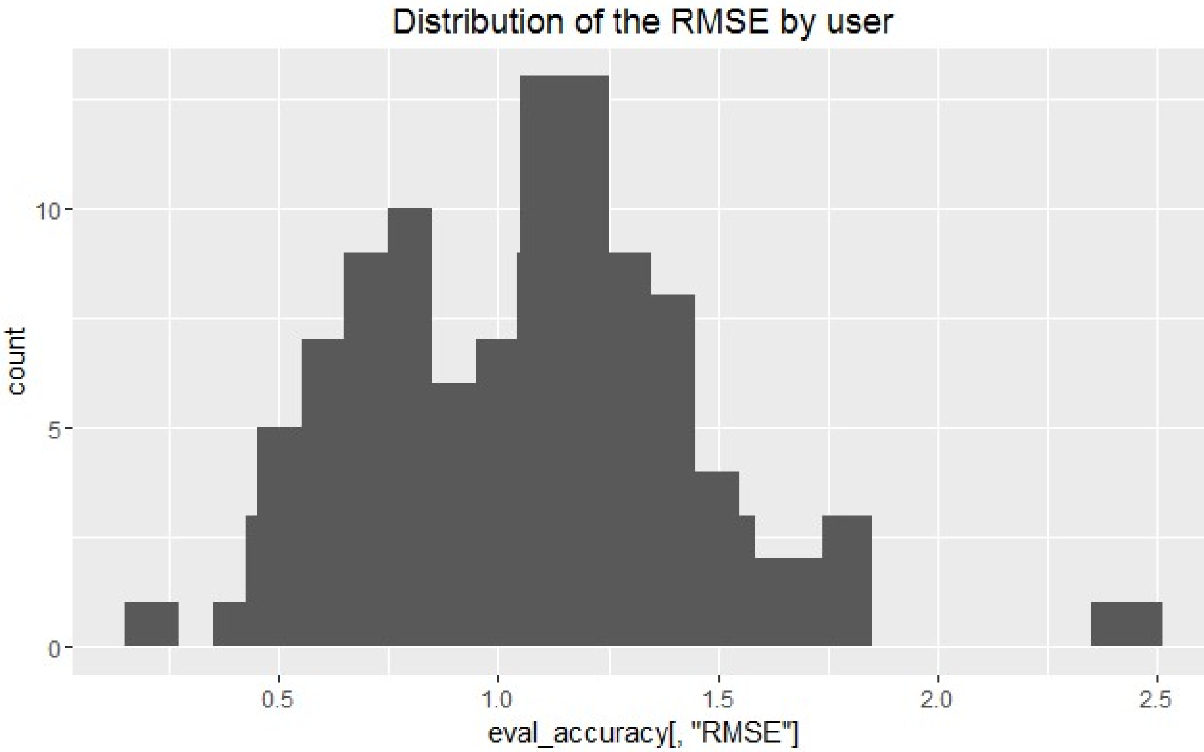
We use the k-fold approach for evaluation. First, re-define the evaluation sets, build IBCF model and create a matrix with predicted ratings.



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

Now, compute 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,] 1.3296234 1.7678984 1.0127311
[2,] 1.5890031 2.5249309 1.3137298
[3,] 1.1654473 1.3582674 0.9069369
[4,] 0.5973857 0.3568697 0.4326998
[5,] 1.6743058 2.8032999 1.3754862
[6,] 0.7380156 0.5446670 0.5595776
```



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

```
RMSE       MSE       MAE
1.1222614 1.2594706 0.8070389
```

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

7.3 EVALUATING THE RECOMMENDATIONS

Another way to measure accuracies is by comparing the recommendations with the purchases having a positive rating. For this, we can make use of a prebuilt evaluate function in recommenderlab library. The function evaluate the recommender performance depending on the number n of items to recommend to each user. Use n as a sequence n = seq(10, 100, 10).

The first rows of the resulting performance matrix is presented below:

