Comparing the Performance of Automated Prediction Tools in R

Felipe Miranda

03/09/2022

1 Introduction

This report presents the analysis of the performance of two different automated prediction tools in *R*, namely, *linear models* and *recommender systems* (or *recommendation systems*). The performance of these two prediction tools is evaluated by comparing the results between their output with the ratings present in four different datasets. 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 the chosen datasets: (*i*)**Movielens**, (*ii*)**Amazon Beauty Products**, (*iii*)**Amazon Fine Foods**, and (*iv*)**Electronic Products**;

Prediction models and results: this section presents the machine learning models used to predict the **ratings** of each dataset 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/capstone.

2 Resources, Methods and Analysis

In this section, all datasets are presented together with the methodology employed to process and analyze their 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 Amazon Beauty Products,

Downloaded from *Kaggle* (https://www.kaggle.com/skillsmuggler/amazon-ratings), the description of this dataset is:

"*This dataset contains product reviews and metadata from Amazon, including 142.8 million reviews spanning May 1996 - July 2014. This dataset includes reviews (ratings, text, helpfulness votes), product metadata (descriptions, category information, price, brand, and image features), and links (also viewed/also bought graphs).*"

This specific dataset has just the ratings of *beauty products*. Its file has **83MB** and is in form of a *CSV* file.

2.3 Amazon Fine Foods

Downloaded from *Kaggle* (https://www.kaggle.com/snap/amazon-fine-food-reviews), the description of this dataset is:

"*… consists of reviews of fine foods from amazon. The data span a period of more than 10 years, including all ~500,000 reviews up to October 2012. Reviews include product and user information, ratings, and a plain text review.*"

Its file has **674MB** and is in form of a *CSV* file.

2.4 Electronic Products

Downloaded from *Kaggle* (https://www.kaggle.com/pritech/ratings-electronics), this large dataset contains the reviews of electronic products, rated by thousands of users.

Its file has **320MB** and is in form of a *CSV* file.

2.5 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)
# Package "digest" is used to manipulate hash codes
if(!require(digest)) install.packages("digest", repos = "http://cran.us.r-project.org")
library(digest)
```

2.6 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, *movie_df* 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")

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)

movies_df <- movielens[-test_index,]
temp <- movielens[test_index,]

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

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

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

################################################################################
# DATASET SAMPLING; TRAINING AND VALIDATION DATA FRAMES
################################################################################
# Creating training and test data frames
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
test_index <- createDataPartition(y = movies_df$rating, times = 1, p = 0.1, list = FALSE)
training_dataframe <- movies_df[-test_index,]
```

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

As *Kaggle* website demands authentication for downloading their datasets, the other datasets, *Amazon Beauty Products*, *Amazon Fine Foods*, and *Electronic Products*, had to be hosted in a different platform. For an easier download process, the platform used was *Dropbox*, which could provide a direct download of the unzipped files, as shown on the following script (all three datasets used the same procedure):

```
# Downloading the dataset from Dropbox
temp <- read.csv("https://www.dropbox.com/s/edu7fcy7fx3xmpb/ratings_Beauty.csv?dl=1")
# Removing all NAs
temp <- na.omit(temp)
# Removing empty scores/ratings
beauty <- temp %>% filter(Rating != "") %>% droplevels()
# Casting UserId and ProductId as characters (in order to avoid some errors)
beauty$UserId <- as.character(beauty$UserId)
beauty$ProductId <- as.character(beauty$ProductId)
# Converting UserId and ProductId to numbers (they are originally hash codes)
beauty$UserId <- as.numeric(digest::digest2int(beauty$UserId))
beauty$ProductId <- as.numeric(digest::digest2int(beauty$ProductId))
# Ordering the data frame by the UserId
beauty<-as.data.frame(beauty[order(beauty$UserId),])
# As both UserId and ProductId are very large numbers, it was decided to
# change it to their respective ranks, e.g.: UserId "1082878268" could be converted
# to "14" or ProductId "568411023" could be converted to "1023" etc.
rownames(beauty) <- c(1:nrow(beauty))
beauty$UserId<-round(rank(beauty$UserId, ties.method = "average"))
beauty$ProductId<-round(rank(beauty$ProductId , ties.method = "average"))

################################################################################
# DATASET SAMPLING; TRAINING AND VALIDATION DATA FRAMES
################################################################################
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use ‘set.seed(1)‘
test_index <- createDataPartition(y = beauty$Rating, times = 1, p = 0.1, list = FALSE)

beauty_df <- beauty[-test_index,]
temp <- beauty[test_index,]

# Make sure UserId and ProductId in validation set are also in beauty_df set
validation <- temp %>%
  semi_join(beauty_df, by = "ProductId") %>%
  semi_join(beauty_df, by = "UserId")

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

