Movielens Ratings

Exploring and Predicting Movie Ratings

Kework K. Kalustian

Contents

Introduction **1**

1 Preparing the Environment and Data Preprocessing **3**
1.1 Retrieving the Movielens Dataset 4
1.1 Data Cleaning and Wrangling 5

2 Analysis I: Exploring Effects on Average Movie Ratings **7**
2.1 Effect of Movie Rating Counts 8
2.2 Effect of User Rating Counts 10
2.3 Effect of Movie Release Years 11
2.4 Effect of Rating Dates 13
2.5 Effect of Genre-Label Rating Counts 14

3 Analysis II and Methods: Predicting Movie Rating **16**
3.1 Finding the best Lambda, or: Regularizing and Penalizing 21
3.2 Predicting Final Movie Ratings by using Ridge Regression 24

4. Results **27**

5. Concluding Remarks **27**

References **27**

Appendix — Environment **27**

Introduction

People like movies. As there have been so many movies released in the past decades, people might find it, however, often hard to decide which movie they should watch, especially when these people have already watched the most popular ones. In this light, it appears helpful to have movie recommendations at hand to facilitate people's movie selection and finally their decision whether they should watch a particular movie. The main idea behind such a recommendation system is that a movie can be potentially worth watching as soon as it is likely that people or, respectively, users will aesthetically appreciate the recommended movie in question. In this vein, an aesthetic appreciation is being present when users would rate the recommended movie on a rating scale high or, respectively, as highly preferred (e.g., in "star"-units or, in a binary version, with a "thumb-up"-rating). Now, based on this assumption, there are three main options for how such a movie recommendation system might work.

The one option is to recommend movies to known people based on, for example, given ratings of these people for other movies. This is, very simply put, the core of the so-called "content-based recommender system" (see for example Gorakala & Usuelli, 2015: pp. 3-4).

The other option is to recommend movies based on unknown movie ratings for new users of a recommendation interface in which the recommendation system is embedded. In this case, a possible recommendation system, could provide movie recommendations for new users, for example, by initially suggesting the most popular movies so that more specific recommendations can be given, once those popular movies have been rated by these new users. This option characterizes basically the main idea of the so-called "knowledge-based recommender system" (see ibid.), since the developer of this system have applied their knowledge about popular movies.

To filter ratings of different users, to recommend movies to those users who rate, for example, the same movies to the same extent, but also watch different movies, is the third option. Accordingly, this option refers to the paradigm that is known as a "collaborative filter recommender system" (see ibid.).

Of course, it is also possible to combine the strengths of these three options so that one could build a dynamic or more differentiated recommendation system. Above all, movie recommendations are, to put it in a nutshell, in the end, predictions and/or classifications of movies that are potentially aesthetically appealing for known and/or new users.

In this project, we focus on the third case. That is, in general, we aim to build a movie recommendation system in which we predict movie ratings based on already given ratings from known users. To put it the other way around: As we assume that ratings pose a reasonable (i.e, quantitatively measurable) operationalization of the latent factor "recommendation", we aim to predict such ratings: Thus, highly accurate or less erroneous rating-predictions mean in this context highly recommended movies for given users.

Building on these background assumptions, we will use in this project 10 million movie ratings of 10,000 movies from about 70,000 users that are listed in the so-called "MovieLens" dataset from the Grouplens (2009) research lab at the Computer Science Department and Engineering of the University of Minnesota (Twin Cities, USA) to proof a concept of a movie recommendation system whose prediction errors yield an $RMSE$-value less than or equal to 0.86490. Hence, we can now ask the following generic research question:

- *How are predictions of movie ratings with an RMSE-value lesser than 0.86490 achievable?*

To answer this question, we will import the just introduced movie dataset and explore in this project movie ratings according to the following hypotheses so that we can decide whether we want to include such effects into our prediction model at stake:

- H_1: If the rating counts for movies are high, the average ratings for movies are high as well.
- H_2: If users rate many movies, the users tend to give on average lower ratings.
- H_3: If movies have been released recently, the given average ratings for these movies tend to be lower than for older movies.
- H_4: If movies have been rated recently, the given average ratings for these movies tend to be lower than for movies that have been rated anciently.
- H_5: If movies are characterized by genre labels, which are rated frequently, their average ratings tend to be lower than for movies with genre labels that are seldom rated.

Once we have these hypotheses visually explored, we will solve the prediction-problem from a supervised machine learning perspective by using training and test datasets for experimentation and cross-validation for model-development. After doing so, we can finally apply our best performing model on the validation dataset, which will be *only* used for the final movie rating

predictions. That is, even for the visual explorations of those hypotheses, we will not include the validation dataset. Hence, we can avoid overfitting and, respectively, a circular prediction strategy. Accordingly, the remainder of this report is structured as follows:

In the first section, the processes of data retrieval, cleaning, and wrangling are documented and illustrated with detailed code-chunks. The second section revolves around visual data explorations according to the introduced hypotheses; code-chunks are here also included. The third section of this report is about the introduced prediction-problem in particular. To solve this problem, we will model a simple least-square-solution by using the training and test datasets. To optimally penalize the coefficients and, respectively, the biases, which form the predictors of that basic least-square-approach, we will make use of cross-validation. The fourth section is a summary of these results. Lastly, in a concluding section, we will wrap up this report.

1 Preparing the Environment and Data Preprocessing

First of all, to quantitatively examine such hypotheses, we will use the scripting language and statistical environment R and, respectively, RStudio. By doing so, we need to load and, respectively, install some packages, which consist of the required functions for the analyses at stake. To load and install these packages conveniently, we should write a generic function to facilitate our workflow. Especially when we are about to load multiple packages and want to ensure reproducibility, this approach appears much smoother since we do not need to repeat the same commands again and again—to simplify the workload with little effort, provides mostly a good reward.

```
# First, we make sure that our working environment and memory are nice and
# clean (i.e., empty).

remove(list = ls(all = T)); gc(T,T,T)

# Generating a blank function to load all desired packages

load_packages <- function(){

# Object of desired packages

desired.packages <- c(# Tidy coding paradigm
                      "tidyverse", "magrittr",
                      # Data import
                      "readr",
                      # Data frames
                      "tibble", "data.table",
                      # Data wrangling
                      "lubridate",
                      # Graphics
                      "ggExtra", "ggrepel", "scales",
                      # Machine Learning
                      "caret",
                      # Reporting
                      "knitr", "kableExtra")

# Object which contains the desired packages as long as they are not already
# installed.
```

```
inst.packages <- desired.packages[!desired.packages %in% installed.packages()]

# Generating a for-loop.

for (packages in inst.packages) {install.packages(packages, dependencies = T)}

sapply(desired.packages, require, character = T)}

# Finally, using the just generated function to load and/or install the desired
# packages.

load_packages()
```

The first step is done. Next, we can begin with the actual work by retrieving the data that we aim to analyze in the following sections.

1.1 Retrieving the Movielens Dataset

After loading and installing the desired packages, we now have to download the introduced movie rating dataset. By doing so, we retrieve the respective movie and user IDs, the movie ratings, the movie titles and genres, and the rating times. The following code lines, which are provided from the HarvardX Data Science Series (2020), will do the job:

```
# Downloading and structuring the data

dl <- tempfile()
download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

ratings <- fread(text = gsub("::", "\t",
                             readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")),
                          "\\::", 3)

colnames(movies) <- c("movieId", "title", "genres")

# If using R 4.0 or later:

movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres))

movielens <- left_join(ratings, movies, by = "movieId")

# Validation set will be 10% of the MovieLens dataset
# Setting the seed to ensure reproducibility

set.seed(1, sample.kind = "Rounding")

test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1,
```

```
                                    list = F)

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

# Making sure that userId and movieId in the validation set are also in edx set

validation <- temp %>%
              semi_join(edx, by = "movieId") %>%
              semi_join(edx, by = "userId")

# Adding rows that are removed from the validation set back into edx set

removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)

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

To obtain now an overview of the training dataset (i.e., the `edx` dataset), we should take a look at the first few rows of all columns:

Table 1: Retrieved training dataset

userId	movieId	rating	timestamp	title	genres
1	122	5	838985046	Boomerang (1992)	Comedy|Romance
1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller

Having the entire datasets downloaded, we clean and, respectively, manipulate the datasets now for our analyses since the (almost) raw datasets still contain information that can be more clarified. For example, the year information within the title column appears not remotely tidy since each year-information should be in a separate column that can be named, for example, "release_year". Similarly, the genre column entails multiple genre labels (i.e., observations) for each movie in one row, which appears likewise untidy, since each observation should also form a single row. Thus, following the "tidyverse"-paradigm, such improper or untidy information should be clarified (see Wickham, 2013). The next section is, therefore, dedicated to this subject.

1.1 Data Cleaning and Wrangling

For cleaning the data, we will finally add, remove, and manipulate already given information. After doing so, we can handle the data conveniently according to our needs. However, it is of ample importance to keep in mind that we do *not* add information that is not already given, we make the given information only easier to analyze. The following code will do the job:

```
# Extracting the unix-timestamps according to the README information of the
# MovieLens dataset.

edx_star <- edx %>%
            mutate(# Converting the given timestamps into dates with time
                   # information.
```

```
                  rating_timestamp = as_datetime(timestamp,
                                                 origin = "1970-01-01",
                                                 tz = "GMT"),
                  # Assigning the dates of the ratings.
                  rate_date = as_date(rating_timestamp),
                  # Assigning the last 6 characters/strings from the titles
                  # (i.e., the parentheses and the year numbers) to a new
                  # column.
                  year =  str_sub(title, -6),
                  # Removing the parentheses of the year information.
                  release_year = gsub("[[:punct:]]", "", year),
                  # Converting the release_year information into integers
                  # (since year information without month and day information
                  # cannot be considered as date).
                  release_year = as.integer(release_year),
                  # Removing the last 6 characters/strings from the
                  # titles.
                  title = str_sub(title, 1, nchar(title) - 6)) %>%
            # Deselecting not needed/redundant columns. As there are multiple
            # "select"-functions, we have to make sure to use the here the one
            # from the dyplr-package which also belongs to the tidyverse.
            dplyr::select(-c(rating_timestamp, timestamp, year))

# As the genre labels are collapsed in one row for each movie, we should
# separate them so that each genre label of a movie gets its own row. Since
# this a time-consuming process, we separate this process form the code chunk
# above, to accelerate this entire wrangling process.

edx_star %<>%
separate_rows(genres, sep = "\\|")

# Exporting the retrieved, cleaned, and wrangled dataset as .csv-file in order
# to load the cleaned and wrangled dataset for faster access when rerunning the
# analyses.

write_csv(edx_star, "edx_star.csv")

# As we later want to predict the ratings of the validation dataset based on
# the edx_star-ratings, we need, of course, apply this very cleaning and
# wrangling procedure to the validation dataset. Hence, the comments from above
# are here omitted as the process is basically the same).

valid_star <- validation  %>%
              mutate(rating_timestamp = as_datetime(timestamp,
                                                    origin = "1970-01-01",
                                                    tz = "GMT"),

                     rate_date = as_date(rating_timestamp),

                     year =  str_sub(title, -6),
```

```
                    release_year = gsub("[[:punct:]]", "", year),

                    release_year = as.integer(release_year),

                    title = str_sub(title, 1, nchar(title) - 6)) %>%

             dplyr::select(-c(rating_timestamp, timestamp, year))

valid_star %<>%
separate_rows(genres, sep = "\\|")

write_csv(valid_star, "valid_star.csv")
```

At a glance at the same first few rows of the cleaned and wrangled training dataset, we see an improvement regarding the clarity of the data in that way that not useful as well as redundant information has been removed whereas other already given information has been reshaped. That is, both datasets are now more conveniently analyzable.

Table 2: Cleaned and wrangled training dataset

userId	movieId	rating	title	genres	rate_date	release_year
1	122	5	Boomerang	Comedy	1996-08-02	1992
1	122	5	Boomerang	Romance	1996-08-02	1992
1	185	5	Net, The	Action	1996-08-02	1995
1	185	5	Net, The	Crime	1996-08-02	1995
1	185	5	Net, The	Thriller	1996-08-02	1995
1	292	5	Outbreak	Action	1996-08-02	1995
1	292	5	Outbreak	Drama	1996-08-02	1995
1	292	5	Outbreak	Sci-Fi	1996-08-02	1995
1	292	5	Outbreak	Thriller	1996-08-02	1995

So, after we have done the needed data cleaning and wrangling, we can now visually explore the general properties of the training dataset according to our hypotheses.

2 Analysis I: Exploring Effects on Average Movie Ratings

Since in both datasets no missing values (i.e., `NA`s) are observable, we can move on to explore the independent (input) variables and the movie ratings as the dependent (output) variable.

```
#Mean and SD of movie ratings

mean(edx_star$rating); sd(edx_star$rating)

# Plotting the frequency distribution of movie ratings

ggplot(data = edx_star) +

geom_histogram(aes(rating), bins = 10, fill = "navyblue", color = "#280000",
               alpha = 0.3) +
```

```
# Label numbers with SI prefixes

scale_y_continuous(labels = label_number_si(accurarcy = NULL)) +

# Customizing the labels on both axes

labs(x = "\nMovie Ratings\n(Stars)", y = "Rating Counts\n") +

# Layout

theme_bw(base_size = 14) +
theme(axis.title.x = element_text(size = 14),
      axis.title.y = element_text(size = 14),
plot.margin = unit(c(.66,.33,.66,.33), "cm"))

# Saving the output

ggsave("ratings.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
       width = 21.33, height = 9, dpi = 320)
```

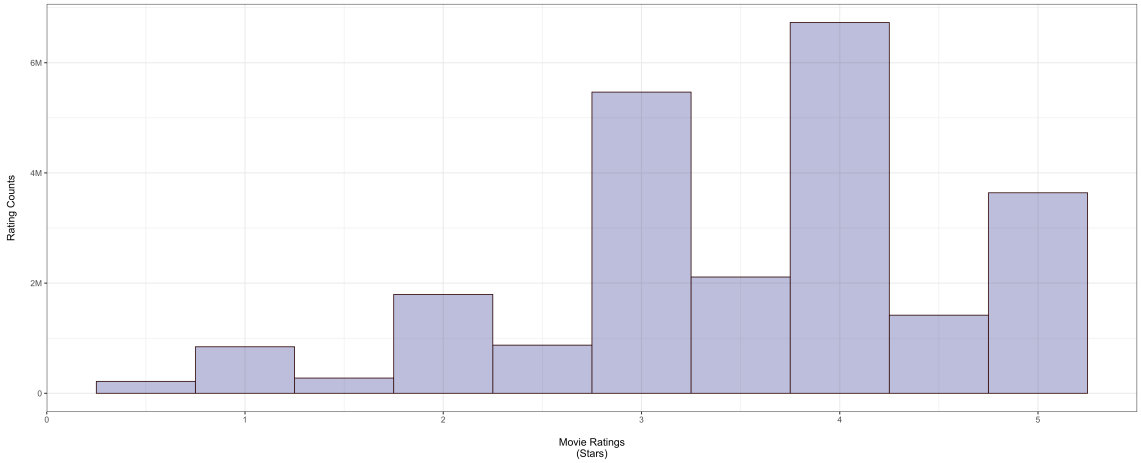

Figure 1: Histogram of movie ratings (training dataset).

As we see in the histogram of the overall movie ratings of the training dataset, the ratings range between 0.5 and 5 star-units, and are slightly left-skewed or, respectively, right-leaning which in turn means that most of the ratings are rather high ($M_{Ratings} = 3.527; SD_{Ratings} = 1.052$). While keeping this in mind for the predictions at stake, we will explore in the next sections the effects, which are hypothesized in the introduction of this project.

2.1 Effect of Movie Rating Counts

Now, we plot the average rating of each movie against the respective rating counts to visualize the assumption of H_1 that higher rating counts positively influence the average rating of a movie:

```
# Visual exploration of H1
```

```r
movie_avg_rating <- edx_star %>%
                    group_by(movieId) %>%
                    summarize(n = n(),
                              movie_rating = mean(rating))

p_m <- ggplot(movie_avg_rating,aes(n, movie_rating)) +

       geom_point(color = "navyblue", size = 0.3, alpha = 0.3) +

       # Trend line

       geom_smooth(color = "darkred", size = 0.75) +

       # Label numbers with SI prefixes and log-scaling

       scale_x_continuous(labels = label_number_si(accurarcy = NULL),
                          trans = "log10")+

       # Customizing the labels on both axes

       labs(x = "\nRating Counts per Movie (Log-scaled with base 10)",
            y = "Average Movie Ratings per Movie\n(Stars)\n") +

       # Layout

       theme_bw(base_size = 14) +
       theme(axis.title.x = element_text(size = 14),
             axis.title.y = element_text(size = 14),
             plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Adding densigrams at the margins

ggMarginal(p_m, type = "densigram", bins = 100, fill = "navyblue",
                  color = "#280000", alpha = 0.3)

