Capstone Movielens Report

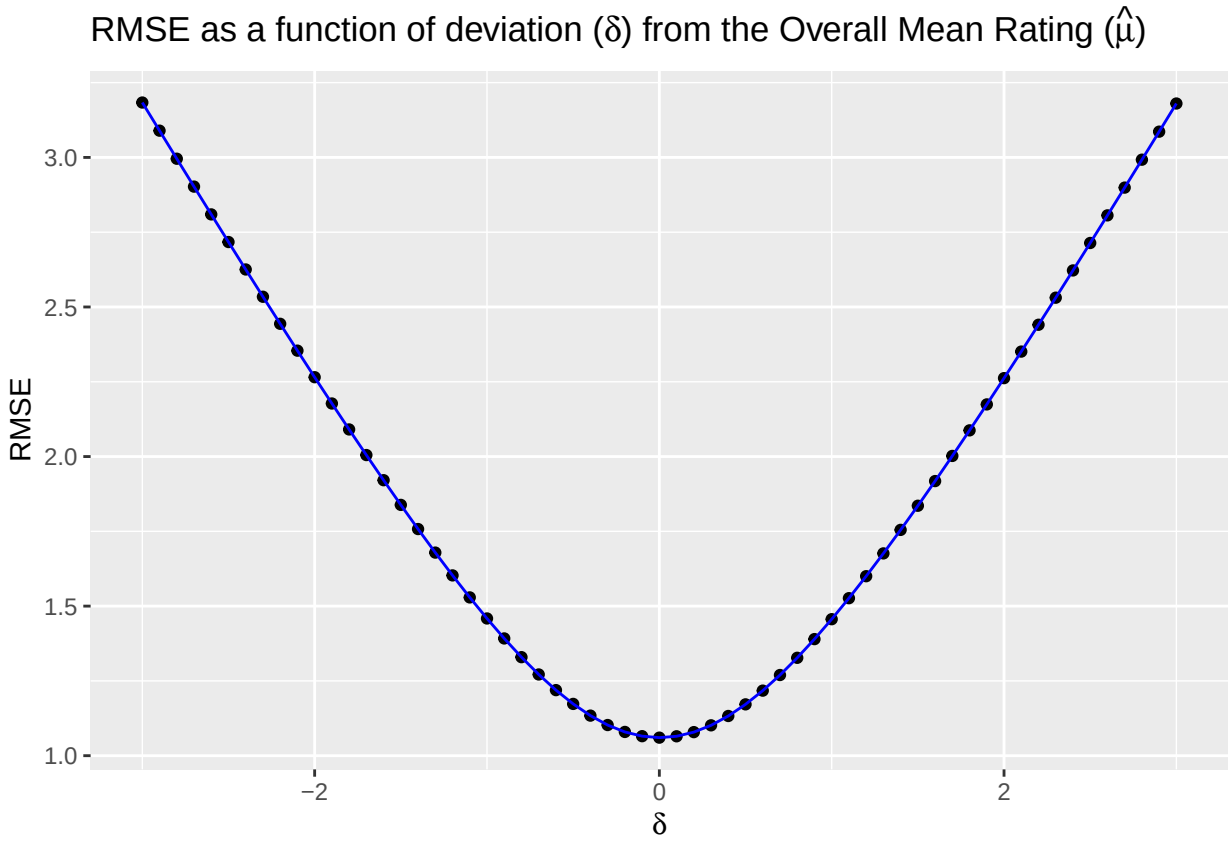
Azamat Kurbanaev

2025-05-04

Contents

Plot dependency of RMSEs vs Overal Mean Rating Deviation

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



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:

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

with N being the number of user/movie combinations for which we make predictions and the sum occurring over all these combinations[1].

Our goal is to achieve a value of less than 0.86490 (compare with the *Netflix Grand Prize* requirement: of at least 0.8563[2]).

Datasets

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[3].

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

```
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

summary(edx)
```

```
##      userId         movieId          rating        timestamp           title              genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:9000055     Length:900005
##  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:999999
##  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
```

`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 83
##  $ 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
```

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

```r
#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
```

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

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

```
##    Drama   Comedy Thriller  Romance
##  3910127  3540930  2325899  1712100
```

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

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

```
## `summarise()` has grouped output by 'movieId'. You can override using the `.groups` argument.
```

```r
print(head(ordered_movie_ratings))
```

```
## # A tibble: 6 x 3
## # Groups:   movieId [6]
##   movieId title                               number_of_ratings
##     <int> <chr>                                           <int>
## 1     296 Pulp Fiction (1994)                             31362
## 2     356 Forrest Gump (1994)                             31079
## 3     593 Silence of the Lambs, The (1991)                30382
## 4     480 Jurassic Park (1993)                            29360
## 5     318 Shawshank Redemption, The (1994)                28015
## 6     110 Braveheart (1995)                               26212
```

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)
```

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

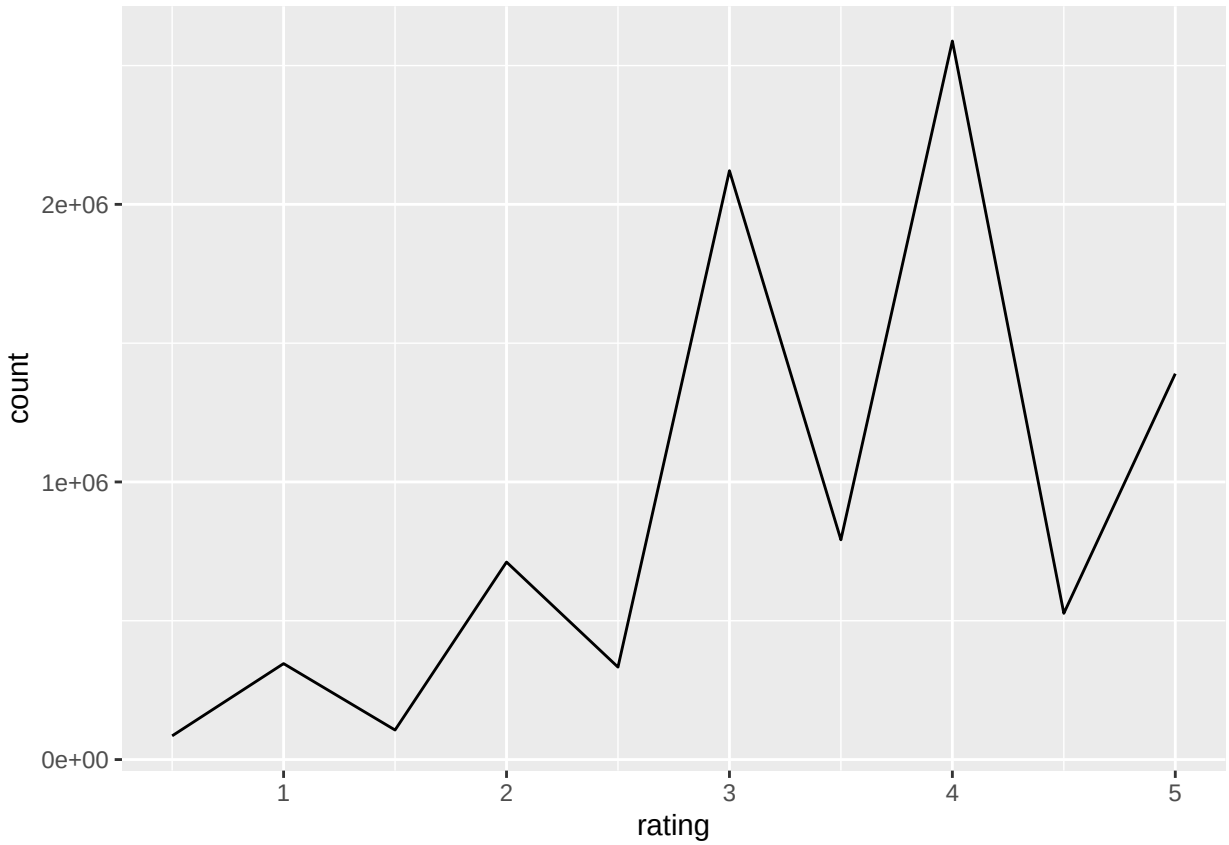
The following code allows us to summarize that in general, half-star ratings are less common than whole-star ratings (e.g., there are fewer ratings of 3.5 than there are ratings of 3 or 4, etc.):

