HarvardX - Data Science - PH125.9x - MovieLens project

Project part of the Capstone course

Ioannis Barbanas - https://github.com/barbanas/movielens_project

2024-05-01

Contents

Chapter 1

Introduction

As part of the training we receive from Professor Rafael A. Irizarry while taking the Data Science course (PH125.9x) of HarvardX, we commissioned to create a Machine Learning project based on the MovieLens data and write a report with our results. With this challenge, we remembered all the course teachings and utilized our newly acquired knowledge in R and Data Science.

The goal of this project is to create a recommendation algorithm for movies. This specific recommendation algorithm is going to predict and suggest movies that users might be interested in watching. The (training and test) data set was already prepared for us and it is a subset of the larger MovieLens data set[1].

The baseline for this project is to minimize the **Root Mean Squared Error (or RMSE)** of predicted ratings for pairs of user/movie, below the threshold of 0.8649.

```
minimum_RMSE <- 0.8649 # as set by the project's guidelines
```

Recommendation algorithms is not something new and it is already used for many different fields like search engines (the magical thing that fills in your sentence while you type what you are searching for), shopping suggestions in eShops (e.g. Amazon.com, eBay.com, etc), movie suggestions (e.g. Netflix, Disney+, etc).

To learn more about recommender systems you can read the following article in Wikepedia[2] or in NVIDIA's web site[3].

1.1 How this report is organized

This report is organised as follows.

In Chapter 2, we describe the data set and prepare it for our analysis. We also add some helper functions that we will use later.

Chapter 3 is showing some visualizations of our data that will help us understand what the features and characteristics of our data are.

In Chapter 4 we are setting the baseline that the success of our models will be measured against.

Chapter 5 proposes, trains and predicts results with the help of three Machine Learning models (from the caret package). All these models were selected with our goal in mind and all of them are (also) used for regression (some of them are also used for classification tasks).

Chapter 6 attacks the problem from a different angle by utilizing a specialized Matrix Factorization model (from the recommender package). This model is designed especially for recommendation problems like this.

[1] https://grouplens.org/datasets/movielens/10m/
[2] https://en.wikipedia.org/wiki/Recommender_system
[3] https://www.nvidia.com/en-us/glossary/recommendation-system/

In Chapter 7 we describe our findings and propose possible ways that may improve the RMSE even more.

Lastly, in Chapter 8 we mention the technical characteristics of the underlying hardware and software that was used for carrying out our analysis.

All the R code has been extensively commented (see `# ...`) in order for the reader to be able to understand what we are doing on each point and why we are doing it.

In case you have any suggestions or recommendations on how we can improve this report, please leave us a comment on the github.

Chapter 2

Preparation of data sets (pre processing) & helper functions

2.1 The MovieLens data set

The data provided by MovieLens is a very large data collection of user ratings made for movies.

The entire training data set (which is called "`edx`") is a table of 9000055 observations (rows) and 6 variables (columns). The variables that consist the `edx` are the following:

```
##      Variable        Data_type
## 1    ‘userId‘        Numerical
## 2   ‘movieId‘        Numerical
## 3    ‘rating‘        Numerical
## 4 ‘timestamp‘        Numerical
## 5     ‘title‘ Character string
## 6    ‘genres‘ Character string
```

The table below shows what each variable (column) stands for.

Variable	Description
‘userId‘	Each user has a unique ID (e.g. 1) that distinguises him/her from all the other users
‘movieId‘	Each movie has a unique ID (e.g. 122) that distinguises it from all the other movies
‘rating‘	Ratings on this data set are ranging from 0, 0.5, 1, 2, 3, 4, 4.5 and 5.0
‘timestamp‘	Date and time stamp as (Unix epoch) when the rating was submitted (e.g. 838985046)
‘title‘	String for Movie title and, in parenthesis (), the year the movie first released
‘genres‘	String of characters that contains the genre(s) to which the movie belongs to (e.g. Drama, et

Each row of data is a single rating (see column `rating`) made by a specific user (`userId`) for a specific movie (`movieId`) on a specific point in time (`timestamp`). Let's see some examples:

```r
head(edx)
```

```
##   userId movieId rating timestamp                        title
## 1      1     122      5 838985046             Boomerang (1992)
## 2      1     185      5 838983525              Net, The (1995)
## 4      1     292      5 838983421              Outbreak (1995)
## 5      1     316      5 838983392              Stargate (1994)
## 6      1     329      5 838983392 Star Trek: Generations (1994)
## 7      1     355      5 838984474      Flintstones, The (1994)
##                          genres
## 1                Comedy|Romance
## 2         Action|Crime|Thriller
## 4  Action|Drama|Sci-Fi|Thriller
## 5       Action|Adventure|Sci-Fi
## 6 Action|Adventure|Drama|Sci-Fi
## 7       Children|Comedy|Fantasy
```

[1]

2.2 Pre processing of the data

Any data wrangling should be done during the pre processing phase, before analysis takes place.

We did not want to change the existing variables or add new variables to the data set (with the help of `mutate()`), although this is possible and it may have been useful in increasing the predicting accuracy of the algos.

For example, seeing how ratings are changing (or not) by the pass of time (from the first time the movie released) would be an interesting experiment.

Another idea would be to examine if the popularity effect (some movies are more popular than others, therefore are watched by far more people and the more people that watch a movie there is an increased probability that movie to get rated).

However, adding more predictions (variables) not only increases the complexity of the calculations involved, but also the time needed to train the algos. Additionally, which and how many predictors to add, how you may combine them together or separate them (to explore the presence or lack of certain effects), is almost, a never ending story.

So, we decided to keep things simple at first and see if the variables in the original data set will be sufficient to predict with enough accuracy, before considering the possibility to add any new predictors.

2.3 Helper functions

The Root Mean Square Error (or RMSE) will be used for evaluating how close our predictions are to the true values in the `final_holdout_test set`.

We were already given the formula that calculates the RMSE. Herein below we create a function that will use this formula and calculate the RSME for pairs of the true_ratings vs predicted_ratings.

```
print(RMSE)
```

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

We also created a function that will print the table that will contain the results of our analysis as we go on.

```
print(show_kable_table)
```

```
## function (table_to_print)
## {
##     kable_styling(knitr::kable(table_to_print, "latex", booktabs = T),
##         full_width = TRUE)
## }
```

[1] For more details about the data set you can read the `README.html` file included in the zip file of the MovieLens data.

Chapter 3

Data exploration and visualisation

In this chapter we will check some facts about the `edx` data set which is the training data set of this project.

We are not going to do the same for the `final_holdout_test` data set that will be used for testing, because our observations and models must be based on the test set alone.

Our aim is to get a bird's eye view of what our data are and what are their characteristics.

3.1 Checking the data contained in the variables

3.1.1 Users

In the `edx` data set there are 69878 unique users and 10677 unique movies.

3.1.1.1 Not all users are equal

In the data set we see that not all users give a lot of ratings. Some of them are more active (give more ratings) than others.

And, as a matter of fact, even the most active Users did not rate the most rated movies.

To illustrate this, let's plot a matrix that shows the ratings given to most rated movies by the most active users:

```r
suppressPackageStartupMessages(library(dplyr)) # data manipulation functions
suppressPackageStartupMessages(library(ggplot2)) # used for creating plots and charts
suppressPackageStartupMessages(library(tidyr)) # data manipulation (reshaping and
                                                # transforming data) to tidy format
library(RColorBrewer) # color palette for charts

top_number <- 100 # how many users and movies to take into account
# Calculate the total number of ratings given by each user
user_ratings_count <- edx |>
  group_by(userId) |>
  summarise(total_ratings = n()) |>
  arrange(desc(total_ratings))

# Calculate the total number of ratings given to each movie
movie_ratings_count <- edx |>
  group_by(movieId) |>
  summarise(total_ratings = n()) |>
  arrange(desc(total_ratings))
```

```r
# Select the first "top_number" users with the most ratings
top_users <- user_ratings_count |>
  slice(1:top_number) |>
  pull(userId)

# get the movieIds of the first "top_number" movies
# (movies that were rated the most)
most_rated_movies <-  movie_ratings_count |>
  slice(1:top_number) |>
  pull(movieId)

# Filter the ratings data set to include only ratings from the selected users
ratings_subset <- edx |>
  filter(userId %in% top_users & movieId %in% most_rated_movies)

# Pivot the data to wide format
ratings_matrix <- ratings_subset |>
  select(userId, title, rating) |>
  pivot_wider(names_from = title, values_from = rating, values_fill = NA)

# Convert data to long format
long_data <- pivot_longer(ratings_matrix,
                          cols = -userId,
                          names_to = "Item",
                          values_to = "Rating")

# Plotting
ggplot(long_data, aes(x = Item, y = factor(userId), fill = Rating)) +
  geom_tile() +
  scale_fill_gradient(low = "white", high = RColorBrewer::brewer.pal(3, "Blues")) +
  labs(
    title = "User Ratings for most rated Movies by most active Users",
    x = paste0("Users (",top_number," most active users)"),
    y = paste0("Movie (",top_number," most rated movies)"),
    fill = "Rating"
  ) +
  theme_classic() +
  theme(axis.text.y = element_text(size = 5)) + # make the font of the y-axis
                                                # values smaller
  #theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1)) +
  theme(axis.text.x = element_blank()) + # Hide x-axis labels
  coord_fixed()
```

The grey areas represent ratings that were not given for that particular movie from that particular user. These are the ratings we will try to predict.