```
rm(test_index, temp, beauty, removed)
```

It is important to note that, differently from *MovieLens 10M* dataset, it was needed to remove empty fields and to transform some of their data to shorter numbers, as they were originally *hashcodes* (https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm) consisting of long strings. The sampling process was the same for all four datasets.

2.7 Movies data frame

The following results presented in this subsection show general data stored in *movies_df* 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 movies_df data frame,
# it was manually processed.
# movies_df has 10677 movies.
length(unique(movies_df$movieId))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

In the histogram shown in Fig. 1, we can observe that the ratings in *movies_df* 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 *movies_df* is **3.512465**.

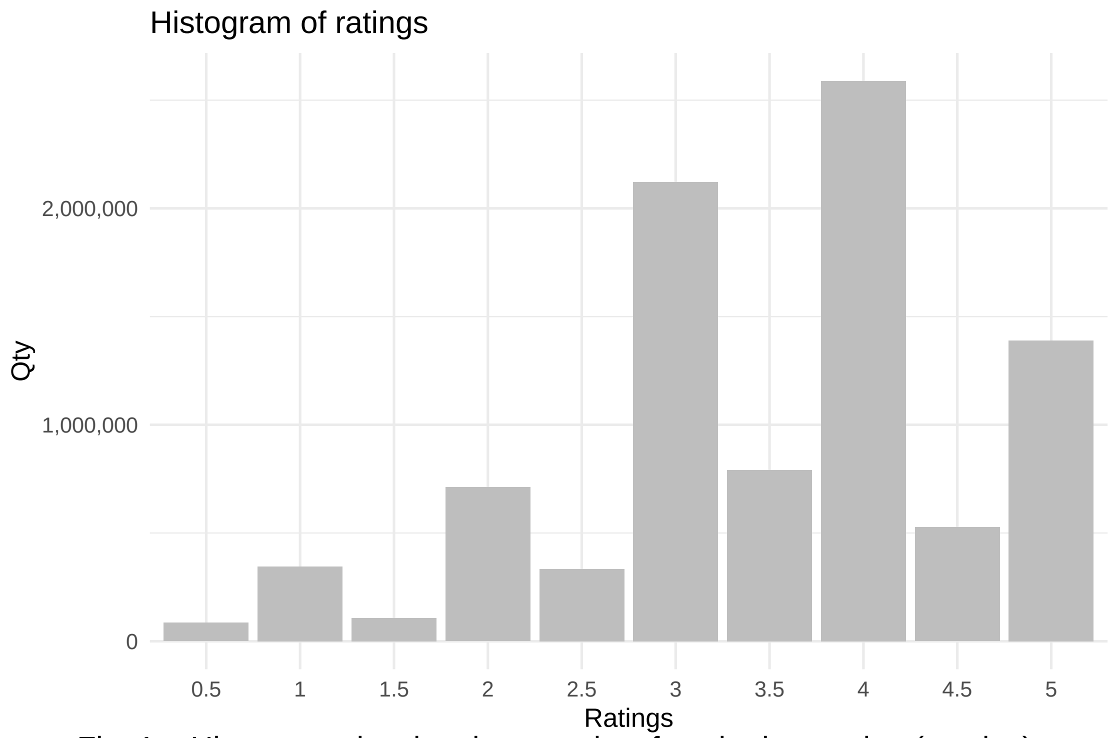


Fig. 1 − Histogram showing the quantity of each given rating (movies).

In the histogram shown in Fig. 2 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.

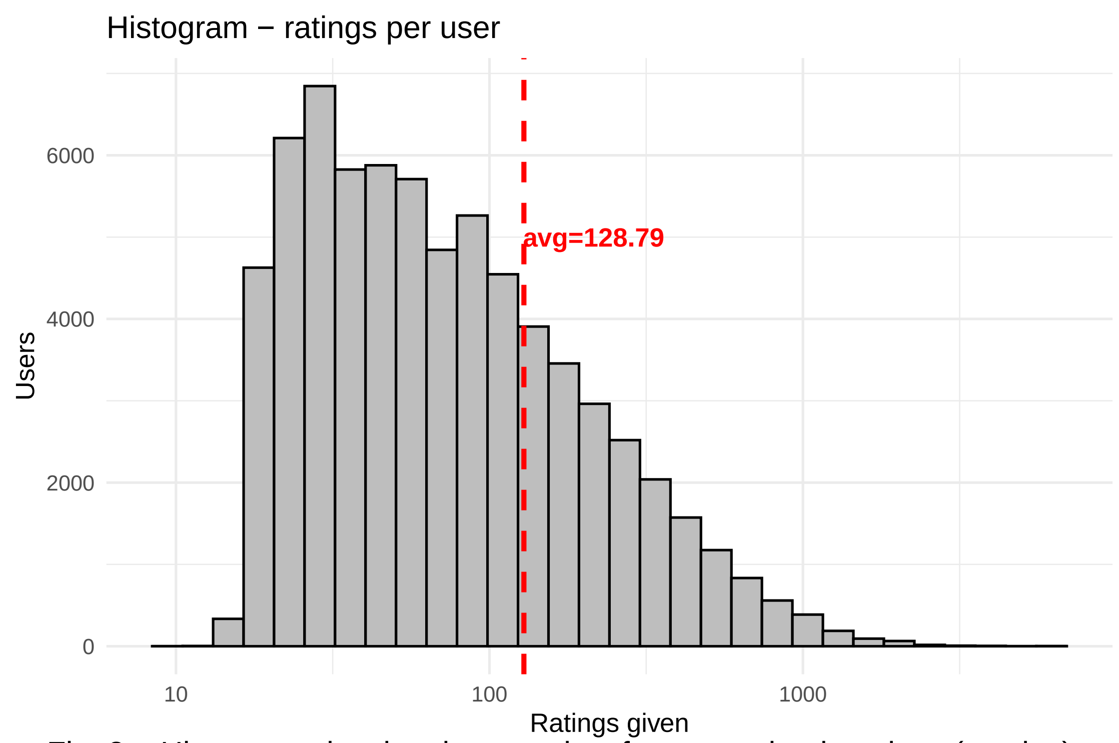


Fig. 2 − Histogram showing the quantity of users and rating given (movies).

The histogram shown in Fig. 3 presents the number of ratings given per year. It is important to note that the *timestamp* in *movies_df* 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.

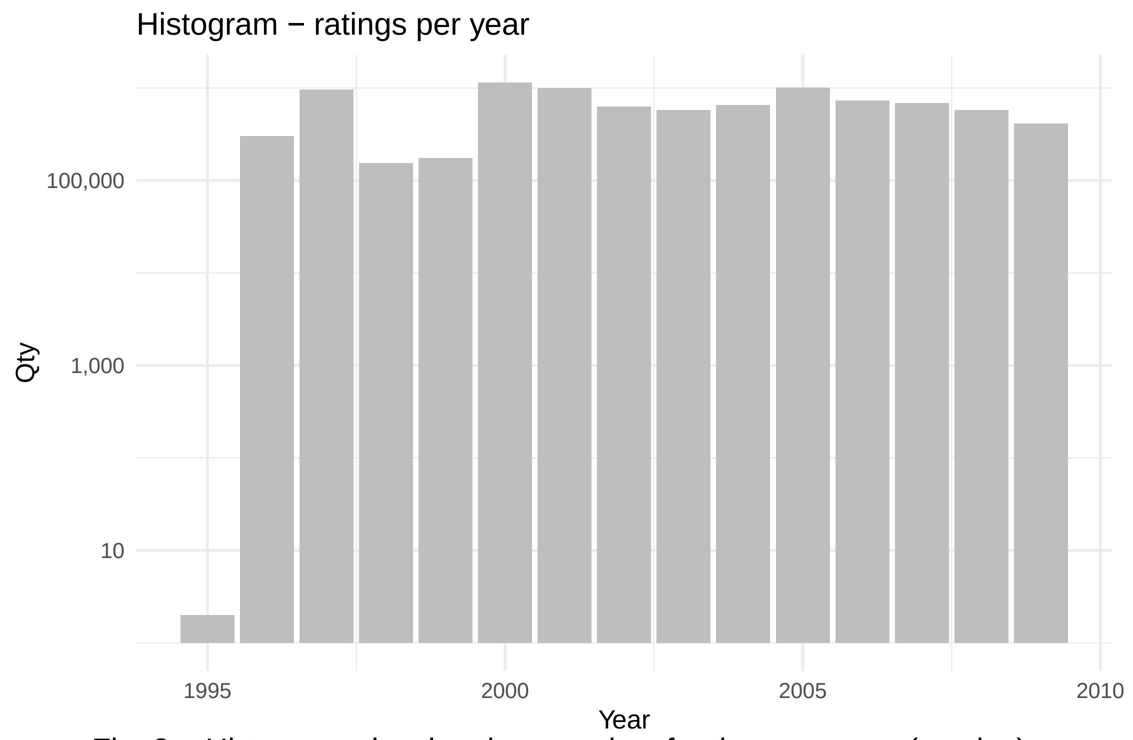


Fig. 3 − Histogram showing the quantity of ratings per year (movies).

2.8 Beauty products data frame

The following results presented in this subsection show general data stored in *beauty_df* 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 'beauty_df' data frame,
# it was manually processed.
# beauty_df has 249269 products.
length(unique(beauty_df$ProductId))
```

```
## [1] 249269
```

```
# beauty_df has 1210084 unique UserIds, which means that 1210084 users rated the movies.
length(unique(beauty_df$UserId))
```

```
## [1] 1210084
```

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

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

```
# The rating "5" was given 1185304 times.
sum(beauty_df$Rating==5)
```

```
## [1] 1185304
```

```
# The lowest rating is 1
min(beauty_df$Rating)
```

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

```
# The rating "1" was given 176557 times.
sum(beauty_df$Rating==1)
```

```
## [1] 176557
```

```
# The average rating of beauty_df 4.146228.
mean(beauty_df$Rating)
```

```
## [1] 4.146228
```

```
# The standard deviation rating of beauty_df is 1.315409.
sd(beauty_df$Rating)
```

```
## [1] 1.315409
```

In the histogram shown in Fig. 4, we can observe that the ratings in *beauty_df* are **5**, resulting on a very high average of **4.146228**. It is important to note that, differently from *movies_df*, the ratings are made of just integer numbers,*i.e.*, there is no ratings like **4.5** or **3.5** in *beauty_df*.

