MovieLens Machine Learning Tutorial

Vincent G. Kienzler

18/11/2019

Contents

1 Tutorial overview

1.1 Purpose

This data-science tutorial and work sample (programmed in R) was developed for a start-up team of software engineers and business analysts engaged in a machine learning training. The team members were invited to enroll in their choice of courses from the Harvard Professional Certificate in Data Science[1], and this tutorial served both as a guide and as a work sample for the capstone project of the course. It is therefore designed for an audience which already has some knowledge of data science, as the core concepts are only briefly summarised. The models and the workflow, however, are developed in details.

The R code and clean datasets for this tutorial can be accessed here: https://github.com/vgkienzler/movielens-tutorial.git

Additional, well-detailed explanations of the various concepts mentioned in this tutorial can be found in the following resources:

- *Introduction to Data Science*, 2019, by R. A. Irizarry[2]
- *An Introduction to Statistical Learning*, corr. 7th printing, 2017, by James, Witten, Hastie and Tibshirani[3]
- *The Elements of Statistical Learning*, 2nd edition, 2009, by Hastie, Tibshirani and Friedman[4]
- *Applied Statistical Learning*, 2013, by Kuhn and Johnson[5]

[1] https://www.edx.org/professional-certificate/harvardx-data-science
[2] https://rafalab.github.io/dsbook/
[3] http://faculty.marshall.usc.edu/gareth-james/ISL/index.html
[4] https://web.stanford.edu/~hastie/ElemStatLearn/
[5] http://appliedpredictivemodeling.com/toc

1.2 The challenge

This tutorial develops two different machine learning algorithms for a movie recommendation system: a linear model and a matrix factorisation model. Their goal is to predict movie ratings for a particular movie and user, based on a set of features (also referred to as independent variables or predictors). These algorithms are trained on a dataset of 10 million ratings. This dataset contains ratings of about 10,000 movies by 72,000 users. It is publicly available here.

Because the dataset contains both a series of known ratings Y (outcomes) and its associated series of features $(X_1, X_2, \cdots, X_p)$, the two algorithms can be developed with various tuning options and parameters and their respective predictions $\hat{Y}$ compared to the real outcome Y in the dataset. This comparison allows to estimate which model is the best at predicting the outcome. The best one can then be applied to new and unknown data, in order to make useful rating predictions.

The results show that matrix factorisation outperforms largely the linear model (matrix factorisation RMSE: 0.79445, linear model RMSE: 0.86495). This tutorial details what such results mean and how they have been achieved.

1.3 Recommendation systems

Recommendation systems have become an essential part of any e-business website or service. Their purpose is to suggest relevant items like movies, news articles, books or goods to users or clients. These items are ranked based on their predicted relevancy to the target user, and only the most relevant are suggested.

Recommendation systems are based on different types of algorithms, which are usually divided into two main categories: **collaborative filtering** and **content-based** algorithms, depending on what data is used to make the predictions.

Collaborative filtering algorithms rely on collecting preference information of many users or clients (hence "collaborative") to provide recommendations to a particular user (hence "filtering"). These algorithms rely only on historical data (past ratings) and no other information such as the user's age or gender is used. Data for such systems usually consist of a matrix of preferences or ratings with users in rows and products in columns. The system learns about the preferences of the target user based on her previous ratings, and suggests items highly rated by the other users with similar preference profiles.

Content-based algorithms, on the other hand, use information about the user (such as occupation, age, gender, location or ethnicity) and the product (such as genre, actors and year of release for a movie) to make the recommendations.

Most recent recommendation systems rely on a mix of both methods. The two algorithms introduced here are largely based on collaborative filtering since no information is available about the users, other than their ratings of a given set of movies. Only two complementary pieces of information are available about the movies: the genre of the movie and the year of release of the movie.

1.4 Document structure

This document is organised as follows:

- Section **2. Method and workflow** introduces the two models and provides details about their algorithms, as well as the workflow.
- Section **3. Data processing** provides a detailed account of the data preparation, analysis and visualisation process.
- Section **4. Modelling** develops and applies the two models and their respective algorithms. It assesses the models' ability to predict movie ratings.

