HarvardX Capstone Course - MovieLens Project - Recommender

Miguel Revilla

02/08/2021

Contents

1 Introduction

1.1 Project goals and key steps

The goal of this project is the creation of a movie recommendation system using the MovieLens dataset, which contains millions of ratings. Information about this data, as well as the download files can be found at MovieLens. This site offers movie rating datasets of different sizes:

- 25M dataset (MovieLens 25 million ratings applied to 62,000 movies by 162,000 users)
- MovieLens Latest Datasets: small (100,000 ratings applied to 9,000 movies by 600 users) and full (27,000,000 ratings applied to 58,000 movies by 280,000 users)
- synthetic datasets
- MovieLens 100K Dataset: 100K movie ratings from 1000 users on 1700 movies
- MovieLens 1M Dataset: 1 million ratings from 6000 users on 4000 movies
- MovieLens 10M Dataset: 10 million ratings applied to 10,000 movies by 72,000 users
- MovieLens 20M Dataset: 20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users.

We have used the MovieLens dataset version included in the dslabs package for some of the exercises in the "PH125.8x: Data Science: Machine Learning Course", which is just a small subset of the Movielens dataset that included ratings applied by 671 users to 9066 movies.

For the purpose of our recommendation system in this project, we will be using the 10M version of this MovieLens dataset, which can be found at: MovieLens 10M Dataset

We have seen in the course the approach of creating a very simple (naive) recommendation model and keep building on it by adding the effect of different predictors. We will first follow that approach and build a complex model taking into account the effect of different predictors (Movie, User, Release Year, Genres and Rate Year) with regularization on them. But we will not stop there. Instead, we will follow the suggestion given at the end of section 33.11.2 (Connection to SVD and PCA) about using the recommenderlab package and exploring different recommendation systems.

Recommendation systems are usually classified in two different categories, content-based approaches and collaborative filtering (CF):

- Content-based approaches are based on the idea of obtaining the preferences of a user related to item properties and make recommendations of items ranking high in those user preferences. These preferences can be found by analyzing current user preferences (movies seen, items purchased, etc.).
- Collaborative filtering (CF) is to discover the similarities on the user's past behavior and make predictions to the user based on a similar preference with other users. This model is then used to predict items (or ratings for items) that the user may have an interest in. In our case, the idea is that given rating data by many users for many items (e.g., 1 to 5 stars for movies rated directly by the users), one can predict a user's rating for an item not known to the user. This is based on the premise that users who agreed on the rating for some items typically also agree on the rating for other items. Collaborative filtering (CF) algorithms are typically divided into two groups, memory-based (aka neighborhood based) algorithms and model-based algorithms (aka latent factor models):
  - Memory-based use the whole (or at least a large sample of the) user database to create recommendations by finding similarities between products or users. The disadvantages of this approach is scalability, since the whole user database has to be processed online for creating recommendations. Let's mention the two most known:
    * User-based: it is the most prominent memory-based algorithm. This method works under the assumption that users with similar item tastes will rate items similarly. Therefore, the missing ratings for a user can be predicted by finding other similar users (a neighborhood).

Within the neighborhood, we can aggregate the ratings of these neighbors on items unknown to the user as the basis for a prediction.
    * Item-based algorithm which: instead of finding the most similar users to each individual, it assesses the similarities between the items that are correlated in their ratings or purchase profile among all users.
  - Model-based algorithms: they 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. One common example of these is the SVD (Singular Value Decomposition), which is a recommender based on SVD approximation with column-mean imputation, or the SVDF (Singular Value Decomposition Funk), which is recommender based on Funk SVD with gradient descend. Or the Recosystem, which uses the LIBMF library for matrix factorization. We will evaluate the three of them in this document.

We will apply to our MovieLens dataset the following CF recommendation systems in order to evaluate which one performs the best for a final test against our validation data:

- Using the recommenderlab package:
  - Memory-based algorithms:
    * Popular-based Collaborative Filtering
    * User-based Collaborative Filtering (UBCF)
    * Item-based Collaborative Filtering (IBCF)
  - Model-based algorithms:
    * SVD Collaborative Filtering
    * SVDF Collaborative Filtering
- Recosystem (matrix factorization)

When using the recommenderlab package for the different models outlined above, we will use the split, cross-validation and bootstrap methods when creating the evaluation schema. These methods are used by the different algorithms to create their train and validation datasets.

Most of the CF models used in this project that were included in the recommenderlab package had problems managing the edx train set from the 10M Movielens DB. They were run on both Windows 10 with 16GB of memory and iOS with 8GB of memory and execution did not complete. Because of this, I did generate a smaller train set (edx_cf) from the edx one of just 10% of it. If edx train set has 9000065 rows, the new train set (edx_cf) was down to around 900K rows I had already applied the Movielens 1M dataset on them and knew that, even if some of them taking a long time to run, they completed their execution in most of the cases. This smaller set (edx_cf) obtained from the edx one was used then to evaluate the CF algorithms and compare their RMSE to choose the best performing one across all of them and apply it on the full edx and validation datasets.

To be mentioned the recommenderlab package, although it provides basic recommendation algorithms, is not a recommendation system but rather an infrastructure for developing and test recommender algorithms and 0-1 data in a unified framework.

NOTE: all the source code included in this RMD file is not executed when knitting to generate the PDF output of it and the plot images are all read from files. The reason for it is that full run time of the code in it would take at least 5 hours. But all the source code, including plots, was executed within the script file delivered with the .RMD and .PDF files.

1.2 Data download

The following code was used to download the MovieLens data and generate both training and test datasets and was provided by the HarvardX PH125.9x Data Science Capstone course

```r
##########################################################
# Create edx set, validation set (final hold-out test set)
##########################################################

# Note: this process could take a couple of minutes

if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")

library(tidyverse)
library(caret)
library(data.table)

# MovieLens 10M dataset:
# https://grouplens.org/datasets/movielens/10m/
# http://files.grouplens.org/datasets/movielens/ml-10m.zip

dl <- tempfile()
download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

ratings <- fread(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")

# if using R 3.6 or earlier:
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))
# if using R 4.0 or later:
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres))


movielens <- left_join(ratings, movies, by = "movieId")

# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]

# Make sure userId and movieId in validation set are also in edx set
validation <- temp %>%
      semi_join(edx, by = "movieId") %>%
      semi_join(edx, by = "userId")

# Add rows removed from validation set back into edx set
removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)
```

```
rm(dl, ratings, movies, test_index, temp, movielens, removed)
```

With the data partition, we end up with a train set (edx) with 90% of the observations and a validation set with the remaining 10% after applying the createDataPartition() function on the movielens dataset:

```
nrow(edx)
```

```
[1] 9000065
```

```
nrow(validation)
```

```
[1] 999999
```

A first look at the data

```
head(edx, 6)
```

```
   userId movieId rating timestamp                          title                        genres
1:      1     122      5 838985046               Boomerang (1992)                Comedy|Romance
2:      1     185      5 838983525                Net, The (1995)          Action|Crime|Thriller
3:      1     292      5 838983421                Outbreak (1995)  Action|Drama|Sci-Fi|Thriller
4:      1     316      5 838983392                Stargate (1994)       Action|Adventure|Sci-Fi
5:      1     329      5 838983392 Star Trek: Generations (1994) Action|Adventure|Drama|Sci-Fi
6:      1     355      5 838984474       Flintstones, The (1994)       Children|Comedy|Fantasy
```

1.3 Data exploration

Let's download some libraries that will be used later.

```
if(!require(dplyr)) install.packages("dplyr", repos = "http://cran.us.r-project.org")
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
if(!require(ggplot2)) install.packages("ggplot2", repos = "http://cran.us.r-project.org")
library(dplyr)
library(lubridate)
library(ggplot2)
```

The data shown in the data download section gives us a few ideas to improve the use of it in our recommendation system:

- The "timestamp" column is the date and time when the user did the rating. It would be useful to extract the year when the rating was done in order to use it as a feature/predictor in combination with the release year. The rest of the timestamp (month, day and time) could be used for a further analysis, but we will skip it in this work.
- The "title" column contains the release year of the movie. It will be useful to extract it and have it separate to use as an additional feature.
- The "genres" column contains one or more different genres applicable to a particular movie. To make better use of this information, it will be separated in individual genres to facilitate the grouping of movie ratings for further analysis.

1.3.1 Extract the year when user did the review

The following code will take data in the timestamp column, convert into a Date & Time format using the as_datetime() function in the lubridate package and then extract the year from it and put it into a new column (rate_year). We do it for both edx and validation sets.

```
edx <- mutate(edx, rate_year = year(as_datetime(timestamp)))
validation <- mutate(validation, rate_year = year(as_datetime(timestamp)))
#edx_train <- mutate(edx_train, rate_year = year(as_datetime(timestamp)))
#edx_test <- mutate(edx_test, rate_year = year(as_datetime(timestamp)))
```

Let's check the years obtained are consistent and there is no weird entry (e.g.: before 1900 or beyond current year of 2020). The outcome shows the range of years users did their ratings is consistent and were done between 1995 and 2009. No need to verify for the test set, as all moviIds in it are presen in the train set.

```
range(edx$rate_year)
```

```
[1] 1995 2009
```

1.3.2 Extract the movie release year

The following code will parse each string in the "title" column and extract from it the YYYY placed within the parenthesis, convert it to a number and put it into a new column, "release_year".

```
edx <- edx %>% mutate(release_year = as.numeric(str_sub(title,-5,-2)))
validation <- validation %>% mutate(release_year =  as.numeric(str_sub(title,-5,-2)))
#edx_train <- edx_train %>% mutate(release_year = as.numeric(str_sub(title,-5,-2)))
#edx_test <- edx_test %>% mutate(release_year =  as.numeric(str_sub(title,-5,-2)))
```

Let's check the years obtained are consistent and there is no weird entry (e.g.: before 1900 or beyond current year of 2020). The outcome shows the range of movie release year is consistent and were released between 1915 and 2008. No need to verify for the test set, as all moviIds in it are presen in the train set.

```
range(edx$release_year)
```

```
[1] 1915 2008
```

Looking now at our train set (edx) we see the data we have extracted.

```
head(edx)
```

```
   userId movieId rating timestamp                         title                         genres
1:      1     122      5 838985046              Boomerang (1992)                 Comedy|Romance
2:      1     185      5 838983525               Net, The (1995)          Action|Crime|Thriller
3:      1     292      5 838983421               Outbreak (1995)  Action|Drama|Sci-Fi|Thriller
4:      1     316      5 838983392               Stargate (1994)       Action|Adventure|Sci-Fi
5:      1     329      5 838983392 Star Trek: Generations (1994) Action|Adventure|Drama|Sci-Fi
6:      1     355      5 838984474       Flintstones, The (1994)       Children|Comedy|Fantasy
   rate_year release_year
1:      1996         1992
```

```
2:      1996         1995
3:      1996         1995
4:      1996         1994
5:      1996         1994
6:      1996         1994
```

1.3.3 Separate data from genres column into individual entries

In order to do this, we will be creating new additional train and test sets, as there will be quite an increase in the number of final rows. And for future calculations when genres data is not involved, it is worth to use the shorter datasets as the execution time of our calculations will be much shorter.

```
edx_split_genre  <- edx %>% separate_rows(genres, sep = "\\|")
validation_split_genre <- validation %>% separate_rows(genres, sep = "\\|")
```

The number of rows in the train set has gone from about 9M rows to more than 23M, about 2.5 times more.

```
nrow(edx_split_genre)
```

```
[1] 23371423
```

Similar thing happens for the test set, going from about 1M rows to 2.6M, 2.6 times more

```
nrow(validation_split_genre)
```

```
[1] 2595771
```

Having a look a the train set, we can see the genres column with individual entries:

```
head(edx_split_genre)
```

```
# A tibble: 6 x 8
  userId movieId rating timestamp title            genres   rate_year release_year
   <int>   <dbl>  <dbl>     <int> <chr>            <chr>        <int>        <dbl>
1      1     122      5 838985046 Boomerang (1992) Comedy        1996         1992
2      1     122      5 838985046 Boomerang (1992) Romance       1996         1992
3      1     185      5 838983525 Net, The (1995)  Action        1996         1995
4      1     185      5 838983525 Net, The (1995)  Crime         1996         1995
5      1     185      5 838983525 Net, The (1995)  Thriller      1996         1995
6      1     292      5 838983421 Outbreak (1995)  Action        1996         1995
```

1.3.4 Create train and test sets from edx dataset

We will split the edx data set into train and test sets and keep the validation set obtained previously for final prediction and evaluation of the best of the models.

```
set.seed(2021, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
test_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1, list = FALSE)
edx_train <- edx[-test_index,]
temp <- edx[test_index,]
```

Make sure userId and movieId in test set are also in train set

```
edx_test <- temp %>%
  semi_join(edx_train, by = "movieId") %>%
  semi_join(edx_train, by = "userId")
```

Add rows removed from test set back into train set

```
removed <- anti_join(temp, edx_test)
edx_train <- rbind(edx_train, removed)
nrow(edx)
```

```
[1] 9000055
```

```
nrow(edx_train)
```

```
[1] 8100069
```

```
nrow(edx_test)
```

```
[1] 899986
```

Separate genres column into individual values

```
edx_train_split_genre  <- edx_train %>% separate_rows(genres, sep = "\\|")
edx_test_split_genre <- edx_test %>% separate_rows(genres, sep = "\\|")
```

2 Methods & Analysis

2.1 Data analysis

A quick look at our edx dataset:

```
glimpse(edx)
```

```
Rows: 9,000,055
Columns: 8
$ userId       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2,...
$ movieId      <dbl> 122, 185, 292, 316, 329, 355, 356, 362, 364, 370, 377, 420, 466, 520, 539, 58...
$ rating       <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 3, 5, 2, 3,...
$ timestamp    <int> 838985046, 838983525, 838983421, 838983392, 838983392, 838984474, 838983653, ...
$ title        <chr> "Boomerang (1992)", "Net, The (1995)", "Outbreak (1995)", "Stargate (1994)", ...
$ genres       <chr> "Comedy|Romance", "Action|Crime|Thriller", "Action|Drama|Sci-Fi|Thriller", "A...
$ rate_year    <dbl> 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996,...
$ release_year <dbl> 1992, 1995, 1995, 1994, 1994, 1994, 1994, 1994, 1994, 1994, 1994, 1994, 1993,...
```

Let's see some measurements for each column, obviously not character type columns.

```
summary(edx)
```

```
    userId          movieId          rating        timestamp            title          
 Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:9000055  
 1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08   Class :character  
 Median :35738   Median : 1834   Median :4.000   Median :1.035e+09   Mode  :character  
 Mean   :35870   Mean   : 4122   Mean   :3.512   Mean   :1.033e+09                     
 3rd Qu.:53607   3rd Qu.: 3626   3rd Qu.:4.000   3rd Qu.:1.127e+09                     
 Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09                     
    genres            rate_year     release_year  
 Length:9000055     Min.   :1995   Min.   :1915  
 Class :character   1st Qu.:2000   1st Qu.:1987  
 Mode  :character   Median :2002   Median :1994  
                    Mean   :2002   Mean   :1990  
                    3rd Qu.:2005   3rd Qu.:1998  
                    Max.   :2009   Max.   :2008  
```

Let's look at how many different users, movies, genres, rate years and release years we have in our data:

```
edx %>% summarize(n_users = n_distinct(userId), n_movies = n_distinct(movieId), n_ratings = n_distinct(
```

```
  n_users n_movies n_ratings n_genres n_rate_years n_release_years
1   69878    10677        10      797           15              94
```

```
edx_split_genre %>% summarize(n_users = n_distinct(userId), n_movies = n_distinct(movieId), n_ratings =
```

```
# A tibble: 1 x 6
  n_users n_movies n_ratings n_genres n_rate_years n_release_years
    <int>    <int>     <int>    <int>        <int>           <int>
1   69878    10677        10       20           15              94
```

To be noticed that splitting the genres column into individual values we have reduced the numer of different genres from 797 to 20.