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

```
## # A tibble: 10 x 2
##    rating   count
##     <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
```

We can visually see that from the following plot:

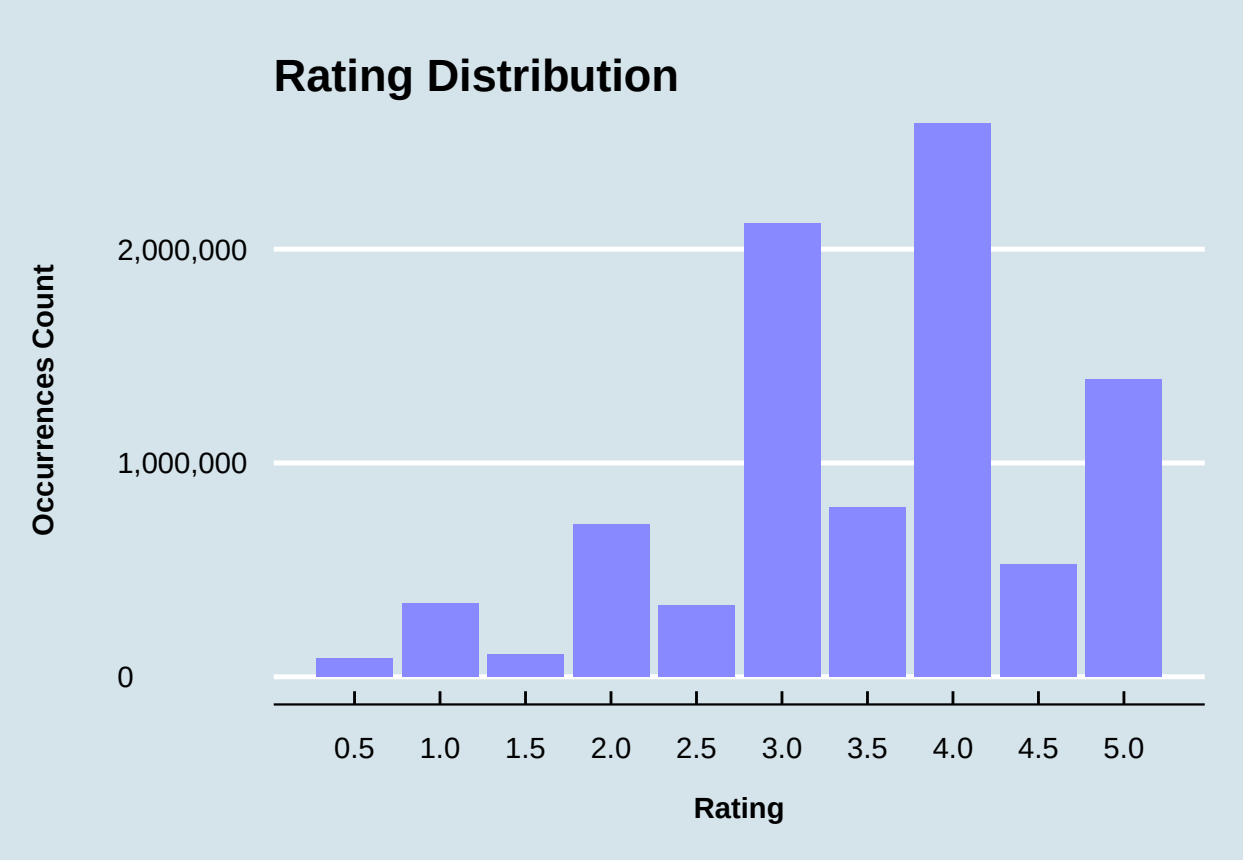
```
edx |>
  group_by(rating) |>
  summarize(count = n()) |>
  ggplot(aes(x = rating, y = count)) +
  geom_line()
```



Further analysis of the `edx` dataset have been also inspired by the article mentioned above[4], from which the code and explanatory notes below were cited.

Rating distribution plot[4]
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") +
  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[4].

Ratings per movie

Movie popularity count[4]

```
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
```

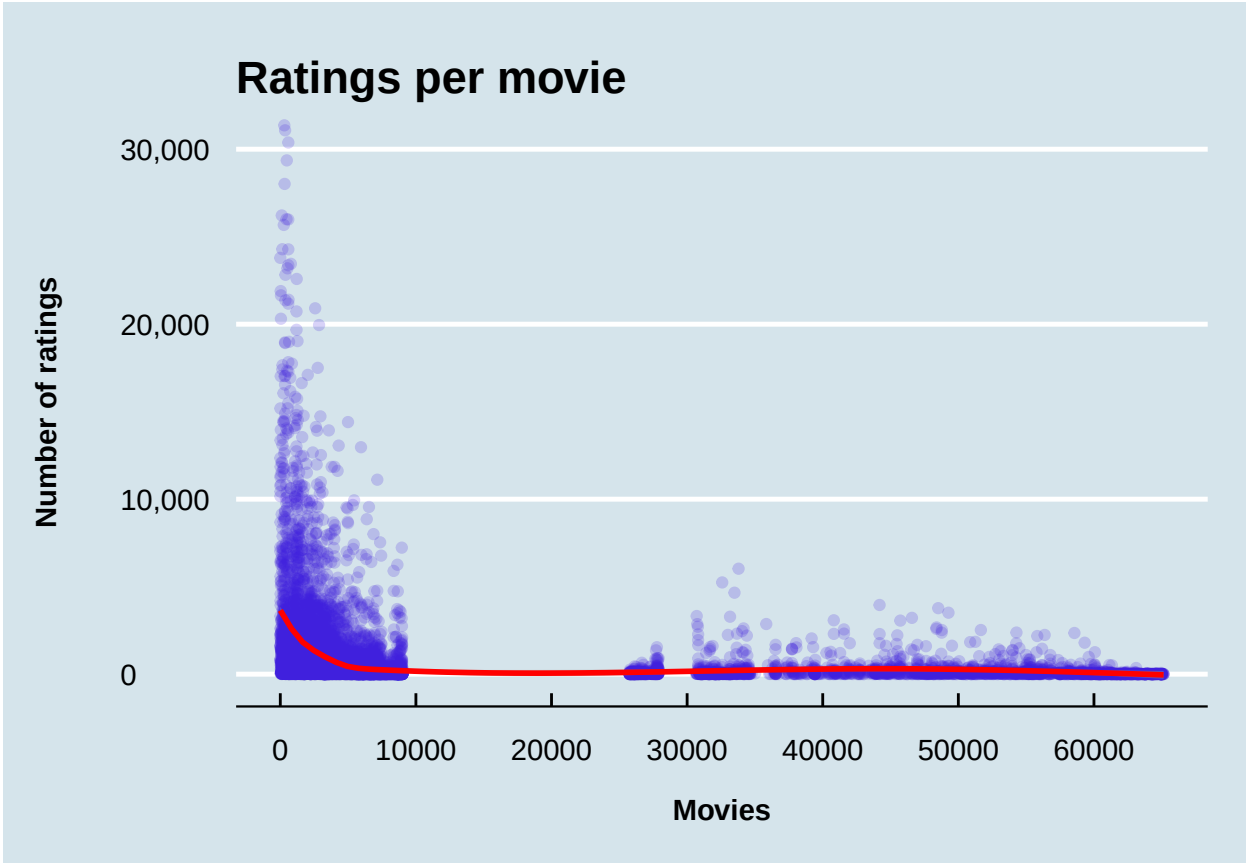
```
summary(edx |> group_by(movieId) |> summarize(count = n()) |> select(count))
```

```
##      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[4]

