MovieLens Recommender System

Carlos Outerelo

Contents

Preface

This project aims to be as educational and detailed as possible to showcase my understanding and proficiency of the R programming language, R Markdown and the topics at hand: data science (manipulation, exploration, visualization…) and machine learning (linear and matrix factorization models in particular). Note that the R Markdown file outputs a .HTML document (that has been uplodad to RPubs where it can be properly visualized) as well as a .PDF file, both of which are available in this project's GitHub repository.

1. Introduction

Recommender systems are now as important as ever, with many of the most popular services of today's digital society (YouTube, Spotify, Amazon, Steam…) making use of them in order to provide users with the content they are most likely to consume and/or enjoy. To properly target their userbases, these businesses and platforms require user data to understand their behavior and preferences, which is where data science comes into play. Data science drives the datamined user information through many different processes which can lead to the creation/construction of intelligent systems able to perform accurate data-backed predictions that are likely to be of interest and/or valuable to the users. For these predictions to be as accurate and successful as possible, current recommendation systems are built upon the most "in vogue" item from within the data science toolset: machine learning algorithms.

Machine learning algorithms build prediction models based on the data they are supplied with. They improve their accuracy upon training, a process that loops through the sampled data contrasting guesses with real values to evaluate the algorithm' success so that it can develop a statistical model which maximizes its accuracy and best fits the supplied data. These models can be used in classification and regression exercises/scenarios (to predict integers/factors and continuous values, respectively) being able to uncover key insights and relationships from within the data that otherwise would be impossible to take grasp of. Recommender systems usually present regression-based scenarios where machine learning can be used to understand user data and provide insightful recommendations based on a set of factors/predictors, assigning each item (either on a global scale and/or on a per-user level) a continuous value which is used to score/rank its worth as a recommendation.

1.1. The Netflix Prize

In October 2006, Netflix released a dataset containing 100 million anonymous movie ratins and challenged the data mining, machine learning and computer science communities to develop an algorithmic model that could surpass their own recommender system, Cinematch. The challenge was known as The Netflix Prize due to the million dollar grand compensation at play, and the chosen evaluation metric to measure all models' accuracy (and thus determine if a given system had surpassed Cinematch) was the the root mean squared error (RMSE), which will be properly detailed later on in the evaluation metrics chapter of this document.

Netflix provided both a training and a testing set to develop the models with, the winning condition being to improve upon the RMSE of Cinematch by a 10% margin. Since Cinematch achieved an RSME of 0.9525 (on the testing set) the winning value would need to be lower than 0.85725, which was not achieved until 2009 (three years after the challenge began) with a matrix factorization model. Whereas Cinematch used was built upon a linear model (albeit highly tweaked and conditioned), the Netflix Prize competition demonstrated that matrix factorization approaches surpass classic nearest-neighbor techniques for product recommender systems.

This document details the data processes involved in the construction of such systems: data import, exploration, visualization and the subsequent model development. Given that Cinematch was built upon the linear model concept, a linear model is to be built (with movie and user bias taken into account). However, the final goal of this project is to construct a recommendation system based upon matrix factorization, contrasting its performance to that of a linear model while aiming to surpass the RMSE of Cinematch.

2. Setup and exploration

2.1. Project libraries

As previously stated, this project uses the R programming language along with several libraries which, as is the norm in most non-basic R projects, are often required to complement R with additional (and specific) functions. The following code snippet installs all of the required libraries if they are not installed already (through the use of conditionals and the built-in require() function), which are presented in alphabetical order for the sake of convenience (note that the installation itself is called by the install.packages() function).

```r
# Installing the required libraries (if they are not installed already)
if(!require(caret)) {
  install.packages("caret")
}
if(!require(cowplot)) {
  install.packages("cowplot")
}
if(!require(data.table)) {
  install.packages("data.table")
}
if(!require(dplyr)) {
  install.packages("dplyr")
}
if(!require(ggplot2)) {
  install.packages("ggplot2")
}
if(!require(ggthemes)) {
  install.packages("ggthemes")
}
if(!require(lubridate)) {
  install.packages("lubridate")
}
if(!require(Metrics)) {
  install.packages("Metrics")
}
if(!require(recosystem)) {
  install.packages("recosystem")
}
if(!require(scales)) {
  install.packages("scales")
}
if(!require(stringr)) {
  install.packages("stringr")
}
if(!require(tibble)) {
  install.packages("tibble")
}
if(!require(tidyr)) {
  install.packages("tidyr")
}
```

Installing a given package does not mean said package (and its associated functions) are yet ready to be used. To do so, it needs to be properly loaded into the R workspace, for which there exists the built-in library() function.

The following code snippet makes use of said function to import/load all of the project's required libraries (once again, in alphabetical order for the sake of convenience).

```r
# Loading the required libraries
library(caret)
library(cowplot)
library(data.table)
library(dplyr)
library(ggplot2)
library(ggthemes)
library(lubridate)
library(Metrics)
library(recosystem)
library(scales)
library(stringr)
library(tibble)
library(tidyr)
```

A brief comment about these libraries:

- caret: this library's name is short for "Classification And Regression Training", being arguably the most popular R package for the matter. It contains a variety of tools to streamline machine learning tasks, and as such many of the functions used throughout this document are from within this package.
- cowplot: this library eases the construction of publication-quality figures.
- data.table: a high-performance library to work, manipulate and operate with dataframes.
- dplyr: arguably the most popular R package for data manipulation. It provides a consistent set of tools for the matter, enabling data manipulation in an intuitive, user-friendly way. Data analysts typically use dplyr to transform existing datasets into a fitting format (dataframes are usually preferred) for data analysis, data exploration and data visualization tasks.
- ggplot2: the most popular library for data visualization. It can greatly improve the quality and aesthetics of one's graphics, being not only highly customizable but also remarkably efficient.
- ggthemes: ggplot2 and its official extension support allows developers to easily create their own tools and presets. ggthemes is built upon that support to provide additional themes for ggplot2's graphics.
- lubridate: a library designed to simplify (and speed-up) time-related tasks and calculations.
- Metrics: this library is built around evaluation metrics, providing functions to simplify their calculation.
- recosystem: this library is built around matrix factorization, bundling a collection of functions to simplify the process at hand.
- scales: this library was included to properly scale some plot's axis to better visualize and understand the data being plotted.
- stringr: this library a set of functions designed to make working with strings as easy as possible.
- tibble: this library re-imagines the dataframe format, tidying it up leading to a cleaner solution.
- tidyr: this library simplifies the process of tidying data so that it can be easily evaluated/studied and standardizing it in format that most functions accept.

Note that some of these libraries are also included in the tidyverse package. However, I rather understand the use-case scenario of each instead of relying on library bundles.

2.2. The MovieLens dataset

In ideal circumstances, this project would have used the very same dataset used in the Netflix prize. However, its size is way too large and demanding for the circumstances at hand so the exercise that entails this project describes and details the construction of a movie recommender system using the MovieLens dataset.
Said dataset is provided by GroupLens (a research lab within the Univeristy of Minnesota) and holds 27 million ratings applied to 58,000 movies by 280,000 users. Once again, that might be too much to ask of most basic personal computers and would imply seriously long training times, so only a small subset of the whole dataset will be used given the sheer size of the latter (particularly, the exercise makes use of the 10M MovieLens subset).

The dataset to work upon needs to be downloaded from its hosting URL, for which two R built-in functions are to be used: tempfile() creates an empty placeholder upon which to load the file's content and, secondly, download.file() is the one function needed to download the file at hand (loading its content into the aforementioned placeholder).
It is worth noting that the downloaded file is compressed and therefore found in ZIP format, otherwise the readr function library could have been used to read and load the dataset's content.

```r
# The dataset ZIP file is downloaded
dl <- tempfile()
download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)
```

- ratings.dat: a four column dataset containing:
  - A user ID
  - A movie ID
  - The score with which said user rated the movie
  - A timestamp
- movies.dat: a three column dataset containing:
  - A movie ID
  - The movie title
  - The movie genres
- tags.dat: a tag-related dataset that will not be used for this exercise.

These files are to be worked with individually, just as is showcased in the next code snippet. The data has to be read properly, and there are various functions to perform this loading process like fread(), from the data.table library, and str_split_fixed(), from the stringr library. Each function has its own particular syntax, so informing oneself regarding their behavior, arguments and intricacies is recommended - their related RDocumentation sites (already hyperlinked) are a good starting point.

Both ratings.dat and movies.dat have their column data separated with a "::" string, which needs to be properly specified in each of these functions. The column names should also be defined to facilitate indexing-related functions.

Both fread() and str_split_fixed() are showcased in the following code snippet, the former with the ratings.dat dataset and the latter with movies.dat (beware the computing time).

```r
# The ratings.dat is imported into the workspace
ratings <- fread(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))
class(ratings)
## [1] "data.table" "data.frame"
head(ratings)
```

userId	movieId	rating	timestamp
1	122	5	838985046
1	185	5	838983525
1	231	5	838983392
1	292	5	838983421
1	316	5	838983392
1	329	5	838983392

```r
# The movies.dat is imported into the workspace
movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")
class(movies)
## [1] "matrix" "array"
head(movies)
##      movieId title
## [1,] "1"     "Toy Story (1995)"
## [2,] "2"     "Jumanji (1995)"
## [3,] "3"     "Grumpier Old Men (1995)"
## [4,] "4"     "Waiting to Exhale (1995)"
## [5,] "5"     "Father of the Bride Part II (1995)"
## [6,] "6"     "Heat (1995)"
##      genres
## [1,] "Adventure|Animation|Children|Comedy|Fantasy"
## [2,] "Adventure|Children|Fantasy"
## [3,] "Comedy|Romance"
## [4,] "Comedy|Drama|Romance"
## [5,] "Comedy"
## [6,] "Action|Crime|Thriller"
```

The head() function outputs the structure of each set and its very first rows, whereas class() returns their class. Note that the fread() function outputs a dataframe whereas str_split_fixed() outputs a matrix. Joining them together requires them both to be of equivalent format and, since most of the relevant functions work best (or at all) with dataframes, the movies matrix will be converted to a dataframe using the as.data.frame() function.

Note that the %>% operator corresponds to the pipe expression from the dplyr package, which allows functions to be easily chained. In the following code snippet, the pipe operator is used to easily apply the the mutate() function (from that same dplyr package) upon the movies dataframe to properly "label" the columns' content, changing their classes accordingly through the as.numeric() and as.character() functions. Beware that the mutate() function is usually used to create new columns for a dataframe although, in this case, it is used to overwrite existing columns (using existing columns' names).

```
# The movies object is converted into a dataframe
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                            title = as.character(title),
                                            genres = as.character(genres))
class(movies)
## [1] "data.frame"
head(movies)
```

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

The head() output content is unchanged, but now class() showcases that movies has been properly converted into a dataframe. Since both movies and ratings are now equal in format, they can be fused together with the left_join() function from the dplyr package, just as is illustrated in the following code snippet. Note that an array needs to be specified so that the two dataframes can be joined (in this case, they are joined by the "movieId" column array).

```
# Ratings and movies dataframes are joined by "movieId"
movielens <- left_join(ratings, movies, by = "movieId")
class(movielens)
## [1] "data.table" "data.frame"
head(movielens)
```

userId	movieId	rating	timestamp	title	genres
1	122	5	838985046	Boomerang (1992)	Comedy|Romance
1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
1	231	5	838983392	Dumb & Dumber (1994)	Comedy
1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller
1	316	5	838983392	Stargate (1994)	Action|Adventure|Sci-Fi
1	329	5	838983392	Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-Fi

The head() output showcases the newly constructed dataset' structure and content. Note that this object (movielens) is of class "data.table" and/or "data.frame", which is ideal (as previously stated) since most of the relevant functions work best (or at all) with dataframes.

2.3. Model evaluation

2.3.1. The validation set

To evaluate the system's effectiveness, a validation set is to be built using a small subset of the dataset at hand; it is upon this validation set that the RMSE will be computed. To split the movielens in a working set and the validation one the R built-in functions nrow() and sample() could be used to randomly select row indexes, dividing the dataset based on said indexes. This split is showcased in the following code snippet, where the validation set is built using 10% of the dataset's indexes; the missing 90% is used to create the working_set object, which is the set to work upon and develop the model with.
Note that the validation set is referred to as temp at this point since it is missing key steps without which it can be hardly considered a valid validation set. Finally, the tibble() function (from the tibble package) is used to create a table-like structure to showcase the different sets' length.

```r
# Using built-in R functions to create the working and validation sets
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
validation_index <- sample(1:nrow(movielens), 0.1*nrow(movielens))
working_set <- movielens[-validation_index,]
temp <- movielens[validation_index,]

tibble(Dataset = c("movielens", "working_set", "temp"),
       "Number of ratings" = c(nrow(movielens), nrow(working_set), nrow(temp)))
```

Dataset	Number of ratings
movielens	10000054
working_set	9000049
temp	1000005

Despite being perfectly valid, built-in R functions are not the usual/preferred approach for this sort of split. The caret package (short for Classification And Regression Training) contains a variety of tools to streamline many machine learning tasks (it is undoubtedly one of the most popular libraries for the matter) and among its many functions lies createDataPartition(), often used for the train-test split (which will be showcased later on) but equally ideal for the construction of the validation set. This function not only divides a given dataset based on a specified proportion but it does so while keeping the classification ratio constant within each set so that both have the same factor/cluster distribution as the original dataset, avoiding certain unfavorable scenarios where there might not be a sufficient amount of a given factor within the working set to allow the algorithm to develop a fitting model.

```r
# Do not run this code snippet, as it is only here for illustration purposes
library(caret)
createDataPartition(
  y,
  times = 1,
  p = 0.5,
  list = TRUE,
  groups = min(5, length(y))
)
```

The arguments of the function are as follows:

- y: a vector of outcomes.
- times: the number of partitions to create.
- p: the percentage of data that goes to training.
- list: whether to hold the results within a list or within a matrix.
- groups: if y (the vector of outcomes) is numerical, then this argument defines the number of breaks in the quantiles.

An important note is that, as opposed to the previous approach, the main argument of the function does not ask for the dataframe itself but uses its vector of outcomes instead (in the case of this exercise, said vector of outcomes is the ratings column/array). More information about the function, its behavior and its arguments can be found in its associated RDocumentation page.

The following code snippet showcases the subset construction using the createDataPartition() function. Note that, once again, the subsets' balance is of 90% for the working set and 10% for the validation one.

```r
# Using createDataPartition to create the working and validation sets
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)
working_set <- movielens[-test_index,]
temp <- movielens[test_index,]

tibble(Dataset = c("movielens", "working_set", "temp"),
       "Number of ratings" = c(nrow(movielens), nrow(working_set), nrow(temp)))
```

Dataset	Number of ratings
movielens	10000054
working_set	9000047
temp	1000007

The tibble() function is used once again to observe the sets' length. There might be minor differences with respect to the previous values (which used R built-in functions), but such differences are negligible.

Let's now ensure the user IDs and movie IDs in the testing set are also present in the training set. To do so, different versions of the join() function (such as the previously used left_join()) are used (in this case, semi_join() and anti_join()). Any and all information regarding these functions can be found in their associated RDocumentation page.

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

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

These procedures performed upon the temp object have lead to the construction of the validation set, which will be the one used to validate the models' effectiveness once developed. To do so, there are many different metrics (although it has already been stated that the main one to focus on will be the RMSE since such was the metric used for The Netflix Price).

2.3.2. Evaluation metrics

Within any and all machine learning projects, measure the performance of the developed models is of key importance. There are many metrics to quantify the predictions' error, which most of the approaches use to evaluate the model's effectiveness. Such error is usually obtained by comparing the models' predicted values with the actual real/observed results (in this exercise, those have been stored in the validation object previously constructed). Most of these metrics (the most relevant and popular ones) revolve around measuring the distance between both values (the closer the points the higher the success).

2.3.2.1. Mean absolute error The Mean Absolute Error (MAE) averages the absolute difference between the model's predicted value and the actual real/observed value. Given the linearity of the metric, all errors are equally weighted (e.g. an error of value 4 is twice as bad as an error with value 2).
Its mathematical formula goes as follows:

$$MAE = \frac{1}{N}\sum_{i}^{N} |\hat{y}_i - y_i|$$

Where N equals the number of observations/ratings, $\hat{y}_i$ is the predicted value and y_i is the real/observed result.

Computing the mean absolute error in R is a simple task thanks to the mae() function from the Metrics library, which contrasts actual and predicted values (arguments for the function) through the mathematical expression just showcased in order to evaluate the predicted values' relative success.