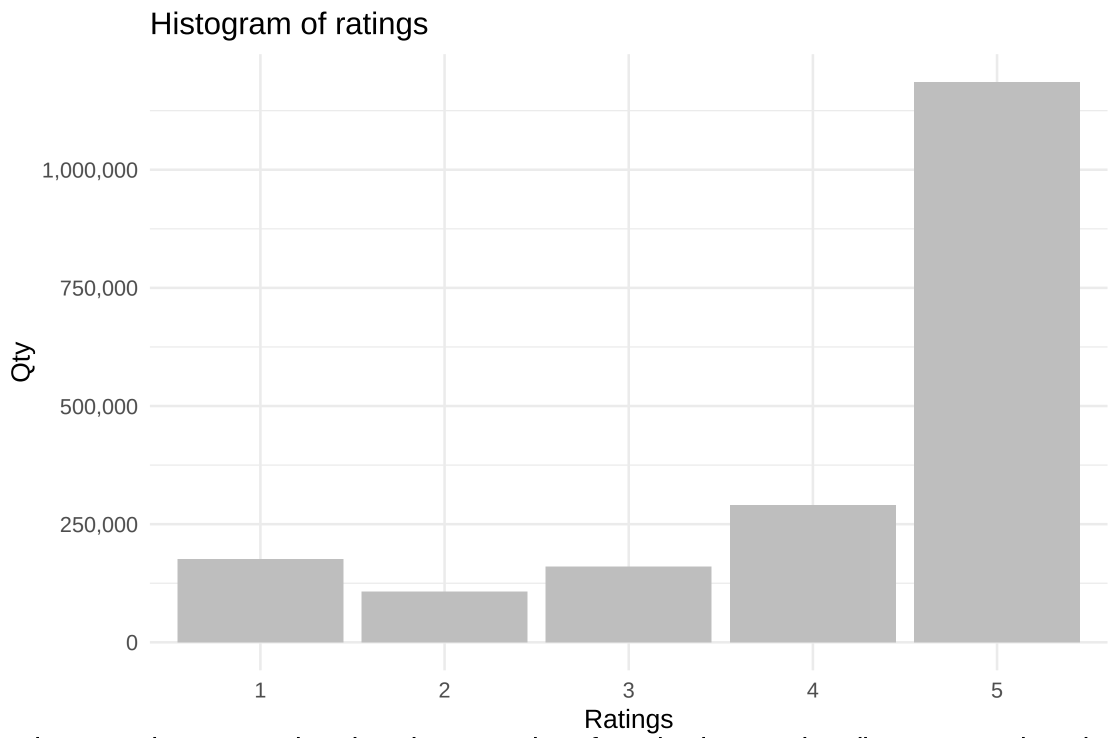
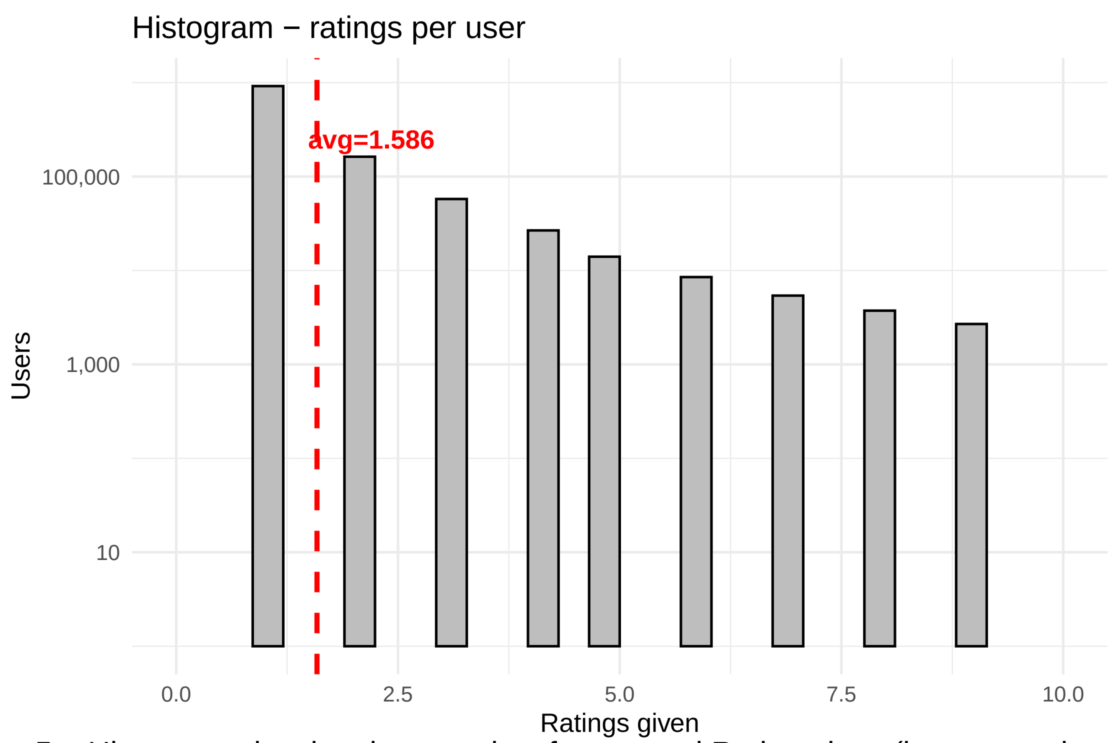


Fig. 4 − Histogram showing the quantity of each given rating (beauty products).

In the histogram shown in Fig. 5 , we can observe that most users have a very low number of ratings given, with an average of **1.586**.



ig. 5 − Histogram showing the quantity of users and Rating given (beauty products

The histogram shown in Fig. 6 presents the number of ratings given per year. It is important to note that the *timestamp* in *beauty_df* 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.

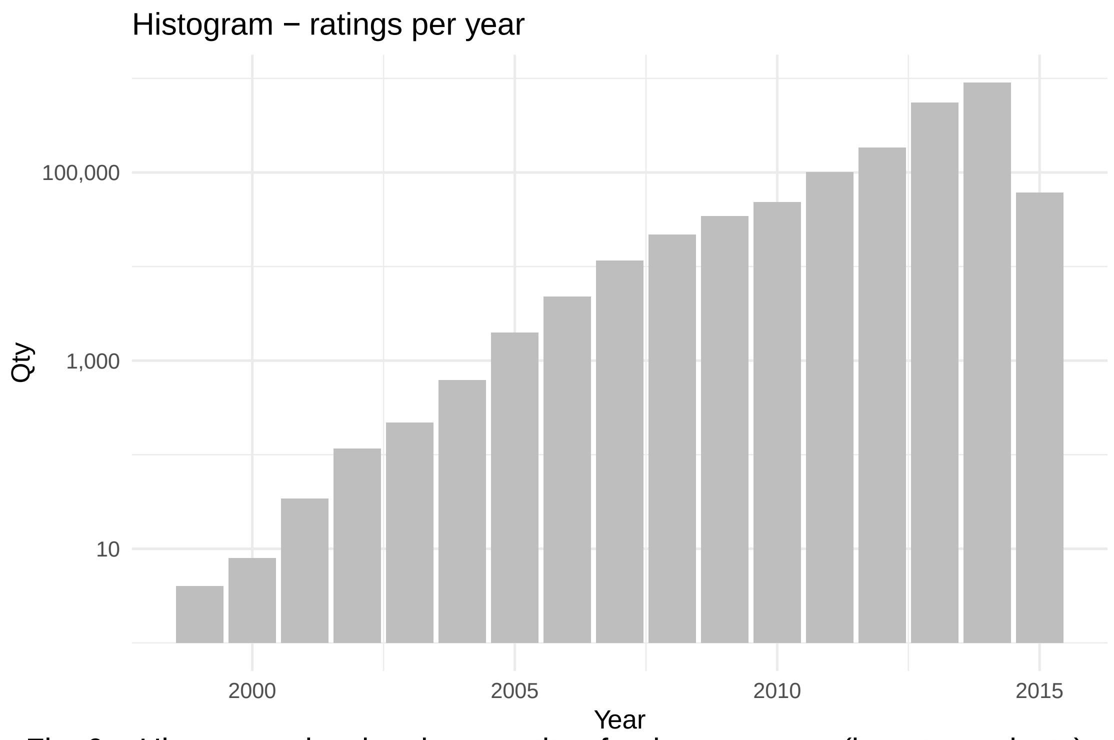


Fig. 6 − Histogram showing the quantity of ratings per year (beauty products).

