Capstone Movielens Report

Azamat Kurbanaev

2025-05-12

Contents

0.1 Introduction / Overview / Executive Summary

The goal of the project is to build a Recommendation System using a 10M version of the MovieLens dataset. Following the Netflix Grand Prize Contest requirements, we will evaluate the *Root Mean Square Error* (*RMSE*) score, which, as shown in Section 23.2 Loss function of the *Course Textbook*, is defined as:

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

with N being the number of user/movie combinations for which we make predictions and the sum occurring over all these combinations(Irizarry 2024b).

Our goal is to achieve a value of less than 0.86490 (compare with the *Netflix Grand Prize* requirement: of at least 0.8563(Andreas Toscher 2009)).

0.1.1 Datasets Overview

To start with we have to generate two datasets derived from the *MovieLens* one mentioned above:

- **edx:** we use it to develop and train our algorithms;
- **final__holdout__test:** according to the course requirements, we use it exclusively to evaluate the ***RMSE*** of our final algorithm.

For this purpose the following package has been developed by the author of this report: `edx.capstone.movielens.data`. The source code of the package is available on GitHub(Kurbanaev 2025a).

Let's install the development version of this package from the GitHub repository and attach the correspondent library to the global environment:

```r
if(!require(edx.capstone.movielens.data)) pak::pak("AzKurban-edX-DS/edx.capstone.movielens.data")

library(edx.capstone.movielens.data)
edx <- edx.capstone.movielens.data::edx
final_holdout_test <- edx.capstone.movielens.data::final_holdout_test
```

Now, we have the datasets listed above:

```
summary(edx)
```

```
##      userId         movieId          rating        timestamp            title              genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:9000055     Length:90000
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08   Class :character   Class :chara
##  Median :35738   Median : 1834   Median :4.000   Median :1.035e+09   Mode  :character   Mode  :chara
##  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
```

```
summary(final_holdout_test)
```

```
##      userId         movieId          rating        timestamp            title              genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:999999      Length:99999
##  1st Qu.:18096   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.467e+08   Class :character   Class :chara
##  Median :35768   Median : 1827   Median :4.000   Median :1.035e+09   Mode  :character   Mode  :chara
##  Mean   :35870   Mean   : 4108   Mean   :3.512   Mean   :1.033e+09
##  3rd Qu.:53621   3rd Qu.: 3624   3rd Qu.:4.000   3rd Qu.:1.127e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
```

0.1.1.1 `edx` Dataset

Let's look into the details of the `edx` dataset:

```
str(edx)
```

```
## 'data.frame':    9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : int  122 185 292 316 329 355 356 362 364 370 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 838984474 838983653 838984885 8
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|A
```

Note that we have 9000055 rows and six columns in there:

```
dim_edx <- dim(edx)
print(dim_edx)
```

```
## [1] 9000055       6
```

First, let's note that we have 10677 different movies:

```
n_movies <- n_distinct(edx$movieId)
print(n_movies)
```

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

and 69878 different users in the dataset:

```r
n_users <- n_distinct(edx$userId)
print(n_users)
```

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

Now, note the expressions below which confirm the fact explained in Section *23.1.1 Movielens data* of the *Course Textbook*(Irizarry 2024h) that not every user rated every movie:

```r
max_possible_ratings <- n_movies*n_users
sprintf("Maximum possible ratings: %s", max_possible_ratings)
```

```
## [1] "Maximum possible ratings: 746087406"
```

```r
sprintf("Rows in `edx` dataset: %s", dim_edx[1])
```

```
## [1] "Rows in ‘edx‘ dataset: 9000055"
```

```r
sprintf("Not every movie was rated: %s", max_possible_ratings > dim_edx[1])
```

```
## [1] "Not every movie was rated: TRUE"
```

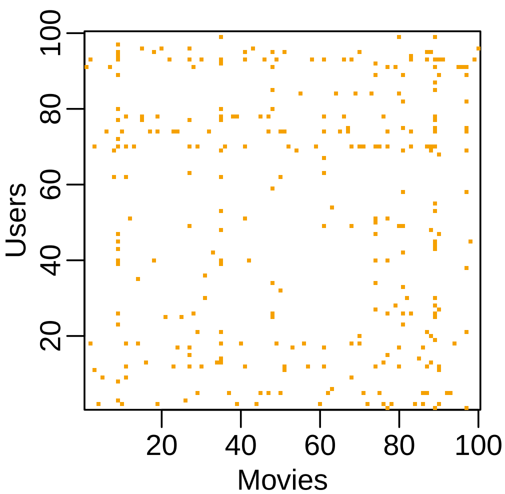
As also explained in that section, we can think of these data as a very large matrix, with users on the rows and movies on the columns, with many empty cells. Therefore, we can think of a recommendation system as filling in the `NA`s in the dataset for the movies that some or all the users do not rate. A sample from the `edx` data below illustrates this idea(Irizarry 2024a):

```r
keep <- edx |>
  dplyr::count(movieId) |>
  top_n(4, n) |>
  pull(movieId)

tab <- edx |>
  filter(movieId %in% keep) |>
  filter(userId %in% c(13:20)) |>
  select(userId, title, rating) |>
  mutate(title = str_remove(title, ", The"),
         title = str_remove(title, ":.*")) |>
  pivot_wider(names_from = "title", values_from = "rating")

print(tab)
```

```
## # A tibble: 5 x 5
##   userId ‘Pulp Fiction (1994)‘ ‘Jurassic Park (1993)‘ ‘Silence of the Lambs (1991)‘ ‘Forrest Gump (19
##    <int>                 <dbl>                  <dbl>                          <dbl>
## 1     13                     4                     NA                             NA
## 2     16                    NA                      3                             NA
## 3     17                    NA                     NA                              5
## 4     18                     5                      3                              5
## 5     19                    NA                      1                             NA
```

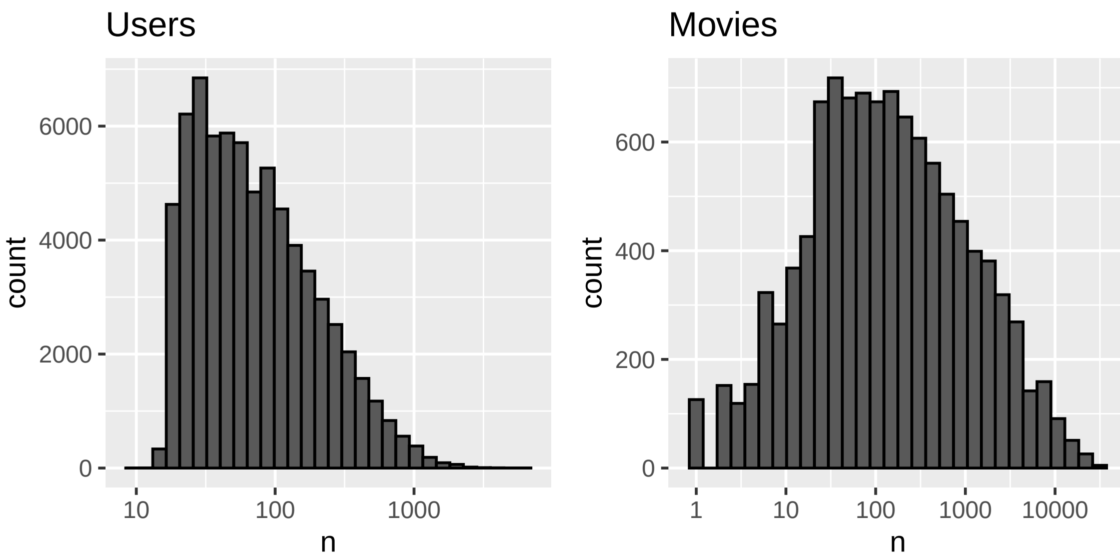
The following plot of the matrix for a random sample of 100 movies and 100 users with yellow indicating a user/movie combination for which we have a rating shows how *sparse* the matrix is:

```
users <- sample(unique(edx$userId), 100)

rafalib::mypar()
edx|>
  filter(userId %in% users) |>
  select(userId, movieId, rating) |>
  mutate(rating = 1) |>
  pivot_wider(names_from = movieId, values_from = rating) |>
  (\(mat) mat[, sample(ncol(mat), 100)])() |>
  as.matrix() |>
  t() |>
  image(1:100, 1:100, z = _ , xlab = "Movies", ylab = "Users")
```



Further observations highlighted there that, as we can see from the distributions the author presented, some movies get rated more than others, and some users are more active than others in rating movies:

```
p1 <- edx |>
  count(movieId) |>
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  ggtitle("Movies")

p2 <- edx |>
  count(userId) |>
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  ggtitle("Users")

gridExtra::grid.arrange(p2, p1, ncol = 2)
```



Finally, we can see that no movies have a rating of 0. Movies are rated from 0.5 to 5.0 in 0.5 increments:

```
#library(dplyr)
s <- edx |> group_by(rating) |>
  summarise(n = n())
print(s)
```

```
## # A tibble: 10 x 2
##    rating       n
##     <dbl>   <int>
##  1    0.5   85374
##  2    1    345679
##  3    1.5  106426
##  4    2    711422
##  5    2.5  333010
##  6    3   2121240
##  7    3.5  791624
##  8    4   2588430
##  9    4.5  526736
## 10    5   1390114
```

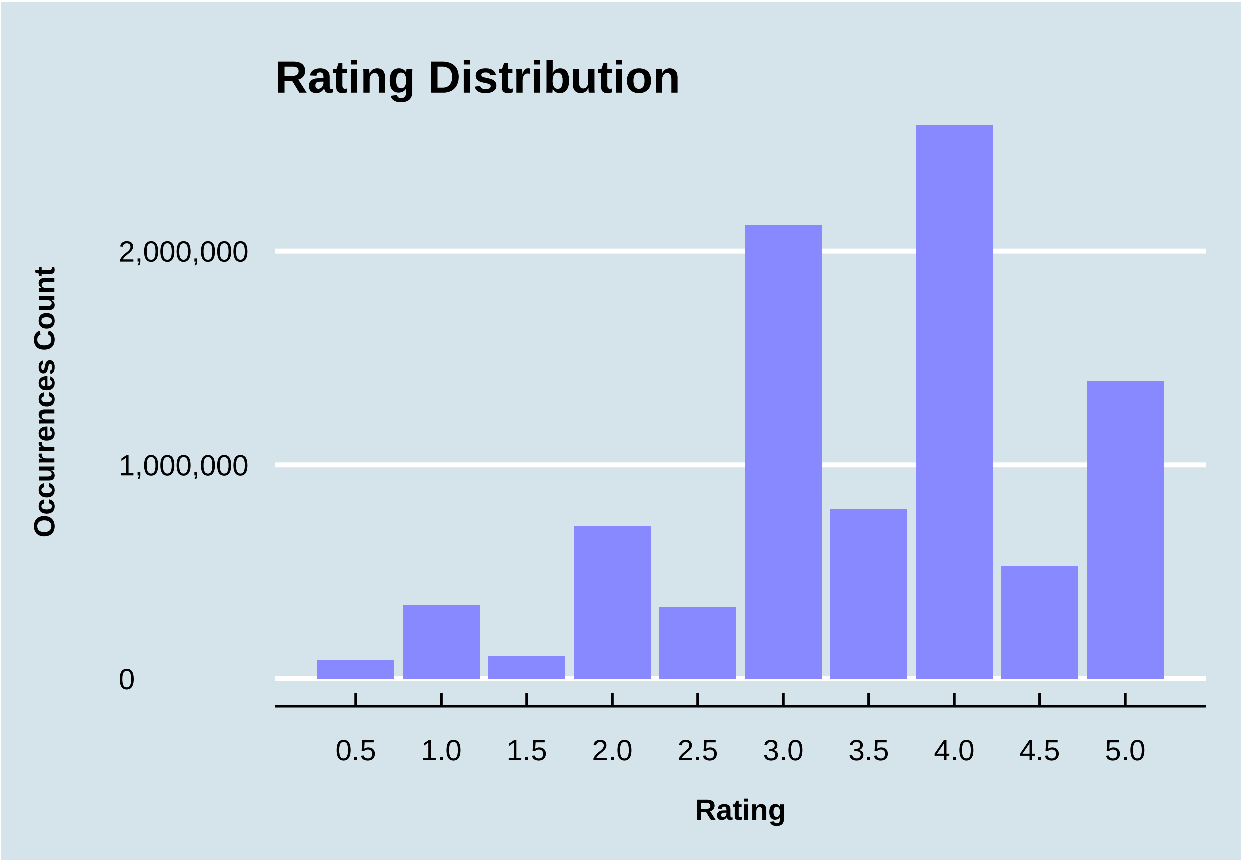
Further analysis of the `edx` dataset have been also inspired by the article mentioned above(Motefaker 2024), from which the code and explanatory notes below were cited.

0.1.1.1.1 Rating distribution plot(Motefaker 2024)

The code below demonstrates another way of visualizing the rating distribution:

```
edx |>
  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") +
```

```r
  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"))
```



This graph is another confirmation of what we found out above: rounded ratings occur more often than half-stared ones. The upward trend previously discussed is now perfectly clear, although it seems to top right between the 3 and 4-star ratings lowering the occurrences count afterward. That might be due to users being more hesitant to rate with the highest mark for whichever reasons they might hold(Motefaker 2024).

