Recommendation Systems: Rating Predictions with *MovieLens* Data

Danilo Ferreira de Oliveira

04/22/2021

Contents

1 Introduction

Companies like *Amazon*, *YouTube* or *Netflix* possess massive user ratings datasets, which they were able to collect from thousands of products sold to or used by their customers. Recommendation systems use those ratings to make user-specific recommendations. With regard to movies, we can predict what rating a user would give pictures they haven't seen by analyzing the ones he or she likes to watch, and then recommend films with relatively high predicted ratings. These are more complicated machine learning challenges because each outcome has a different set of predictors, seeing that different users rate different movies and different quantities of movies [Irizarry, 2021].

Netflix used to have a stars rating system (which now changed to a like-dislike system) and its recommendation algorithm would predict how many stars a particular user would give a specific movie. One star suggested the movie in question was not good, whereas five stars suggested it was an excellent one. In October 2006, *Netflix* proposed the data science community a challenge to improve their algorithm by 10%. Inspired by that challenge, we will develop the foundations to a movie recommendation system. Since *Netflix* data is not publicly available, we will use *GroupLens* research lab's database, which you can find here. We will be creating our own recommendation system using tools shown throughout all courses in the *HarvardX: Data Science Professional Certificate* series, including *R Language*, for statistical computing and its most famous IDE, *RStudio*.

There are various datasets available in the *MovieLens* website, but we will use the 10M version one to make computations a little easier. Released in January 2009, it is a stable benchmark dataset, and provides approximately 10 million ratings applied to near 10,000 movies by 72,000 users.

We will initially run a *R* script provided by *HarvardX* to generate our datasets. We continue by analyzing and visualizing the obtained training dataframe, and by developing rating prediction algorithms. With root mean squared error (RMSE) as our loss function, we select the best algorithm and, for its final test, we predict movie ratings in the validation set (the final hold-out test set) as if they were unknown. RMSE will again be used to evaluate how close predictions are to the true values in the final hold-out test set.

When training machine learning models, we will first follow the approach made by Irizarry [2021] and further develop it with the addition of genre effects. We will also train a matrix factorization model.

2 Analysis

2.1 Gathering and structuring datasets

Before we can start gathering our data and generating two different sets, we load the necessary R libraries for the whole report's development.

```
library(caret)
library(tidyverse)
library(ggplot2)
library(lubridate)
library(stringr)
library(recosystem)
library(data.table)
library(kableExtra)
library(tibble)
```

Next, we download the *MovieLens* data and run code provided by *edX* to generate our datasets for training and evaluating the final model. The `edx` set will be used all along the report to train and choose the best model for rating predictions, while `validation` will only be used in the end to evaluate the quality of the final model with a root mean square error function. The provided script begins by downloading files from the *GroupLens* database and performing transformations, in order to obtain the `movielens` dataframe below.

```
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")

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")
```

The validation set will be 10% of `movielens`. We need to make sure all user and movie's *IDs* in `validation` are also in `edx`, so we apply `semi_join()` transformations. Then, we add rows removed from the validation set back into the training set.

```
set.seed(1, sample.kind="Rounding")
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]

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

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

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

We can now start analyzing the `edx` dataframe.

2.2 Exploratory data analysis

To begin, let's take a look at our dataset's columns and variables.

```
edx %>% as_tibble()
```

```
>> # A tibble: 9,000,055 x 6
>>    userId movieId rating timestamp title                  genres
>>     <int>   <dbl>  <dbl>     <int> <chr>                  <chr>
>>  1      1     122      5 838985046 Boomerang (1992)       Comedy|Romance
>>  2      1     185      5 838983525 Net, The (1995)        Action|Crime|Thriller
>>  3      1     292      5 838983421 Outbreak (1995)        Action|Drama|Sci-Fi|T~
>>  4      1     316      5 838983392 Stargate (1994)        Action|Adventure|Sci-~
>>  5      1     329      5 838983392 Star Trek: Generation~ Action|Adventure|Dram~
```

```
>>  6      1     355      5 838984474 Flintstones, The (199~ Children|Comedy|Fanta~
>>  7      1     356      5 838983653 Forrest Gump (1994)    Comedy|Drama|Romance|~
>>  8      1     362      5 838984885 Jungle Book, The (199~ Adventure|Children|Ro~
>>  9      1     364      5 838983707 Lion King, The (1994)  Adventure|Animation|C~
>> 10      1     370      5 838984596 Naked Gun 33 1/3: The~ Action|Comedy
>> # ... with 9,000,045 more rows
```

We observed that columns `title` and `genres` are of class "character" (<chr>), while `movieId` and `rating` are numeric (<dbl>), `userId` and `timestamp` are integers (<int>). We should also analyze the number of unique values for each important feature column.

```
edx %>% summarize(n_userIds = n_distinct(userId), n_movieIds = n_distinct(movieId),
                  n_titles = n_distinct(title), n_genres = n_distinct(genres))
```

Table 1: Distinct values for userId, movieId, titles and genres

n_userIds	n_movieIds	n_titles	n_genres
69878	10677	10676	797

Note, from Table 1, that there are more unique `movieId` variables than there are `title` ones. We need to investigate why.

