MovieLens Project

Felipe Miranda

03/05/2022

1 Introduction

This report presents the analysis of *MovieLens* data set and has the main objective of **evaluating the performance of machine learning techniques to predict the** *movie ratings* **by using its contents**. This work is divided in the following sections:

Resources, Methods and Analysis: this section presents the fundamental resources, methods and analysis used to retrieve useful data in MovieLens;

Prediction models and results: this section presents the machine learning models used to predict the **movie ratings** and their respective results;

Concluding remarks: this section presents further discussion about all results and approaches used in this work.

The complete code used in this analysis is available on *GitHub*: https://github.com/fm-33/movielens.

2 Resources, Methods and Analysis

In this section, *MovieLens* data set, which is the central subject of this work, is presented together with the methodology employed to process and analyze its main contents.

2.1 MovieLens

About what *MovieLens* is, its webpage (https://movielens.org/info/about) gives a good introduction about it:

"*MovieLens is a research site run by GroupLens Research at the University of Minnesota. MovieLens uses 'collaborative filtering' technology to make recommendations of movies that you might enjoy, and to help you avoid the ones that you won't. Based on your movie ratings, MovieLens generates personalized predictions for movies you haven't seen yet. MovieLens is a unique research vehicle for dozens of undergraduates and graduate students researching various aspects of personalization and filtering technologies.*"

MovieLens 10M, which is the file used in this work, is a smaller data set of 63 MB. It contains about 10 million ratings and 100,000 tags applied to 10,000 movies by 72,000 users. *MovieLens* was released in 2009.

2.2 Libraries

In order to better manipulate, process and present the data in this work, the following packages had to be imported in the *R* script. Their use is described in the comments above their import statements.

```
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")
# Package "tidyverse" has many uses when manipulating data
library(tidyverse)
# Package "caret" has many uses when working with machine learning
library(caret)
# Package "data.table" has many uses when working with tables
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")
library(data.table)
# Package "ggplot2" is used for data visualization
if(!require(ggplot2)) install.packages("ggplot2", repos = "http://cran.us.r-project.org")
library(ggplot2)
# Package "Rcpp" (required when testing the script in other computers)
if(!require(Rcpp)) install.packages("Rcpp", repos = "http://cran.us.r-project.org")
library(Rcpp)
# Package "lubridate" is used with the timestamp in order to handle date-time
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
library(lubridate)
# Package "recosystem" is a Recommender System used on the analysis
if(!require(recosystem)) install.packages("recosystem", repos = "http://cran.us.r-project.org")
library(recosystem)
```

2.3 Data loading, test and validation data frames

The following script is used to download *MovieLens 10M* and to create the *test* and *validation* data frames, called, respectively, *edx* and *validation.* It is important to note that the data sets are only a portion of the whole data set, *i.e.*, they are small-scale samples of *MovieLens 10M.*

```
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)
```

2.4 *edx* data frame

The following subsections present the data retrieved from *edx* data frame along with the commented *R* script.

2.4.1 General Overview

The following results presented in this subsection show general data stored in *edx* data frame. A brief discussion about the analysis and the results is presented on the comments of each code chunk.

```
# As function sumary() returns no useful information about edx data frame,
# it was manually processed.
# edx has 10677 movies.
length(unique(edx$movieId))
```

```
## [1] 10677
```

```
# edx has 797 unique genres.
length(unique(edx$genres))
```

```
## [1] 797
```

```
# As the number of genres was quite large, I decided to check it out.
# It turned out that the genres can also be the combination of more than one.
head(unique(edx$genres),30)
```

```
##  [1] "Comedy|Romance"
##  [2] "Action|Crime|Thriller"
##  [3] "Action|Drama|Sci-Fi|Thriller"
##  [4] "Action|Adventure|Sci-Fi"
##  [5] "Action|Adventure|Drama|Sci-Fi"
##  [6] "Children|Comedy|Fantasy"
##  [7] "Comedy|Drama|Romance|War"
##  [8] "Adventure|Children|Romance"
##  [9] "Adventure|Animation|Children|Drama|Musical"
## [10] "Action|Comedy"
## [11] "Action|Romance|Thriller"
## [12] "Action|Comedy|Crime|Thriller"
## [13] "Action|Comedy|War"
## [14] "Comedy"
## [15] "Comedy|Drama|Romance"
## [16] "Adventure|Animation|Children|Comedy|Musical"
## [17] "Action|Sci-Fi"
## [18] "Animation|Children|Drama|Fantasy|Musical"
## [19] "Animation|Children"
## [20] "Action|Drama|War"
## [21] "Action|Thriller"
## [22] "Adventure|Drama|Western"
## [23] "Action|Adventure|Mystery|Thriller"
## [24] "Action|Adventure|Thriller"
## [25] "Action|Adventure|Romance|Thriller"
## [26] "Action|Adventure|Sci-Fi|War"
## [27] "Action|Drama|Thriller"
## [28] "Drama|Romance"
## [29] "Action|Adventure"
## [30] "Children|Comedy|Fantasy|Musical"
```

```r
# edx has 69878 unique userIds, which means that 69878 users rated the movies.
length(unique(edx$userId))
```

```
## [1] 69878
```

```r
# The highest is 5.
max(edx$rating)
```

```
## [1] 5
```

```r
# The rating "5" was given 1390114 times.
sum (edx$rating==5)
```

```
## [1] 1390114
```

```r
# The lowest is 0.5
min(edx$rating)
```

```
## [1] 0.5
```

```r
# The rating "0.5" was given 85374 times.
sum (edx$rating==0.5)
```

```
## [1] 85374
```

```r
# The average rating is 3.512465.
mean(edx$rating)
```

```
## [1] 3.512465
```