# Saving the output

ggsave("H1_movie_effects.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
       width = 21.33, height = 9, dpi = 320)
```

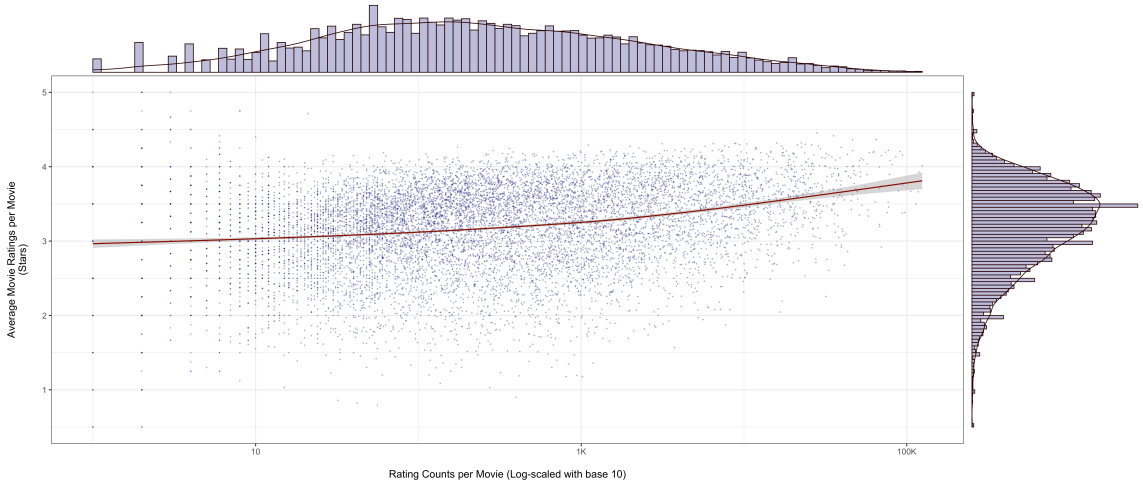

Figure 2: Semi-logarithmic scatterplot of average movie ratings (training dataset) against rating counts with fitted trend line and densigrams.

According to the visual hints in the graph above, we can state that hypothesis H_1 seems quite reasonable. Based on this graph, we have so far no reason to reject the hypothesis that higher rating counts tend to have a positive influence on higher average movie ratings. So we can include this hypothesized effect into our model at stake.

2.2 Effect of User Rating Counts

To visually check now our second hypothesis H_2, we graph the average movie ratings of the users against the counts of their given ratings.

```
# Visual exploration of H2

user_avg_rating <- edx_star %>%
                   group_by(userId) %>%
                   summarize(n = n(),
                             movie_rating = mean(rating))

p_u <- ggplot(user_avg_rating, aes(n, movie_rating)) +

       geom_point(color = "navyblue", size = 0.1, alpha = 0.3) +

       # Trend line

       geom_smooth(color = "darkred", size = 0.5) +

       # Label numbers with SI prefixes and log-scaling

       scale_x_continuous(labels = label_number_si(accurarcy = NULL),
                          trans = "log10") +

       # Customizing the labels on both axes

       labs(x = "\nRating Counts (Log-scaled with base 10)",
            y = "Average Movie Ratings per User\n(Stars)\n") +
```

```
        # Layout

        theme_bw(base_size = 14) +
        theme(axis.title.x = element_text(size = 14),
              axis.title.y = element_text(size = 14),
              plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Adding densigrams at the margins

ggMarginal(p_u, type = "densigram", bins =  100, fill = "navyblue",
            color = "#280000", alpha = 0.3)

# Saving the output

ggsave("H2_user_effects.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
       width = 21.33, height = 9, dpi = 320)
```

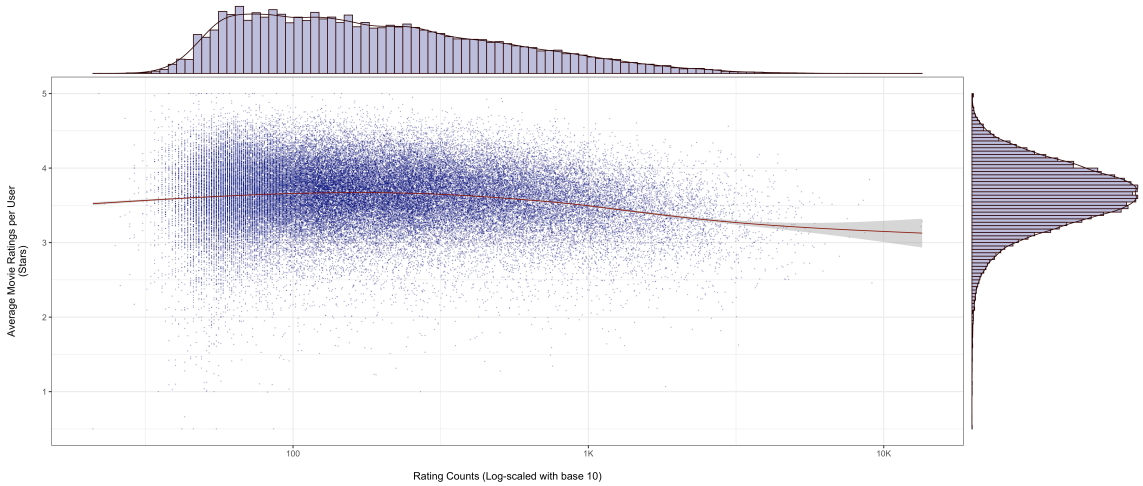

Figure 3: Semi-logarithmic scatterplot of average movie ratings by users (training dataset) against the counts of their given ratings with fitted trend line and densigrams.

As we see in the plot above, a decreasing trend can be observed. That is, higher rating counts per user tend to be negatively associated with lower average movie ratings. This visual finding seems, however, not that surprising when considering the possible underlying positive effect of high movie consumption on higher aesthetic judgment-standards. Above all, we should include this visually observable effect into our final model as it goes inline with our second hypothesis H_2. Furthermore, some interesting properties are here observable: While many users have rated approximately 50 up to 1000 movies (high density), two users have indeed rated more than 10,000 movies.

2.3 Effect of Movie Release Years

To visualize the assumption of our third hypothesis H_3, we will plot the given average movie ratings against the movie release years.

```
# Visual exploration of H3

release_year_avg_rating <- edx_star %>%
```

```
                    group_by(release_year) %>%
                    summarize(movie_rating = mean(rating))

p_y <- ggplot(release_year_avg_rating, aes(release_year, movie_rating)) +

       geom_point(color = "navyblue", alpha = 0.3 ) +

       # Trend line

       geom_smooth(color = "darkred", size = 0.3, alpha = 0.2) +

       # Label numbers with SI prefixes and log-scaling

       labs(x = "\nRelease Year",
            y = "Average Movie Ratings per Movie Release Year\n(Stars)\n") +

       # Layout

       theme_bw(base_size = 14) +
       theme(axis.title.x = element_text(size = 14),
             axis.title.y = element_text(size = 14),
             plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Saving the output

ggsave("H3_year_effects.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
       width = 21.33, height = 9, dpi = 320)
```

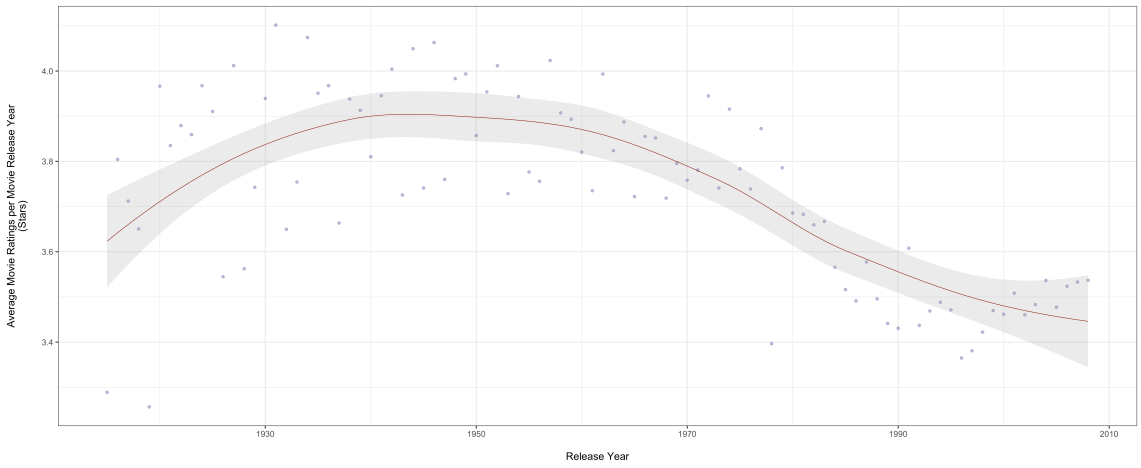

Figure 4: Scatterplot of average movie ratings by release year (training dataset) with fitted trend line.

In the plot above, we see a clear decreasing trend of the given average movie ratings: The later the rated movies have been released, the lower the average ratings turn out. This effect is expectable since the more recent the rated movies are, the lesser time is available to rate such new movies. The same holds for the rating counts of such new movies. Thus, to include this effect, which is stated in hypothesis H_3, into our model at stake, appears reasonable.

2.4 Effect of Rating Dates

Next, we will plot the given average movie ratings against the rating dates on which the users have rated the movies in question.