```
edx %>% group_by(title) %>%
  summarize(n_movieIds = n_distinct(movieId)) %>% filter(n_movieIds > 1)
```

Table 2: Movies with more than one ID

title	n_movieIds
War of the Worlds (2005)	2

Table 2 shows that *War of the Worlds (2005)* has two distinct `movieId` numbers. Let's see how many reviews each of them have.

Table 3: Different values of movieId and genres for *War of the Worlds (2005)*

movieId	title	genres	n
34048	War of the Worlds (2005)	Action|Adventure|Sci-Fi|Thriller	2460
64997	War of the Worlds (2005)	Action	28

Its second `movieId` received only 28 reviews (Table 3). This is a very low value, and since there is a possibility both IDs are in the validation set, we won't meddle with it.

2.2.1 Ratings

We can now visualize the data. First, we see how the ratings distribution is configured in Figure 1, and from it we can affirm that full star ratings are more common than those with half stars, since 4, 3 and 5 are the most frequent ones.

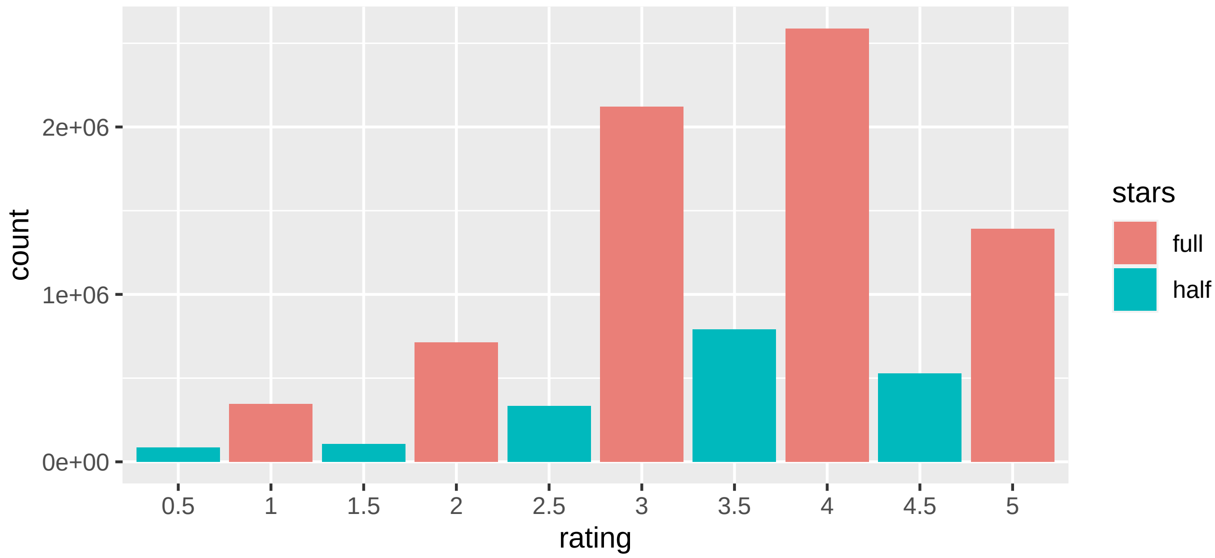

Figure 1: Ratings distribution

2.2.2 Movies and users

Our dataset has 10,676 different movies and 69,878 different users. By multiplying these numbers, we can verify that not every user rated every movie. We can think of the dataset as a very large matrix that has users as its row variables, movies as its columns and ratings (or `NA` values) within its cells. The task of a recommendation system can be regarded as filling in the empty values in this matrix. Figure 2 lets us evaluate the frequency distribution of reviews that each user gives and each movie receives by plotting these values.

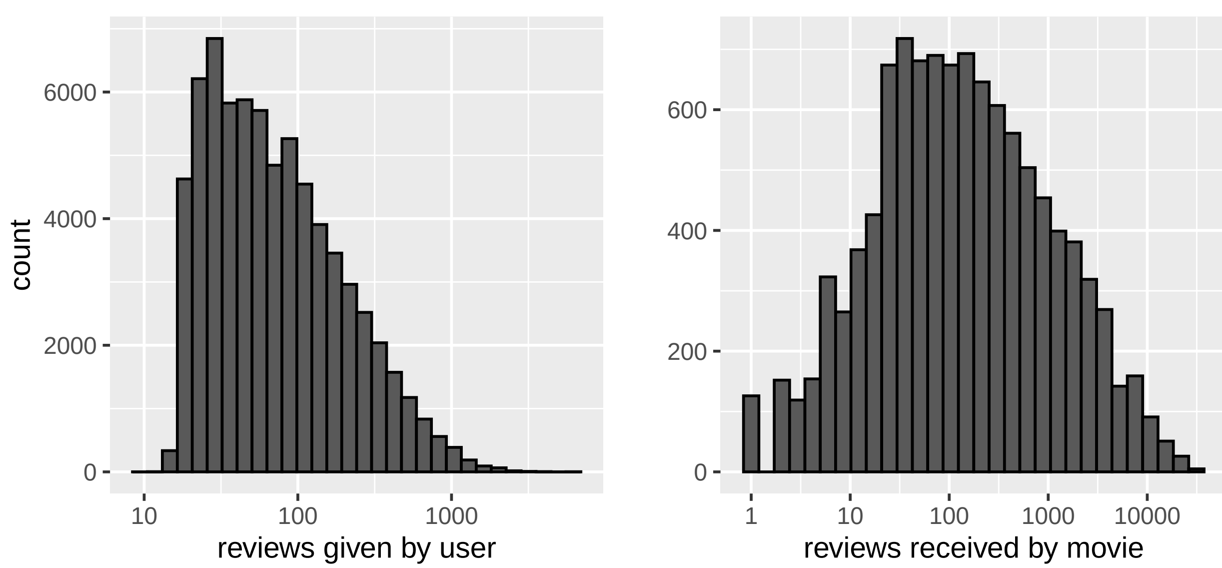

Figure 2: Frequency of reviews given by a specific user (left) and reviews received by a specific movie (right)

The x-axis in the generated plots from Figure 2 are both on a logarithmic scale, and the one on the left is right-skewed, meaning its average value is higher than its mode. By inspecting them, we can affirm that the majority of users give 20 to 200 reviews, while each movie frequently receives between 20 and a 1000 ratings, the latter being a pretty broad range.

Plotting histograms for average ratings, we get Figure 3. From the top plot, it is observable that it is rare for a user to average rating movies below 2.0, since there are few users to the left of that number (only 185 of all 69878 in the dataset, to be more exact). We can say the same about averaging over 4.5 (762 users). And from all 10,676 movies, we can also see that few of them average below 1.5 (only 61) and over 4.25 rating (57 films).

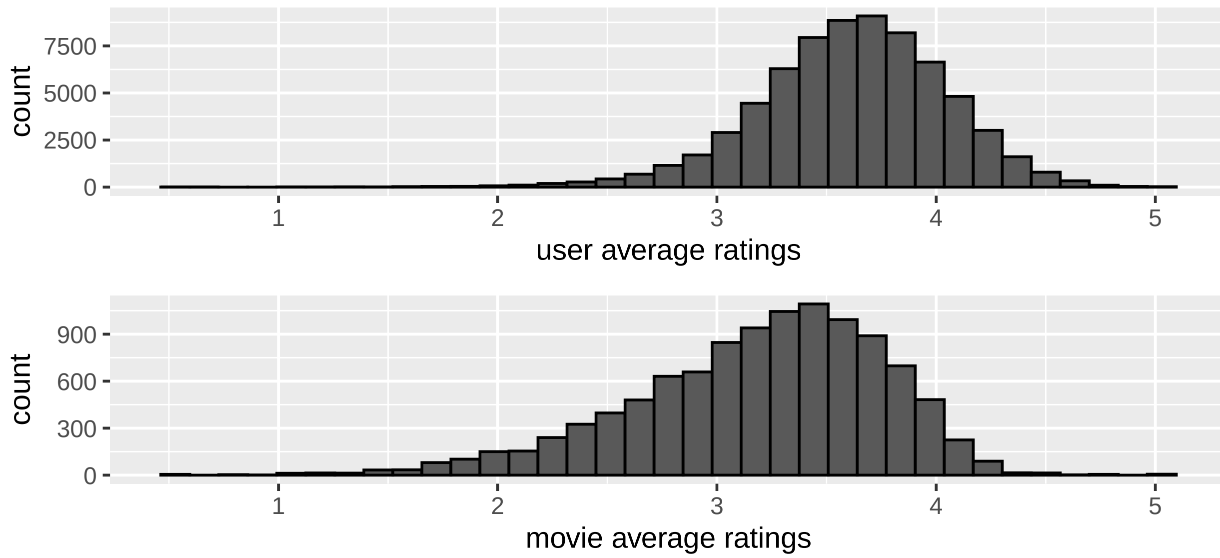

Figure 3: Frequency of average rating given by users (top) and average rating received by movies (bottom)

By exploring the top 10 most reviewed movies in the dataset (Fig. 4) and how many times they were reviewed, we identify that movies which receive the most reviews are normally blockbusters.

Checking for their average ratings in Figure 5, we also notice they are relatively high values, the lowest of them being 3.66. We can infer that the more a movie is reviewed, the better its rating is. It makes sense, given that the better the movie, the higher its ratings and the more people go watch it; subsequently, more users can rate it.

2.2.3 Release date

Since there isn't a column for movie's release date, we will check it by using the `str_extract` function from the `stringr` package, as the `title` column includes each movie's release date inside parenthesis.

```
release_date <- edx$title %>% str_extract('\\([0-9]{4}\\)') %>%
  str_extract('[0-9]{4}') %>% as.integer()
summary(release_date)
```

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>>    1915    1987    1994    1990    1998    2008
```

We first apply this function extracting the date along with the parenthesis, and then the number from inside them. We do it like this because, by extracting only the number, movies like *2001: A Space Odyssey (1968)* and *2001 Maniacs (2005)* show up when searching for year 2001. To go even further, it also extracts the numbers 1000 and 9000 for *House of 1000 Corpses (2003)* and *Detroit 9000 (1973)*, respectively.

As an example, we see in Table 4 that the earliest release date in our dataset (1915) only has one movie (and a rather controversial one):