```
IBCF run fold/sample [model time/prediction time]
     1  [1.25sec/0.13sec]
     2  [1.57sec/0.16sec]
     3  [1.55sec/0.09sec]
     4  [1.54sec/0.14sec]
          TP        FP        FN        TN precision     recall        TPR
FPR
10  2.609524  7.009524 72.09524 360.2857 0.2712871 0.03969245 0.03969245
0.01895376
20  5.057143 14.180952 69.64762 353.1143 0.2628713 0.07716881 0.07716881
0.03831023
30  7.438095 21.419048 67.26667 345.8762 0.2577558 0.11753797 0.11753797
0.05782728
40  9.704762 28.761905 65.00000 338.5333 0.2522912 0.15644577 0.15644577
0.07770426
50 11.809524 36.152381 62.89524 331.1429 0.2461284 0.19068919 0.19068919
0.09791065
60 14.047619 43.276190 60.65714 324.0190 0.2445266 0.22418937 0.22418937
0.11729812
```

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

```
TP         FP        FN        TN
10 10.80000  28.15238 291.4857 1437.562
20 21.19048  56.71429 281.0952 1409.000
30 31.33333  85.52381 270.9524 1380.190
40 40.44762 115.35238 261.8381 1350.362
50 49.20000 145.28571 253.0857 1320.429
```

```
60 8.87619 173.96190 243.4095 1291.752
```

7.4 COMPARING THE MODELS

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

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  [1.53sec/0.13sec]
     2  [1.48sec/0.14sec]
     3  [1.39sec/0.14sec]
     4  [1.36sec/0.12sec]
IBCF run fold/sample [model time/prediction time]
     1  [1.46sec/0.11sec]
     2  [1.46sec/0.15sec]
     3  [1.45sec/0.13sec]
     4  [1.39sec/0.13sec]
UBCF run fold/sample [model time/prediction time]
     1  [0.02sec/0.81sec]
     2  [0.03sec/0.81sec]
     3  [0.02sec/0.83sec]
     4  [0.02sec/0.83sec]
UBCF run fold/sample [model time/prediction time]
     1  [0sec/0.77sec]
     2  [0.03sec/0.77sec]
     3  [0.01sec/0.72sec]
     4  [0.02sec/0.79sec]
RANDOM run fold/sample [model time/prediction time]
     1  [0.04sec/0.28sec]
```

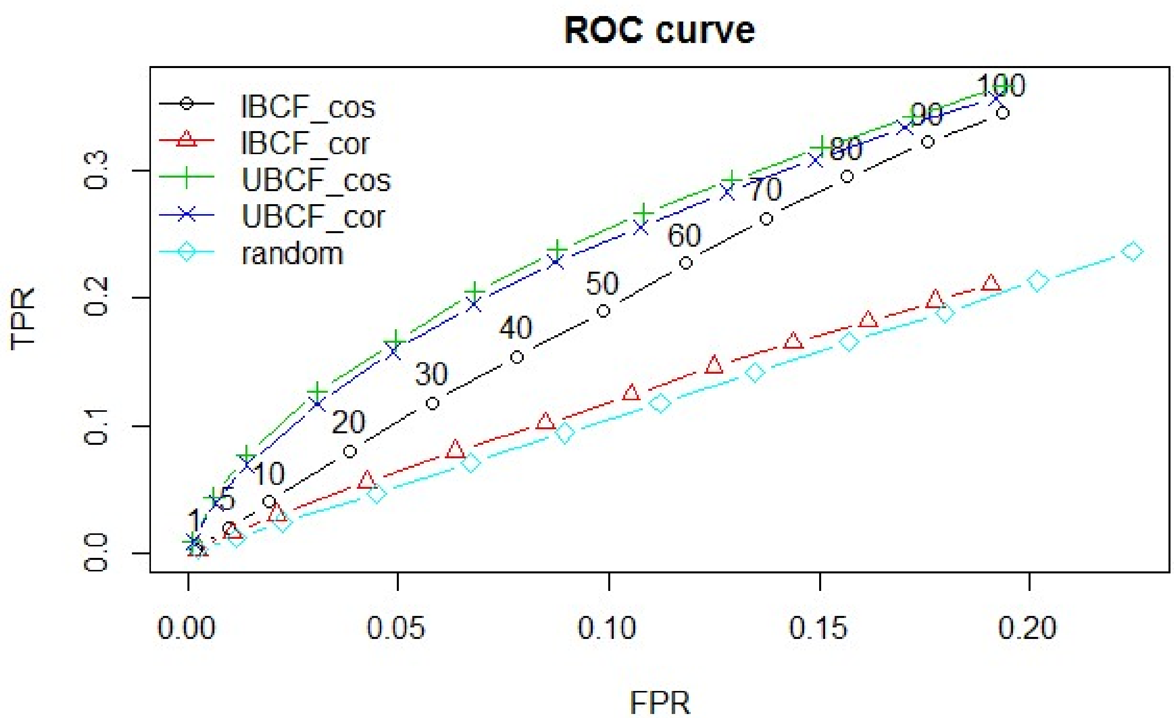
```
     2  [0.01sec/0.38sec]
     3  [0sec/0.3sec]
     4  [0sec/0.34sec]
IBCF_cos IBCF_cor UBCF_cos UBCF_cor   random
    TRUE     TRUE     TRUE     TRUE     TRUE
```