```
# Visual exploration of H4

rate_date_avg_rating <- edx_star %>%
                          group_by(rate_date) %>%
                          summarize(date_rating = mean(rating))

ggplot(data = rate_date_avg_rating ) +


geom_point(aes(rate_date, date_rating), color = "navyblue", size = 0.66,
           alpha = 0.3) +

# Trend line

geom_smooth(aes(rate_date, date_rating), color = "darkred", size = 0.3,
            alpha = 0.2) +

# Customizing the labels on the x-axis

scale_x_date(date_breaks = "1 year", date_labels = "%Y") +

# Customizing the labels on both axes

labs(x = "\nRating Date", y = "Average Movie Ratings per Date\n(Stars)\n") +

# Layout

theme_bw(base_size = 14) +
theme(axis.title.x = element_text(size = 14),
      axis.title.y = element_text(size = 14),
      plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Saving the output

ggsave("H4_date_effects.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
       width = 21.33, height = 9, dpi = 320)
```

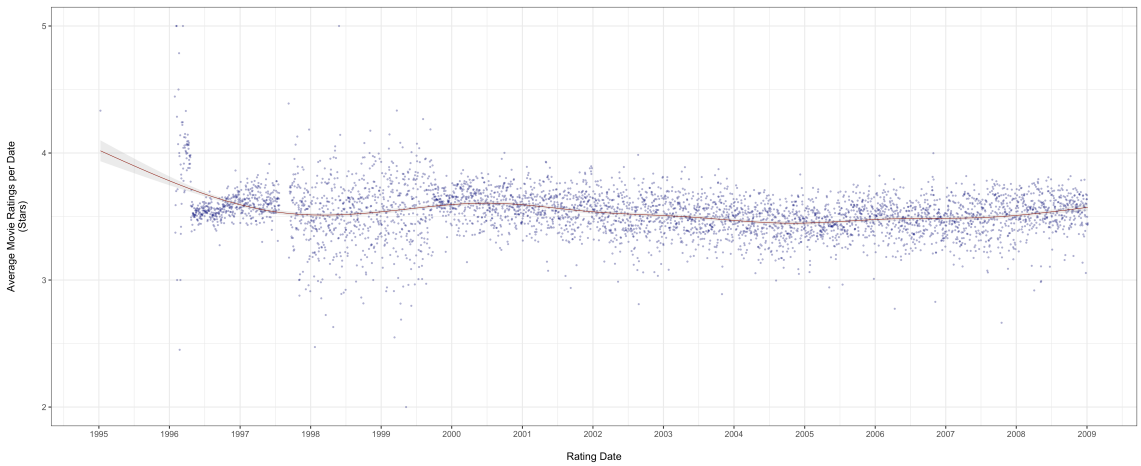

Figure 5: Scatterplot of average movie ratings by rating dates (training set) with fitted trend line.

Here we see a quite minimal decreasing trend of the average ratings per date with proceeding years or, respectively, rating dates. Approximately, between the last quarter of the year 1997 and the end of 1999, high variability in given ratings can be observed, while the variability in given ratings between the years 2000 and 2009 remains relatively constant. As a minimal trend is, after all, observable, we can and should take this effect also into account when fitting our model at stake. That is, we can retain the fourth hypothesis (H_4).

2.5 Effect of Genre-Label Rating Counts

Finally, we will graph the average movie ratings against the rating counts of genre labels.

```
# Visual exploration of H5

genre_avg_rating <- edx_star %>%
                    group_by(genres) %>%
                    summarize(counts = n(),
                              movie_rating = mean(rating))

p_g <- ggplot(data = genre_avg_rating, aes(counts, movie_rating)) +

       geom_point(color = "navyblue", alpha = 0.3) +

       # Trend line

       geom_smooth(color = "darkred", size = 0.3, alpha = 0.2) +

       # Label numbers with SI prefixes and square root-scaling

       scale_x_continuous(labels = label_number_si(accurarcy = NULL),
                          trans = "sqrt") +

       # Indicating each genre label with an annotation/label

       geom_label_repel(aes(label = genres), xlim = c(7, NA), ylim = c(0.5, 5),
                        point.padding = 0.1, min.segment.length = 0.1,
```

```
                        box.padding = 0.1, direction = "y", seed = 1) +

      # Customizing the labels on both axes

      labs(x = "\nRating Counts (Square root-scaling)",
           y = "Average Movie Ratings by Genre Rating Counts\n(Stars)\n") +

      # Layout

      theme_bw(base_size = 14) +
      theme(axis.title.x = element_text(size = 14),
            axis.title.y = element_text(size = 14),
           plot.margin = unit(c(.66, .33, .66, .33), "cm"))

# Saving the output

ggsave("H5_genre_effects.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
       width = 21.33, height = 9, dpi = 320)
```

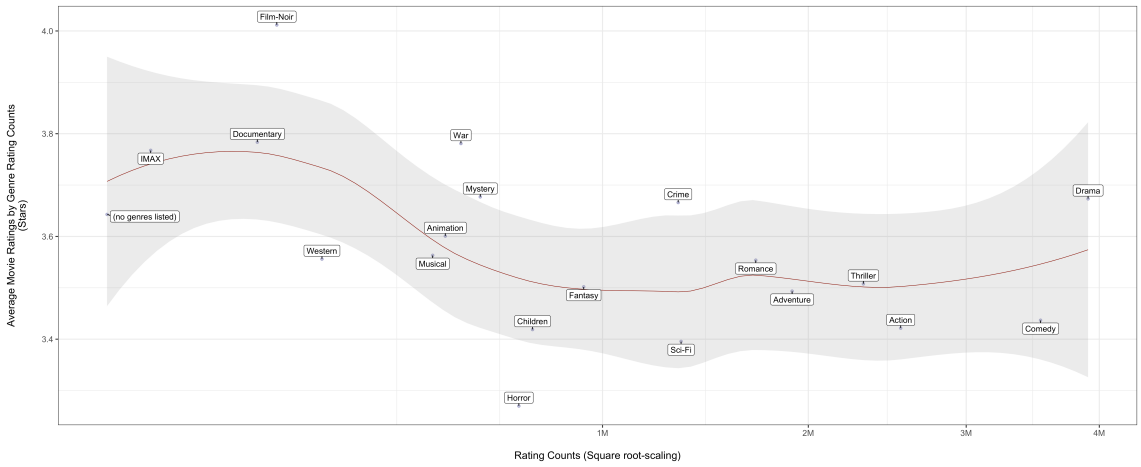

Figure 6: Semi-square root scaled scatterplot of average movie ratings by counted genre ratings (training dataset) with fitted trend line.

In Figure 6, we see a general decreasing trend: Higher genre rating counts tend to have a negative influence on the average movie ratings. That is, we can also add this last effect according to our hypothesis H_5 to our model. The more interesting aspect of this plot revolves, however, around the uncertainty or reliability of that high average ratings. In particular, we see that movies whose genre labels are not listed get relative high ratings, however, if we consider the total rating counts for this very label, we will find out that only 7 ratings belong to this label: That is, we should be careful to take this effect into account as such few ratings appear rather misleading, especially when it comes to predictions (see also Table 3; we will return to this issue and treat it in section 3.1).

Table 3: Average Ratings ordered by descending Genre Rating Counts

genres	counts	movie_rating
Musical	433080	3.563305
Western	189394	3.555918
Film-Noir	118541	4.011625
Documentary	93066	3.783487
IMAX	8181	3.767693
(no genres listed)	7	3.642857

So we see in the table above that the label "(no genres listed)" is indeed an exception since the next label "IMAX" already consists 8181 rating counts.

In the next central section, we will finally include these visually explored effects into our model to predict average movie ratings.

3 Analysis II and Methods: Predicting Movie Rating

When it comes to predicting values, one of the most intuitive ways for doing so might be to model and, respectively, fit linear relations. Accordingly, we will use in a first attempt a simple least square-regression within a machine learning framework to predict movie ratings. For doing so, we split the `edx_star` dataset in training and test datasets to assess our modeling strategy before we apply the final model to the validation dataset: the `valid_star` dataset. The following code lines will now split the `edx_star` dataset:

```
# Setting the seed to ensure reproducibility

set.seed(1, sample.kind = "Rounding")

#  Creating the index data partition to split the actual training dataset into
#  a sub-training dataset (90% of the actual training data) and, respectively,
#  a sub-test dataset (10% of the actual training dataset) for experimental/
#  cross-validation purposes.