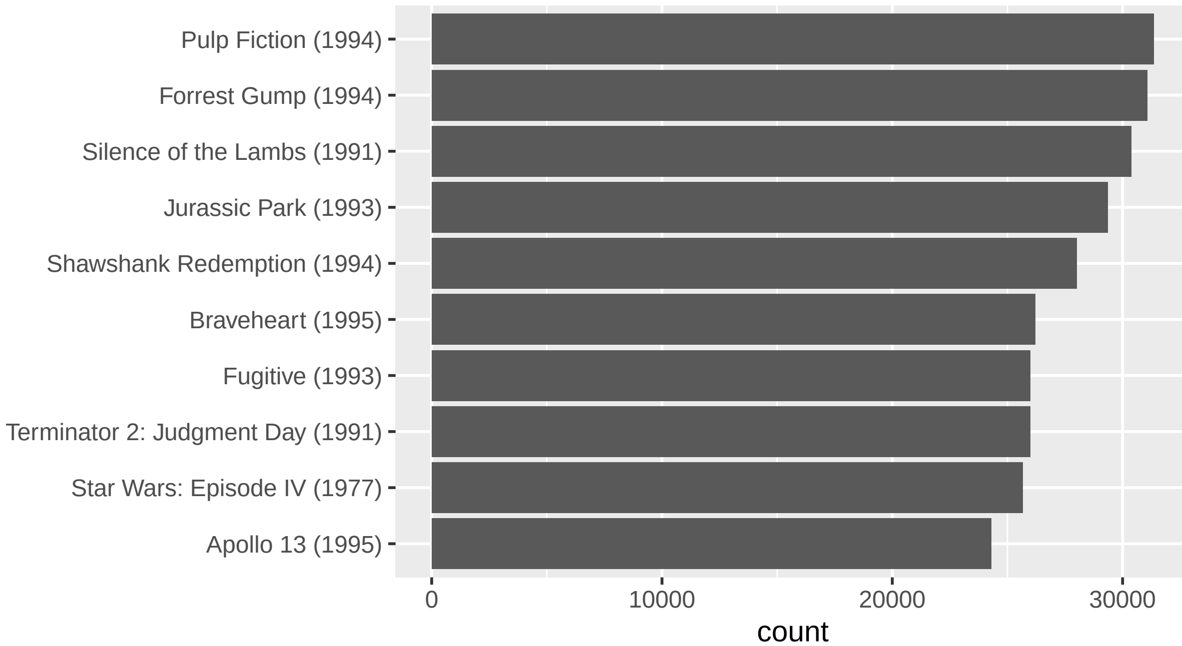

Figure 4: Number of reviews for the top 10 most reviewed movies

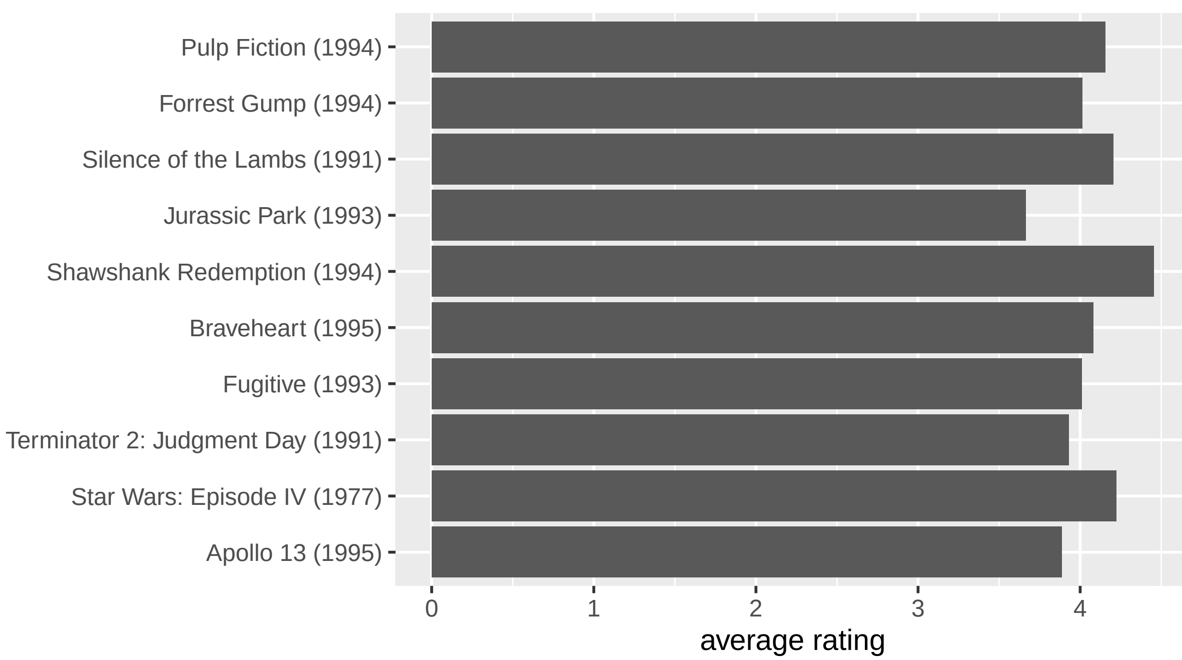

Figure 5: Average rating for the top 10 most reviewed movies

Table 4: Movie with the earliest release date in the dataset

movieId	title	genres	n
7065	Birth of a Nation, The (1915)	Drama|War	180

We now compute the number of ratings for each movie and plot it against the year the movie came out, using a boxplot for each year (Figure 6). A logarithmic transformation is applied on the y-axis (number of ratings) when creating the plot.

We see that, on average, movies that came out in the mid 90's get more ratings, reaching values over 30,000 reviews. We also see that with newer movies, starting near 1998, the number of ratings decreases with year: the more recent a movie is, the less time users have had to rate it.

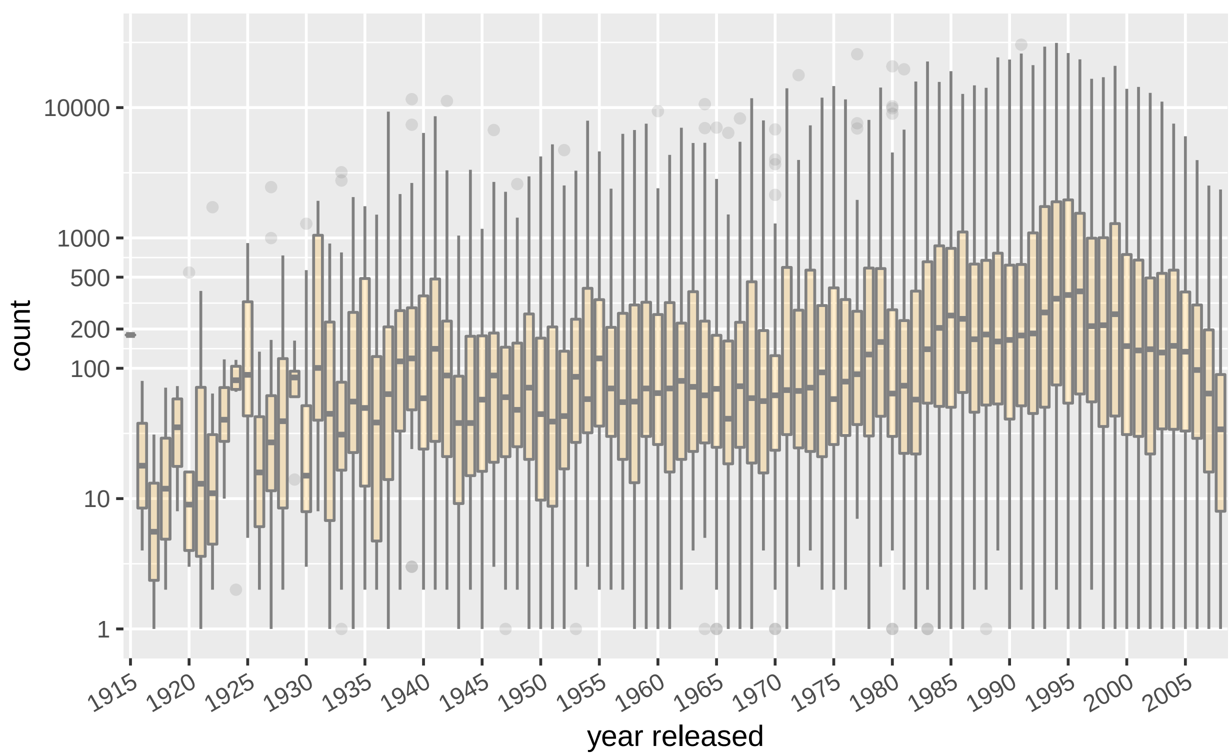

Figure 6: Number of reviews movies received against the year they were released

Similarly to before, we compute the average of ratings for each movie and plot it against the release year with boxplots, as shown in Figure 7. We can observe that most movies released until 1973 are normally spread between 3.0 and 4.0 ratings. From then on, the range gets bigger and values get lower (median values go from averaging over 3.5 to near 3.25). There are even movies from around the 2000's that average 0.5 ratings, but they are outliers and probably only received very few ratings.

2.2.4 Genres and genre combinations

The `genres` variable actually represents combinations of a series of unique genres. We will count the number of reviews each different combination that appear in our dataset has, and arrange them in descending order. Table 5 shows, as an example, the top 7 most reviewed genre combinations.