0.1.1.1.2 Ratings per movie

Movie popularity count(Motefaker 2024)

```r
print(edx |>
  group_by(movieId) |>
  summarize(count = n()) |>
  slice_head(n = 10)
)
```

```
## # A tibble: 10 x 2
##    movieId count
##      <int> <int>
##  1       1 23790
##  2       2 10779
##  3       3  7028
##  4       4  1577
##  5       5  6400
##  6       6 12346
##  7       7  7259
##  8       8   821
##  9       9  2278
## 10      10 15187
```

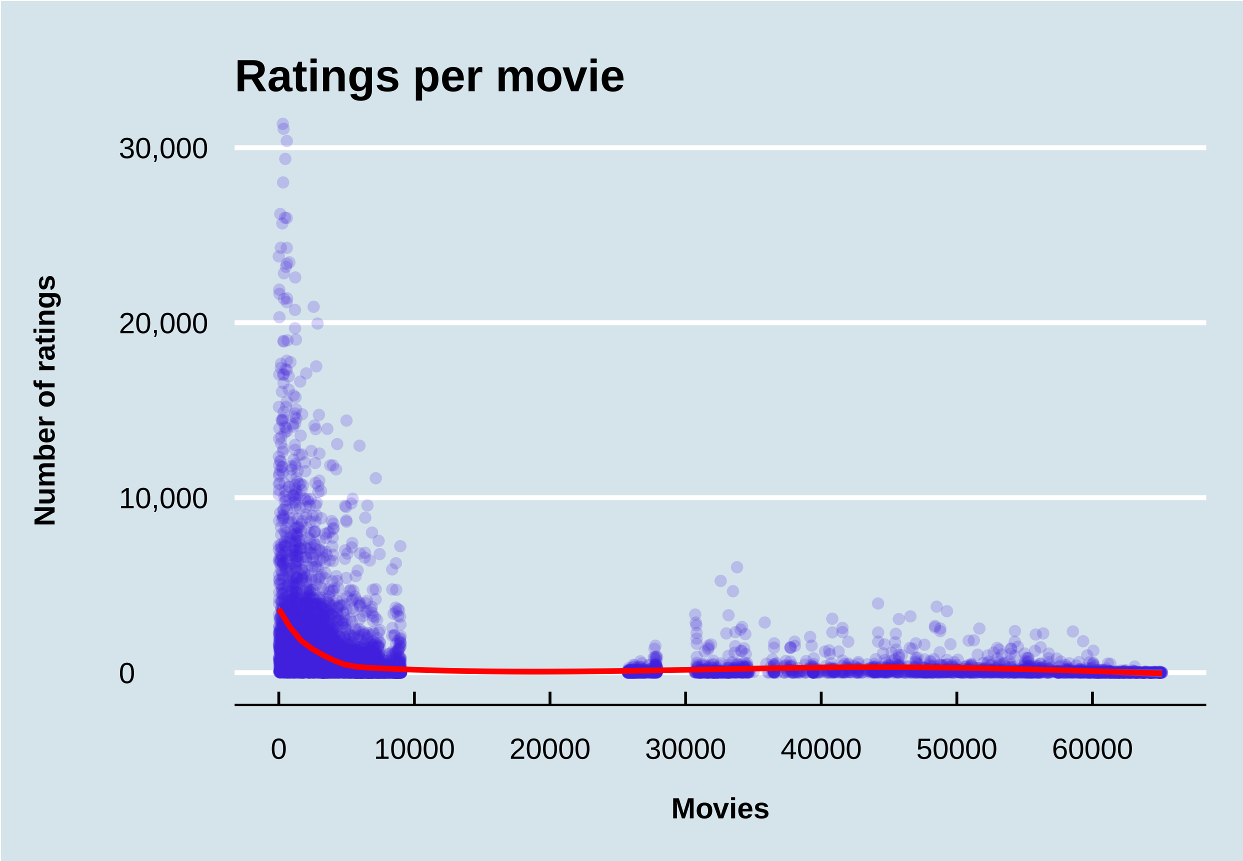
```
summary(edx |> group_by(movieId) |> summarize(count = n()) |> select(count))
```

```
##      count
##  Min.   :    1.0
##  1st Qu.:   30.0
##  Median :  122.0
##  Mean   :  842.9
##  3rd Qu.:  565.0
##  Max.   :31362.0
```

Ratings per movie plot(Motefaker 2024)

```
edx |>
  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"))
```

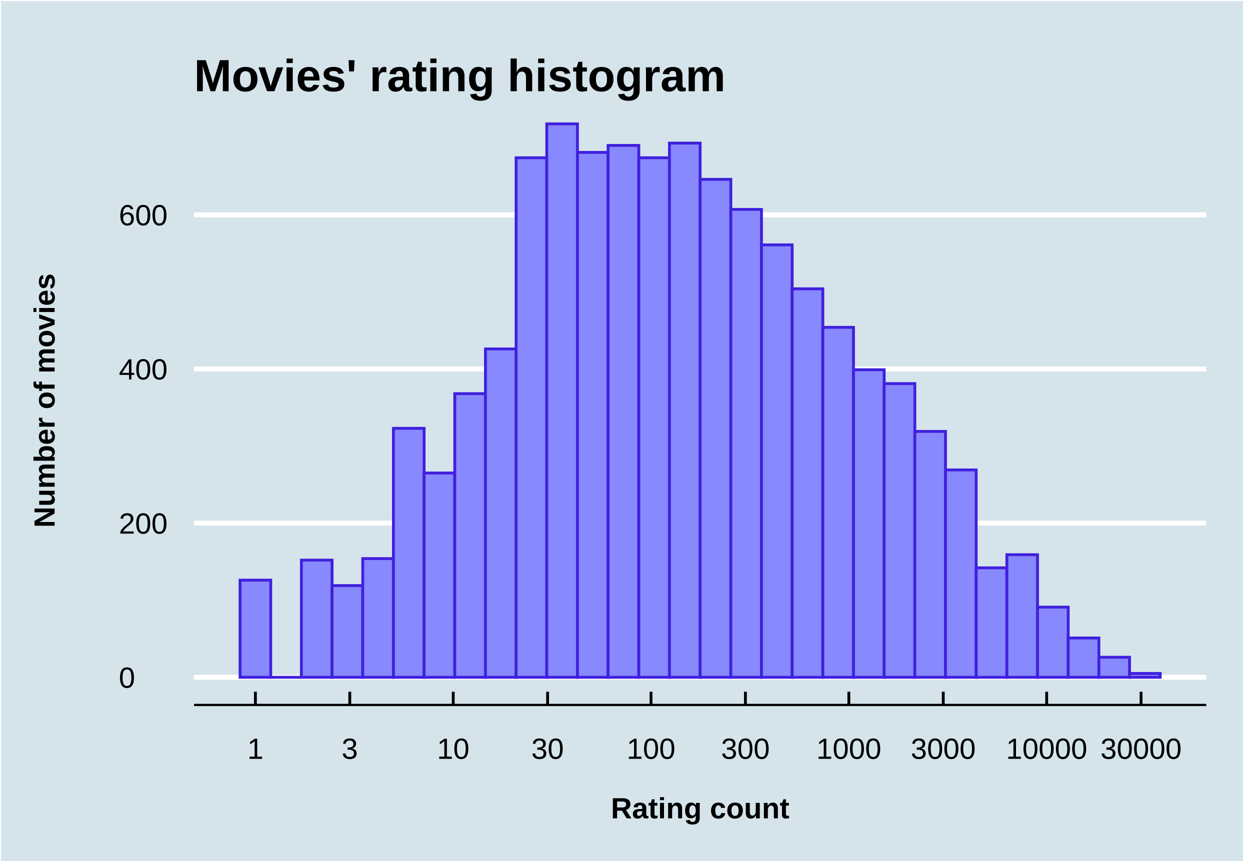
```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



Movies' rating histogram(Motefaker 2024)

```
edx |>
  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"))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



0.1.1.1.3 Ratings per user(Motefaker 2024)

User rating count (activity measure)

```
print(edx |>
  group_by(userId) |>
  summarize(count = n()) |>
  slice_head(n = 10)
)
```

```
## # A tibble: 10 x 2
##    userId count
##     <int> <int>
##  1      1    19
##  2      2    17
##  3      3    31
##  4      4    35
##  5      5    74
##  6      6    39
##  7      7    96
##  8      8   727
##  9      9    21
```

```
## 10      10    112
```

User rating summary

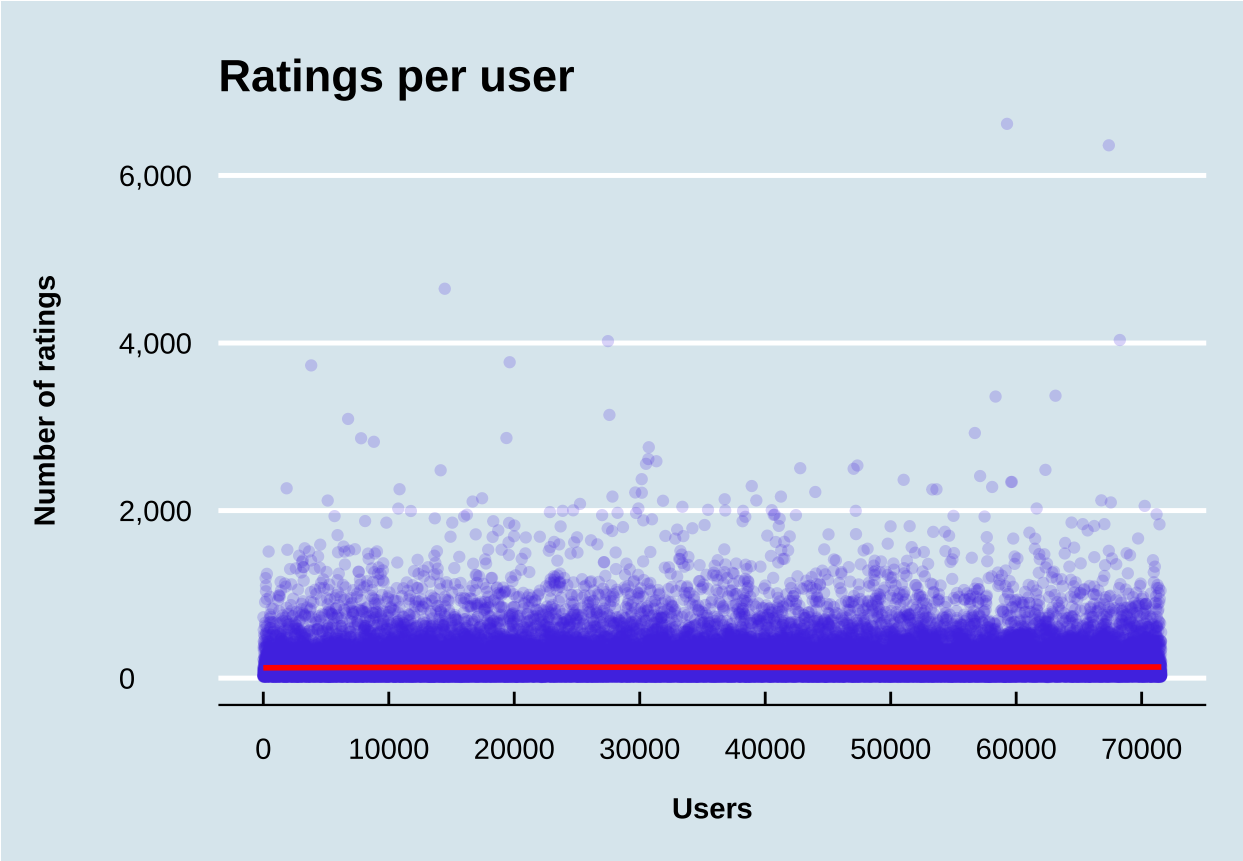
```r
summary(edx |> group_by(userId) |> summarize(count = n()) |> select(count))
```

```
##      count
##  Min.   :  10.0
##  1st Qu.:  32.0
##  Median :  62.0
##  Mean   : 128.8
##  3rd Qu.: 141.0
##  Max.   :6616.0
```

Ratings per user plot

```r
edx |>
  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"))
```

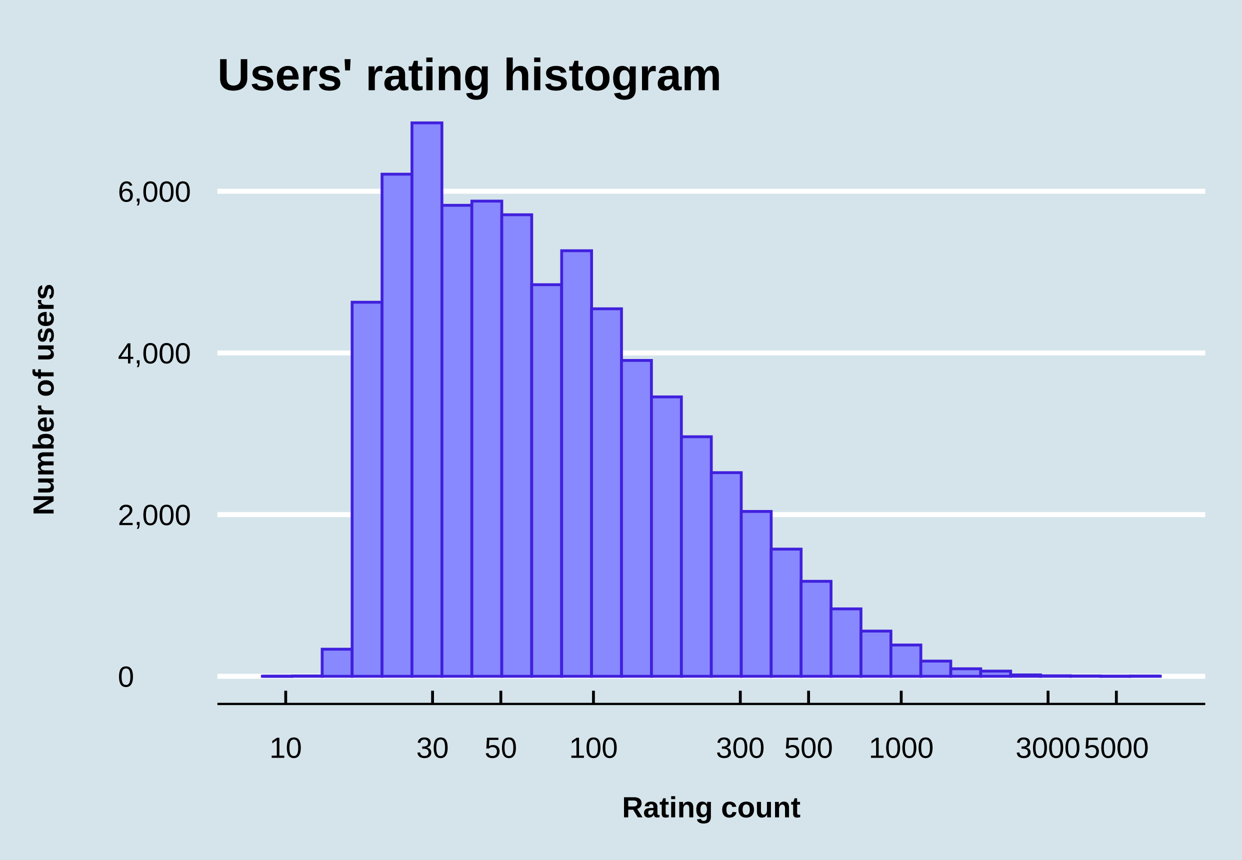
```
## 'geom_smooth()' using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



Users' rating histogram

```
edx |>
  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"))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



0.2 Methods / Analysis

> All the source code of the R-scripts is available on the project's GitHub repository(Kurbanaev 2025b).