CV_test_index <- createDataPartition(y = edx_star$rating, times = 1, p = .1,
                                     list = F)

# The sub-training dataset (90% of the actual training dataset).

CV_train_set <- edx_star[-CV_test_index,]

# The sub-test dataset (10% of the actual training dataset).

CV_test_set <- edx_star[CV_test_index,]

# Ensuring that in both sets the same movies and users are included.

CV_test_set <- CV_test_set %>%
               semi_join(CV_train_set, by = "userId") %>%
               semi_join(CV_train_set, by = "movieId")
```

After splitting the `edx_star` dataset into training and test datasets for experimentation and, respectively, cross-validation, we should define the loss or, respectively, cost function with which we can evaluate the performance of our model. The brief of this project is to underbid a loss with an $RMSE = 0.86490$, where the *R*esidual (or *R*oot) *M*ean *S*quare *E*rror is generally defined by

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where $\hat{y}$ stands, in our case, for the predicted ratings and y for the actual ratings, while n for the total number of these ratings, whereas i indicates an index. That is, we will square the residuals resulting from the actual and predicted ratings to weight higher prediction errors relatively high. We then take the mean of these squared residuals, and, finally, we will take from this very mean the square root to emphasize the squared proportional or, respectively, mean (vertical) deviation from the regression line. We can interpret this loss function straightforward similar to standard deviations according to which a zero loss ($RMSE = 0$) would indicate a perfect prediction. A loss of 1 ($RMSE = 1$) would pose, in this case, a deviation of 1 star. So, to put it the other way around: In achieving fewer prediction errors than 0.86490—as it is the challenge of this project—means that our rating predictions deviate (vertically) fewer than about 0.86490 star-units. Because of this, the $RMSE$, as loss function, can be interpreted in an analogous way to the standard deviation since the $RMSE$ is nothing else than the standard deviation of the unexplained variance and therefore of the difference between actual and predicted values (i.e., residuals) (see also Irizarry, 2020, and also the well-known "Netflix challenge" Koren, 2009).

```
# Loss function as the objective or, respectively, key performance indicator
# for the models at stake.

RMSE <- function(actual_ratings, predicted_ratings){
                sqrt(mean((actual_ratings - predicted_ratings)^2))}
```

Next, we will evaluate different models by using a least-square regression approach to hit this brief:

```
# Constant a.k.a y-intercept

mu <- mean(CV_train_set$rating)

# Naïve Rating Prediction

RMSE_0 <- RMSE(CV_test_set$rating, mu)
```

With this naïve mean rating (μ), we obtained the y-intercept or constant of the dependent variable since we ignore at this point every possible independent variable (theoretically, they all are set to zero) by assuming that every user would give the same rating to every movie. That is, the constant ensures that prospective predictors/coefficients for the final model (or more precisely, their residuals) are not systematically too high or too low. That is, we have to include this constant into our further models (see, e.g., Irizarry, 2020).

Next, we add as the first predictor the movie biases regarding the average ratings (see for this, in particular, H_1):

```
# Calculating the bias effect of movies (i.e, their specific given information)
# on the given ratings.

movie_avgs <- CV_train_set %>%
```

```
              group_by(movieId) %>%
              summarize(b_m = mean(rating - mu))

pred_ratings_1 <- CV_test_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  mutate(pred =  mu + b_m,
                  # constraining the prediction to the range between 0.5
                  # and 5 star units.
                  CV_test_pred = ifelse(pred < 0.5, 0.5,
                                        ifelse(pred > 5, 5, pred))) %>%
                  .$pred

RMSE_1 <- RMSE(CV_test_set$rating, pred_ratings_1)
```

This first model shows, as expected, an improvement regarding the prediction error compared to the naïve guess. That is, if we take movie effects into account, we reduce the error in predicting the actual movie ratings. Next, we will also include into this first model the specific biases or, respectively, effects of the users according to H_2 to reduce the prediction error one step further.

```
# Calculating the bias effect of users (i.e., their specific attitudes on the
# rated movies) on the given ratings.

user_avgs <- CV_train_set %>%
             left_join(movie_avgs, by = "movieId") %>%
             group_by(userId) %>%
             summarize(b_u = mean(rating - (mu + b_m)))


pred_ratings_2 <- CV_test_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  mutate(pred =  mu + b_m + b_u,
                  # constraining the prediction to the range between 0.5
                  # and 5 star units.
                         CV_test_pred = ifelse(pred < 0.5, 0.5,
                                               ifelse(pred > 5, 5, pred))) %>%
                  .$CV_test_pred


RMSE_2 <- RMSE(CV_test_set$rating, pred_ratings_2)
```

This second model shows also an improvement regarding the loss compared to the naïve guess and compared to the first model. In fact, we already have underbid the $RMSE = 0.86490$ with an $RMSE = 0.8570049$. However, as this is stepwise experimentation with the training and test datasets, we should try to reduce the prediction error as far as possible, before we apply the final model to the actual validation dataset—the `valid_star`-dataset.

After all, we can now state if we take movie- as well as user-effects into account, we reduce the error in predicting the movie ratings once more. So the strategy to include step by step predictors into our model works. Next, we will add to this second model the specific biases of the movie release year to reduce the error even more.

```
# Calculating the bias effect of the release years (i.e., their
# specific effects on the rated movies) on the given rating.

release_year_avgs <- CV_train_set %>%
                     left_join(movie_avgs, by = "movieId") %>%
                     left_join(user_avgs, by = "userId") %>%
                     group_by(release_year) %>%
                     summarize(b_y = mean(rating - (mu + b_m + b_u)))

pred_ratings_3 <- CV_test_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  left_join(release_year_avgs, by = "release_year") %>%
                  mutate(pred = mu + b_m + b_u + b_y,
                  # constraining the prediction to the range between 0.5
                  # and 5 star units.
                         CV_test_pred = ifelse(pred < 0.5, 0.5,
                                               ifelse(pred > 5, 5, pred))) %>%
                  .$CV_test_pred

RMSE_3 <- RMSE(CV_test_set$rating, pred_ratings_3)
```

Having added this predictor to the previous model, we could further reduce the error. That is, if we consider that the release year of a movie also has an effect on the average movie rating, we can predict the actual movie ratings even better. However, we could presumably make use of further information. In this vein, we will, therefore, include in this augmented model the specific biases of the rating dates on which the movies have been rated by the users, to reduce the error further.

```
# Calculating the bias effect of the rating_dates (i.e., their specific effects
# on the rated movies) on the given rating.

rate_date_avgs <- CV_train_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  left_join(release_year_avgs, by = "release_year") %>%
                  group_by(rate_date) %>%
                  summarize(b_d = mean(rating - (mu + b_m + b_u + b_y)))

pred_ratings_4 <- CV_test_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  left_join(release_year_avgs, by = "release_year") %>%
                  left_join(rate_date_avgs, by = "rate_date") %>%
                  mutate(pred =  mu + b_m + b_u + b_y + b_d,
                  # constraining the prediction to the range between 0.5
                  # and 5 star units.
                         CV_test_pred = ifelse(pred < 0.5, 0.5,
                                               ifelse(pred > 5, 5, pred))) %>%
                  .$CV_test_pred
```

```
RMSE_4 <- RMSE(CV_test_set$rating, pred_ratings_4)
```

This fourth model shows, as expected, further improvements regarding the loss compared to the previous models. That is, if we take movie- as well as user-effects, release years, and the rating dates into account, we reduce the error in predicting the movie ratings.

Finally, we will add to this last augmented model biases of genre labels based on the visual exploration of this effect (see Figure 6 and H_5).