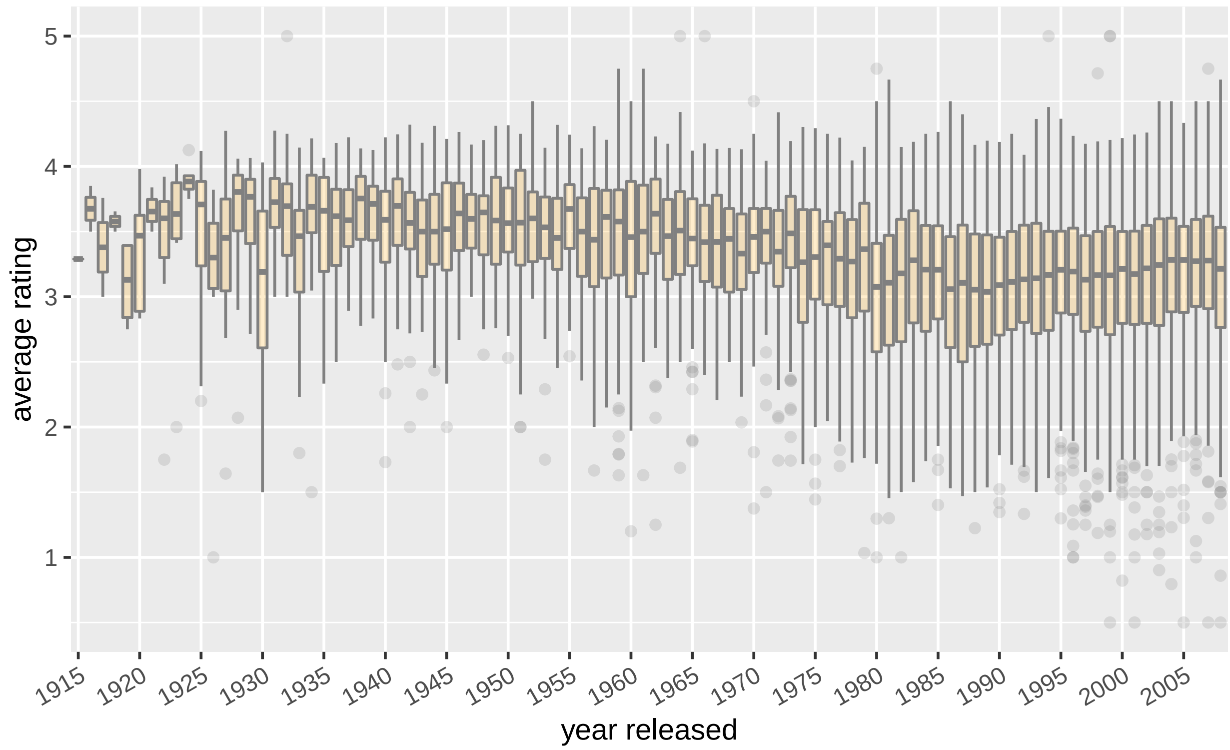

Figure 7: Average rating the movies received versus the year they were released

Table 5: Top genre combinations rated

rank	genres	count
1	Drama	733296
2	Comedy	700889
3	Comedy|Romance	365468
4	Comedy|Drama	323637
5	Comedy|Drama|Romance	261425
6	Drama|Romance	259355
7	Action|Adventure|Sci-Fi	219938

Let's check the genre effect on ratings. We will filter for genre combinations that were reviewed over a thousand times, put them in ascending order of average rating and visualize 15 examples, equally distant inside the arranged dataframe. The indexes to select combinations in the arranged set are represented by variable `gcomb_15`.

```
gcomb_15 <- c(1,seq(31,415,32),444)
edx %>% group_by(genres) %>%
  summarize(n = n(), avg = mean(rating), se = sd(rating)/sqrt(n())) %>%
  filter(n > 1000) %>% arrange(desc(avg)) %>%  .[gcomb_15,] %>%
  mutate(genres = reorder(genres, avg)) %>%
  ggplot(aes(x = genres, y = avg,
             ymin = avg - 2*se, ymax = avg + 2*se)) + geom_point() +
  geom_errorbar() + theme(axis.text.x=element_text(angle = 30, hjust = 1)) +
  labs(x='', y= 'average rating')
```

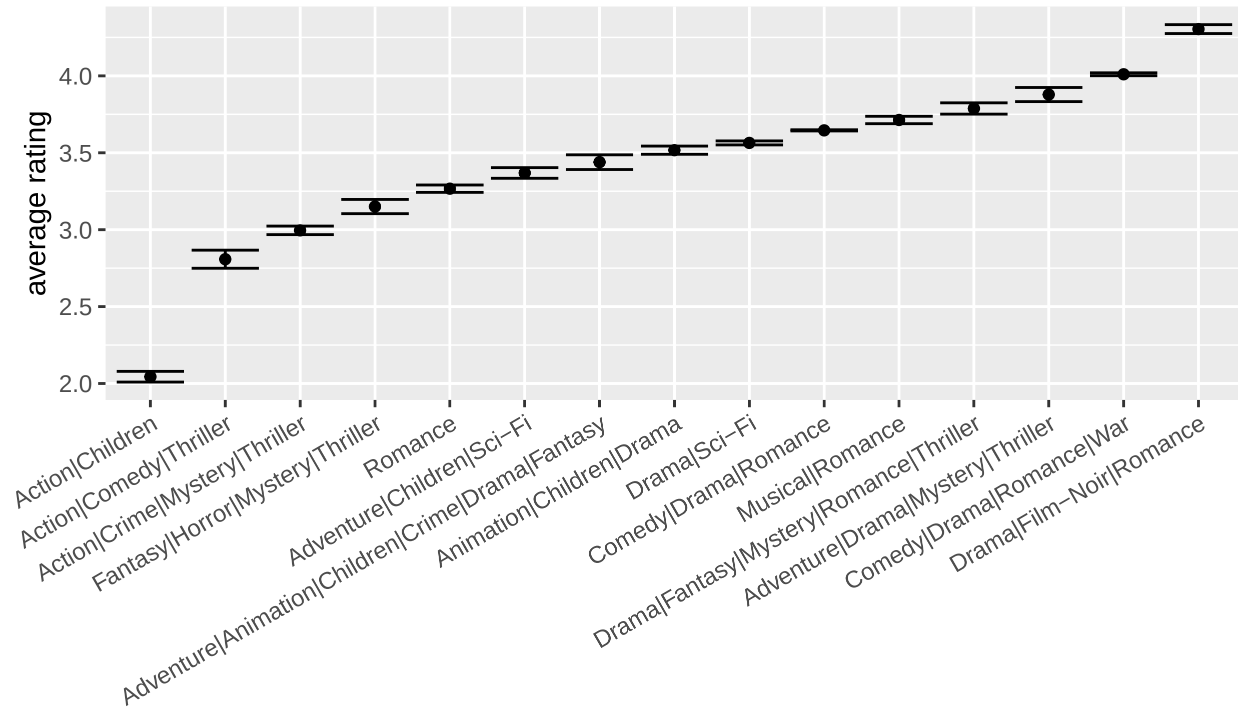

Figure 8: Distribution for selected genre combinations' average ratings

Inspecting Figure 8, we see the clear effect that genre combinations have on ratings, with average values ranging from approximately 2.0 to over 4.5. We can also investigate the effect of each genre that composes these combinations separately. There are 20 different genres in this dataset, and they are shown in Table 6:

Table 6: All unique genres present in the dataset

Comedy	Drama	War	Film-Noir
Romance	Sci-Fi	Animation	Horror
Action	Adventure	Musical	Documentary
Crime	Children	Western	IMAX
Thriller	Fantasy	Mystery	(no genres listed)

We can count how many reviews each genre have had by applying the `sapply()` function:

```
all_genres_count <- sapply(all_genres, function(g) {
  sum(str_detect(edx$genres, g))
}) %>% data.frame(review_count=.) %>%
  arrange(desc(.))
```

Results are displayed on Table 7. The number of reviews per genre range from near 4 million to only 7.

Table 7: Number of reviews per genre

all_genres_count[1:10]		all_genres_count[11:20]	
Genre	Count	Genre	Count
Drama	3910127	Horror	691485
Comedy	3540930	Mystery	568332
Action	2560545	War	511147
Thriller	2325899	Animation	467168
Adventure	1908892	Musical	433080
Romance	1712100	Western	189394
Sci-Fi	1341183	Film-Noir	118541
Crime	1327715	Documentary	93066
Fantasy	925637	IMAX	8181
Children	737994	(no genres listed)	7

We must investigate the least reviewed genre `(no genres listed)`, which actually represents no genre at all. We can see all movies that belong to it and their respective averages and standard errors:

```
edx[which(edx$genres=='(no genres listed)'),] %>%
  summarize(movieId=movieId[1], userId=userId[1], title=title[1], genres=genres[1],
            n = n(), avg = mean(rating), se = sd(rating)/sqrt(n()))
```

```
>>   movieId userId                title             genres n      avg        se
>> 1    8606   7701 Pull My Daisy (1958) (no genres listed) 7 3.642857 0.4185332
```

There is one movie in the dataset with no genres listed and it has only 7 reviews, giving its average rating a high standard error (0.4185). For this reason, we won't be plotting this genre's average rating along with the others in Figure 9. The forementioned plot shows us that average genre ratings vary from under 3.3 to over 4.0, clarifying even further the effect genres have on ratings.

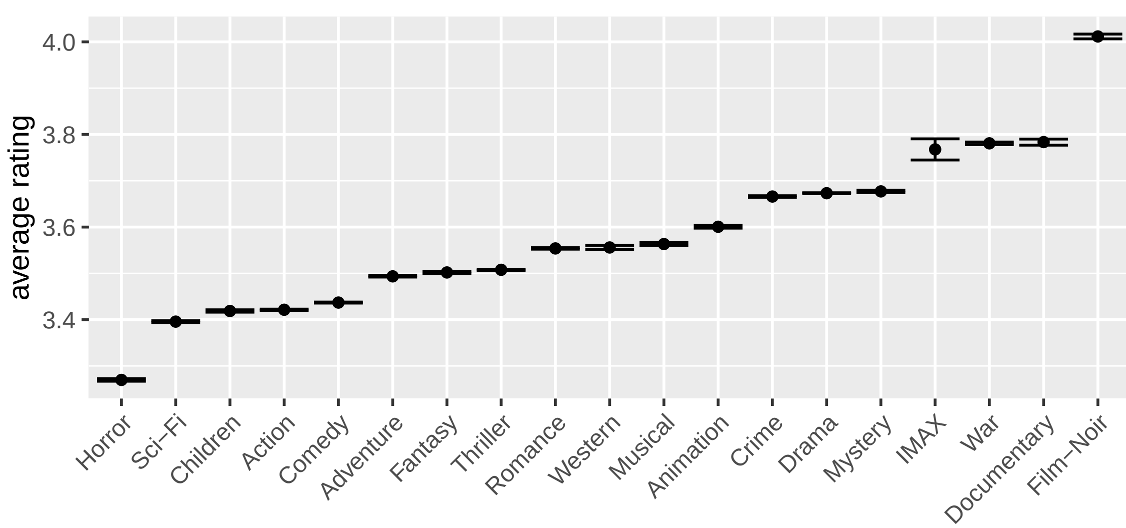

Figure 9: Average rating for each genre

We now observe the top genres reviewed in this dataset and how many reviews each one of them have. We accounted for genres that were reviewed over a million times, resulting in 8 distinct possibilities. The complete genre reviews count has already been shown in the form of Table 7, but we will plot the top 8 most reviewed genres (Fig. 10) to better visualize them and to also compare their shape with their respective movies count plot (Fig. 11).