0.2.1 Defining Logging and Time Measuring Helper Functions

First, let's define some helper functions for logging and time-measuring features that we will use in our R scripts. Some of them are listed below:

```r
## Logging Helper functions ----------------------------------------------------
open_logfile <- function(file_name){
  log_file_name <- as.character(Sys.time()) |>
    str_replace_all(':', '_') |>
    str_replace(' ', 'T') |>
    str_c(file_name)

  log_open(file_name = log_file_name)
}
print_start_date <- function(){
```

```r
  print(date())
  Sys.time()
}
put_start_date <- function(){
  put(date())
  Sys.time()
}
print_end_date <- function(start){
  print(date())
  print(Sys.time() - start)
}
put_end_date <- function(start){
  put(date())
  put(Sys.time() - start)
}

msg.set_arg <- function(msg_template, arg, arg.name = "%1") {
  msg_template |>
    str_replace_all(arg.name, as.character(arg))
}
msg.glue <- function(msg_template, arg, arg.name = "%1"){
  msg_template |>
    msg.set_arg(arg, arg.name) |>
    str_glue()
}

print_log <- function(msg){
  print(str_glue(msg))
}
put_log <- function(msg){
  put(str_glue(msg))
}

get_log1 <- function(msg_template, arg1) {
  str_glue(str_replace_all(msg_template, "%1", as.character(arg1)))
}
print_log1 <- function(msg_template, arg1){
  print(get_log1(msg_template, arg1))
}
put_log1 <- function(msg_template, arg1){
  put(get_log1(msg_template, arg1))
}

get_log2 <- function(msg_template, arg1, arg2) {
  msg_template |>
    str_replace_all("%1", as.character(arg1)) |>
    str_replace_all("%2", as.character(arg2)) |>
    str_glue()
}
print_log2 <- function(msg_template, arg1, arg2){
  print(get_log1(msg_template, arg1, arg2))
}
put_log2 <- function(msg_template, arg1, arg2){
```

```r
  put(get_log1(msg_template, arg1, arg2))
}

# ...
```

The full source code of these functions is available in the Logging Helper functions section of the capstone-movielens.main.R script on *GitHub*.

0.2.2 Preparing train and set datasets

We will split the `edx` dataset into a training set, which we will use to build and train our models, and a test set in which we will compute the accuracy of our predictions, the way described in Section 23.1.1 Movielens data of the *Course Textbook* mentioned above(Irizarry 2024a).We will also use the *5-Fold Cross Validation* method as described in Section 29.6 Cross validation of the *Course Textbook*. To prepare datasets for processing, we will use the following functions, specifically designed for these operations:

```r
make_source_datasets()
init_source_datasets()
```

The full source code of the function listed above is available in the Initialize input datasets section of the data.helper.functions.R script on *GitHub*.

0.2.2.1 The `make_source_datasets` function

Let's take a closer look at the objects we will receive as a result of executing this function.

```r
make_source_datasets <- function(){
  # ...
  list(edx_CV = edx_CV,
       edx.mx = edx.mx,
       edx.sgr = edx.sgr,
       tuning_sets = tuning_sets,
       movie_map = movie_map,
       date_days_map = date_days_map)
}
```

0.2.2.1.1 `edx.mx` Matrix Object

We will use the array representation described in Section 17.5 of the Textbook, for the training data: we denote ranking for movie j by user i as $y_{i,j}$. To create this matrix, we use `tidyr::pivot_wider` function:

```r
  put_log("Function: `make_source_datasets`: Creating Rating Matrix from `edx` dataset...")
  edx.mx <- edx |>
    mutate(userId = factor(userId),
           movieId = factor(movieId)) |>
```

```
    select(movieId, userId, rating) |>
    pivot_wider(names_from = movieId, values_from = rating) |>
    column_to_rownames("userId") |>
    as.matrix()

  put_log("Function: `make_source_datasets`:
Matrix created: `edx.mx` of the following dimentions:")
```

```
  str(edx.mx)
```

```
##  num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   ..$ : chr [1:10677] "122" "185" "292" "316" ...
```

0.2.2.1.2 `edx.sgr` Object

To account for the Movie Genre Effect more accurately, we need a dataset with split rows for movies belonging to multiple genres:

```
  put_log("Function: `make_source_datasets`:
To account for the Movie Genre Effect, we need a dataset with split rows
for movies belonging to multiple genres.")
  edx.sgr <- splitGenreRows(edx)
```

```
str(edx.sgr)
```

```
## tibble [23,371,423 x 6] (S3: tbl_df/tbl/data.frame)
##  $ userId   : int [1:23371423] 1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : int [1:23371423] 122 122 185 185 185 292 292 292 292 316 ...
##  $ rating   : num [1:23371423] 5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int [1:23371423] 838985046 838985046 838983525 838983525 838983525 838983421 838983421
##  $ title    : chr [1:23371423] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, The (19
##  $ genres   : chr [1:23371423] "Comedy" "Romance" "Action" "Crime" ...
```

```
summary(edx.sgr)
```

```
##      userId         movieId          rating        timestamp             title                 genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:23371423    Length:23371
##  1st Qu.:18140   1st Qu.:  616   1st Qu.:3.000   1st Qu.:9.472e+08   Class :character   Class :chara
##  Median :35784   Median : 1748   Median :4.000   Median :1.042e+09   Mode  :character   Mode  :chara
##  Mean   :35886   Mean   : 4277   Mean   :3.527   Mean   :1.035e+09
##  3rd Qu.:53638   3rd Qu.: 3635   3rd Qu.:4.000   3rd Qu.:1.131e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
```

Note that we use the `splitGenreRows` function to split rows of the original dataset:

```
splitGenreRows <- function(data){
  put("Splitting dataset rows related to multiple genres...")
  start <- put_start_date()
  gs_splitted <- data |>
    separate_rows(genres, sep = "\\|")
  put("Dataset rows related to multiple genres have been splitted to have single genre per row.")
  put_end_date(start)
  gs_splitted
}
```

> The source code of the function mentioned above is also available in the Initialize input datasets section of the data.helper.functions.R script on *GitHub*.

0.2.2.1.3 `movie_map` Object

To be able to map movie IDs to titles we create the following lookup table:

```
movie_map <- edx |> select(movieId, title, genres) |>
    distinct(movieId, .keep_all = TRUE)

  put_log("Function: `make_source_datasets`: Dataset created: movie_map")
```

```
str(movie_map)
```

```
## 'data.frame':    10677 obs. of  3 variables:
##  $ movieId: int  122 185 292 316 329 355 356 362 364 370 ...
##  $ title  : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|Adv
```

```
summary(movie_map)
```

```
##     movieId         title              genres
##  Min.   :    1   Length:10677       Length:10677
##  1st Qu.: 2754   Class :character   Class :character
##  Median : 5434   Mode  :character   Mode  :character
##  Mean   :13105
##  3rd Qu.: 8710
##  Max.   :65133
```

Note that titles cannot be considered unique, so we can't use them as IDs(Irizarry 2024a).

0.2.2.1.4 `date_days_map` Object

We have a `timestamp` field in the `edx` dataset. To be able to map the date, year, and number of days since the earliest record in the `edx` dataset with the corresponding value in this field, we create the following lookup table:

```r
put_log("Function: `make_source_datasets`: Creating Date-Days Map dataset...")
date_days_map <- edx |>
  mutate(date_time = as_datetime(timestamp)) |>
  mutate(date = as_date(date_time)) |>
  mutate(year = year(date_time)) |>
  mutate(days = as.integer(date - min(date))) |>
  select(timestamp, date_time, date, year, days) |>
  distinct(timestamp, .keep_all = TRUE)

put_log("Function: `make_source_datasets`: Dataset created: date_days_map")
```

```r
str(date_days_map)
```

```
## 'data.frame':    6519590 obs. of  5 variables:
##  $ timestamp: int  838985046 838983525 838983421 838983392 838984474 838983653 838984885 838983707 83
##  $ date_time: POSIXct, format: "1996-08-02 11:24:06" "1996-08-02 10:58:45" "1996-08-02 10:57:01" "199
##  $ date     : Date, format: "1996-08-02" "1996-08-02" "1996-08-02" "1996-08-02" ...
##  $ year     : num  1996 1996 1996 1996 1996 ...
##  $ days     : int  571 571 571 571 571 571 571 571 571 571 ...
```

```r
summary(date_days_map)
```

```
##    timestamp           date_time                                date                 year           days
##  Min.   :7.897e+08   Min.   :1995-01-09 11:46:49.00   Min.   :1995-01-09   Min.   :1995   Min.   :
##  1st Qu.:9.783e+08   1st Qu.:2001-01-01 05:05:01.75   1st Qu.:2001-01-01   1st Qu.:2001   1st Qu.:218
##  Median :1.091e+09   Median :2004-08-03 01:08:18.50   Median :2004-08-03   Median :2004   Median :349
##  Mean   :1.066e+09   Mean   :2003-10-10 23:15:02.07   Mean   :2003-10-10   Mean   :2003   Mean   :319
##  3rd Qu.:1.152e+09   3rd Qu.:2006-07-04 20:41:57.50   3rd Qu.:2006-07-04   3rd Qu.:2006   3rd Qu.:419
##  Max.   :1.231e+09   Max.   :2009-01-05 05:02:16.00   Max.   :2009-01-05   Max.   :2009   Max.   :51
```

0.2.2.1.5 `edx_CV` Object

Here we have a list of sample objects we need to perform the *5-Fold Cross Validation* as explained in Section 29.6.1 K-fold cross validation of the *Course Textbook*:

```r
start <- put_start_date()
edx_CV <- lapply(kfold_index,  function(fold_i){

  put_log1("Method `make_source_datasets`:
Creating K-Fold Cross Validation Datasets, Fold %1", fold_i)

  #> We split the initial datasets into training sets, which we will use to build
  #> and train our models, and validation sets in which we will compute the accuracy
  #> of our predictions, the way described in the `Section 23.1.1 Movielens data`
  #> (https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movielens-data)
  #> of the Course Textbook.

  split_sets <- edx |>
    sample_train_validation_sets(fold_i*1000)
```

```
    train_set <- split_sets$train_set
    validation_set <- split_sets$validation_set

    put_log("Function: `make_source_datasets`:
Sampling 20% from the split-row version of the `edx` dataset...")
    split_sets.gs <- edx.sgr |>
      sample_train_validation_sets(fold_i*2000)

    train.sgr <- split_sets.gs$train_set
    validation.sgr <- split_sets.gs$validation_set

    # put_log("Function: `make_source_datasets`: Dataset created: validation.sgr")
    # put(summary(validation.sgr))

    #> We will use the array representation described in `Section 17.5 of the Textbook`
    #> (https://rafalab.dfci.harvard.edu/dsbook-part-2/linear-models/treatment-effect-models.html#sec-a
    #> for the training data.
    #> To create this matrix, we use `tidyr::pivot_wider` function:

    put_log("Function: `make_source_datasets`: Creating Rating Matrix from Train Set...")
    train_mx <- train_set |>
      mutate(userId = factor(userId),
             movieId = factor(movieId)) |>
      select(movieId, userId, rating) |>
      pivot_wider(names_from = movieId, values_from = rating) |>
      column_to_rownames("userId") |>
      as.matrix()

    put_log("Function: `make_source_datasets`:
Matrix created: `train_mx` of the following dimentions:")
    put(dim(train_mx))

    list(train_set = train_set,
         train_mx = train_mx,
         train.sgr = train.sgr,
         validation_set = validation_set)
  })
  put_end_date(start)
  put_log("Function: `make_source_datasets`:
Set of K-Fold Cross Validation datasets created: edx_CV")
```

```
str(edx_CV)
```

```
## List of 5
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172311 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172311] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172311] 122 185 292 329 356 362 364 370 420 466 ...
##   .. ..$ rating   : num [1:7172311] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172311] 838985046 838983525 838983421 838983392 838983653 838984885 8389
##   .. ..$ title    : chr [1:7172311] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Star Tre
##   .. ..$ genres   : chr [1:7172311] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Th
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
```

```
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "329" ...
##   ..$ train.sgr     : tibble [18,669,190 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669190] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669190] 122 122 185 185 292 292 292 292 316 316 ...
##   .. ..$ rating   : num [1:18669190] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669190] 838985046 838985046 838983525 838983525 838983421 838983421 838
##   .. ..$ title    : chr [1:18669190] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, T
##   .. ..$ genres   : chr [1:18669190] "Comedy" "Romance" "Action" "Crime" ...
##   ..$ validation_set:'data.frame':   1827744 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827744] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827744] 316 355 377 588 260 376 648 1049 110 1252 ...
##   .. ..$ rating   : num [1:1827744] 5 5 5 5 5 3 2 3 4.5 4 ...
##   .. ..$ timestamp: int [1:1827744] 838983392 838984474 838983834 838983339 868244562 868245920 8682
##   .. ..$ title    : chr [1:1827744] "Stargate (1994)" "Flintstones, The (1994)" "Speed (1994)" "Alad
##   .. ..$ genres   : chr [1:1827744] "Action|Adventure|Sci-Fi" "Children|Comedy|Fantasy" "Action|Roma
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172306 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172306] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172306] 122 185 292 316 329 355 356 364 370 377 ...
##   .. ..$ rating   : num [1:7172306] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172306] 838985046 838983525 838983421 838983392 838983392 838984474 8389
##   .. ..$ title    : chr [1:7172306] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172306] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Th
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,201 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669201] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669201] 122 122 185 185 185 292 292 316 316 329 ...
##   .. ..$ rating   : num [1:18669201] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669201] 838985046 838985046 838983525 838983525 838983525 838983421 838
##   .. ..$ title    : chr [1:18669201] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, T
##   .. ..$ genres   : chr [1:18669201] "Comedy" "Romance" "Action" "Crime" ...
##   ..$ validation_set:'data.frame':   1827749 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827749] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827749] 362 520 539 594 539 590 733 1210 1252 1408 ...
##   .. ..$ rating   : num [1:1827749] 5 5 5 5 3 5 3 4 4 3.5 ...
##   .. ..$ timestamp: int [1:1827749] 838984885 838984679 838984068 838984679 868246262 868245608 8682
##   .. ..$ title    : chr [1:1827749] "Jungle Book, The (1994)" "Robin Hood: Men in Tights (1993)" "Sl
##   .. ..$ genres   : chr [1:1827749] "Adventure|Children|Romance" "Comedy" "Comedy|Drama|Romance" "An
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172307 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172307] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172307] 122 185 292 316 329 355 362 370 377 420 ...
##   .. ..$ rating   : num [1:7172307] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172307] 838985046 838983525 838983421 838983392 838983392 838984474 8389
##   .. ..$ title    : chr [1:7172307] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172307] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Th
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
```

```
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,195 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669195] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669195] 122 122 185 185 185 292 292 292 316 329 ...
##   .. ..$ rating   : num [1:18669195] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669195] 838985046 838985046 838983525 838983525 838983525 838983421 8389
##   .. ..$ title    : chr [1:18669195] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669195] "Comedy" "Romance" "Action" "Crime" ...
##   ..$ validation_set:'data.frame':   1827748 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827748] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827748] 356 364 539 616 590 719 780 786 151 213 ...
##   .. ..$ rating   : num [1:1827748] 5 5 5 5 5 3 3 3 4.5 5 ...
##   .. ..$ timestamp: int [1:1827748] 838983653 838983707 838984068 838984941 868245608 868246191 8682
##   .. ..$ title    : chr [1:1827748] "Forrest Gump (1994)" "Lion King, The (1994)" "Sleepless in Seat
##   .. ..$ genres   : chr [1:1827748] "Comedy|Drama|Romance|War" "Adventure|Animation|Children|Drama|Mu
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172311 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172311] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172311] 122 185 292 316 329 355 356 362 364 370 ...
##   .. ..$ rating   : num [1:7172311] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172311] 838985046 838983525 838983421 838983392 838983392 838984474 83898
##   .. ..$ title    : chr [1:7172311] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172311] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,192 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669192] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669192] 122 122 185 185 292 292 316 316 329 329 ...
##   .. ..$ rating   : num [1:18669192] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669192] 838985046 838985046 838983525 838983525 838983421 838983421 8389
##   .. ..$ title    : chr [1:18669192] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669192] "Comedy" "Romance" "Action" "Thriller" ...
##   ..$ validation_set:'data.frame':   1827744 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827744] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827744] 377 520 588 616 110 648 1049 1356 1148 1276 ...
##   .. ..$ rating   : num [1:1827744] 5 5 5 5 5 2 3 3 4 3.5 ...
##   .. ..$ timestamp: int [1:1827744] 838983834 838984679 838983339 838984941 868245777 868244699 8682
##   .. ..$ title    : chr [1:1827744] "Speed (1994)" "Robin Hood: Men in Tights (1993)" "Aladdin (1992
##   .. ..$ genres   : chr [1:1827744] "Action|Romance|Thriller" "Comedy" "Adventure|Animation|Children
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172301 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172301] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172301] 122 185 292 316 355 356 364 370 420 466 ...
##   .. ..$ rating   : num [1:7172301] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172301] 838985046 838983525 838983421 838983392 838984474 838983653 83898
##   .. ..$ title    : chr [1:7172301] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172301] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,194 x 6] (S3: tbl_df/tbl/data.frame)
```

```
##   .. ..$ userId   : int [1:18669194] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669194] 122 122 185 185 292 292 316 329 329 355 ...
##   .. ..$ rating   : num [1:18669194] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669194] 838985046 838985046 838983525 838983525 838983421 838983421 838
##   .. ..$ title    : chr [1:18669194] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, T
##   .. ..$ genres   : chr [1:18669194] "Comedy" "Romance" "Crime" "Thriller" ...
##   ..$ validation_set:'data.frame':   1827754 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827754] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827754] 329 362 377 594 110 376 539 736 1252 1408 ...
##   .. ..$ rating   : num [1:1827754] 5 5 5 5 5 3 3 3 4 3.5 ...
##   .. ..$ timestamp: int [1:1827754] 838983392 838984885 838983834 838984679 868245777 868245920 8682
##   .. ..$ title    : chr [1:1827754] "Star Trek: Generations (1994)" "Jungle Book, The (1994)" "Speed
##   .. ..$ genres   : chr [1:1827754] "Action|Adventure|Drama|Sci-Fi" "Adventure|Children|Romance" "Ac
```