We notice that even the 100 most rated movies of all times, were not rated by the 100 most active users. This means that we have a lot of missing (NA) ratings for the pair user/movie.

3.1.1.2 Ratings given by least active Users

This becomes even more evident when we see that the least active users did rate even less times the most rated movies.

```r
#Count the total number of rows in the tibble
total_rows <- nrow(user_ratings_count)
```

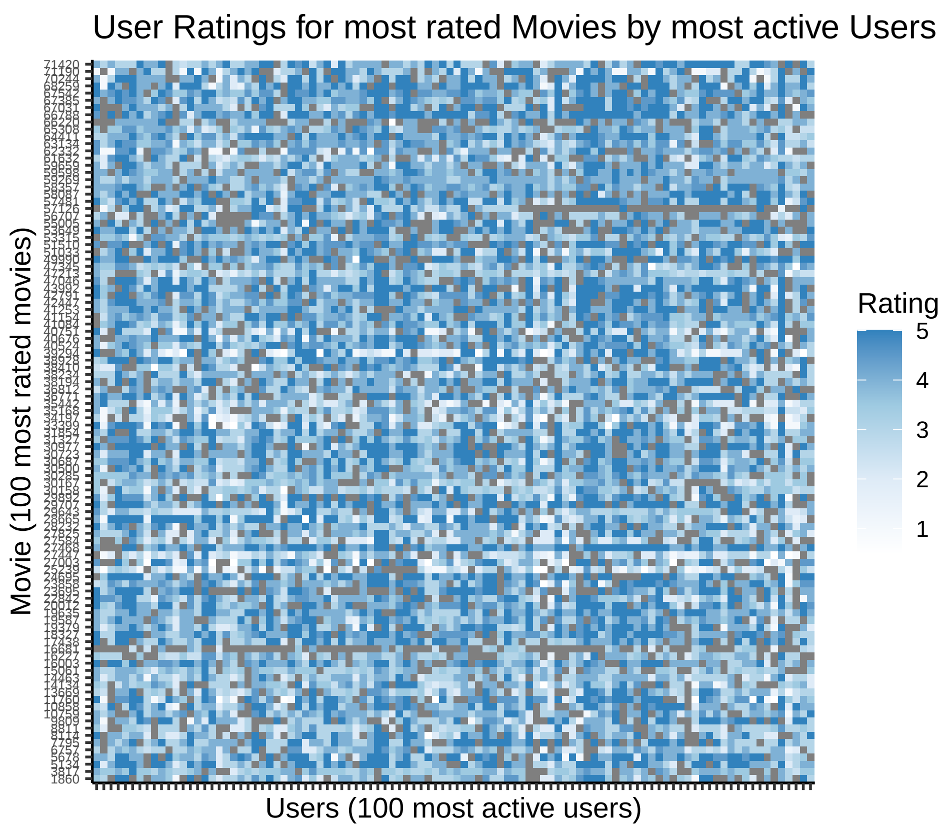


Figure 3.1: Number of ratings given to most rated Movies by most active users

```r
# Select the X users with the least ratings
least_active_users <- user_ratings_count |>
  slice((total_rows - top_number):total_rows) |>
  pull(userId)

# Filter the ratings dataset to include only ratings from the selected users
ratings_subset <- edx |>
  filter(userId %in% least_active_users & movieId %in% most_rated_movies)

# Pivot the data to wide format
ratings_matrix <- ratings_subset |>
  select(userId, title, rating) |>
  pivot_wider(names_from = title, values_from = rating, values_fill = NA)

# Convert data to long format
long_data <- pivot_longer(ratings_matrix,
                          cols = -userId,
                          names_to = "Item",
                          values_to = "Rating")
```

```r
# Plotting
ggplot(long_data, aes(x = Item, y = factor(userId), fill = Rating)) +
  geom_tile() +
  scale_fill_gradient(low = "white", high = RColorBrewer::brewer.pal(3, "Blues")) +
  labs(
    title = "Ratings given to most rated Movies by least active Users",
    x = paste0("Users (",top_number," least active users)"),
    y = paste0("Movie (",top_number," most rated movies)"),
    fill = "Rating"
  ) +
  theme_classic() +
  #theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1)) +
  theme(axis.text.y = element_text(size = 5)) +
  theme(axis.text.x = element_blank()) + # Hide x-axis labels
  coord_fixed()
```