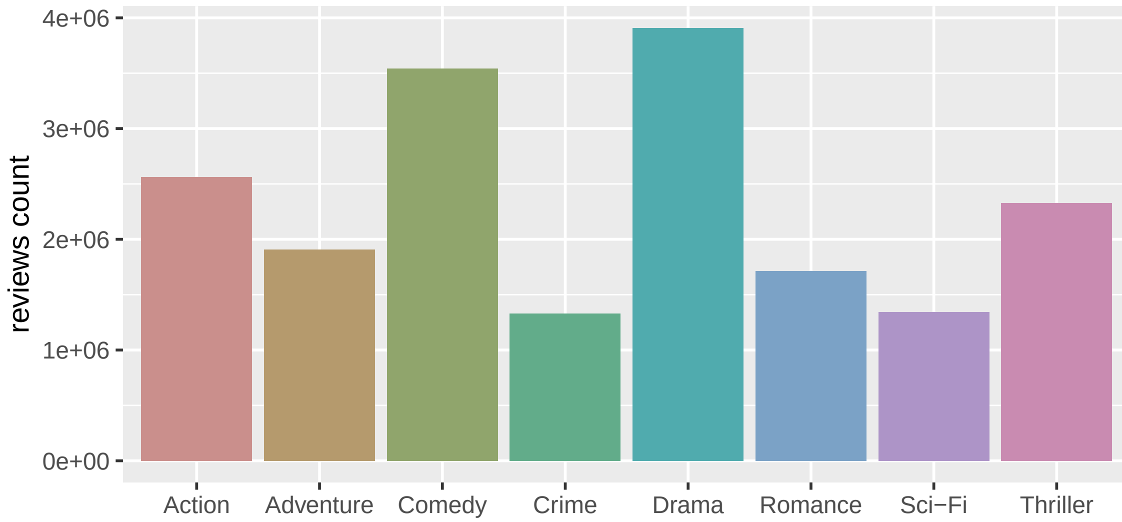

Figure 10: Number of reviews for each of the top 8 most reviewed genres

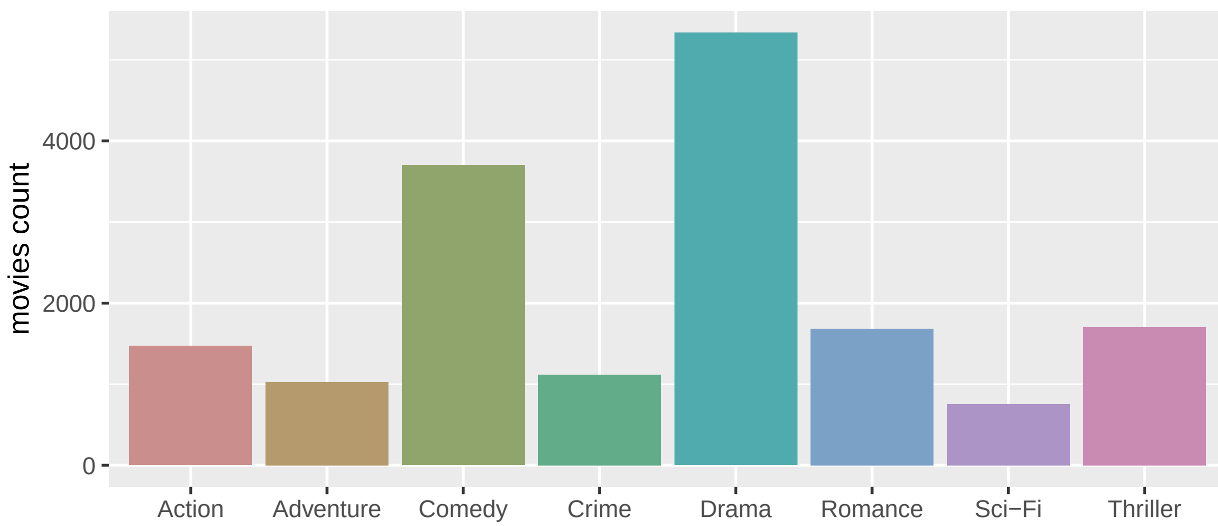

Figure 11: Number of movies that belong to the top 8 most reviewed genres

Generally, the barplot's shapes and proportions look much the same for both review (Fig. 10) and movie counts (Fig. 11), with the exception of *Action* having less movies than *Romance* and *Thriller*, even though it has more reviews. Both *Romance* and *Thriller* have very close proportions, but they change positions from one plot to the other. Meanwhile, *Comedy* and *Drama* dominate in both counts.

2.2.5 Rating date

Since the `timestamp` column doesn't provide us much value as it is, we create another column with the `as_datetime()` function that gives us the date and time a movie was rated.

```
edx <- mutate(edx, date = as_datetime(timestamp))
edx %>% as_tibble()
```

```
>> # A tibble: 9,000,055 x 7
>>    userId movieId rating timestamp title        genres       date
>>     <int>    <dbl>  <dbl>     <int> <chr>        <chr>        <dttm>
>>  1       1     122      5 838985046 Boomerang (~ Comedy|Roma~ 1996-08-02 11:24:06
>>  2       1     185      5 838983525 Net, The (1~ Action|Crim~ 1996-08-02 10:58:45
>>  3       1     292      5 838983421 Outbreak (1~ Action|Dram~ 1996-08-02 10:57:01
>>  4       1     316      5 838983392 Stargate (1~ Action|Adve~ 1996-08-02 10:56:32
>>  5       1     329      5 838983392 Star Trek: ~ Action|Adve~ 1996-08-02 10:56:32
>>  6       1     355      5 838984474 Flintstones~ Children|Co~ 1996-08-02 11:14:34
>>  7       1     356      5 838983653 Forrest Gum~ Comedy|Dram~ 1996-08-02 11:00:53
>>  8       1     362      5 838984885 Jungle Book~ Adventure|C~ 1996-08-02 11:21:25
>>  9       1     364      5 838983707 Lion King, ~ Adventure|A~ 1996-08-02 11:01:47
>> 10       1     370      5 838984596 Naked Gun 3~ Action|Come~ 1996-08-02 11:16:36
>> # ... with 9,000,045 more rows
```

The new column `date` is of a Date-Time class (`<dttm>`) called "POSIXct", and it makes it better to understand the time of rating. Now we can plot the time effect on the dataset, showing the possible influence time (in which a movie was rated) has on the average value. Along with average ratings from 1995 to 2009, Figure 12 presents a *LOESS* smooth line that better shows the relationship between variables.

```
edx %>% mutate(date = round_date(date, unit = "week")) %>%
  group_by(date) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(date, rating)) +
  geom_point() + labs(y='average rating') +
  geom_smooth()
```

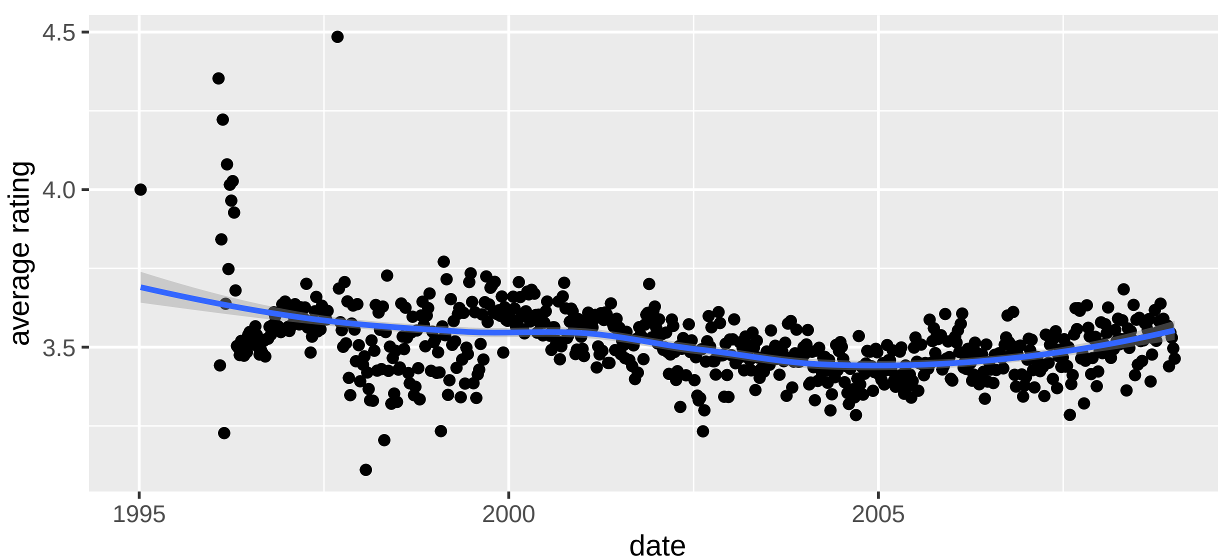

Figure 12: Review date effect on average rating

The review date definitely has some effect on ratings, but it isn't significant, so we will not be adding this effect in modeling.