```
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"))
```

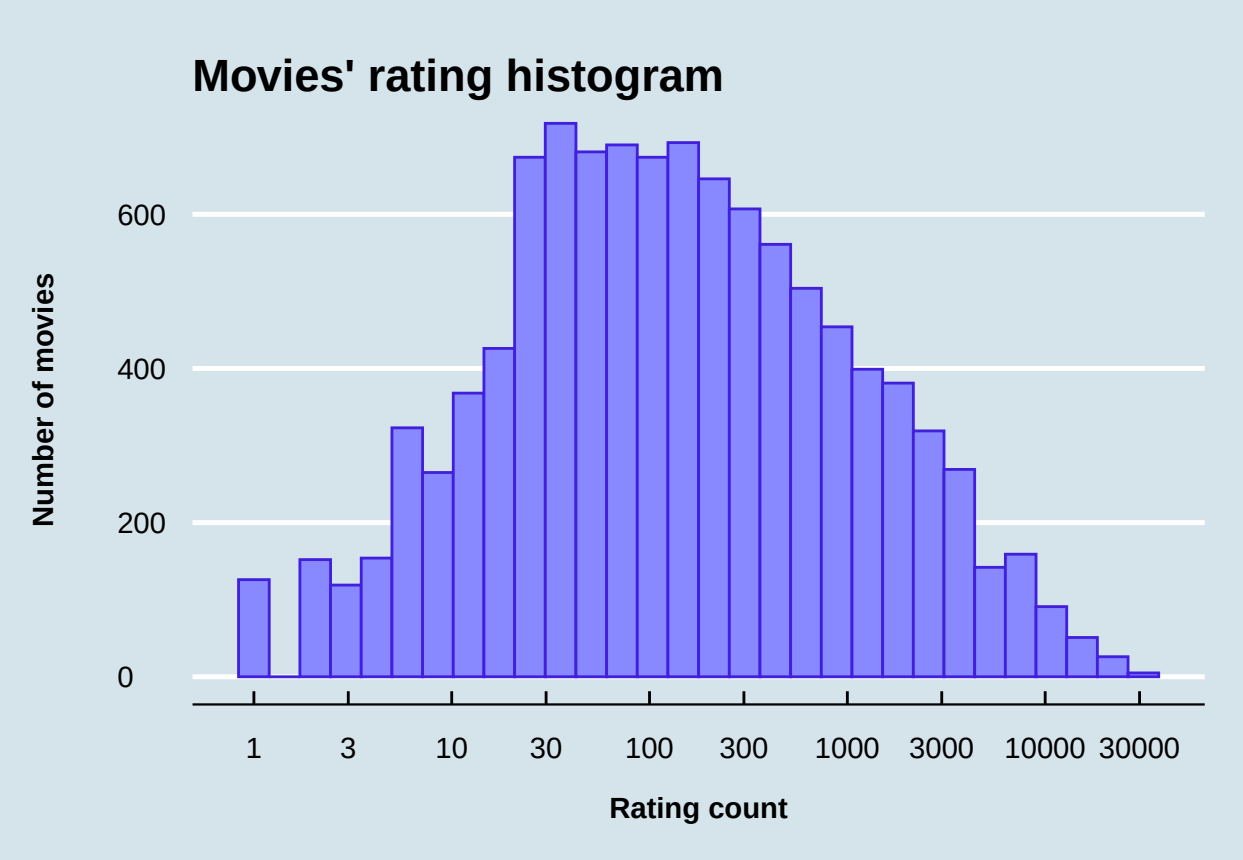
```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



Movies' rating histogram[4]

```
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`.
```



Ratings per user[4]

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

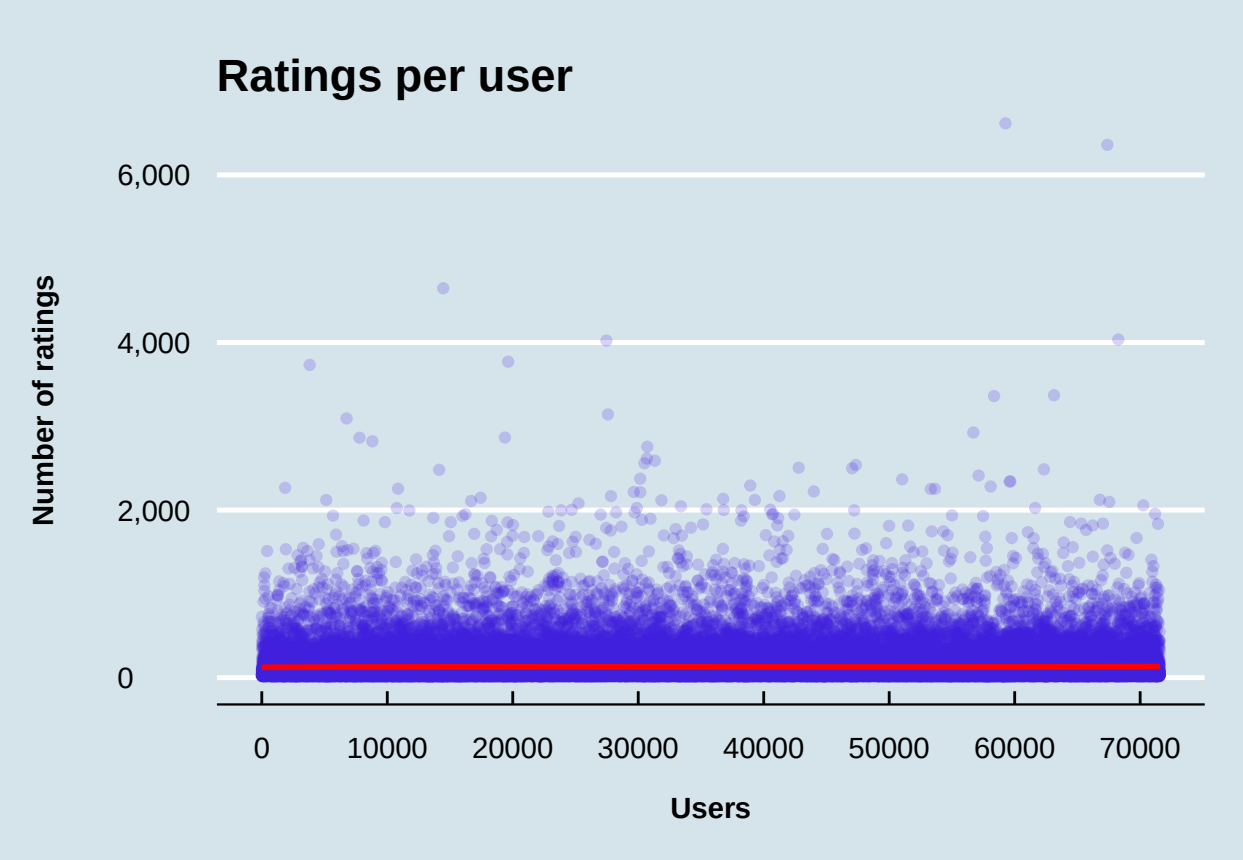
```
summary(edx |> group_by(userId) |> summarize(count = n()) |> select(count))
```

```
##      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