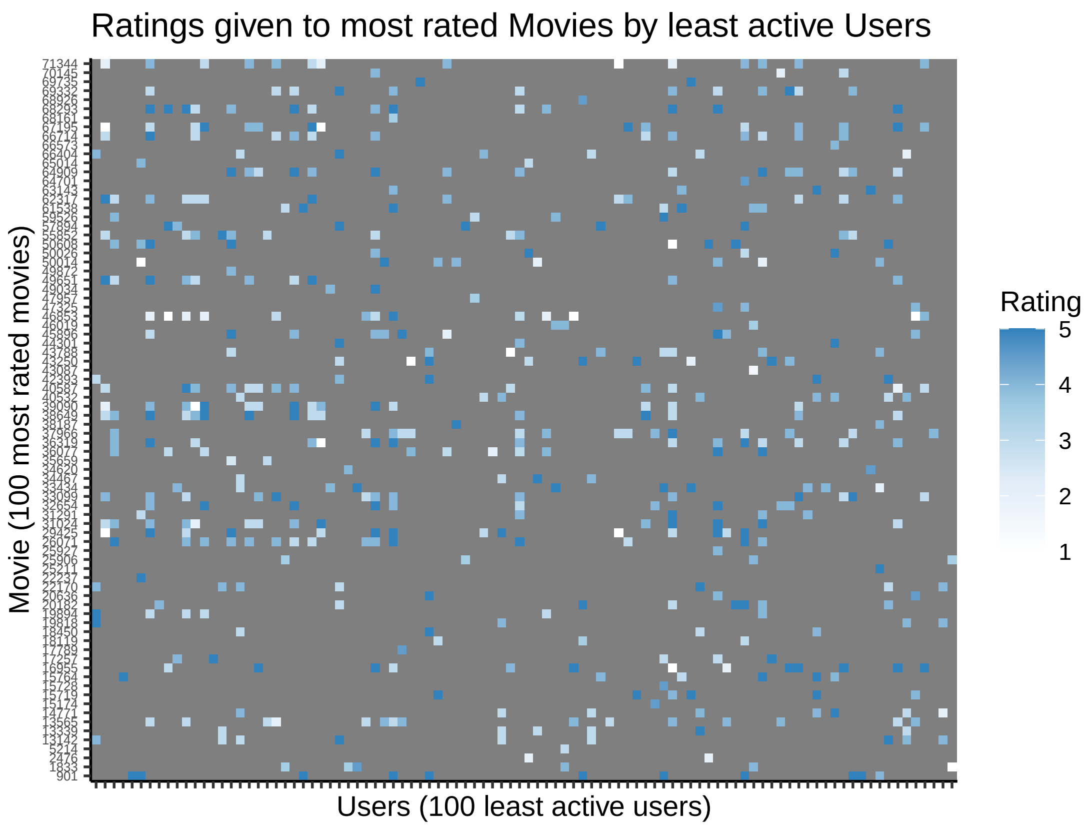


Figure 3.2: User Ratings given to the most rated Movies by least active Users

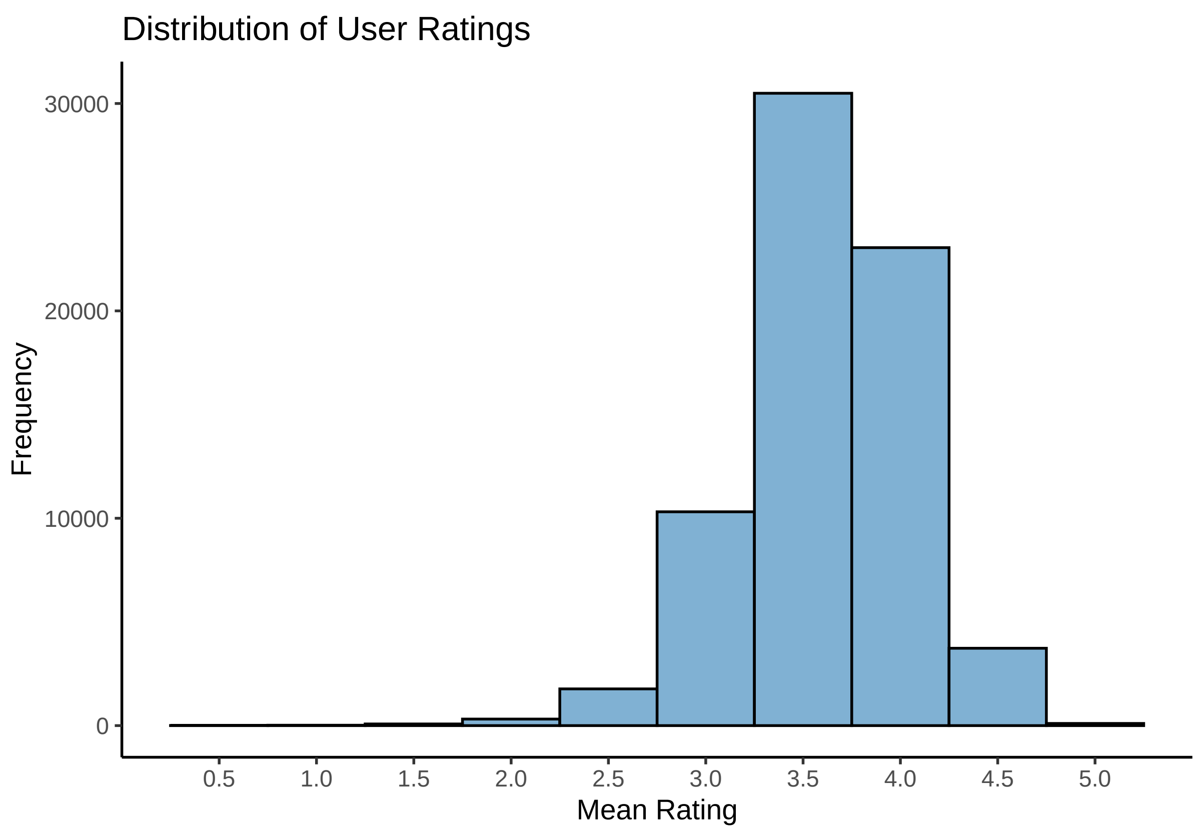
A lot of NA (grey) values here. Users that do not give a lot of ratings, are harder to estimate their preferences with accuracy.

3.1.1.3 Distribution of Users' ratings

Let's see the distribution of the ratings of the users.

```
# aggregate ratings by user
user_ratings <- edx |>
  group_by(userId) |>
  summarise(mean_rating = mean(rating))
```

```
# plot the results
ggplot(user_ratings, aes(x = mean_rating)) +
  geom_histogram(binwidth = 0.5,
                 fill = RColorBrewer::brewer.pal(10, "Set3")[5],
                 color = "black") +
  labs(title = "Distribution of User Ratings",
       x = "Mean Rating",
       y = "Frequency") +
  scale_x_continuous(breaks = seq(min_rating, max_rating, by = 0.5)) +
  #scale_y_log10() + # Add log scale on y-axis
  theme_classic()
```



We notice that ratings skew towards the higher end of the scale.

3.1.2 Movies

3.1.2.1 Average rating

We notice that the average rating of the `edx` dataset is 3.512465.

The table below shows a breakdown of how many ratings were given in total:

```
library(kableExtra) # used for styling the kable tables

total_number_of_ratings <- edx |>
  filter(!is.na(rating)) |>
  nrow()

ratings_table <- edx |> group_by(rating) |>
  summarize(count = n())

ratings_table <- ratings_table |>
  mutate(percentage = count / total_number_of_ratings)

colnames(ratings_table) <- c("Rating", "Number of ratings", "Percentage")

show_kable_table(ratings_table)
```

Rating	Number of ratings	Percentage
0.5	85374	0.009486
1.0	345679	0.038409
1.5	106426	0.011825
2.0	711422	0.079046
2.5	333010	0.037001
3.0	2121240	0.235692
3.5	791624	0.087958
4.0	2588430	0.287602
4.5	526736	0.058526
5.0	1390114	0.154456

3.1.2.2 Average movie rating distribution

The plot below confirms our previous finding. Most movies got a rating between 3 and 4:

```r
# Aggregating ratings by movieId and calculating count of ratings for each movie
movie_ratings <- edx |>
  group_by(movieId) |>
  summarise(avg_rating = mean(rating),
            rating_count = n())

# Sorting movies by rating count (optional)
movie_ratings <- movie_ratings |>
  arrange(desc(rating_count))
```

```r
# Plotting the distribution of ratings for movies
ggplot(movie_ratings, aes(x = avg_rating)) +
  geom_histogram(binwidth = 0.1,
                 fill = RColorBrewer::brewer.pal(10, "Set3")[5],
                 color = "black") +
  labs(title = "Distribution of Movie ratings by movieId",
       x = "Average Rating",
       y = "Number of Movies") +
  theme_classic()
```

3.1.3 Ratings

3.1.3.1 Ratings are in increments of 0.5 but whole numbers are preferred

Ratings are made on a 5-star scale, with half-star increments and vary from 0.5 to 5.

The plot below shows a histogram of the various ratings given by users to movies:

```r
library(ggplot2) # used for creating plots and charts

# Create histogram
binwidth <- 0.5
edx |> ggplot(aes(x = rating)) +
  geom_histogram(binwidth = binwidth,
                 fill = RColorBrewer::brewer.pal(10, "Set3")[5],
                 color = "black",
                 alpha = 0.8,
                 center = TRUE) +
  labs(title = "Ratings of the train (edx) data set", x = "Ratings", y = "Count") +
```