This code snippet is a part of the make_source_datasets *function* code described above.

Note that we used the `sample_train_validation_sets` function call to split the original dataset (`edx` in this case):

```
    split_sets <- edx |>
      sample_train_validation_sets(fold_i*1000)
```

which returns a pair of train/validation sets:

```
sample_train_validation_sets <- function(data, seed){
  put_log("Function: `sample_train_validation_sets`: Sampling 20% of the `data` data...")
  set.seed(seed)
  validation_ind <-
    sapply(splitByUser(data),
           function(i) sample(i, ceiling(length(i)*.2))) |>
    unlist() |>
    sort()

  put_log("Function: `sample_train_validation_sets`:
Extracting 80% of the original `data` not used for the Validation Set,
excluding data for users who provided no more than a specified number of ratings: {min_nratings}.")

  train_set <- data[-validation_ind,]

  put_log("Function: `sample_train_validation_sets`: Dataset created: train_set")
  put(summary(train_set))

  put_log("Function: `sample_train_validation_sets`:
To make sure we don't include movies in the Training Set that should not be there,
we exclude entries using the semi_join function from the Validation Set.")
  tmp.data <- data[validation_ind,]

  validation_set <- tmp.data |>
    semi_join(train_set, by = "movieId") |>
    semi_join(train_set, by = "userId") |>
```

```
    as.data.frame()

  # Add rows excluded from `validation_set` into `train_set`
  tmp.excluded <- anti_join(tmp.data, validation_set)
  train_set <- rbind(train_set, tmp.excluded)

  put_log("Function: `sample_train_validation_sets`: Dataset created: validation_set")
  put(summary(validation_set))

  # CV train & test sets Consistency Test
  validation.left_join.Nas <- train_set |>
    mutate(tst.col = rating) |>
    select(userId, movieId, tst.col) |>
    data.consistency.test(validation_set)

  put_log("Function: `sample_train_validation_sets`:
Below are the data consistency verification results")
  put(validation.left_join.Nas)

  # Return result datassets ----------------------------------------------------
  list(train_set = train_set,
       validation_set = validation_set)
}
```

The sample_train_validation_sets function is defined in the same script as the make_source_datasets‘ one, from where it is called.

0.2.2.2 Common Helper Functions

For our further analysis, we are going to use the following *common helper functions*:

0.2.2.2.1 `clamp` function

As explained in Section 24.4 User effects of the *Course Textbook* we know ratings can't be below 0.5 or above 5. For this reason, we will use the `clamp` function described in that section:

```
clamp <- function(x, min = 0.5, max = 5) pmax(pmin(x, max), min)
```

0.2.2.2.2 Functions to calculate *(Root) Mean Squared Error*

We will need the following functions to calculate *(R)MSEs*:

```
mse <- function(r) mean(r^2)

mse_cv <- function(r_list) {
  mses <- sapply(r_list, mse(r))
  mean(mses)
}
```

```r
rmse <- function(r) sqrt(mse(r))
# rmse_cv <- function(r_list) sqrt(mse_cv(r_list))

rmse2 <- function(true_ratings, predicted_ratings) {
  rmse(true_ratings - predicted_ratings)
}
```

All the *common helper functions*, including those described above, are defined in the common-helper.functions.R script on *GitHub*.

0.2.3 Overall Mean Rating (Naive) Model

Let's begin our analysis by evaluating the simplest model described in Section *23.3 The First Model* of the *Course Textbook*, and then gradually refine it through further research. It is about a model that assumes the same rating for all movies and users with all the differences explained by random variation would look as follows:

$$Y_{i,j} = \mu + \varepsilon_{i,j}$$

with $\varepsilon_{i,j}$ independent errors sampled from the same distribution centered at 0 and μ the *true* rating for all movies.

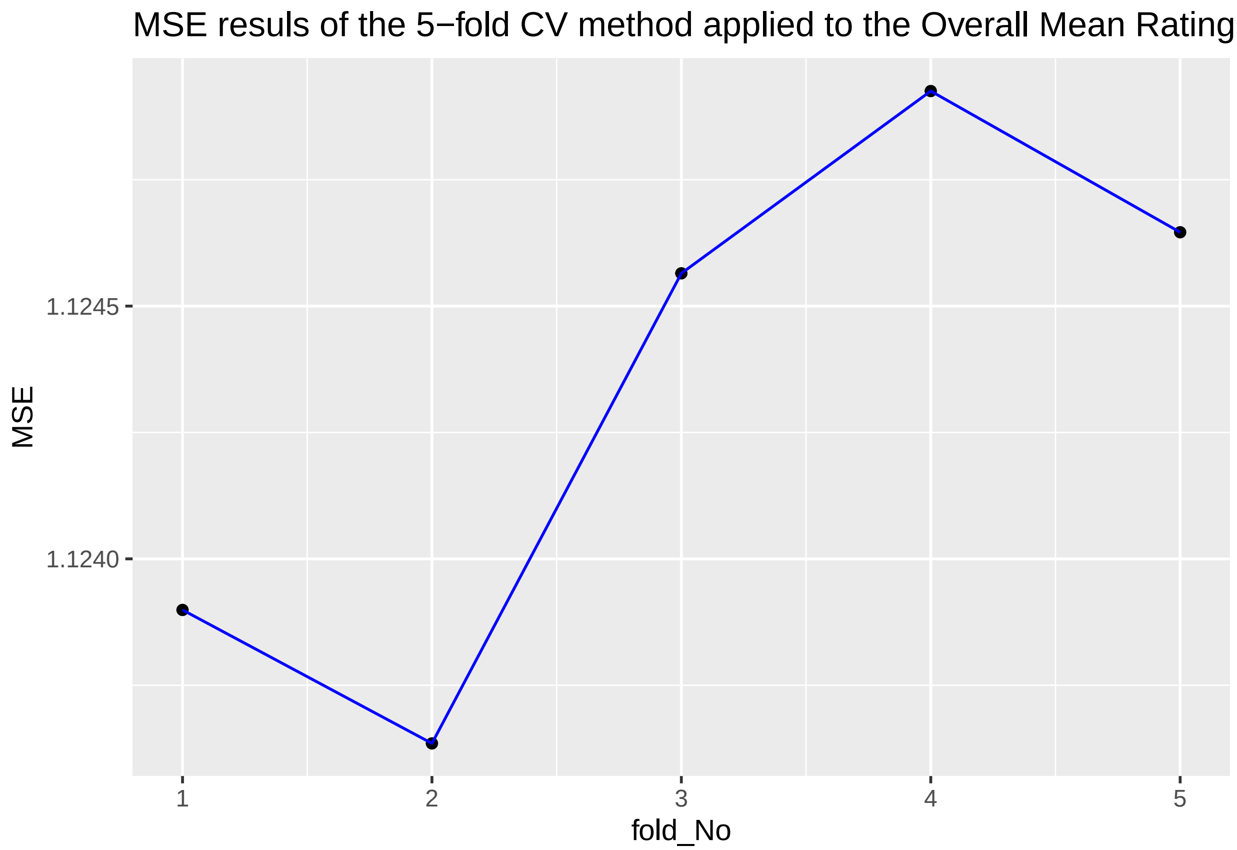
We know that the estimate that minimizes the RMSE is the least squares estimate of μ and, in this case, is the average of all ratings:

```r
mu <- mean(edx$rating)
print(mu)
```

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

If we predict all unknown ratings with $\hat{\mu}$, we obtain the following RMSE:

```r
mu.MSEs <- naive_model_MSEs(mu)
data.frame(fold_No = 1:5, MSE = mu.MSEs) |>
  data.plot(title = "MSE resuls of the 5-fold CV method applied to the Overall Mean Rating Model",
            xname = "fold_No",
            yname = "MSE")
```




```
mu.RMSE <- sqrt(mean(mu.MSEs))
mu.RMSE
```

```
## [1] 1.060346
```

For the *Mean Squared Error* data visualization we used data.plot function] defined in the Data Visualization section of the data.helper.function.R script.

```
data.plot <- function(data,
                      title,
                      xname,
                      yname,
                      xlabel = NULL,
                      ylabel = NULL,
                      line_col = "blue",
                      # scale = 1,
                      normalize = FALSE) {
  y <- data[, yname]

  if (normalize) {
    y <- y - min(y)
  }

  if (is.null(xlabel)) {
```

```r
    xlabel = xname
  }
  if (is.null(ylabel)) {
    ylabel = yname
  }

  aes_mapping <- aes(x = data[, xname], y = y)

  data |>
    ggplot(mapping = aes_mapping) +
    ggtitle(title) +
    xlab(xlabel) +
    ylab(ylabel) +
    geom_point() +
    geom_line(color=line_col)
}
```

Here we also used naive_model_MSEs function defined in the common-helper.functions.R script (already mentioned above) to compute *Mean Squared Errors* using *5-Fold Cross Validation* method:

```r
naive_model_MSEs <- function(val) {
  sapply(edx_CV, function(cv_item){
    mse(cv_item$validation_set$rating - val)
  })
}
```

One more function, defined in the same script, that we will need for further analysis of the current model, is the naive_model_RMSE one:

```r
naive_model_RMSE <- function(val){
  sqrt(mean(naive_model_MSEs(val)))
}
```

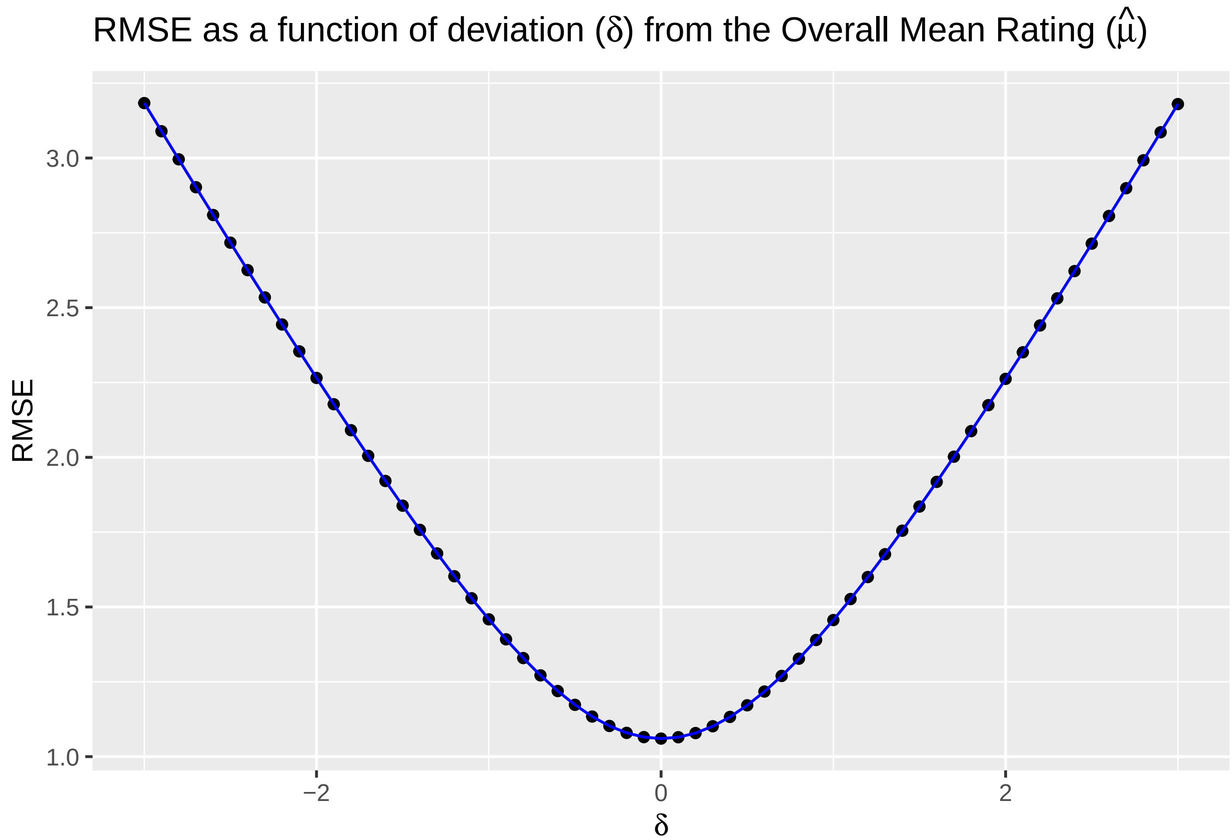
0.2.3.1 Ensure that `mu.RMSE` value is the best for the current model