```
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"))
```

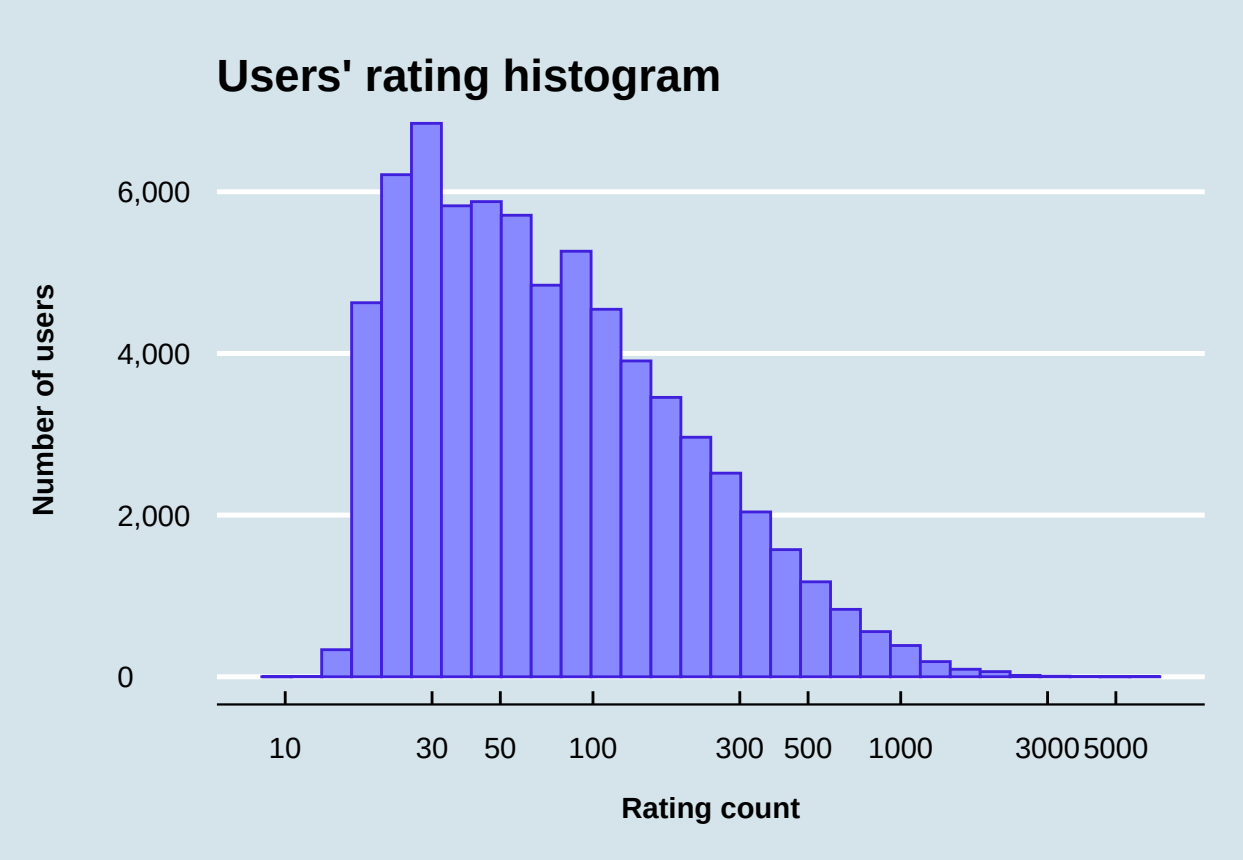
```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



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`.
```



Methods / Analysis

All the source code of the R-scripts is available on the project's GitHub repository.

Defining Logging and Mesuring helper functions

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:

```
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(){
  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)
}

print_log <- function(msg){
  print(str_glue(msg))
}
put_log <- function(msg){
  put(str_glue(msg))
}
put_log1 <- function(msg_template, arg1){
  msg <- str_replace_all(msg_template, "%1", as.character(arg1))
  put(str_glue(msg))
}
put_log2 <- function(msg_template, arg1, arg2){
  msg <- msg_template |>
    str_replace_all("%1", as.character(arg1)) |>
    str_replace_all("%2", as.character(arg2)) |>
    str_glue()

  put(msg)
}
```

The full source code of these functions is available in the Logging Helper function section of the Capstone MovieLens Main R Script.

Preparing train and set datasets

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:

```
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*[5] that not every user rated every movie:

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

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

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

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

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

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

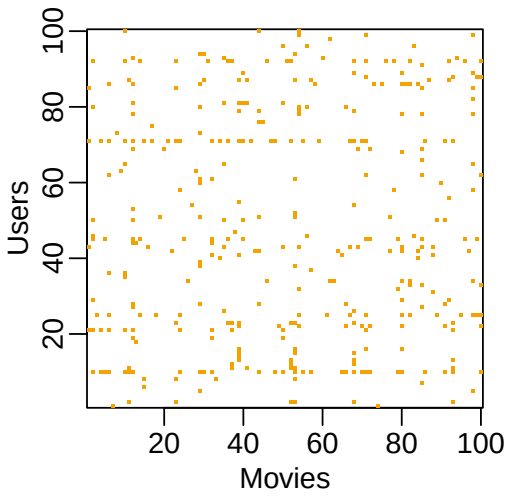
As also explained in that section, we can think of these data as a very large matrix, with users on the rows and movies on the columns, with many empty cells. Therefore, we can think of a recommendation system as filling in the `NA`s in the dataset for the movies that some or all the users do not rate. A sample from the `edx` data below illustrates this idea[6]:

```
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                 
```

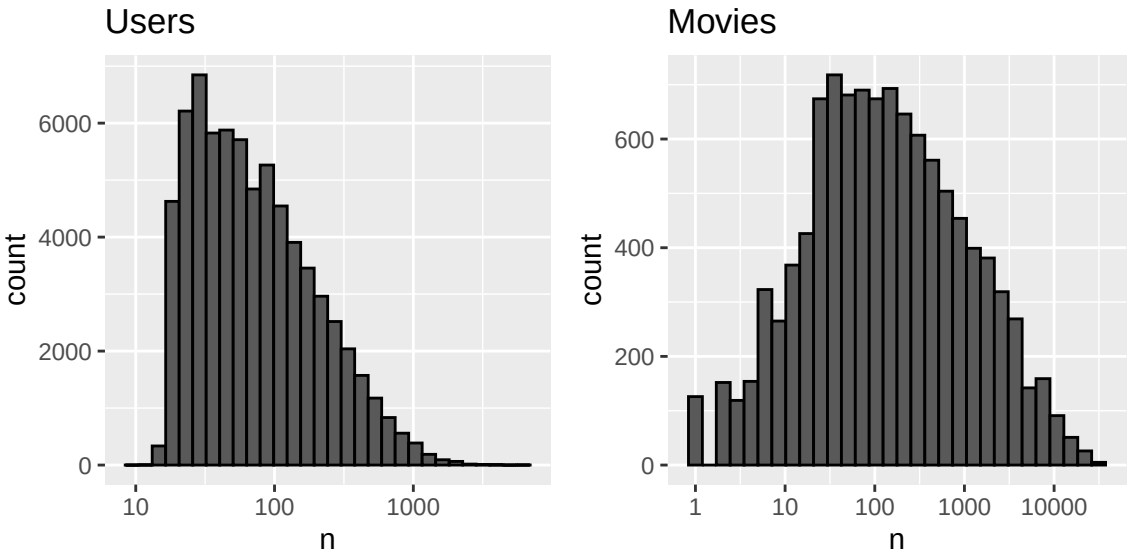
The following plot of the matrix for a random sample of 100 movies and 100 users with yellow indicating a user/movie combination for which we have a rating shows how *sparse* the matrix is:

```
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)
```



Taking into consideration these observations, we came up with a decision to use the data from the dataset only for users who have provided at least 100 ratings.