2.3.2.2. Mean squared error (MSE) The Mean Squared Error (MSE), also known as the Mean Squared Deviation (MSD), averages the squared error of the model's predictions. The squared nature of this metric implies that the further the error lies from 1, the heavier it weights upon the MSE metric.

- If an error larger than one doubles its value, its MSE multiplies its value times four.
- If an error lower than one halves its value, its MSE is divided by four

MSE mathematical formula goes as follows:

$$MSE = \frac{1}{N}\sum_{u,i}^{N} (\hat{y}_{u,i} - y_{u,i})^2$$

where N equals the number of ratings, $y_{u,i}$ is the rating of movie i by user u and $\hat{y}_{u,i}$ is the prediction of movie i by user u.

Computing the mean squared error in R is a simple task thanks to the mse() function from the Metrics library, which contrasts actual and predicted values (arguments for the function) through the mathematical expression just showcased in order to evaluate the predicted values' relative success.

2.3.2.3. Root mean squared error (RMSE) The Root Mean Squared Error (RMSE), also known as the Root Mean Squared Error (RMSD), is the most popular metric to evaluate machine learning models. Concept-wise is similar to the previously discussed Mean Squared Error (MSE) but since RMSD square roots the result, its errors are measured in the same units as the values themselves. However, that does not exempt RMSE from its squared nature, meaning that the further the error lies from 1, the heavier it weights upon the RMSE metric:

- If an error larger than one doubles its value, its RMSE multiplies its value times four.
- If an error lower than one halves its value, its RMSE is divided by four.

RMSE mathematical formula goes as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{u,i}^{N}(\hat{y}_{u,i} - y_{u,i})^2}$$

Where N equals the number of ratings, $y_{u,i}$ is the observed/real rating assigned to a given movie i by a given user i and $\hat{y}_{u,i}$ is the model's predicted rating for that same movie and user.

Computing the root mean squared error in R is a simple task thanks to the rmse() function from the Metrics library, which contrasts actual and predicted values (arguments for the function) through the mathematical expression just showcased in order to evaluate the predicted values' relative success.
This is the most relevant metric since The Netflix Prize challenge was based on it; the goal of this exercise is to surpass Cinematch RSME value of 0.9525 and getting as close as possible to the winning RSME of 0.85725.

2.4. Data exploration

Understanding the data to work with (its structure, content, rating distribution…) can be key to build a better model. For example, linear models have a tendency to chase outliers in the training data, so observing whether or not there is a significant amount of outliers within said data or if, on the other hand, it follows a somewhat linear/regular pattern. This can help determine the approach to use or, as is the case in this exercise, the validity of the approaches already selected.

The core functions with which to begin the data exploration are the following, some of which have already been used throughout previous chapters:

- dim(): this function returns the dimensions of an object. In this exercise, those objects are the sets to work with and the output of the function returns their number of rows and columns (respectively). Note that there exist functions that return the rows and columns individually (nrow() and ncol() respectively).
- class(): this function returns the class of an object. Has been used to ensure that the sets to work with are of data.frame class.
- str(): this function returns a summary of a dataset' structure, showcasing each column's class and first items.
- head(): this function returns the first rows of a dataset, showcasing part of its content as well as its structure.
- summary(): this function returns relevant statistical parameters for all columns of a given dataset. Those parameters are:
  - The minimum and maximum values

  - The first and third quartiles
  - The median and the mean.

Given that the working_set subset is the one to be used to construct the model, the following exploration will be performed upon it. The movielens object would also be valid, but it is arguably better to restrict the exploratory analysis to the data with which the model will be built.

Let's first evaluate the dataset's dimensions and class using both dim() and class():

```
dim(working_set)
## [1] 9000055       6
class(working_set)
## [1] "data.table" "data.frame"
```

As can be seen, the dataset's dimensions show that the set, of class data.table, data.frame, is made of 9000055 rows and 6 columns. The class of those columns, as well as their first items and the overall structure of the dataset, can be appreciated through the use of the str() function.
Note that the argument vec.len = 2 is used in order to limit the number of items per column/feature (otherwise the output would be too cluttered).

```
str(working_set, vec.len = 2)
## Classes 'data.table' and 'data.frame':   9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 ...
##  $ movieId  : num  122 185 292 316 329 ...
##  $ rating   : num  5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 ...
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

From the output of the function, it can be seen that:

- The userId column holds the integers used to identify and represent all of the users who have rated a movie.
- The movieId column holds the numeric values, each matching a different movie (the movie title associated with each of these values can be seen in the title column).
- The rating column holds the numeric values associated with the rating with which a given user (the one from that same row) has rated a given movie (the one from that same row). It is based upon a star-based system, meaning that a 5-star rating (5 value) is the highest possible outcome, but the first items of the column do not cover the whole rating range/spectrum.
- The timestamp column holds the integer values which, if converted to an easy-to-read date format, should showcase the precise moment when the user of that same row rated the associated movie (as in, the one from that same row).
- The title column holds strings, character based results corresponding to the movie that the user from that same row has rated.
- The genres column holds strings, character based results corresponding to the genre or genres associated with the movie from that same row. Note that a single string is used even when multiple genres are used, with a "|" character separating each genre. That means that finding genres within this column might require the use of functions such as str_detect() from the stringr to find patterns within each string (e.g. finding Comedy in the "Comedy|Romance" string).

The function head() showcases the very first rows of the dataset "as is", being an accurate visual representation of the item and its content.

```
head(working_set)
```

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

Lastly, the function summary() answers some previous unresolved question: the range/spectrum of the user ratings.

```
summary(working_set)
##      userId          movieId          rating        timestamp
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08
##  Median :35738   Median : 1834   Median :4.000   Median :1.035e+09
##  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
##     title              genres
##  Length:9000055     Length:9000055
##  Class :character   Class :character
##  Mode  :character   Mode  :character
##
##
##
```

Not every column gets meaningful insight from the function's outcome, particularly identification and character based columns. The timestamp column hides its information due to its hard to interpret integer-based format (could mean something once converted to a proper time-based format) but the rating column showcases relevant statistics on the subject. For one, it can be seen that the ratings range/spectrum goes from half a star to the desirable 5-star rating, but some other insights can be observed such as an overall higher rating tendency (either users are more likely to rate movies they like or they like more movies than they do not).
This insightful information can be portrayed in an easier-to-read and easier-to-understand format, so the following chapter will revolve around doing so.

2.4.1. Ratings

Visualizing the ratings' distribution can uncover some hidden insights that might help to better understand user behavior. There are multiple ways to properly visualize the data in question, with the most common approaches being the chart-based information table and the graphic plot. The following code snippet showcases the former, counting the occurrences of each rating: the function group_by() groups the content of the working_set dataframe by rating, and the resulting object is then summarized so that only the rating count remains through the use of the summarize() and n() functions.

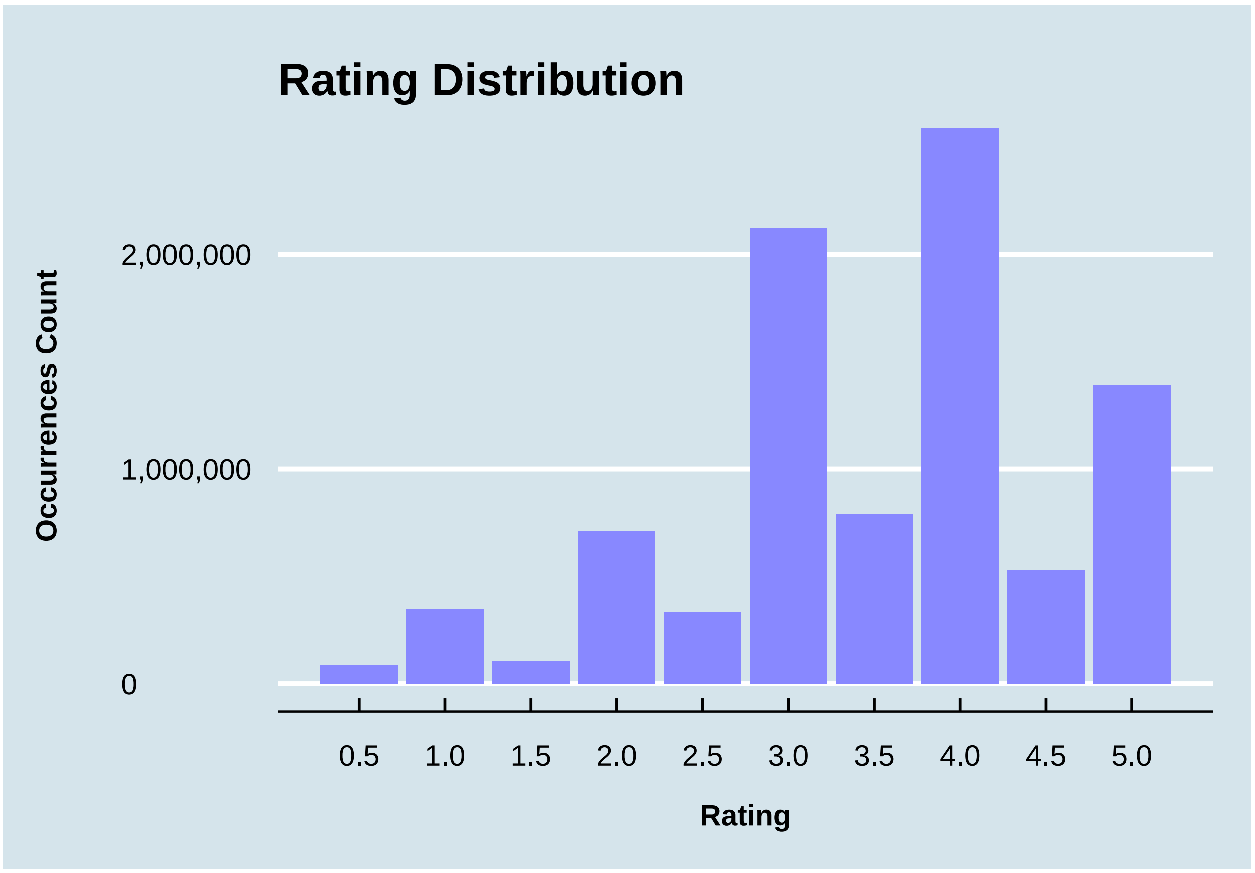
```
# Rating count
working_set %>%
  group_by(rating) %>%
  summarize(count = n())
```

rating	count
0.5	85374
1.0	345679
1.5	106426
2.0	711422
2.5	333010
3.0	2121240
3.5	791624
4.0	2588430
4.5	526736
5.0	1390114

It can be easily seen that users are more likely to use rounded ratings than half-stared ones. There is also an upwards trend that is a bit harder to appreciate with the chart (although the presence of six-number figures in the latter half of the table definitely indicates so), so plotting the information might help to evaluate how the ratings are distributed throughout the rating range/spectrum.

Plotting graphs in R is feasable with built-in tools/libraries, however the recommended approach usually requires the use of the ggplot() function from the ggplot2 package due to its versatile controls and potential tweaks. Another library which greatly complements the former is ggthemes, which includes some visually noteworthy themes to use alongside ggplot(). Note that the tidyverse package includes the dplyr library as well as both ggplot2 and ggthemes, so it is perfectly possible to just import said package into the workspace for every function to work as intended.

```
# Rating distribution plot
working_set %>%
  group_by(rating) %>%
  summarize(count = n()) %>%
  ggplot(aes(x = rating, y = count)) +
  geom_bar(stat = "identity", fill = "#8888ff") +
  ggtitle("Rating Distribution") +
  xlab("Rating") +
  ylab("Occurrences Count") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



The graph confirms what was previously observed: rounded ratings occur more often than half-stared ones. The upwards trend previously discussed is now perfectly clear, although it seems to top right between the 3 and 4 star ratings lowering the occurrences count afterwards. That might be due to users being more hesitant to rate with the highest mark for whichever reasons they might hold.

2.4.2. Timestamps

As previously stated when analyzing the output of the summary() function, the statistical values it returned might hold some (somewhat) meaningful insight. To properly observe and discuss those results, the timestamps held by the dataset have to be translated into a more understandable format. R provides the built-in function as.Date() for similar purposes, but such function is unable to properly interpret a date from an integer (even if an origin is fed). As was the case with ggplot(), there exists a library built specifically for this tasks and purposes: lubridate was and is designed to simplify time-related tasks and calculations. Its associated RDocumentation page showcases most (if not all) of its functions but, among them, the most relevant one for this exercise is as_datetime() as it properly transforms the integer values from the working dataset to objects of class "Date".

The README file provided with the dataset describes and details the different variables/parameters that construct the set itself, the timestamp among them. As stated in said file, the dataset's timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970; feeding the as_datetime() function with that information (through the non-optional origin argument) properly converts the timestamp array to the proper format.

The following code snippet uses the sample() function to showcase a selection of dates with the corrected format (after the use of the as_datetime() function).

```
# as_datetime() showcase
sample(as_datetime(working_set$timestamp, origin = "1970-01-01"), replace = TRUE, size = 20)
##  [1] "2000-03-06 18:22:15 UTC" "2005-07-20 20:25:32 UTC"
##  [3] "1997-01-28 22:56:24 UTC" "2008-04-12 08:30:06 UTC"
##  [5] "2005-08-25 13:32:34 UTC" "1999-04-09 12:58:35 UTC"
##  [7] "2007-09-22 22:49:32 UTC" "2000-11-23 08:15:41 UTC"
##  [9] "2006-10-02 19:44:50 UTC" "2004-01-08 15:54:24 UTC"
## [11] "1999-12-13 21:24:29 UTC" "1996-08-07 14:04:00 UTC"
## [13] "2008-03-06 13:23:56 UTC" "1999-12-12 20:17:36 UTC"
## [15] "2007-05-22 14:22:07 UTC" "2005-11-04 22:08:45 UTC"
## [17] "2001-10-28 13:22:30 UTC" "2005-02-19 04:56:36 UTC"
## [19] "2000-05-03 02:03:22 UTC" "1999-11-26 18:43:38 UTC"
```

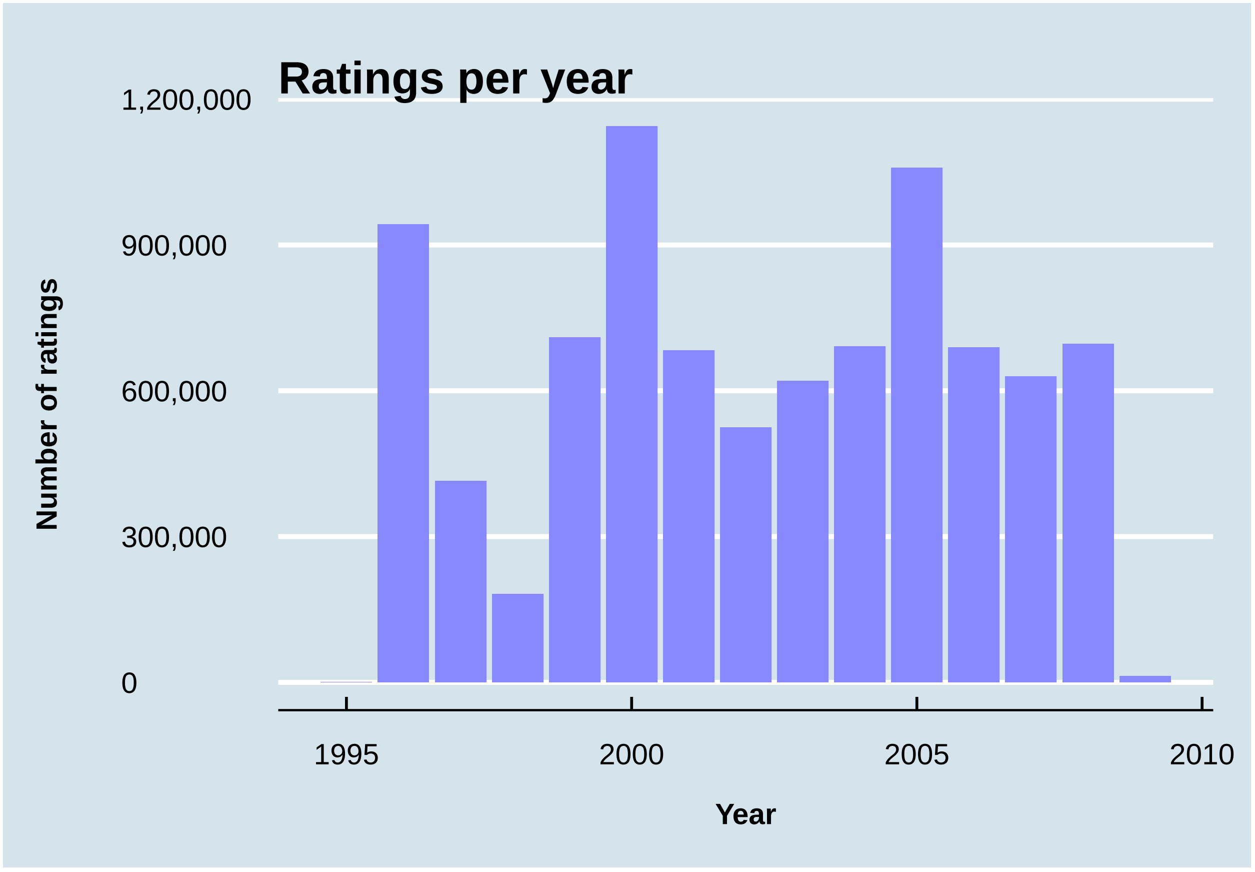
This array can be used to discern the number of ratings through time, which might indicate the growth of the platform or any given tendency in such regard. Doing so is possible with either a chart or with the use of a graphic plot, as was previously the case with the dataset ratings. Note that (in both cases) the dates are simplified to their year for convenience reasons (otherwise there would be too many information breaks), something easily achievable through the use of the year() function.

```
# Yearly rating count
working_set %>%
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) %>%
  group_by(year) %>%
  summarize(count = n())
```

year	count
1995	2
1996	942772
1997	414101
1998	181634
1999	709893
2000	1144349
2001	683355
2002	524959
2003	619938
2004	691429
2005	1059277
2006	689315
2007	629168
2008	696740
2009	13123