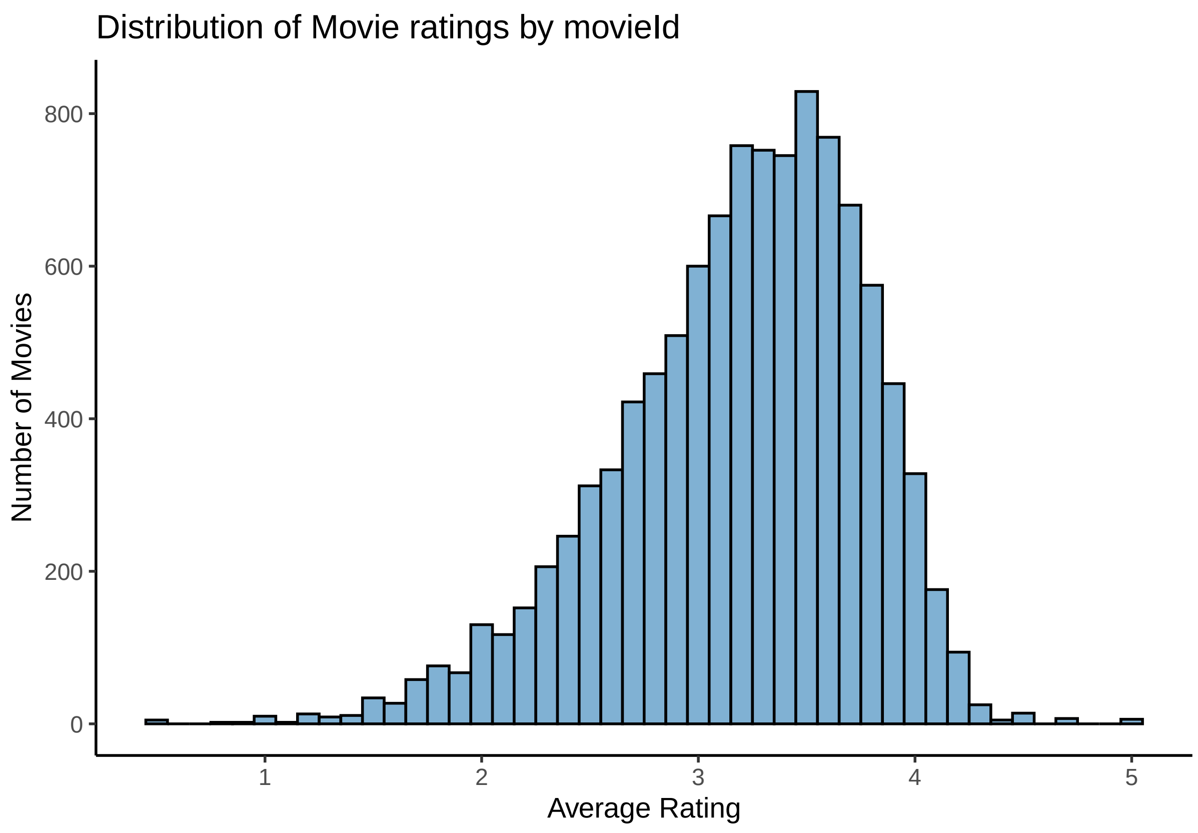


Figure 3.3: (#fig:distribution3_plot)Distribution of Movie ratings

```
  scale_x_continuous(breaks = seq(min_rating, max_rating, by = 0.5)) +
  theme_classic()
```

The histogram below shows that most movies did not get more than 8000 ratings and very few of them have more than 25000 ratings:

```
ratings_per_movie <- edx |>
  group_by(movieId) |>
  summarise(total_ratings = n()) |>
  arrange(desc(total_ratings)) |> # Sort by total ratings in ascending order
  top_n(1000)
```

```
## Selecting by total_ratings
```

```
# Plotting
binwidth <- 200
ggplot(ratings_per_movie, aes(x = total_ratings)) +
  geom_histogram(binwidth = binwidth,
                 fill = RColorBrewer::brewer.pal(10, "Set3")[5],
                 color = "black") +
  labs(
    title = "Total Number of Ratings per Movie",
    x = "Total Ratings",
    y = "Number of Movies (log 10 scale)"
  ) +
  #xlim(NA, 30000) +
  #scale_x_log10() + # Add log scale on x-axis
```

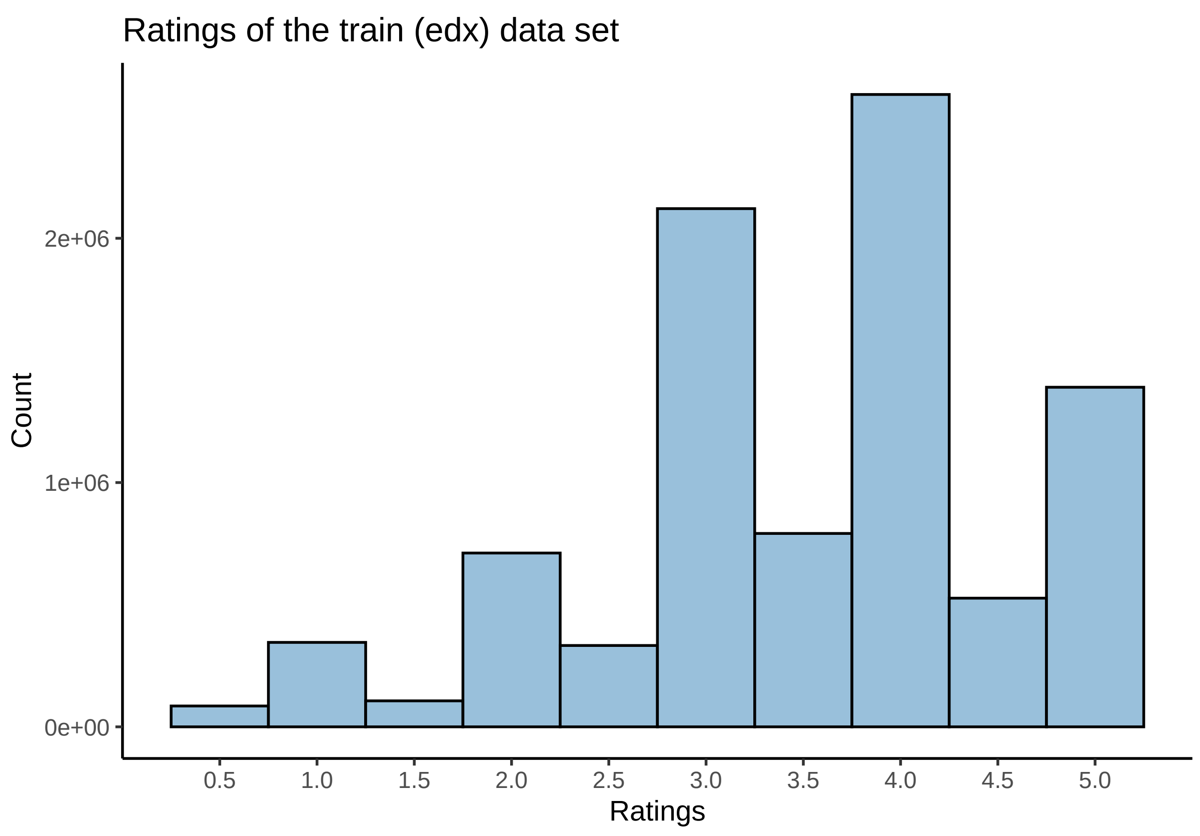


Figure 3.4: Ratings of the train (edx) data set

```
  scale_x_continuous(breaks = seq(0,
                                  max(as.numeric(ratings_per_movie$total_ratings)),
                                  by = binwidth * 20)) +
  scale_y_log10() + # Add log scale on y-axis
  theme_classic()
```

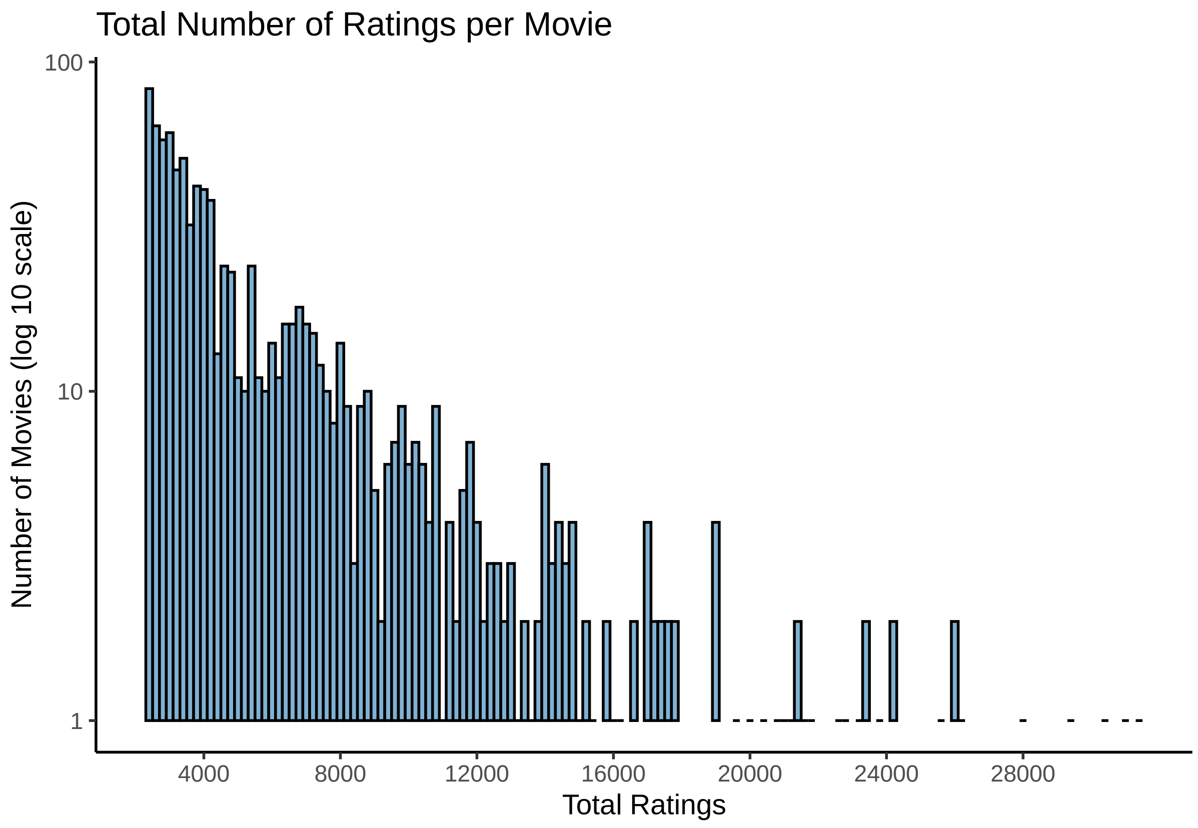


Figure 3.5: Total number of Ratings per Movie

Chapter 4

Setting the baseline

4.1 This Project's goal

Before starting our quest we need to set some baseline for the evaluation of the performance of our models. There is already a minimum threshold that defines the success of the project: achieving a RSME under **0.8649**.