Now let's 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 the Section 23.1.1 Movielens data of the *Course Textbook* mentioned above[6]:

```
# Let's ignore the data for users who have not provided at least 100 ratings:
edx100 <- edx |>
  group_by(userId) |>
  filter(n() >= 100) |>
  ungroup()

print(edx100 |> summarize(n_distinct(userId), n_distinct(movieId)))
```

```
## # A tibble: 1 x 2
##   `n_distinct(userId)` `n_distinct(movieId)`
##                  <int>                 <int>
## 1                24115                 10665
```

```
# For each one of these users, we will split their ratings into 80% for training
# and 20% for testing:

set.seed(2006)
indexes <- split(1:nrow(edx100), edx100$userId)
test_ind <- sapply(indexes, function(i) sample(i, ceiling(length(i)*.2))) |>
  unlist() |>
  sort()

test_set <- edx100[test_ind,]
train_set <- edx100[-test_ind,]

# To make sure we don't include movies in the training set that should not be
# there, we remove entries using the semi_join function:
test_set <- test_set |> semi_join(train_set, by = "movieId") |> as.data.frame()
summary(test_set)
```

```
##      userId         movieId          rating        timestamp            title              genres
##  Min.   :    8   Min.   :    1   Min.   :0.500   Min.   :8.248e+08   Length:1397040     Length:13970
##  1st Qu.:18161   1st Qu.: 1022   1st Qu.:3.000   1st Qu.:9.655e+08   Class :character   Class :chara
##  Median :35666   Median : 2109   Median :3.500   Median :1.058e+09   Mode  :character   Mode  :chara
##  Mean   :35902   Mean   : 4616   Mean   :3.473   Mean   :1.051e+09
##  3rd Qu.:53645   3rd Qu.: 3993   3rd Qu.:4.000   3rd Qu.:1.135e+09
##  Max.   :71565   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
```

```
train_set <- mutate(train_set, userId = factor(userId), movieId = factor(movieId))
summary(train_set)
```

```
##      userId           movieId           rating        timestamp            title              genres
##  59269  :   5292   356    :  12769   Min.   :0.500   Min.   :8.248e+08   Length:5539951     Length:5
##  67385  :   5088   296    :  12439   1st Qu.:3.000   1st Qu.:9.657e+08   Class :character   Class :c
##  14463  :   3718   480    :  12390   Median :3.500   Median :1.059e+09   Mode  :character   Mode  :c
##  68259  :   3228   593    :  12333   Mean   :3.472   Mean   :1.051e+09
##  27468  :   3218   260    :  12142   3rd Qu.:4.000   3rd Qu.:1.135e+09
##  19635  :   3016   1196   :  11586   Max.   :5.000   Max.   :1.231e+09
##  (Other):5516391   (Other):5466292
```

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:

```
y <- dplyr::select(train_set, movieId, userId, rating) |>
pivot_wider(names_from = movieId, values_from = rating) |>
column_to_rownames("userId") |>
as.matrix()

dim(y)
```

```
## [1] 24115 10630
```

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

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

summary(movie_map)
```

```
##     movieId         title              genres
##  1      :    1   Length:10630       Length:10630
##  2      :    1   Class :character   Class :character
##  3      :    1   Mode  :character   Mode  :character
##  4      :    1
##  5      :    1
##  6      :    1
##  (Other):10624
```

Note that titles cannot be considered unique, so we can't use them as IDs[6].

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:

```
mu <- mean(y, na.rm = TRUE)
print(mu)
```

```
## [1] 3.47184
```

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

```
rmse(test_set$rating - mu)
```

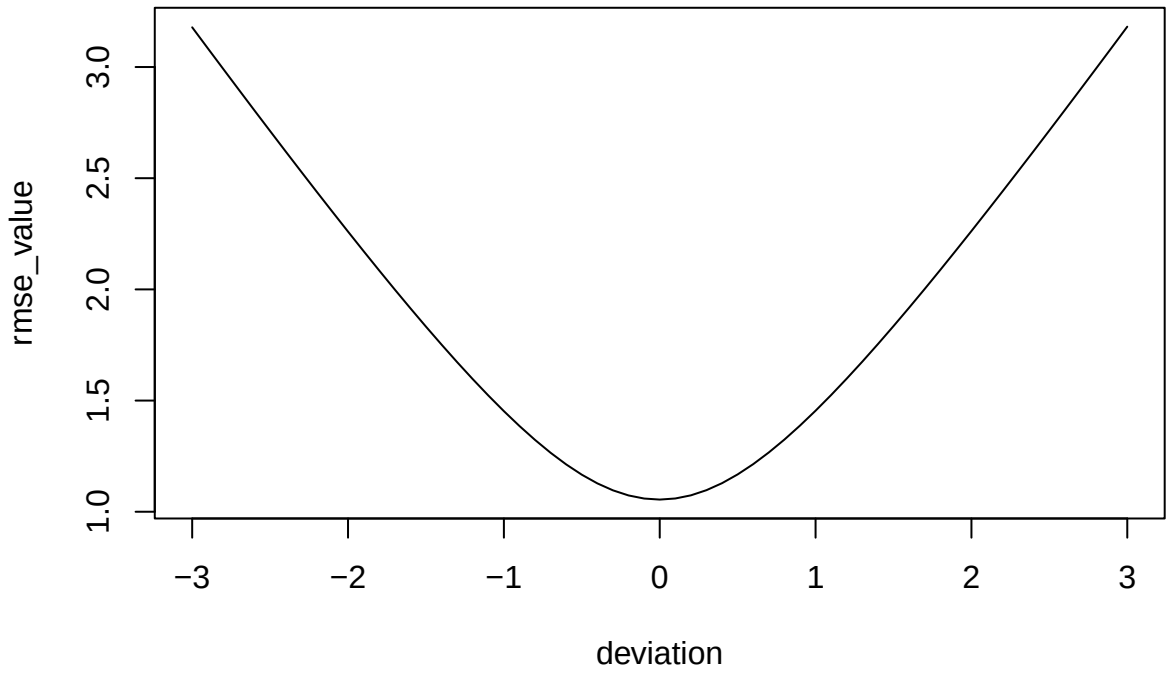
```
## [1] 1.054739
```

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

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

```
##  [1] -3.0 -2.9 -2.8 -2.7 -2.6 -2.5 -2.4 -2.3 -2.2 -2.1 -2.0 -1.9 -1.8 -1.7 -1.6 -1.5 -1.4 -1.3 -1.2 
## [37]  0.6  0.7  0.8  0.9  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4
```