If we plug in any other number, we will get a higher RMSE. Let's prove that by the following small investigation:

```r
  deviation <- seq(0, 6, 0.1) - 3

  deviation.RMSE <- sapply(deviation, function(delta){
    naive_model_RMSE(mu + delta)
  })
```

Let's make a quick investigation of the `deviation.RMSE` result we have just got:

```r
data.frame(delta = deviation,
           delta.RMSE = deviation.RMSE) |>
data.plot(title = TeX(r'[RMSE as a function of deviation ($\delta$) from the Overall Mean Rating ($\hat
              xname = "delta",
              yname = "delta.RMSE",
              xlabel = TeX(r'[$\delta$]'),
              ylabel = "RMSE")
```




```
which_min_deviation <- deviation[which.min(deviation.RMSE)]
min_rmse = min(deviation.RMSE)

print_log1("Minimum RMSE is achieved when the deviation from the mean is: %1",
         which_min_deviation)
```

```
## Minimum RMSE is achieved when the deviation from the mean is: 0
```

```
print_log1("Is the previously computed RMSE the best for the current model: %1",
         mu.RMSE == min_rmse)
```

```
## Is the previously computed RMSE the best for the current model: TRUE
```

```
RMSEs.ResultTibble.OMR <- RMSEs.ResultTibble |>
  RMSEs.AddRow("Overall Mean Rating Model", mu.RMSE)
```

```
RMSE_kable(RMSEs.ResultTibble.OMR)
```

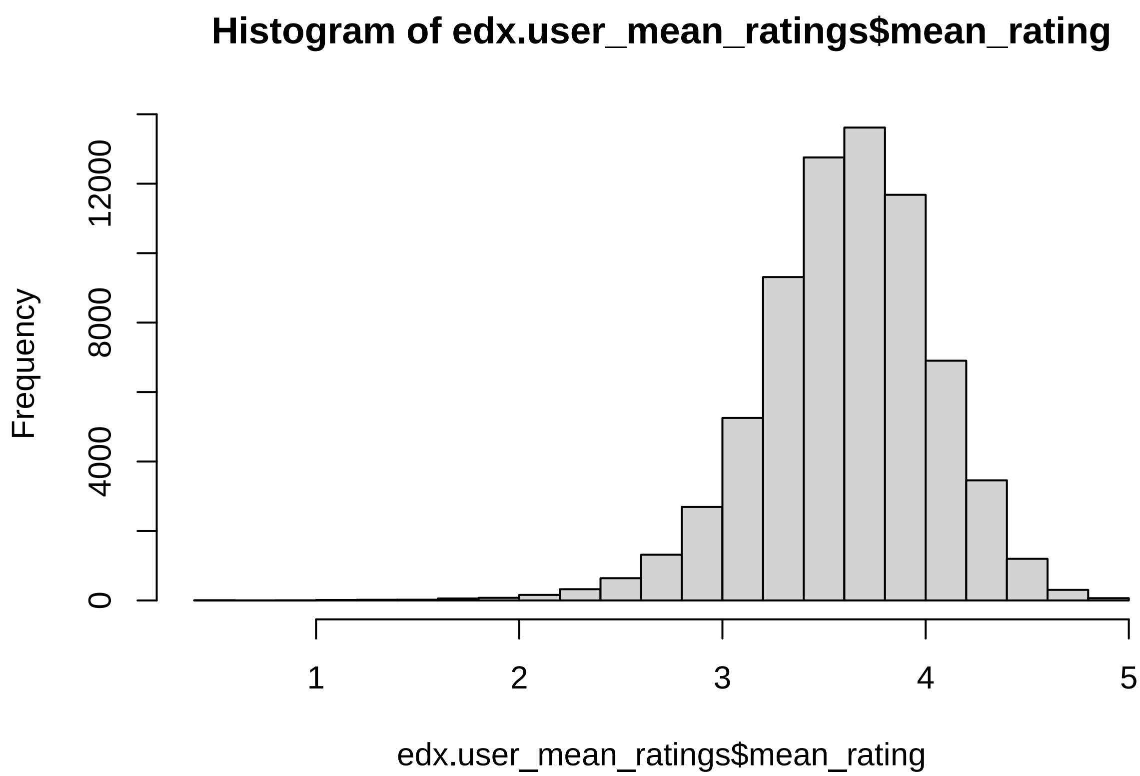
Method	RMSE	Comment
Project Objective	0.864900	
Overall Mean Rating Model	1.060346	

To win the grand prize of $1,000,000, a participating team had to get an RMSE of at least 0.8563(Andreas Toscher 2009). So we can definitely do better!(Irizarry 2024c)

0.2.4 User Effect Model

To improve our model let's now take into consideration user effects as explained in Section *23.4 User effects* of the *Course Textbook*. If we visualize the average rating for each user the way the author shows, we can see that there is substantial variability in the average ratings across users:

```
hist(edx.user_mean_ratings$mean_rating, nclass = 30)
```



Following the author's further explanation, to account for this variability, we will use a linear model with a *treatment effect* α_i for each user. The sum $\mu + \alpha_i$ can be interpreted as the typical rating user i gives to movies. So we write the model as follows:

$$Y_{i,j} = \mu + \alpha_i + \varepsilon_{i,j}$$

Statistics textbooks refer to the αs as treatment effects. In the Netflix challenge papers, they refer to them as *bias*(Irizarry 2024d; Yehuda Koren and Volinsky 2009).

As it is stated here(Irizarry 2024d), it can be shown that the least squares estimate $\hat{\alpha}_i$ is just the average of $y_{i,j} - \hat{\mu}$ for each user i. So we can compute them this way:

```
a <- rowMeans(y - mu, na.rm = TRUE)
```

These considerations alows us to compute a *User Mean Ratings* the following way:

```
put_log("Computing Average Ratings per User (User Mean Ratings)...")
user.mean_ratings <- rowMeans(edx.mx, na.rm = TRUE)
user_ratings.n <- rowSums(!is.na(edx.mx))
```

```r
edx.user_mean_ratings <-
  data.frame(userId = names(user.mean_ratings),
             mean_rating = user.mean_ratings,
             n = user_ratings.n)

put_log("User Mean Ratings have been computed.")
```

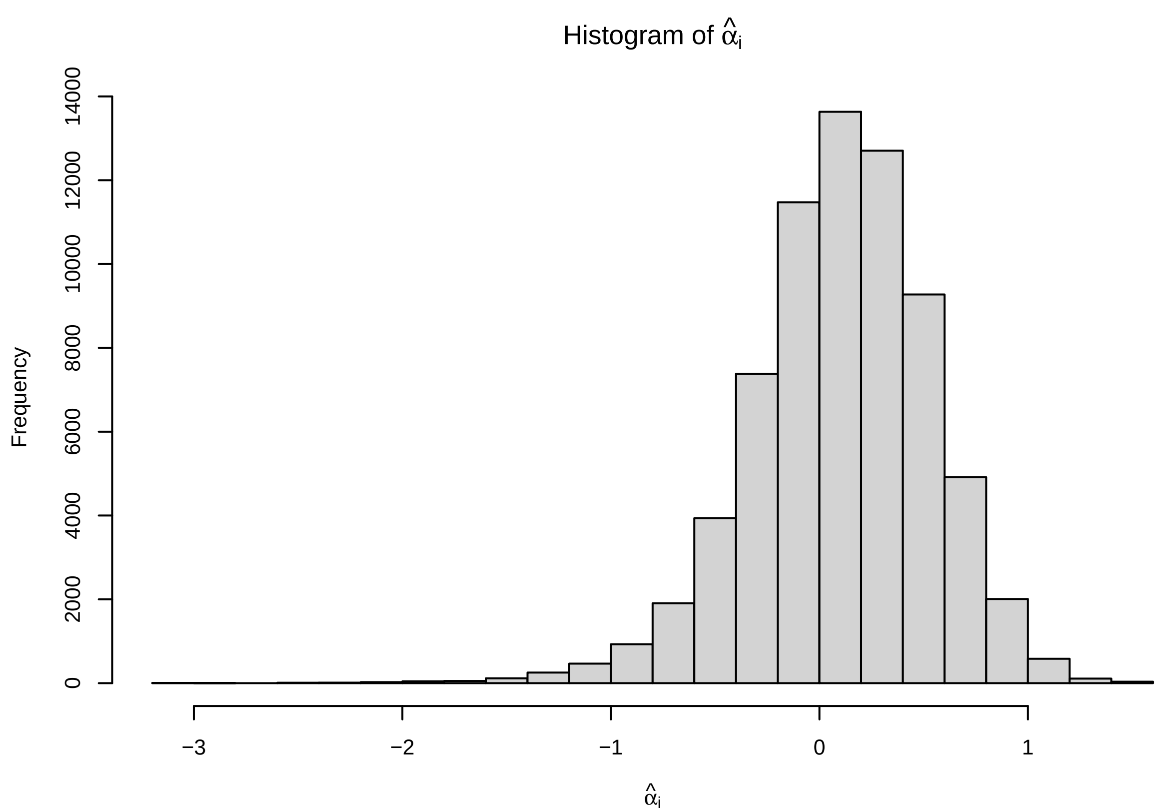
```r
str(edx.user_mean_ratings)
```

```
## 'data.frame':    69878 obs. of  3 variables:
##  $ userId     : chr  "1" "2" "3" "4" ...
##  $ mean_rating: num  5 3.29 3.94 4.06 3.92 ...
##  $ n          : num  19 17 31 35 74 39 96 727 21 112 ...
```

And then we compute a *User Effect* this way:

```r
put_log("Computing User Effect per users ...")
edx.user_effect <- edx.user_mean_ratings |>
  mutate(userId = as.integer(userId),
         a = mean_rating - mu)

put_log("A User Effect Model has been builded")
```

```r
par(cex = 0.7)
hist(edx.user_effect$a, 30, xlab = TeX(r'[$\hat{alpha}_{i}$]'),
     main = TeX(r'[Histogram of $\hat{alpha}_{i}$]'))
```



```r
str(edx.user_effect)
```

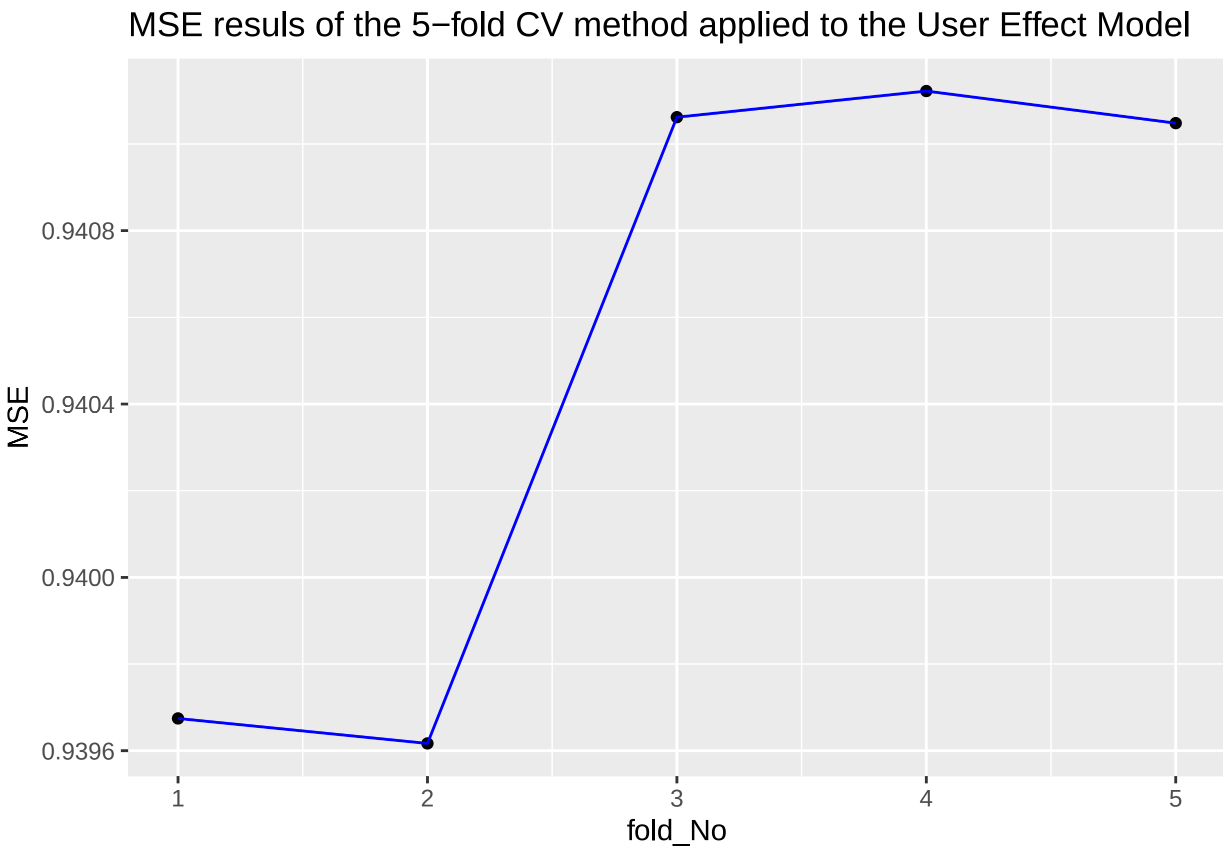
```
## 'data.frame':    69878 obs. of  4 variables:
##  $ userId     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ mean_rating: num  5 3.29 3.94 4.06 3.92 ...
##  $ n          : num  19 17 31 35 74 39 96 727 21 112 ...
##  $ a          : num  1.488 -0.218 0.423 0.545 0.406 ...
```

> The full source code of the *User Effect* computation is available in the Model building: User Effect section of the capstone-movielens.main.R script on *GitHub*.