Looking at the different ratings users have given to movies, we can see they are 10 different values ranging from 0.5 to 10 and can be given by seq(0.5, 5, 0.5). This means that our rating outcome is of categorical type with 10 different classes.

```
unique(edx$rating) %>% sort()
```

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

Also, we have seen in the Quiz within this Capstone course that:

- The five most given ratings in order from most to least are: 4, 3, 5, 3.5, 2
- In general, half star ratings are less common than whole star ratings

We will confirm that in a later plot. Let's have a look at the movies with highest number of user ratings. As seen by the titles, they are all blockbusters watched by a big audience.

```
edx %>% group_by(movieId, title) %>%
  summarize(n = n()) %>%
  arrange(desc(n)) %>%
  head()
```

```
# A tibble: 6 x 3
# Groups:   movieId [6]
  movieId title                                  n
    <dbl> <chr>                              <int>
1     296 Pulp Fiction (1994)                31362
2     356 Forrest Gump (1994)                31079
3     593 Silence of the Lambs, The (1991)   30382
4     480 Jurassic Park (1993)               29360
5     318 Shawshank Redemption, The (1994)   28015
6     110 Braveheart (1995)                  26212
```

On the opposite side, if we check for the movies with the lowest number of user ratings which we can see by the title they all seem to be artsy movies seeing by not many people. In our case, the movies have just a single rating.

```
edx %>% group_by(movieId, title) %>%
  summarize(n = n(), genres = genres) %>%
  arrange(n) %>%
  head()
```

```
# A tibble: 6 x 4
# Groups:   movieId, title [6]
  movieId title                                  n genres
    <dbl> <chr>                              <int> <chr>
1    3191 Quarry, The (1998)                    1 Drama
2    3226 Hellhounds on My Trail (1999)         1 Documentary
3    3234 Train Ride to Hollywood (1978)        1 Comedy
4    3356 Condo Painting (2000)                 1 Documentary
5    3383 Big Fella (1937)                      1 Drama|Musical
6    3561 Stacy's Knights (1982)                1 Drama
```

2.1.1 Ratings Distribution

Let's plot the distribution of ratings without taking into consideration any of the predictors

```
edx %>% ggplot(aes(rating)) +
  geom_histogram(bins = 20) +
  ggtitle("Rating Distribution")
```

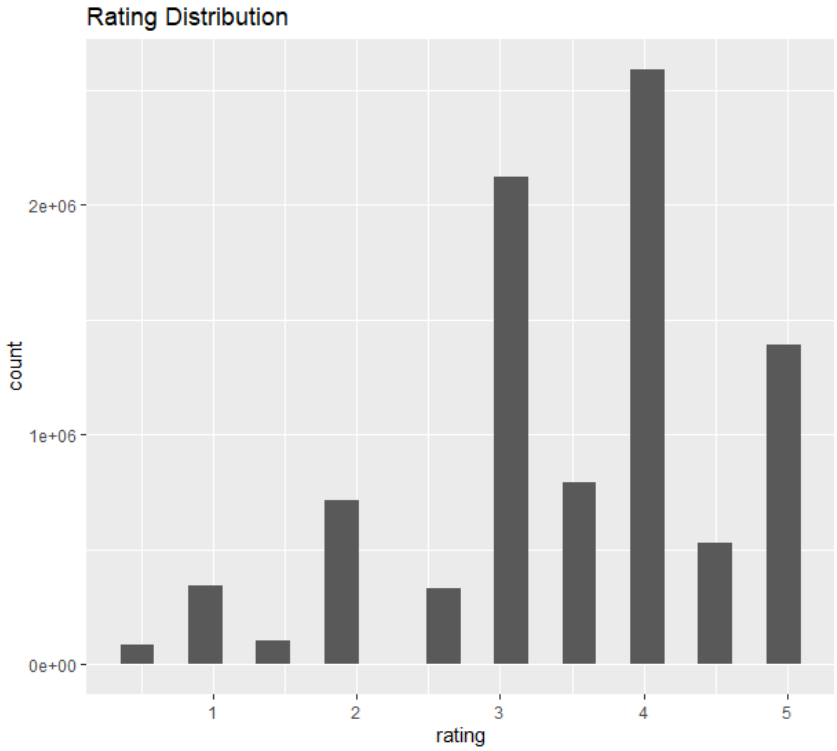

Figure 1: Ratings Distribution

We can confirm from the plot above that round ratings are the ones having most entries as indicated earlier. In particular, the 3.0 with more than 2M entries and the 4.0 with more than 2.5M entries.

2.1.2 Distribution of movies versus their average rating

Let's examine the distribution of movies based on their rating average. We are only considering those movies with more than 100 ratings. The overal rating across all movies already seen in previous section with the summary function is 3.512 (mean) and a median value of 4.000.

```
edx %>%
  group_by(movieId) %>%
  summarize(average_rating = mean(rating)) %>%
  filter(n()>=100) %>%
  ggplot(aes(average_rating)) +
  geom_histogram(bins = 30, color = "black") +
  xlab("Movie average rating") +
```

```
  ylab("Number of movies") +
  ggtitle("Distribution of movies versus their average rating")
```

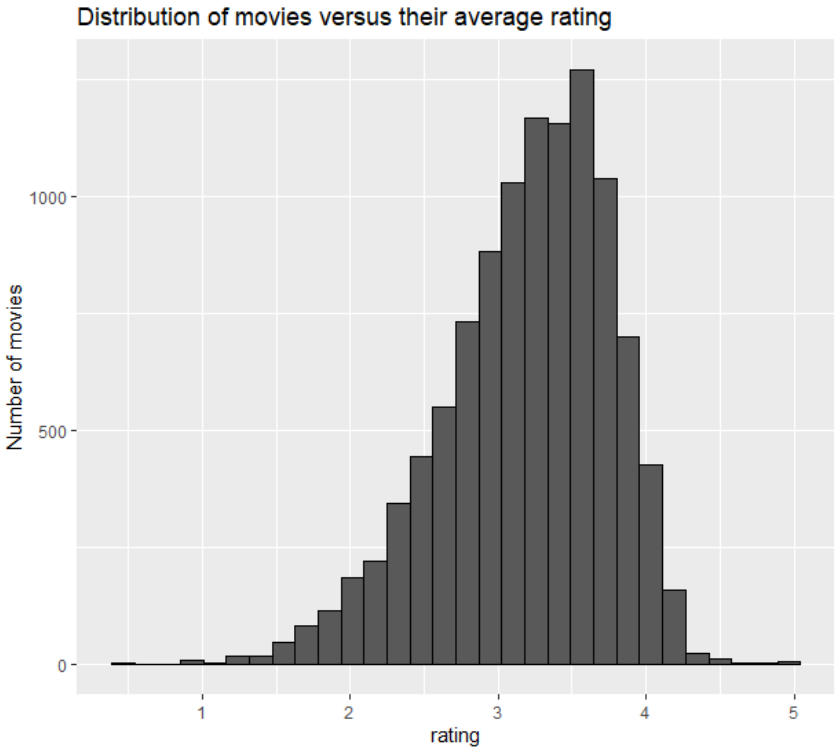

Figure 2: Movies versus average rating

Another way to see movies versus their rating average, using all points and geom_point() function rather than a histogram, would be the following plot:

```
edx %>%
  group_by(movieId) %>%
  summarize(average_rating = mean(rating)) %>%
  ggplot(aes(movieId, average_rating)) +
  geom_point(alpha = .50, color = "black") +
  geom_smooth() +
  ggtitle("Distribution of movies versus their average rating")
```

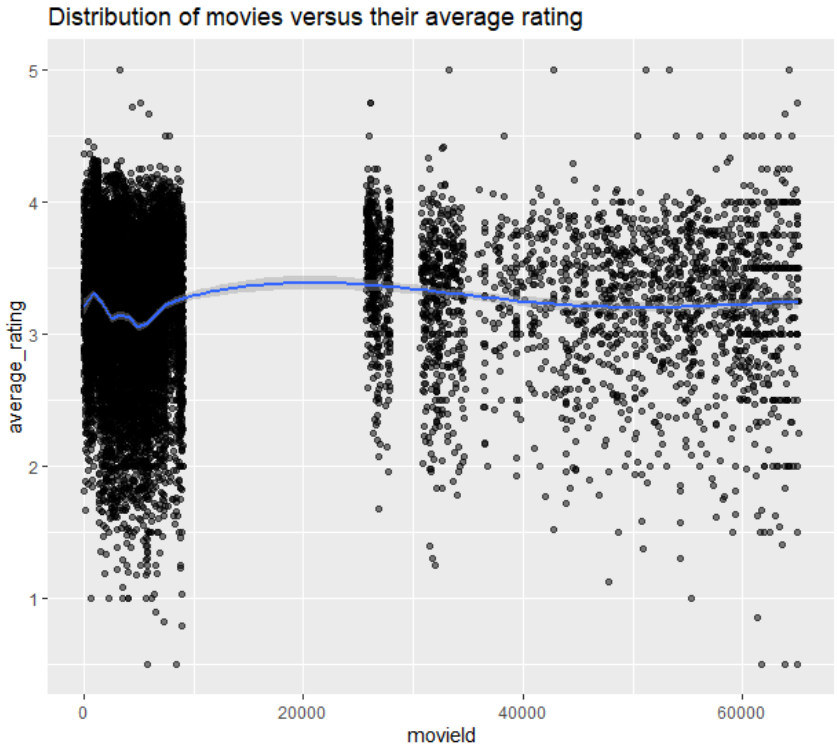

Figure 3: All movies versus their average rating.

To be noticed there are some ranges of movieIds for which the 10M MovieLens dataset does not have any entry. The geom_smooth() function shows us almost a horizontal line, with a bit of wiggle at the beginning and below the global average of 3.512.

2.1.3 Distribution of movies versus their number of ratings

A different way to examine the movies is plotting them against their number of ratings.

```
edx %>%
  group_by(movieId) %>%
  summarize(n = n()) %>%
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  xlab("number of ratings") +
  ylab("number of movies") +
  ggtitle("Distribution of Movies versus number of ratings")
```

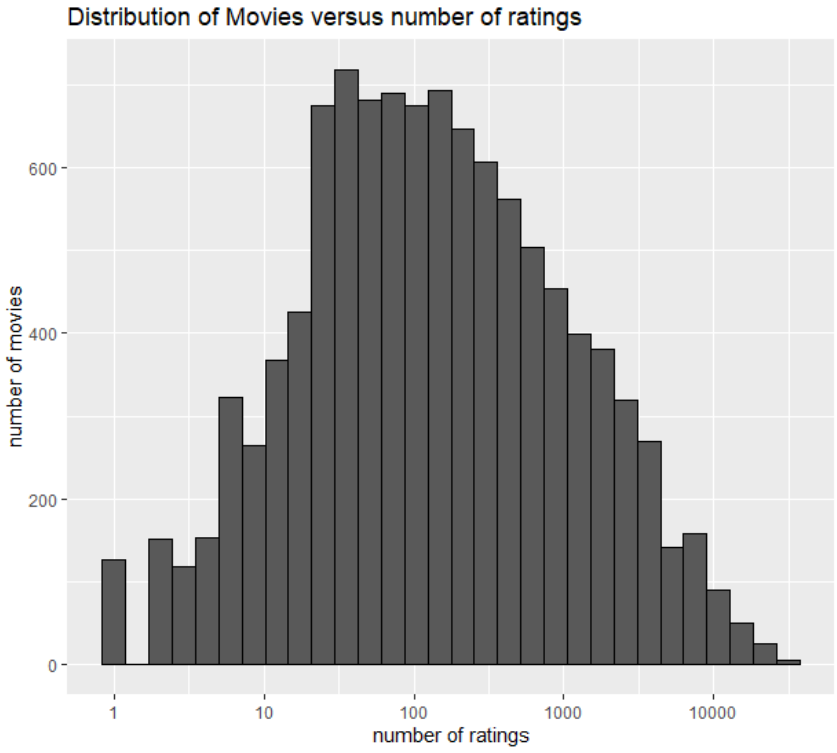

Figure 4: Movies versus number of ratings.