2.9 Fine foods data frame

The following results presented in this subsection show general data stored in *food_df* 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 'food_df' data frame,
# it was manually processed.
# food_df has 74258 products.
length(unique(food_df$ProductId))
```

```
## [1] 74258
```

```
# food_df has 256047 unique userIds, which means that 69878 256047 rated the products.
length(unique(food_df$UserId))
```

```
## [1] 256047
```

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

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

```
# The rating "5" was given 339932 times.
sum(food_df$Rating==5)
```

```
## [1] 339932
```

```
# The lowest rating is 1
min(food_df$Rating)
```

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

```
# The rating "1" was given 49259 times.
sum(food_df$Rating==1)
```

```
## [1] 49259
```

```
# The average rating of food_df 4.182434.
mean(food_df$Rating)
```

```
## [1] 4.182434
```

```
# The standard deviation rating of food_df is 1.31313.
sd(food_df$Rating)
```

```
## [1] 1.31313
```

In the histogram shown in Fig. 7, we can observe that the ratings in *food_df* are **5**, resulting on a very high average of **4.182434**. It is important to note that, differently from *movies_df*, the ratings are made of just integer numbers,*i.e.*, there is no ratings like **4.5** or **3.5** in *food_df*.

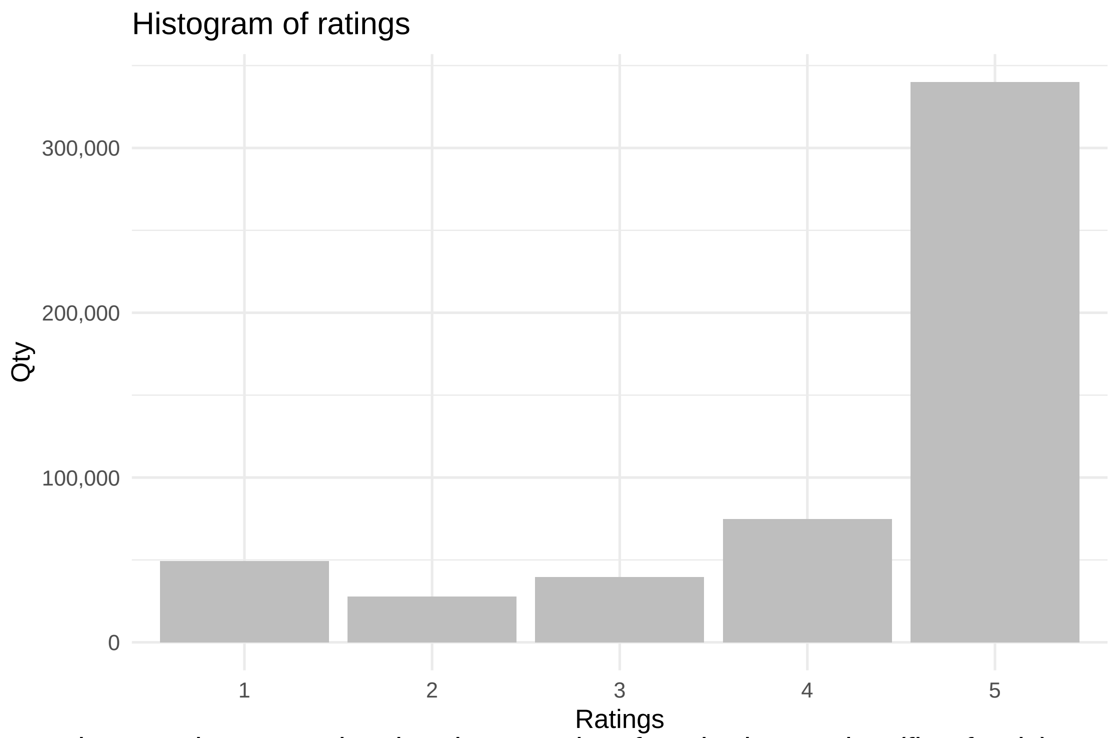


Fig. 7 − Histogram showing the quantity of each given rating (fine foods).

In the histogram shown in Fig. 8 , we can observe that most users have a very low number of ratings given, with an average of **2.075**.

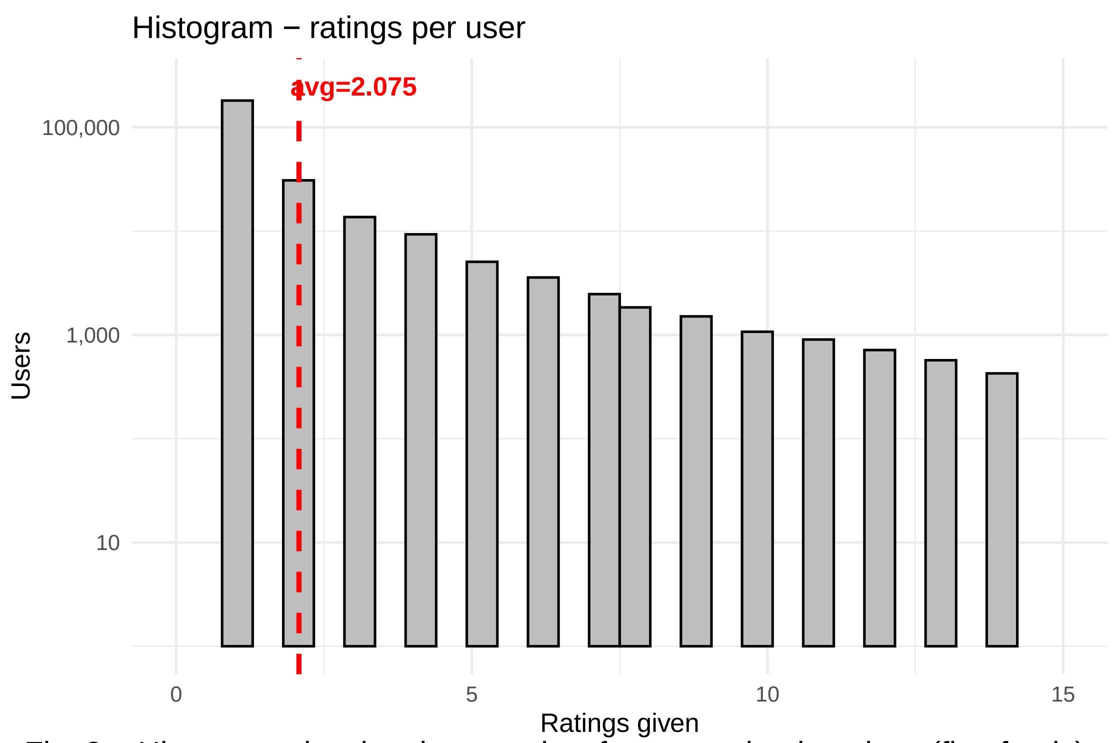


Fig. 8 − Histogram showing the quantity of users and rating given (fine foods).

The histogram shown in Fig. 9 presents the number of ratings given per year. It is important to note that the *timestamp* in *beauty_df* 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.

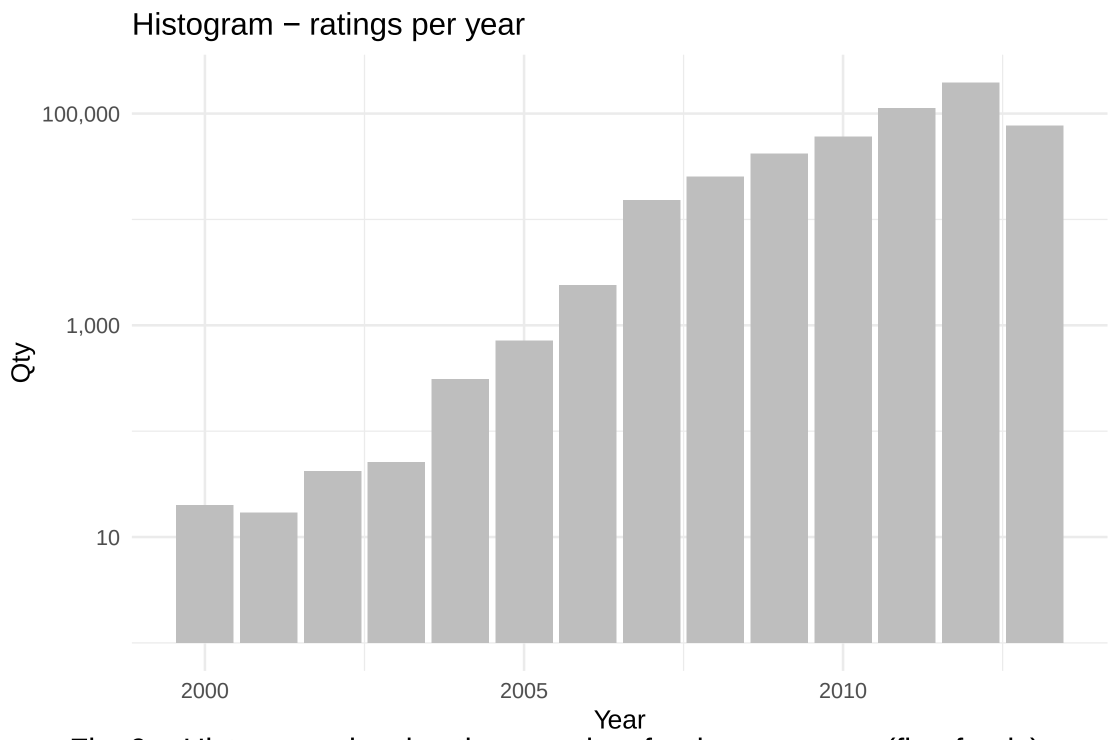


Fig. 9 − Histogram showing the quantity of ratings per year (fine foods).

2.10 Electronic products data frame

The following results presented in this subsection show general data stored in *electronics_df* 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 'electronics_df' data frame,
# it was manually processed.
# electronics_df has 475970 products.
length(unique(electronics_df$ProductId))
```

```
## [1] 475970
```

```
# electronics_df has 4199593 unique UserIds, which means that 4199593 users rated the movies.
length(unique(electronics_df$UserId))
```

```
## [1] 4199593
```