- Section **5. Results** presents the final results, when the models are applied to the `validation` dataset;
- Section **6. Conclusion** discusses these results and concludes.

2 Method and workflow

2.1 Methods applied

2.1.1 Reminders

The general purpose of machine learning is to provide an estimate for the relationship f between an outcome Y and a set of predictors $X_1, \cdots, X_p$ defined such that:

$$Y = f(X_1, \cdots, X_p) + \epsilon_i$$

ϵ_i is the difference, or error, between $f(X_1, \cdots, X_p)$ and the real value Y. Here, the i of ϵ stands for "irreducible". It means that, even if the perfect f was ever identified, this error would still exist and be different from 0. This is due to the fact that the predictors $X_1, \cdots, X_p$ might not suffice to calculate exactly Y, simply because Y might be determined by unknown predictors other than $X_1, \cdots, X_p$. For instance, the heights of parents can help predict the heights of their kids (taller parents would tend to have taller kids). However, is the heights of the parents enough to predict those of the kids? Obviously not, many other factors might also influence this outcome: the kids' age, for one, but also their gender, diet, etc.

In addition, f is unknown. It is only a concept at this point. All the work of a data-scientist consists in identifying this f as accurately as possible. Unfortunately, chances are that f can only be approximated. This approximation, or estimate, is referred to as $\hat{f}$.

Knowing f (or rather $\hat{f}$, an estimate) can be useful for two reasons. It can be used to understand better how the predictors influence the outcome (*inference*), when $\hat{f}$ takes a form that can be interpreted by a human intelligence. It can also be used to predict values of Y from the predictors (*prediction*). Once an estimate for f is available, an estimate of Y can be calculated:

$$\hat{Y} = \hat{f}(X_1, \cdots, X_p)$$

$$Y = \hat{Y} + \epsilon$$

Here ϵ is the difference, or error, between the estimate $\hat{Y}$ and the real value Y. It is made of two parts: the *reducible* error (ϵ_r), and the *irreducible* error (ϵ_i), already introduced.

$$\epsilon = \epsilon_i + \epsilon_r$$

ϵ_i cannot be reduced, even if and when $f = \hat{f}$, it is not equal to 0.

ϵ_r can be reduced and depends on the quality of the machine learning algorithm used to calculate $\hat{f}$. It is equal to 0 if and only if $f = \hat{f}$.

All the work of developing a machine learning model consists in identifying $\hat{f}$ in such a way that the error ϵ is as small as possible. The central question of any machine learning challenge is therefore: what shape should f take?

In this tutorial two models are developed and their respective errors calculated. As it will become clear by the end of this tutorial, not all models are born equal. Some models are good at predicting outcomes, but are difficult to interpret (inference). Others are just the opposite: they make it easy to understand what influences an outcome, but are less good at predicting it.

These two models are detailed next.

2.1.2 Linear modelling

2.1.2.1 Building the model The first method consists in building a linear model (linear regression on categorical variables). f can use the deviation of a rating y to the mean of all the ratings, μ, based on a sum of biases:

$$y = \mu + \sum_i b_i + \epsilon$$

ϵ contains the part of a rating that cannot be explained by the biases identified.

Because it is not possible to access the mean of *all* ratings, but only the mean of the ratings in the dataset available, and because the biases cannot be exactly calculated for the same reasons, the model becomes:

$$y = \hat{\mu} + \sum_i \hat{b}_i + \epsilon'$$

Where $\hat{\mu}$ and $\hat{b}_i$ are estimates of μ and the biases b calculated with the limited amount of data available in the dataset (if the dataset where to change, $\hat{\mu}$, ϵ' and $\hat{b}_i$ would change).