```
rmse_value <- sapply(deviation, function(diff){
  rmse(test_set$rating - mu + diff)
})

plot(deviation, rmse_value, type = "l")
```




```
sprintf("Minimum RMSE is achieved when the deviation from the mean is: %s",
        deviation[which.min(rmse_value)])
```

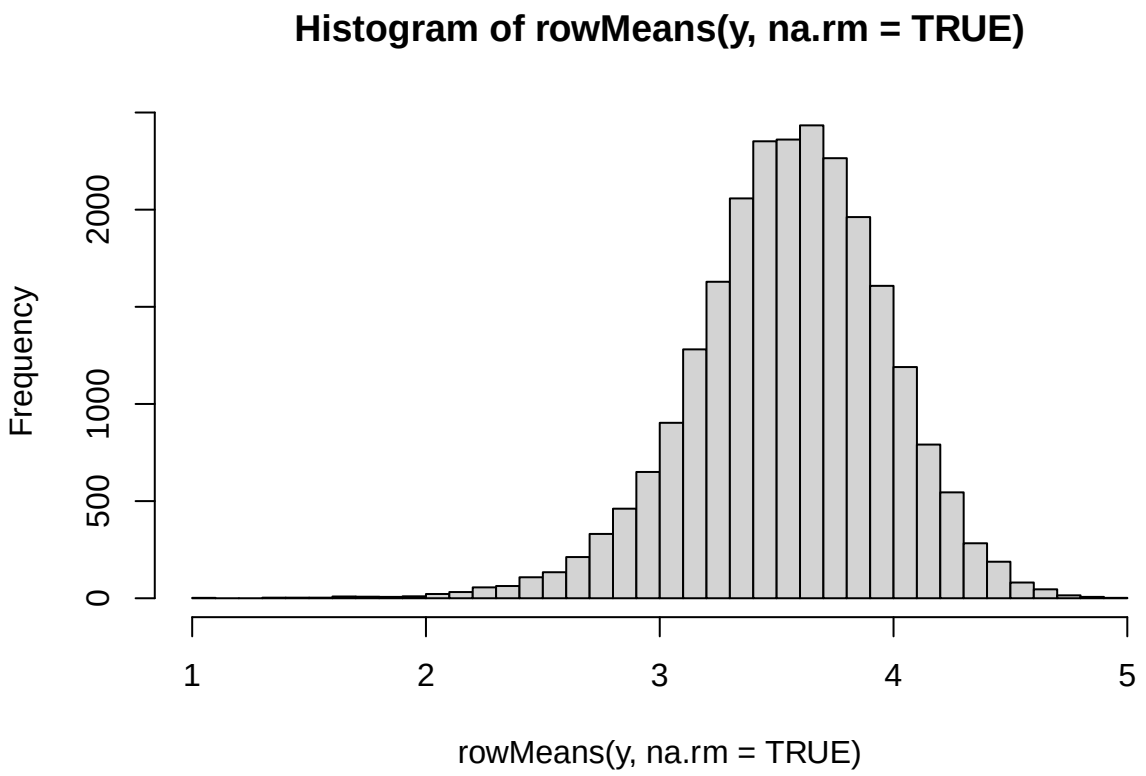
```
## [1] "Minimum RMSE is achieved when the deviation from the mean is: 0"
```

To win the grand prize of $1,000,000, a participating team had to get an RMSE of at least 0.8563[2]. So we can definitely do better![7]

Taking into account User effects

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 the author shows, we can see that there is substantial variability in the average ratings across users:

```
hist(rowMeans(y, na.rm = TRUE), 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*[8, 9].

As it is stated here[8], 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 compute them this way:

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

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

```
# Compute the RMSE taking into account user effects:
user_effects_rmse <- test_set |>
    left_join(data.frame(userId = as.integer(names(a)), a = a), by = "userId") |>
    mutate(resid = rating - clamp(mu + a)) |>
    filter(!is.na(resid)) |>
    pull(resid) |> rmse()

print(user_effects_rmse)
```

```
## [1] 0.9707208
```

Taking into account Movie effects

In Section *23.5 Movie effects* 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 $y_{i,j} - \hat{\mu} - \hat{\alpha}_i$:

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

We can now construct predictors and see how much the `RMSE` improves[10]:

```
user_and_movie_effects_rmse <- test_set |>
    left_join(data.frame(userId = as.integer(names(a)), a = a), by = "userId") |>
    left_join(data.frame(movieId = as.integer(names(b)), b = b), by = "movieId") |>
    mutate(resid = rating - clamp(mu + a + b)) |>
    filter(!is.na(resid)) |>
    pull(resid) |> rmse()

print(user_and_movie_effects_rmse)
```

```
## [1] 0.8662207
```

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.

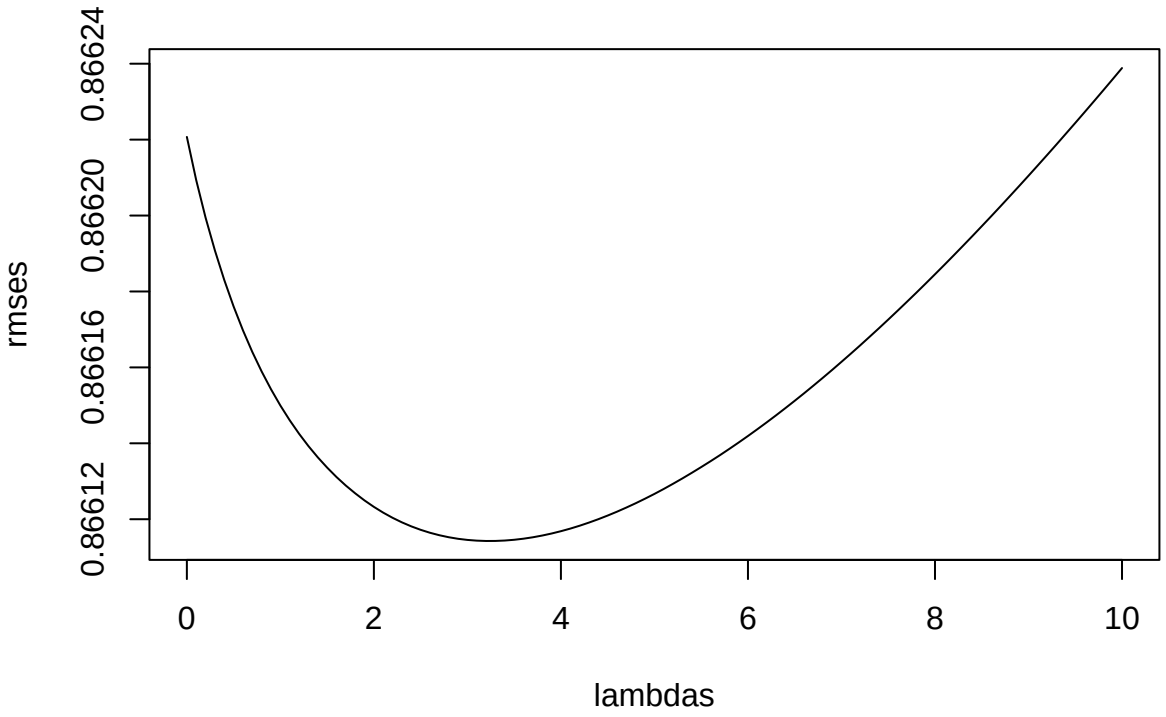
This approach will have our desired effect: when our sample size n_j is very large, we obtain a stable estimate and the penalty λ is effectively ignored since $n_j + \lambda \approx n_j$. Yet when the n_j is small, then the estimate $\hat{\beta}_i(\lambda)$ is shrunken towards 0. The larger the λ, the more we shrink[11].