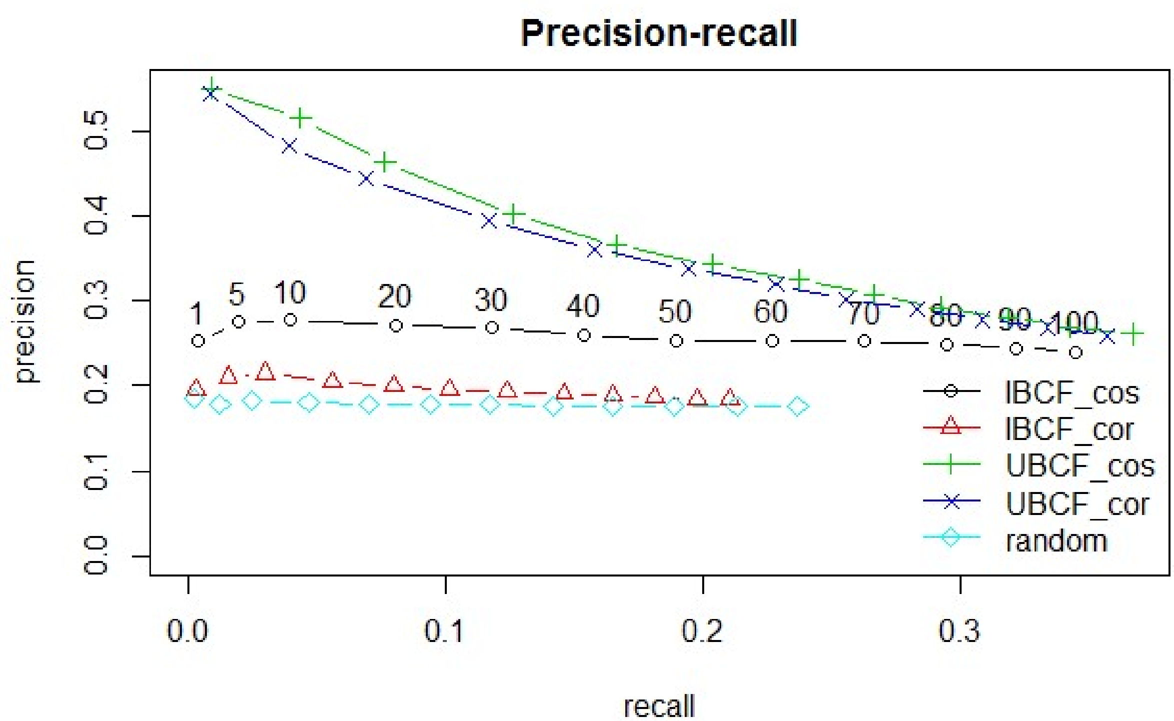
The following table presents as an example the first rows of the performance evaluation matrix for the IBCF with Cosine distance:

```
precision      recall          TPR          FPR
1  0.2517850 0.003481586 0.003481586 0.001999699
5  0.2743549 0.019848578 0.019848578 0.009649068
10 0.2772911 0.039960250 0.039960250 0.019123124
20 0.2719631 0.080425975 0.080425975 0.038447395
30 0.2680793 0.118143097 0.118143097 0.057908428
40 0.2595672 0.153864467 0.153864467 0.078282294
```

7.5 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 UBCF with cosine distance, so it's the best-performing technique.