We can see in the histogram there are many movies with very few ratings (e.g.: less than 10) that will have an adverse effect in our prediction model and will have to be taken into account This indicates there is an effect of the movie feature over the rating outcome

2.1.4 Distribution of users versus their average rating

Let's examine the distribution users against their rating average. We are only considering those users who have rated more than 100 movies.

```
edx %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating)) %>%
  filter(n()>=100) %>%
  ggplot(aes(b_u)) +
  geom_histogram(bins = 30, color = "black") +
  xlab("User average rating") +
  ylab("Number of users") +
  ggtitle("Distribution of users versus their average rating")
```

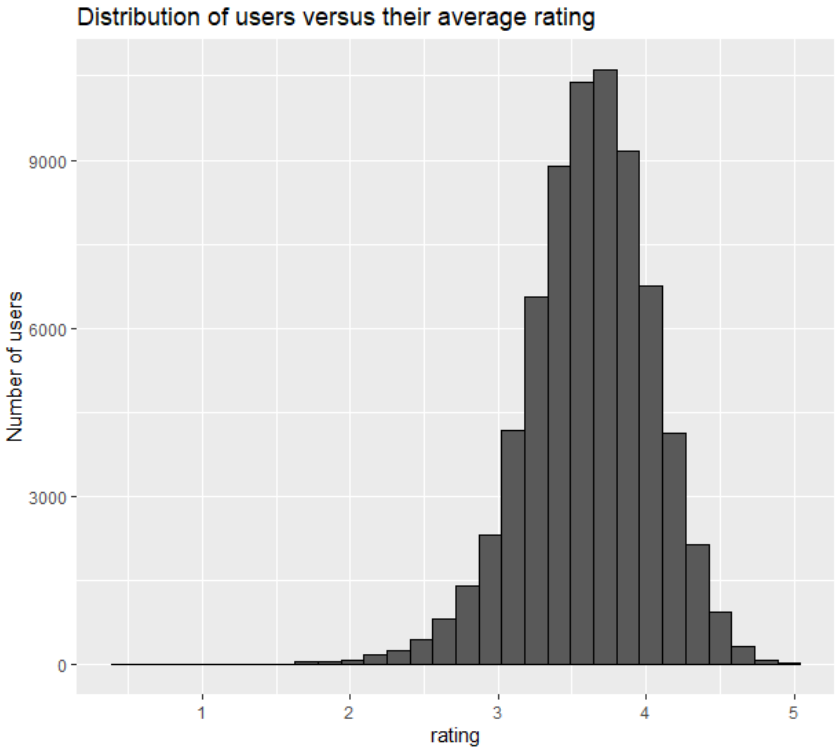

Figure 5: Users versus average rating

The histogram shows quite a uniform distribution around the global rating average. But we will check in the next section for a rating bias due to user.

Another way to see users versus their average ratings, using all points and geom_point()) function rather than as a histogram, would be the following plot:

```
edx %>%
  group_by(userId) %>%
  summarize(average_rating = mean(rating)) %>%
  ggplot(aes(userId, average_rating)) +
  geom_point(alpha = .50, color = "black") +
  geom_smooth() +
  ggtitle("Distribution of users versus their average rating")
```

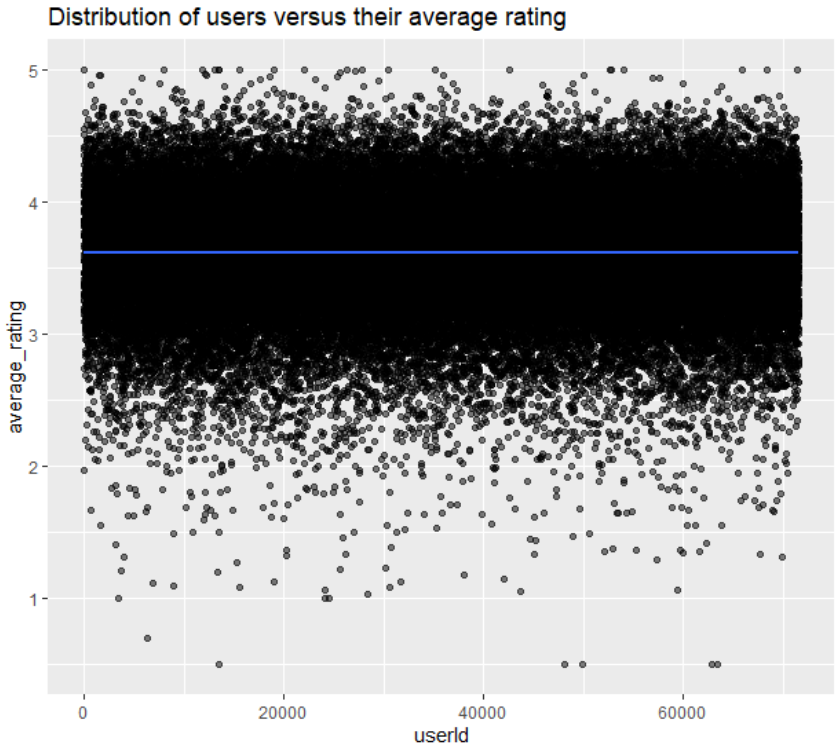

Figure 6: All users versus their average rating

The plot above shows user ratings are quite constrained within the band of 3.0 to 4.0, and the geom_smooth() line is just a horizontal line with "y" value the global average of ratings.

2.1.5 Distribution of users versus their number of ratings

A different way to examine the users is plotting them against their number of ratings.

```
edx %>%
  group_by(userId) %>%
  summarize(n = n()) %>%
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  xlab("Number of ratings") +
  ylab("Number of users") +
  ggtitle("Users versus their number of ratings")
```

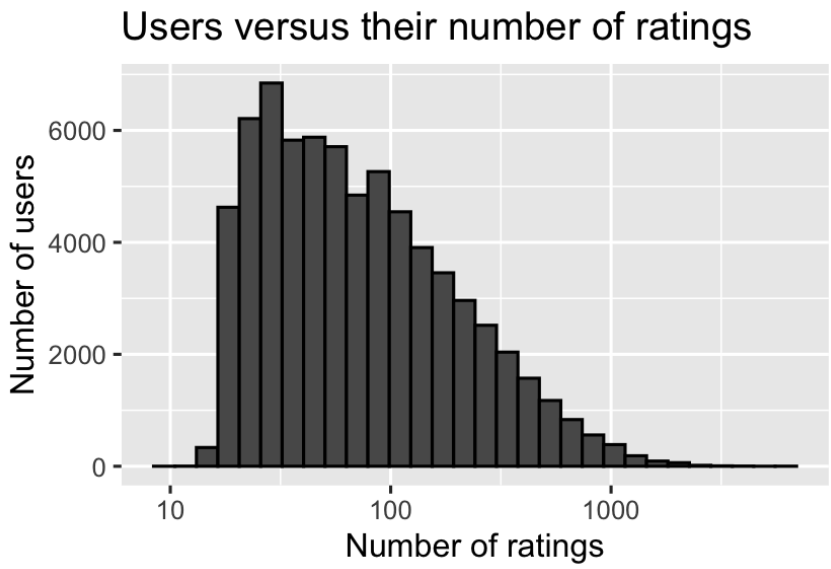

Figure 7: Users versus number of ratings

The histogram above shows how there are users quite active (e.g.: UserId 59269 with 6616 ratings) and users not very active (e.g.: least active users have 10, 12, 13 or 14 ratings). This means there might be also a user effect on rating that will be later evaluated.

2.1.6 Ratings versus genres

Let's examine the distribution of the average rating of each genre. The number of different genres is 20 as can be seen below.

```
n_distinct(edx_split_genre$genres)
```

```
[1] 20
```

The plot would be:

```
edx_split_genre %>%
  group_by(genres) %>%
  summarize(b_g = mean(rating)) %>%
  ggplot(aes(b_g)) +
  geom_histogram(bins = 30, color = "black") +
  xlab("Genre average rating") +
  ylab("Number of genres") +
  ggtitle("Distribution of genres versus rating")
```

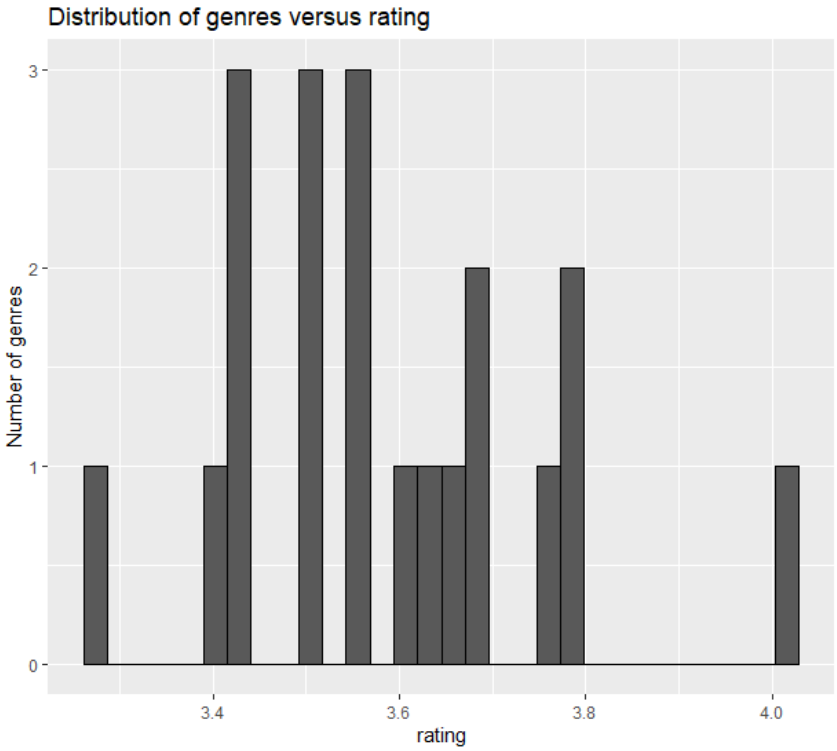

Figure 8: Genres versus ratings

To be reminded for the picture above that the total number of different genres is 20. The plot shows that almost half of the genres (9 out of 20) fall within the average rating interval of 3.4-3.6.

```
edx_split_genre %>%
  group_by(genres) %>%
  summarize(b_g = mean(rating)) %>%
  arrange(desc(b_g))
```

```
# A tibble: 20 x 2
   genres               b_g
   <chr>              <dbl>
 1 Film-Noir           4.01
 2 Documentary         3.78
 3 War                 3.78
 4 IMAX                3.77
 5 Mystery             3.68
 6 Drama               3.67
 7 Crime               3.67
 8 (no genres listed)  3.64
 9 Animation           3.60
10 Musical             3.56
11 Western             3.56
12 Romance             3.55
13 Thriller            3.51
14 Fantasy             3.50
```

```
15 Adventure           3.49
16 Comedy              3.44
17 Action              3.42
18 Children            3.42
19 Sci-Fi              3.40
20 Horror              3.27
```

The Horror genre is the one with the lowest average rating (3.27) and Film-Noir the one with the highest one (4.01). Obviously, Film-Noir movies are not the kind of ones most watched and, by extension, rated by not as many people as blockbusters. This means there is a bias associated with the genre that we need to take into account.

2.1.7 Genres trend versus release year

Let's examine the trend of movie genres versus the release year of the movie. As there are 20 different genres, in order to avoid a crowdy graph, we will consider only the 5 genres with the most entries.

```
edx_split_genre %>%
  group_by(genres) %>%
  summarize(n = n()) %>%
  arrange(desc(n)) %>%
  head(5)
```

```
# A tibble: 5 x 2
  genres          n
  <chr>       <int>
1 Drama     3910127
2 Comedy    3540930
3 Action    2560545
4 Thriller  2325899
5 Adventure 1908892
```

We now know the target genres are Drama, Comedy, Action, Thriller and Adventure. The plot would be then:

```
edx_split_genre %>%
  filter(genres %in% c("Drama", "Comedy", "Action", "Thriller", "Adventure")) %>%
  group_by(genres, release_year) %>%
  summarize(n = n()) %>%
  ggplot(aes(x = release_year, y = n)) +
  geom_line(aes(color=genres)) +
  #scale_y_log10()
  xlab("release year") +
  ylab("Genres") +
  ggtitle("Distribution of genres versus release year")
```

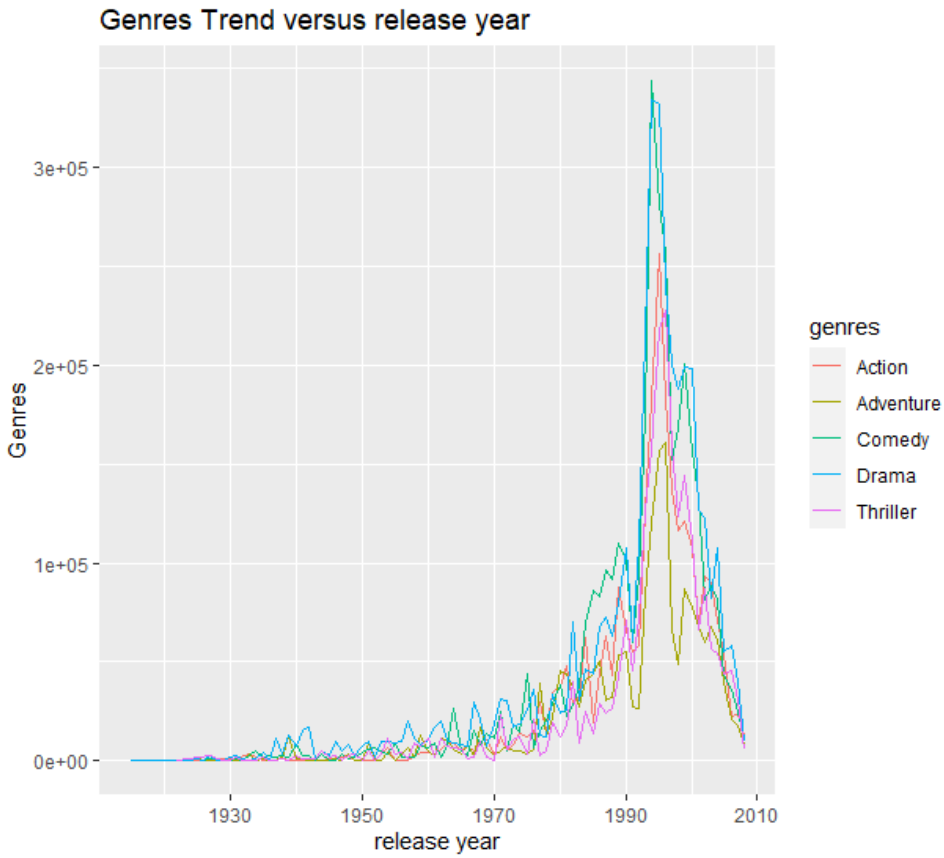

Figure 9: Genres versus release year

The plot shows that genre preference changes during time, and that could have an effect on the final rating of movies. For example, prior to 1990, the most ratings where for the Comedy genre, but after 1990, user taste changes and Drama is the genre rated the most. This means the release year has some influence on the genres and, as genre affect rating, release year would also have an effect on the rating that we have to take into account in our model.