```
# Calculating the bias effect of the genres (i.e., their specific framing
# on the rated movies) on the given rating.

genres_avgs <- CV_train_set %>%
               left_join(movie_avgs, by = "movieId") %>%
               left_join(user_avgs, by = "userId") %>%
               left_join(release_year_avgs, by = "release_year") %>%
               left_join(rate_date_avgs, by = "rate_date") %>%
               group_by(genres) %>%
               summarize(b_g = mean(rating - (mu + b_m  + b_u + b_y + b_d)))

# Finally, predicting the ratings of the sub-test set of the training
# dataset with each lambda based on the regularized input variables.

pred_ratings_5 <- CV_test_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  left_join(release_year_avgs, by = "release_year") %>%
                  left_join(rate_date_avgs, by = "rate_date") %>%
                  left_join(genres_avgs, by = "genres") %>%
                  mutate(pred = mu + b_m + b_u + b_y + b_d + b_g,
                  # constraining the prediction to the range between 0.5
                  # and 5 star units.
                         CV_test_pred = ifelse(pred < 0.5, 0.5,
                                               ifelse(pred > 5, 5, pred))) %>%
                  .$CV_test_pred

RMSE_5 <- RMSE(CV_test_set$rating, pred_ratings_5)
```

When taking now an overview of the different $RMSE$-values from the respective models, we see a promising improvement regarding the error reduction. Although the models 3 and 4 just slightly could improve the loss proportion, they are worthy enough to include them into the final model since they contribute to a lower loss when they are combined with the genre bias as it is the case in the last model.

Table 4: RMSE per Prediction Model

Model	RMSE
NULL) Naïve Rating	1.0517408
1) Movie Bias	0.9406850
2) Movie + User Biases	0.8566526
3) Movie + User + Release Year Biases	0.8563110
4) Movie + User + Release Year + Rating Date Biases	0.8555773
5) Movie + User + Release Year + Rating Date + Genre Biases	0.8554675

Now, as we have evaluated the explanatory power of these models based on predictions on the `edx_star` dataset, we could apply the last model (the fifth one) on the `valid_star` dataset, especially as we hit the brief to underbid the $RMSE = 0.86490$—we got an $RMSE_{Model5} = 0.8554675$. However, we can still improve our modeling strategy by taking the sample variability of the predictors into account. For doing so, we need to regularize and penalize misleading estimates of high average ratings from small rating counts and, therefore, the effects of these coefficients. We will treat this issue in the next section.

3.1 Finding the best Lambda, or: Regularizing and Penalizing

As already mentioned in section 2, some average ratings are relatively high, while the respective rating counts are low (see especially Table 3 and Figure 6). To treat this issue, we can make use of two intertwined concepts: regularization and penalizing. The general idea behind regularization is simple yet powerful: We compute the proportion of the sum of residuals (i.e., in this case, the difference between the ratings and the respective included effects/biases and, respectively, coefficients) to the total number of the data points to get the means of these residuals. This procedure is now still the same as we would do it with least-square regression—except for one crucial difference: Instead of dividing the sum of residuals through the number of data points, we add now the penalty term to the denominator before we divide the sum of residuals by the penalty term-adjusted or, therefore, the regularized number of data points. This may sound a bit complicated, but we have already treated this issue above just without the penalty term. At this point, we can also name this procedure like it is: By regularizing the sum of residuals as the proportion of the number of data points *and* the additional penalty term, we compute a so-called "Ridge Regression" (for a detailed introduction into regularized regressions with distinguishing between "LASSO" and "Ridge Regression" see especially: Gareth, Witten, Hastie, et al., 2015: pp. 215—226). That is, the penalty term has to be greater than zero, otherwise, we would still compute a least-squares regression.

Yet, one question arises: Which penalty term should we add to the number of data points? As the penalty term is indeed a tuning parameter, we can run cross-validation on the training and test datasets of the `edx-star` dataset to find the best penalty term, which we call lambda (λ), so that we can include this penalty term later into our final model to predict the actual ratings of the `valid_star` dataset.

Now, having sketched this optimization strategy, we can apply it on the data with the following lines of code:

```
# A sequence of values (lambdas) at an interval of 0.2 units as a set of
# regularization parameters in order to test/validate the optimal value that
# minimizes the value of the just introduced loss based on the sub-test
# set of the actual training dataset.
```

```r
lambdas <- seq(0, 10, .2)

# Cross validation of those lambdas by applying them each time on the
# respective predictor/input variable

rmse_validation <- sapply(lambdas, function(l){

# Constant a.k.a y-intercept

mu <- mean(CV_train_set$rating)

# Calculating the bias effect of movies (i.e, their specific given information)
# on the given rating by penalizing

movie_avgs <- CV_train_set %>%
              group_by(movieId) %>%
              summarize(b_m = sum(rating - mu) / (n() + l))

# Calculating the bias effect of users (i.e., their specific attitudes on the
# rated movies) on the given rating by

user_avgs <- CV_train_set %>%
             left_join(movie_avgs, by = "movieId") %>%
             group_by(userId) %>%
             summarize(b_u = sum(rating - (mu + b_m)) / (n() + l))

# Calculating the bias effect of the release years (i.e., their specific
# effects on the rated movies) on the given rating.

release_year_avgs <- CV_train_set %>%
                     left_join(movie_avgs, by = "movieId") %>%
                     left_join(user_avgs, by = "userId") %>%
                     group_by(release_year) %>%
                     summarize(b_y = sum(rating - (mu + b_m + b_u)) /
                                     (n() + l))

# Calculating the bias effect of the rating_dates (i.e., their specific effects
# on the rated movies) on the given rating.

rate_date_avgs <- CV_train_set %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  left_join(release_year_avgs, by = "release_year") %>%
                  group_by(rate_date) %>%
                  summarize(b_d = sum(rating - (mu + b_m + b_u + b_y)) /
                                      (n() + l))

# Calculating the bias effect of the genres (i.e., their specific framing
# on the rated movies) on the given rating.

genres_avgs <- CV_train_set %>%
```

```
                left_join(movie_avgs, by = "movieId") %>%
                left_join(user_avgs, by = "userId") %>%
                left_join(release_year_avgs, by = "release_year") %>%
                left_join(rate_date_avgs, by = "rate_date") %>%
                group_by(genres) %>%
                summarize(b_g = sum(rating - (mu + b_m + b_u + b_y + b_d)) /
                                  (n() + l))

# Finally, predicting the ratings of the sub-test set of the training
# dataset with each lambda based on the regularized input variables.

pred_ratings_CV <- CV_test_set %>%
                    left_join(movie_avgs, by = "movieId") %>%
                    left_join(user_avgs, by = "userId") %>%
                    left_join(release_year_avgs, by = "release_year") %>%
                    left_join(rate_date_avgs, by = "rate_date") %>%
                    left_join(genres_avgs, by = "genres") %>%
                    mutate(pred = mu + b_m + b_u + b_y + b_d + b_g,
                    # constraining the prediction to the range between 0.5
                    # and 5 star units.
                           CV_pred = ifelse(pred < 0.5, 0.5,
                                            ifelse(pred > 5, 5, pred))) %>%
                    .$CV_pred


# Getting the respective loss function values for each lambda. That is, we will
# get 51 RMSE values.

return(RMSE(CV_test_set$rating, pred_ratings_CV))})
```

Once this computational time-consuming code chunk is finally run, we can graph the results with a scatter plot as follows:

```
# Retrieving the best value according to the loss function.

ggplot() +

# Scatter plot

geom_point(aes(lambdas, rmse_validation), color = "navyblue", size = 2,
           alpha = 0.66) +

# Indicating the optimal lambda with an annotation/label

annotate(geom = "point", x = lambdas[which.min(rmse_validation)],
         y = 0.8553969, size = 3,  shape = 21, fill = "darkgreen") +

geom_label(aes(x = lambdas[which.min(rmse_validation)], y = 0.8554069,
               label = "Best  \n(RMSE = 0.8553969)"),
               color = "darkgreen", size = 3) +
```

```
# Customizing the breaks on the x-axis

scale_x_continuous(breaks = c(0, 2, lambdas[which.min(rmse_validation)], 4, 6,
                              8, 10),
                   labels = c("0", "2", "3.6","4", "6", "8", "10" )) +

# Customizing the labels on both axes

labs(x = "\n ", y = "RMSE\n") +

# Layout

theme_bw(base_size = 14) +
theme(axis.text.x = element_text(size = 14),
      axis.text.y = element_text(size = 14),
      plot.margin = unit(c(.66,.33,.66,.33), "cm"))