The results evidentiate that this subset of the MovieLens dataset holds no information prior to 1995 nor there is any rating post 2009. Despite being able to do so with this chart, finding relevant trends or patters is easier with a plot, for which the ggplot() function is once again appreciated.

```r
# Ratings per year plot
working_set %>%
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) %>%
  ggplot(aes(x = year)) +
  geom_bar(fill = "#8888ff") +
  ggtitle("Ratings per year") +
  xlab("Year") +
  ylab("Number of ratings") +
  scale_y_continuous(labels = comma) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```

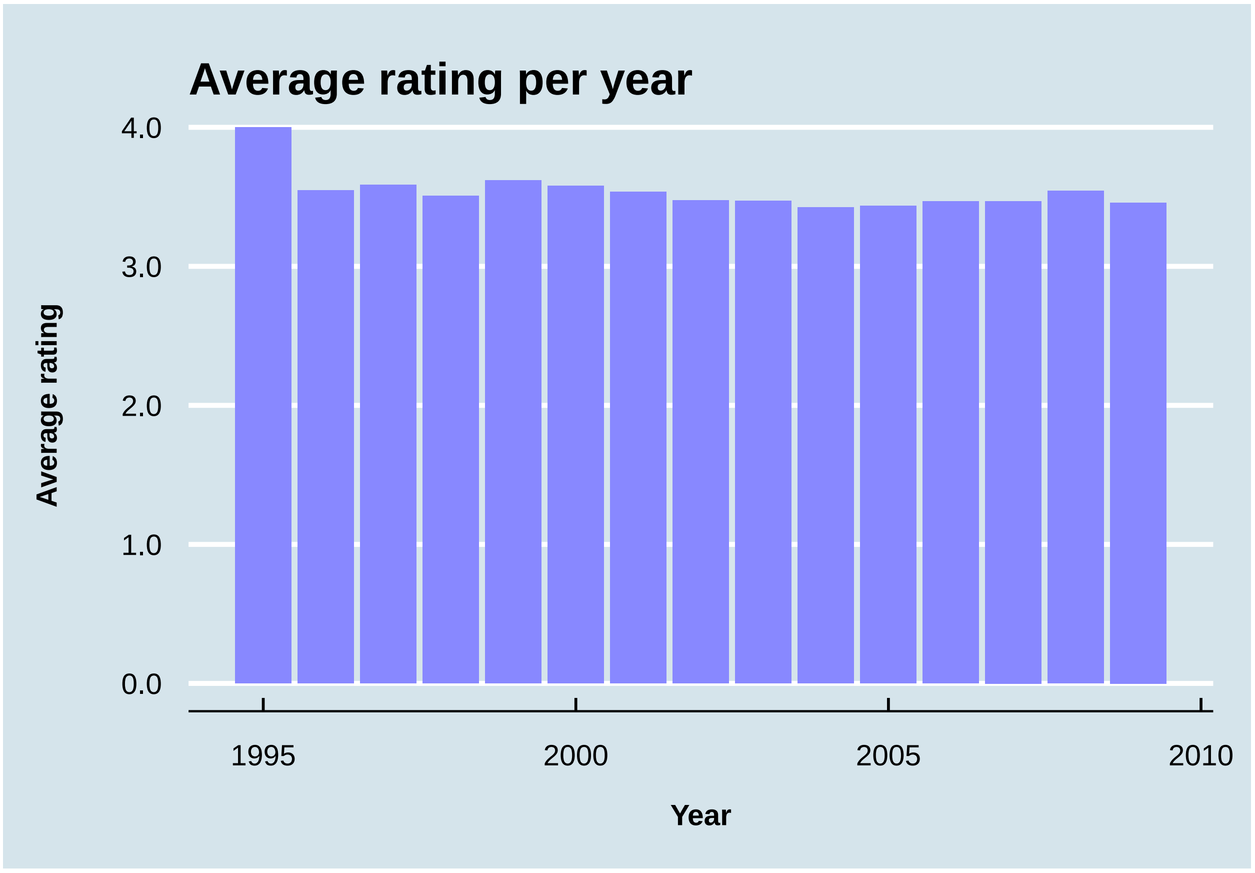


The plot suggests that from 1996 to 1998 the platform seemed to be declining in popularity, but 1999 broke that trend followed up by a notable popularity spike (of almost 1.2 million ratings) circa 2000. From that point on, the popularity of the platform appears to be somewhat stable with around 600.000 ratings per year (albeit a second notable spike in 2005 which surpassed the million ratings mark).

The dataset's timestamps have been used to determine the number of ratings per year, but they can also be used to uncover any potential trend in the rating scores (e.g. users might be more likely to rate movies higher in recent times than they were in the past).

To properly evaluate that information, an additional graph is to be plotted.

```r
# Average rating per year plot
working_set %>%
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) %>%
  group_by(year) %>%
  summarize(avg = mean(rating)) %>%
  ggplot(aes(x = year, y = avg)) +
  geom_bar(stat = "identity", fill = "#8888ff") +
  ggtitle("Average rating per year") +
  xlab("Year") +
  ylab("Average rating") +
  scale_y_continuous(labels = comma) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



Despite it being an informative plot, there is no particular insight to highlight other than the almost-negligible effect of time in the users' ratings.

2.4.3. Ratings per movie

Not all movies are rated equal and, as is to be expected, not all of them are equally popular. That is to mean that some movies have more ratings than others due to their overall popularity, maybe due to its critics' acclaim or perhaps just due to them being blockbusters. That distribution can be studied through either a table or a plot in order to observe any potential trends.

```
# Movie popularity count
working_set %>%
  group_by(movieId) %>%
  summarize(count = n()) %>%
  slice_head(n = 10)
```

movieId	count
1	23790
2	10779
3	7028
4	1577
5	6400
6	12346
7	7259
8	821
9	2278
10	15187

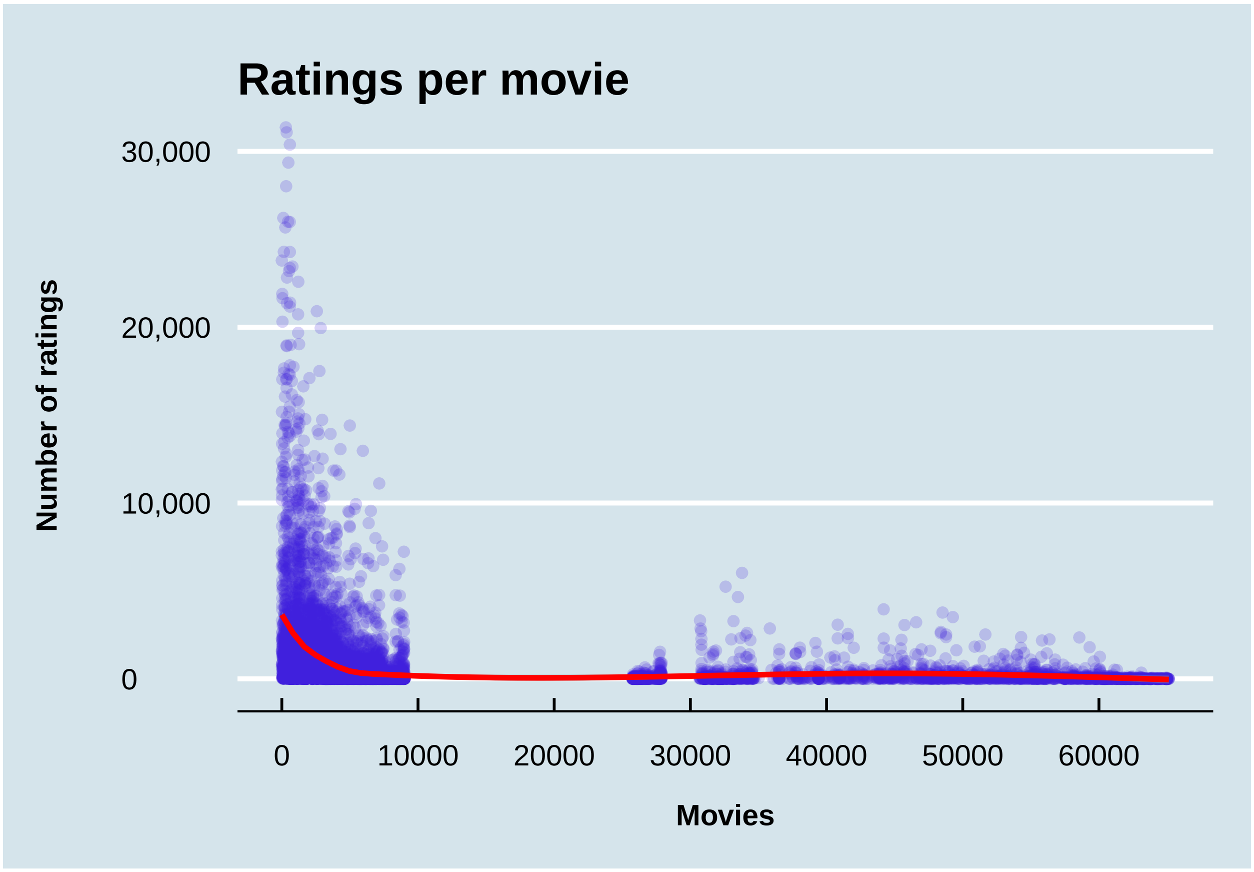
At first glance, the information chart shown above does not convey much. Not only is said chart extremely large (10677 rows, hence the use of the slice_head() function to reduce the output's length) but there is also much diversity in the count values so, to clear things out, the previously detailed summary() function can be used to observe the most meaningful statistical measures.

```
# Movie popularity summary
summary(working_set %>% group_by(movieId) %>% summarize(count = n()) %>% select(count))
##      count
##  Min.   :    1.0
##  1st Qu.:   30.0
##  Median :  122.0
##  Mean   :  842.9
##  3rd Qu.:  565.0
##  Max.   :31362.0
```

The minimum value of 1 is understandable, since there might exist some obscure movies that almost no one has watched, but it is worth noting that the most popular movie in the dataset has been rated 31362 times. This latter value seems to be quite an outlier, given the output returned by the summary() function within the previous code snippet where the highlight is arguably the third quartile value since it clearly showcases that 75% of the movies have been rated less than `quantile(working_set %>% group_by(movieId) %>% summarize(count = n()) %>% select(count), probs = 0.75, na.rm = TRUE)` times. With all of these values, it is somewhat feasible to evaluate a given movie's popularity (with respect to the rest of the movies of the dataset) based on its rating count.

To further study these results and uncover any potential hidden insight, a graphic plot is highly recommended (and almost mandatory). As was the case in previous chapters, ggplot() and its auxiliary functions are the ideal tools to do so.

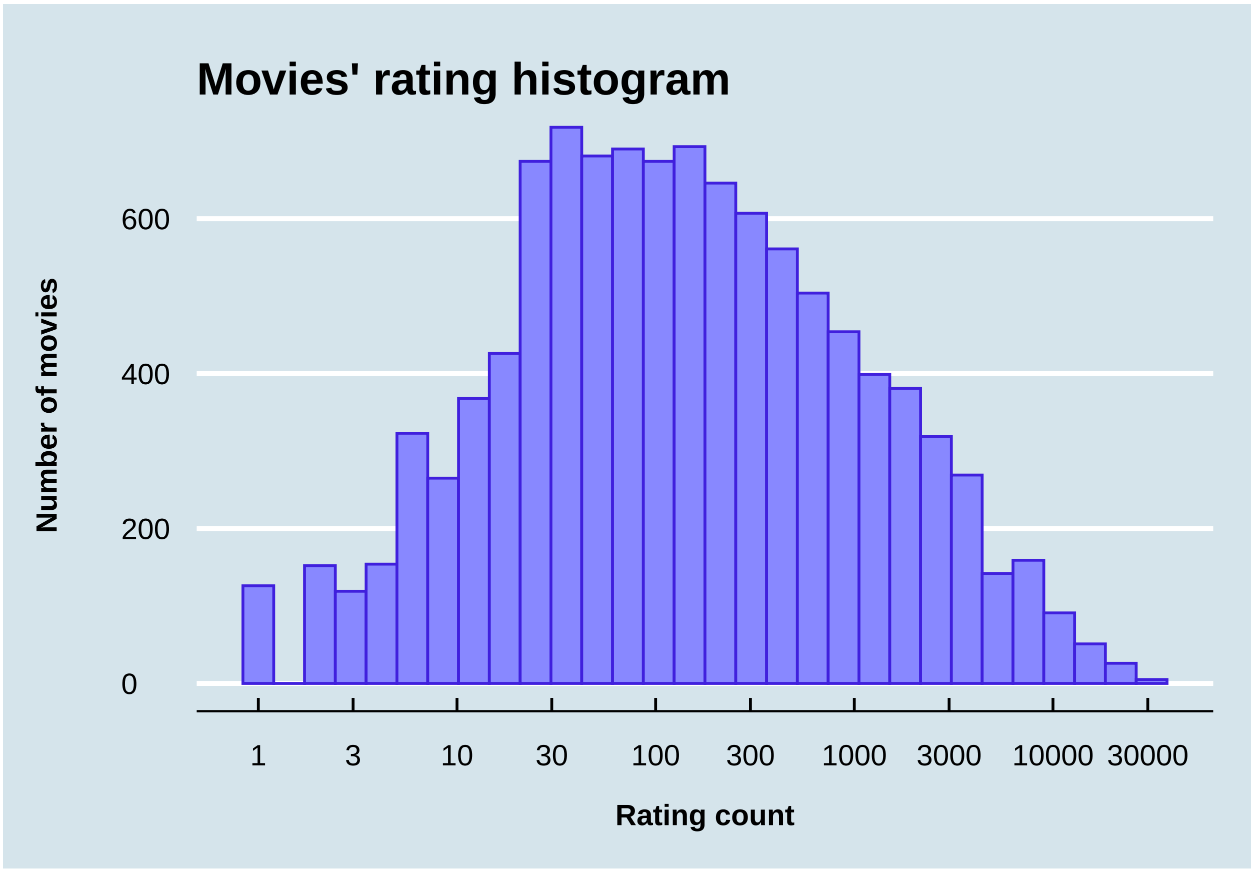
```r
# Ratings per movie plot
working_set %>%
  group_by(movieId) %>%
  summarize(count = n()) %>%
  ggplot(aes(x = movieId, y = count)) +
  geom_point(alpha = 0.2, color = "#4020dd") +
  geom_smooth(color = "red") +
  ggtitle("Ratings per movie") +
  xlab("Movies") +
  ylab("Number of ratings") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



The plot clearly shows that there is a significant gap in movie identification numbers between the 10000 and 25000 marks, although that might be the least meaningful insight to take from the plot. Movie popularity is now visually observable, proving that most of the points/movies have few ratings (the summary() output indicated that most movies have been rated less than `quantile(working_set %>% group_by(movieId) %>% summarize(count = n()) %>% select(count), probs = 0.75, na.rm = TRUE)` times) and, as a matter of fact, older movies in the dataset (first entries, identifiable by their low movieId number) have been rated more times than recent flicks (which makes sense give that users have had more time to rate said films).

A smooth density line/curve has been plotted since scatterplots can be misleading if the number of points (users in this case) is on the larger side since the points often overlap each other making it impossible to determine the denser regions of the plot, which fortunately are clearly highlighted by smooth density lines/curves. However, said information is better showcased with a histogram (for which the function geom_histogram() from the ggplot2 package is ideal.