2.3 Feature engineering

Down the line, we will need all genres available in the dataset as binary values, so it is necessary to create columns for all 20 of them. For each row, we attribute value 1 to new columns that are represented inside the `genres` cell, and value 0 to the rest. This means: if a determinate row has its `genres` value as `Comedy|Drama`, the newly created columns `Comedy` and `Drama` will have value 1, while the rest will remain zero. Here we see the dataframe's first 6 rows, with only `movieId`, `title` and `genres`, so we can compare genres further in this analysis.

```
edx[,c(2,5,6)] %>% as_tibble(.rows = 6)
```

```
>> # A tibble: 6 x 3
>>   movieId title                         genres
>>     <dbl> <chr>                         <chr>
>> 1     122 Boomerang (1992)              Comedy|Romance
>> 2     185 Net, The (1995)               Action|Crime|Thriller
>> 3     292 Outbreak (1995)               Action|Drama|Sci-Fi|Thriller
>> 4     316 Stargate (1994)               Action|Adventure|Sci-Fi
>> 5     329 Star Trek: Generations (1994) Action|Adventure|Drama|Sci-Fi
>> 6     355 Flintstones, The (1994)       Children|Comedy|Fantasy
```

We apply the transformation by doing the following `for()` command. We will also permanently remove columns such as `timestamp`, `title`, `genres` and `date`, since they won't be needed to train prediction models.

```
for (var in all_genres) {
  edx <- edx %>% mutate(genre=ifelse(str_detect(genres,as.character(var)),1,0))
  colnames(edx)[ncol(edx)] <- as.character(var)
}
rm(var)
edx <- edx[,-(4:7),drop=FALSE]
edx[,4:23] %>% as_tibble(.rows=6)
```

```
>> # A tibble: 6 x 20
>>   Comedy Romance Action Crime Thriller Drama `Sci-Fi` Adventure Children Fantasy
>>    <dbl>   <dbl>  <dbl> <dbl>    <dbl> <dbl>    <dbl>     <dbl>    <dbl>   <dbl>
>> 1      1       1      0     0        0     0        0         0        0       0
>> 2      0       0      1     1        1     0        0         0        0       0
>> 3      0       0      1     0        1     1        1         0        0       0
>> 4      0       0      1     0        0     0        1         1        0       0
>> 5      0       0      1     0        0     1        1         1        0       0
>> 6      1       0      0     0        0     0        0         0        1       1
>> # ... with 10 more variables: War <dbl>, Animation <dbl>, Musical <dbl>,
>> #   Western <dbl>, Mystery <dbl>, Film-Noir <dbl>, Horror <dbl>,
>> #   Documentary <dbl>, IMAX <dbl>, (no genres listed) <dbl>
```

Comparing each row in this tibble above with the ones demonstrated before, it is possible to see that the genres in the first one are now represented as 1's, while genres that didn't appear are kept as zero values. Therefore, the transformation occurred accurately.

2.4 Creating training and test sets

Before training any model, we need to create training and test sets. We first set the seed to guarantee model's reproducibility, and we divide the `edx` dataframe into 80-20% partitions.

```r
set.seed(123, sample.kind = "Rounding")
index <- createDataPartition(edx$rating, times=1, p=0.2, list= FALSE)
test <- edx[index,]
train <- edx[-index,]
```

To make sure all user and movie's *IDs* in `test` are also in `train`, we apply `semi_join()` functions, just as we did for the first two datasets in the beginning of the report.

```r
test <- test %>%
  semi_join(train, by = "movieId") %>%
  semi_join(train, by = "userId")
```

2.5 Prediction models

Now that we have our training and test sets, we can start evaluating prediction models. We will begin by following approaches proposed by Irizarry [2021], starting with a Overall Average Rating model, going through Movie, Movie + User and Regularized Movie + User effects algorithms. We will also apply a method that includes Genre effect and one that performs Matrix Factorization. To evaluate their quality, we will use the RMSE function.

If N is the number of user-movie combinations, $Y_{u,i}$ is the rating for movie i by user u, and $\hat{y}_{u,i}$ is our prediction, then RMSE is defined as follows:

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

which is represented by the function:

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

2.5.1 Overall Average Rating

We start with a model that assumes the same rating for all movies and all users, with all the differences explained by random variation: If μ represents the true rating for all movies and users and ε represents independent errors sampled from the same distribution centered at zero, then:

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

In this case, the least squares estimate of μ (the estimate that minimizes the root mean squared error) is the average rating of all movies across all users [Irizarry, 2021]. So we predict all ratings as the mean rating of the training set and compare it to the actual ratings in the test set (`RMSE(test$rating, mu)`). The calculated RMSE is shown in Table 8.

```r
mu <- mean(train$rating)
```

Table 8: RMSE result for Overal Average Rating Model

Method	RMSE
Just the Average	1.060013

We can observe that this value is actually the standard deviation of this distribution. Its value is too high, thus not acceptable.

$$\text{RMSE}_{\mu} = \sqrt{\frac{1}{N}\sum_{u,i}^{N}(\mu - Y_{u,i})^2} = \sigma$$

```
sd(train$rating)
```

```
>> [1] 1.060411
```

2.5.2 Movie effect model

We can improve our model by adding a term, b_i, that represents the effect of each movie's average rating:

$$Y_{u,i} = \mu + b_i + \varepsilon_{u,i}$$

Here, b_i is approximately the average of $Y_{u,i}$ minus the overall mean for each movie i. We can again use least squares to estimate b_i as follows:

```
fit <- lm(rating ~ as.factor(movieId), data=edx)
```

Because there are thousands of b's, the `lm()` function will be very slow or cause R to crash, so it is not recommendable using linear regression to calculate these effects [Irizarry, 2021]. We will then calculate all b_i's in the way shown below, followed by attributing each line in the test set with its corresponding b_i and adding μ to it, obtaining our estimates for each rating.

```
movie_avgs <- train %>% group_by(movieId) %>% summarize(b_i = mean(rating-mu))

predicted_ratings <- mu + test %>% left_join(movie_avgs, by='movieId') %>%
  pull(b_i)

summary(predicted_ratings)
```

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>>   0.500   3.217   3.586   3.512   3.878   4.750
```

From the summary above, we can see that predictions obtained range from 0.5 and 4.75, which are values inside the real range (0.5 to 5). Calculating `RMSE(test$rating, predicted_ratings)`, we get the results shown in Table 9.

Table 9: RMSE results with Movie effect model

Method	RMSE
Just the Average	1.0600131
Movie effect model	0.9436204

A 0.12 drop is a big improvement, but we can still do better.

2.5.3 Movie + User effects model

We further improve our model by adding b_u, the user-specific effect:

$$Y_{u,i} = \mu + b_i + b_u + \varepsilon_{u,i}$$

To fit this model, we could again calculate the least squares estimate by using `lm` [Irizarry, 2021]:

```
lm(rating ~ as.factor(movieId) + as.factor(userId))
```

But for the same reason as explained before, we will follow a simpler model approach where b_u is calculated with the difference between rating $Y_{u,i}$ and both overall mean rating and movie-specific coefficient. It is demonstrated in the code chunk below:

```
user_avgs <- train %>%  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%  summarize(b_u = mean(rating - mu - b_i))

predicted_ratings <- test %>%
  left_join(movie_avgs, by='movieId') %>%  left_join(user_avgs, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%  pull(pred)

summary(predicted_ratings)
```

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>> -0.7161  3.1353  3.5651  3.5127  3.9450  6.0547
```

Predicted ratings, obtained from the sum of μ to each test set line's corresponding coefficients, now have a spread of -0.7161 to 6.0547, which is bigger than the real spread. There are 208 values under 0.5 and 4149 over 5. But what we really need to evaluate is the RMSE value by applying its function.