2.4.2 Genres Overview

The following results presented in this subsection addresses specific data stored in *edx* data frame regarding *genres*. A brief discussion about the analysis and the results is presented on the comments of each code chunk. One interesting point that is important to note is that, even if some genres have more releases, the average rating of all genres are quite similar, being around the average of the whole data frame: **3.51** (from **0** to **5**).

```
# 5336 movies marked with "Drama" genre.
drama <- edx %>% filter(str_detect(genres,"Drama"))
length(unique(drama$movieId))
```

```
## [1] 5336
```

```
# The highest rating is 5.
max(drama$rating)
```

```
## [1] 5
```

```
# The rating 5 was given 711447 times.
sum(drama$rating==5)
```

```
## [1] 711447
```

```
# The lowest rating is 0.5
min(drama$rating)
```

```
## [1] 0.5
```

```
# The rating 5 was given 23282 times.
sum(drama$rating==0.5)
```

```
## [1] 23282
```

```
# The average rate is 3.673131.
mean(drama$rating)
```

```
## [1] 3.673131
```

```
# 3703 movies marked with "Comedy" genre.
comedy <- edx %>% filter(str_detect(genres,"Comedy"))
length(unique(comedy$movieId))
```

```
## [1] 3703
```

```
# The highest is 5.
max(comedy$rating)
```

```
## [1] 5
```

```r
# The rating 5 was given 492512 times.
sum(comedy$rating==5)
```

```
## [1] 492512
```

```r
# The lowest is 0.5
min(comedy$rating)
```

```
## [1] 0.5
```

```r
# The rating 5 was given 38345 times.
sum(comedy$rating==0.5)
```

```
## [1] 38345
```

```r
# The average rate is 3.436908.
mean(comedy$rating)
```

```
## [1] 3.436908
```

```r
# 1705 movies marked with "Thriller" genre.
thriller <- edx %>% filter(str_detect(genres,"Thriller"))
length(unique(thriller$movieId))
```

```
## [1] 1705
```

```r
# The highest is 5.
max(thriller$rating)
```

```
## [1] 5
```

```r
# The rating 5 was given 332142 times.
sum(thriller$rating==5)
```

```
## [1] 332142
```

```r
# The lowest is 0.5
min(thriller$rating)
```

```
## [1] 0.5
```

```r
# The rating 5 was given 19250 times.
sum(thriller$rating==0.5)
```

```
## [1] 19250
```

```r
# The average rate is 3.507676.
mean(thriller$rating)
```

```
## [1] 3.507676
```

```r
# 1685 movies marked with "Romance" genre.
romance <- edx %>% filter(str_detect(genres,"Romance"))
length(unique(romance$movieId))
```

```
## [1] 1685
```

```r
# The highest is 5.
max(romance$rating)
```

```
## [1] 5
```

```r
# The rating 5 was given 274910 times.
sum(romance$rating==5)
```

```
## [1] 274910
```

```r
# The lowest is 0.5
min(romance$rating)
```

```
## [1] 0.5
```

```r
# The rating 5 was given 13533 times.
sum(romance$rating==0.5)
```

```
## [1] 13533
```

```r
# The average rate is 3.553813.
mean(romance$rating)
```

```
## [1] 3.553813
```

```r
# 1473 movies marked with "Action" genre.
action <- edx %>% filter(str_detect(genres,"Action"))
length(unique(action$movieId))
```

```
## [1] 1473
```

```r
# The highest is 5.
max(action$rating)
```

```
## [1] 5
```

```
# The rating 5 was given 340607 times.
sum(action$rating==5)
```

```
## [1] 340607
```

```
# The lowest  is 0.5
min(action$rating)
```

```
## [1] 0.5
```

```
# The rating 5 was given 27453 times.
sum(action$rating==0.5)
```

```
## [1] 27453
```

```
# The average rate is 3.421405
mean(action$rating)
```

```
## [1] 3.421405
```

```
# 1025 movies marked with "Action" genre.
adventure <- edx %>% filter(str_detect(genres,"Adventure"))
length(unique(adventure$movieId))
```

```
## [1] 1025
```

```
# The highest is 5.
max(adventure$rating)
```

```
## [1] 5
```

```
# The rating 5 was given 281465 times.
sum(adventure$rating==5)
```

```
## [1] 281465
```

```
# The lowest is 0.5
min(adventure$rating)
```

```
## [1] 0.5
```

```
# The rating 5 was given 23282 times.
sum(adventure$rating==0.5)
```

```
## [1] 18776
```

```
# The average rate is 3.493544
mean(adventure$rating)
```

```
## [1] 3.493544
```

2.4.3 Ratings, users and timestamp

In the histogram shown in Fig. 1, we can observe that the ratings in *edx* are predominantly formed by *integer numbers*, indicating that the users tend to give ratings like **2**, **3** or **4** rather than **2.5** or **3.5**. It is important to remember that the average rating of all movies in *edx* is **3.512465**.