2.1.8 Ratings versus rate year

Let's examine the distribution of the ratings versus the rating year.

```
edx %>% group_by(rate_year) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(rate_year, rating)) +
  geom_point() +
  geom_smooth() +
  xlab("rate year") +
  ylab("rating average") +
  ggtitle("Distribution of rating average versus rate year")
```

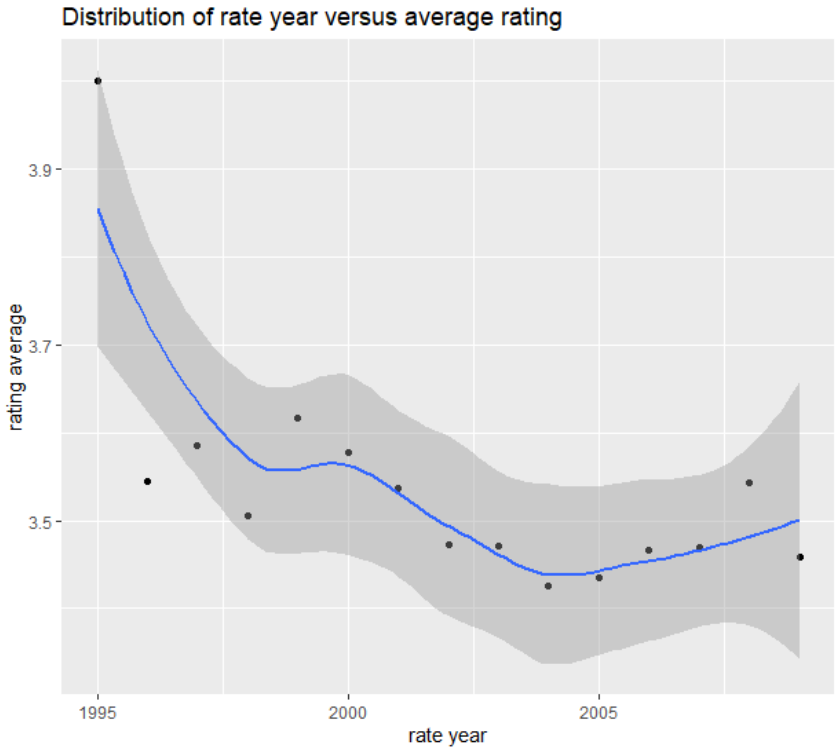

Figure 10: Rating average versus rate year

The plot above shows that the average rating of movies during a particular year has decreased since 1995 till almost 2005 when it slowly picked up. Is it because user taste changed with the years? Maybe because new movies coming out are not as good? This makes the rate_year a good candidate to have some effect on the rating.

2.1.9 Ratings versus release year

Let's examine the distribution of the ratings versus the release year of movies

```
edx %>% group_by(release_year) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(release_year, rating)) +
  geom_point() +
  geom_smooth() +
  xlab("release year") +
  ylab("rating average") +
  ggtitle("Distribution of release year versus average rating")
```

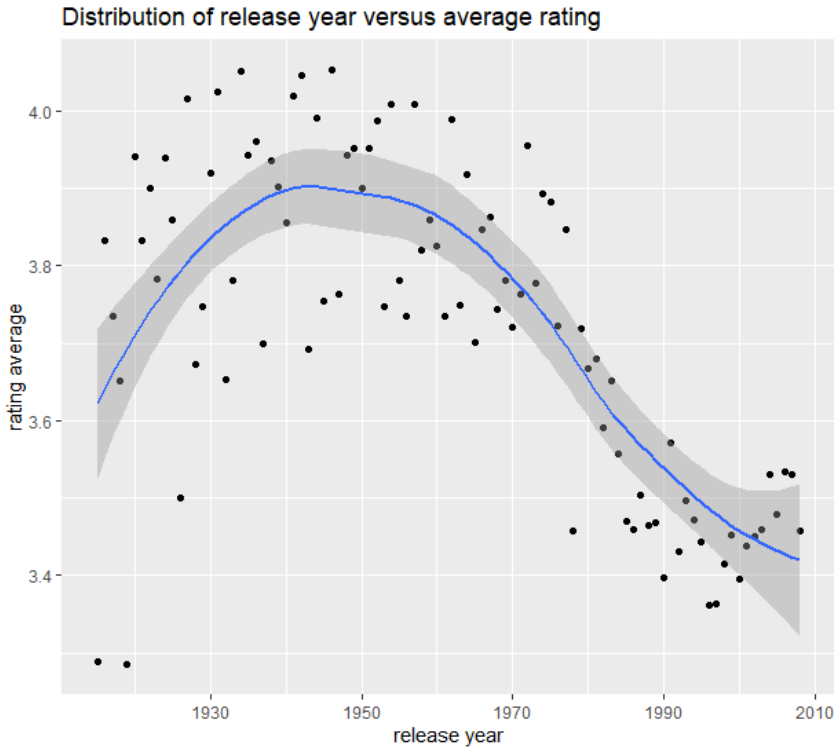

Figure 11: Rating average versus release year

The plot above shows movies from the 40's and 50's have got the highest ratings from users and, since then, the average rating of movies has decreased. Is it because in those two decades were better than movies before and after? Somehow, it seems like the release year has some effect on the rating, so we have to take into account in our prediction model.

2.2 Recomendation Models

The purpose of the movie recommendation system we are creating is to generate a set of predicted movie ratings against the validation set that minimizes as much as possible the residual mean squared error (RMSE).

As the RMSE is something we will be calculating quite a few times while training our algorithm, it is worth to create a function for it:

```
RMSE <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

Based on the different plots analyzed in previous section, we have seen there might be some potential biases affecting the rating due the movie, the user, the genre, the rate year and the release year.

We will start following the line in the course text book with a model that includes the bias effect of the different features/predictors as well as introducing some penalty for some of them. Then, we will go on with the recommendation systems based on collaborative filtering (CF).

2.2.1 Model 1: regularized bias effect of the predictors

We will take into account the bias introduced by each of the predictors (movie, user, genre, release year and rate year. We will train our model to find the lambda value that minimizes the RMSE value. For this purpose, we use the train set, edx_train_split_genre, that has individual genre entries.

Note: the lambda calculation took about 30 minutes to complete on a laptop with Windows10 OS and 16GB of RAM using most of the time the full memory and with no other application running.

```
mu <- mean(edx_train$rating)
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){

  b_i <- edx_train_split_genre %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))

  b_u <- edx_train_split_genre %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))

  b_rly <- edx_train_split_genre %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    group_by(release_year) %>%
    summarize(b_rly = sum(rating - b_u - b_i - mu)/(n()+l))

  b_g <- edx_train_split_genre %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(b_rly, by="release_year") %>%
    group_by(genres) %>%
    summarize(b_g = sum(rating - b_rly - b_u - b_i - mu)/(n()+l))

  b_rty <- edx_train_split_genre %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(b_rly, by="release_year") %>%
    left_join(b_g, by="genres") %>%
    group_by(rate_year) %>%
    summarize(b_rty = sum(rating - b_g - b_rly - b_u - b_i - mu)/(n()+l))

  predicted_ratings <-
    edx_train_split_genre %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(b_rly, by = "release_year") %>%
    left_join(b_g, by = "genres") %>%
    left_join(b_rty, by = "rate_year") %>%
    mutate(pred = mu + b_i + b_u + b_rly + b_g) %>%
    pull(pred)

  return(RMSE(predicted_ratings, edx_train_split_genre$rating))
```

```
})
```

Plotting lambdas versus rmses:

```
qplot(lambdas, rmses)
```

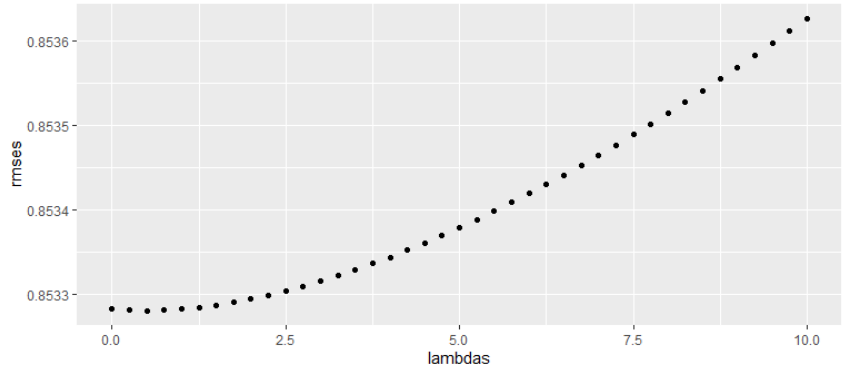

Figure 12: RMSES versus lambdas

The minimum value of the RMSE obtained using the train set is:

```
min(rmses)
```

```
[1] 0.8532805
```

The lambda that minimizes the RMSE is:

```
lambda <- lambdas[which.min(rmses)]
lambda
```

```
[1] 0.5
```

Compute regularized estimates of b_i using lambda

```
movie_avgs_reg <- edx_train_split_genre %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n()+lambda), n_i = n())
```

Compute regularized estimates of b_u using lambda

```
user_avgs_reg <- edx_train_split_genre %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = sum(rating - mu - b_i)/(n()+lambda), n_u = n())
```

Compute regularized estimates of b_rly using lambda

```
rel_year_avgs_reg <- edx_train_split_genre %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  left_join(user_avgs_reg, by='userId') %>%
  group_by(release_year) %>%
  summarize(b_rly = sum(rating - mu - b_i - b_u)/(n()+lambda), n_y = n())
```

Compute regularized estimates of b_g using lambda

```
genre_avgs_reg <- edx_train_split_genre %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  left_join(user_avgs_reg, by='userId') %>%
  left_join(rel_year_avgs_reg, by='release_year') %>%
  group_by(genres) %>%
  summarize(b_g = sum(rating - mu - b_i - b_u - b_rly)/(n()+lambda), n_y = n())
```

Compute regularized estimates of b_rty using lambda

```
rt_year_avgs_reg <- edx_train_split_genre %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  left_join(user_avgs_reg, by='userId') %>%
  left_join(rel_year_avgs_reg, by='release_year') %>%
  left_join(genre_avgs_reg, by='genres') %>%
  group_by(rate_year) %>%
  summarize(b_rty = sum(rating - mu - b_i - b_u - b_rly - b_g)/(n()+lambda), n_y = n())
```

Predict ratings for our validation set:

```
predicted_ratings_reg <- edx_test_split_genre %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  left_join(user_avgs_reg, by='userId') %>%
  left_join(rel_year_avgs_reg, by='release_year') %>%
  left_join(genre_avgs_reg, by='genres') %>%
  left_join(rt_year_avgs_reg, by='rate_year') %>%
  mutate(pred = mu + b_i + b_u + b_rly + b_g + b_rty) %>%
  .$pred
```

Calculate RMSE against validation set with generated prediction:

```
model_1_rmse <- RMSE(edx_test_split_genre$rating,predicted_ratings_reg)
model_1_rmse
```

```
[1] 0.8623166
```

```
errors <- tibble(algorithm = "POPULAR", method = "split", RMSE = 0.9410653)
errors
```

```
# A tibble: 1 x 3
  algorithm method  RMSE
  <chr>     <chr>  <dbl>
1 REG BIAS         0.862
```

Note: the RMSE value obtained against the training set is lower than the one obtained against the test set due to overtraining.

2.2.2 Collaborative Filtering (CF) Models - General Properties

We will be using the **recommender** package to test most of the CF models applied in the following sections. It has to be mentioned the recommenderlab is not a recommender algorithm itself, but it provides the infrastructure to develop and test recommender algorithms for rating data. We will be applying some of the basic recommender algorithms this package comes with (e.g.: POPULAR, UBCF, IBCF, SVD and SVDF). It also allows the user to develop and use his/her own algorithms in the framework via a simple registration procedure, but that is out of the scope of this project.

We will evaluate them using the RMSE, although they provide also other error measures like the MSE (Mean Square Error) and MAE (Mean Average Error) as it will be seen in later sections.

First we have to install the recommenderlab package:

```
if(!"recommenderlab" %in% rownames(installed.packages())){install.packages("recommenderlab")}
library("recommenderlab")
```

```
Loading required package: Matrix
Loading required package: arules
Loading required package: proxy
Loading required package: registry
Attaching package: 'recommenderlab'
```

To see the CF algorithms included in the recommender package:

```
recommendation_model = recommenderRegistry$get_entries(dataType = "realRatingMatrix")
names(recommendation_model)
```

```
 [1] "HYBRID_realRatingMatrix"       "ALS_realRatingMatrix"
 [3] "ALS_implicit_realRatingMatrix" "IBCF_realRatingMatrix"
 [5] "LIBMF_realRatingMatrix"        "POPULAR_realRatingMatrix"
 [7] "RANDOM_realRatingMatrix"       "RERECOMMEND_realRatingMatrix"
 [9] "SVD_realRatingMatrix"          "SVDF_realRatingMatrix"
[11] "UBCF_realRatingMatrix"
```

We will also install the e1071 package, as it will be needed for some of the functions of the algorithsm included in the recommenderlab package. It contains Functions for latent class analysis, short time Fourier transform, fuzzy clustering, support vector machines, shortest path computation, bagged clustering, naive Bayes classifier, etc.