The UBCF with cosine 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.

7.6 OPTIMIZING A NUMERIC PARAMETER

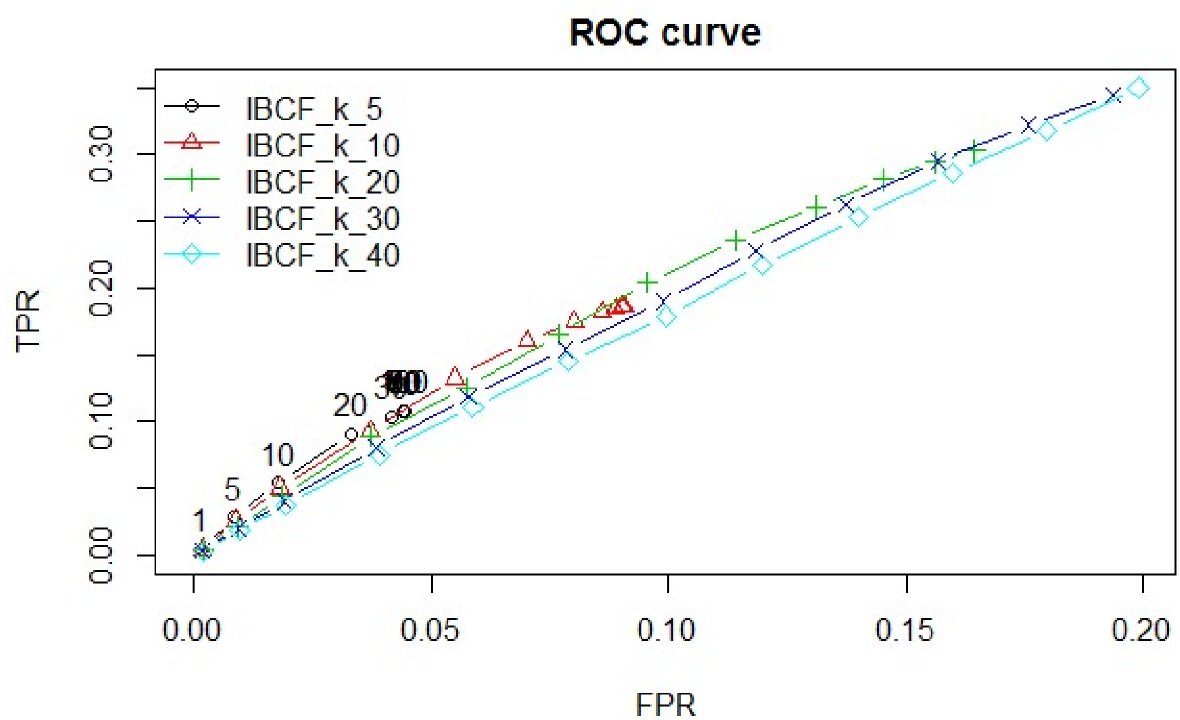
IBCF takes account of the k-closest items.we will explore more values, ranging between 5 and 40, in order to tune this parameter:

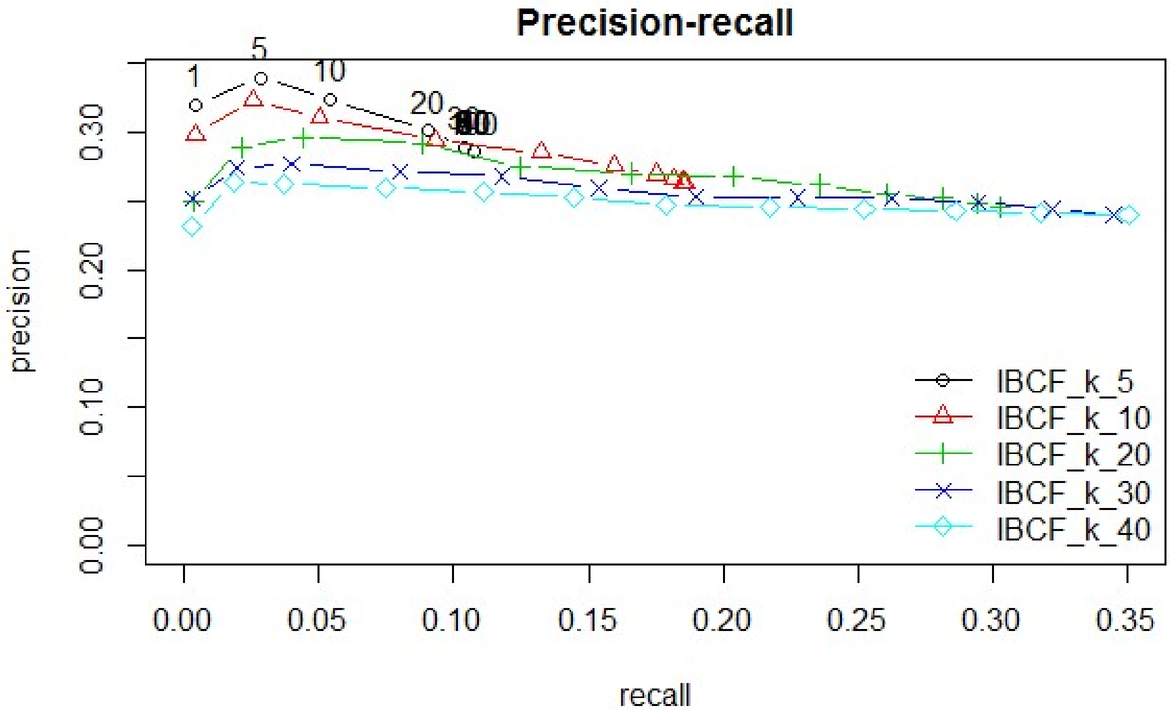
```
vector_k <- c(5, 10, 20, 30, 40)
models_to_evaluate <- lapply(vector_k, function(k){
  list(name = "IBCF",
       param = list(method = "cosine", k = k))
})
names(models_to_evaluate) <- paste0("IBCF_k_", vector_k)
```

Now let us 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  [1.48sec/0.08sec]
     2  [1.37sec/0.1sec]
     3  [1.36sec/0.07sec]
     4  [1.31sec/0.08sec]
IBCF run fold/sample [model time/prediction time]
     1  [1.28sec/0.09sec]
     2  [1.34sec/0.09sec]
     3  [1.38sec/0.07sec]
     4  [1.3sec/0.06sec]
IBCF run fold/sample [model time/prediction time]
     1  [1.31sec/0.11sec]
     2  [1.35sec/0.12sec]
     3  [1.47sec/0.12sec]
```