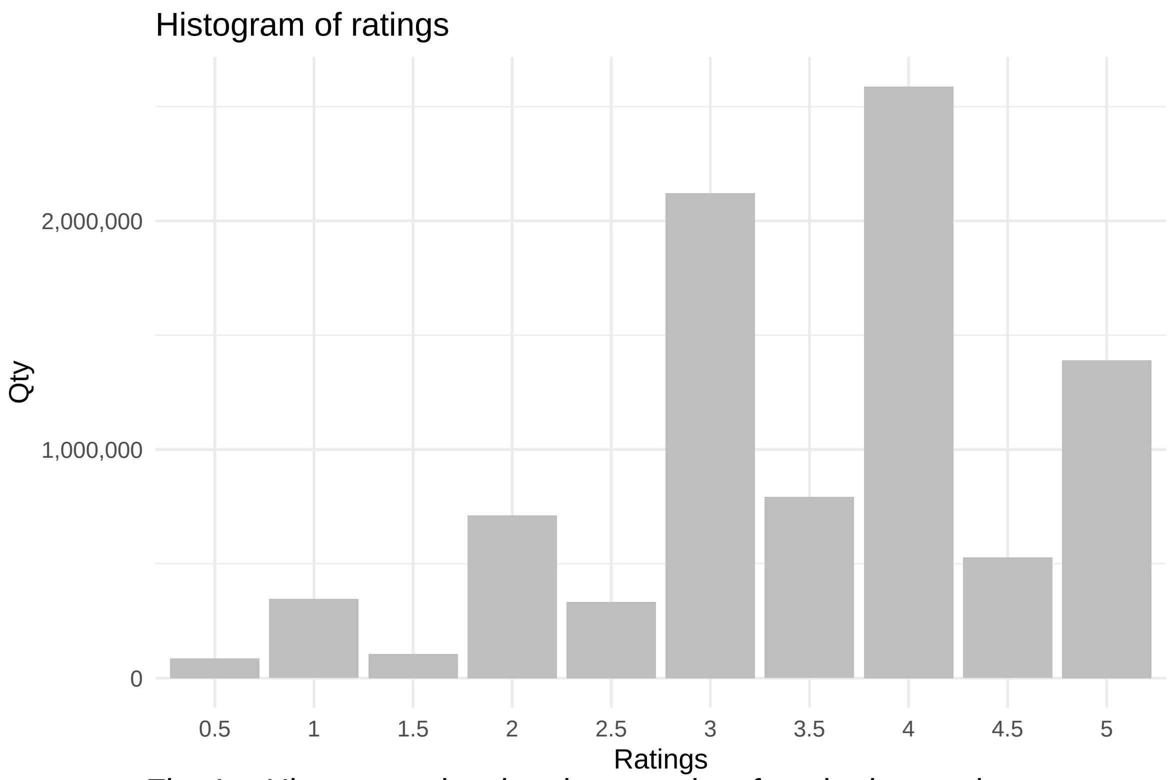

Fig. 1 − Histogram showing the quantity of each given rating.

In the histogram shown in Fig. 2 and in the density plot shown in Fig. 3, we can observe that most users have less than **100** ratings given. It is important to note that, besides being a bit counterintuitive, the average of 128.79 ratings per user is explained by some users who gave more than **1000** ratings, as show in the following tables.

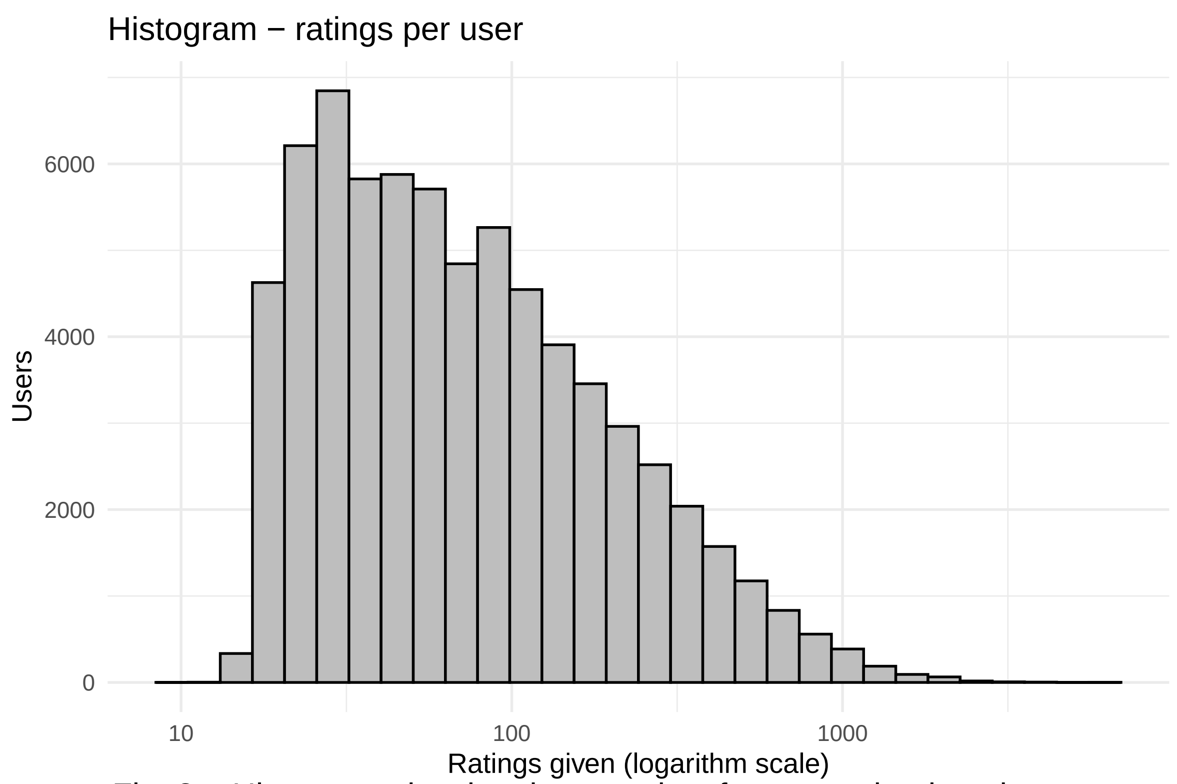

Fig. 2 − Histogram showing the quantity of users and rating given.