```
if(!"e1071" %in% rownames(installed.packages())){install.packages("e1071")}
library(e1071)
```

```
if(!"devtools" %in% rownames(installed.packages())){install.packages("devtools")}
library(devtools)
```

Although the plan was to use the edx dataset (9M rows) for the evaluation of the CF models, it was too big for some of them (e.g.: UBCF and IBCF), and either creating the evaluation schema originated from the rating matrix obtained from edx dataset or calculating predictions for the models never completed to execute. Because of this, to initially evaluate the CF models, we have created a subset (edx_cf) sized in a 10% of the original edx dataset. I had previously verified the CF models used worked fine with the 1M dataset from MovieLens available at https://grouplens.org/datasets/movielens/ and the zip file for it directly downloaded from http://files.grouplens.org/datasets/movielens/ml-1m.zip

```
set.seed(2021, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
cf_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1, list = FALSE)
edx_cf <- edx[cf_index,]
nrow(edx_cf)
```

```
[1] 900007
```

2.2.2.1 Rating Matrix A rating matrix, of realRatingMatrix type, is the basic input for all recommender CF models. Each row represents a user and each column a movie, and element ij in the matrix would be the rating of user "i" for movie "j". So, before getting into the CF algorithms, let's obtain the rating matrix from our edx_cf dataset for future use. Most CF algorithms take as input a rating matrix and split it into train and test. Because of this, there is no need at this point to partition the edx_cf dataset into train and test sets as we did for bias model and we will generate our rating matrix directly from the edx_cf dataset, leaving the validation set for final evaluation of the best model.

Let's first create the sparse matrix

```
sparse_ratings <- sparseMatrix(i = edx_cf$userId, j = edx_cf$movieId, x = edx_cf$rating,
                               dims = c(max(edx_cf$userId), max(edx_cf$movieId)),
                               dimnames = list(paste("u", seq(max(edx_cf$userId)), sep = ""),
                                               paste("m", seq(max(edx_cf$movieId)), sep = "")))
sparse_ratings
```

```
71567 x 65133 sparse Matrix of class "dgCMatrix"
```

```
str(sparse_ratings)
```

```
Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
  ..@ i       : int [1:900007] 29 83 100 141 143 174 262 281 294 299 ...
  ..@ p       : int [1:65134] 0 2351 3382 4104 4246 4886 6138 6847 6936 7145 ...
  ..@ Dim     : int [1:2] 71567 65133
  ..@ Dimnames:List of 2
  .. ..$ : chr [1:71567] "u1" "u2" "u3" "u4" ...
  .. ..$ : chr [1:65133] "m1" "m2" "m3" "m4" ...
  ..@ x       : num [1:900007] 5 3 5 4 4.5 4 4.5 4 4 5 ...
  ..@ factors : list()
```

Let's have a look at its first 10 rows and columns

```
sparse_ratings[1:10,1:10]
```

```
10 x 10 sparse Matrix of class "dgCMatrix"
   [[ suppressing 10 column names 'm1', 'm2', 'm3' ... ]]

u1  . . . . . . . . . .
u2  . . . . . . . . . .
u3  . . . . . . . . . .
u4  . . . . . . . . . .
u5  . . . . . . . . . .
u6  . . . . . . . . . .
u7  . . . . . . . . . .
```

```
u8  . . . . . .
u9  . . . . . .
u10 . . . . . .
```

Not a very good example the first 10 rows and columns, as there is not a single rating in them. Normally, there should be a few ratings among most empty values. Let's convert now the sparse matrix into a realRatingMatrix:

```
real_ratings <- new("realRatingMatrix", data = sparse_ratings)
real_ratings
```

```
71567 x 65133 rating matrix of class 'realRatingMatrix' with 900007 ratings.
```

From the rating matrix we can easily plot histograms for different measurements: The *breaks* parameter is the number of cells for the histogram

- Histograms of ratings

```
hist(getRatings(real_ratings), breaks=100)
```

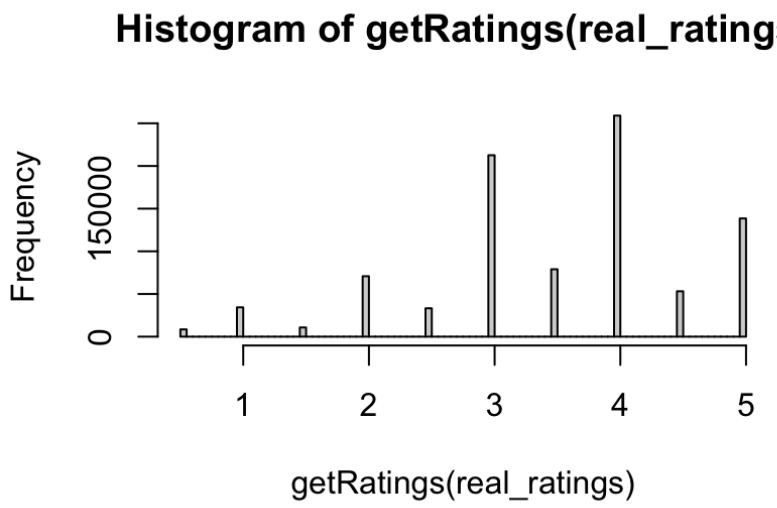

Figure 13: Histogram of ratings

- Histogram of normalized ratings using row centering

```
hist(getRatings(normalize(real_ratings)), breaks=100)
```

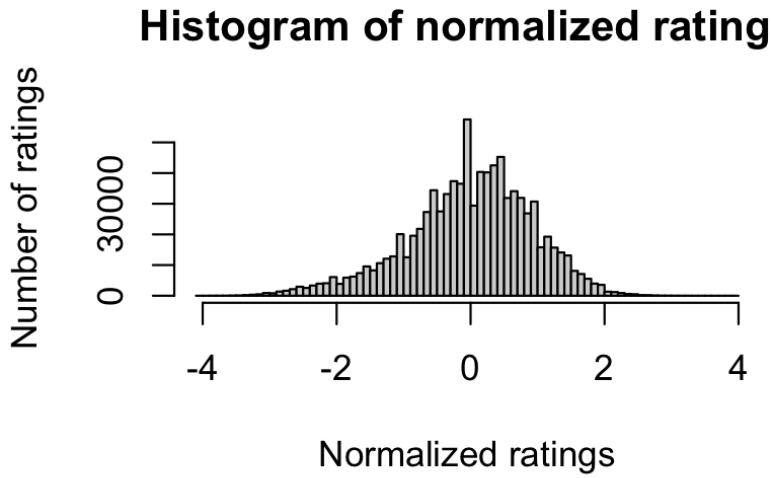

Figure 14: Histogram of normalized ratings

- Histogram of normalized ratings using Z-score normalization

```
hist(getRatings(normalize(real_ratings, method="Z-score")), breaks=100)
```

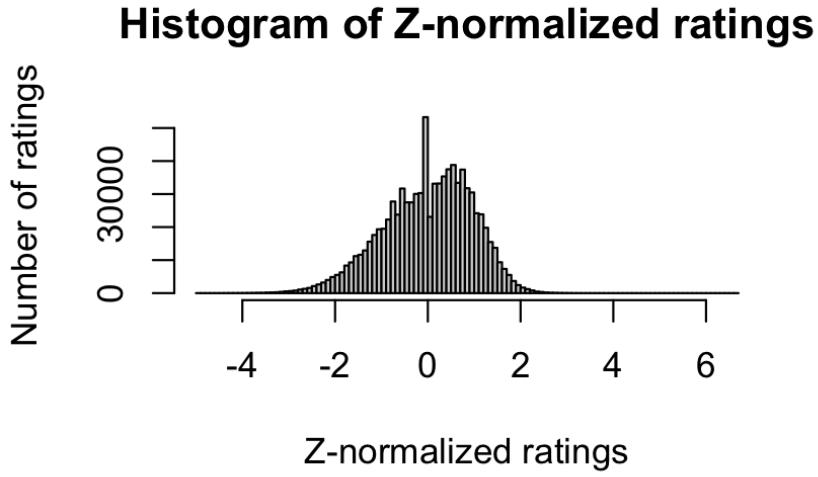

Figure 15: Histogram of Z-normalized ratings

- Histogram of the number of rated items per user

```
hist(rowCounts(real_ratings), breaks=50)
```

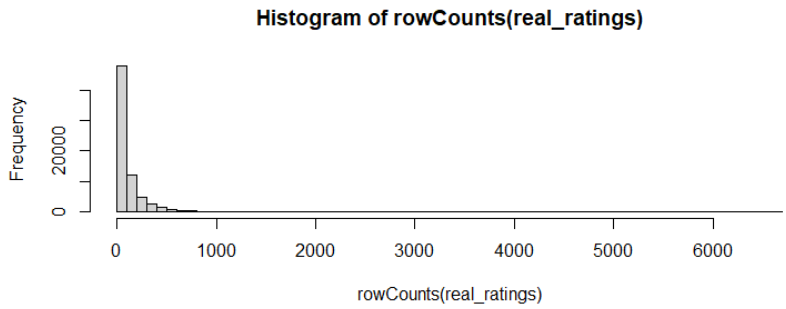

Figure 16: Histograms of rated movies per user

- Histogram of number of movies per average rating

```
hist(colMeans(real_ratings), breaks=20)
```

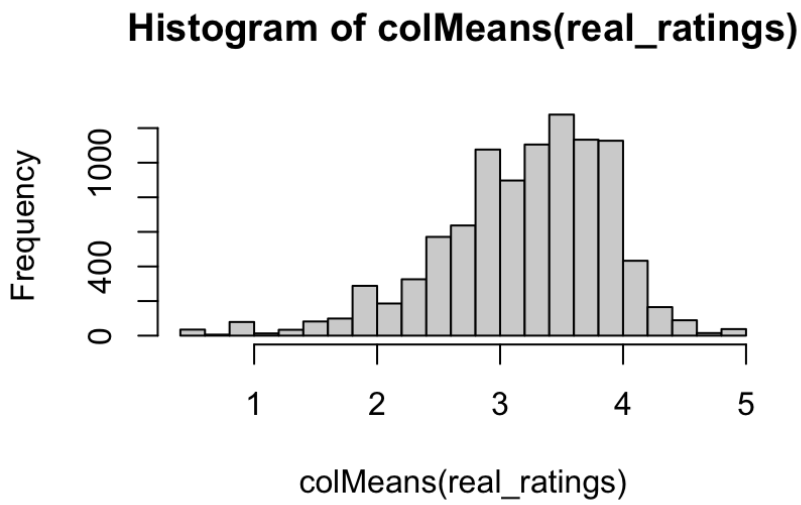

Figure 17: Histogram of number of movies per average rating

2.2.2.2 Similarity Matrix Recommending movies is dependent on creating a relationship of similarity between pairs of users. With the help of the recommenderlab, we can compute similarities using various operators like cosine, pearson as well as jaccard. To create the similarity matrix of the first 4 users in our rating matrix

```
similarity_mat = similarity(real_ratings[1:4,],method = "cosine",which="users")
as.matrix(similarity_mat)
```

```
   u1 u2 u3 u4
u1  0 NA NA  1
u2 NA  0 NA NA
```

```
u3 NA NA  0 NA
u4  1 NA NA  0
```

Each cell in this matrix represents the similarity that is shared between the two users. And showing it as an image:

```
image(as.matrix(similaritY_mat),main="Users Similarities")
```

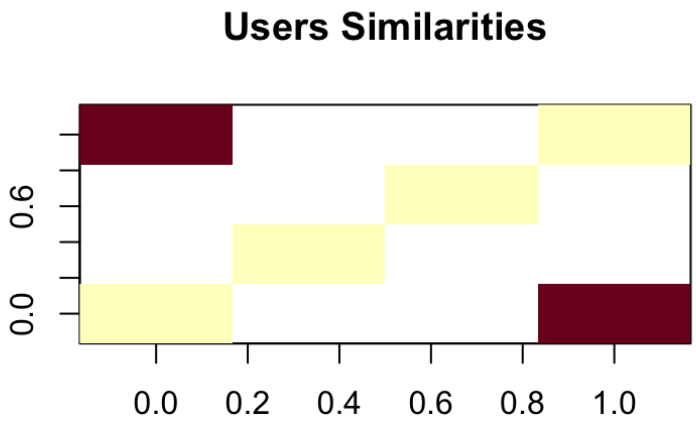

Figure 18: User similarities - first 4 users in real matrix

We can also describe the similarity that is shared between the movies:

```
movie_similarity = similarity(real_ratings[,1:4],method="cosine",which="items")
as.matrix(movie_similarity)
```

```
          m1        m2        m3        m4
m1 0.0000000 0.9465773 0.9592979 0.8923014
m2 0.9465773 0.0000000 0.9394917 0.9860239
m3 0.9592979 0.9394917 0.0000000 0.9620599
m4 0.8923014 0.9860239 0.9620599 0.0000000
```

And also show it as an image.