```r
# Movies' rating histogram
working_set %>%
  group_by(movieId) %>%
  summarize(count = n()) %>%
  ggplot(aes(x = count)) +
  geom_histogram(fill = "#8888ff", color = "#4020dd") +
  ggtitle("Movies' rating histogram") +
  xlab("Rating count") +
  ylab("Number of movies") +
  scale_y_continuous(labels = comma) +
  scale_x_log10(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



This histogram better conveys the information provided by the summary() function, where the quantiles' values state that half the movies are rated between 30 and 565 times.

2.4.4. Ratings per user

As was and is the case with movies, not all users see the same amount of activity within the platform: the most active users might have rated a significant portion of the dataset's filmography whereas the least active users might have rated his/her favorite movie and no more. To evaluate this information, let's mimic the previous chapter approach.

```
# User rating count (activity measure)
working_set %>%
  group_by(userId) %>%
  summarize(count = n()) %>%
  slice_head(n = 10)
```

userId	count
1	19
2	17
3	31
4	35
5	74
6	39
7	96
8	727
9	21
10	112

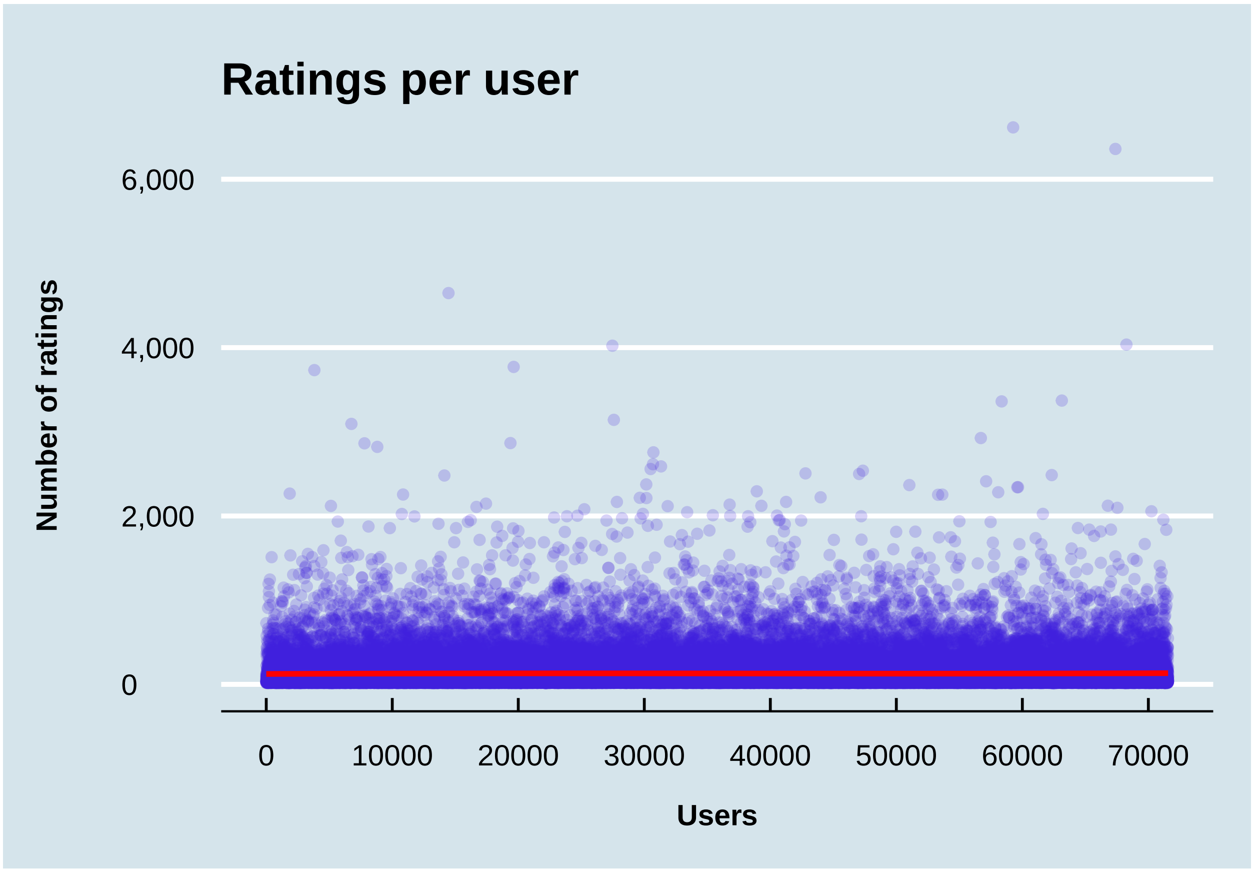
Once again, the table is extremely large (69878 rows, hence the use of the slice_head() function to reduce the output's length) and does not convey much by itself; the function summary() provides the statistical measures needed to properly interpret it.

```
# User rating summary
summary(working_set %>% group_by(userId) %>% summarize(count = n()) %>% select(count))
##      count
##  Min.   :  10.0
##  1st Qu.:  32.0
##  Median :  62.0
##  Mean   : 128.8
##  3rd Qu.: 141.0
##  Max.   :6616.0
```

The minimum value of 10 is higher than I personally expected, since all users have at least rated as many movies (goes a bit beyond rating your favorite flick). The most active user in the dataset has rated 6616 movies, although once again that seems to be quite an outlier: the rest of the values returned by the function showcase so (particularly the third quartile, which indicates that 75% of the users have rated less than `quantile(working_set %>% group_by(userId) %>% summarize(count = n()) %>% select(count), probs = 0.75, na.rm = TRUE)` movies. These values can be used to evaluate the activity of an user, which might be a valuable indicator for a platform.

To further study these results and uncover any potential hidden insight, a plot is highly recommended (and almost mandatory). As was the case in previous chapters, ggplot() and its auxiliary functions are the ideal tools to do so.

```r
# Ratings per user plot
working_set %>%
  group_by(userId) %>%
  summarize(count = n()) %>%
  ggplot(aes(x = userId, y = count)) +
  geom_point(alpha = 0.2, color = "#4020dd") +
  geom_smooth(color = "red") +
  ggtitle("Ratings per user") +
  xlab("Users") +
  ylab("Number of ratings") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```

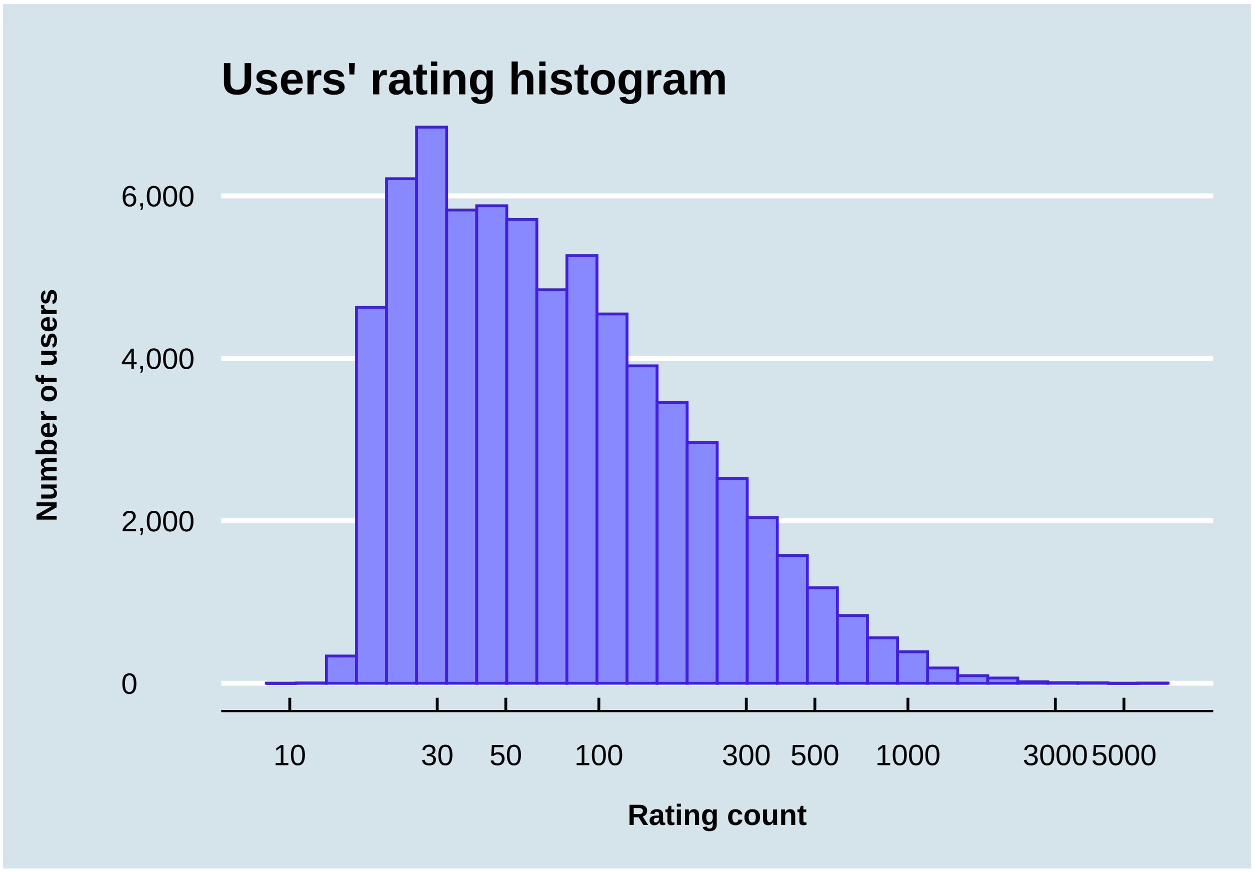


Once again, both a scatterplot and a smooth density line/curve have been drawn altogether to avoid misinterpretations. Judging by the scatterplot and given the sheer amount of users (which makes their points overlap), the plot seems to indicate that the average number of movies being rated by each user to be around the 500-1000 mark despite the fact that most users have rated less than `quantile(working_set %>% group_by(userId) %>% summarize(count = n()) %>% select(count), probs = 0.75, na.rm = TRUE)` movies (the data's third quartile). That is why the smooth density line/curve is mandatory: it clearly showcases the trend, which is to rate around NA movies.

Note that there is not any significant difference between older users and recent users (assumming longevity based on their userId) since both the scatterplot and the smooth density line stay relatively constant. It is also worth noting that this plot's outliers are more of an anomaly than the "Rations per movie" outliers, although that is but a minor insight of no consequence.

Mimicking the movies exploration, an histogram of the users and their rating count could provide some relevant information or better convey some of the discussed observations. For such plot, function geom_histogram() from the ggplot2 package is an ideal tool.

```
# Users' rating histogram
working_set %>%
  group_by(userId) %>%
  summarize(count = n()) %>%
  ggplot(aes(x = count)) +
  geom_histogram(fill = "#8888ff", color = "#4020dd") +
  ggtitle("Users' rating histogram") +
  xlab("Rating count") +
  ylab("Number of users") +
  scale_y_continuous(labels = comma) +
  scale_x_log10(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```

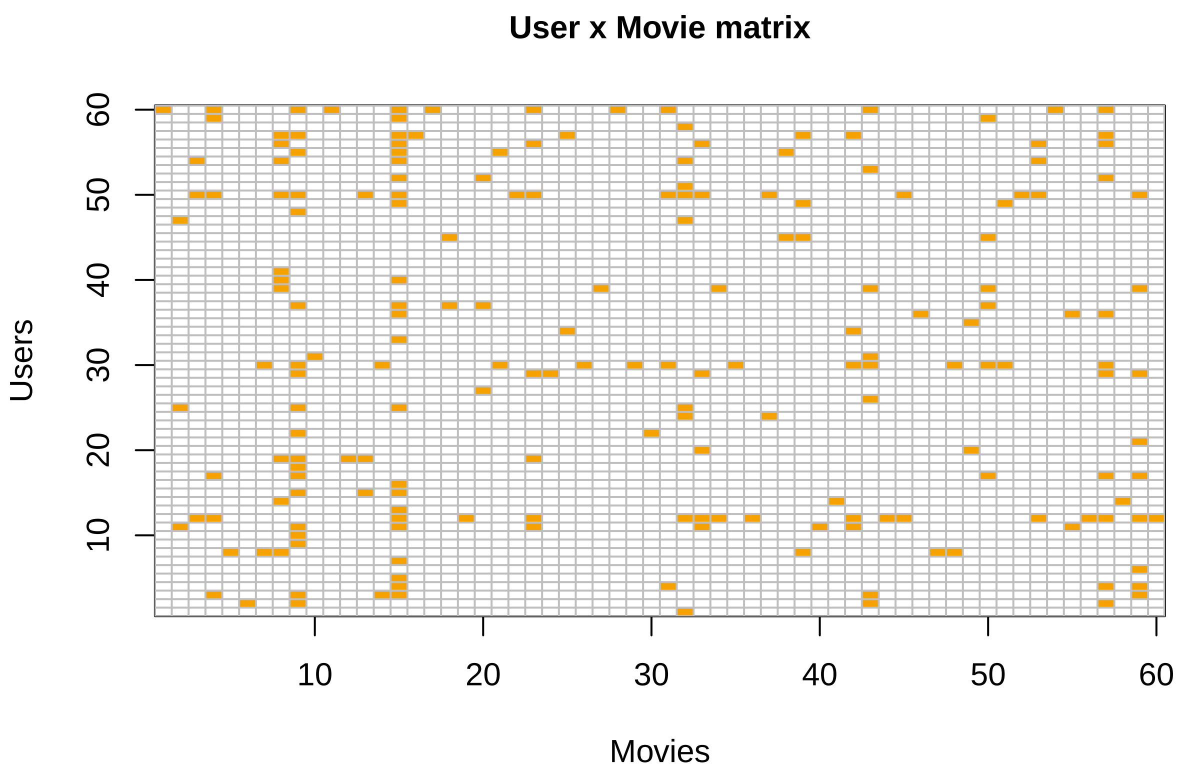


This histogram better conveys the information provided by the summary() function, where the quartiles' value state that half of the userbase rate between 32 and 141 movies.

Lastly, it is highly recommended to plot a matrix of users and movies where the matrix cells are highlighted if the corresponding user has rated the corresponding film (serving as a heatmap of sorts). This plot could also provide some relevant insights: for one, it would make it easier to observe the scarcity of ratings in the dataset, but it would also showcase which movies are more popular, which users are more active (although similar information can be subtracted from previous plots) as well as the overall activity of the platform. However, the sheer amount of both users and movies would make such graph too large for this document, so it would be wise to limit both movies and users to a manageable quantity (both the sample() and the select() functions accept a size-based argument, which can be used lo limit the variables at play).
Note that ggplot() is not used for this graph since the image() function is specifically design for these plots.

```
# User x Movie matrix construction
limit <- 60
user_movie_matrix <- working_set %>%
  filter(userId %in% sample(unique(working_set$userId), limit)) %>%
  select(userId, movieId, rating) %>%
  mutate(rating = 1) %>%
  spread(movieId, rating) %>%
  select(sample(ncol(.), limit)) %>%
  as.matrix() %>%
  t(.) # This function transposes the matrix

# Matrix plot
user_movie_matrix %>%
  image(1:limit, 1:limit,., xlab = "Movies", ylab = "Users") +
  abline(h = 0:limit + 0.5, v = 0:limit + 0.5, col = "grey") +
  title(main = list("User x Movie matrix", cex = 1, font = 2))
```



This graph visually showcases the most popular movies and the most active users within the selected subset (the most popular movies would be those with more highlighted cells in their respective column, and the most active users would be those with more highlighted cells in their respective rows). The density of highlighted cells could serve as an indicator of the platforms' overall activity but, unfortunately, the shown grid only covers a small portion of the dataset and any apparent insight could be misleading.