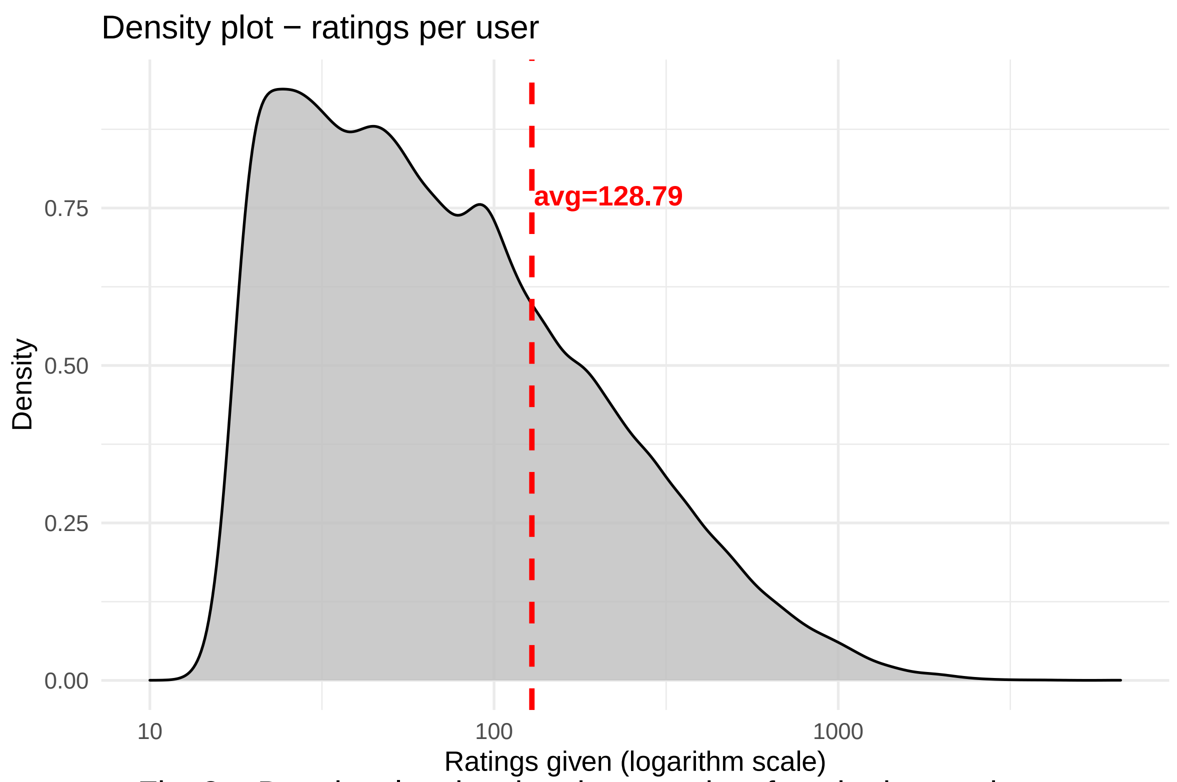

Fig. 3 − Density plot showing the quantity of each given rating.

The following tables show the users with higher and lower number of ratings given. It can be observed that both groups are very far from the average of **128.79** ratings per user.

```r
# Creating a summary table grouped by userId
ratings_qty <- edx %>% group_by(rating) %>%
  summarize(count = n())
# Creating a summary table grouped by userId
ratings_per_user <- edx %>% group_by(userId) %>%
  summarize(ratings_per_user_qty = n(),
            ratings_given_mean = mean(rating),
            mu_user = mean(rating),
            sd_user = sd(rating),)
```

```r
# Average of 128.7967 ratings given per user
mean(ratings_per_user$ratings_per_user_qty)
```

```
## [1] 128.7967
```

```r
# Sorting (descending) and showing the users according to their number of rating
ratings_per_user_descending <- ratings_per_user %>% arrange(desc(ratings_per_user_qty))
print(head(ratings_per_user_descending,10))
```

```
## # A tibble: 10 x 5
##    userId ratings_per_user_qty ratings_given_mean mu_user sd_user
##     <int>                <int>              <dbl>   <dbl>   <dbl>
##  1  59269                 6616               3.26    3.26   0.639
##  2  67385                 6360               3.20    3.20   0.957
##  3  14463                 4648               2.40    2.40   0.688
##  4  68259                 4036               3.58    3.58   1.05
##  5  27468                 4023               3.83    3.83   0.734
##  6  19635                 3771               3.50    3.50   0.778
##  7   3817                 3733               3.11    3.11   0.579
##  8  63134                 3371               3.27    3.27   0.957
##  9  58357                 3361               3.00    3.00   0.798
## 10  27584                 3142               3.00    3.00   0.719
```

```r
# Sorting (ascending) and showing the users according to their number of rating
ratings_per_user_ascending <- ratings_per_user %>% arrange(ratings_per_user_qty)
print(head(ratings_per_user_ascending,10) )
```

```
## # A tibble: 10 x 5
##    userId ratings_per_user_qty ratings_given_mean mu_user sd_user
##     <int>                <int>              <dbl>   <dbl>   <dbl>
##  1  62516                   10               2.25    2.25   1.14
##  2  22170                   12               4       4      0.739
##  3  15719                   13               3.77    3.77   1.24
##  4  50608                   13               3.92    3.92   1.44
##  5    901                   14               4.71    4.71   0.469
##  6   1833                   14               3       3      1.24
##  7   2476                   14               2.93    2.93   1.33
##  8   5214                   14               1.79    1.79   1.22
##  9   9689                   14               3.57    3.57   1.22
## 10  10364                   14               4.32    4.32   1.27
```

The histogram shown in Fig. 4 presents the number of ratings given per year. As the number of ratings in 1995 is quite low when compared to the following years, it was necessary to check this year individually. It is important to note that the *timestamp* in *edx* is encoded using *Unix Timestamp* < https://www.unixtimestamp.com/>, which is a specific way to *track time as a running total of seconds*, starting at the Unix Epoch on January 1st, 1970 at UTC.