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

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

```
# The rating "5" was given 4078876 times.
sum(electronics_df$Rating==5)
```

```
## [1] 4078876
```

```
# The lowest rating is 1
min(electronics_df$Rating)
```

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

```
# The rating "1" was given 860965 times.
sum(electronics_df$Rating==1)
```

```
## [1] 860965
```

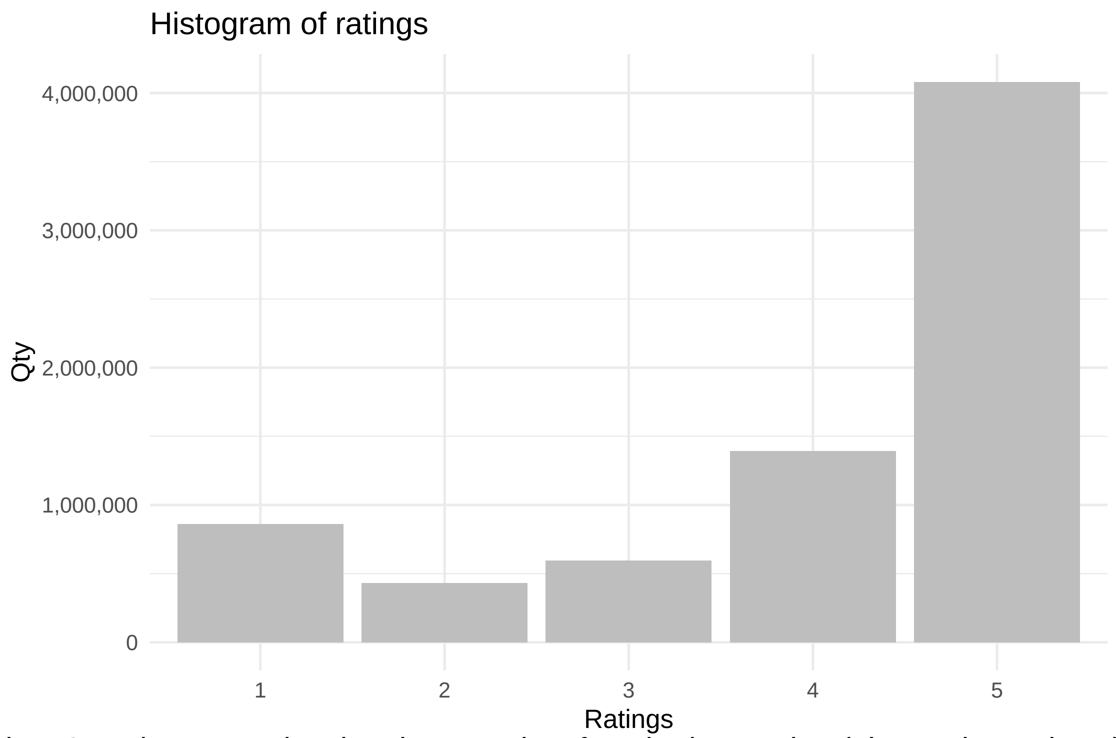
```
# The average rating of electronics_df 4.005487.
mean(electronics_df$Rating)
```

```
## [1] 4.005487
```

```
# The standard deviation rating of electronics_df is 1.38653.
sd(electronics_df$Rating)
```

```
## [1] 1.38653
```

In the histogram shown in Fig. 10, we can observe that the ratings in *food_df* are **5**, resulting on a very high average of **4.005487**. It is important to note that, differently from *movies_df*, the ratings are made of just integer numbers,*i.e.*, there is no ratings like **4.5** or **3.5** in *food_df*.



-ig. 10 − Histogram showing the quantity of each given rating (electronic products)

The histogram shown in Fig. 11 presents the number of ratings given per year. It is important to note that the *timestamp* in *beauty_df* 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.

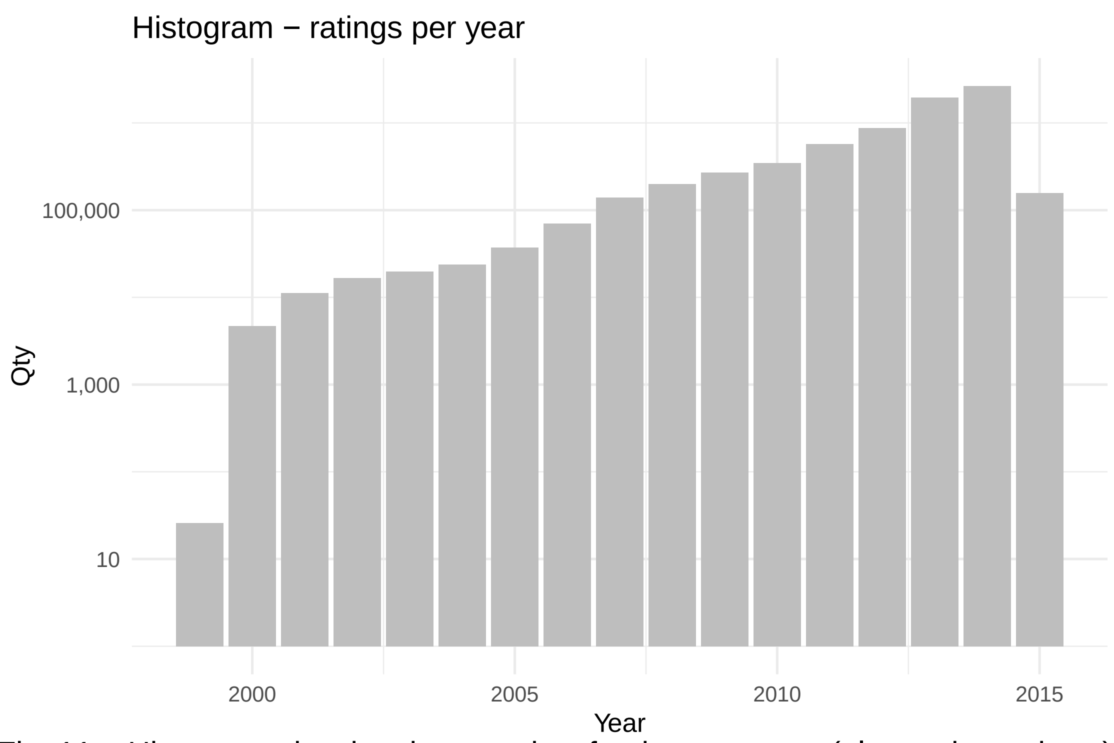


Fig. 11 − Histogram showing the quantity of ratings per year (electronic products).