As it is shown later, the dataset available (referred to as the "10M MovieLens dataset") contains enough features to include the following biases in the model: movie, user, genre, movie release year and date of the rating. This model could be represented as follows:

$$\hat{y} = \hat{\mu} + \hat{b}_m + \hat{b}_u + \hat{b}_g + \hat{b}_y + \hat{b}_t$$

This means that the prediction of a rating $\hat{y}$ of user u made at time and date t of movie m of genre g released in year y is based on the average of all the ratings in the dataset, $\hat{\mu}$, to which are added biases which account for the difference of a rating to the mean. These biases are the following:

- $\hat{b}_m$: Movie bias, or movie effect. It is linked to the fact, simply, that some movies are better than others and therefore receive better ratings on average.
- $\hat{b}_u$: User bias, or user effect. Reflects the fact that some users have higher (or lower) expectations than others, and give harsher or better ratings.
- $\hat{b}_u$: Genre bias, or genre effect. This reflects the idea that some genres might attract better ratings than others.
- $\hat{b}_y$: Release year bias. This bias reflects the idea that the release date of a movie might have an impact on its ratings. For example, older movies might be less well rated because of older special effects, slower pace of action, etc. This effect might partly be captured in the movie effect.
- $\hat{b}_t$: Rating time stamp bias. This represents the fact that a rating might depend of the general state of the world at the date and time of the rating. It might depend on other movies released at this time, what are the important problems of this period, the socio-political context, the state of mind of the viewer, etc.

From now on, in the rest of the tutorial, for the shake of simplicity, the $\hat{}$ is dropped on μ and all the biases b_i. The reader is invited to remember that the limited dataset available only makes it possible to discuss *estimates* and not the real biases or means.

2.1.2.2 Regularisation Let's look at how each b_i is calculated. The first solution is to use linear regression calculation available in r. However, this approach is very slow as there are potentially several thousands of coefficients to calculate. In this particular situation, the theory tells us that the least square estimate of the biases for each category (the regression coefficients), is the average of the biases across this

category. In other words, the least square estimate of the movie bias is the average of the biases of all the ratings y_m for movie m, where N_m is the number of ratings for movie m.

$$b_m = \frac{1}{N_m} \sum_{i=1}^{N_m} y_{m,i}$$

As the next section on data visualisation shows, some movies have very few ratings. Similarly, some users or genres are only linked to few observations. In such cases, the average depends on a few ratings. It is likely that this average is not an accurate estimate of the real bias. To compensate for such cases, the bias can be calculated including a penalty factor, λ:

$$b_m = \frac{1}{(N_m + \lambda)} \sum_{i=1}^{N_m} y_{m,i}$$

In this case, whenever N_m is small, the corresponding bias is reduced by the addition of λ. However, when N_m is large, λ only has a limited impact on the result ($N_m \approx N_m + \lambda$). This technique, called regularisation, is used to estimate each of the biases.

2.1.3 Matrix factorisation

The second model consists in applying matrix factorisation using the `recosystem` package[6]. In this approach, a rating matrix with users in row and movies in columns is built based on the dataset. Such a matrix would look like this, with `?` referring to missing ratings:

	movie 1	movie 2	movie 3	movie 4
user 1	2	?	?	4
user 2	?	?	?	1
user 3	?	3	?	5
user 4	1	?	3	4

A popular technique to estimate the missing ratings `?` is the matrix factorisation algorithm. The rating matrix $R_{n \times p}$ is extracted from the dataset and is estimated by the product of two matrices, $P_{k \times n}$ and $Q_{k \times p}$, such that:

$$R \approx P^\top Q$$

A typical solution for P and Q is given by the following optimization equation:

$$\min_{P,Q} \sum_{(i,j) \in R} [l(p_i, q_j; r_{i,j}) + \mu_P ||p_i||_1 + \mu_Q ||q_j||_1 + \frac{\lambda_P}{2} ||p_i||_2^2 + \frac{\lambda_Q}{2} ||q_j||_2^2]$$

Where:

- p_i: the i-th column of P.
- q_j: the j-th column of Q.
- The rating from user u on movie m would be predicted as $p_u^\top q_v$.
- $r_{i,j}$: the observed rating in R, row i, column j.
- l: the loss function.

[6] https://cran.r-project.org/web/packages/recosystem/index.html

- $\mu_P, \mu_Q, \lambda_P, \lambda_Q$: penalty parameters to avoid overfitting.