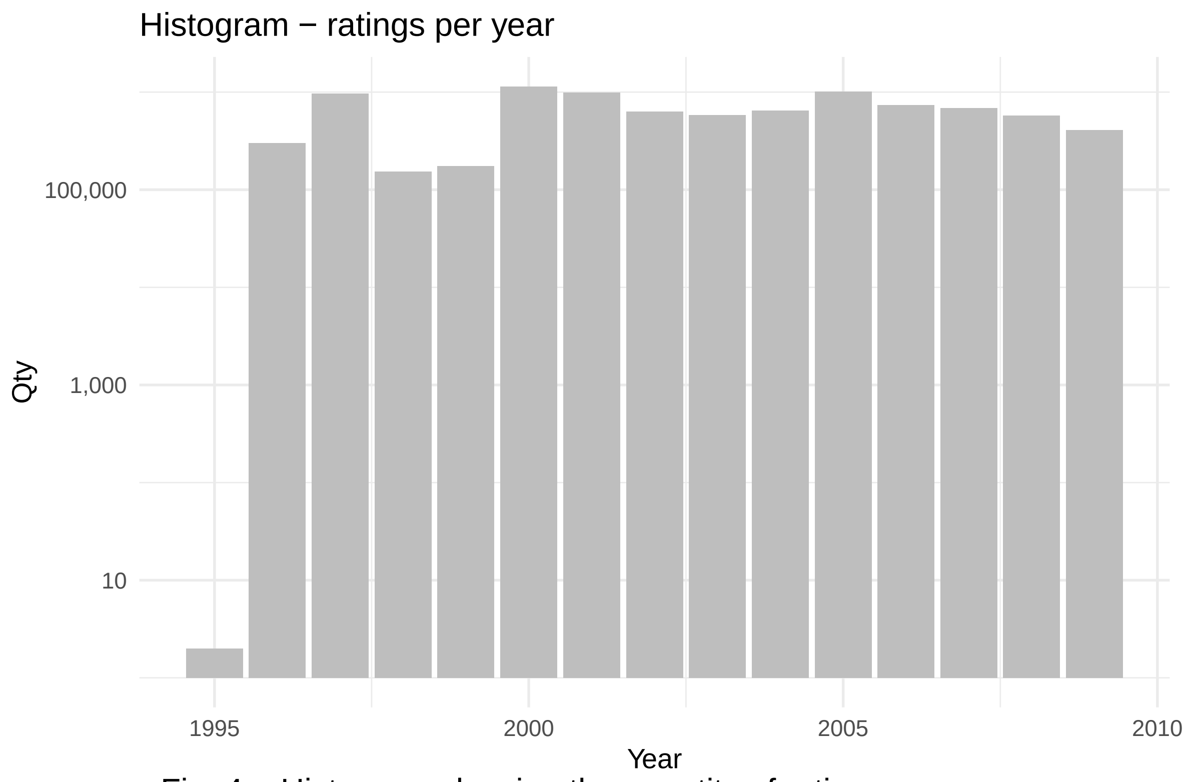

Fig. 4 − Histogram showing the quantity of ratings per year.

```
# 2 ratings given in 1995 (timestamp of January first 1996)
sum(edx$timestamp <= 820522677)
```

```
## [1] 2
```

3 Prediction models and results

In this section, it is presented the machine learning and prediction tools and techniques along with their respective results. The accuracy of the results are evaluated using the *root-mean-square error* (RMSE), which is already implemented in *caret* package. In this work, the intended RMSE of the predicted results has to be less than **0.86490** in order to achieve its objective.

The following predictions were performed with both the test data frames (*training_dataframe* and *test_dataframe*) and the validation data frames (*edx* and *validation*). The following code chunk shows how *training_dataframe* and *test_dataframe* were created for the aforementioned scenarios:

```
################################################################################
# TRAINING AND TEST DATA FRAMES
################################################################################
# Creating training and test data frames
```

```r
set.seed(1, 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)
training_dataframe <- edx[-test_index,]
temp <- edx[test_index,]
# Make sure userId and movieId in training_dataframe are also in test_dataframe
test_dataframe <- temp %>%
  semi_join(training_dataframe, by = "movieId") %>%
  semi_join(training_dataframe, by = "userId")
# Add rows removed from test_dataframe set back into training_dataframe
# Adding back rows into train set
removed <- anti_join(temp, test_dataframe)
```

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```r
training_dataframe <- rbind(training_dataframe, removed)
rm(test_index, temp, removed)
```

3.1 Linear models

Linear models are some of the most common and simpler tools used to make all sorts of numerical predictions. In *R*, function *lm()* is implemented to fit linear models using one or more predictors, reducing the work needed to create linear models. In this work, this first approach is used to test if there is any **linear correlation** between the predictors and the *ratings* present in the data frames.