Finally, we are ready to compute the `RMSE` (additionally using the clamp helper function we defined above to keep predictions in the proper range):

```r
put_log("Computing the RMSE taking into account user effects...")
start <- put_start_date()
edx.user_effect.MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |>
    left_join(edx.user_effect, by = "userId") |>
    mutate(resid = rating - clamp(mu + a)) |>
    pull(resid) |> mse()
})
put_end_date(start)

edx.user_effect.RMSE <- sqrt(mean(edx.user_effect.MSEs))

RMSEs.ResultTibble.UE <- RMSEs.ResultTibble.OMR |>
  RMSEs.AddRow("User Effect Model", edx.user_effect.RMSE)
```

```r
data.frame(fold_No = 1:5, MSE = edx.user_effect.MSEs) |>
  data.plot(title = "MSE resuls of the 5-fold CV method applied to the User Effect Model",
            xname = "fold_No",
            yname = "MSE")
```




```r
RMSE_kable(RMSEs.ResultTibble.UE)
```

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	

The full source code of the *User Effect Model RMSE* computation is available in the Compute RMSE for User Effect Model section of the capstone-movielens.main.R script on *GitHub*.

0.2.5 User+Movie Effect (UME) Model

In 23.5 Movie effects section of the *Course Textbook* the author draws our attention to the fact that some movies are generally rated higher than others. He also explains that a linear model with a *treatment effect* β_j for each movie can be used in this case, which can be interpreted as movie effect or the difference between the average ranking for movie j and the overall average μ:

$$Y_{i,j} = \mu + \alpha_i + \beta_j + \varepsilon_{i,j}$$

The author then shows how to use an approximation by first computing the least square estimate $\hat{\mu}$ and $\hat{\alpha}_i$, and then estimating $\hat{\beta}_j$ as the average of the residuals $y_{i,j} - \hat{\mu} - \hat{\alpha}_i$:

```r
b <- colMeans(y - mu - a, na.rm = TRUE)
```

Inspired by this idea, a few support functions were developed by the author of this report, which we will use for our further analysis.

0.2.5.1 UME Model: Support Functions

> The full source code of the functions described in this section is available in the User+Movie Effect Model Functions section of the UM-effect.functions.R script on *GitHub*.

0.2.5.1.1 `train_user_movie_effect` Function

We use this function to build and train our model using the `train_set` dataset:

```r
train_user_movie_effect <- function(train_set, lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_user_movie_effect
`lambda` is `NA`")
  }

  UM.effect <- train_set |>
    left_join(edx.user_effect, by = "userId") |>
    mutate(resid = rating - (mu + a)) |>
    group_by(movieId) |>
    summarise(b = mean_reg(resid, lambda), n = n())

  stopifnot(!is.na(mean(UM.effect$b)))
  UM.effect
}
```

> The function described above accepts the `lambda` parameter, which we will need later for the *Regularization* method. We also use the mean_reg function call, which we will also need for the *Regularization*. We will explain that later in the [Regularization Method] section. For now, we omit the `lambda` parameter, accepting its default value `lambda = 0`. In this case, the `mean_reg` function is equivalent to the simple `mean` one.

```r
## Regularization ------------------------------------------------------------
mean_reg <- function(vals, lambda = 0, na.rm = TRUE){
  if (is.na(lambda)) {
    stop("Function: mean_reg
`lambda` is `NA`")
  }

  names(lambda) <- NULL
  sums <- sum(vals, na.rm = na.rm)
  N <- ifelse(na.rm, sum(!is.na(vals)), length(vals))
  sums/(N + lambda)
}
```

0.2.5.1.2 `train_user_movie_effect.cv` Function

We use the train_user_movie_effect.cv function to build and train our model using the `5-Fold Cross Validation` method. Below, we provide the most important part of the code of that function:

```
train_user_movie_effect.cv <- function(lambda = 0){
# ...
  start <- put_start_date()
  user_movie_effects_ls <- lapply(edx_CV, function(cv_fold_dat){
    cv_fold_dat$train_set |> train_user_movie_effect(lambda)
  })
  put_end_date(start)
  put_log("Function: train_user_movie_effect.cv:
User+Movie Effect list have been computed")

  user_movie_effects_united <- union_cv_results(user_movie_effects_ls)

  user_movie_effect <- user_movie_effects_united |>
    group_by(movieId) |>
    summarise(b = mean(b), n = mean(n))
  # ...
  user_movie_effect
}
```

> Here we use the function call union_cv_results, which is defined in the script common-helper.functions.R, to aggregate the `Cross Validation` results.

```
union_cv_results <- function(data_list) {
  out_dat <- data_list[[1]]

  for (i in 2:CVFolds_N){
    out_dat <- union(out_dat,
                     data_list[[i]])
  }

  out_dat
}
```

0.2.5.1.3 `calc_user_movie_effect_MSE` Function

The code of the function calc_user_movie_effect_MSE defined in the UM-effect.functions.R script to calculate the `Mean Squared Error (MSE)` of the `UME Model` for the given `Test Set` is provided below:

```
calc_user_movie_effect_MSE <- function(test_set, um_effect){
  mse.result <- test_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(um_effect, by = "movieId") |>
    mutate(resid = rating - clamp(mu + a + b)) |>
    pull(resid) |> mse()
```

```
  stopifnot(!is.na(mse.result))
  mse.result
}
```

0.2.5.1.4 `calc_user_movie_effect_MSE.cv` Function

The code of the function calc_user_movie_effect_MSE.cv defined in the UM-effect.functions.R script to calculate the `5-Fold Cross Validation` MSE result of the `UME Model` is provided below:

```
calc_user_movie_effect_MSE.cv <- function(um_effect){
  put_log("Function: user_movie_effects_MSE.cv:
Computing the RMSE taking into account User+Movie Effects...")
  start <- put_start_date()
  user_movie_effects_MSEs <- sapply(edx_CV, function(cv_fold_dat){
    cv_fold_dat$validation_set |> calc_user_movie_effect_MSE(um_effect)
  })
  put_end_date(start)

  put_log1("Function: user_movie_effects_MSE.cv:
MSE values have been plotted for the %1-Fold Cross Validation samples.",
           CVFolds_N)

  mean(user_movie_effects_MSEs)
}
```

0.2.5.1.5 `calc_user_movie_effect_RMSE` Function

The code of the function calc_user_movie_effect_RMSE defined in the UM-effect.functions.R script to calculate the `Root Mean Squared Error (RMSE)` of the `UME Model` for the given `Test Set` is provided below:

```
calc_user_movie_effect_RMSE <- function(test_set, um_effect){
  mse <- test_set |> calc_user_movie_effect_MSE(um_effect)
  sqrt(mse)
}
```

0.2.5.1.6 `calc_user_movie_effect_RMSE.cv` Function

The code of the function calc_user_movie_effect_RMSE.cv defined in the UM-effect.functions.R script to calculate the `5-Fold Cross Validation` RMSE result of the `UME Model` is provided below:

```
calc_user_movie_effect_RMSE.cv <- function(um_effect){
  user_movie_effects_MSE <- calc_user_movie_effect_MSE.cv(um_effect)
  um_effect_RMSE <- sqrt(user_movie_effects_MSE)
  put_log2("Function: user_movie_effects_RMSE.cv:
%1-Fold Cross Validation ultimate RMSE: %2", CVFolds_N, um_effect_RMSE)
  um_effect_RMSE
}
```

0.2.5.2 Model Building

> The full source code of builing and training the current model is available in the Model building: User+Movie Effect section of the capstone-movielens.main.R script on *GitHub*.

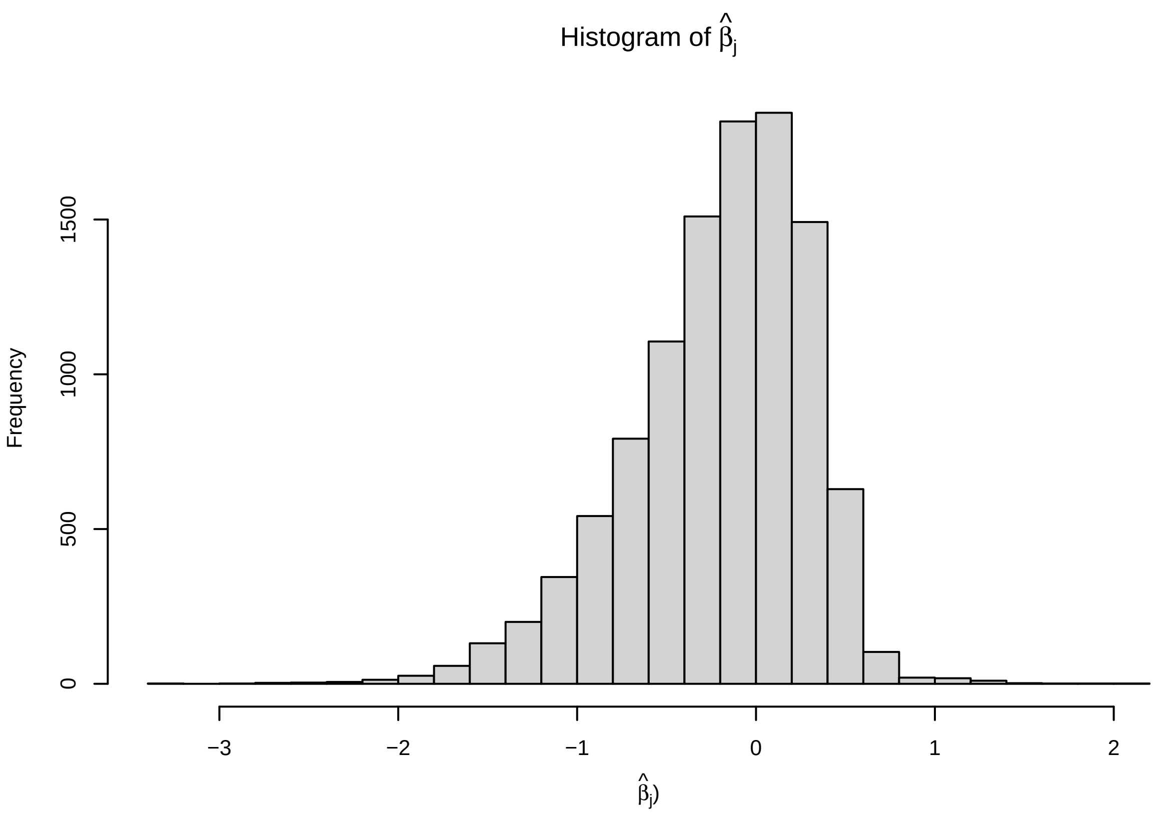
Below, we provide the most significant part of the code for training our model using the `5-Fold Cross Validation` method:

```
cv.UM_effect <- train_user_movie_effect.cv()
```

```
str(cv.UM_effect)
```

```
## tibble [10,677 x 3] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ b      : num [1:10677] 0.335 -0.306 -0.365 -0.598 -0.444 ...
##  $ n      : num [1:10677] 18907 8593 5574 1253 5065 ...
```

```
par(cex = 0.7)
hist(cv.UM_effect$b, 30, xlab = TeX(r'[$\hat{beta}_{j}$)]'),
     main = TeX(r'[Histogram of $\hat{beta}_{j}$]'))
```



We can now construct predictors and see how much the `RMSE` improves(Irizarry 2024e):

```
cv.UM_effect.RMSE <- calc_user_movie_effect_RMSE.cv(cv.UM_effect)

RMSEs.ResultTibble.UME <- RMSEs.ResultTibble.UE |>
  RMSEs.AddRow("User+Movie Effect Model", cv.UM_effect.RMSE)
```

```
RMSE_kable(RMSEs.ResultTibble.UME)
```

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	

0.2.6 Regularizing User+Movie Effect Model

0.2.6.1 Utilizing Penalized least squares

Section *23.6 Penalized least squares* of the *Course Textbook* explains why and how we should use *Penalized least squares* to improve our predictions. The author also explains that the general idea of penalized regression is to control the total variability of the movie effects: $\sum_{j=1}^{n} \beta_j^2$. Specifically, instead of minimizing the least squares equation, we minimize an equation that adds a penalty:

$$\sum_{i,j} \left(y_{u,i} - \mu - \alpha_i - \beta_j\right)^2 + \lambda \sum_{j} \beta_j^2$$

The first term is just the sum of squares and the second is a penalty that gets larger when many β_is are large. Using calculus, we can actually show that the values of β_i that minimize this equation are:

$$\hat{\beta}_j(\lambda) = \frac{1}{\lambda + n_j} \sum_{i=1}^{n_i} \left(Y_{i,j} - \mu - \alpha_i\right)$$

where n_j is the number of ratings made for movie j.

This approach will have our desired effect: when our sample size n_j is very large, we obtain a stable estimate and the penalty λ is effectively ignored since $n_j + \lambda \approx n_j$. Yet when the n_j is small, then the estimate $\hat{\beta}_i(\lambda)$ is shrunken towards 0. The larger the λ, the more we shrink(Irizarry 2024f).

We will implement the *Regularization* method on our models (starting from the current model) in two steps:

1. **Preconfiguration:** Preliminary determination of the optimal range of λ values for the `5-Fold Cross Validation` samples;
2. **Fine-tuning:** figuring out the value of λ that minimizes the model's RMSE.

0.2.6.2 Regularization: Common Helper Functions

> The full source code of the functions described below are available in the Model Tuning section of the common-helper.functions.R script on *GitHub*.

0.2.6.2.1 tune.model_param Function

The function searches for the parameter value corresponding to the minimum value of the `RMSE` from the list of values specified by the `param_values` parameter.

Sugnature

```
tune.model_param <- function(param_values,
                             fn_tune.test.param_value,
                             break.if_min = TRUE,
                             steps.beyond_min = 2){

# ...
  list(tuned.result = data.frame(RMSE = RMSEs_tmp,
                                 parameter.value = param_vals_tmp),
       best_result = param_values.best_result)
}
```

Parameters

- **param_values:** A list of values to search for the value corresponding to the minimum value of the `RMSE` ;
- **fn_tune.test.param_value:** A helper function that calculates the value of the `RMSE` for a given parameter value.;
- **break.if_min = TRUE:** A Boolean parameter that determines whether the function should terminate after completing the number of steps specified by the parameter `steps.beyond_min`, after the minimum value of the `RMSE` has been found;
- **steps.beyond_min = 2:** (takes effect only if `break.if_min` parameter is `TRUE`) Specifies the number of steps after finding the minimum value of the `RMSE`, upon completion of which the function should terminate.