For more details about the maths behind this model please refer to this page, where the information communicated here is extracted from. However, using the R `recosystem` package will make this exercise easy and straightforward. It provides functions to easily run model training (building P and Q) and parameter tuning (selection of the penalty factors λ_P and λ_Q).

2.2 Error (loss) function

Measuring errors is essential in the development of any machine learning model. In some machine learning algorithms, like in matrix factorisation, the error function which measure errors (conventionally called loss function) is part of the algorithm itself. In other cases, the error function is used to guide the development and select what model provides the best prediction or classification. Selecting the right loss function is therefore important, and different loss functions lead to different results.

Errors (losses) can be measured in different ways. In the case of movie rating, the simplest way could consist in subtracting the prediction $\hat{y}$ to the real values y and adding all the results for all the observations. However, positive errors would compensate negative errors and an overall error measure of 0 could hide very large errors. For this reason the absolute value or the squared values of the differences is commonly included in the loss function. Let's discuss briefly the three most common loss functions: the *Mean Squared Error (MSE)*, the *Root Mean Squared Error* and the *Mean Absolute Error (MAE)*.

- *Mean Squared Error*: it is the simplest to calculate. It gives more importance to large errors compared to small errors. The unit of the MSE is not the same as the unit of the outcome.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- *Root Mean Squared Error*: it has the same unit as the outcome and also gives more weight to large errors. This metrics is therefore better when outliers are undesirable.

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

- *Mean Absolute Error*: it has the same unit as the outcome and gives equal weight to large and small errors.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

  More information about loss functions can be found here[7] and here[8] as well as in the resources mentioned at the beginning of this tutorial. The results of both the algorithms developed in this document are estimated using the RMSE loss function.

2.3 Workflow

In order to develop a machine learning algorithm, the following workflow is usually recommended.

1. **Framing the problem.** This step consists in identifying the problem that we are trying to solve and the dataset which would be required to solve this problem.

[7] https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23
[8] https://towardsdatascience.com/importance-of-loss-function-in-machine-learning-eddaaec69519

2. **Collecting data.** This step involves gathering enough data to build a dataset which makes it possible to build accurate algorithms.

3. **Preparing data.** This step involves cleaning and organising the data, such as downloading and parsing the dataset, adding column names, changing the column types, splitting columns into several columns if necessary, dealing with missing values, etc. The original dataset is then split into three sets: a training set, a testing set and a validation set. The training and testing sets are used to develop and train the models. The validation set is used only at the very end, once the models have been fully developed. Using the validation set to make any sort of decision during the development of the algorithms will lead to overfitting: the model will perform below expectations once applied to new, unknown data.

4. **Data exploration and visualisation.** This step involves exploring the various features in the dataset, such as their variation and distribution. This steps plays an important role to become familiar with the data and start thinking about what ML algorithm will be best for the challenge at hand.

5. **Building the models and running the algorithms.** This step involves actually developing the algorithm and validating it.

6. **Communication.** Once the algorithm has been developed it is important to communicate the results, its characteristics and how it has been developed. This transparency is what will help build trust in the model.

Steps 4 and 5 are interlinked to a certain extent. During step 5, it might be required to explore the data further, do additional visualisation and change the formatting of the data to better fit the model developed. For example, in the case of the 10M MovieLens dataset, it is not obvious from the get-go that it could help to split the movie title, a string which includes the release year: `Two Towers, The (2002)`, into two columns: the movie name only `Two Towers, The` and the release year `2002`.

In the present situation, only steps 3 to 6 are necessary, since the problem is already well defined and the dataset of movie ratings is already constituted.

The diagram below summarises the process (credit: Wikipedia, article "Data Visualisation")[9].

3 Data processing

3.1 Data preparation

Let's start with downloading and preparing data. Running each of these blocks is somewhat time consuming. For this reason, some important datasets (`movielens`, `train_set`, `test_set`, `validation`) are saved for future use, in order to save time.