As the *electronics_df* has a very high number of users, the histogram showing the quantity of users and rating given could not be plotted due to memory constraints.

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 was set to **0.86490** in order to achieve its objective (similar to the work in : https://github.com/fm-33/movielens).

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

```r
################################################################################
# TRAINING AND TEST DATA FRAMES
################################################################################
# Creating training and test data frames
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
test_index <- createDataPartition(y = movies_df$rating, times = 1, p = 0.1, list = FALSE)
training_dataframe <- movies_df[-test_index,]
temp <- movies_df[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)
```

The same procedure was used to create the other *training* and *test* data frames. It is also important to note that all four datasets had the same columns used as predictors in this work: (*i*) **movie/product id**, (*ii*)**user (evaluator) id**, and (*iii*)**timestamp**.

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.

3.1.1 Movies data frame

```r
# lm() function using timestamp as a predictor.
model <- lm(formula=rating ~ timestamp, data = training_dataframe)
prediction_movies_lm_test_ts <- predict(model, newdata = test_dataframe)
rmse_movies_lm_test_ts <- RMSE(prediction_movies_lm_test_ts, test_dataframe$rating)
model <- lm(formula=rating ~ timestamp, data = movies_df)
prediction_validation_ts <- predict(model, newdata = validation)
rmse_movies_lm_validation_ts <- RMSE(prediction_validation_ts, validation$rating)
# lm() function using timestamp and userId as predictors.
model <- lm(formula=rating ~ timestamp+userId, data = training_dataframe)
prediction_movies_lm_test_ts_ui <- predict(model, newdata = test_dataframe)
rmse_movies_lm_test_ts_ui <- RMSE(prediction_movies_lm_test_ts_ui, test_dataframe$rating)
model <- lm(formula=rating ~ timestamp+userId, data = movies_df)
prediction_movies_lm_validation_ts_ui <- predict(model, newdata = validation)
rmse_movies_lm_validation_ts_ui <- RMSE(prediction_movies_lm_validation_ts_ui, validation$rating)
# lm() function using timestamp, userId and movieId as predictors.
model <- lm(formula=rating ~ timestamp+userId+movieId, data = training_dataframe)
prediction_movies_lm_test_ts_ui_mi <- predict(model, newdata = test_dataframe)
rmse_movies_lm_test_ts_ui_mi <- RMSE(prediction_movies_lm_test_ts_ui_mi, test_dataframe$rating)
model <- lm(formula=rating ~ timestamp+userId+movieId, data = movies_df)
prediction_movies_lm_validation_ts_ui_mi <- predict(model, newdata = validation)
rmse_movies_lm_validation_ts_ui_mi <- RMSE(prediction_movies_lm_validation_ts_ui_mi, validation$rating)

# Compiling all results and and showing them in a table
results_movies_lm <- tibble(predictors = "timestamp", RMSE = rmse_movies_lm_test_ts, type = "test")
results_movies_lm <- bind_rows(results_movies_lm, tibble(predictors = "timestamp",
RMSE = rmse_movies_lm_validation_ts, type = "validation"))
results_movies_lm <- bind_rows(results_movies_lm, tibble(predictors = "timestamp and userId",
RMSE = rmse_movies_lm_test_ts_ui, type = "test"))
results_movies_lm <- bind_rows(results_movies_lm, tibble(predictors = "timestamp and userId",
RMSE = rmse_movies_lm_validation_ts_ui, type = "validation"))
```

```
results_movies_lm <- bind_rows(results_movies_lm, tibble(predictors = "timestamp, userId and movieId",
RMSE = rmse_movies_lm_test_ts_ui_mi, type = "test"))
results_movies_lm <- bind_rows(results_movies_lm, tibble(predictors = "timestamp, userId and movieId",
RMSE = rmse_movies_lm_validation_ts_ui_mi, type = "validation"))
```

```
results_movies_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*. 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.

3.1.2 Beauty products data frame

```
# 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$Rating)
model <- lm(formula=Rating ~ Timestamp, data = beauty_df)
prediction_validation_ts <- predict(model, newdata = validation)
rmse_lm_validation_ts <- RMSE(prediction_validation_ts, validation$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$Rating)
model <- lm(formula=Rating ~ Timestamp+UserId, data = beauty_df)
prediction_lm_validation_ts_ui <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui <- RMSE(prediction_lm_validation_ts_ui, validation$Rating)
# lm() function using Timestamp, UserId and ProductId as predictors.
model <- lm(formula=Rating ~ Timestamp+UserId+ProductId, 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$Rating)
model <- lm(formula=Rating ~ Timestamp+UserId+ProductId, data = beauty_df)
prediction_lm_validation_ts_ui_mi <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui_mi <- RMSE(prediction_lm_validation_ts_ui_mi,
validation$Rating)
# Compiling all results and and showing them in a table
results_beauty_lm <- tibble(predictors = "timestamp",
RMSE = rmse_lm_test_ts, type = "test")
results_beauty_lm <- bind_rows(results_beauty_lm, tibble(predictors = "timestamp",
RMSE = rmse_lm_validation_ts, type = "validation"))
results_beauty_lm <- bind_rows(results_beauty_lm, tibble(predictors = "timestamp
and userId", RMSE = rmse_lm_test_ts_ui, type = "test"))
results_beauty_lm <- bind_rows(results_beauty_lm, tibble(predictors = "timestamp
and userId", RMSE = rmse_lm_validation_ts_ui, type = "validation"))
results_beauty_lm <- bind_rows(results_beauty_lm, tibble(predictors = "timestamp,
userId and movieId",
RMSE = rmse_lm_test_ts_ui_mi, type = "test"))
results_beauty_lm <- bind_rows(results_beauty_lm, tibble(predictors = "timestamp,
userId and movieId",
RMSE = rmse_lm_validation_ts_ui_mi, type = "validation"))
```

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

predictors	RMSE	type
timestamp	1.235810	test
timestamp	1.236352	validation
timestamp and userId	1.235810	test
timestamp and userId	1.236351	validation
timestamp, userId and movieId	1.235808	test
timestamp, userId and movieId	1.236348	validation

As can be noted on the results, the results were, again, 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 *ProductId*.

3.1.3 Fine foods data frame

```
# 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$Rating)
model <- lm(formula=Rating ~ Timestamp, data = food_df)
prediction_validation_ts <- predict(model, newdata = validation)
rmse_lm_validation_ts <- RMSE(prediction_validation_ts, validation$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$Rating)
model <- lm(formula=Rating ~ Timestamp+UserId, data = food_df)
prediction_lm_validation_ts_ui <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui <- RMSE(prediction_lm_validation_ts_ui, validation$Rating)
# lm() function using Timestamp, UserId and ProductId as predictors.
model <- lm(formula=Rating ~ Timestamp+UserId+ProductId, 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$Rating)
model <- lm(formula=Rating ~ Timestamp+UserId+ProductId, data = food_df)
prediction_lm_validation_ts_ui_mi <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui_mi <- RMSE(prediction_lm_validation_ts_ui_mi, validation$Rating)