Support function
We will use the following function to calculate *RMSE* in this section:

```
reg_rmse <- function(b){
  test_set |>
    left_join(data.frame(userId = as.integer(names(a)), a = a), by = "userId") |>
    left_join(data.frame(movieId = as.integer(names(b)), b = b), by = "movieId") |>
    mutate(resid = rating - clamp(mu + a + b)) |>
    filter(!is.na(resid)) |>
    pull(resid) |> rmse()
}
```

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))
```

```
## [1] 0.8661143
```

which is achieved for the following λ:

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

```
## [1] 3.2
```

Using this λ we can compute the regularized estimates:

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

str(b_reg)
```

```
##  Named num [1:10630] -0.516 0.327 -0.903 -0.154 -0.29 ...
##  - attr(*, "names")= chr [1:10630] "5" "6" "19" "22" ...
```

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)
```

```
## [1] 0.8661143
```

Accounting for Date effects

Yearly rating count[4]

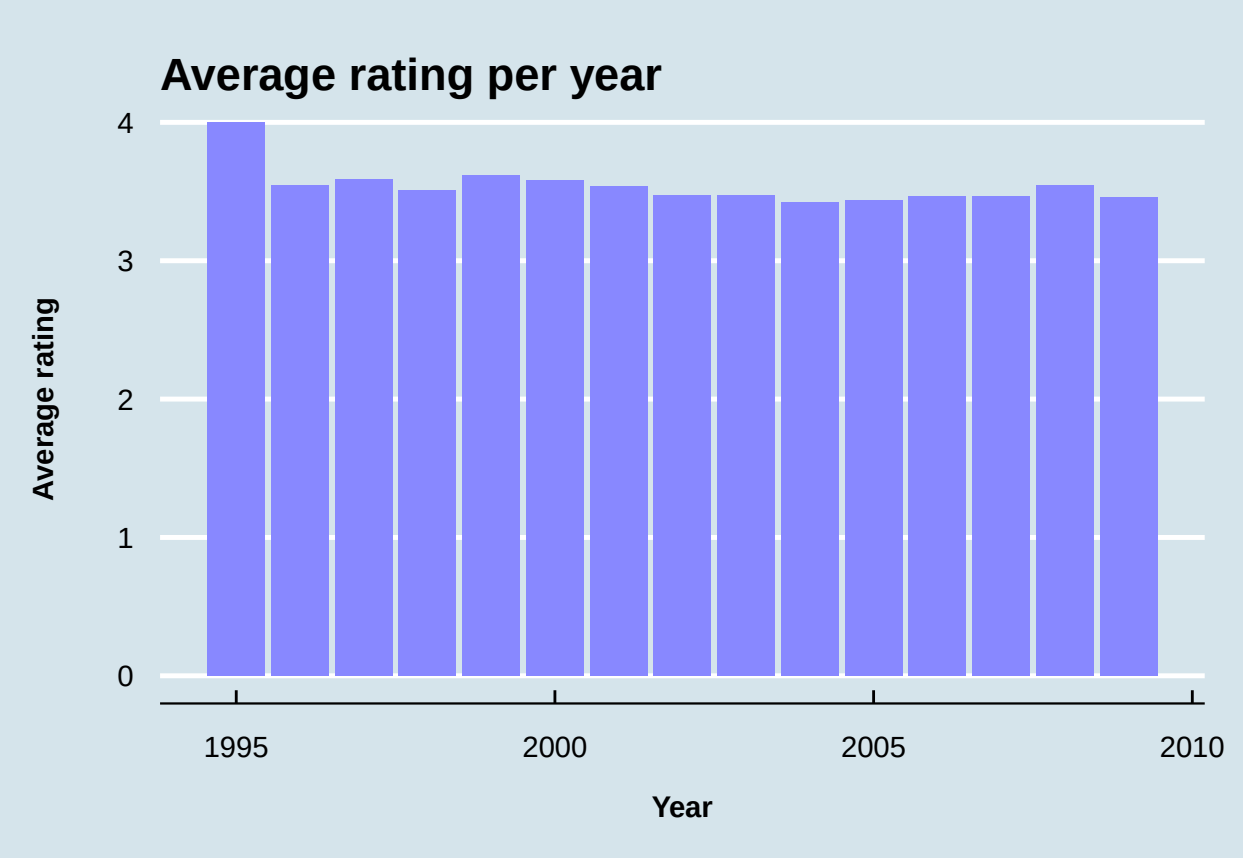
```
print(edx |>
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) |>
  group_by(year) |>
  summarize(count = n())
)
```

```
## # 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
```

Average rating per year plot[4]

```
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:

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

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)[12].

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

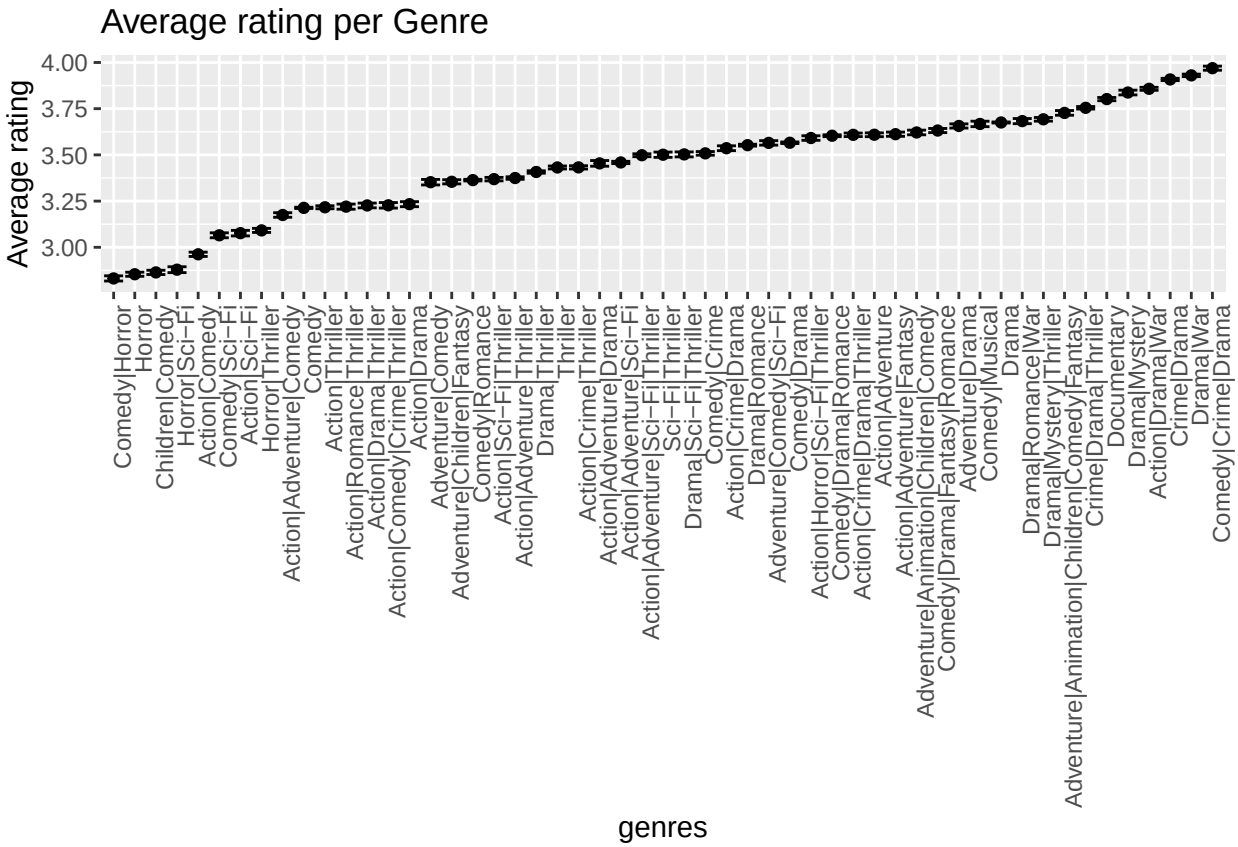
```
# Preparing data for plotting:
genre_ratins_grp <- train_set |>
  mutate(genre_categories = as.factor(genres)) |>
  group_by(genre_categories) |>
  summarize(n = n(), rating_avg = mean(rating), se = sd(rating)/sqrt(n())) |>
  filter(n > 20000) |>
  mutate(genres = reorder(genre_categories, rating_avg))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

dim(genre_ratins_grp)
```

```
## [1] 53  4
```