In this section we will add to this minimum threshold some additional thresholds. One can argue that these additional thresholds are also models for solving the problem, but we prefer to think of them as prediction metrics when no intelligence (or very little intelligence) is used.

4.2 Guessing the rating

The code below will guess the rating with two ways: without bias (simple guessing) and with bias (guessing based on the prevalence each ratings has - which means that ratings that have a higher prevalance (e.g. the rating 3) will be chosen more often than ratings with lower prevalence (like 0.5)).

We start with unbiased guessing.

```r
suppressPackageStartupMessages(library(kableExtra)) # used for styling the kable tables

set.seed(2024)

# make random guessing of the rating a user may give to a movie
B <- nrow(final_holdout_test)
random_predictions_based_on_absolute_guessing <- data.frame(random_rating =
                                                            replicate(B,
                                                            sample(rating_range,
                                                                   size = 1,
                                                                   replace = TRUE)))

# Construct a table that will hold the results of the various models
results_overview_table <-
    tibble(Model = c("Objective of the Project is RMSE < 0.8649",
                     "Random guessing"),
                     RMSE = c(minimum_RMSE,
                              RMSE(final_holdout_test$rating,
                              random_predictions_based_on_absolute_guessing$random_rating)
                             )
          )
```

```r
# inspect results so far
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.86490
Random guessing	1.94191

So far we just made totally random guesses regarding ratings. As we expected, just guessing movie ratings, does not produce great RMSE.

Below we will try to guess again but this time we will use the prevalence each rating has in our `edx` data set. This is based on the visualization we saw in the previous Chapter (see "Ratings of the train (edx) data set") that not each rating has the same probability of being selected. For example, a rating between 3 and 4 is more common (has more probabilities to be selected) than a rating of 0.5 or 1.

```r
library(kableExtra) # used for styling the kable tables

how_many_movies_do_not_have_rating <- edx |>
  filter(!edx$rating %in% ratings) |>
  count()

if (how_many_movies_do_not_have_rating != 0) {
  print("There are movies that do not have a rating!")

} else {
  #print("All movies have ratings")
  total_ratings <- nrow(edx) # how many ratings we have?

  # calculate the prevelance of each rating (0.5, 1, 2, ..., 5)
  prevelance <- edx |>
    group_by(rating) |>
    summarize(count = n(), prevelance = n() / total_ratings) |>
    pull(prevelance)

  # guess in random again but this time, use as the probability to select
  # a rating, according to the prevalence of this rating in the edx data set
  random_predictions_based_on_biased_guessing <-
    data.frame(random_biased_rating = replicate(B, sample(rating_range,
                                                size = 1,
                                                replace = TRUE,
                                                prob = prevelance)))

  # add the result to our comparison table
  results_overview_table <- bind_rows(results_overview_table,
                                     tibble(Model = "Random guessing (biased)",
                                            RMSE = RMSE(final_holdout_test$rating,

  }

# inspect results so far
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.86490
Random guessing	1.94191
Random guessing (biased)	1.49915

Although with this last try the RMSE was improved, in general, guessing in random did not produce reliable predictions. This was to be expected as all these predictions are not intelligent and they are just useful in setting a baseline. For example, if we come up with a model that performs worse than what we can perform by guessing then we will know for sure that our model is no good.

4.3 Simple model based on average rating

We will now apply our knowledge from the Data Science course to calculate the mean rating, then add the movie effect and then the user effect when predicting.

Below we assume that any user will give the mean rating (`mu`) to any movie and we calculate the RMSE of these "predictions".

```r
# Start with the mean rating of all movies from all users
mu <- mean(edx$rating) # mean rating independently from movie and user bias

# create as many predictions as the rows of the final_holdout_test
y_hat_mean <- rep(mu, nrow(final_holdout_test))

# calculate the RMSE according to this method
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model = "Give mean rating to all movies",
                                           RMSE =
                                             RMSE(final_holdout_test$rating,
                                                  y_hat_mean)))

# inspect results so far
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.86490
Random guessing	1.94191
Random guessing (biased)	1.49915
Give mean rating to all movies	1.06120

Results improved a little, but we still have work to do.

Now we will add the movie effect (`b_i`) and see if our RMSE improves. The movie effect compensates for the fact that some movies are rated higher and some other movies are rated lower than the average rating. The code below is based partially on the previous code of the course.

```r
# estimate the movie bias which is the mean rating per movie
# minus the mean rating of all movies
b_i <- edx |>
  group_by(movieId) |>
  summarize(b_i = mean(rating - mu))
# get ratings of each movie and subtract the mean rating

b_i # Movie bias
```

```
## # A tibble: 10,677 x 2
##    movieId     b_i
##      <int>   <dbl>
##  1       1   0.415
##  2       2  -0.307
##  3       3  -0.365
##  4       4  -0.648
## 5        5 -0.444
## 6        6  0.303
## 7        7 -0.154
## 8        8 -0.378
## 9        9 -0.515
## 10      10 -0.0866
## # i 10,667 more rows
```

As we can see the movie bias is different for each movie. Some movies will take a lower rating and some a higher rating than the average rating (`mu`) when the movie bias will be incorporated in our model. We will do this now.

```
# calculate the rating based on the mean rating and the movie effect
y_hat_b_i <- final_holdout_test |>
  left_join(b_i, by = "movieId") |>
  mutate(b_i = b_i + mu) |> pull(b_i)

# inspect the results
head(y_hat_b_i)
```

```
## [1] 2.93512 3.66352 3.05565 3.53006 4.41537 2.94528
```

```
# calculate the RMSE of the movie bieas and add it to our comparison table
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model = "Mean rating + movie bias",
                                           RMSE = RMSE(final_holdout_test$rating,
                                                       y_hat_b_i)))

# inspect results so far
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.864900
Random guessing	1.941906
Random guessing (biased)	1.499153
Give mean rating to all movies	1.061202
Mean rating + movie bias	0.943909

The movie effect improved a little the RMSE of our predictions

We will proceed further to add in our mix the user effect (`b_u`). The user effect is based on the fact that some users are give mostly lower ratings (e.g. 1 and 2) while other users give mostly higher ratings (e.g. 4 and 5).

```
# calculate the bias a user has when it rates a movie
b_u <- edx |>
  left_join(b_i, by = 'movieId') |>
  group_by(userId) |>
  summarize(b_u = mean(rating - mu - b_i))  # note here that we subtract not only
                                            # the mu but also the b_i
b_u # User bias
```

```
## # A tibble: 69,878 x 2
##    userId    b_u
##     <int>  <dbl>
## 1       1  1.68
## 2       2 -0.236
##  3      3  0.264
##  4      4  0.652
##  5      5  0.0853
##  6      6  0.346
##  7      7  0.0238
##  8      8  0.203
##  9      9  0.232
## 10     10  0.0833
## # i 69,868 more rows
```

```
# calculate the y_hat for the user effect + the movie effect
y_hat_b_u <- final_holdout_test |>
  left_join(b_i, by='movieId') |>
  left_join(b_u, by='userId') |>
  mutate(y_hat = mu + b_i + b_u) |>
  pull(y_hat)

# show the results
head(y_hat_b_u)
```

```
## [1] 4.61436 5.34276 4.73489 3.29365 4.17896 2.70887
```

```
# add the RMSE of this model to our comparison table
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model="Mean rating + movie bias + user bias",
                                           RMSE = RMSE(final_holdout_test$rating,
                                                       y_hat_b_u)))

# inspect results so far
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.864900
Random guessing	1.941906
Random guessing (biased)	1.499153
Give mean rating to all movies	1.061202
Mean rating + movie bias	0.943909
Mean rating + movie bias + user bias	0.865349

```
# save the results table for later use
saveRDS(results_overview_table, file = paste0(path_of_files,
                                              "results/results_overview_table.rds"))
```

Clearly, the mean rating, the movie effect and the user effect play an important role when a user chooses a rating for a movie because our RMSE, in this last try, is the best RMSE we managed to reach so far.

In the next Chapter we will recruit some Machine Learning algorithms in our effort to find alternative ways of improving the accuracy of our predictions.

Chapter 5

Machine Learning (ML) Models

In this Chapter we will try to see if we can use Machine Learning algos (from the caret package of R) to do the training and the prediction of movie ratings.

We first tried this with the full data set (the `edx` data set) on a 12 cores Intel Xeon machine with 40 GB of RAM available. However, on all cases, the R Studio crashed due to insufficient memory. One strange thing about R is that does not just report an error message like "Insufficient memory" and stop executing your code at the point where the error occurred (which is what is happening in all programming languages we know). Instead when this error happens, the entire R Studio crashes.

So, it was necessary to have a **reduced** data set (both for training and for test) in order to just be able to run the ML algos and get some results.

We therefore selected a very small percentage (0.01 or 1%) of the original data set in order to see if the algos will be trained like this without crashing R Studio. If this proves to be successful, we would be able to increase this percentage and get even better RMSE (we believe that the more data you have, the more accurate the prediction could be).