# Compiling all results and and showing them in a table
results_food_lm <- tibble(predictors = "timestamp", RMSE = rmse_lm_test_ts, type = "test")
results_food_lm <- bind_rows(results_food_lm, tibble(predictors = "timestamp",
RMSE = rmse_lm_validation_ts, type = "validation"))
results_food_lm <- bind_rows(results_food_lm, tibble(predictors = "timestamp and userId",
RMSE = rmse_lm_test_ts_ui, type = "test"))
results_food_lm <- bind_rows(results_food_lm, tibble(predictors = "timestamp and userId",
RMSE = rmse_lm_validation_ts_ui, type = "validation"))
results_food_lm <- bind_rows(results_food_lm, tibble(predictors = "timestamp, userId and movieId",
RMSE = rmse_lm_test_ts_ui_mi, type = "test"))
results_food_lm <- bind_rows(results_food_lm, tibble(predictors = "timestamp, userId and movieId",
RMSE = rmse_lm_validation_ts_ui_mi, type = "validation"))
```

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

predictors	RMSE	type
timestamp	1.271761	test
timestamp	1.269335	validation
timestamp and userId	1.271759	test
timestamp and userId	1.269335	validation
timestamp, userId and movieId	1.271685	test
timestamp, userId and movieId	1.269335	validation

As can be noted on the results, the results were, again, 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 *ProductId*.

3.1.4 Electronic products data frame

```
# 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$Rating)
model <- lm(formula=Rating ~ Timestamp, data = electronics_df)
prediction_validation_ts <- predict(model, newdata = validation)
rmse_lm_validation_ts <- RMSE(prediction_validation_ts, validation$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$Rating)
model <- lm(formula=Rating ~ Timestamp+UserId, data = electronics_df)
prediction_lm_validation_ts_ui <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui <- RMSE(prediction_lm_validation_ts_ui, validation$Rating)
# lm() function using Timestamp, UserId and ProductId as predictors.
model <- lm(formula=Rating ~ Timestamp+UserId+ProductId, 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$Rating)
model <- lm(formula=Rating ~ Timestamp+UserId+ProductId, data = electronics_df)
prediction_lm_validation_ts_ui_mi <- predict(model, newdata = validation)
rmse_lm_validation_ts_ui_mi <- RMSE(prediction_lm_validation_ts_ui_mi, validation$Rating)

# Compiling all results and and showing them in a table
results_electronics_lm <- tibble(predictors = "timestamp", RMSE = rmse_lm_test_ts, type = "test")
results_electronics_lm <- bind_rows(results_electronics_lm,
tibble(predictors = "timestamp",
RMSE = rmse_lm_validation_ts, type = "validation"))
results_electronics_lm <- bind_rows(results_electronics_lm,
tibble(predictors = "timestamp and userId",
RMSE = rmse_lm_test_ts_ui, type = "test"))
results_electronics_lm <- bind_rows(results_electronics_lm,
tibble(predictors = "timestamp and userId",
RMSE = rmse_lm_validation_ts_ui, type = "validation"))
results_electronics_lm <- bind_rows(results_electronics_lm,
tibble(predictors = "timestamp, userId and movieId",
RMSE = rmse_lm_test_ts_ui_mi, type = "test"))
results_electronics_lm <- bind_rows(results_electronics_lm,
tibble(predictors = "timestamp, userId and movieId",
RMSE = rmse_lm_validation_ts_ui_mi, type = "validation"))
```

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

predictors	RMSE	type
timestamp	1.271761	test
timestamp	1.287493	validation
timestamp and userId	1.271759	test
timestamp and userId	1.287492	validation
timestamp, userId and movieId	1.271685	test
timestamp, userId and movieId	1.287556	validation

As can be noted on the results, the results were, again, 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 *ProductId*.

3.2 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** (*movies_df* 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 (the same procedure was followed for all data frames):

```
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))
movies_df_recosys <- with(movies_df,
  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*, *movies_df_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 **5**. It is important to note that the default value for *niter* is **20**, but it was reduced to provide faster processing. The following code chunk shows how the predictions were made (the same procedure was followed for all data frames):

```
# 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_movies_tests <- RMSE(prediction_recosys, test_dataframe$rating)
# Printing RMSE
rmse_1_iter_movies_tests
```

It can be observed on results shown in Fig. 12-15 that the RMSE improved when the *niter* was increased in all scenarios.Besides all four datasets having the same range of ratings (from **0** to **5**), an interesting outcome is that *recosystem* only reached the intended RMSE of **0.86490** when working with *MovieLens* dataset, having considerably higher RMSE in all other scenarios, with some results being less accurate when compared to *linear models.*

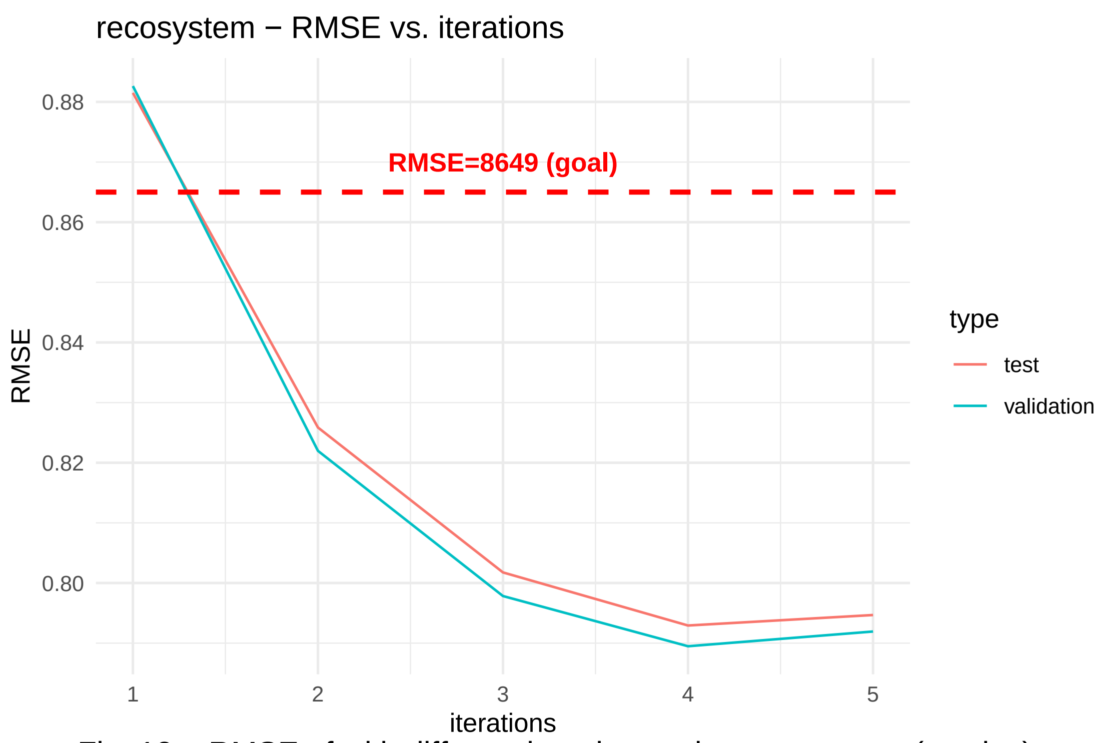


Fig. 12 − RMSE of with different iterations using recosystem (movies).