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

```
## # A tibble: 53 x 4
##    genres                  rating_avg      se      n
##    <fct>                        <dbl>   <dbl>  <int>
##  1 Comedy|Horror                 2.83 0.00701  24892
##  2 Horror                        2.85 0.00556  48671
##  3 Children|Comedy               2.86 0.00575  40435
##  4 Horror|Sci-Fi                 2.88 0.00787  20524
##  5 Action|Comedy                 2.96 0.00581  34118
##  6 Comedy|Sci-Fi                 3.06 0.00668  27888
##  7 Action|Sci-Fi                 3.08 0.00749  27971
##  8 Horror|Thriller               3.09 0.00543  49751
##  9 Action|Adventure|Comedy       3.17 0.00616  28256
## 10 Comedy                        3.21 0.00167 439352
## # i 43 more rows
```

```
# 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)])
```

```
## [1] "The worst ratings are for the genre category: Comedy|Horror"
```

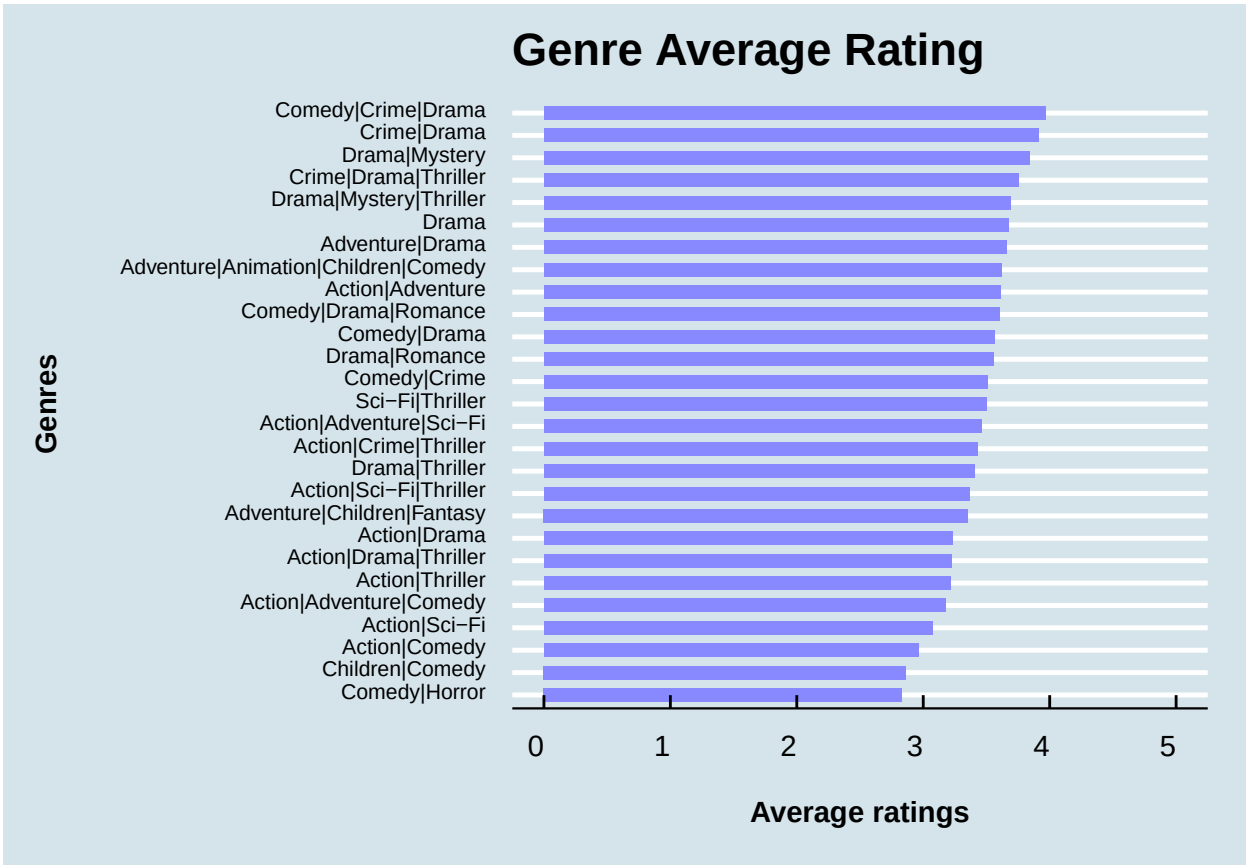
```
sprintf("The best ratings are for the genre category: %s",
        genre_ratins_grp$genres[which.max(genre_ratins_grp$genres)])
```

```
## [1] "The best ratings are for the genre category: Comedy|Crime|Drama"
```

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[4]:

```
# 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*[12]:

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

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[13].

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

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

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

Conclusion

Hello Conclusion!

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

References

[1] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.2: Loss function. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#sec-netflix-loss-function (visited on 02/18/2025) (cit. on p. 2).

[2] Robert M. Bell Andreas Toscher Michael Jahrer. *The BigChaos Solution to the Netflix Grand Prize. commendo research & consulting*. Sept. 5, 2009. URL: https://www.asc.ohio-state.edu/statistics/statgen/joul_aut2009/BigChaos.pdf (visited on 02/18/2025) (cit. on pp. 2, 20).

[3] Azamat Kurbanaev. *edX Data Science: Capstone, MovieLens Datasets. Package: edx.capstone.movielens.data*. Version 0.0.0.9000. Feb. 5, 2025. URL: https://github.com/AzKurban-edX-DS/edx.capstone.movielens.data (visited on 02/05/2025) (cit. on p. 2).

[4] Amir Motefaker. *Movie Recommendation System using R - BEST*. Version 284. July 18, 2024. URL: https://www.kaggle.com/code/amirmotefaker/movie-recommendation-system-using-r-best/notebook (visited on 02/18/2025) (cit. on pp. 6–10, 24, 25, 27).

[5] Rafael A. Irizarry. *Introduction to Data Science, Part II. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/ (visited on 02/18/2025) (cit. on p. 15).

[6] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.1.1: Movielens data. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movielens-data (visited on 02/18/2025) (cit. on pp. 15, 17, 18).

[7] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.3: A first model. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#a-first-model (visited on 02/18/2025) (cit. on p. 20).

[8] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.4: User effects. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#user-effects (visited on 02/18/2025) (cit. on p. 21).

[9] Robert Bell Yehuda Koren Yahoo Research and Chris Volinsky. *Matrix Factorization Techniques for Recommender Systems*. Aug. 1, 2009. URL: https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf (visited on 02/18/2025) (cit. on p. 21).

[10] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.5: Movie effects. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movie-effects (visited on 02/18/2025) (cit. on p. 22).

[11] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.6: Penalized least squares. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#penalized-least-squares (visited on 02/18/2025) (cit. on p. 22).

[12] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.7: Exercises. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#exercises (visited on 02/18/2025) (cit. on pp. 26, 28).

[13] Francesco Ricci. *Recommender Systems Handbook*. Ed. by Paul B. Kantor Lior Rokach Bracha Shapira. Springer, New York, 2011. ISBN: ISBN 978-0-387-85819-7. DOI: 10.1007/978-0-387-85820-3. URL: https://github.com/vwang0/recommender_system/blob/master/Recommender%20Systems%20Handbook.pdf (cit. on p. 28).