```
     4  [1.35sec/0.11sec]
IBCF run fold/sample [model time/prediction time]
     1  [1.37sec/0.12sec]
     2  [1.37sec/0.14sec]
     3  [1.37sec/0.14sec]
     4  [1.37sec/0.13sec]
IBCF run fold/sample [model time/prediction time]
     1  [1.34sec/0.16sec]
     2  [1.38sec/0.16sec]
     3  [1.33sec/0.14sec]
     4  [1.33sec/0.16sec]
```





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 IBCF 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.

8 POTENTIAL NEXT STEPS

There are many different directions further work on this could go, following 3 main paths:

1) **Hybrid Recommender Systems –** This project analyzed two popular recommendation algorithms – User-based collaborative filtering and Item-based collaborative filtering.

   A major problem with the item based approach is its accuracy and narrow focus. The recommendations may not be very interesting or unique. Many of the recommendations are already known to the user.

   A major problem with the collaborative filtering is that it suffers from cold start problems. Many users who are just starting out won't receive accurate recommendations or any recommendations at all – until enough data is gathered from the community of users.

   By combining the item-based and user-based collaborative filtering, a hybrid recommender system can overcome each ones' shortcoming. It can start by using the item-based approach to avoid the cold-start problem. Once enough data is collected from the community of users, the system can use the collaborative filtering approach to produce more interesting and personalized recommendations.

2) **Improving precision with constant feedback –** One way to improve the precision of the systems' recommendations is to ask for customer feedback. Collecting customer feedback can be done in many different ways, through multiple channels. For example. We can build a back end code in NodeJS or Python that implements our recommendation engine on the server side. When the customer browses the movie set, the recommendation engine can give its recommendations to the user and ask for user feedback on the recommendations. Also it can ask for the viewers to rate the movies, they have already seen. The recommendation engines can use this data to make more precise recommendations.

3) **Uplifting the model –** Uplift modeling, also called true lift modeling and net modeling among other terms, aims to fine tune recommender models that target only the viewers who can be persuaded to watch the movie. This approach can be used to develop a focused target marketing model that can maximize the revenues by personalized contacts and promotions to the viewer.

9 CONCLUSIONS AND SUMMARY

In this project, we have developed and evaluated a collaborative filtering recommender (CFR) system for recommending movies. The online app was created to demonstrate the User-based Collaborative Filtering approach for recommendation model.

Let's discuss the strengths and weaknesses of the User-based Collaborative Filtering approach in general.

Strengths: User-based Collaborative Filtering gives recommendations that can be complements to the item the user was interacting with. This might be a stronger recommendation than what a item-based recommender can provide as users might not be looking for direct substitutes to a movie they had just viewed or previously watched.

Weaknesses: User-based Collaborative Filtering is a type of Memory-based Collaborative Filtering that uses all user data in the database to create recommendations. Comparing the pairwise correlation of every user in your dataset is not scalable. If there were millions of users, this computation would be very time consuming. Possible ways to get around this would be to implement some form of dimensionality reduction, such as Principal Component Analysis, or to use a model-based algorithm instead. Also, user-based collaborative filtering relies on past user choices to make future recommendations. The implications of this is that it assumes that a user's taste and preference remains more or less constant over time, which might not be true and makes it difficult to pre-compute user similarities offline.

10 REFERENCES

Bibliography references -

1. Book Chapter - Difference Spatial and Temporal Correlation.pdf
2. Research paper - MovieGEN: A Movie Recommendation System by Eyrun A. Eyjolfsdottir, Gaurangi Tilak, Nan Li
3. Research paper - Application of Dimensionality Reduction in Recommender System -- A Case Study Badrul M. Sarwar, George Karypis, Joseph A. Konstan, John T. Riedl
4. Book Chapter - Evaluating Recommendation Systems.pdf
5. Research paper - Evaluating Recommender Systems : An evaluation framework to predict user satisfaction for recommender systems in an electronic programme guide context by Joost de Wit
6. Book - Mining of Massive Datasets by Jure Leskovec, Anand Rajaraman, Jeffrey D. Ullman Stanford Univ
7. Book - Building a Recommendation System with R by Suresh K Gorakala, Michele Usuelli
8. Research Paper - Recommender System Using CollaborativeFiltering Algorithm by Ala Alluhaidan
9. Recommender Systems Handbook by Francesco Ricci, Lior Rokach, Bracha Shapira and Paul B. Kantor
10. Research paper - recommenderlab: A Framework for Developing and Testing Recommendation Algorithms by Michael Hahsler

Website References –

1. http://www.dataperspective.info/2014/05/basic-recommendation-engine-using-r.html
2. https://www.codementor.io/spark/tutorial/building-a-recommender-with-apache-spark-python-example-app-part1
3. https://www.linkedin.com/pulse/create-recommendation-engine-using-r-simple-steps-minta-thomas
4. https://www.youtube.com/watch?v=RljxwNGgiqU
5. https://www.analyticsvidhya.com/blog/2016/03/exploring-building-banks-recommendation-system/
6. https://www.analyticsvidhya.com/blog/2016/10/creating-interactive-data-visualization-using-shiny-app-in-r-with-examples/
7. https://ashokharnal.wordpress.com/2014/12/18/using-recommenderlab-for-predicting-ratings-for-movielens-data/
8. http://rstudio-pubs-static.s3.amazonaws.com/150913_3ccebcc146d84e5e98c40fc548322929.html
9. https://muffynomster.wordpress.com/2015/06/07/building-a-movie-recommendation-engine-with-r/

10. http://datamining-r.blogspot.com.au/2014/09/movie-recommender-system-in-r.html

Programming References –

1. https://www.r-bloggers.com/
2. http://www.dataperspective.info/
3. https://www.yhat.com/products/rodeo