2.4.5. Genres

As previously stated, the genres column holds strings, character based results corresponding to the genre or genres associated with the movie from that same row. Note that a single string is used even when multiple genres are used, with a "|" character separating each genre. The following code snippet showcases the genres'

```
# Genres count
working_set %>%
  group_by(genres) %>%
  summarize(count = n()) %>%
  slice_head(n = 8)
```

genres	count
(no genres listed)	7
Action	24482
Action|Adventure	68688
Action|Adventure|Animation|Children|Comedy	7467
Action|Adventure|Animation|Children|Comedy|Fantasy	187
Action|Adventure|Animation|Children|Comedy|IMAX	66
Action|Adventure|Animation|Children|Comedy|Sci-Fi	600
Action|Adventure|Animation|Children|Fantasy	737

Setting aside the sheer amount of genres (797), the previous code snippet illustrates the biggest problem at hand: having multiple genres in a simple string means that finding genres within this particular column requires the use of functions such as str_detect() from the stringr package to find character patterns within each string (e.g. finding Comedy in the "Comedy|Romance" string). To understand the individual genres found within the dataset note that they are documented in its associated README file, being the following:

- Action
- Adventure
- Animation
- Children
- Comedy
- Crime
- Documentary
- Drama
- Fantasy
- Film-Noir
- Horror
- Musical
- Mystery
- Romance
- Sci-Fi
- Thriller
- War
- Western

Note that the previous code snippet's output showcases the "IMAX" genre although there is no mention of it in the dataset's README (although technically speaking IMAX should be more of a tag than a genre).

The str_detect() function can be used alongside the sum() to iterate through a vector of strings where the different genres have been defined and count each genre's ratings. The following code snippet does precisely so, using sapply() to apply the summation function to each individual element of the genres' vector (beware the computing time).

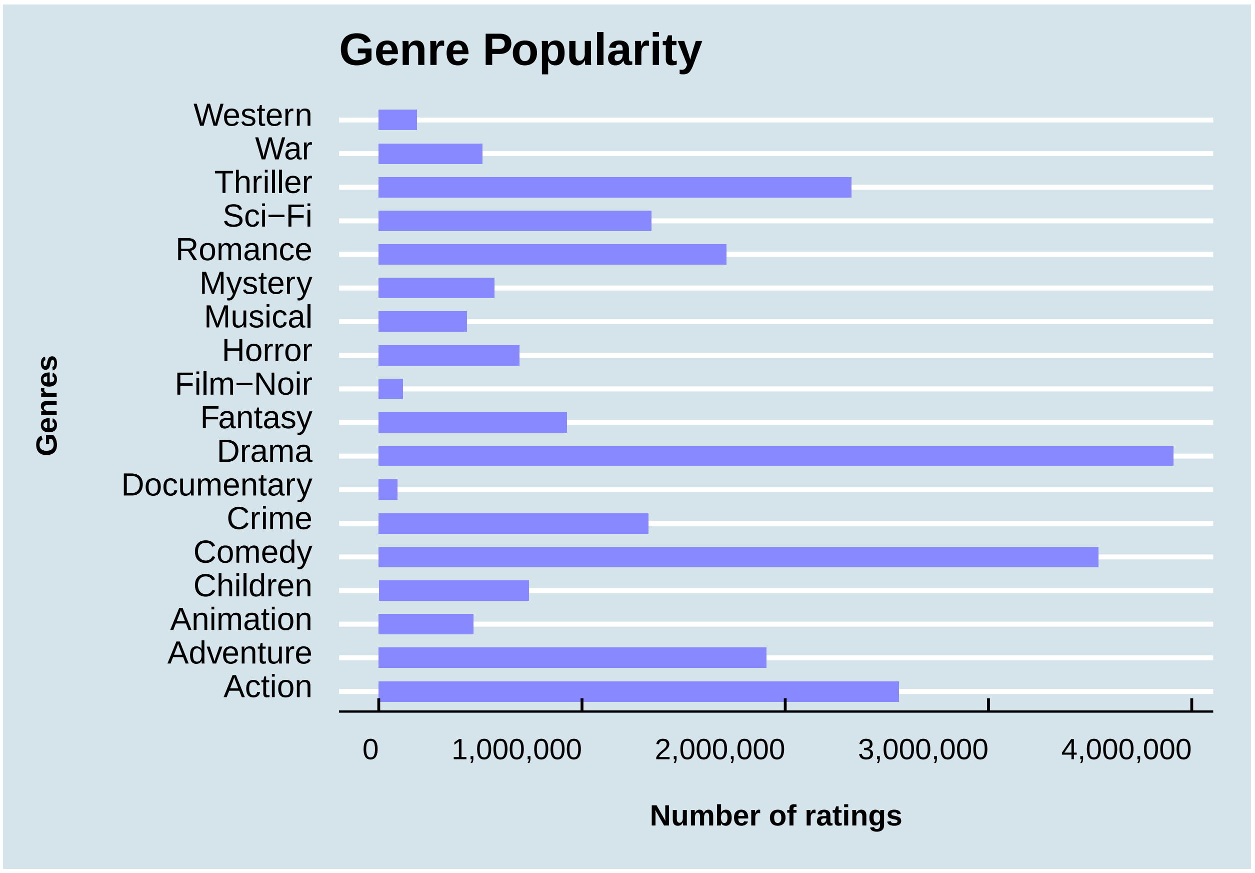
```
# Individual genres count
genres <- c("Action", "Adventure", "Animation",
            "Children", "Comedy", "Crime",
            "Documentary", "Drama", "Fantasy",
            "Film-Noir", "Horror", "Musical",
            "Mystery", "Romance", "Sci-Fi",
            "Thriller", "War", "Western")

genres_df <- data.frame(
  Genres = genres,
  Count = sapply(genres, function(x) {
    sum(str_detect(working_set$genres, x))
  })
)

print(genres_df)
##                  Genres   Count
## Action           Action 2560545
## Adventure     Adventure 1908892
## Animation     Animation  467168
## Children       Children  737994
## Comedy           Comedy 3540930
## Crime             Crime 1327715
## Documentary Documentary   93066
## Drama             Drama 3910127
## Fantasy         Fantasy  925637
## Film-Noir     Film-Noir  118541
## Horror           Horror  691485
## Musical         Musical  433080
## Mystery         Mystery  568332
## Romance         Romance 1712100
## Sci-Fi           Sci-Fi 1341183
## Thriller       Thriller 2325899
## War                 War  511147
## Western         Western  189394
```

The resulting dataset showcases the popularity of the different genres. To better visualize this information, there is no better tool than ggplot() and its auxiliary functions.

```r
# Genre popularity plot
genres_df %>%
  ggplot(aes(x = Count, y = Genres)) +
  ggtitle("Genre Popularity") +
  geom_bar(stat = "identity", width = 0.6, fill = "#8888ff") +
  xlab("Number of ratings") +
  ylab("Genres") +
  scale_x_continuous(labels = comma) +
  theme_economist() +
  theme(plot.title = element_text(vjust = 3.5),
        axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        axis.text.x = element_text(vjust = 1, hjust = 1, angle = 0),
        axis.text.y = element_text(vjust = 0.25, hjust = 1, size = 12),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



The plot clearly showcases the most popular genres in a visually appealing format, where the higher count is achieved by dramatic movies followed up by comedic, action-based and thriller flicks.

However, a genre's popularity is not necessarily correlated with its overall rating. To observe that, the previous approach can be somewhat replicated albeit modifying the constructed dataframe to reflect the average rating of a genre (based on each rating placed upon a movie of such kind) instead of merely counting occurrences.

```r
# Average rating for each genre
genres_df_2 <- data.frame(
  Genres = genres,
  Rating = sapply(genres, function(x) {
    mean(working_set[str_detect(working_set$genres, x)]$rating)
  })
)
print(genres_df_2)
##                  Genres   Rating
## Action           Action 3.421405
## Adventure     Adventure 3.493544
## Animation     Animation 3.600644
## Children       Children 3.418715
## Comedy           Comedy 3.436908
## Crime             Crime 3.665925
## Documentary Documentary 3.783487
## Drama             Drama 3.673131
## Fantasy         Fantasy 3.501946
## Film-Noir     Film-Noir 4.011625
## Horror           Horror 3.269815
## Musical         Musical 3.563305
## Mystery         Mystery 3.677001
## Romance         Romance 3.553813
## Sci-Fi           Sci-Fi 3.395743
## Thriller       Thriller 3.507676
## War                 War 3.780813
## Western         Western 3.555918
```

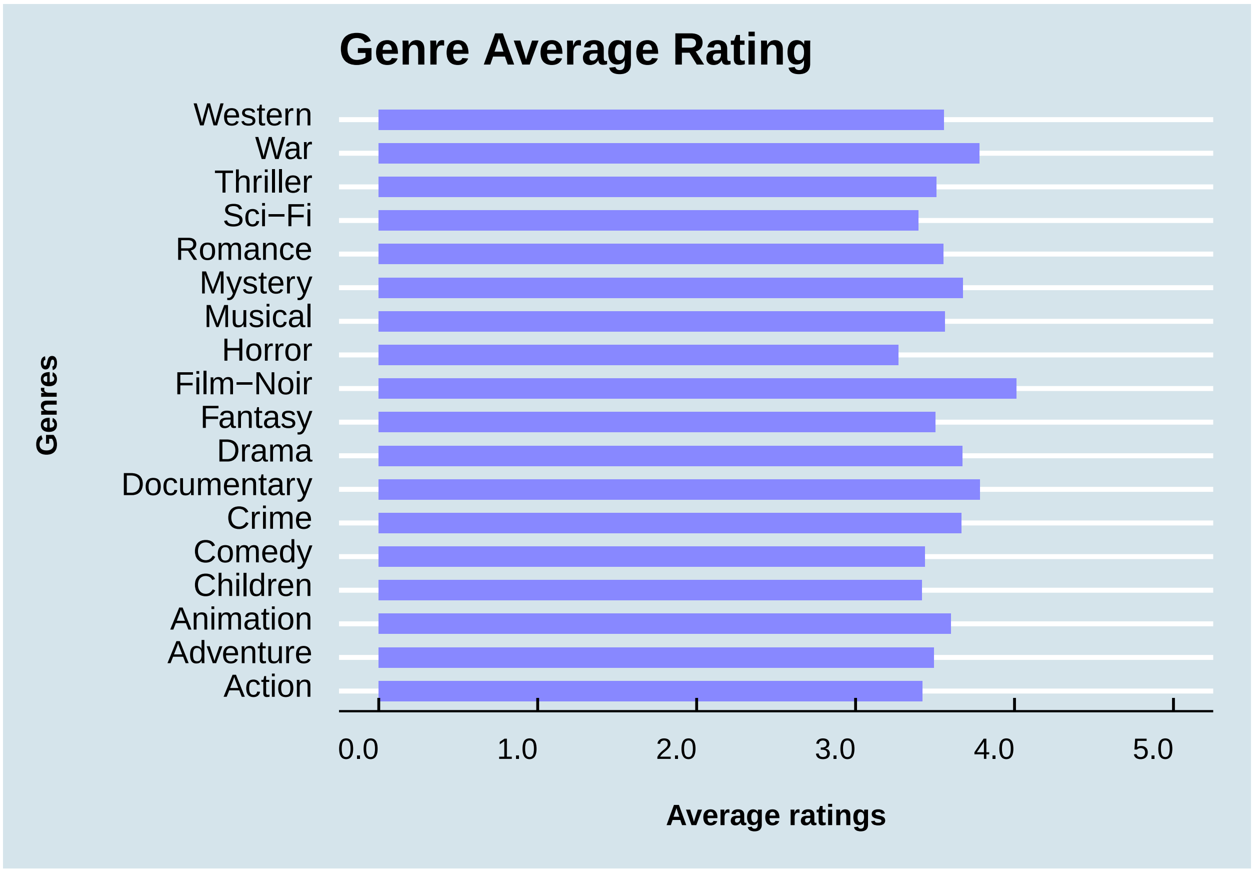
The printed results are surprisingly close to each other. To examine them further, the already detailed summary() function provides a straightforward solution.

```r
# Genre rating summary
summary(genres_df_2)
##     Genres              Rating
##  Length:18          Min.   :3.270
##  Class :character   1st Qu.:3.451
##  Mode  :character   Median :3.555
##                     Mean   :3.573
##                     3rd Qu.:3.671
##                     Max.   :4.012
```

The maximum score is obtained by the Film-Noir genre whereas the lowest score is obtained by the Horror genre, but there is not much difference between these values (4.0116 and 3.2698 respectively) nor are they far from the overall average rating of 3.5125.
However, it is worth noting that the best rated genre (Film-Noir) deviates from the average rating more than the lowest rated genre (Horror) does, and it does so by a significant margin (it is 2.06 times farther). That means that the best rated genre (Film-Noir) is more of an outlier than the lowest rated one, which is somewhat more of a merit.

Let's evaluate these results graphically through ggplot() and its auxiliary functions.

```r
# Genre rating plot
genres_df_2 %>%
  ggplot(aes(x = Rating, y = Genres)) +
  ggtitle("Genre Average Rating") +
  geom_bar(stat = "identity", width = 0.6, fill = "#8888ff") +
  xlab("Average ratings") +
  ylab("Genres") +
  scale_x_continuous(labels = comma, limits = c(0.0, 5.0)) +
  theme_economist() +
  theme(plot.title = element_text(vjust = 3.5),
        axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        axis.text.x = element_text(vjust = 1, hjust = 1, angle = 0),
        axis.text.y = element_text(vjust = 0.25, hjust = 1, size = 12),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



2.4.6. Preliminar Questions

Let's answer a few questions about the data to better understand its structure and content before proceeding with the system modelling.

How many rows and columns are there in the training dataset?

```
dim(working_set)[1] # Rows
## [1] 9000055
dim(working_set)[2] # Columns
## [1] 6
```

How many zeros were given as ratings in the training dataset?

```
sum(working_set$rating == 0.0)
## [1] 0
```

How many threes were given as ratings in the training dataset?

```
sum(working_set$rating == 3.0)
## [1] 2121240
```

How many different movies are in the training dataset?

```
dim(as.data.frame(table(working_set$movieId)))[1]
## [1] 10677
```

How many different users are in the training dataset?

```
dim(as.data.frame(table(working_set$userId)))[1]
## [1] 69878
```

How many movie ratings belong to the drama, comedy, thriller and romance genres (respectively) in the working_set dataset?

```
genres_quiz <- c("Drama", "Comedy", "Thriller", "Romance")
sapply(genres_quiz, function(x) {
  sum(str_detect(working_set$genres, x))
})
##    Drama   Comedy Thriller  Romance
##  3910127  3540930  2325899  1712100
```

Which of the following movies ("Forrest Gump", "Jurassic Park", "Pulp Fiction", "Shawshank Redemption" and "Speed 2: Cruise Control") has the greatest number of ratings?

```
ratings_quiz = c("Forrest Gump",
                 "Jurassic Park \\(1993",
                 "Pulp Fiction",
                 "Shawshank Redemption",
                 "Speed 2: Cruise Control")
sapply(ratings_quiz, function(x) {
  sum(str_detect(working_set$title, x))
})
##            Forrest Gump   Jurassic Park \\(1993            Pulp Fiction
##                   31079                   29360                   31362
##    Shawshank Redemption Speed 2: Cruise Control
##                   28015                    2566
```

What are the five most given ratings in order from most to least?

```
as.data.frame(table(working_set$rating)) %>% arrange(desc(Freq))
```

Var1	Freq
4	2588430
3	2121240
5	1390114
3.5	791624
2	711422
4.5	526736
1	345679
2.5	333010
1.5	106426
0.5	85374

3. System modelling

3.1. Train-test split

Machine learning's first step usual involves splitting the working set (in this exercise, that would be working_set) in a training set (which will be used to train the machine learning model so that it can learn from the supplied data in order to make successful predictions) and a testing set (which will be used to contrast the prediction with testing values and assess their success). Note that although similar concept-wise, the testing set is not the same as the validation set, and it is upon this latter one that the RMSE will be computed.

As was the case with the construction of said validation set, the train-test split can be achieved using the R built-in functions nrow() and sample() to randomly select row indexes from within the dataset and thus create both sets pseudo-randomly. These functions and the aforementioned subsets are showcased in the following code snippet, where two approaches are showcased: the first one uses a training index and the latter uses a testing index instead. Note that both approaches divide the working_set object with a balance of 90% for the training set and 10% for the testing one.

```r
# Train-test split using R built-in functions

# Approach 1: training index

set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
train_index <- sample(1:nrow(movielens), 0.9*nrow(movielens))
train_set <- movielens[train_index,]
temp_test_set <- movielens[-train_index,]

tibble(Dataset = c("movielens", "train_set", "temp_test_set"),
       "Number of ratings" = c(nrow(movielens), nrow(train_set), nrow(temp_test_set)))
```

Dataset	Number of ratings
movielens	10000054
train_set	9000048
temp_test_set	1000006

```r
# Approach 2: testing index

set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
test_index <- sample(1:nrow(movielens), 0.1*nrow(movielens))
train_set <- movielens[-test_index,]
temp_test_set <- movielens[test_index,]

tibble(Dataset = c("movielens", "train_set", "temp_test_set"),
       "Number of ratings" = c(nrow(movielens), nrow(train_set), nrow(temp_test_set)))
```

Dataset	Number of ratings
movielens	10000054
train_set	9000049
temp_test_set	1000005

There might be minor differences in the arrays' length due to the small differences in the approaches, but in practice both would work identically. However, as was the case with the validation set previously constructed, built-in R functions are not the usual/preferred approach: the createDataPartition() not only splits the working dataset into the training and testing subsets but it does so while keeping the classification ratio constant within each set so that both have the same factor/cluster distribution as the original dataset, avoiding certain unfavorable scenarios where there might not be a sufficient amount of a given factor within the training set to allow the algorithm to develop a fitting model.