```
image(as.matrix(movie_similarity), main="Movies Similarities")
```

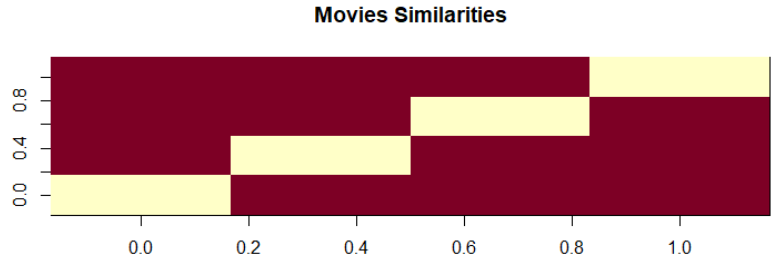

Figure 19: Movie similarities - first 4 movies in real matrix

2.2.2.3 HeatMap of Movie Ratings We can also see an image of the realRatingMatrix that is called a heatmap. This shows users on the rows and movies on the columns and each intersection pixel has the rating value in a scale of gray color. For example, a heatmap containing the first 25 rows and 25 columns:

```
image(real_ratings[1:25,1:25],axes=FALSE,main="HeatMap of first 25  rows and 25 columns")
```

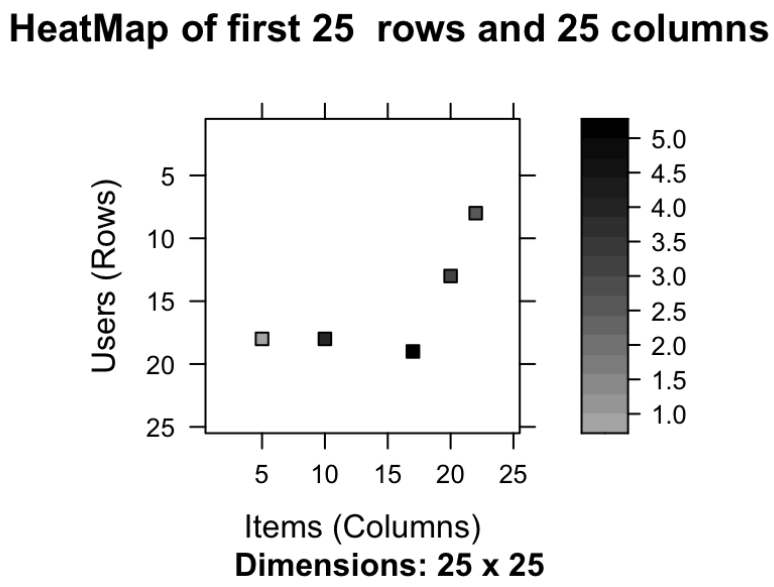

Figure 20: Heatmap of first 25 users and movies

We can see there are only 5 ratings among the 25x25 (= 625) pixels, which indicates a high level of sparsity (very few ratings among the 25x25 matrix), although the sparsity is even higher for the whole rating matrix (900007 / (71567 * 65133) = 0.00019).

For finding useful data in our dataset, we can take into account only those users who have rated, for example, at least 50 films. We can also apply this threshold to movies that have to be seen at least for a minimum number of users (e.g.: 50). In this way, we have filtered a list of watched films from least-watched ones. We can easily do that on our rating matrix:

```
movie_ratings = real_ratings[rowCounts(real_ratings)>50 , colCounts(real_ratings)>50]
movie_ratings
```

```
3006 x 2834 rating matrix of class 'realRatingMatrix' with 204983 ratings.
```

We have reduced from 71567 users x 65133 movies to 3006 users x 2834 movies and the sparsity ratio is lower (204983 / (3006 * 2834) = 0.024). We can also look for the most active users and most viewed movies:

```
minimum_movies =  quantile(rowCounts(movie_ratings),0.98)
minimum_movies
```

```
98%
138
```

```
minimum_users  =  quantile(colCounts(movie_ratings),0.98)
minimum_users
```

```
98%
221
```

The heatmap of these top users and movies would be:

```
image(movie_ratings[rowCounts(movie_ratings)>minimum_movies, colCounts(movie_ratings)>minimum_users], ma
```

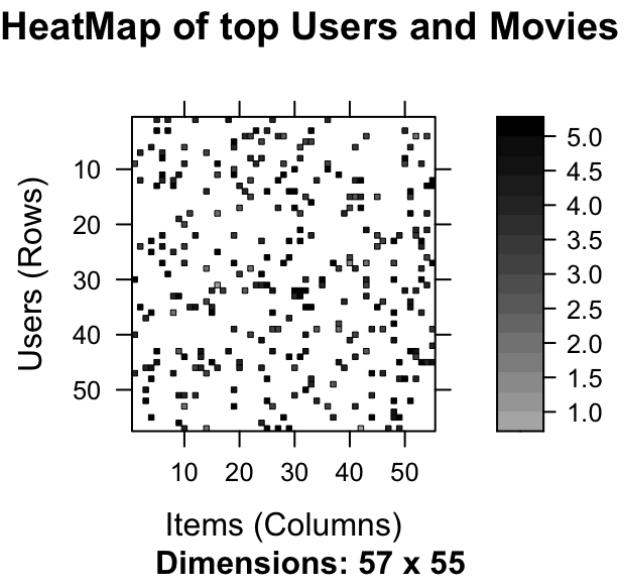

Figure 21: HeatMap of top Users and Movies

And the rating matrix for these top users and movies would be:

```
new_movie_ratings <- movie_ratings[rowCounts(movie_ratings)>minimum_movies, colCounts(movie_ratings)>mi
new_movie_ratings
```

```
57 x 55 rating matrix of class 'realRatingMatrix' with 287 ratings.
```

As seen above, the top is made of 57 users and 55 movies. The sparsity ratio in this case is 0.09 (287 / (57 * 55)).

2.2.2.4 Matrix Normalization An important operation for rating matrices is normalization. These operation is important, as it allows us to eliminate the user bias, where users can be high raters or low raters and have the adverse effect of distorting the rating distribution. By normalizing the matrix, we remove the user rating bias by subtracting the row mean from all ratings in the row. This is can be easily done using normalize(). The *method* parameter allows to choose the type of normalization, *center* or *Z-score*. The Z-score centering removes the row mean from each row value and then divides by the standard deviation. By default, *center* is applied.

```
normalized_ratings <- normalize(movie_ratings)
normalized_ratings
```

```
3006 x 2834 rating matrix of class 'realRatingMatrix' with 204983 ratings.
Normalized using center on rows.
```

The heatmap of the first 25 rows and columns would be:

```
image(normalized_ratings[1:25,1:25],axes=FALSE,main="HeatMap of 1st 25rows & 25columns (norm. matrix)")
```

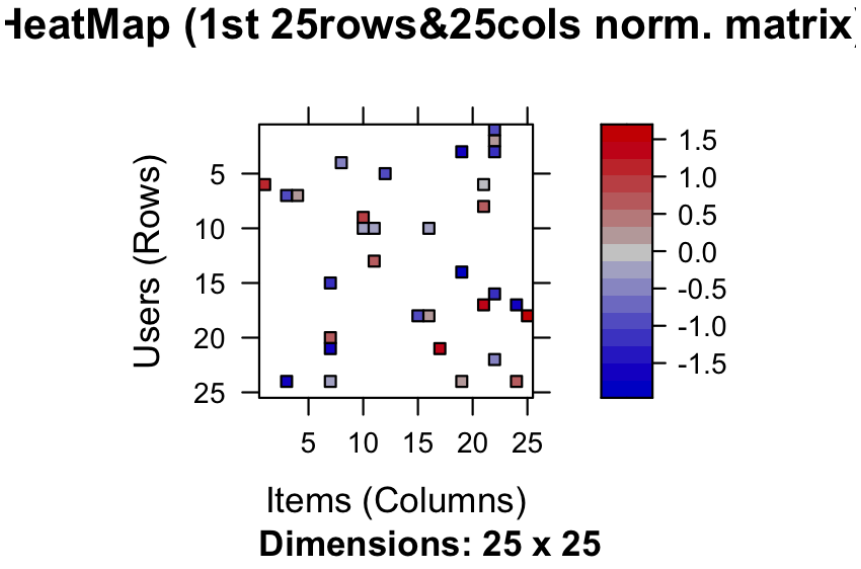

Figure 22: Heatmap of the first 25 users and movies in normalized rating matrix

Using the Z-score centering

```
normalized_ratings <- normalize(movie_ratings, "Z-score")
normalized_ratings
```

```
2423 x 2694 rating matrix of class 'realRatingMatrix' with 1549141 ratings.
Normalized using z-score on rows.
```

The heatmap of the first 25 rows and columns would be:

```
image(normalized_ratings[1:25,1:25],axes=FALSE,main="HeatMap of first 25  rows and 25 columns (normaliz
```

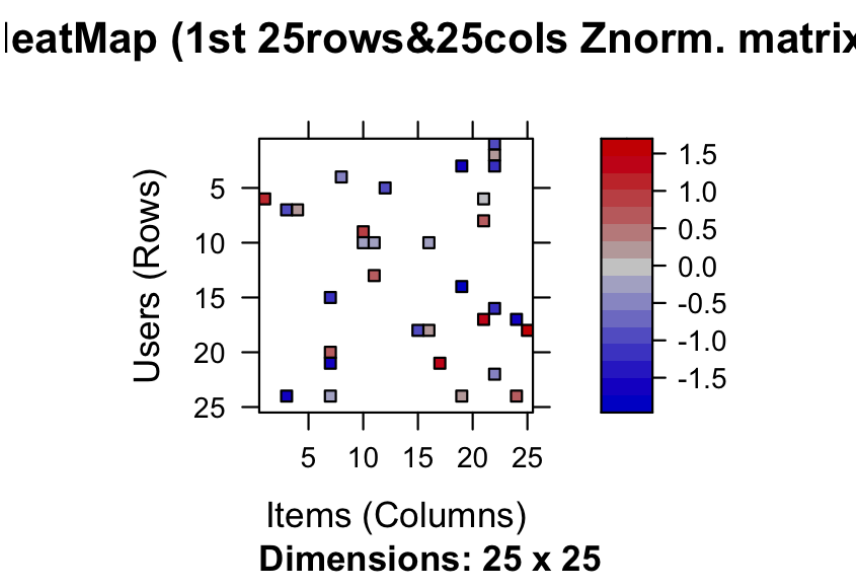

Figure 23: Heatmap of the first 25 users and movies in Z-normalized rating matrix.

2.2.2.5 Matrix Binarization The rating matrix, realRatingMatrix, is sometimes useful to convert it into a binary matrix. This is done by choosing a threshold rate. For this example we will define a matrix that will consist of 1 if the rating is above the average rating and 0 otherwise.

```
threshold <- mean(edx_cf$rating)
threshold
```

```
[1] 3.513019
```

```
binary_matrix <- binarize(movie_ratings, minRating = threshold)
binary_matrix
```

```
3006 x 2834 rating matrix of class 'binaryRatingMatrix' with 88828 ratings.
```

The heatmap of the first 25 rows and columns would be:

```
image(binary_matrix[1:25,1:25],axes=FALSE,main="HeatMap of first 25  rows and 25 columns (binarized mat
```

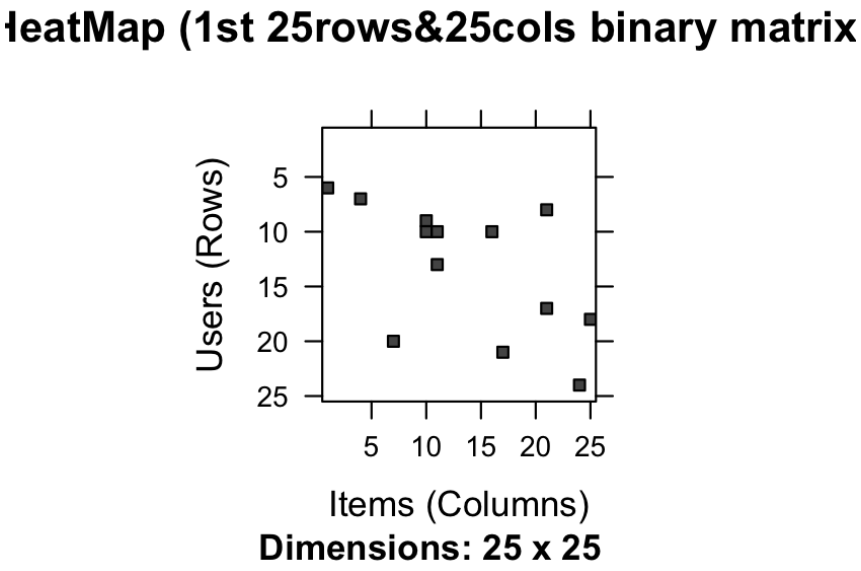

Figure 24: Heatmap of the first 25 users and movies in binarized rating matrix.

2.2.3 Collaborative Filtering (CF) Evaluation Schemas - Data Preparation

The recommenderlab framework does the data split between train and test data on its own from input matrix of realRatingMatrix type. We will use the edx_cf dataset that created with a 10% of the edx one.

To evaluate the different CF algorithms within the recommenderlab framework, we need an evaluation schema, where we specify different parameters. These parameters are:

- *method*: instructs how to split the input data in between train and test. We will be creating an evaluation schema for each of the available methods (split, cross-validation and bootstrap).
- *given*: indicates the number of randomly chosen items of the test users that are given to the recommender algorithm and the remaining items are withheld for evaluation. The value for the *given* parameter cannot exceed the number of rated movies for the user who has rated the least movies. To find out this value:

```
min(rowCounts(real_ratings))
```

```
# [1] 0
```

```
real_ratings
```

```
71567 x 65133 rating matrix of class 'realRatingMatrix' with 900007 ratings.
```

We do have a problem with the value above, as it is zero and we want for CF algorithms this *given* parameter to be greater than zero. Let's avoid those rows (users) who have evaluated less than 10 or 20 movies:

```
real_ratings_small <- real_ratings[rowCounts(real_ratings)>10, colCounts(real_ratings)>10]
min(rowCounts(real_ratings_small))
```

```
[1] 5
```

```
real_ratings_small
```

```
23533 x 5609 rating matrix of class 'realRatingMatrix' with 681620 ratings.
```

```
real_ratings_small <- real_ratings[rowCounts(real_ratings)>20, colCounts(real_ratings)>20]
min(rowCounts(real_ratings_small))
```

```
[1] 11
```

```
real_ratings_small
```

```
11881 x 4331 rating matrix of class 'realRatingMatrix' with 497019 ratings.
```

Let's go with the last value of 11. This means there minimum number of movies rated by some of our users is 11, and we are constrained by that the number of ratings to give to our algorithms. Let's pick 5 as the value for the *given* parameter. This means that the minimum ratings of a user that we will keep as "unknown" to the algorithms will be 6. It is the algorithm the one that, randomly, chooses which ratings to give and which to keep/hide.

Let's create now the different evaluation schemas:

- The split method randomly assigns the proportion of objects specified by the *train* parameter to the training set and the rest is used for the test set.

```
set.seed(1, sample.kind="Rounding")
e_split <- evaluationScheme(real_ratings_small, method="split", train=0.8, given=15)
```

- The cross-validation method creates a k-fold cross-validation scheme. The data is randomly split into k parts (as indicated in the *k* parameter) and in each run k-1 parts are used for training and the remaining part is used for testing. After all k runs each part was used as the test set exactly once.