# Saving the output

ggsave("lambdas.png", path = "~/Desktop/R_Projects/MovieLens/Plots",
        width = 21.33, height = 9, dpi = 320)
```

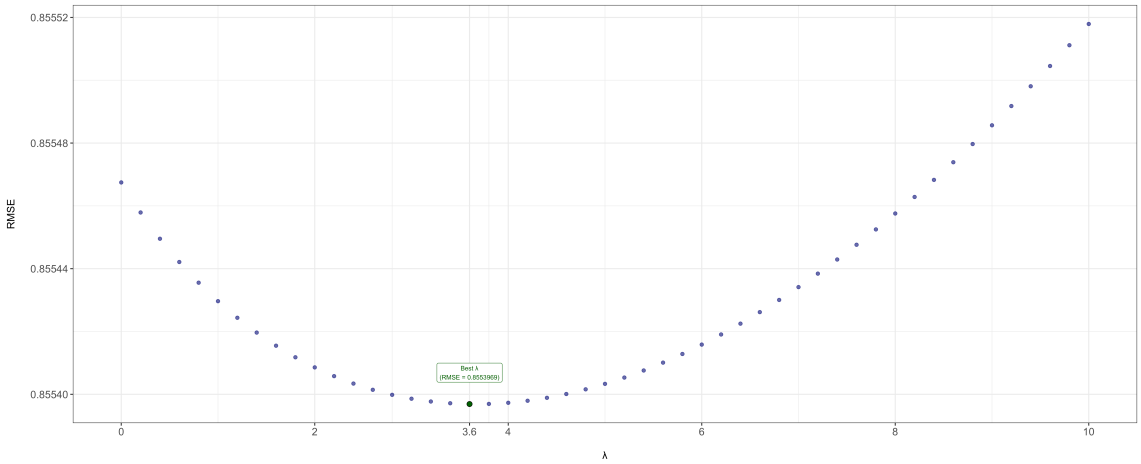

Figure 7: Cross-validation: RMSE-reducing Lambdas

As we see in the plot above, we achieved with a penalty term (i.e., $\lambda = 3.6$) fewer errors with an $RMSE = 0.8553969$ on the test dataset of the cross-validation dataset as without regularizing and penalizing the effects of the coefficients on the movie ratings (the previous unregularized and unpenalized prediction errors yielded an $RMSE = 0.8554675$). With this satisfying finding, we can finally apply this last model to the `valid_star` dataset to predict the actual movie ratings.

3.2 Predicting Final Movie Ratings by using Ridge Regression

Once we have evaluated our predictive model through simple stepwise least-squares regression and additionally by using cross-validation to regularize and penalize coefficients, we now move on and use the best λ in a Ridge Regression to predict the actual movie ratings of the `valid_star` dataset.

```r
# Getting the lambda value which minimizes the value of the loss.

l <- 3.6

## Running the Ridge Regression with the optimal lambda. Basically, the same
## lines of code as in the cross-validation, however, with the actual training
## and validation datasets and an optimal lambda-value.

# Constant a.k.a y-intercept

mu <- mean(edx_star$rating)

# Calculating the bias effect of movies (i.e, their specific given information)
# on the given rating

movie_avgs <- edx_star %>%
              group_by(movieId) %>%
              summarize(b_m = sum(rating - mu) / (n() + l))

# Calculating the bias effect of users (i.e., their specific attitudes on the
# rated movies) on the given rating.

user_avgs <- edx_star %>%
             left_join(movie_avgs, by = "movieId") %>%
             group_by(userId) %>%
             summarize(b_u = sum(rating - (mu + b_m)) / (n() + l))

# Calculating the bias effect of the release years (i.e., their specific
# effects on the rated movies) on the given rating.

release_year_avgs <- edx_star %>%
                     left_join(movie_avgs, by = "movieId") %>%
                     left_join(user_avgs, by = "userId") %>%
                     group_by(release_year) %>%
                     summarize(b_y = sum(rating - (mu + b_m + b_u)) /
                               (n() + l))

# Calculating the bias effect of the rating_dates (i.e., their specific effects
# on the rated movies) on the given rating.

rate_date_avgs <- edx_star %>%
                  left_join(movie_avgs, by = "movieId") %>%
                  left_join(user_avgs, by = "userId") %>%
                  left_join(release_year_avgs, by = "release_year") %>%
                  group_by(rate_date) %>%
                  summarize(b_d = sum(rating - (mu + b_m + b_u + b_y)) /
                                      (n() + l))

# Calculating the bias effect of the genres (i.e., their specific framing
# on the rated movies) on the given rating.
```

```
genres_avgs <- edx_star %>%
               left_join(movie_avgs, by = "movieId") %>%
               left_join(user_avgs, by = "userId") %>%
               left_join(release_year_avgs, by = "release_year") %>%
               left_join(rate_date_avgs, by = "rate_date") %>%
               group_by(genres) %>%
               summarize(b_g = sum(rating - (mu + b_m  + b_u + b_y + b_d)) /
                                   (n() + l))

# Finally, predicting the ratings of the sub-test set of the actual training
# dataset with each lambda based on the regularized input variables.

pred_ratings <- valid_star %>%
                left_join(movie_avgs, by = "movieId") %>%
                left_join(user_avgs, by = "userId") %>%
                left_join(release_year_avgs, by = "release_year") %>%
                left_join(rate_date_avgs, by = "rate_date") %>%
                left_join(genres_avgs, by = "genres") %>%
                mutate(pred = mu + b_m + b_u + b_y + b_d + b_g,
                # constraining the prediction to the range between 0.5
                # and 5 star units.
                       final_pred = ifelse(pred < 0.5, 0.5,
                                           ifelse(pred > 5, 5, pred))) %>%
                .$final_pred