Details

During execution, the function uses a helper function specified by the `fn_tune.test.param_value` parameter, which calculates the `RMSE` value for the given parameter from the list determined by the `param_values` parameter.

> Note that the algorithm assumes that the dependence of the `RMSE` on the input parameter is a monotonically decreasing function until a minimum is reached and monotonically increasing thereafter. That is, it is assumed that the function has a single minimum on the given interval.

Value

The function returns a data structure containing the found value of the input parameter `param_values` for which the `RMSE` value is minimal, as well as the minimum `RMSE` value itself, along with a sequence of all calculated `RMSE` values.

Source Code

Below is the most significant part of the source code of the tune.model_param function:

```
tune.model_param <- function(param_values,
                             fn_tune.test.param_value,
                             break.if_min = TRUE,
                             steps.beyond_min = 2){
  n <- length(param_values)
  param_vals_tmp <- numeric()
  RMSEs_tmp <- numeric()
  RMSE_min <- Inf
  i_max.beyond_RMSE_min <- Inf
  prm_val.best <- NA

  # ...

  for (i in 1:n) {
    put_log1("Function: `tune.model_param`:
Iteration %1", i)
    prm_val <- param_values[i]
    param_vals_tmp[i] <- prm_val

    RMSE_tmp <- fn_tune.test.param_value(prm_val)
    RMSEs_tmp[i] <- RMSE_tmp

    plot(param_vals_tmp[RMSEs_tmp], RMSEs_tmp[RMSEs_tmp])

    if(RMSE_tmp > RMSE_min){
      warning("Function: `tune.model_param`:
`RSME` reached its minimum: ", RMSE_min, "
for parameter value: ", prm_val)
      put_log2("Function: `tune.model_param`:
Current `RMSE` value is %1 related to parameter value: %2",
               RMSE_tmp,
               prm_val)

      if (i > i_max.beyond_RMSE_min) {
        warning("Function: `tune.model_param`:
Operation is breaked (after `RSME` reached its minimum) on the following step: ", i)
        break
      }
      next
    }

    RMSE_min <- RMSE_tmp
    prm_val.best <- prm_val

    if (break.if_min) {
      i_max.beyond_RMSE_min <- i + steps.beyond_min
    }
  }
```

```r
  param_values.best_result <- c(param.best_value = prm_val.best,
                               best_RMSE = RMSE_min)


  list(tuned.result = data.frame(RMSE = RMSEs_tmp,
                               parameter.value = param_vals_tmp),
       best_result = param_values.best_result)
}
```

0.2.6.2.2 model.tune.param_range Function

The function fine-tunes the model by searching for the best possible value of the input parameter over a given interval for which the corresponding `RMSE` value is minimal.

Sugnature

```r
model.tune.param_range <- function(loop_starter,
                             tune_dir_path,
                             cache_file_base_name,
                             fn_tune.test.param_value,
                             max.identical.min_RMSE.count = 4,
                             endpoint.min_diff = 0,
                             break.if_min = TRUE,
                             steps.beyond_min = 2){
  # ...
  list(best_result = param_values.best_result,
       param_values.endpoints = c(prm_val.leftmost, prm_val.rightmost, seq_increment),
       tuned.result = data.frame(parameter.value = parameter.value,
                                RMSE = result.RMSE))
}
```

Parameters

loop_starter

A numeric vector of the form `c(start, end, dvs)`, where `start` and `end` are the endpoints of the interval on which the parameter value that minimizes `RMSE` is sought. `dvs` is a divisor for splitting the interval to transform it into a sequence of values among which the value that minimizes `RMSE` is sought. For this purpose, the sequence step is calculated as follows:

$$step = \frac{end - start}{dvs}$$

The sequence obtained as a result of the transformation is equivalent to the one generated by the function `seq` as follows:

```r
seq(start, end, step)
```

In fact, the `seq` function is called internally to generate the sequence during the execution of the `model.tune.param_range` function.

tune_dir_path

To improve performance, the algorithm caches intermediate results in the file system. This parameter specifies the path to the directory where the files are cached.

cache_file_base_name

The algorithm generates unique names for cache files based on this and the `loop_starter` parameter, as well as some other intermediate values calculated during the execution.

fn_tune.test.param_value

This is a helper function name that is passed to the same-named parameter of the tune.model_param function that is called internally during the execution (see the description of the `tune.model_param` function above).

max.identical.min_RMSE.count = 4

If more than one identical minimum `RMSE` value is calculated during execution, the number of identical minimums is limited by the value of this parameter. When it is reached, the algorithm considers the task execution to be complete.

endpoint.min_diff = 0

Defines the sensitivity threshold for determining the neighborhood boundaries of the minimum `RMSE` value (for details, see the `Details` section below).

break.if_min = TRUE

This is a parameter that is required for the tune.model_param function that is called internally during execution (see the description of the `tune.model_param` function above).

steps.beyond_min = 2

This is a parameter that is required for the tune.model_param function that is called internally during execution (see the description of the `tune.model_param` function above).

Details

First, the function generates a sequence of the `input parameter values` based on the `loop_starter` parameter values, as described above (see the loop_starter parameter description for the details), which is used as one of the input parameters for the helper function tune.model_param, which is repeatedly called during the execution, performing fine-tuning of the model.

The `tune.model_param` function returns a range of input parameter values associated with the set of corresponding *Root Mean Squared Errors* that is also guaranteed to include their minimum value, as described in the Value subsection of the tune.model_param function description.

Next, the function figures out the boundary indices of a range of values from the neighborhood of the minimum `RMSE` by calling internally another helper function `get_fine_tune.param.endpoints.idx` (see the description of the function get_fine_tune.param.endpoints.idx below) and, using one more helper function `get_best_param.result` (see the description of the function get_best_param.result below), a pair of values, corresponding to the best result: minimal `RMSE` and corresponding input parameter value (which is considered the best).

The found values of the boundary indices are then used as the endpoints of a new interval to regenerate the sequence based on it in the next iteration, just as it was done based on the values of the `loop_starter` parameter at the very beginning of the execution. The value of `step divisor`, used to calculate the step of the generated sequence, remains unchanged during the entire execution (see the description of the loop_strarter parameter above for details).

Thus, with each subsequent iteration, the minimum RMSE value is calculated more accurately, over an ever-decreasing interval, with the boundary values tending to the minimum RMSE value, and the sequence generated with an ever-decreasing step.

The calculation is completed when subsequent calculated values of the minimum RMSE stop improving and reach the most accurate possible value.

Value

A data structure containing the minimum RMSE value reached during the fine-tuning process, the corresponding input parameter value, and information about the final sequence, on which the best output values were found.

Source Code

Below is the simplified version of the source code of the model.tune.param_range function:

```
model.tune.param_range <- function(loop_starter,
                                    tune_dir_path,
                                    cache_file_base_name,
                                    fn_tune.test.param_value,
                                    max.identical.min_RMSE.count = 4,
                                    is.cv = TRUE,
                                    endpoint.min_diff = 0, #1e-07,
                                    break.if_min = TRUE,
                                    steps.beyond_min = 2){

  seq_start <- loop_starter[1]
  seq_end <- loop_starter[2]
  interval_divisor <- loop_starter[3]

  if (interval_divisor < 4) {
    interval_divisor <- 4
  }

  prm_val.leftmost <- seq_start
  prm_val.rightmost <- seq_end

  RMSE.leftmost <- NA
  RMSE.rightmost <- NA

  best_RMSE <- NA
  param.best_value <- 0


  param_values.best_result <- c(param.best_value = param.best_value,
                                best_RMSE = best_RMSE)
  # Start repeat loop
  repeat{
    seq_increment <- (seq_end - seq_start)/interval_divisor

    if (seq_increment < 0.0000000000001) {
      warning("Function `model.tune.param_range`:
```

```
parameter value increment is too small.")
      break
    }

    test_param_vals <- seq(seq_start, seq_end, seq_increment)

    tuned_result <- tune.model_param(test_param_vals,
                                     fn_tune.test.param_value,
                                     break.if_min,
                                     steps.beyond_min)

    tuned.result <- tuned_result$tuned.result
    plot(tuned.result$parameter.value, tuned.result$RMSE)

    bound.idx <- get_fine_tune.param.endpoints.idx(tuned.result)
    start.idx <- bound.idx["start"]
    end.idx <- bound.idx["end"]
    best_RMSE.idx <- bound.idx["best"]

    prm_val.leftmost.tmp <- tuned.result$parameter.value[start.idx]
    RMSE.leftmost.tmp <- tuned.result$RMSE[start.idx]

    prm_val.rightmost.tmp <- tuned.result$parameter.value[end.idx]
    RMSE.rightmost.tmp <- tuned.result$RMSE[end.idx]

    min_RMSE <- tuned.result$RMSE[best_RMSE.idx]
    min_RMSE.prm_val <- tuned.result$parameter.value[best_RMSE.idx]

    seq_start <- prm_val.leftmost.tmp
    seq_end <- prm_val.rightmost.tmp

    if (is.na(best_RMSE)) {
      prm_val.leftmost <- prm_val.leftmost.tmp
      RMSE.leftmost <- RMSE.leftmost.tmp

      prm_val.rightmost <- prm_val.rightmost.tmp
      RMSE.rightmost <- RMSE.rightmost.tmp

      param.best_value <- min_RMSE.prm_val
      best_RMSE <- min_RMSE
    }

    if (RMSE.leftmost.tmp - min_RMSE >= endpoint.min_diff) {
      prm_val.leftmost <- prm_val.leftmost.tmp
      RMSE.leftmost <- RMSE.leftmost.tmp
    }

    if (RMSE.rightmost.tmp - min_RMSE >= endpoint.min_diff) {
      prm_val.rightmost <- prm_val.rightmost.tmp
      RMSE.rightmost <- RMSE.rightmost.tmp
    }

    if (end.idx - start.idx <= 0) {
```

```r
      warning("`tuned.result$parameter.value` sequential start index are the same or greater than end o
      break
    }

    if (best_RMSE == min_RMSE) {
      warning("Currently computed minimal RMSE equals the previously reached best one: ",
              best_RMSE, "
Currently computed minial value is: ", min_RMSE)

      if (sum(tuned.result$RMSE[tuned.result$RMSE == min_RMSE]) >= max.identical.min_RMSE.count) {
        warning("Minimal `RMSE`identical values count reached it maximum allowed value: ",
                max.identical.min_RMSE.count)

        param_values.best_result <-
          get_best_param.result(tuned.result$parameter.value,
                                tuned.result$RMSE)
        break
      }

    } else if (best_RMSE < min_RMSE) {
      stop("Current minimal RMSE is greater than previously computed best value: ",
           best_RMSE, "
Currently computed minial value is: ", min_RMSE)
    }

    best_RMSE <- min_RMSE
    param.best_value <- min_RMSE.prm_val

    param_values.best_result <-
      get_best_param.result(tuned.result$parameter.value,
                            tuned.result$RMSE)
  }
  # End repeat loop

  n <- length(tuned.result$parameter.value)
  parameter.value <- tuned.result$parameter.value
  result.RMSE <- tuned.result$RMSE

  if (result.RMSE[1] == best_RMSE) {
    parameter.value[1] <- prm_val.leftmost
    result.RMSE[1] <- RMSE.leftmost
  }
  if (result.RMSE[n] == best_RMSE) {
    parameter.value[n+1] <- prm_val.rightmost
    result.RMSE[n+1] <- RMSE.rightmost
  }

  list(best_result = param_values.best_result,
       param_values.endpoints = c(prm_val.leftmost, prm_val.rightmost, seq_increment),
       tuned.result = data.frame(parameter.value = parameter.value,
                                 RMSE = result.RMSE))
}
```

The full version of the code of the model.tune.param_range is available in the Model Tuning section of the common-helper.functions.R script.

0.2.6.2.3 get_fine_tune.param.endpoints.idx Function

Sugnature

```
get_fine_tune.param.endpoints.idx <- function(preset.result) {
  # ...

  c(start = i,
    end = j,
    best = best_RMSE.idx)
}
```

Parameters

- **fn_tune.test.param_value:** A helper function that calculates the value of the `RMSE` for a given parameter value.;
- **break.if_min = TRUE:** A Boolean parameter that determines whether the function should terminate after completing the number of steps specified by the parameter `steps.beyond_min`, after the minimum value of the `RMSE` has been found;
- **steps.beyond_min = 2:** (takes effect only if `break.if_min` parameter is `TRUE`) Specifies the number of steps after finding the minimum value of the `RMSE`, upon completion of which the function should terminate.

Details

During execution, the function uses a helper function specified by the `fn_tune.test.param_value` parameter, which calculates the `RMSE` value for the given parameter from the list determined by the `param_values` parameter.

Note

Value

Source Code

The source code of the get_fine_tune.param.endpoints.idx function is shown below:

```r
get_fine_tune.param.endpoints.idx <- function(preset.result) {
  best_RMSE <- min(preset.result$RMSE)
  best_RMSE.idx <- which.min(preset.result$RMSE)
  # best_lambda <- preset.result$parameter.value[best_RMSE.idx]

  preset.result.N <- length(preset.result$RMSE)
  i <- best_RMSE.idx
  j <- i

  while (i > 1) {
    i <- i - 1

    if (preset.result$RMSE[i] > best_RMSE) {
      break
    }
  }

  while (j < preset.result.N) {
    j <- j + 1

    if (preset.result$RMSE[j] > best_RMSE) {
      break
    }
  }

  c(start = i,
    end = j,
    best = best_RMSE.idx)
}
```

0.2.6.2.4 get_best_param.result Function

Sugnature

```r
get_best_param.result <- function(param_values, rmses){
  # ...

  c(param.best_value = param_values[best_pvalue_idx],
    best_RMSE = rmses[best_pvalue_idx])
}
```

Parameters

- **fn_tune.test.param_value:** A helper function that calculates the value of the `RMSE` for a given parameter value.;
- **break.if_min = TRUE:** A Boolean parameter that determines whether the function should terminate after completing the number of steps specified by the parameter `steps.beyond_min`, after the minimum value of the `RMSE` has been found;

- **steps.beyond_min = 2:** (takes effect only if `break.if_min` parameter is `TRUE`) Specifies the number of steps after finding the minimum value of the `RMSE`, upon completion of which the function should terminate.

Details

During execution, the function uses a helper function specified by the `fn_tune.test.param_value` parameter, which calculates the `RMSE` value for the given parameter from the list determined by the `param_values` parameter.

Note

Value

Source Code

The source code of the get_best_param.result function is shown below:

```
get_best_param.result <- function(param_values, rmses){
  best_pvalue_idx <- which.min(rmses)
  c(param.best_value = param_values[best_pvalue_idx],
    best_RMSE = rmses[best_pvalue_idx])
```

!!! Note!

0.2.6.2.5 Function

See the `func.model.tune.param_range` Function description above.