```
e_cross <- evaluationScheme(real_ratings_small, method="cross-validation", k=10, given=15)
```

- The bootstrap method creates the training set by taking a bootstrap sample (sampling with replacement) of size train times number of users in the data set (as indicated in the *train* parameter). All objects not in the training set are then used for testing.

```
e_bootstrap <- evaluationScheme(real_ratings_small, method="bootstrap", train=0.8, given=15)
```

2.2.4 Collaborative Filtering (CF) Popular Model

```
rec_popular <- Recommender(getData(e_split, "train"), "POPULAR")
predict_popular <- predict(rec_popular, getData(e_split, "known"), type="ratings")
class(predict_popular)
```

2.2.4.1 Split

```
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

```
rmse_popular <- calcPredictionAccuracy(predict_popular, getData(e_split, "unknown"))
rmse_popular
```

```
     RMSE       MSE       MAE
0.9410653 0.8856038 0.7267630
```

We can see these error figures for each individual user:

```
head(calcPredictionAccuracy(predict_popular, getData(e_split, "unknown"), byUser=TRUE))
```

```
          RMSE       MSE       MAE
u19  0.9976508 0.9953072 0.7781450
u96  1.0809779 1.1685131 0.7796928
u212 0.8371931 0.7008922 0.7057743
u249 0.7953660 0.6326071 0.5277210
u274 0.6230444 0.3881843 0.5017042
u300 0.7684000 0.5904386 0.6205414
```

```
rec_popular_k <- Recommender(getData(e_cross, "train"), "POPULAR")
predict_popular_k <- predict(rec_popular_k, getData(e_cross, "known"), type="ratings")
rmse_popular_k <- calcPredictionAccuracy(predict_popular_k, getData(e_cross, "unknown"))
rmse_popular_k
```

2.2.4.2 Cross-Validation

```
     RMSE       MSE       MAE
0.9595539 0.9207438 0.7450115
```

```
rec_popular_b <- Recommender(getData(e_bootstrap, "train"), "POPULAR")
predict_popular_b <- predict(rec_popular_b, getData(e_bootstrap, "known"), type="ratings")
rmse_popular_b <- calcPredictionAccuracy(predict_popular_b, getData(e_bootstrap, "unknown"))
rmse_popular_b
```

2.2.4.3 Bootstrap

```
     RMSE       MSE       MAE
0.9539857 0.9100887 0.7396942
```

To carry results along, let's create a table where we can keep adding the outcomes of our different models for final decision on best one. In the POPULAR model, the best RMSE (0.9410653) is obtained for the split method.

```
errors <- bind_rows(errors,
                    tibble(algorithm = "POPULAR", method = "split", RMSE = 0.9410653))
errors %>% knitr::kable()
```

```
|algorithm |method    |      RMSE|
|:---------|:---------|---------:|
|REG BIAS  |          | 0.8623166|
|POPULAR   |split     | 0.9410653|
```

2.2.5 User Based Collaborative Filtering (UBCF) Model

```
rec_ubcf <- Recommender(getData(e_split, "train"), "UBCF")
predict_ubcf <- predict(rec_ubcf, getData(e_split, "known"), type="ratings")
rmse_ubcf <- calcPredictionAccuracy(predict_ubcf, getData(e_split, "unknown"))
rmse_ubcf
```

2.2.5.1 Split

```
     RMSE       MSE       MAE
1.1937031 1.4249272 0.9320218
```

```
rec_ubcf_k <- Recommender(getData(e_cross, "train"), "UBCF")
predict_ubcf_k <- predict(rec_ubcf_k, getData(e_cross, "known"), type="ratings")
rmse_ubcf_k <- calcPredictionAccuracy(predict_ubcf_k, getData(e_cross, "unknown"))
rmse_ubcf_k
```

2.2.5.2 Cross-Validation

```
     RMSE       MSE       MAE
1.1891657 1.4141150 0.9285035
```

```
rec_ubcf_b <- Recommender(getData(e_bootstrap, "train"), "UBCF")
predict_ubcf_b <- predict(rec_ubcf_b, getData(e_bootstrap, "known"), type="ratings")
rmse_ubcf_b <- calcPredictionAccuracy(predict_ubcf_b, getData(e_bootstrap, "unknown"))
rmse_ubcf_b
```

2.2.5.3 Bootstrap

```
     RMSE       MSE       MAE
1.1983469 1.4360353 0.9341079
```

The best result for UBCF is RMSE of 1.1891657 for cross-validation method

```
errors <- bind_rows(errors,
                    tibble(algorithm = "UBCF", method = "cross-val", RMSE = 1.1891657))
errors %>% knitr::kable()
```

```
|algorithm |method    |      RMSE|
|:---------|:---------|---------:|
|REG BIAS  |          | 0.8623166|
|POPULAR   |split     | 0.9410653|
|UBCF      |cross-val | 1.1891657|
```

2.2.6 Item Based Collaborative Filtering (IBCF) Model

```
rec_ibcf <- Recommender(getData(e_split, "train"), "IBCF")
predict_ibcf <- predict(rec_ibcf, getData(e_split, "known"), type="ratings")
rmse_ibcf <- calcPredictionAccuracy(predict_ibcf, getData(e_split, "unknown"))
rmse_ibcf
```

2.2.6.1 Split

```
    RMSE      MSE      MAE
1.344568 1.807864 1.005900
```

```
rec_ibcf_k <- Recommender(getData(e_cross, "train"), "IBCF")
predict_ibcf_k <- predict(rec_ibcf_k, getData(e_cross, "known"), type="ratings")
rmse_ibcf_k <- calcPredictionAccuracy(predict_ibcf_k, getData(e_cross, "unknown"))
rmse_ibcf_k
```

2.2.6.2 Cross-Validation

```
    RMSE      MSE      MAE
1.352365 1.828892 1.018934
```

```
rec_ibcf_b <- Recommender(getData(e_bootstrap, "train"), "IBCF")
predict_ibcf_b <- predict(rec_ibcf_b, getData(e_bootstrap, "known"), type="ratings")
rmse_ibcf_b <- calcPredictionAccuracy(predict_ibcf_b, getData(e_bootstrap, "unknown"))
rmse_ibcf_b
```

2.2.6.3 Bootstrap

```
    RMSE      MSE      MAE
1.366012 1.865990 1.027792
```

The best result for IBCF is RMSE of 1.688562 for split method

```
errors <- bind_rows(errors,
                    tibble(algorithm = "IBCF", method = "split", RMSE = 1.344568))
errors %>% knitr::kable()
```

algorithm	method	RMSE
REG BIAS		0.8623166
POPULAR	split	0.9410653
UBCF	cross-val	1.1891657
IBCF	split	1.3445680

2.2.7 Singular Value Decomposition (SVD) CF Model

```
rec_svd <- Recommender(getData(e_split, "train"), "SVD")
predict_svd <- predict(rec_svd, getData(e_split, "known"), type="ratings")
rmse_svd <- calcPredictionAccuracy(predict_svd, getData(e_split, "unknown"))
rmse_svd
```

2.2.7.1 Split

```
     RMSE       MSE       MAE
1.0433052 1.0884857 0.8125679
```

```
rec_svd_k <- Recommender(getData(e_cross, "train"), "SVD")
predict_svd_k <- predict(rec_svd_k, getData(e_cross, "known"), type="ratings")
rmse_svd_k <- calcPredictionAccuracy(predict_svd_k, getData(e_cross, "unknown"))
rmse_svd_k
```

2.2.7.2 Cross-Validation

```
     RMSE       MSE       MAE
1.0604307 1.1245132 0.8290041
```

```
rec_svd_b <- Recommender(getData(e_bootstrap, "train"), "SVD")
predict_svd_b <- predict(rec_svd_b, getData(e_bootstrap, "known"), type="ratings")
rmse_svd_b <- calcPredictionAccuracy(predict_svd_b, getData(e_bootstrap, "unknown"))
rmse_svd_b
```

2.2.7.3 Bootstrap

```
     RMSE       MSE       MAE
1.0516476 1.1059626 0.8198608
```

The best result for SVD is RMSE of 1.0657260 for method bootstrap

```
errors <- bind_rows(errors,
                    tibble(algorithm = "SVD", method = "split", RMSE = 1.0433052))
errors %>% knitr::kable()
```

```
|algorithm |method    |      RMSE|
|:---------|:---------|---------:|
|REG BIAS  |          | 0.8623166|
|POPULAR   |split     | 0.9410653|
|UBCF      |cross-val | 1.1891657|
|IBCF      |split     | 1.3445680|
|SVD       |split     | 1.0433052|
```

2.2.8 Singular Value Decomposition Funk (SVDF) CF MODEL

```
rec_svdf <- Recommender(getData(e_split, "train"), "SVDF")
predict_svdf <- predict(rec_svdf, getData(e_split, "known"), type="ratings")
rmse_svdf <- calcPredictionAccuracy(predict_svdf, getData(e_split, "unknown"))
rmse_svdf
```

2.2.8.1 Split

```
     RMSE       MSE       MAE
0.9756173 0.9518291 0.7520042
```

```
rec_svdf_k <- Recommender(getData(e_cross, "train"), "SVDF")
predict_svdf_k <- predict(rec_svdf_k, getData(e_cross, "known"), type="ratings")
rmse_svdf_k <- calcPredictionAccuracy(predict_svdf_k, getData(e_cross, "unknown"))
rmse_svdf_k
```

2.2.8.2 Cross-Validation

```
     RMSE       MSE       MAE
0.9870395 0.9742470 0.7637286
```

```
rec_svdf_b <- Recommender(getData(e_bootstrap, "train"), "SVDF")
predict_svdf_b <- predict(rec_svdf_b, getData(e_bootstrap, "known"), type="ratings")
rmse_svdf_b <- calcPredictionAccuracy(predict_svdf_b, getData(e_bootstrap, "unknown"))
rmse_svdf_b
```

2.2.8.3 Bootstrap

```
     RMSE       MSE       MAE
0.9869011 0.9739739 0.7623337
```

The best result for SVDF is RMSE of 0.9756173 for method split

```
errors <- bind_rows(errors,
                    tibble(algorithm = "SVDF", method = "split", RMSE = 0.9756173))
errors %>% knitr::kable()
```

```
|algorithm |method    |      RMSE|
|:---------|:---------|---------:|
|REG BIAS  |          | 0.8623166|
|POPULAR   |split     | 0.9410653|
|UBCF      |cross-val | 1.1891657|
|IBCF      |split     | 1.3445680|
|SVD       |split     | 1.0433052|
|SVDF      |split     | 0.9756173|
```

2.2.9 Recosystem Model

The Recosystem is an R wrapper of the 'libmf' library (http://www.csie.ntu.edu.tw/~cjlin/libmf/) for recommender system using matrix factorization and typically used to approximate an incomplete matrix using the product of two matrices in a latent space. The concept of latent space is related to the latent factors defined in it and that will be tuned in some of the parameters used in the turne() method later. The main task of the recommender system is to predict unknown entries in the rating matrix based on observed values.

LIBMF itself is a parallelized library, meaning that users can take advantage of multicore CPUs to speed up the computation. The recosystem is a wrapper of LIBMF, hence the features of LIBMF are all included in recosystem. Also, unlike most other R packages for statistical modeling which store the whole dataset and model object in memory, LIBMF (and hence recosystem) is much hard-disk-based, for instance the constructed model which contains information for prediction can be stored in the hard disk, and prediction result can also be directly written into a file rather than kept in memory. That is to say, recosystem will have a comparatively small memory usage. This is noticed in the steps below where their input is read from a file or their outcome written to a file.

I could have used the LIBMF_realRatingMatrix model that comes within the recommenderlab package, but preferred to used the recosystem package on its own to make things a bit different.

We will use the edx_cf dataset from which we obtained the realRatingMatrix for the CF methods used so far. We will have to split it in between train and test sets, as the recosystem model doesn't do the split on its own as previous CF models. But first we filter it to select just the userId, movieId and rating columns, as that is all we need

```
names(edx_cf)
```

```
[1] "userId"       "movieId"      "rating"       "timestamp"    "title"
[6] "genres"       "rate_year"    "release_year"
```

```
edx_cf_clean <- select(edx_cf, userId, movieId, rating)
head(edx_cf_clean)
```

```
   userId movieId rating
1:      1     364    5.0
```

```
2:      1     466    5.0
3:      2     260    5.0
4:      2    1356    3.0
5:      3    1276    3.5
6:      3    1288    3.0
```

Let's create train and test sets

```
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)
test_index <- createDataPartition(y = edx_cf_clean$rating, times = 1, p = 0.1, list = FALSE)
trainset <- edx_cf_clean[-test_index,]
testset <- edx_cf_clean[test_index,]
```

Write trainset and testset tables on disk

```
write.table(trainset , file = "trainset.txt" , sep = " " , row.names = FALSE, col.names = FALSE)
write.table(testset, file = "testset.txt" , sep = " " , row.names = FALSE, col.names = FALSE)
train_set <- data_file("trainset.txt")
test_set <- data_file("testset.txt")
class(train_set)
```

```
[1] "DataSource"
attr(,"package")
[1] "recosystem"
```

Create recommender

```
set.seed(2021, sample.kind="Rounding")
r = Reco()
class(r)
```

```
[1] "RecoSys"
attr(,"package")
[1] "recosystem"
```

We will use the tune() method to tune the model parameters. This method uses cross validation to tune the model parameters.These parameters are:

- dim: the number of latent factors. I have used the 3 values given as example on the tune() method page (https://www.rdocumentation.org/packages/recosystem/versions/0.3/topics/tune)
- lrate: the learning rate, which can be thought of as the step size in gradient descent. As for previous one, I've used the 3 values given as example on the tune() method page on the web.
- nthread: integer, the number of threads for parallel computing. Default is 1 and I have kept it as that.
- niter: integer, the number of iterations. Default is 20. I have used 20 for this case as well as the for the final use of recosystem on the validation set. The output of this method is a list with two components:
- min: parameter values with minimum cross validation RMSE. This is a list that can be passed to the opts argument in $train() and we will do that in the call to the train() method.
- res: a data frame giving the supplied candidate values of tuning parameters, and one column showing the RMSE associated with each combination.

```
opts = r$tune(train_set, opts = list(dim = c(10, 20, 30), lrate = c(0.05, 0.1, 0.2),
                                     nthread = 1, niter = 10))
```

```
0%   10   20   30   40   50   60   70   80   90   100%
[----|----|----|----|----|----|----|----|----|----|
**************************************************|
```