```r
# lm() function using timestamp as a predictor.
model <- lm(formula=rating ~ timestamp, data = training_dataframe)
prediction_lm_test_ts <- predict(model, newdata = test_dataframe)
rmse_lm_test_ts <- RMSE(prediction_lm_test_ts, test_dataframe$rating)
model <- lm(formula=rating ~ timestamp, data = edx)
prediction_validation_ts <- predict(model, newdata = validation)
rmse_lm_validation_ts <- RMSE(prediction_validation_ts, validation$rating)
# lm() function using timestamp and userId as predictors.
model <- lm(formula=rating ~ timestamp+userId, data = training_dataframe)
prediction_lm_test_ts_ui <- predict(model, newdata = test_dataframe)
rmse_lm_test_ts_ui <- RMSE(prediction_lm_test_ts_ui, test_dataframe$rating)
model <- lm(formula=rating ~ timestamp+userId, data = edx)
prediction_lm_validation_ts_ui <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui <- RMSE(prediction_lm_validation_ts_ui, validation$rating)
# lm() function using timestamp, userId and movieId as predictors.
model <- lm(formula=rating ~ timestamp+userId+movieId, data = training_dataframe)
prediction_lm_test_ts_ui_mi <- predict(model, newdata = test_dataframe)
rmse_lm_test_ts_ui_mi <- RMSE(prediction_lm_test_ts_ui_mi, test_dataframe$rating)
model <- lm(formula=rating ~ timestamp+userId+movieId, data = edx)
prediction_lm_validation_ts_ui_mi <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui_mi <- RMSE(prediction_lm_validation_ts_ui_mi, validation$rating)

# Impossible to compute (> 48 Gb vector)
# model <- lm(formula=rating ~ timestamp+userId+movieId+genres, data = training_dataframe)
# prediction_test_ts <- predict(model, newdata = test_dataframe)
# RMSE(prediction_test_ts, test_dataframe$rating)
# Impossible to compute (> 48 Gb vector)
# model <- lm(formula=rating ~ genres, data = training_dataframe)
# prediction_test_ts <- predict(model, newdata = test_dataframe)
# RMSE(prediction_test_ts, test_dataframe$rating)

results_lm <- tibble(predictors = "timestamp", RMSE = rmse_lm_test_ts, type = "test")
results_lm <- bind_rows(results_lm,
                        tibble(predictors = "timestamp", RMSE = rmse_lm_validation_ts,
                               type = "validation"))
results_lm <- bind_rows(results_lm,
                        tibble(predictors = "timestamp and userId",
                               RMSE = rmse_lm_test_ts_ui,
                               type = "test"))
results_lm <- bind_rows(results_lm,
                        tibble(predictors = "timestamp and userId",
                               RMSE = rmse_lm_validation_ts_ui,
                               type = "validation"))
results_lm <- bind_rows(results_lm,
                        tibble(predictors = "timestamp, userId and movieId",
                               RMSE = rmse_lm_test_ts_ui_mi,
                               type = "test"))
results_lm <- bind_rows(results_lm,
                        tibble(predictors = "timestamp, userId and movieId",
                               RMSE = rmse_lm_validation_ts_ui_mi,
                               type = "validation"))
results_lm %>% knitr::kable()
```

predictors	RMSE	type
timestamp	1.059432	test
timestamp	1.060557	validation
timestamp and userId	1.059424	test
timestamp and userId	1.060551	validation
timestamp, userId and movieId	1.059393	test
timestamp, userId and movieId	1.060522	validation

As can be noted on the results, the results were all quite poor, in both test and validation scenarios, independently from the predictors used. The results indicate that there is no linear correlation between the variables used as predictors, *i.e.*, *timestamp*, *userId*, and *movieId*, and *ratings*. One interesting outcome of the use of function *lm()* was that more than 3 predictors resulted in a very large vector, having more than 48 Gb, which was could not be processed by a common personal computer.

3.2 Incremental approach

In this approach, the predictions were calculated using *mean*, *user*, and *movie* effect/bias. The analysis were made incrementing the number of predictors, checking the resulting RMSE of each model and scenario.

3.2.1 Only the mean

The first step was made by calculating the mean of the training data frames of both scenarios, *i.e.*, *training_dataframe* for the tests and *edx* for the validation.

```
# Calculating the mean of the training data frame
mean_rating_test <- mean(training_dataframe$rating)
# RMSE between mean_rating_test and the ratings in the test data frame
rmse_mean_test <- RMSE(test_dataframe$rating, mean_rating_test)
rmse_mean_test
```

```
## [1] 1.060054
```

```
# Calculating the mean of the edx data frame
mean_rating_validation <- mean(edx$rating)
# RMSE between mean_rating_validation and the ratings in "validation" data frame
rmse_mean_validation  <- RMSE(validation$rating, mean_rating_validation)
rmse_mean_validation
```

```
## [1] 1.061202
```

The results were poor, similar to the numbers given by the use of *linear model*, but it is important to note that the mean was intended to use together with other increments in this approach.

3.2.2 Mean and movie effect/bias

The next step on this approach is considering the effect/bias of the **movies** (column *movieId*) together with the mean. No special pre-processing or filtering was made in order to calculating the predictions in any scenario.

```
movie_effect_test <- training_dataframe %>%  group_by(movieId) %>%
  summarize(movie_effect = mean(rating - mean_rating_test))
prediction_movie_test <- mean_rating_test + test_dataframe %>%
  left_join(movie_effect_test, by = "movieId") %>%
  pull(movie_effect)
rmse_mean_movie_test <- RMSE(prediction_movie_test, test_dataframe$rating)
rmse_mean_movie_test
```

```
## [1] 0.9429615
```

```
movie_effect_validation <- edx %>%  group_by(movieId) %>%
  summarize(movie_effect = mean(rating - mean_rating_validation))
prediction_movie_validation <- mean_rating_validation + validation %>%
  left_join(movie_effect_validation, by = "movieId") %>%  pull(movie_effect)
rmse_mean_movie_validation <- RMSE(prediction_movie_validation, validation$rating)
rmse_mean_movie_validation
```

```
## [1] 0.9439087
```

The results presented a sensible improvement when compared to just the **mean**. In both scenarios (test and validation) the RMSE was lowered from **1.06** to **0.94**. This improvement indicates that the field *movieId* (which has a unique code to each movie) can be used to better predict the ratings.

3.2.3 Mean, movie and user effect/bias

The final step on this approach is considering the effect/bias of the **users** (column *userId*). Again, no special pre-processing or filtering was made in order to calculating the predictions in any scenario.