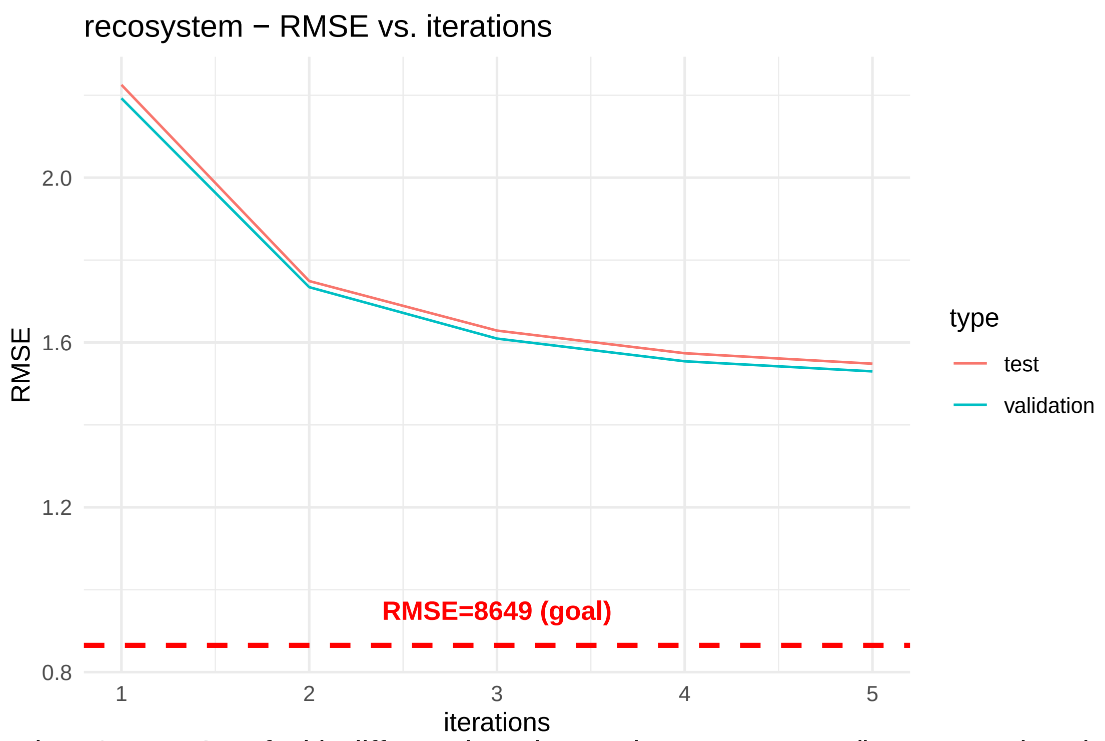


Fig. 13 − RMSE of with different iterations using recosystem (beauty products).

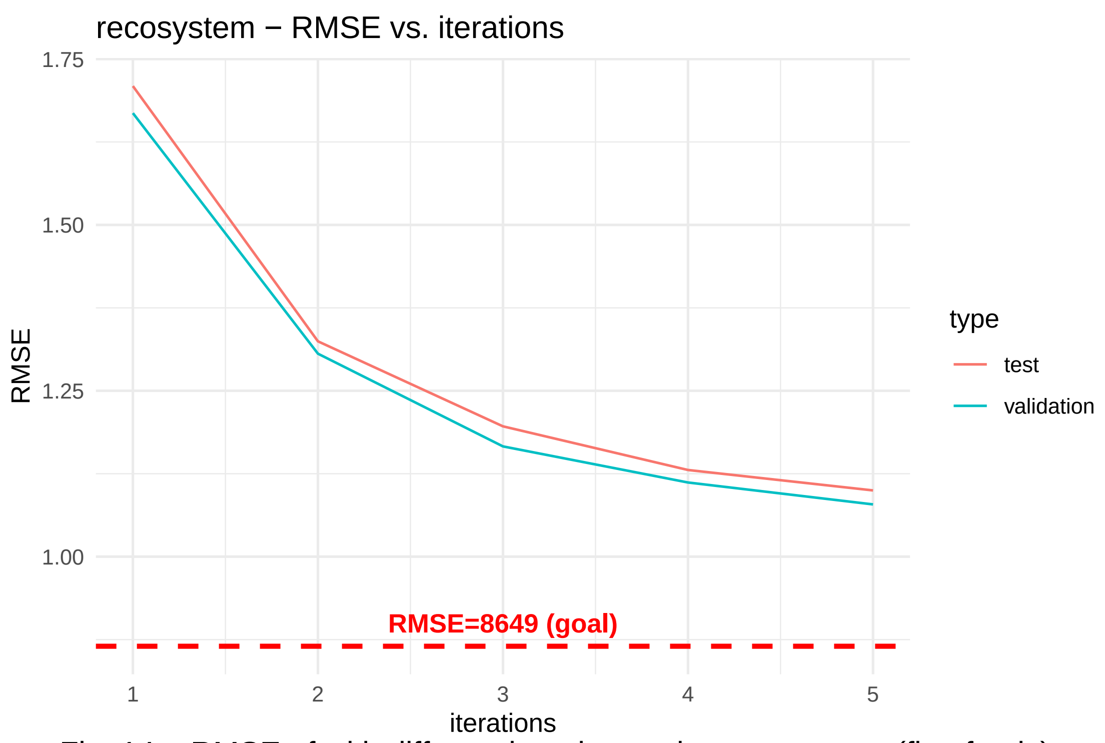


Fig. 14 − RMSE of with different iterations using recosystem (fine foods).

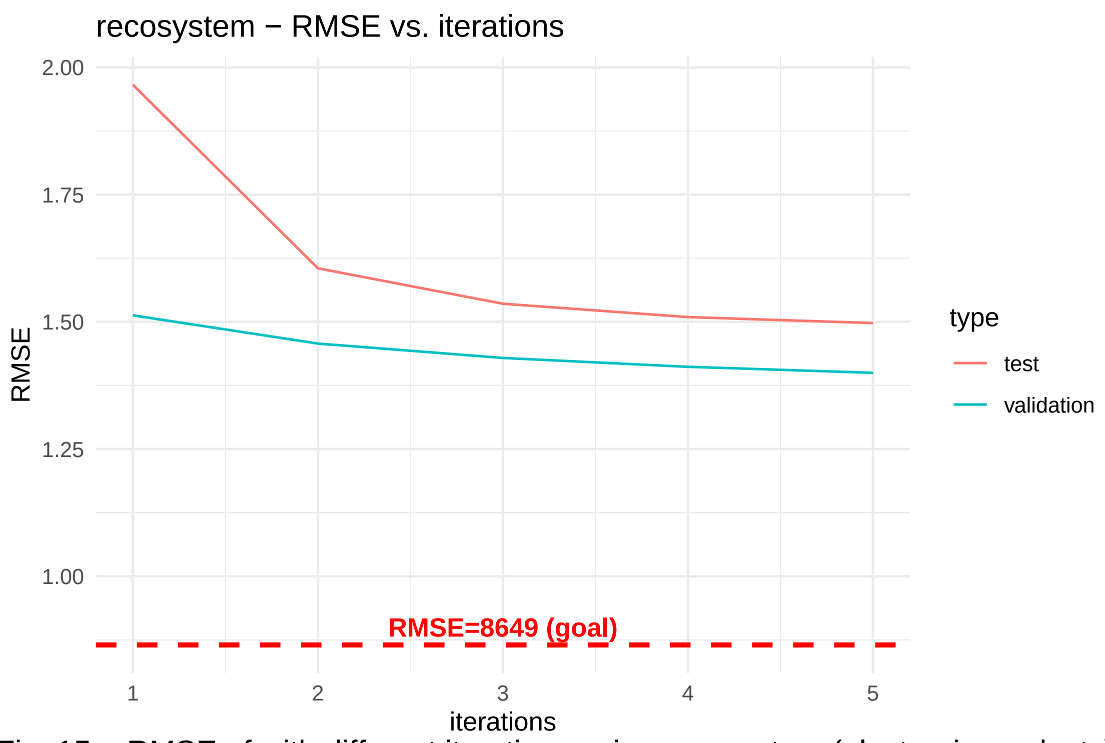


Fig. 15 − RMSE of with different iterations using recosystem (electronic products).

4 Concluding remarks

The application of different mathematical approaches to predict product ratings had interesting results. *Linear models*, which is one of the simplest mathematical tools, achieved similarly poor results in all scenarios, indicating that the relationship between the used data is nonlinear and complex, independently from its source.

The use of *recosystem*, which is a package especially made to build recommender systems, showed the best results when using *MovieLens* dataset, achieving RMSE of about **0.79**, considerably bellow the intended RMSE. On the other hand,when working with the other three datasets, *Amazon Beauty Products*, *Amazon Fine Foods*, and *Electronic Products*, *recosystem* achieved some results less accurate than with the use of *linear models*

The main conclusion of this work is that, besides being a very powerful technology, machine learning is not a perfect solution for an automatic prediction in all scenarios. One key difference between all four datasets used in this work is that *MovieLens* was the only one maintained by a specialized group, *GroupLens*. The origin of the data, how it was collected and organized have a sensible impact on the data quality, which can have negative effects when using automated tools to predict outputs. In this context, ISO 8000 is a global standard exclusively for *Data Quality*.

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