Note that, as opposed to the previous approach, the main argument of the function does not ask for the dataframe itself but uses its vector of outcomes instead (in the case of this exercise, said vector of outcomes is the ratings column/array). The following code snippet showcases the subset construction using the createDataPartition() function with two different approaches: one using a training index and another using a testing index (the subsets' balance is unchanged, being 90% for the training set and 10% for the testing one).

```r
# Train-test split using createDataPartition

# Approach 1: training index

set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
train_index <- createDataPartition(movielens$rating, times = 1, p = 0.9, list = FALSE)
train_set <- movielens[train_index,]
temp_test_set <- movielens[-train_index,]

tibble(Dataset = c("movielens", "train_set", "temp_test_set"),
       "Number of ratings" = c(nrow(movielens), nrow(train_set), nrow(temp_test_set)))
```

Dataset	Number of ratings
movielens	10000054
train_set	9000050
temp_test_set	1000004

```r
# Approach 2: testing index

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)
train_set <- movielens[-test_index,]
temp_test_set <- movielens[test_index,]

tibble(Dataset = c("movielens", "train_set", "temp_test_set"),
       "Number of ratings" = c(nrow(movielens), nrow(train_set), nrow(temp_test_set)))
```

Dataset	Number of ratings
movielens	10000054
train_set	9000047
temp_test_set	1000007

Let's work upon the second approach and ensure the user IDs and movie IDs in the testing set are also present in the training set:

```r
# Make sure userId and movieId in the testing set are also in the training set set
test_set <- temp_test_set %>%
      semi_join(train_set, by = "movieId") %>%
      semi_join(train_set, by = "userId")

# Add rows removed from the testing set back into the training set set
removed <- anti_join(temp_test_set, test_set)
train_set <- rbind(train_set, removed)
```

With the training and testing sets prepared, the following chapters will focus on the modelling itself.

3.2. Random guessing

Random guessing is never an acceptable solution, although it can be used as a baseline of sorts to compare the accuracy and success of other approaches. To simulate a random guessing approach, a vector of all possible ratings is created (using the seq() function) which will be referred to as rating_range. The goal is to use that array to evaluate ratings' distribution and, in order to do that, a custom function is required: the following code snippet refers to it as guess_right(), and it accepts an array and a value as arguments to compute the ratio of items in the array being equal to the input value.

Theory implies that running an infinite amount of random guesses would result in a normal distribution of them. Running an infinite loop is obvious non-possible, but a Monte Carlo simulation of the with a large number would somewhat mimic that distribution. To do so, the following code snippet makes use of the replicate() function alongside the sample() function to randomly pick 1000 items from the train_set$rating vector across 10000 Monte Carlo simulations.
Each simulation creates an array of length 1000, which is then passed onto the sapply() function as sapply(rating_range, guess_right, i) (i being the array at hand). With those arguments, sapply() runs the previously created guess_right() function with the i array and the first item in the rating_range as arguments. That done, sapply() will repeat this process with the second item of the rating_range array, and so on. The result will be a vector of 10 distributions, one for each of the elements in the rating_range array (its distribution nature means that the sum of all these values would be equal to the unit).

The Monte Carlo simulation yields an object of dimension 10x10000 since 10000 iterations, each obtaining a length 10 array, can be represented as a matrix of 10000 columns and 10 rows. This matrix is then used in a for loop to evaluate the probability of each possible rating (through averages), yielding once again an array of length 10 (its distribution nature means that the sum of all these values would be equal to the unit) which the following code snippet refers to as guess_prob.

The prediction array would randomly pick ratings based on the probability distributions stored within the guess_prob() array. To do so, it uses the sample() function with the argument prob = guess_prob, randomly choosing ratings from the rating_range array until a vector of length (the number of rows in the validation set) is created (the argument size = nrow(validation) achieves that).

The following code snippet showcases all of this processes:

```r
# Random guessing model and predictions
rating_range <- seq(0.5, 5, 0.5)
guess_right <- function(x, y) {
  mean(y == x)
}

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

```r
simulation <- replicate(10000, {
  i <- sample(train_set$rating, 1000, replace = TRUE)
  sapply(rating_range, guess_right, i)
})

guess_prob <- c()
for(i in 1:nrow(simulation)) {
  guess_prob <- append(guess_prob, mean(simulation[i,]))
}

y_hat_random <- sample(rating_range,
                       size = nrow(validation),
                       replace = TRUE,
                       prob = guess_prob)
```

With the vector of predictions $\hat{y}$ already prepared, the next step is to evaluate its performance through the metrics already detailed in the evaluation metrics chapter. Note that the most relevant metric for this exercise is the RMSE and that Cinematch achieved an RSME value of 0.9525 (upon the testing set) whereas the winning value achieved an RMSE value lower than 0.85725

A tibble is built to showcase all of this information along with the random guessing performance. Doing so is streamlined by the use of the tibble() function from the homonymous package.

```r
# Evaluation tibble construction
evaluation <- tibble(Model = c("Cinematch", "The Netflix Prize", "Random guessing"),
                     MAE = c(NA, NA, Metrics::mae(validation$rating, y_hat_random)),
                     MSE = c(NA, NA, Metrics::mse(validation$rating, y_hat_random)),
                     RMSE = c(0.9525, 0.85725, Metrics::rmse(validation$rating, y_hat_random)))
print(evaluation)
## # A tibble: 3 x 4
##   Model               MAE   MSE  RMSE
##   <chr>             <dbl> <dbl> <dbl>
## 1 Cinematch         NA    NA    0.952
## 2 The Netflix Prize NA    NA    0.857
## 3 Random guessing    1.17  2.25 1.50
```

The obtained error values are way too high, as is to be expected from a random guess.

3.3. Linear model

Despite their simplicity and rigidness, linear (and logistic) models are important in the machine learning field since they are easily interpretable, fast (computing-wise) and form the basis of more complex deep learning neural networks. Concept-wise, these approaches just fit lines (or hyperplanes, depending on the dimensionality) through the training data but, depending on the scenario, the obtained predictions might be astoundingly accurate.

Note that there are non-negligible caviats that one needs to be aware of when working with linear models, since their advantages come at a cost. For one, linear models have a tendency to chase outliers in the training data (data exploration showcase the most relevant outliers within the working dataset), which might backfire when new data is supplied. However, linear models can be tweaked and tuned with different bias effects and later regularized to minimize the impact of its limitations.

The linear model to be built follows this mathematical expression:

$$\hat{y} = \mu + b_i + b_u + \epsilon_{u,i}$$

Where $\hat{y}$ is the prediction itself, μ is the average rating, b_i a movie-based bias, b_u a user-based bias and $\epsilon_{u,i}$ an inherent error term centered at 0 (thus negligible). Each of these elements builds upon each other, further developing the model, so the following paragraphs and chapters will go over their individual contributions.

3.3.1. Mean baseline

Taking aside movie and user bias, the linear model is merely the mean of the set's ratings (plus the inherent error term).

$$\hat{y} = \mu + \epsilon_{u,i}$$

In this simple linear model, the prediction array should be equal to μ in all of its indexes. To build such an array, the rep() function can be used to repeat μ a given number of times (in this case, that number should be equal to the amount of ratings in the validation set).
To add the model's results to the previously constructed tibble, the bind_rows() function is used, as showcased in the following code snippet.

```
# Mean baseline model construction
mu <- mean(train_set$rating)
y_hat_mean <- rep(mu, nrow(validation))

evaluation <- bind_rows(evaluation, tibble(Model = "Linear model (mean baseline)",
                                           MAE = Metrics::mae(validation$rating, y_hat_mean),
                                           MSE = Metrics::mse(validation$rating, y_hat_mean),
                                           RMSE = Metrics::rmse(validation$rating, y_hat_mean)))
print(evaluation)
## # A tibble: 4 x 4
##   Model                          MAE   MSE  RMSE
##   <chr>                        <dbl> <dbl> <dbl>
## 1 Cinematch                   NA     NA    0.952
## 2 The Netflix Prize           NA     NA    0.857
## 3 Random guessing              1.17   2.25 1.50
## 4 Linear model (mean baseline)  0.856  1.13 1.06
```

Despite the error being higher than ideal, note the significant improvement when comparing these recent results whith those of random guessing.

3.3.2. Movie bias

The linear model constructed with the ratings' mean (μ) can be improved by adding movie bias (b_i).

$$\hat{y} = \mu + b_i + \epsilon_{u,i}$$

Each individual movie has its own associated bias, as is showcased in the following code snippet. Note that two variables are summarized (via the summarize() function): one takes μ into account whereas the other one does not (the first one being an adjusted movie bias and the latter one being an isolated movie bias).
The slice_head() function is used to reduce the output's length since it is only needed to showcase the bias' values.

```r
# Bias per movie table
b_i <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu),
            b_i_isolated = mean(rating))
b_i %>% slice_head(n = 10)
```

movieId	b_i	b_i_isolated
1	0.4151725	3.927638
2	-0.3070658	3.205399
3	-0.3654817	3.146984
4	-0.6481659	2.864299
5	-0.4437933	3.068672
6	0.3028191	3.815284
7	-0.1538759	3.358589
8	-0.3778732	3.134592
9	-0.5146601	2.997805
10	-0.0866405	3.425825

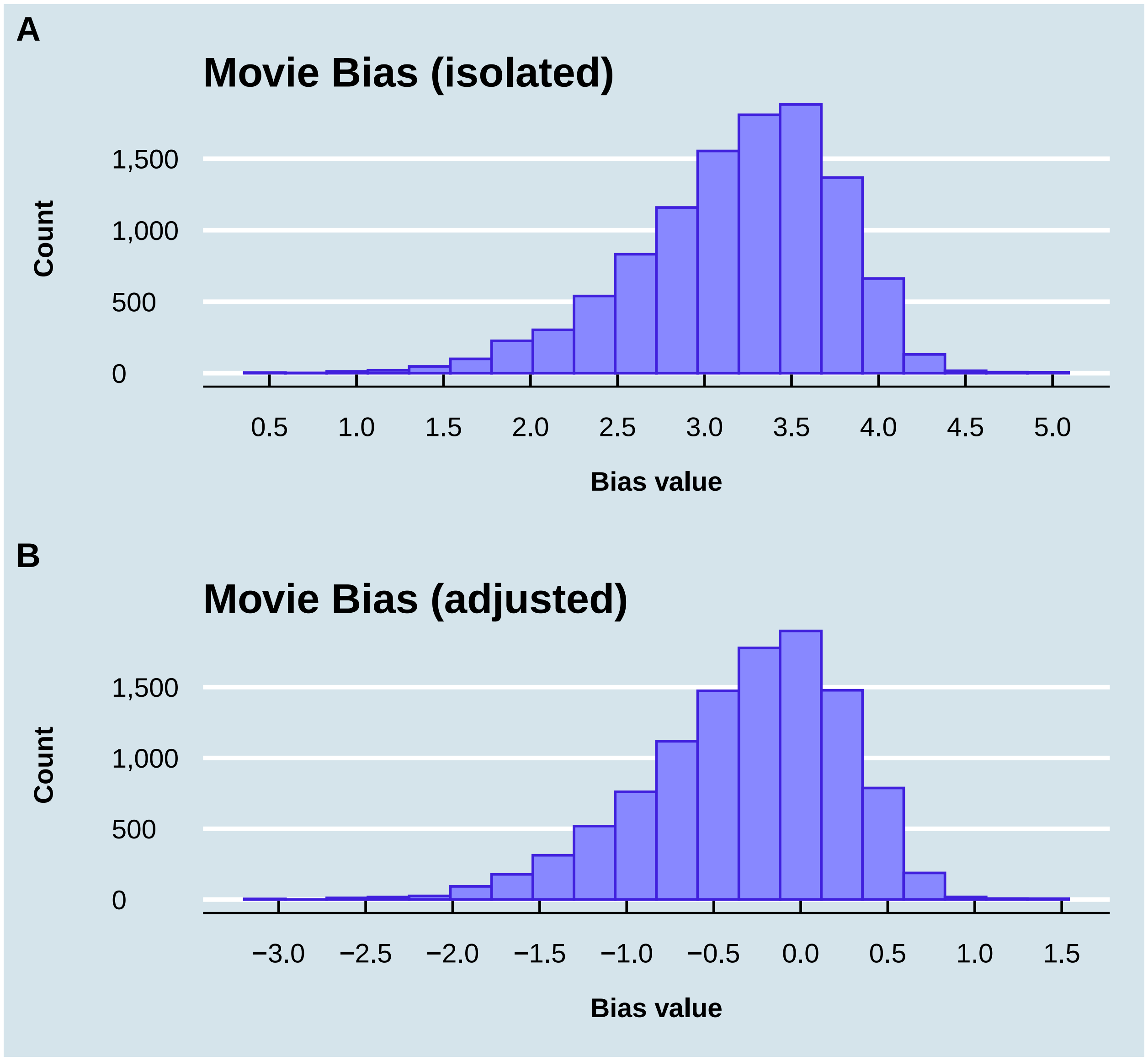
To properly evaluate the set's b_i values and their distribution, a plot built with ggplot() and its auxiliary functions is highly recommended. To contrast the adjusted and isolated bias (in order to observe and evaluate their values, distributions and differences), their associated graphs would benefit from being plotted in tandem. Doing so is streamlined by the function plot_grid() from the cowplot package, which as its name suggests, helps to place plots in a grid-like distribution so that they can be easily compared/contrasted.
Note that there are several other functions (from several other packages) that work alongside ggplot() to place plots againts one another, but this document covers cowplot due to the high customizability of its plot_grid() function.

The following code snippet showcases the aforementioned functions at play. Note that the plots at hand are placed one over the other (instead of side by side) to adequate the document's formatting, but that can be easily changed via the function's parameters (using `ncol = 2` instead of `nrow = 2` would distribute the plots horizontally instead of vertically).

```r
# Isolated movie bias plot
b_i_isolated_plot <- b_i %>%
  ggplot(aes(x = b_i_isolated)) +
  geom_histogram(bins = 20, fill = "#8888ff", color = "#4020dd") +
  ggtitle("Movie Bias (isolated)") +
  xlab("Bias value") +
  ylab("Count") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```

```r
# Adjusted movie bias plot
b_i_plot <- b_i %>%
  ggplot(aes(x = b_i)) +
  geom_histogram(bins = 20, fill = "#8888ff", color = "#4020dd") +
  ggtitle("Movie Bias (adjusted)") +
  xlab("Bias value") +
  ylab("Count") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))

# Both b_i plots are combined with plot_grid()
plot_grid(b_i_isolated_plot, b_i_plot, labels = "AUTO", nrow = 2)
```



The isolated bias plot showcases a normal albeit slightly left skewed distribution which spikes at 3.2678571 and possesses a mean value of 3.1917355. On the other hand, the adjusted bias plot showcases a normal albeit slightly left skewed distribution which spikes at -0.2446081 and possesses a mean value of -0.3207297. These two plots are highly similar in shape, although not quite identical.

Using the movie bias to build upon the previous linear model leads to the predictions and estimates showcased in the following code snippet:

```
# Linear model construction (mean + movie bias)
y_hat_b_i <- mu + validation %>%
  left_join(b_i, by = "movieId") %>%
  .$b_i

evaluation <- bind_rows(evaluation,
                        tibble(Model = "Linear model (mean + movie bias)",
                               MAE = Metrics::mae(validation$rating, y_hat_b_i),
                               MSE = Metrics::mse(validation$rating, y_hat_b_i),
                               RMSE = Metrics::rmse(validation$rating, y_hat_b_i)))
print(evaluation)
## # A tibble: 5 x 4
##   Model                              MAE    MSE  RMSE
##   <chr>                            <dbl>  <dbl> <dbl>
## 1 Cinematch                       NA     NA     0.952
## 2 The Netflix Prize               NA     NA     0.857
## 3 Random guessing                  1.17   2.25  1.50
## 4 Linear model (mean baseline)     0.856  1.13  1.06
## 5 Linear model (mean + movie bias)  0.738  0.891 0.944
```

The error reduction, product of taking movie bias into account, results in a RMSE value lower of that of Cinematch.