Table 10: RMSE results with Movie + User effects model

Method	RMSE
Just the Average	1.0600131
Movie effect model	0.9436204
Movie + User effects model	0.8662966

Table 10 shows another big improvement in RMSE, with a 0.08 drop. Let's proceed improving this model to see how much better it can get.

2.5.4 Movie + User + Genre effects model

Now we try an approach proposed by Irizarry [2021], presented but not built up and developed within the book, that adds the genre effect to the model. Its formula is written as shown below:

$$Y_{u,i} = \mu + b_i + b_u + \sum_{k=1}^{K} x_{u,i}^k \beta_k + \varepsilon_{u,i}$$

Here $x_{u,i}^k$ is equal to 1 if $g_{u,i}$ is genre k. We already determined $x_{u,i}$ values earlier by creating binary columns for each of the 20 genres available, both in the training and test data. Now we determine β_k values with the same approach utilized in the last two models, where each β_k is the mean of the subtraction of overall average rating, movie-specific coefficient b_i and user-specific coefficient b_u from ratings $Y_{u,i}$. This way, if a movie belongs to both *Comedy* and *Drama*, one β will throw the average rating up (high ratings for *Drama*) and the other throws it down (low average for *Comedy*).

To make all process faster down the line, we will first add b_i and b_u columns to the training and test sets, and then we calculate all β_k's.

```
train <- train %>% left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId')
test <- test %>% left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId')
rm(user_avgs,movie_avgs)

beta_k <- vector()
for (i in seq_along(all_genres)) {
  b_value<- train %>%
    group_by(!!sym(all_genres[[i]])) %>%
    summarize(beta_k=mean(rating-mu-b_i-b_u)) %>%
    filter((.[1])==1) %>% .[[2]]
  beta_k <- append(beta_k, b_value)
}
rm(i,b_value)

df_beta<-data.frame(beta_k)
rownames(df_beta) <- all_genres
```

To determine these values within a `for()` loop, we first grouped the training set per specific genre column with the aid of argument `!!` and function `sym` (to unquote column names), forming two different groups (genre = 0 and genre = 1). We then obtained each specific genre's beta-value by filtering for parameters equal to 1 (the genre is present). We appended each `beta_k` to its vector and when all of them were obtained, we transformed the vector into the dataframe shown in Table 11.

Table 11: Genre-specific beta values

df_beta[1:10]		df_beta[11:20]	
genre	beta_k	genre	beta_k
Comedy	-0.0022244	War	0.0017305
Romance	-0.0035617	Animation	-0.0153201
Action	-0.0126152	Musical	-0.0102387
Crime	0.0079932	Western	-0.0066510
Thriller	-0.0047380	Mystery	0.0138908
Drama	0.0108468	Film-Noir	0.0306395
Sci-Fi	-0.0119931	Horror	0.0066441
Adventure	-0.0148381	Documentary	0.0624559
Children	-0.0241908	IMAX	-0.0016830
Fantasy	-0.0055682	(no genres listed)	0.1164495

The summation within this model's prediction equation is represented below:

$$\sum_{k=1}^{K} x_{u,i}\beta_k = x_{u,i}^{\{1\}}\beta_1 + x_{u,i}^{\{2\}}\beta_2 + ... + x_{u,i}^{\{20\}}\beta_{20}$$

We calculate these sums by performing a matrix multiplication between the test set genre columns (`test[,4:23]`) and all 20 β_k values calculated. This multiplication between a (1799965, 20) matrix and a (20, 1) vector results in a (1799965, 1) vector. We then add this as a column to the test set, so we can calculate predictions.

```
sum_x_beta <- as.matrix(test[,4:23])%*%as.matrix(df_beta)
sum_x_beta <- data.frame(sum_x_beta=sum_x_beta[,1])

test <- data.frame(test, sum_x_beta)
rm(df_beta,beta_k,sum_x_beta,all_genres)

predicted_ratings <- test %>%
  mutate(pred = mu + b_i + b_u + sum_x_beta) %>%
  pull(pred)

summary(predicted_ratings)
```

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>> -0.7183  3.1260  3.5581  3.5058  3.9406  6.0652
```

Analyzing the values obtained for ratings, we see there are 216 values under 0.5 and 4157 predicted ratings over 5.

Table 12: RMSE results with Movie + User + Genre effects model

Method	RMSE
Just the Average	1.0600131
Movie effect model	0.9436204
Movie + User effects model	0.8662966
Movie + User + Genre effects model	0.8661932

While the work needed to obtain this prediction model was arduous, there was very little improvement from the previous model (as seen in Table 12), so we will try adding regularization.

2.5.5 Regularized Movie + User effects model

When a movie receives ratings from just a few users, or users give very few reviews overall, we have more uncertainty. Therefore, larger estimates of both b_i and b_u, negative or positive, are more likely [Irizarry, 2021]. Consider a case in which we have a movie with 500 user ratings and another film with just one. While the first movie's b_i can be pretty accurate, the estimate for the second movie will simply be the observed deviation from the average rating, which is a clear sign of overtraining. In this cases, it is better to predict it as just the overall average rating μ, hence the need of some form of model penalization. Regularization permits us to penalize these large estimates that are formed using small sample sizes.

We will add regularization to one of the previously tried models. Since adding genre didn't have much effect on RMSE and both optimal λ determination (which we'll perform in this section) and β_k calculations are computationally costly, we will use the Movie + User effects model. So now, instead of minimizing the least squares equation, we minimize an equation that adds a penalty. We need to apply regularization to both user and movie effects, and we do it as follows:

$$\sum_{u,i} \left(y_{u,i} - \mu - b_i - b_u\right)^2 + \lambda \left(\sum_i b_i^2 + \sum_u b_u^2\right)$$

The formula below shows the new form of calculating b_i, with the inclusion of λ in the mean values calculation. When our sample size n_i is very large, then the regularization cost λ is effectively ignored since $n_i + \lambda \approx n_i$ [Irizarry, 2021]. However, when n_i is small, then the estimate $\hat{b}_i(\lambda)$ is shrunken towards 0. The larger λ, the smaller b_i.

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

In the same way, $\hat{b}_u(\lambda)$ is the regularized b_u, which includes λ in the division denominator.

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

We first calculate various sets of predictions and their respective RMSE values by testing lambdas from 0 to 10, with a 0.25 step-size, and then determine which regularization parameter gives us the lowest RMSE.

```
train <- train %>% select(userId, movieId, rating)
test <- test %>% select(userId, movieId, rating)

lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
  b_i <- train %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))
  b_u <- train %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))
  predicted_ratings <-
    test %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
    pull(pred)
  return(RMSE(test$rating, predicted_ratings))
})
qplot(lambdas, rmses, xlab='lambda', ylab='RMSE')
```

By analyzing the plot (Fig. 13) and retrieving values from tables `rmses` and `lambdas`, we identify that the optimal λ value is 5, and its corresponding RMSE is 0.8655751 (Table 13).

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>> -0.5784  3.1407  3.5624  3.5100  3.9346  5.9789
```

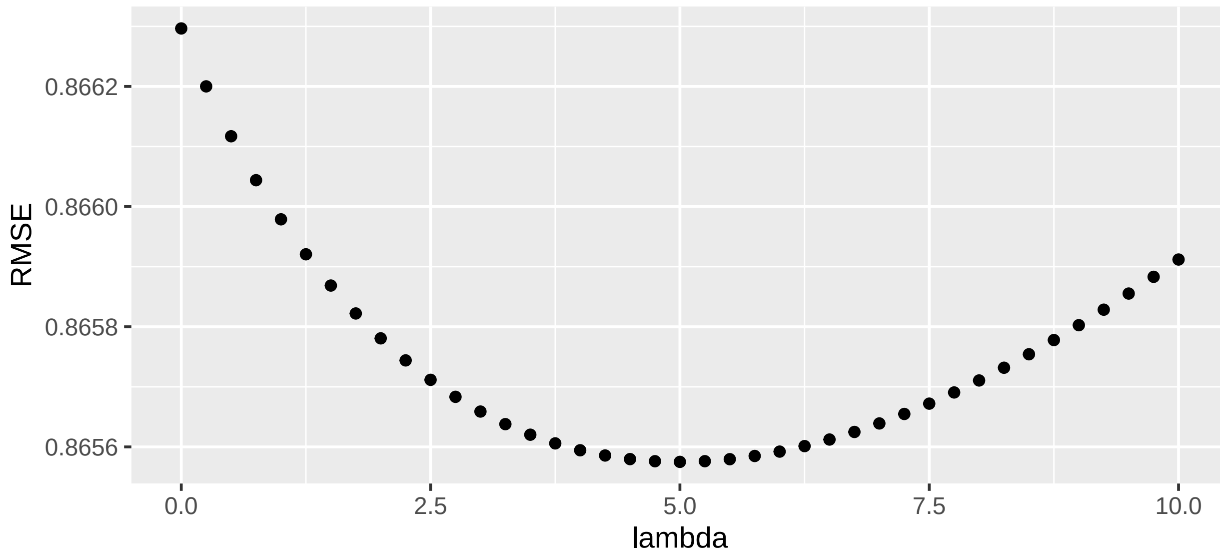

Figure 13: RMSE values per lambda

As usual, the `summary` function shows us that there are predicted values under 0.5 (137 predictions) and over 5.0 (2671).