```
user_movie_effect_test <- training_dataframe %>%
  left_join(movie_effect_test, by = "movieId") %>%
  group_by(userId) %>%
  summarize(user_movie_effect = mean(rating - mean_rating_test - movie_effect))
prediction_user_test <- test_dataframe %>%
  left_join(movie_effect_test, by = "movieId") %>%
  left_join(user_movie_effect_test, by = "userId") %>%
  mutate(user_movie_effect = mean_rating_test + movie_effect + user_movie_effect) %>%
  pull(user_movie_effect)
rmse_mean_user_movie_test <- RMSE(prediction_user_test, test_dataframe$rating)
rmse_mean_user_movie_test
```

```
## [1] 0.8646843
```

```
user_movie_effect_validation <- edx %>%
  left_join(movie_effect_validation, by = "movieId") %>%
  group_by(userId) %>%
  summarize(user_movie_effect = mean(rating - mean_rating_validation - movie_effect))
predicted_ratings <- validation %>%
  left_join(movie_effect_validation, by = "movieId") %>%
  left_join(user_movie_effect_validation, by = "userId") %>%
  mutate(user_movie_effect = mean_rating_validation + movie_effect + user_movie_effect) %>%
  pull(user_movie_effect)
rmse_mean_user_movie_validation <- RMSE(predicted_ratings, validation$rating)
rmse_mean_user_movie_validation
```

```
## [1] 0.8653488
```

Again, the addition one more predictor improved the results. In the test scenario, the RMSE achieved the objective of less than **0.8649** (which is the target RMSE of this work) but was a little higher in the validation scenario, having a RMSE of **0.8653**. The following table presents all scenarios how incrementing the number of predictors enhanced the performance of the predictions. The results show that both predictors, *movieId* and *userId*, can be used in other models to predict the ratings.

```
results_incremental_approach<- tibble(predictors = "mean", RMSE = rmse_mean_test,
                                     type = "test")
results_incremental_approach<-
  bind_rows(results_incremental_approach, tibble(predictors = "mean",
                                                 RMSE = rmse_mean_validation,
                                                 type = "validation"))
results_incremental_approach<- bind_rows(results_incremental_approach,
                                         tibble(predictors = "mean, movie",
                                                RMSE = rmse_mean_movie_test,
                                                type = "test"))
results_incremental_approach<- bind_rows(results_incremental_approach,
                                         tibble(predictors = "mean, movie",
                                                RMSE = rmse_mean_movie_validation,
                                              type = "validation"))
results_incremental_approach<- bind_rows(results_incremental_approach,
                                         tibble(predictors = "mean, movie, user",
                                                RMSE = rmse_mean_user_movie_test,
                                                type = "test"))
results_incremental_approach<- bind_rows(results_incremental_approach,
                                         tibble(predictors = "mean, movie, user",
                                                RMSE = rmse_mean_user_movie_validation,
                                                type = "validation"))

results_incremental_approach%>% knitr::kable()
```

predictors	RMSE	type
mean	1.0600537	test
mean	1.0612018	validation
mean, movie	0.9429615	test
mean, movie	0.9439087	validation
mean, movie, user	0.8646843	test
mean, movie, user	0.8653488	validation

3.3 recosystem: Recommender System using Matrix Factorization

recosystem < https://CRAN.R-project.org/package=recosystem> is a package specially developed to build *recommender systems* in *R*, being a powerful tool for data scientists. *Recommender systems* (or *recommendation systems*) are algorithms specialized in predict *ratings* or *preferences* that some item (*e.g.*, products, video games and, in this work, movies) would receive.

In order to use *recosystem*, both the **training** (*edx* and *training_dataframe*) and **test/validation** (*test_dataframe* and *validation*) data frames had to be converted to objects of class *DataSource*, as can be seem in the following code chunk:

```
set.seed(1, sample.kind="Rounding")
# Test and validation data frames to use with "recosystem" library
train_recosys <- with(training_dataframe,
  data_memory(user_index = userId, item_index = movieId, rating = rating))
test_recosys <- with(test_dataframe,
  data_memory(user_index = userId, item_index = movieId, rating = rating))
edx_recosys <- with(edx,
  data_memory(user_index = userId, item_index = movieId, rating = rating))
validation_recosys <- with(validation,
  data_memory(user_index = userId, item_index = movieId, rating = rating))
```

It is important to note that all data in the *DataSource* objects *train_recosys*, *test_recosys*, *edx_recosys*, *validation_recosys* are exactly the same stored in the original data frames, being just in different data structures.

The first step to predict the ratings using *recosystem* is declaring an object of *RecoSys* class. With this object, it is possible to use the functions/methods to perform the next required steps: (*i*) tune; (*ii*) train; and finally (*iii*) predict the ratings.

The methodology adopted to work with *recosystem* was an gradual increase on the number of iterations on *tune()* and *train()* functions/methods. We started with the parameter **niter = 1**, which is the number of iterations performed by *tune()* or *train()*, and gradually increased *niter* up to **20**.