# The moment of truth: Result of the final prediction

RMSE_FINAL <- RMSE(valid_star$rating, pred_ratings)
```

Once this final prediction procedure is completed, we can review our final result of our final model and compare it to its non-regularized training version:

Table 5: RMSE per Prediction Model

Model	RMSE
NULL) Naïve Rating	1.0517408
1) Movie Bias	0.9406850
2) Movie + User Biases	0.8566526
3) Movie + User + Release Year Biases	0.8563110
4) Movie + User + Release Year + Rating Date Biases	0.8555773
5) Movie + User + Release Year + Rating Date + Genre Biases	**0.8554675**
Final Prediction) Regularized Model No. 5	**0.8619019**

As we see, the final predictions consist not fewer errors compared to the simple least-squares- or cross-validation-setting (during the model developments or experimentation scenarios), nonetheless, the final predictions of the actual movie ratings are still useful as they indeed hit the brief in underbidding an $RMSE$ of 0.86490 for about 0.35%.

4. Results

So we achieved with our final model on the validation dataset `valid_star` a quite satisfying result regarding the movie rating predictions by underbidding the $RMSE = 0.86490$ with a *final* prediction error with an $RMSE = 0.861902$. That is, we can state at the ending of this report that our strategy paid off. However, it seems, above all, reasonable to consider other approaches to achieve an even better result—such as tree-based algorithms, support vector machines, or even neural networks—since these algorithms are well-known for their test power and robustness. On the other hand, as the computational costs of these algorithms are relatively high, we can stick with our result, at this stage, as a proof of concept.

5. Concluding Remarks

At the end of our project, we can now answer our initial research question as follows: Once the general data properties are explored and implemented in a predictive model, a Ridge Regression provides the desired result. That is, we achieve fewer prediction errors ($RMSE = 0.861902$) as it is the brief of this project ($RMSE \leq 0.86490$) as soon as we regularize and penalize the effects of the independent variables (coefficients), according to the introduced hypotheses, on the dependent variable.

References

Gareth, J., Witten, D., Hastie, T., Tibshirani, R. (2013). *An introduction to statistical learning. With Applications in R.*, *112*. New York: Springer.

Gorakala, S. & Usuelli, M. (2015). *Building a Recommendation System with R*. Birmingham: Packt.

Grouplens (2020). MovieLens 10M Dataset, (online). Retrieved August 24, 2020, from https://grouplens.org/datasets/movielens/10m/

Irizarry, R.A. (2020). *Introduction to Data Science*, (online). Retrieved August 24, 2020, from https://rafalab.github.io/dsbook/index.html

Koren, Y. (2009). The BellKor Solution to the Netflix Grand Prize, (online). Retrieved August 24, 2020, from https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf

Wickham, H. (2014). Tidy Data. In: *Journal of Statistical Software*, *59*(10), doi:http://dx.doi.org/10.18637/jss.v059.i10

Appendix — Environment

Overview of the used statistical computing environment and the used operating system:

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 4.0.2 (2020-06-22)
##  os       macOS Catalina 10.15.6
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  de_DE.UTF-8
##  ctype    de_DE.UTF-8
```

```
##  tz       Europe/Berlin
##  date     2020-08-25
##
## - Packages ------------------------------------------------------------------
##  package      * version    date       lib source
##  assertthat     0.2.1      2019-03-21 [1] CRAN (R 4.0.0)
##  backports      1.1.8      2020-06-17 [1] CRAN (R 4.0.0)
##  blob           1.2.1      2020-01-20 [1] CRAN (R 4.0.2)
##  broom          0.7.0      2020-07-09 [1] CRAN (R 4.0.2)
##  caret        * 6.0-86     2020-03-20 [1] CRAN (R 4.0.0)
##  cellranger     1.1.0      2016-07-27 [1] CRAN (R 4.0.0)
##  class          7.3-17     2020-04-26 [1] CRAN (R 4.0.2)
##  cli            2.0.2      2020-02-28 [1] CRAN (R 4.0.0)
##  codetools      0.2-16     2018-12-24 [1] CRAN (R 4.0.2)
##  colorspace     1.4-1      2019-03-18 [1] CRAN (R 4.0.0)
##  crayon         1.3.4      2017-09-16 [1] CRAN (R 4.0.0)
##  data.table   * 1.13.0     2020-07-24 [1] CRAN (R 4.0.2)
##  DBI            1.1.0      2019-12-15 [1] CRAN (R 4.0.0)
##  dbplyr         1.4.4      2020-05-27 [1] CRAN (R 4.0.2)
##  digest         0.6.25     2020-02-23 [1] CRAN (R 4.0.0)
##  dplyr        * 1.0.1      2020-07-31 [1] CRAN (R 4.0.2)
##  ellipsis       0.3.1      2020-05-15 [1] CRAN (R 4.0.0)
##  evaluate       0.14       2019-05-28 [1] CRAN (R 4.0.0)
##  fansi          0.4.1      2020-01-08 [1] CRAN (R 4.0.0)
##  farver         2.0.3      2020-01-16 [1] CRAN (R 4.0.0)
##  fastmap        1.0.1      2019-10-08 [1] CRAN (R 4.0.0)
##  forcats      * 0.5.0      2020-03-01 [1] CRAN (R 4.0.0)
##  foreach        1.5.0      2020-03-30 [1] CRAN (R 4.0.0)
##  fs             1.5.0      2020-07-31 [1] CRAN (R 4.0.2)
##  generics       0.0.2      2018-11-29 [1] CRAN (R 4.0.0)
##  ggExtra      * 0.9        2019-08-27 [1] CRAN (R 4.0.2)
##  ggplot2      * 3.3.2      2020-06-19 [1] CRAN (R 4.0.0)
##  ggrepel      * 0.8.2      2020-03-08 [1] CRAN (R 4.0.0)
##  glue           1.4.1      2020-05-13 [1] CRAN (R 4.0.0)
##  gower          0.2.2      2020-06-23 [1] CRAN (R 4.0.2)
##  gtable         0.3.0      2019-03-25 [1] CRAN (R 4.0.0)
##  haven          2.3.1      2020-06-01 [1] CRAN (R 4.0.2)
##  hms            0.5.3      2020-01-08 [1] CRAN (R 4.0.0)
##  htmltools      0.5.0      2020-06-16 [1] CRAN (R 4.0.2)
##  httpuv         1.5.4      2020-06-06 [1] CRAN (R 4.0.0)
##  httr           1.4.2      2020-07-20 [1] CRAN (R 4.0.2)
##  ipred          0.9-9      2019-04-28 [1] CRAN (R 4.0.0)
##  iterators      1.0.12     2019-07-26 [1] CRAN (R 4.0.0)
##  jsonlite       1.7.0      2020-06-25 [1] CRAN (R 4.0.2)
##  kableExtra   * 1.1.0      2019-03-16 [1] CRAN (R 4.0.2)
##  knitr        * 1.29       2020-06-23 [1] CRAN (R 4.0.2)
##  labeling       0.3        2014-08-23 [1] CRAN (R 4.0.0)
##  later          1.1.0.1    2020-06-05 [1] CRAN (R 4.0.0)
##  lattice      * 0.20-41    2020-04-02 [1] CRAN (R 4.0.2)
##  lava           1.6.7      2020-03-05 [1] CRAN (R 4.0.0)
##  lifecycle      0.2.0      2020-03-06 [1] CRAN (R 4.0.0)
```

```
##  lubridate    * 1.7.9      2020-06-08 [1] CRAN (R 4.0.2)
##  magrittr     * 1.5        2014-11-22 [1] CRAN (R 4.0.0)
##  MASS           7.3-51.6   2020-04-26 [1] CRAN (R 4.0.2)
##  Matrix         1.2-18     2019-11-27 [1] CRAN (R 4.0.2)
##  mgcv           1.8-31     2019-11-09 [1] CRAN (R 4.0.2)
##  mime           0.9        2020-02-04 [1] CRAN (R 4.0.0)
##  miniUI         0.1.1.1    2018-05-18 [1] CRAN (R 4.0.0)
##  ModelMetrics   1.2.2.2    2020-03-17 [1] CRAN (R 4.0.0)
##  modelr         0.1.8      2020-05-19 [1] CRAN (R 4.0.2)
##  munsell        0.5.0      2018-06-12 [1] CRAN (R 4.0.0)
##  nlme           3.1-148    2020-05-24 [1] CRAN (R 4.0.2)
##  nnet           7.3-14     2020-04-26 [1] CRAN (R 4.0.2)
##  pillar         1.4.6      2020-07-10 [1] CRAN (R 4.0.2)
##  pkgconfig      2.0.3      2019-09-22 [1] CRAN (R 4.0.0)
##  plyr           1.8.6      2020-03-03 [1] CRAN (R 4.0.0)
##  pROC           1.16.2     2020-03-19 [1] CRAN (R 4.0.0)
##  prodlim        2019.11.13 2019-11-17 [1] CRAN (R 4.0.0)
##  promises       1.1.1      2020-06-09 [1] CRAN (R 4.0.2)
##  purrr        * 0.3.4      2020-04-17 [1] CRAN (R 4.0.0)
##  R6             2.4.1      2019-11-12 [1] CRAN (R 4.0.0)
##  Rcpp           1.0.5      2020-07-06 [1] CRAN (R 4.0.2)
##  readr        * 1.3.1      2018-12-21 [1] CRAN (R 4.0.0)
##  readxl         1.3.1      2019-03-13 [1] CRAN (R 4.0.0)
##  recipes        0.1.13     2020-06-23 [1] CRAN (R 4.0.2)
##  reprex         0.3.0      2019-05-16 [1] CRAN (R 4.0.0)
##  reshape2       1.4.4      2020-04-09 [1] CRAN (R 4.0.0)
##  rlang          0.4.7      2020-07-09 [1] CRAN (R 4.0.2)
##  rmarkdown      2.3        2020-06-18 [1] CRAN (R 4.0.2)
##  rpart          4.1-15     2019-04-12 [1] CRAN (R 4.0.2)
##  rstudioapi     0.11       2020-02-07 [1] CRAN (R 4.0.0)
##  rvest          0.3.6      2020-07-25 [1] CRAN (R 4.0.2)
##  scales       * 1.1.1      2020-05-11 [1] CRAN (R 4.0.0)
##  sessioninfo    1.1.1      2018-11-05 [1] CRAN (R 4.0.0)
##  shiny          1.5.0      2020-06-23 [1] CRAN (R 4.0.2)
##  stringi        1.4.6      2020-02-17 [1] CRAN (R 4.0.0)
##  stringr      * 1.4.0      2019-02-10 [1] CRAN (R 4.0.0)
##  survival       3.1-12     2020-04-10 [1] CRAN (R 4.0.2)
##  tibble       * 3.0.3      2020-07-10 [1] CRAN (R 4.0.2)
##  tidyr        * 1.1.1      2020-07-31 [1] CRAN (R 4.0.2)
##  tidyselect     1.1.0      2020-05-11 [1] CRAN (R 4.0.0)
##  tidyverse    * 1.3.0      2019-11-21 [1] CRAN (R 4.0.0)
##  timeDate       3043.102   2018-02-21 [1] CRAN (R 4.0.0)
##  vctrs          0.3.2      2020-07-15 [1] CRAN (R 4.0.2)
##  viridisLite    0.3.0      2018-02-01 [1] CRAN (R 4.0.0)
##  webshot        0.5.2      2019-11-22 [1] CRAN (R 4.0.2)
##  withr          2.2.0      2020-04-20 [1] CRAN (R 4.0.0)
##  xfun           0.16       2020-07-24 [1] CRAN (R 4.0.2)
##  xml2           1.3.2      2020-04-23 [1] CRAN (R 4.0.0)
##  xtable         1.8-4      2019-04-21 [1] CRAN (R 4.0.0)
##  yaml           2.2.1      2020-02-01 [1] CRAN (R 4.0.0)
##
```

```
## [1] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```