Table 13: RMSE results with Regularized Movie + User effects model

Method	RMSE
Just the Average	1.0600131
Movie effect model	0.9436204
Movie + User effects model	0.8662966
Movie + User + Genre effects model	0.8661932
Regularized Movie + User effects model	0.8655751

Once more there is improvement, but not much. The approach we are taking seems to begin stagnating, so we try a different one: matrix factorization.

2.5.6 Matrix Factorization with ***recosystem*** R package

Some groups of movies, as well as some groups of users, have similar rating patterns. We can identify them by inspecting their correlations, which is basically what matrix factorization does. This family of methods, which is a class of collaborative filtering algorithms used in recommendation systems that became widely known during the *Netflix* challenge, works by decomposing the (incomplete) user-movie interaction matrix into the product of two lower dimensionality rectangular matrices [Koren, 2009b]. This incomplete matrix $R_{m \times n}$, as mentioned in Section 2.2.2, is composed by users on the rows and movies on the columns, with each cell being an attributed rating (or the absence of one).

So we approximate the whole rating matrix $R_{m \times n}$ by the product of two matrices of lower dimensions, $P_{k \times m}$ and $Q_{k \times n}$, as described above and written below [Qiu, 2021].

$$R \approx P'Q$$

Let p_u be the u-th column of P, and q_i be the i-th column of Q, then the rating given by user u on item i would be predicted as $p'_u q_i$. A typical solution for P and Q is the following [Chin et al., 2015b,a]:

$$\min_{P,Q} \sum_{(u,i)\in R} [f(p_u, q_i; r_{u,i}) + \mu_P ||p_u||_1 + \mu_Q ||q_i||_1 + \frac{\lambda_P}{2} ||pu||_2^2 + \frac{\lambda_Q}{2} ||q_i||_2^2]$$

where (u, i) are locations of observed entries in R, $r_{u,i}$ is the observed rating, f is the loss function, and μ_P, μ_Q, λ_P and λ_Q are, respectively, pairs of L1 and L2 regularization costs for both P and Q during gradient descent, to avoid overtraining.

The *recosystem* package easily performs this factorization. But to do it, we need to input a parse matrix in triplet form, meaning that each line in the file must contain three numbers: `user_index`, `item_index` and `rating` [Qiu, 2021]. User and item indexes may start with either 0 or 1, and this can be specified by the `index1` parameter in `data_memory()`. To train and predict, we need to provide objects of class `DataSource`, such as the previously mentioned `data_memory()`, which saves datasets as R objects.

After determining the training and test sets, we define an R object with function `Reco()` and tune it to identify the best parameters possible for our dataset. We need to determine the number of latent factors (k rows for $P_{k\times m}$ and $Q_{k\times n}$) with parameter `dim`, L2 regularization costs with `costp_l2` and `costq_l2`, and learning rate for gradient descent `lrate`. The `loss` parameter has a default value of `'l2'`, so we maintain it and keep L1 costs as zero.

```
train_dm <- data_memory(user_index = train$userId, item_index = train$movieId,
                        rating = train$rating, index1 = TRUE)
rm(train)

test_dm <- data_memory(user_index = test$userId, item_index = test$movieId,
                       index1 = TRUE)
test_rating <- test$rating
rm(test)

set.seed(123, sample.kind = "Rounding")
r <- Reco()
params = r$tune(train_dm, opts = list(dim = c(15, 20),
                                      costp_l1 = 0,
                                      costp_l2 = c(0.01, 0.1),
                                      costq_l1 = 0,
                                      costq_l2 = c(0.01, 0.1),
                                      lrate = c(0.075, 0.1), nthread = 2))

optimal_params = params$min

r$train(train_dm, opts = c(optimal_params, nthread = 1, niter = 20))
```

From tuning, we obtained optimal parameters `dim` = 20, `costp_l2` = 0.01, `costql2`=0.1 and `lrate` = 0.1. The model was then trained with these values, and it is now possible to predict ratings for the test set with `r$predict()`.

```
predicted_ratings = r$predict(test_dm, out_memory())

summary(predicted_ratings)
```

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>>  -1.594   3.057   3.543   3.472   3.963   6.300
```

Our predicted ratings are obtained with an R object of class `Output`, called `out_memory()`, and is saved as `predicted_ratings`. Inspecting the predictions obtained, we observe there are 859 values under 0.5 and 6221 over 5 stars rating. Table 14 shows the results for function `RMSE(test_rating, predicted_ratings)`.

Table 14: RMSE results with Matrix Factorization Model

Method	RMSE
Just the Average	1.0600131
Movie effect model	0.9436204
Movie + User effects model	0.8662966
Movie + User + Genre effects model	0.8661932
Regularized Movie + User effects model	0.8655751
Matrix Factorization Model	0.7951871

This model provides us a much better result than the last three, with an approximate 0.07 drop, so we choose it to perform the final evaluation on the final hold-out set `validation`.

3 Results

3.1 Training final model with full edx dataset

To guarantee this final prediction model can be applied to the whole validation set, we will perform training with a full `edx` set, in the same way done in the previous Section. Once again, training will be performed with the optimal parameters defined by the tuning process.

```
train_dm <- data_memory(user_index = edx$userId, item_index = edx$movieId,
                          rating = edx$rating, index1 = TRUE)
rm(edx)

r$train(train_dm, opts = c(optimal_params, nthread = 1, niter = 20))
```

3.2 Final predicted ratings and RMSE

We finally predict ratings for the validation dataset, and get the following range of values:

```
validation_dm <- data_memory(user_index = validation$userId,
                              item_index = validation$movieId, index1 = TRUE)
validation_rating <- validation$rating

predicted_ratings = r$predict(validation_dm, out_memory())
rm(train_dm,validation)

summary(predicted_ratings)
```

```
>>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
>>  -1.883   3.058   3.545   3.473   3.965   6.159
```

We note there are 433 values under 0.5 rating and 3586 over 5 stars, but the real model evaluation happens with RMSE.

Table 15: Validation set's RMSE result with Matrix Factorization Model

Method	Validation RMSE
Matrix Factorization Model	0.7887703

Table 15 provides us the final RMSE, which achieves the value of 0.7887703. It is an excellent result, if either compared to other models in this report or to the winning RMSE value in the *Netflix* challenge, which was 0.8712 [Koren, 2009a]. Even though that challenge utilized a different dataset, it is still remarkable.

4 Conclusion

We began this report with a brief introduction about recommendation systems, explaining how they work and their groundbreaking achievements. We then continued by obtaining the data and generating our dataframes, followed by important visualizations, such as the ratings distribution and its variability between different movies and users. Genre and genre combinations also showed strong effects on ratings, while some attributes did not add significant value, like the rating date effect.

The size of the utilized datasets affected and limited which algorithms we could use. Models like `caret` package's k-NN and linear regression methods would either crash R or run too slow. Yet, various machine learning models were evaluated, starting with simpler ones like the Overall Average Rating and Movie effects algorithms. More complex models were utilized, like Movie + User and, specially, Movie + User + Genre effects. Since the latter did not have a great influence on the RMSE value, we didn't use it during the following regularized modeling. Even penalization addition to Movie + User effects didn't improve much either. The best and easiest model applied was Matrix Factorization with the *recosystem* R package. From the third model, we had already reached Netflix challenge's prize-winning RMSE, and by the end, a value under 0.8 was achieved.

As future work goes, We can tune the Matrix Factorization model with a wider range of values to further optimize it. We can also determine "ceiling" and "roof" values for the predictions, i.e., we transform predicted ratings under 0.5 stars or over 5 into these two limit values, further improving our RMSE results.

References

W.S. Chin, Y. Zhuang, Y.C. Juan, and C.J. Lin. A learning-rate schedule for stochastic gradient methods to matrix factorization. In *PAKDD*, 2015a. URL https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/mf_adaptive_pakdd.pdf.

W.S. Chin, Y. Zhuang, Y.C. Juan, and C.J. Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM TIST*, 2015b. URL https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/libmf_journal.pdf.

R.A. Irizarry. Introduction to data science: Data analysis and prediction algorithms with r, 2021. URL https://rafalab.github.io/dsbook/.

Y. Koren. *The BellKor Solution to the Netflix Grand Prize*, 2009a. URL https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf.

Y. Koren. Matrix factorization techniques for recommender systems. In *IEEE CS*, 2009b. URL https://www.inf.unibz.it/~ricci/ISR/papers/ieeecomputer.pdf.

Y. Qiu. *recosystem: Recommender System Using Parallel Matrix Factorization*, 2021. URL https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html.