3.3.3. User bias

The linear model constructed with the ratings' mean (μ) and the movie bias (b_i) can be improved upon by adding user bias (b_u), thus bringing every piece together.

$$\hat{y} = \mu + b_i + b_u + \epsilon_{u,i}$$

As can be appreciated, this formula finally resembles the one initially presented, being every element now in place. Note that, as was the case with the movie bias, two variables are summarized via the summarize() function: one takes μ and b_i into account whereas the other one does not (the first one being an adjusted user bias and the latter one being an isolated user bias).
The slice_head() function is used to reduce the output's length since it is only needed to showcase the bias' values.

```
# Bias per user
b_u <- train_set %>%
  left_join(b_i, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i),
            b_u_isolated = mean(rating))
b_u %>% slice_head(n = 10)
```

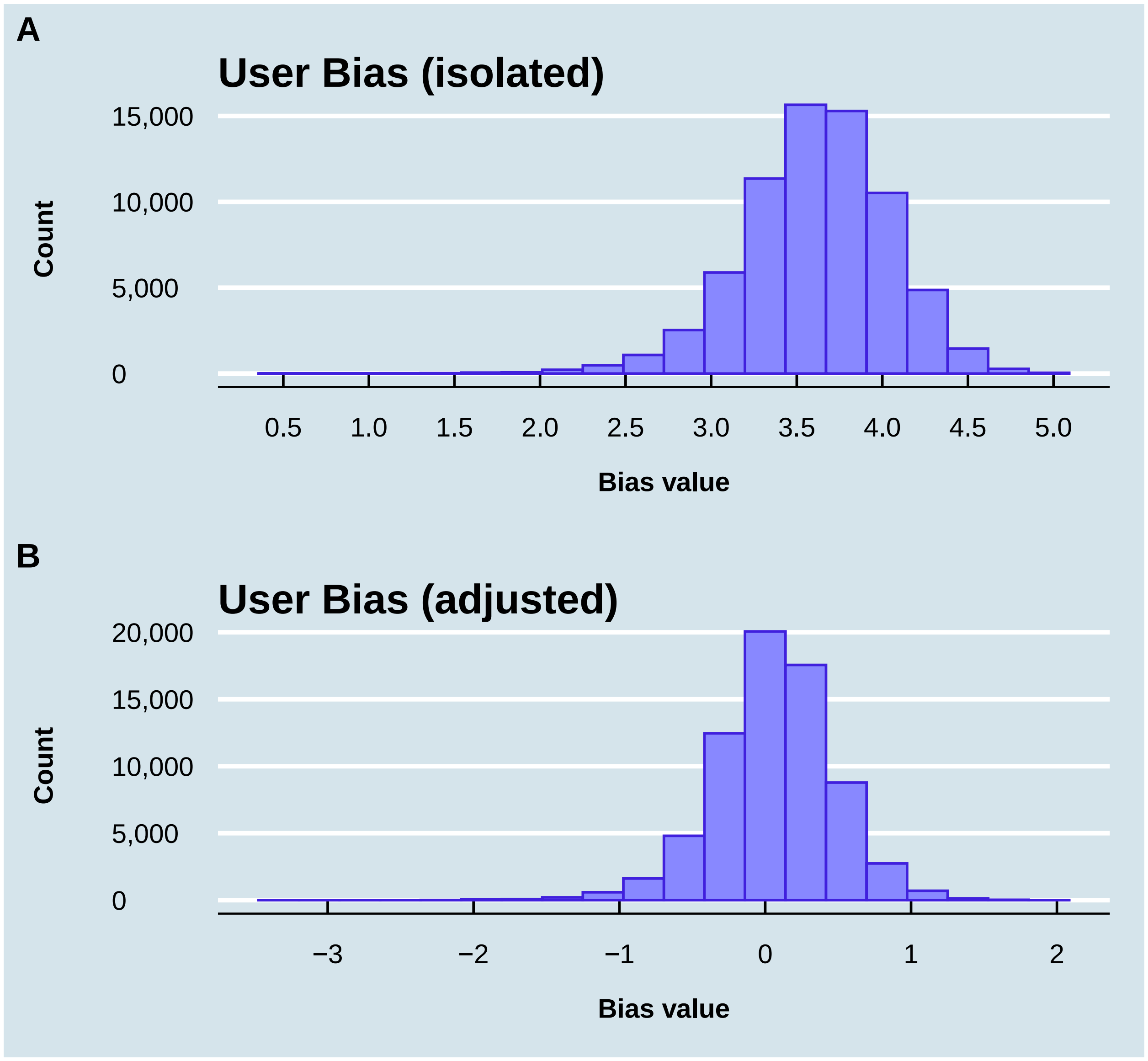
userId	b_u	b_u_isolated
1	1.6792347	5.000000
2	-0.2364086	3.294118
3	0.2643303	3.935484
4	0.6520781	4.057143
5	0.0852677	3.918919
6	0.3462454	3.948718
7	0.0238214	3.864583
8	0.2030957	3.386520
9	0.2320728	4.047619
10	0.0833311	3.830357

To properly evaluate the set's b_u values and their distribution, a plot built with ggplot() and its auxiliary functions is highly recommended. Note that there are two distinct plots: the first one corresponds to the isolated user bias effect whereas the following one takes into account the ratings' mean and the movie bias, both participant in the linear model being developed. While with the movie bias effect both of these plots were identical, that is the case no more as the adapted user bias takes the movie bias into account and, since that is no constant value, the whole graph is changed (it is not a simple horizontal displacement, which was the case with the movie bias' plots).

```r
# Isolated user bias plot
b_u_isolated_plot <- b_u %>%
  ggplot(aes(x = b_u_isolated)) +
  geom_histogram(bins = 20, fill = "#8888ff", color = "#4020dd") +
  ggtitle("User Bias (isolated)") +
  xlab("Bias value") +
  ylab("Count") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))

# Adjusted user bias plot
b_u_plot <- b_u %>%
  ggplot(aes(x = b_u)) +
  geom_histogram(bins = 20, fill = "#8888ff", color = "#4020dd") +
  ggtitle("User Bias (adjusted)") +
  xlab("Bias value") +
  ylab("Count") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))

# Both b_u plots are combined with plot_grid()
plot_grid(b_u_isolated_plot, b_u_plot, labels = "AUTO", nrow = 2)
```



The isolated bias plot showcases a normal distribution spiking at 3.6351351 and possesses a mean value of 3.6136016. On the other hand, the adjusted bias plot showcases a normal albeit slightly left skewed distribution which spikes at 0.0728787 and possesses a mean value of 0.061343. These two plots are similar in shape, although not quite as similar as the movie bias plots and definitely not identical (the reason being no other than the movie bias being taken into account, μ merely displaces the plot horizontally).

Using the user bias to build upon the previous linear model leads to the predictions and estimates showcased in the following code snippet:

```r
# Linear model construction (mean + movie bias + user bias)
y_hat_b_u <- validation %>%
  left_join(b_i, by='movieId') %>%
  left_join(b_u, by='userId') %>%
  mutate(y_hat = mu + b_i + b_u) %>%
  .$y_hat
```

```
evaluation <- bind_rows(evaluation,
                        tibble(Model = "Linear model (mean + movie and user bias)",
                               MAE = Metrics::mae(validation$rating, y_hat_b_u),
                               MSE = Metrics::mse(validation$rating, y_hat_b_u),
                               RMSE = Metrics::rmse(validation$rating, y_hat_b_u)))
print(evaluation)
## # A tibble: 6 x 4
##   Model                                       MAE    MSE  RMSE
##   <chr>                                     <dbl>  <dbl> <dbl>
## 1 Cinematch                                NA     NA     0.952
## 2 The Netflix Prize                        NA     NA     0.857
## 3 Random guessing                           1.17   2.25  1.50
## 4 Linear model (mean baseline)              0.856  1.13  1.06
## 5 Linear model (mean + movie bias)          0.738  0.891 0.944
## 6 Linear model (mean + movie and user bias)  0.669  0.749 0.865
```

The inclusion of both bias reduces all error metrics almost to The Netflix Prize threshold value. Is the model that good?

3.4. Movie recommendations

Having completed the linear model through the use of all its different elements (thus reducing the error metrics as much as feasible), it is time to observe its movie recommendation to properly evaluate its behavior.

To output the model's predicted best movies, they are to be sorted by $\hat{y}$ in descending order (since it weight the ratings mean and both bias). Note that most movies have multiple ratings and it is likely that the best (and worst) movies have been rated as such multiple times so, to avoid duplicated films in the prediction output, the function unique() is to be used since it returns a vector, dataframe or array without any duplicate elements or rows.
Let's limit the prediction to the top 10 movies through the use of the slice_head() function (with n = 10) as argument to match said criteria).

```
# Top 10 movie recommendation by the linear model
top10_prediction_linear <- test_set %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  mutate(y_hat = mu + b_i + b_u) %>%
  arrange(desc(y_hat)) %>%
  select(title) %>%
  unique() %>%
  slice_head(n = 10)
top10_prediction_linear_df <- data.frame(Title = top10_prediction_linear,
                                         Rating = rep(NA, 10),
                                         Count = rep(NA, 10))

for (i in 1:10) {
  indexes <- which(test_set$title == as.character(top10_prediction_linear[i]))
  top10_prediction_linear_df$Rating[i] <- mean(test_set$rating[indexes])
  top10_prediction_linear_df$Count[i] <- sum(
    test_set$title == as.character(top10_prediction_linear[i])
  )
}
```

```
print(top10_prediction_linear_df)
##                                                                         title
## 1                                                      Usual Suspects, The (1995)
## 2                                                Shawshank Redemption, The (1994)
## 3                                                         Schindler's List (1993)
## 4                                     Eternal Sunshine of the Spotless Mind (2004)
## 5                                          Wallace & Gromit: A Close Shave (1995)
## 6                    Star Wars: Episode IV - A New Hope (a.k.a. Star Wars) (1977)
## 7   Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)
## 8                                   Star Wars: Episode VI - Return of the Jedi (1983)
## 9                                                             Pulp Fiction (1994)
## 10      Good, the Bad and the Ugly, The (Buono, il brutto, il cattivo, Il) (1966)
##      Rating Count
## 1  4.378820  2389
## 2  4.476213  3111
## 3  4.363390  2584
## 4  4.182771   859
## 5  4.271807   642
## 6  4.210435  2894
## 7  4.279059  2125
## 8  3.995625  2514
## 9  4.181039  3502
## 10 4.108665   704
```

Obtaining the worst 10 predicted results follows a similar procedure, although $\hat{y}$ is to be sorted in ascending order.

```
# Worst 10 movie recommendation by the linear model
worst10_prediction_linear <- test_set %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  mutate(y_hat = mu + b_i + b_u) %>%
  arrange(b_i) %>%
  select(title) %>%
  unique() %>%
  slice_head(n = 10)
worst10_prediction_linear_df <- data.frame(Title = worst10_prediction_linear,
                                           Rating = rep(NA, 10),
                                           Count = rep(NA, 10))

for (i in 1:10) {
  indexes <- which(test_set$title == as.character(worst10_prediction_linear[i]))
  worst10_prediction_linear_df$Rating[i] <- mean(test_set$rating[indexes])
  worst10_prediction_linear_df$Count[i] <- sum(
    test_set$title == as.character(worst10_prediction_linear[i])
  )
}
```

```
print(worst10_prediction_linear_df)
##                                                        title    Rating Count
## 1                            Confessions of a Superhero (2007) 3.0000000     1
## 2                    War of the Worlds 2: The Next Wave (2008) 1.0000000     1
## 3                          SuperBabies: Baby Geniuses 2 (2004) 0.9000000     5
## 4                                        Disaster Movie (2008) 1.1250000     8
## 5                                   From Justin to Kelly (2003) 1.1764706    17
## 6  When Time Ran Out... (a.k.a. The Day the World Ended) (1980) 2.5000000     2
## 7                                             Criminals (1996) 2.5000000     2
## 8                                   Mountain Eagle, The (1926) 2.0000000     2
## 9                                        PokÃ©mon Heroes (2003) 0.8684211    19
## 10                                        Roller Boogie (1979) 2.5000000     2
```

There is a clear issue with the proposed films: although the top 10 movie recommendations have an overall higher rating than the worst 10 ones, there are cases where a given movie has the same average rating and overall count in both sets. What's more, most of the suggested movies have not enough ratings to properly assess them as top (or worst) movie recommendation (you can't recommend a film to a population based on a single rating by a single user). To correct such problematic, the model needs to be regularized.

3.4. Regularization

Regularization consists in penalizing values that differ from the actual observed result, applying a stronger penalty the further they fall from the desired value. Through this mechanism, regularization fixes fitting issues such as:

- Overfitting: overfitted models work great for the training data since it is overly adapted to it, but does not generalize well to real world scenarios. Overfitted lines are often characterized by an extremely complex shape which follows most of the training data points, and regularization can help simplifying its shape so that it better fits the actual data behavior/pattern.
- Underfitting: underfitted models are too simple, generalizable but not meaningful (less variance but more bias towards the wrong answers, meaning that they hardly represent the data).

There are many regularization approaches and many great examples in literature that explain them and their intricacies in extensive detail. However, that goes beyond the scope of this document.
The most important aspect is that isolating the bias effect from the regularization mathematical expression leads to the following equalities:

$$\hat{b}_i = \frac{1}{n_i + \lambda} \sum_{u=1}^{n_i} (y_{u,i} - \hat{\mu})$$

$$\hat{b}_u = \frac{1}{n_u + \lambda} \sum_{i=1}^{n_u} (y_{u,i} - \hat{b}_i - \hat{\mu})$$

With the last two formulas in mind, a regularization function can be constructed: the mean is obtained as usual, the bias effect follows said formulas and the predictions are constructed just like they were in the latest linear model (which contemplated both movie and user bias). The function should return the RMSE value since the goal is to iterate through penalty values (λ) to minimize the RMSE.

```
# Regularization function
regularization <- function(lambda, train_set, test_set){
  mu <- mean(train_set$rating)

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

  b_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    filter(!is.na(b_i)) %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - mu - b_i) / (n() + lambda))

  predicted_ratings <- test_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    filter(!is.na(b_i), !is.na(b_u)) %>%
    mutate(pred = mu + b_i + b_u) %>%
    pull(pred)

  return(Metrics::rmse(predicted_ratings, test_set$rating))
}
```

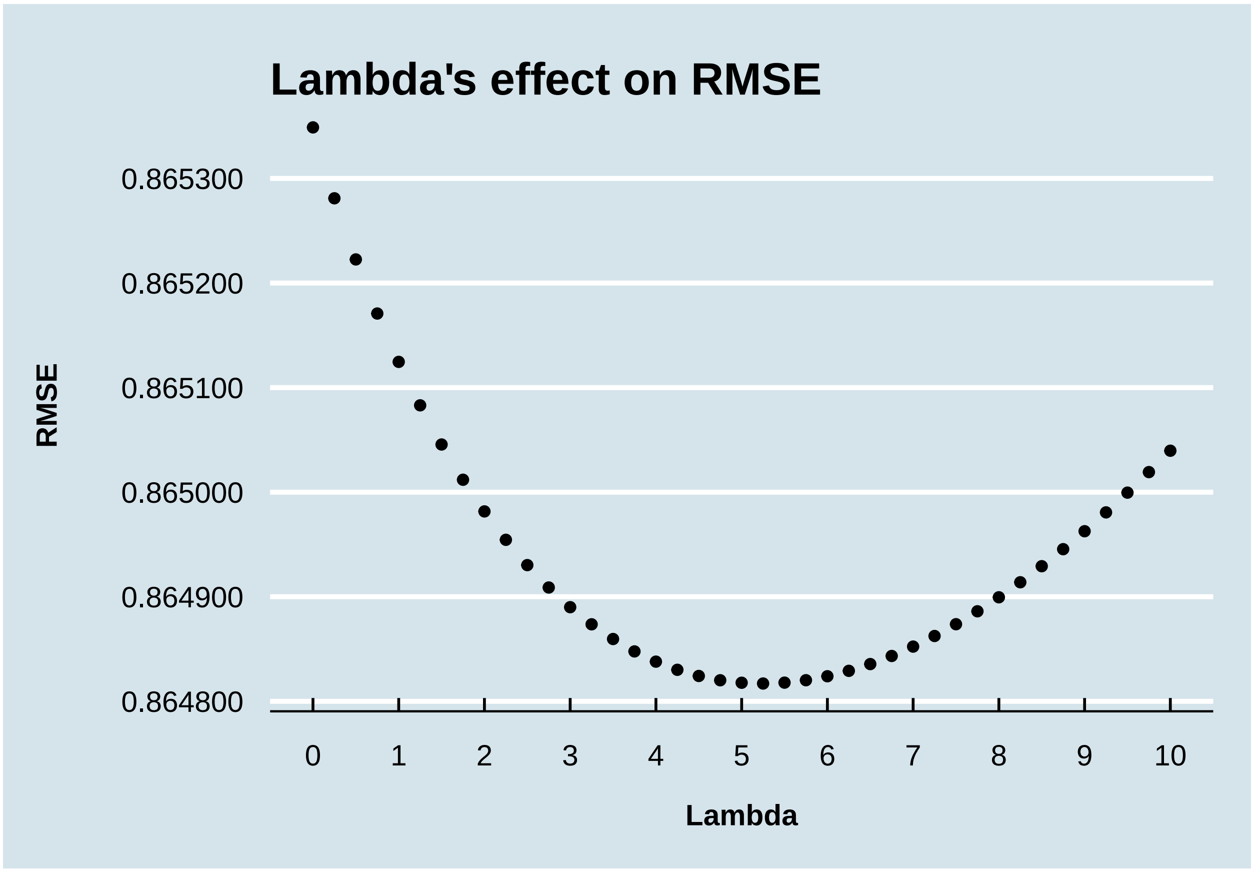
The next step would be to make use of the recently created function to iterate through the penalty values. To do so, sapply() iterates through the elements of an array applying a supplied function to each (beware the computing time). Holding each returned value in a vector is recommended to evaluate the results and plot them if needed.

```
# The regularization function at play
lambdas <- seq(0, 10, 0.25)
lambdas_rmse <- sapply(lambdas,
                       regularization,
                       train_set = train_set,
                       test_set = test_set)
lambdas_tibble <- tibble(Lambda = lambdas, RMSE = lambdas_rmse)
print(lambdas_tibble)
## # A tibble: 41 x 2
##    Lambda  RMSE
##     <dbl> <dbl>
##  1   0     0.865
##  2   0.25 0.865
##  3   0.5   0.865
##  4   0.75 0.865
##  5   1     0.865
##  6   1.25 0.865
##  7   1.5   0.865
##  8   1.75 0.865
##  9   2     0.865
## 10   2.25 0.865
## # ... with 31 more rows
```

From these results, it can be seen that the best RMSE result is achieved with 5.25. Note that the improvement is somewhat subtle.