```r
library(caret) # train and predict with ML algos
library(dplyr) # data manipulation

# libraries used for parallel execution by utilizing multiple cores of the CPU
library(doParallel)
library(parallel)

library(kableExtra) # used for styling the kable tables
```

```r
set.seed(20240423)

if (percentage_of_data_to_use == 1) {
  # use the entire data sets (100% of the data)
  train_set <- edx
  test_set <- final_holdout_test

} else {
# use a percentage_of_data_to_use of the data
  train_set <- edx |>
    slice(1:round(n() * percentage_of_data_to_use))

  test_set <- final_holdout_test |>
    slice(1:round(n() * percentage_of_data_to_use))

}
```

5.1 Machine Learning Models (Part I)

In our second attempt, we will use the very simple prediction formula:

- **rating ~ userId + movieId**

which means "*predict the 'rating' a user may give to a movie by examining the predictors (variables)* `userId` *and* `movieId` *only*".

```r
# set what we want to predict and our predictors for the job
prediction_formula <- as.formula("rating ~ userId + movieId")
```

As we learned in the course, there are many other Machine Learning models to choose from the caret package which, at the moment of writing this report, contains 239(!) models.

To somehow limit the choices, we checked which algos are preferred for use in a movie recommendation system and we selected the following:

- **Gradient Boosting Machines (GBM)**: a popular machine learning algorithm used for both, regression and classification. Our MovieLens project falls under the "regression" category.
- **Support Vector Regression (SVR)**: this algo is used for regression tasks. SVR is available in caret through the svmRadial method.
- **Random Forest**: an ensemble learning method used for both, regression and classification. This model proved to be the heavier in terms of RAM memory needed.

Since the above algos are part of the caret package we can use the `train()` function to train these models.

During the training we chose to use *cross validation* (instead of splitting the `edx` data set into train and test set).

5.1.1 Training

To increase the execution speed of our code, we will use Parallel processing. Parallel processing is the execution of multiple tasks (or calculations), simultaneously. This is done on by utilizing multiple CPU cores and it will allow the training of our model to take less time to be completed.

There are several ways to implement parallel processing in R. We selected the `doParallel` package for the job.

```r
# Set how many cores will be used in the parallel execution for
# train() and predict() below.
# I leave 2 cores free for the basic operation of the OS itself
number_of_cores <- detectCores() - 2

# # set this to TRUE if you want to use Tune Grids for the algos.
# Note that TuneGrids increase the time a model needs for training
EnableTuneGrids <- TRUE

cl <- makeCluster(number_of_cores) # create a parallel environment
registerDoParallel(cl)

# send to the parallel environment the data for training
clusterExport(cl, c("train_set"))

  #
  # 1st ML algo
```

```
#
file <- paste0(path_of_files,
               "fits/fit_svmRadial_with_",
               percentage_of_data_to_use,
               "_of_the_data.rds")
if(file.exists(file) == TRUE) {
# if we have trained this model before, just load it from its file,
# otherwise train() it then save it to its file
    fit_svmRadial <- readRDS(file)

} else {
# we have not trained this model before: do this now
    if (EnableTuneGrids == FALSE) {
      tg <- NULL
    } else {
      tg <- NULL
      #tg <- data.frame(# Number of neighbors: choose a range based on
      #   data size and complexity
      #   k = seq(3, 50, 2)
      # )
    }

    # do the training
    fit_svmRadial <- train(prediction_formula,
                    data = train_set,
                    method = "svmRadial",
                    trControl = trainControl(allowParallel = TRUE,
                                             # this runs the training in parallel
                                             verboseIter = TRUE,
                                             method = "cv"), # use cross validation
                    tuneGrid = tg
    )

    # save the fit
    saveRDS(fit_svmRadial, file = file)
    #rm(fit_svmRadial) # we do this in order to save memory;
    # we can reload the object after ALL calculations have finished
    #gc() # clear free up memory (garbage collection)

}

#
# 2nd ML algo
#
file <- paste0(path_of_files,
               "fits/fit_rf_with_",
               percentage_of_data_to_use,
               "_of_the_data.rds")
if(file.exists(file) == TRUE) {
# if we have trained this model before, just load it from its file, otherwise train()
# it then save it to its file
    fit_rf <- readRDS(file)

} else {
# we have not trained this model before: do this now
    if (EnableTuneGrids == FALSE) {
      tg <- NULL
      ns <- NULL
    } else {
      tg <- expand.grid(
        # Number of variables sampled at each split: try different ratios
        mtry = seq(2, sqrt(ncol(edx)))  # ncol is the number of predictor variables
        #splitrule = c("gini", "extratrees", "variance"),
        #ntree = c(100, 200, 500), # Number of trees: explore a range suitable
        #  for data size and complexity
        #min.node.size = seq(1, 51, 10) # Minimum size of terminal nodes: prevent
        # overfitting with large trees
      )
      ns <- seq(1, 51, 10)
    }

    # train
    fit_rf <- train(prediction_formula,
                    data = train_set,
                    method = "rf",
                    trControl = trainControl(allowParallel = TRUE,
                                             verboseIter = TRUE,
                                             method = "cv"),
                    tuneGrid = tg,
                    nodesize = ns
    )

    # save the results
    saveRDS(fit_rf, file = file)
    #rm(fit_rf) # we do this in order to save memory; we can
    # reload the object after ALL calculations have finished
    #gc() # clear free up memory (garbage collection)

}

#
# 3rd ML algo
#
file <- paste0(path_of_files, "fits/fit_gbm_with_",
               percentage_of_data_to_use,
               "_of_the_data.rds")
if(file.exists(file) == TRUE) {
    fit_gbm <- readRDS(file)

} else {
  # train the model from scratch
    if (EnableTuneGrids == FALSE) {
      tg <- NULL
    } else {
      tg <- NULL #expand.grid(parameter = seq(0.01, 1, by = 0.01)) #),
               # family = "binomial"), # binomial for classification)
    }

    # train
    fit_gbm <- train(prediction_formula,
                     data = train_set,
                     method = "gbm",  #tried: xgboost, lightgbm but they are not part
                                      # of the caret package
                     trControl = trainControl(allowParallel = TRUE,
                                              verboseIter = TRUE,
                                              method = "cv"),
                      tuneGrid = tg
      )

      # save the results
      saveRDS(fit_gbm, file = file)
      #rm(fit_gbm) # we do this in order to save memory; we can
      # reload the object after ALL calculations have finished
      #gc() # clear free up memory (garbage collection)

  }

# Stop the parallel backend
stopCluster(cl)
```

The training of all three models has been finished and we saved the fits into their respective files. Doing so will allow us to load the fits directly without having to re-train the models. Note that training is the most time (and energy) consuming task even with parallel processing that helped us a lot in finishing the calculations faster.

5.1.2 Predicting movie ratings

Now, let's do some predictions on the `test_set` by using the three fits.