Note

Sugnature

Parameters

- **fn_tune.test.param_value:** A helper function that calculates the value of the `RMSE` for a given parameter value.; - **break.if_min = TRUE:** A Boolean parameter that determines whether the function should terminate after completing the number of steps specified by the parameter `steps.beyond_min`, after the minimum value of the `RMSE` has been found; - **steps.beyond_min = 2:** (takes effect only if `break.if_min` parameter is `TRUE`) Specifies the number of steps after finding the minimum value of the `RMSE`, upon completion of which the function should terminate.

Details

During execution, the function uses a helper function specified by the `fn_tune.test.param_value` parameter, which calculates the `RMSE` value for the given parameter from the list determined by the `param_values` parameter.

Note

Value

Source Code

Below is the most significant part of the code of the function function:

!!! Note!

!!! Note!

!!! Note!

0.2.6.2.6 Function

Sugnature

Parameters

- **fn_tune.test.param_value:** A helper function that calculates the value of the `RMSE` for a given parameter value.; - **break.if_min = TRUE:** A Boolean parameter that determines whether the function should terminate after completing the number of steps specified by the parameter `steps.beyond_min`, after the minimum value of the `RMSE` has been found; - **steps.beyond_min = 2:** (takes effect only if `break.if_min` parameter is `TRUE`) Specifies the number of steps after finding the minimum value of the `RMSE`, upon completion of which the function should terminate.

Details

During execution, the function uses a helper function specified by the `fn_tune.test.param_value` parameter, which calculates the `RMSE` value for the given parameter from the list determined by the `param_values` parameter.

> Note

Value

Source Code

Below is the most significant part of the code of the function function:

0.2.6.3 UME Model Regularization: Support Function

> The regularize.test_lambda.UM_effect.cv function described below are defined in the Regularization section of the UM-effect.functions.R script.

0.2.6.3.1 `regularize.test_lambda.UM_effect.cv` Function

This function calculates *RMSE* of the *UME Model* using *5-Fold CV* method for the given λ parameter value:

```
regularize.test_lambda.UM_effect.cv <- function(lambda){
  if (is.na(lambda)) {
    stop("Function: regularize.test_lambda.UM_effect.cv
`lambda` is `NA`")
  }
  um_effect <- train_user_movie_effect.cv(lambda)
  calc_user_movie_effect_RMSE.cv(um_effect)
}
```

> Note that we reuse the function train_user_movie_effect.cv calling it from the regularize.test_lambda.UM_effect.cv, but now with the λ parameter different from the default (zero) value.

Let's now figure out the λ that minimizes the *RMSE*:

```
# Here we will simply compute the RMSE for different values of `lambda`
n <- colSums(!is.na(y))

sums <- colSums(y - mu - a, na.rm = TRUE)
lambdas <- seq(0, 10, 0.1)

rmses <- sapply(lambdas, function(lambda){
```

```
  b <-  sums / (n + lambda)
  reg_rmse(b)
})

# Here is a plot of the RMSE versus `lambda`:
plot(lambdas, rmses, type = "l")
```

Now we can determine the minimal *RMSE*:

```
# print(min(rmses))
```

which is achieved for the following λ:

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

Using this λ we can compute the regularized estimates:

```
b_reg <- sums / (n + lambda)

str(b_reg)
```

Finally, let's verify that the penalized estimates $\hat{b}_i(\lambda)$ we have just computed actually result in the minimal *RMSE* figured out above:

```
reg_rmse(b_reg)
```

0.2.7 Accounting for Date effects

0.2.7.0.1 Yearly rating count(Motefaker 2024)

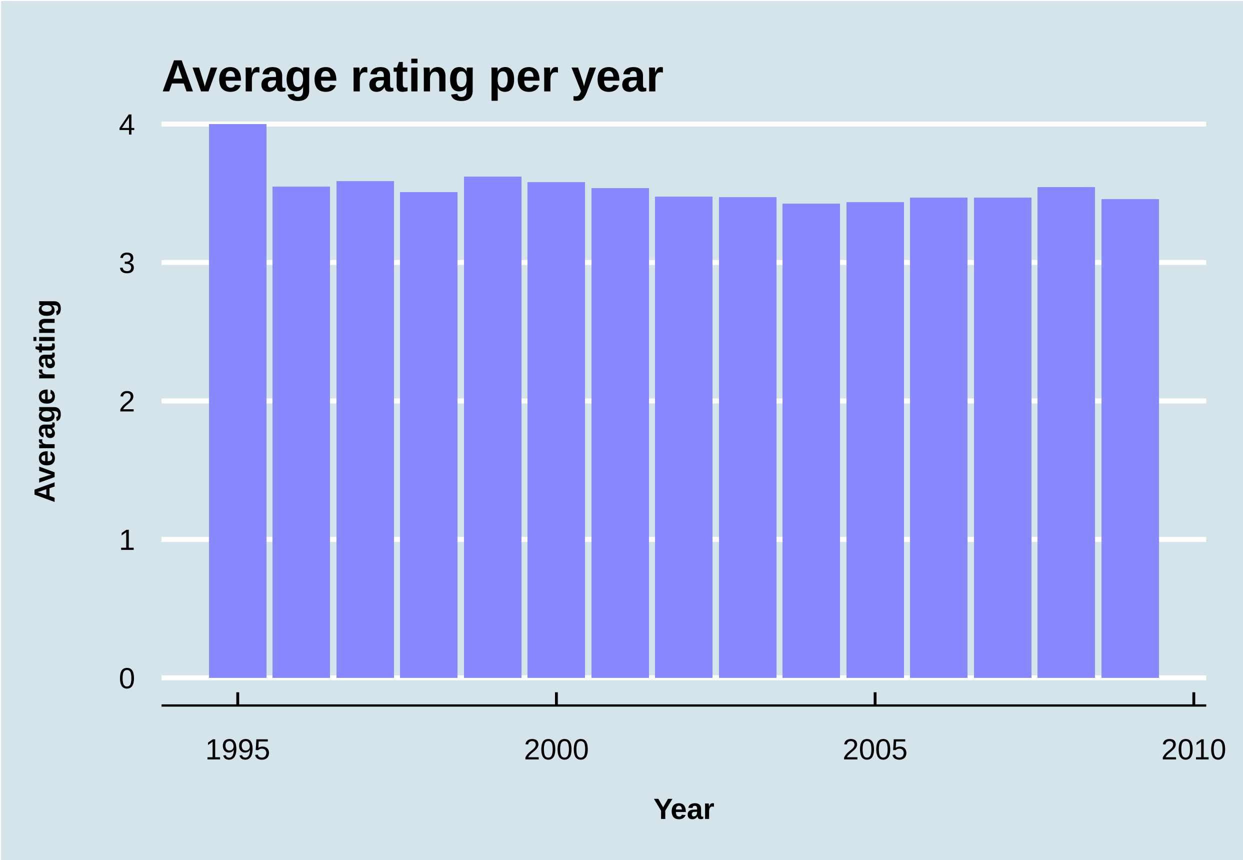
```
print(edx |>
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) |>
  group_by(year) |>
  summarize(count = n())
)
```

```
## # A tibble: 15 x 2
##     year   count
##    <dbl>   <int>
##  1  1995       2
##  2  1996  942772
##  3  1997  414101
##  4  1998  181634
##  5  1999  709893
##  6  2000 1144349
##  7  2001  683355
##  8  2002  524959
##  9  2003  619938
```

```
## 10  2004  691429
## 11  2005 1059277
## 12  2006  689315
## 13  2007  629168
## 14  2008  696740
## 15  2009   13123
```

0.2.7.0.2 Average rating per year plot(Motefaker 2024)

```
edx |>
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) |>
  group_by(year) |>
  summarize(rating_avg = mean(rating)) |>
  ggplot(aes(x = year, y = rating_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"))
```



We use the following models to account for the `date` effect:

$$Y_{i,j} = \mu + \alpha_i + \beta_j + f(d_{i,j}) + \varepsilon_{i,j}$$

0.2.8 Accounting for Genre effect

As mentioned in Section 23.7: Exercises of the *Chapter "23 Regularization" of the Course Textbook* the `Movielens` dataset also has a genres column. This column includes every genre that applies to the movie (some movies fall under several genres)(Irizarry 2024g).

0.2.8.1 Genre Data Analysis

Movie Genres Data

The following code computes movie rating summaries by popular genres like Drama, Comedy, Thriller, and Romance:

```
#library(stringr)
genres = c("Drama", "Comedy", "Thriller", "Romance")
sapply(genres, function(g) {
  sum(str_detect(edx$genres, g))
})
```

Further, we can find out the movies that have the greatest number of ratings using the following code:

```
ordered_movie_ratings <- edx |> group_by(movieId, title) |>
  summarize(number_of_ratings = n()) |>
  arrange(desc(number_of_ratings))
print(head(ordered_movie_ratings))
```

and figure out the most given ratings in order from most to least:

```
ratings <- edx |>  group_by(rating) |>
     summarise(count = n()) |>
     arrange(desc(count))
print(ratings)
```

The following code allows us to summarize that in general, half-star ratings are less common than whole-star ratings (e.g., there are fewer ratings of 3.5 than there are ratings of 3 or 4, etc.):

```
print(edx |> group_by(rating) |> summarize(count = n()))
```

We can visually see that from the following plot:

```
edx |>
  group_by(rating) |>
  summarize(count = n()) |>
  ggplot(aes(x = rating, y = count)) +
  geom_line()
```

0.2.8.1.1 Movie Genres Effect

The plot below shows strong evidence of a genre effect (for illustrative purposes, the plot shows only categories with more than 20, 000 ratings).

```
# Preparing data for plotting:
genre_ratins_grp <- train_set |>
  mutate(genre_categories = as.factor(genres)) |>
  group_by(genre_categories) |>
  summarize(n = n(), rating_avg = mean(rating), se = sd(rating)/sqrt(n())) |>
  filter(n > 20000) |>
  mutate(genres = reorder(genre_categories, rating_avg)) |>
  select(genres, rating_avg, se, n)

dim(genre_ratins_grp)
genre_ratins_grp_sorted <- genre_ratins_grp |> sort_by.data.frame(~ rating_avg)
print(genre_ratins_grp_sorted)

# Creating plot:
genre_ratins_grp |>
  ggplot(aes(x = genres, y = rating_avg, ymin = rating_avg - 2*se, ymax = rating_avg + 2*se)) +
  geom_point() +
  geom_errorbar() +
  ggtitle("Average rating per Genre") +
  ylab("Average rating") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

Below are worst and best ratings categories:

```
sprintf("The worst ratings are for the genre category: %s",
        genre_ratins_grp$genres[which.min(genre_ratins_grp$genres)])

sprintf("The best ratings are for the genre category: %s",
        genre_ratins_grp$genres[which.max(genre_ratins_grp$genres)])
```

Another way of visualizing a genre effect is shown in the section Average rating for each genre of the article "Movie Recommendation System using R - BEST" written by Amir Moterfaker(Motefaker 2024):

```
# For better visibility, we reduce the data for plotting
# while keeping the worst and best rating rows:
plot_ind <- odd(1:nrow(genre_ratins_grp))
plot_dat <- genre_ratins_grp_sorted[plot_ind,]

plot_dat |>
  ggplot(aes(x = rating_avg, 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 = 8),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```

If we define $g_{i,j}$ as the genre for user's i rating of movie j, we can use the following models to account for the `genre` effect:

To account for *genre effects* we will use the model suggested in the Section 23.7: Exercises of the *Chapter "23 Regularization" of the Course Textbook*(Irizarry 2024g):

$$Y_{i,j} = \mu + \alpha_i + \beta_j + g_{i,j} + \varepsilon_{i,j}$$

where $g_{i,j}$ is an *aggregation function* which is explained in detail in *Section 22.3: "Review of Aggregation Functions" of "Recommender Systems Handbook"* (*Chapter 22: "Aggregation of Preferences in Recommender Systems"*, p. 712) book(Ricci 2011).

In the formula above $g_{i,j}$ denotes a *genre effect* for user's i rating of movie j, so that:

$$g_{i,j} = \sum_{k=1}^{K} x_{i,j}^k \gamma_k$$

with $x_{i,j}^k = 1$ if $g_{i,j}$ includes genre k, and $x_{i,j}^k = 0$ otherwise.

$$Y_{i,j} = \mu + \alpha_i + \beta_j + g_{i,j} + f(d_{i,j})$$

$$\sum_{i=1}^{n_i} (Y_{i,j} - \mu - \alpha_i)$$

0.3 Conclusion

Hello Conclusion!

This is a great conclusion, isn't it?!!

Andreas Toscher, Robert M. Bell, Michael Jahrer. 2009. "The BigChaos Solution to the Netflix Grand Prize: Commendo Research & Consulting." September 5, 2009. https://www.asc.ohio-state.edu/statistics/statgen/joul_aut2009/BigChaos.pdf.

Irizarry, Rafael A. 2024a. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.1.1: Movielens Data: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movielens-data.

———. 2024b. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.2: Loss Function: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#sec-netflix-loss-function.

———. 2024c. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.3: A First Model: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#a-first-model.

———. 2024d. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.4: User Effects: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#user-effects.

———. 2024e. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.5: Movie Effects: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movie-effects.

———. 2024f. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.6: Penalized Least Squares: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#penalized-least-squares.

———. 2024g. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.7: Exercises: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#exercises.

———. 2024h. *Introduction to Data Science, Part II: Statistics and Prediction Algorithms Through Case Studies*. https://rafalab.dfci.harvard.edu/dsbook-part-2/.

Kurbanaev, Azamat. 2025a. "edX Data Science: Capstone, MovieLens Datasets: Package: Edx.capstone.movielens.data." February 5, 2025. https://github.com/AzKurban-edX-DS/edx.capstone.movielens.data.

———. 2025b. "edX Data Science: Capstone-MovieLens Project: A Movie Recommendation System Using the MovieLens Dataset." May 5, 2025. https://github.com/AzKurban-edX-DS/Capstone-MovieLens/tree/main.

Motefaker, Amir. 2024. "Movie Recommendation System Using r - BEST." July 18, 2024. https://www.kaggle.com/code/amirmotefaker/movie-recommendation-system-using-r-best/notebook.

Ricci, Francesco. 2011. *Recommender Systems Handbook*. Edited by Paul B. Kantor Lior Rokach Bracha Shapira. Springer, New York. https://doi.org/10.1007/978-0-387-85820-3.

Yehuda Koren, Robert Bell, Yahoo Research, and Chris Volinsky. 2009. "Matrix Factorization Techniques for Recommender Systems." August 1, 2009. https://datajobs.com/data-science-repo/Recommender-Systems-%5BNetflix%5D.pdf.