Once again, plotting the results is highly recommended to visualize how different lambda values affect the RMSE. ggplot() and its auxiliary functions are the ideal tools to do so.

```r
# Lambda's effect on RMSE plot
lambdas_tibble %>%
  ggplot(aes(x = Lambda, y = RMSE)) +
  geom_point() +
  ggtitle("Lambda's effect on RMSE") +
  xlab("Lambda") +
  ylab("RMSE") +
  scale_y_continuous(n.breaks = 6, labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



The plot clearly showcases that the lambda that returns the lowest RMSE is 5.25 (the closest point to 5 from the right). Having obtained this value, the regularized linear model can be constructed; the following code snippet showcases the process.

```r
# Regularized linear model construction
lambda <- lambdas[which.min(lambdas_rmse)]

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

```r
b_i_regularized <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n()+lambda))

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

y_hat_regularized <- validation %>%
  left_join(b_i_regularized, by = "movieId") %>%
  left_join(b_u_regularized, by = "userId") %>%
  mutate(prediction = mu + b_i + b_u) %>%
  pull(prediction)

evaluation <- bind_rows(evaluation,
                        tibble(Model = "Linear model with regularized bias",
                               MAE  = Metrics::mae(validation$rating, y_hat_regularized),
                               MSE  = Metrics::mse(validation$rating, y_hat_regularized),
                               RMSE = Metrics::rmse(validation$rating, y_hat_regularized)))
print(evaluation)
## # A tibble: 7 x 4
##   Model                                       MAE    MSE  RMSE
##   <chr>                                     <dbl>  <dbl> <dbl>
## 1 Cinematch                                NA     NA     0.952
## 2 The Netflix Prize                        NA     NA     0.857
## 3 Random guessing                           1.17   2.25  1.50
## 4 Linear model (mean baseline)              0.856  1.13  1.06
## 5 Linear model (mean + movie bias)          0.738  0.891 0.944
## 6 Linear model (mean + movie and user bias)  0.669  0.749 0.865
## 7 Linear model with regularized bias         0.669  0.748 0.865
```

As previously stated, the RMSE improvement is somewhat subtle.

3.4.1. Movie recommendations

The process to obtain the top 10 movie recommendations is replicated (now using regularized data).

```r
# Top 10 movie recommendation by the regularized linear model
top10_prediction_regularized <- test_set %>%
  left_join(b_i_regularized, by = "movieId") %>%
  left_join(b_u_regularized, by = "userId") %>%
  mutate(y_hat = mu + b_i + b_u) %>%
  arrange(desc(y_hat)) %>%
  select(title) %>%
  unique() %>%
  slice_head(n = 10)
top10_prediction_regularized_df <- data.frame(Title = top10_prediction_regularized,
                                              Rating = rep(NA, 10),
                                              Count = rep(NA, 10))
```

```
for (i in 1:10) {
  indexes <- which(test_set$title == as.character(top10_prediction_regularized[i]))
  top10_prediction_regularized_df$Rating[i] <- mean(test_set$rating[indexes])
  top10_prediction_regularized_df$Count[i] <- sum(
    test_set$title == as.character(top10_prediction_regularized[i])
  )
}
print(top10_prediction_regularized_df)
##                                                                              title
## 1                                                      Usual Suspects, The (1995)
## 2                                                Shawshank Redemption, The (1994)
## 3                                    Eternal Sunshine of the Spotless Mind (2004)
## 4                    Star Wars: Episode IV - A New Hope (a.k.a. Star Wars) (1977)
## 5                                                             Donnie Darko (2001)
## 6                                                         Schindler's List (1993)
## 7                               Star Wars: Episode VI - Return of the Jedi (1983)
## 8  Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)
## 9                                                             Shining, The (1980)
## 10      Good, the Bad and the Ugly, The (Buono, il brutto, il cattivo, Il) (1966)
##      Rating Count
## 1  4.378820  2389
## 2  4.476213  3111
## 3  4.182771   859
## 4  4.210435  2894
## 5  4.085655   718
## 6  4.363390  2584
## 7  3.995625  2514
## 8  4.279059  2125
## 9  4.002934  1193
## 10 4.108665   704
```

Obtaining the worst 10 predicted results follows a similar procedure, although $\hat{y}$ is to be sorted in ascending order.

```
# Worst 10 movie recommendation by the regularized linear model
worst10_prediction_regularized <- test_set %>%
  left_join(b_i_regularized, by = "movieId") %>%
  left_join(b_u_regularized, by = "userId") %>%
  mutate(y_hat = mu + b_i + b_u) %>%
  arrange(y_hat) %>%
  select(title) %>%
  unique() %>%
  slice_head(n = 10)
worst10_prediction_regularized_df <- data.frame(Title = worst10_prediction_regularized,
                                                Rating = rep(NA, 10),
                                                Count = rep(NA, 10))
```

```
for (i in 1:10) {
  indexes <- which(test_set$title == as.character(worst10_prediction_regularized[i]))
  worst10_prediction_regularized_df$Rating[i] <- mean(test_set$rating[indexes])
  worst10_prediction_regularized_df$Count[i] <- sum(
    test_set$title == as.character(worst10_prediction_regularized[i])
  )
}
print(worst10_prediction_regularized_df)
##                                           title    Rating Count
## 1                       Battlefield Earth (2000) 1.6650246   203
## 2  Police Academy 4: Citizens on Patrol (1987) 2.0572519   131
## 3               Karate Kid Part III, The (1989) 2.0639810   211
## 4                        PokÃ©mon Heroes (2003) 0.8684211    19
## 5           Turbo: A Power Rangers Movie (1997) 1.3695652    46
## 6                                 Kazaam (1996) 1.8378378   111
## 7                       Shanghai Surprise (1986) 1.1500000    10
## 8               Free Willy 3: The Rescue (1997) 1.8617021    47
## 9                                  Steel (1997) 1.4333333    30
## 10                          Iron Eagle IV (1995) 1.6323529    34
```

3.5. Matrix factorization

The end of the line. This exercise's goal was to reach this final step, since it was the approach that surpassed The Netflix Prize challenge and has been proven to be superior to classic nearest-neighbor techniques for product recommendations. This highly recommended video by Dr. David Eisenbud (director of the Mathematical Sciences Research Institute) brilliantly explains the core concept, its usage and its limitations. Even Wikipedia's article on the matter is highly explanatory but, all in all, this mathematical model helps the system split a set into multiple smaller objects through an ordered rectangular array of numbers or functions in order to discover the features or information underlying the interactions between users and items.

Despite the complexities of the matter, the recosystem package bundles a collection of functions to simplify the process at hand, which goes as follows:

- First, the training and testing set need to be converted into the library's own input format.
- Secondly, an empty system/model object is to be created. The function Reco() is the one responsible of such task.
- Thirdly, the model object is to be tuned. Doing so requires the use of the tune() function, which has a great amount of customizability. Detailing every intricacy goes beyond the scope of this document, but the hyperlinked RDocumentation page is highly documented. Beware that this tuning process requires a fair amount of computing power and time (perhaps the most costly process of the entire document).
- The forth step trains the model (stored within the previously created object) so that it fits the supplied data. The function train() is the one responsible of such task.
- Lastly, the predict() function yields the $\hat{y}$ prediction with which is possible to compute the model's RMSE and thus evaluate its performance

The following code snippet goes over these 5 steps (the commented lines highlight the one at hand). As was stated during their detailing, some of them are bound to a considerable amount of computing power and time (so beware of that).

```
# 1. The training and testing sets need to be converted into recosystem input format
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
train_recosystem <- with(train_set, data_memory(user_index = userId,
                                                item_index = movieId,
                                                rating     = rating))
test_recosystem <- with(test_set, data_memory(user_index = userId,
                                              item_index = movieId,
                                              rating     = rating))

# 2. The model object is created
recommendation_system <- Reco()

# 3. The model is tuned
tuning <- recommendation_system$tune(train_recosystem, opts = list(dim = c(10, 20, 30),
                                                                   lrate = c(0.1, 0.2),
                                                                   nthread  = 4,
                                                                   niter = 10))

# 4. The model is trained
recommendation_system$train(train_recosystem, opts = c(tuning$min,
                                                       nthread = 4,
                                                       niter = 20))
## iter      tr_rmse          obj
##    0       0.9722   1.1997e+07
##    1       0.8719   9.8831e+06
##    2       0.8384   9.1609e+06
##    3       0.8159   8.7400e+06
##    4       0.8006   8.4596e+06
##    5       0.7892   8.2674e+06
##    6       0.7797   8.1173e+06
##    7       0.7717   8.0035e+06
##    8       0.7648   7.9044e+06
##    9       0.7590   7.8246e+06
##   10       0.7538   7.7609e+06
##   11       0.7491   7.7002e+06
##   12       0.7449   7.6481e+06
##   13       0.7412   7.6077e+06
##   14       0.7377   7.5675e+06
##   15       0.7346   7.5331e+06
##   16       0.7317   7.5006e+06
##   17       0.7290   7.4737e+06
##   18       0.7266   7.4491e+06
##   19       0.7243   7.4265e+06

# 5. A prediction is made
y_hat_MF <-  recommendation_system$predict(test_recosystem, out_memory())
```

With $\hat{y}$ obtained, the model's RMSE can be computed and added to the evaluation table.

```
# The model's RMSE is computed and added to the evaluation table
evaluation <- bind_rows(evaluation,
                        tibble(Model = "Matrix factorization",
                               MAE  = Metrics::mae(validation$rating, y_hat_MF),
                               MSE  = Metrics::mse(validation$rating, y_hat_MF),
                               RMSE = Metrics::rmse(validation$rating, y_hat_MF)))
print(evaluation)
## # A tibble: 8 x 4
##   Model                                        MAE    MSE  RMSE
##   <chr>                                      <dbl>  <dbl> <dbl>
## 1 Cinematch                                 NA     NA     0.952
## 2 The Netflix Prize                         NA     NA     0.857
## 3 Random guessing                            1.17   2.25  1.50
## 4 Linear model (mean baseline)               0.856  1.13  1.06
## 5 Linear model (mean + movie bias)           0.738  0.891 0.944
## 6 Linear model (mean + movie and user bias)  0.669  0.749 0.865
## 7 Linear model with regularized bias         0.669  0.748 0.865
## 8 Matrix factorization                       0.603  0.613 0.783
```

3.5.1. Movie recommendations

The process to obtain the top 10 movie recommendations has changed ever-so-slightly, now requiring the use of a newly create tibble which results from joining together the testing data, which was used to construct the y_hat_MF prediction array, and that very same array. After that, procedure's the same one as in previous movie recommendation code snippets.

```
# Top 10 movie recommendation by the matrix factorization model
top10_prediction_MF <- tibble(title = test_set$title, y_hat = y_hat_MF) %>%
  arrange(desc(y_hat)) %>%
  select(title) %>%
  unique() %>%
  slice_head(n = 10)
top10_prediction_MF_df <- data.frame(Title = top10_prediction_MF,
                                     Rating = rep(NA, 10),
                                     Count = rep(NA, 10))

for (i in 1:10) {
  indexes <- which(test_set$title == as.character(top10_prediction_MF[i,]))
  top10_prediction_MF_df$Rating[i] <- mean(test_set$rating[indexes])
  top10_prediction_MF_df$Count[i] <- sum(
    test_set$title == as.character(top10_prediction_MF[i,])
  )
}
print(top10_prediction_MF_df)
```

Obtaining the worst 10 predicted results follows a similar procedure, although $\hat{y}$ is to be sorted in ascending order.

```r
# Worst 10 movie recommendation by the matrix factorization model
worst10_prediction_MF <- tibble(title = test_set$title, y_hat = y_hat_MF) %>%
  arrange(y_hat) %>%
  select(title) %>%
  unique() %>%
  slice_head(n = 10)
worst10_prediction_MF_df <- data.frame(Title = worst10_prediction_MF,
                                       Rating = rep(NA, 10),
                                       Count = rep(NA, 10))

for (i in 1:10) {
  indexes <- which(test_set$title == as.character(worst10_prediction_MF[i,]))
  worst10_prediction_MF_df$Rating[i] <- mean(test_set$rating[indexes])
  worst10_prediction_MF_df$Count[i] <- sum(
    test_set$title == as.character(worst10_prediction_MF[i,])
  )
}
print(worst10_prediction_MF_df)
```

4. Conclusion

This document details to considerable length the process of building a recommender system. The MovieLens dataset has been loaded and manipulated to fit the workspace, and its content has been throughfully explored with many of its most relevant aspects being plotted for its proper visual examination. A baseline model was built through random guessing, after which both a linear model and a matrix factorization based model were developed. Both of these models represent different approaches in the construction of a recommender system: the first one being the one upon which the Cinematch algorithm was built whereas the latter one was the approach to dethrone Cinematch in The Netflix Prize challenge.

All in all, both models provide reliable approaches as shown in the evaluation table/tibble and through the recommended movies. However, having used both, it is undeniable that matrix factorization yields better results (as was to be expected since the linear nature of linear models usually lead to underfitting issues in non-linear data). Increasing the model's dimensionality allow the model to better represent more complex data distributions at the risk of being overfit (as is, overly adapted to the supplied data and not generalizable to new data), so a validation test was an utmost necessity.

4.1. Limitations

The most notorious limitation is that only two of the dataset's predictors have been used: users and movies. Genres could play a significant role and lead to a relevant decrease in the RMSE. Besides that, it is worth noting that some models (particularly the matrix factorization one) could have benefited from a finer tuning process (although the potential improvements are unclear, I actually doubt that it would have improved that much).
Time has also been a limitation for me. I'm sure it has been noticed that the first chapters have been written with more affection than the last ones, so I hope that has not been much of an issue and that the points and processes I have been trying to showcase have been understood (I will most likely rewrite whichever parts I deem necesary in the future).

4.2. Future work

I would love to come back to this project to expand upon it with new and different approaches, perhaps even documenting a few intricacies that have been left aside to keep this document' scope manageable. Truth is that there exist an overwhelming amount of machine learning approaches that would be interesting to test and build a recommender system with. Also, as I have already commented, the latter parts of the document would benefit from a polish layer, so I will most likely rewrite whichever parts I deem necesary as soon as I find the time.

5. Bibliography

Work In Progress (WIP)