```
# Make predictions
# Predict based on the Support Vector Machine (SVM) Radial model
file <- paste0(path_of_files,
               "predictions/predictions_svmRadial_with_",
               percentage_of_data_to_use,
               "_of_the_data.rds")
if(file.exists(file) == TRUE) {
# predictions have already been made. No need to do these again: just
# load them from the file.
# We do (this to save time the 2nd, 3rd, etc time we run this script)
  predictions_svmRadial <- readRDS(file)

} else {
    # do the predictions with this model
    predictions_svmRadial <- predict(fit_svmRadial,
                                     newdata = test_set)

    # save the results
    saveRDS(predictions_svmRadial, file = file)

}

# Predict based on the Random Forest model
file <- paste0(path_of_files,
               "predictions/predictions_rf_with_",
               percentage_of_data_to_use,
               "_of_the_data.rds")
if(file.exists(file) == TRUE) {
# predictions have already been made. No need to do these again: just
# load them from the file (this is to save time the 2nd, 3rd, etc
# time you run this script).
    predictions_rf <- readRDS(file)

} else {
    # predict
    predictions_rf <-predict(fit_rf,
                             newdata = test_set)

    # save the results
    saveRDS(predictions_rf, file = file)

}

# Predict based on the Gradient Boosting Machine model
file <- paste0(path_of_files,
               "predictions/predictions_gbm_with_",
               percentage_of_data_to_use,"
               _of_the_data.rds")
if(file.exists(file) == TRUE) {
# predictions have already been made. No need to do these again: just load them
# from the file (we do this to save time the 2nd, 3rd, etc time you run this script)
    predictions_gbm <- readRDS(file)

} else {
    # predict
    predictions_gbm <- predict(fit_gbm,
                               newdata = test_set)

    # save the results
    saveRDS(predictions_gbm, file = file)

}
```

Three predictions were made based on each of our three models respectively.

Below we will calculate the RMSE of each model and will summarize the results to our comparison table.

```
# read our results table
results_overview_table <- readRDS(paste0(path_of_files,
                                         "results/results_overview_table.rds"))

# add the RMSE of the prediction results into the comparison table
# add the results of the SVM model
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model =
                                             paste0("Support Vector Machine (SVM) ",
                                                    "Radial (with ",
                                                    percentage_of_data_to_use,
                                                    " of the data)"),
                                           RMSE =
                                             RMSE(test_set$rating,
                                                  predictions_svmRadial)))

# add the results of the RF model
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model = paste0("Random Forest (with ",
                                                          percentage_of_data_to_use,
                                                          " of the data)"),
                                           RMSE = RMSE(test_set$rating,
                                                       predictions_rf)))

# add the results of GBM model
```

```r
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model =paste0("Gradient Boosting Machine ",
                                                         "(with ",
                                                         percentage_of_data_to_use,
                                                         " of the data)"),
                                           RMSE = RMSE(test_set$rating,
                                                       predictions_gbm)))

# the GBM model seems that it estimated with greater accuracy. Save its
# RMSE for future use
RMSE_of_gbm <- RMSE(test_set$rating, predictions_gbm)
saveRDS(RMSE_of_gbm, file = paste0(path_of_files, "objects/RMSE_of_gbm.rds"))
```

The predictions are finished in reasonable time. In general, predictions are faster than the training of a model, especially if we have a lot of data.

5.1.3 Conclusion

Using only 0.01 of the `edx` (train set) and the `final_holdout_test` (test set), the last three machine learning models that we trained (via the caret package), failed to beat either the threshold of 0.8649 or the "Mean rating + movie bias + user bias" model. The results are showing on the three last lines of the table below:

```r
# print the results so far
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.864900
Random guessing	1.941906
Random guessing (biased)	1.499153
Give mean rating to all movies	1.061202
Mean rating + movie bias	0.943909
Mean rating + movie bias + user bias	0.865349
Support Vector Machine (SVM) Radial (with 0.01 of the data)	1.064084
Random Forest (with 0.01 of the data)	1.063379
Gradient Boosting Machine (with 0.01 of the data)	0.999810

From the results table it is obvious that:

- All machine learning models produced an RMSE which is *above* the threshold of **0.8649**, which means that failed to accomplish our goal. Even the **Gradient Boosting Machine** model that performed better that any other ML model, had an RMSE = **0.99981**. Our experiments showed that more data will yield better (more accurate) results because a larger part of the reality is captured on the additional data and therefore the prediction will inevitably be more accurate.

- The data that were used for training and test were an extremely small sample (0.01) of the data sets `edx` and `final_holdout_test`. This percentage was selected in order to allow us see what will be the cost in time and energy for training our Machine Learning models with just a fraction of the entire data set. And although the percentage of data to use was extremely small, the time that required to train these three models easily exceeded the 24Hours on a machine with technical specifications (on CPU, GPU and RAM memory) above the technical specifications of an average home computer today. This leads to the conclusion that any percentage of data above 0.01 and surely for the 100% of the data, will require computing capabilities that are comparable to those of a GPU Server.

- It is evident that the successful completion of the goal of this project with a Machine Learning model, requires a different plan of attack (a different approach) since access to (or ownership of) a GPU Server is not common (a GPU Server may costs today around 250.000 USD depending on the configuration selected).

On the next Chapter we will explore a Machine Learning algorithm that is designed especially for this kind of problem and it works efficiently with large amounts of data like the data on hand.

5.2 Machine Learning Models (Part II)

In our third try we researched for the existence of Machine Learning models that are specialized in solving the problem set on this project. This is similar to looking for specific ML models inside the caret package that can solve a problem you have, only this time you are looking everywhere (in the models included in the caret package and outside of it).

We set some minimum criteria for researching and selecting our next Machine Learning Model candidate. According to these criteria, our next ML model candidate must:

- be specialized on this kind of problem (recommender system algorithms) because each tool is designed for a specific task. When you have a nail you use a hammer, not a screwdriver.
- can handle easily (without crashing R Studio) and efficiently (without you to have to wait days or weeks for the train to be completed) the size of 9000055 observations found inside the `edx` data set. We should note here that in this project we were given a limited version (only 10 million ratings) of the original MovieLens data set which contains around 33 million ratings in total[1]. If our algo fails to calculate a fit and make predictions on such large data sets, it will be impossible to be of any use to the original, full MovieLens data set or anything larger than that.

Our research pointed to various algorithms that are promising in achieving our set goal:

- **Collaborative Filtering** (which are further divided into User-Based Collaborative Filtering and Item-Based Collaborative Filtering)
- **Hybrid Recommender Systems** which are combining Collaborative Filtering and Content-Based Filtering methods together
- **Matrix Factorization Techniques** (which can use a Singular Value Decomposition (SVD) algorithm or an Alternating Least Squares (ALS) algorithm)
- **Ensemble methods** that can combine the results of various algos into a voting system that possibly can reduce the RMSE even further

From the above we selected the **Matrix Factorization** Techniques to begin with, and see if we it we can achieve our goal with its help. For that purpose we selected the **recommenderlab** package in R[2] which is a "*Recommender System using Matrix Factorization*". Recosystem is an R wrapper of the LIBMF library developed by Yu-Chin Juan, Wei-Sheng Chin, Yong Zhuang, Bo-Wen Yuan, Meng-Yuan Yang, and Chih-Jen Lin (https://www.csie.ntu.edu.tw/~cjlin/libmf/), an open source library for recommender system using parallel marix factorization (Chin, Yuan, et al. 2015). This package offers a framework for developing and evaluating recommender systems. It contains various algorithms, evaluation metrics and utilities for building and assessing recommendation models.

There is also additional reading material that one can read on this topic[3].

[1]https://grouplens.org/datasets/movielns/

[2]https://cran.r-project.org/web/packages/recosystem/index.html and https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html

[3]"Building a Recommendation System with R" by Suresh K. Gorakala and "How to Code a Recommendation System in R" by Ander Fernández

5.2.1 Training

In this attempt we will use the `recosystem` library.