```
# Creating an object of class RecoSys called "r"
r <- Reco()
# Setting tuning parameters
tuning_parameters <- r$tune(train_recosys, opts = list(dim = c(20, 30),
                                                       nthread = 4, niter = 1))
# Training stage
r$train(train_recosys, opts = c(tuning_parameters$min, nthread = 4, niter = 1))
# Predicting results
prediction_recosys <- r$predict(test_recosys, out_memory())
# Calculating RMSE
rmse_1_iter_tests <- RMSE(prediction_recosys, test_dataframe$rating)
# Printing RMSE
rmse_1_iter_tests

# Setting tuning parameters
tuning_parameters <- r$tune(edx_recosys, opts = list(dim = c(20, 30),
                                                     nthread = 4, niter = 1))
# Training stage
r$train(edx_recosys, opts = c(tuning_parameters$min, nthread = 4, niter = 1))
# Predicting results
prediction_recosys <- r$predict(validation_recosys, out_memory())
# Calculating RMSE
rmse_1_iter_final <- RMSE(prediction_recosys, validation$rating)
# Printing RMSE
rmse_1_iter_final
```

It can be observed on results shown in Fig. 5 that the RMSE started at **0.881** with **niter = 1** and were already at **0.82** with **niter = 2**. The RMSE results stayed around **0.79** from **niter = 3** to **niter = 20**, being way below the goal of less than **0.86490**.

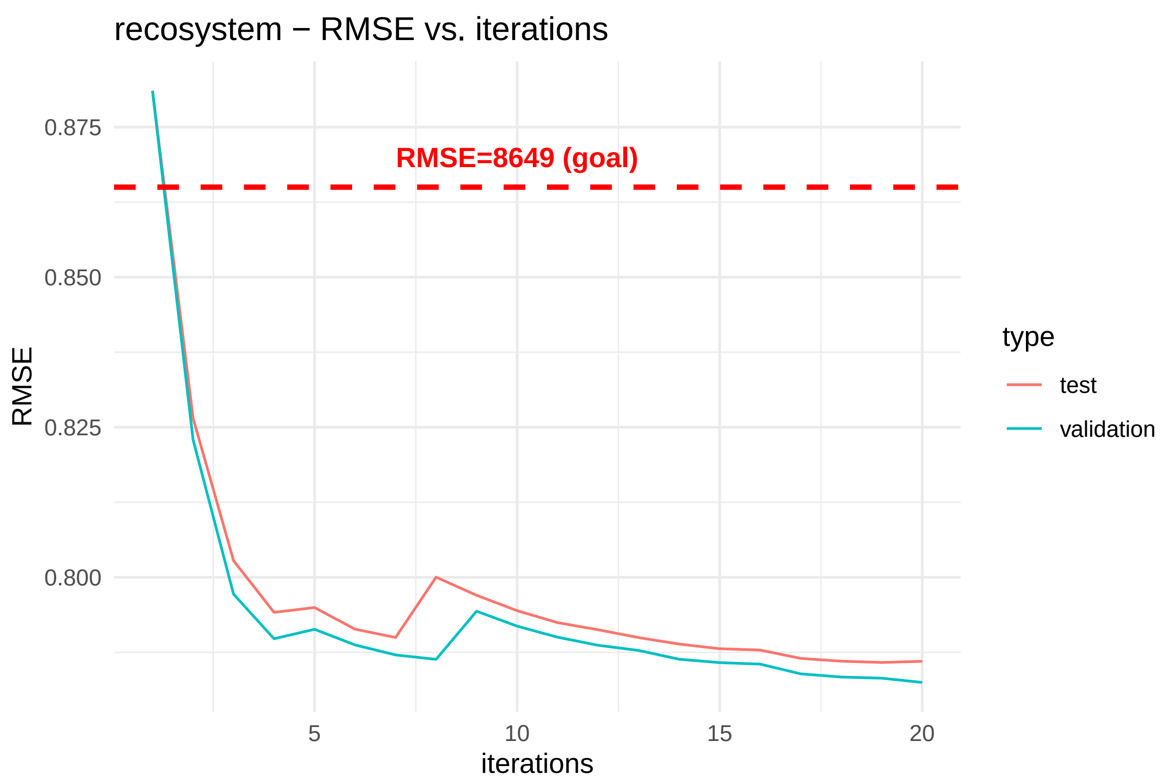

Fig. 5 − RMSE of with different iterations using recosystem.

4 Concluding remarks

The application of different mathematical approaches to predict the movie ratings had interesting results. Linear models, which is one of the simplest mathematical tools, achieved poor results, indicating that the relationship between the used data is nonlinear and complex. The use of an incremental approach, starting with the *mean* and later including *user* and *movie* effects/bias, achieved interesting results, almost reaching the intended RMSE of less than **0.86490**. It is important to note that not all movies received the same number of ratings, fact that probably had an impact on the predictions. Finaly, the use of *recosystem*, which is a package especialy made to build recommender systems, showed the best results by far, achieving RMSE of about **0.79** even when set to work with minimal parameters. The main conclusion of this work is that machine learning can be used to both retrieve as well to predict useful data about a given subject. This use can be seen everyday as a crucial tool of multibillion-dollar companies like Google, Amazon and Meta/Facebook, which use machine learning in various ways to offer better services to their users.

5 References

https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/lm

https://cran.r-project.org/web/packages/recosystem/recosystem.pdf

https://cran.r-project.org/web/packages/caret/caret.pdf

https://github.com/MicrosoftLearning/Principles-of-Machine-Learning-R/blob/master/Module5/Bias-Variance-Trade-Off.ipynb