We can have a look at the parameters providing the minimum RMSE value with:

```
opts$min
```

```
$dim
[1] 10

$costp_l1
[1] 0.1

$costp_l2
[1] 0.01

$costq_l1
[1] 0

$costq_l2
[1] 0.01

$lrate
[1] 0.05

$loss_fun
[1] 0.9828133
```

Let's train our model. The train() method trains a recommender model. It will read a training data file and create a model file at the specified locations. The model file contains necessary information for prediction.

```
r$train(train_set, opts = c(opts$min, nthread = 1, niter = 20))
```

```
iter      tr_rmse          obj
   0       1.5883   2.4072e+06
   1       1.0081   1.1888e+06
   2       0.9370   1.0765e+06
   3       0.9053   1.0285e+06
   4       0.8870   1.0010e+06
   5       0.8747   9.8261e+05
   6       0.8657   9.6888e+05
   7       0.8586   9.5825e+05
   8       0.8526   9.4921e+05
   9       0.8474   9.4115e+05
  10       0.8430   9.3439e+05
  11       0.8388   9.2818e+05
  12       0.8347   9.2208e+05
```

```
  13       0.8309   9.1636e+05
  14       0.8272   9.1081e+05
  15       0.8236   9.0562e+05
  16       0.8199   9.0029e+05
  17       0.8164   8.9541e+05
  18       0.8129   8.9036e+05
  19       0.8095   8.8565e+05
```

Let's generate the predictions and write them to a file on disk.

```
pred_file <- tempfile()
r$predict(test_set, out_file(pred_file))
```

```
prediction output generated at /var/folders/ch/j_33ttj55gj7rx7cy03_z6gm0000gn/T//Rtmp9K9T26/file6752707
```

We can read the first 10 predicted values from the file

```
print(scan(pred_file, n = 10))
```

```
[1] 3.25282 2.85731 4.75491 3.23846 3.51303 2.96160 3.02487 2.70690 4.83498 3.32530
```

Let's read the predictions from file.

```
predicted_ratings_reco <- scan(pred_file)
```

```
Read 90002 items
```

```
class(predicted_ratings_reco)
```

```
[1] "numeric"
```

```
str(predicted_ratings_reco)
```

```
num [1:90002] 3.25 2.86 4.75 3.24 3.51 ...
```

Let's calculate the RMSE

```
rmse_reco <- RMSE(predicted_ratings_reco,testset$rating)
rmse_reco
```

```
[1] 0.9258568
```

```
errors <- bind_rows(errors,
                    tibble(algorithm = "Recosystem", method = "    ", RMSE = 0.9258568))
errors %>% knitr::kable()
```

```
|algorithm  |method    |      RMSE|
|:----------|:---------|---------:|
|REG BIAS   |          | 0.8623166|
|POPULAR    |split     | 0.9410653|
|UBCF       |cross-val | 1.1891657|
|IBCF       |split     | 1.3445680|
|SVD        |split     | 1.0433052|
|SVDF       |split     | 0.9756173|
|Recosystem |          | 0.9258568|
```

2.2.10 Regularized bias effect of movie and user

To be able to compare the results of the CF models with the model of regularized biases of predictors, they should all be using the same data. The CF models did make use only of movie, user and rating columns from the edx_cf dataset, which is a subset of the edx one used on the bias model. Let's apply the bias model on the edx_cf dataset and only taking into account information on movie, user and rating.

Let's split the edx_cf into train and test portions.

```
set.seed(2021, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
cf_test <- createDataPartition(y = edx_cf$rating, times = 1, p = 0.1, list = FALSE)
edx_cf_train <- edx_cf[-cf_test,]
temp <- edx_cf[cf_test,]
edx_cf_test <- temp %>%
  semi_join(edx_cf_train, by = "movieId") %>%
  semi_join(edx_cf_train, by = "userId")
removed <- anti_join(temp, edx_cf_test)
nrow(removed)
edx <- rbind(edx_cf_train, removed)
nrow(edx_cf_train)
```

```
[1] 810005
```

```
nrow(edx_cf_test)
```

```
[1] 89279
```

Let's calculate the mean for our edx_cf set.

```
mu <- mean(edx_cf_train$rating)
mu
```

```
[1] 3.512986
```

```
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){

  b_i <- edx_cf_train %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))
```

```
  b_u <- edx_cf_train %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))

  predicted_ratings <-
    edx_cf_train %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
    pull(pred)

  return(RMSE(predicted_ratings, edx_cf_train$rating))
})
```

Plotting lambdas versus rmses:

```
qplot(lambdas, rmses)
```

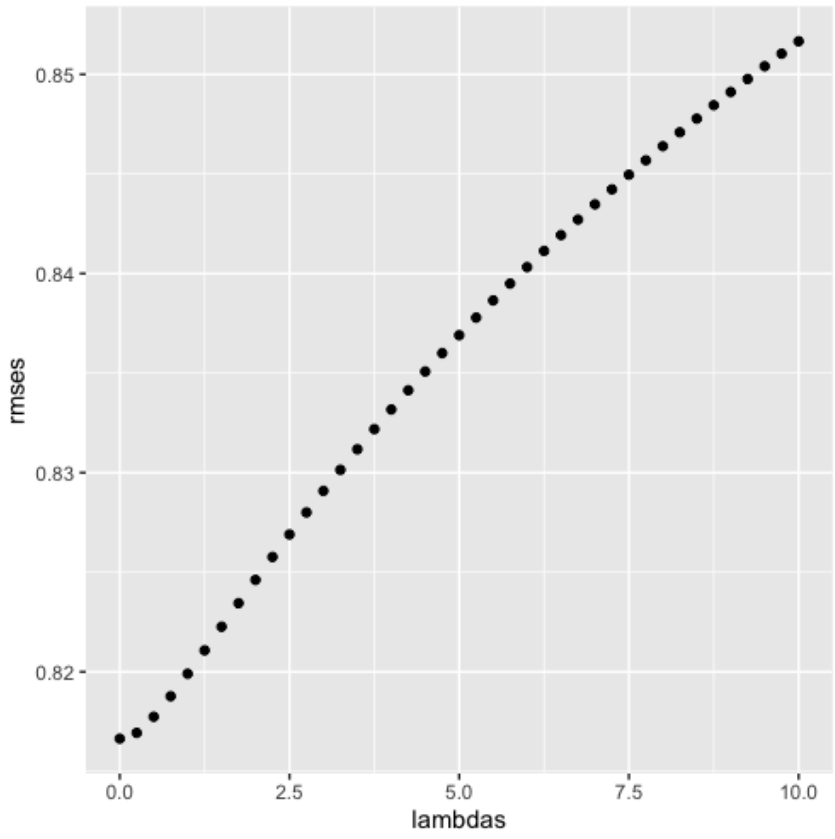

Figure 25: RMSEs versus lambdas

The minimum value of the RMSE obtained using the train set is:

```
min(rmses)
```

```
[1] 0.8166448
```

The lambda that minimizes the RMSE is:

```
lambda <- lambdas[which.min(rmses)]
lambda
```

```
[1] 0
```

Compute regularized estimates of b_i using lambda

```
movie_avgs_reg <- edx_cf_train %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n()+lambda), n_i = n())
```

Compute regularized estimates of b_u using lambda

```
user_avgs_reg <- edx_cf_train %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = sum(rating - mu - b_i)/(n()+lambda), n_u = n())
```

Predict ratings for our test set:

```
predicted_ratings_cf <- edx_cf_test %>%
  left_join(movie_avgs_reg, by='movieId') %>%
  left_join(user_avgs_reg, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  .$pred
```

Calculate RMSE against validation set with generated prediction:

```
rmse_cf <- RMSE(edx_cf_test$rating,predicted_ratings_cf)
rmse_cf
```

```
[1] 0.905792
```

As we can see, the RMSE is higher than when using full edx set and all the predictors, but still better than any of our CF models.

```
errors <- bind_rows(errors,
                    tibble(algorithm = "REG BIAS CF", method = "    ", RMSE = 0.905792))
errors %>% knitr::kable()
```

```
|algorithm    |method    |      RMSE|
|:-----------|:---------|---------:|
```

```
|REG BIAS    |          | 0.8623166|
|POPULAR     |split     | 0.9410653|
|UBCF        |cross-val | 1.1891657|
|IBCF        |split     | 1.3445680|
|SVD         |split     | 1.0433052|
|SVDF        |split     | 0.9756173|
|Recosystem  |          | 0.9258568|
|REG BIAS CF |          | 0.9057920|
```

2.3 Results

As we can see, the best RMSE result of all the CF models is achieved by the Recosystem one. We have already proved the model that takes into account all the regularized bias effect of all the predictors is capable of taking the RMSE below the RMSE threshold of 0.86490 and even that type of model operating on the same dataset (edx_cf) as the CF models gives an RMSE (0.905792) better than the CF models. So, let's see if the best of the CF models, the recosystem, is capable of outperforming the bias model result on the full edx and validation datasets.

```
nrow(edx)
```

```
[1] 9000055
```

```
nrow(validation)
```

```
[1] 999999
```

Let's get the matrix for both edx and validation datasets

```
train_for_mf <- edx %>%
  select(userId, movieId, rating)
train_for_mf <- as.matrix(train_for_mf)
val_for_mf <- validation %>%
  select(userId, movieId, rating)
val_for_mf <- as.matrix(val_for_mf)
```

Write filtered train and validation sets to disk

```
write.table(train_for_mf , file = "trainset.txt" , sep = " " , row.names = FALSE, col.names = FALSE)
write.table(val_for_mf, file = "valset.txt" , sep = " " , row.names = FALSE, col.names = FALSE)
```

Create data sets from written files in correct format for recosystem

```
train_set <- data_file("trainset.txt")
val_set <- data_file("valset.txt")
class(train_set)
```

```
[1] "DataSource"
attr(,"package")
[1] "recosystem"
```

Use the recosystem library to perform the matrix factorization First, let's build the recommender

```
set.seed(2021, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
r <-Reco()
```

Let's tune the training set

```
opts <- r$tune(train_set, opts = list(dim = c(10, 20, 30), lrate = c(0.1, 0.2),
                                      costp_l1 = 0, costq_l1 = 0,
                                      nthread = 1, niter = 10))
```

```
0%   10   20   30   40   50   60   70   80   90   100%
[----|----|----|----|----|----|----|----|----|----|
**************************************************|
```

Let's train the model

```
r$train(train_set, opts = c(opts$min, nthread = 1, niter = 20))
```

```
iter    tr_rmse          obj
0        0.9725   1.2027e+07
1        0.8723   9.8761e+06
2        0.8384   9.1752e+06
3        0.8157   8.7387e+06
4        0.8002   8.4624e+06
5        0.7886   8.2648e+06
6        0.7793   8.1159e+06
7        0.7715   7.9993e+06
8        0.7649   7.9044e+06
9        0.7593   7.8278e+06
10       0.7542   7.7631e+06
11       0.7496   7.7023e+06
12       0.7456   7.6558e+06
13       0.7419   7.6116e+06
14       0.7385   7.5734e+06
15       0.7354   7.5394e+06
16       0.7325   7.5085e+06
17       0.7299   7.4791e+06
18       0.7274   7.4542e+06
19       0.7251   7.4329e+06
```

Make predictions on the validation set

```
pred_file <- tempfile()
r$predict(val_set, out_file(pred_file))
```

```
prediction output generated at /var/folders/ch/j_33ttj55gj7rx7cy03_z6gm0000gn/T//RtmpkjoQgM/file23cf54e
```

Let's read predictions file

```
predicted_mf <- scan(pred_file)
```

```
Read 999999 items
```

Let's see what is our predictions object

```
str(predicted_mf)
```

```
num [1:999999] 4.12 5.09 4.77 3.44 4.49 ...
```

Let's check our validation object

```
str(validation$rating)
```

```
num [1:999999] 5 5 5 3 2 3 3.5 4.5 5 3 ...
```

As seen above, both are of the same type and length. Let's calculate the RMSE:

```
rmse_mf <- RMSE(predicted_mf,validation$rating)
rmse_mf
```

```
[1] 0.7829094
```

2.4 Conclusions

Using the recosystem collaborative filtering model, that uses matrix factorization, the final RMSE achieved on the validation set has been 0.7829094, which goes below the lowest threshold set in the project (0.86490) and improves quite a bit the outcome (0.8623166) of the model of regularized bias for all predictors created in the first model of this document.

I was expecting better results from the UBCF and IBCF models, in particular the UBCF, as all the readings I found about CF models was putting it as the best one producing results, but their results were worse than the POPULAR, SVD and SVDF models.

Computing resources (MacBook Pro with 8GB of RAM) have been a bottle neck in this project when working with CF models, like UBCF and IBCF, using the edx dataset. Generating evaluation schemas caused the R session to hung and the laptop to run out of memory. That was the main reason to generate the edx_cf dataset (900K rows) as a 10% of the edx one (9M records), although I had already verified all CF models run OK with the 1M Movielens dataset (see Appendix A for its link). Even with that, R session did hung several times when training some of the CF models and required to rerun the step again after restarting R.

3 Appendix A - Bibliography

- MovieLens
- MovieLens 10M Dataset
- MovieLens 1M Dataset
- recommenderlab: A Framework for Developing and Testing Recommendation Algorithms
- Recommender: Create a Recommender Model
- Recommender System — Matrix Factorization
- RecommenderLab Tutorial
- recommenderlab: Lab for Developing and Testing Recommender Algorithms
- Package 'recommenderlab'
- Sistemas de recomendación en R (Recommendation systems in R)
- Movie Recommendation System
- Predicting Ratings with Matrix Factorization Methods
- Recosystem - Recommender System using Matrix Factorization