We start by loading the library and **converting** our data sets into the format that the `recosystem` library recognizes. This is necessary in order to be able to train our model and then make predictions on the test data.

```r
library(parallel) # used for parallel processing
library(recosystem) # building, evaluating recommendation systems

library(kableExtra) # used for styling the kable tables

set.seed(20240424)

percentage_of_data_to_use <- 1 # how much data to take from the train set (1 = 100%)

file <- paste0(path_of_files,
               "predictions/y_hat_rec_MF_with_",
               percentage_of_data_to_use,
               "_of_the_data.rds")
if(file.exists(file) == TRUE) {
# if the algo is already trained and its predictions already made,
# load its predictions from the corresponding file
    y_hat_rec_MF <- readRDS(file)

} else {
  # set how many cores will be used in the parallel execution for training
  number_of_cores <- detectCores() - 2

  # specify the indices that point to the user, movie and rating in the edx (train)
  # data set.
  train_set_reco <- with(edx, data_memory(user_index = userId,
                                          item_index = movieId,
                                          rating = rating))

  # do the same for the final_holdout_test (test) data set.
  test_set_reco <- with(final_holdout_test, data_memory(user_index = userId,
                                                        item_index = movieId,
                                                        rating = rating))

  #
  # Note that we use the ENTIRE edx and final_holdout_test
  # (not just a percentage of them)
  #

  # initialize a new object of a recommendation system
  recommendation_system <- Reco()

  # set the tuning parameters
  tuning_MF <- recommendation_system$tune(train_set_reco,
                                          opts = list(dim = c(10, 20, 30, 40),
                                                      lrate = c(0.1, 0.2, 0.3, 0.4),
                                                      nthread  = number_of_cores,
                                                      niter = 20))

  # do the training with the train set
  recommendation_system$train(train_set_reco,
                              opts = c(tuning_MF$min,
                                              nthread = number_of_cores,
                                              niter = 20)
                                    )

  # do the predictions on the test set
  y_hat_rec_MF <- recommendation_system$predict(test_set_reco, out_memory())

  # save the results of the predictions
  saveRDS(y_hat_rec_MF, file = file)

}
```

5.2.2 Predicting movie ratings

Once the training is finished, which was rather quick considering that we feed the **entire** data set, the `y_hat_rec_MF` now contains our predictions. We use that to evaluate the RMSE of this model.

```
library(dplyr)

# calculate the RMSE of the predictions of our model
RMSE_of_recommender_system <- RMSE(final_holdout_test$rating, y_hat_rec_MF)

# save the results to this file for future use
saveRDS(RMSE_of_recommender_system,
        file = paste0(path_of_files,
                      "objects/RMSE_of_recommender_system.rds"))

# add the results to the comparison table
results_overview_table <- bind_rows(results_overview_table,
                                    tibble(Model =
                                             paste0("Matrix factorization",
                                               " (with 100% of the data)"),
                                           RMSE = RMSE_of_recommender_system))
```

5.2.3 Conclusion

The result of our latest effort is showing on the last line of the table below:

```
# print the comparison table that contains all the models
show_kable_table(results_overview_table)
```

Model	RMSE
Objective of the Project is RMSE < 0.8649	0.864900
Random guessing	1.941906
Random guessing (biased)	1.499153
Give mean rating to all movies	1.061202
Mean rating + movie bias	0.943909
Mean rating + movie bias + user bias	0.865349
Matrix factorization (with 100% of the data)	0.779908

This last method (Matrix Factorization) succeeded in beating the threshold of **0.8649** by **0.084992** (RMSE achieved = **0.779908**). Thus our goal has been achieved.

It is worth mentioning that:

- during the training of the model the end user is able to see a nice 0% to 100% progress bar that looks like this:

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

This bar gives you an idea of how long the whole process will take and how much it remains for the completion of the calculation. This is very useful and we hope to see this also in the caret package.

- the Matrix Factorization (MF) method achieved this result with the **full** data set (edx) with **all** 9000055 observations, not just a percentage of it. Judging from the time needed for training this model as well as the way this algo was designed, we believe that this method can also handle larger data sets, by utilizing the computing power and the limiter RAM memory of an average home computer, without any problem.

Chapter 6

Conclusion

The project developed a recommender system based on limited MovieLens dataset (of 10 million user ratings given for movies). We did a data exploration of the data set and inspected the data visually.

We set the minimum RSME and some unintelligent models as our baseline.

Then we examined various models in order to see which performs best:

- two naive models:
  - Random guessing of rating
  - Random guessing of rating (biased on the probability calculated by the prevalence)
- three models based on the:
  - Mean rating (`mu <- mean(edx$rating)`)
  - Mean rating + movie bias (`mu + b_i`)
  - Mean rating + movie bias + user bias (`mu + b_i + b_u`)
- three Machine Learning Models from the `caret` package:
  - a Support Vector Machine (SVM) Radial model
  - a Random Forest (RF) model
  - a Gradient Boosting Machine (GBM) model
- a Matrix Factorization (MF) model from the `recommeder` package

The last model proved to be the winner since it produced a RMSE of **0.779908** on the `final_holdout_test`.

6.1 Ideas for further exploration

Further improvement may be accomplished by:

- altering the tuning parameters of the Matrix Factorization (MF) model
- trying other methods included in the `recommender` package (like the User-Based Collaborative Filtering (UBCF), the Item-Based Collaborative Filtering (IBCF), etc)
- trying any other Matrix Factorization (MF) model apart from those included in the `recommender` package
- using Regularization (see page 645 of the "Introduction to Data Science book.pdf") and the tuning parameter λ in order to penalize predictions that are based on small sample sizes.
- adding/producing any additional variables (predictors) in the train data set. According to our experiments this is not always the case, as more predictors may improve or deteriorate the results. However, it is a path worth exploring since in this report we only used the three variables (`userId, movieId, rating`) which were already provided in the original `edx` data set.

Chapter 7

Appendix

7.1 System Information

For the shake of reproducability we include below the specifications of the machine (computer) that was used for carrying out our experiments and the successful completion of this project.

This project was carried out on a machine with the following hardware and software characteristics. In case you try to reproduce the results and your R Studio is crashing, please make sure that your machine has these minimum hardware requirements (especially regarding the Total R.A.M Memory available).

```
## [1] "OS: Linux version #115-Ubuntu SMP Mon Apr 15 09:52:04 UTC 2024 (release: 5.15.0-105-generic)
```

```
## [1] "Virtual Machine: x86_64"
```

```
## [1] "CPU: 2 x Intel Xeon E5-2697 v2 @ 2.70 GHz (from which we used 1 processor with 12 cores)"
```

```
## [1] "Total R.A.M memory: 40 GB"
```

7.2 Session Info

In the section below the version information of R, the OS and attached (or loaded packages) are showing.

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.4 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=el_GR.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=el_GR.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=el_GR.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=el_GR.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] recosystem_0.5.1   doParallel_1.0.17  iterators_1.0.14   foreach_1.5.2
##  [5] caret_6.0-94       lattice_0.22-5     RColorBrewer_1.1-3 tidyr_1.3.1
##  [9] ggplot2_3.4.4      dplyr_1.1.4        kableExtra_1.4.0
##
## loaded via a namespace (and not attached):
##  [1] viridisLite_0.4.2    splines_4.1.2        prodlim_2023.08.28
##  [4] highr_0.9            stats4_4.1.2         yaml_2.3.8
##  [7] globals_0.16.3       ipred_0.9-14         pillar_1.9.0
## [10] glue_1.7.0           pROC_1.18.5          digest_0.6.34
## [13] hardhat_1.3.1        colorspace_2.1-0     recipes_1.0.10
## [16] htmltools_0.5.7      Matrix_1.6-5         plyr_1.8.9
## [19] timeDate_4032.109    pkgconfig_2.0.3      listenv_0.9.1
## [22] bookdown_0.39        purrr_1.0.2          scales_1.3.0
## [25] svglite_2.1.3        gower_1.0.1          lava_1.8.0
## [28] timechange_0.3.0     tibble_3.2.1         generics_0.1.3
## [31] farver_2.1.1         withr_3.0.0          nnet_7.3-19
## [34] cli_3.6.2            survival_3.5-7       magrittr_2.0.3
## [37] evaluate_0.23        float_0.3-2          fansi_1.0.6
## [40] future_1.33.2        parallelly_1.37.1    nlme_3.1-155
## [43] MASS_7.3-60.0.1      xml2_1.3.6           class_7.3-22
## [46] tools_4.1.2          data.table_1.15.0    lifecycle_1.0.4
## [49] stringr_1.5.1        kernlab_0.9-32       munsell_0.5.0
## [52] compiler_4.1.2       systemfonts_1.0.5    rlang_1.1.3
## [55] grid_4.1.2           rstudioapi_0.15.0    labeling_0.4.3
## [58] rmarkdown_2.26       gtable_0.3.4         ModelMetrics_1.2.2.2
## [61] codetools_0.2-19     reshape2_1.4.4       R6_2.5.1
## [64] lubridate_1.9.3      knitr_1.45           fastmap_1.1.1
## [67] future.apply_1.11.2  utf8_1.2.4           stringi_1.8.3
## [70] Rcpp_1.0.12          vctrs_0.6.5          rpart_4.1.23
## [73] tidyselect_1.2.0     xfun_0.41
```

```r
# create a bib db of R packages
knitr::write_bib(c(.packages(), "bookdown", "knitr", "rmarkdown"), "packages.bib")
```

```
## tweaking foreach
```