Step 1 - Load the packages needed:

```
library(knitr)
library(caret)
library(data.table)
library(stringr)
library(lubridate)
library(tidyverse)
library(recosystem)
library(ggthemes)
library(ggrepel)
library(scales)
```

[9] https://en.wikipedia.org/wiki/Data_visualization

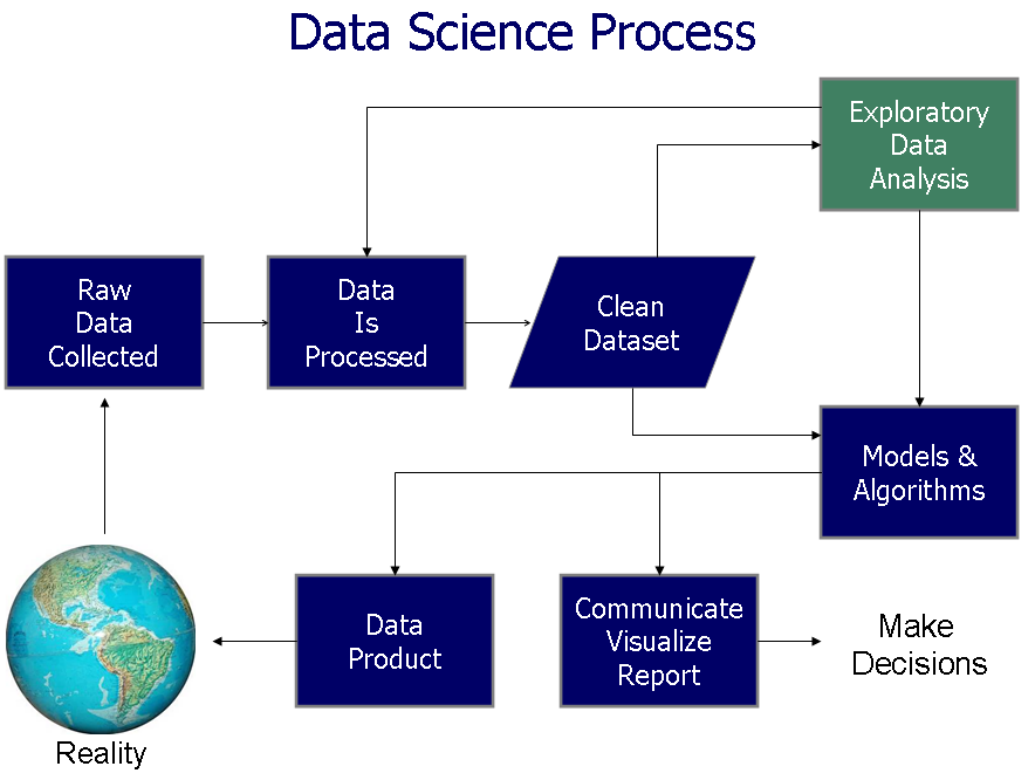


Figure 1: Datascience workflow

Step 2 - Download the "MovieLens 10M" dataset from https://grouplens.org/datasets/movielens/10m/:

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

Step 3 - Format the data properly. From the downloaded dataset, create a data frame `movielens` with proper column names (`userId`, `movieId`, `rating`, `timestamp`, `title`, `genres` and `year`):

```
# Import data from ratings.dat and add column names userId, movieId,
# rating and timestamp
ratings <- fread(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

# Import data from movies.dat, split the data string into 3 values
# allocated to 3 columns (movieId, title and genres)
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(levels(movieId))[movieId],
         title = as.character(title),
         genres = as.character(genres))

# Join movies and ratings together in movielens
movielens <- left_join(ratings, movies, by = "movieId")

# Split the title column in two columns, "title" and "year"
movielens <- movielens %>% mutate(ttyr = title) %>%
  extract(title, c("title", "year"), regex = "(.*)\\s\\((\\d+)\\)", convert = TRUE)

# Save "movielens" data frame in "data" directory for later use if needed
# Creates "data" repository in the current directory if it doesn't exist
ifelse(!dir.exists(file.path("data")), dir.create(file.path("data")), FALSE)
save("movielens", file = "data/movielens.RData")
```

Step 4 - Split the `movielens` dataset into a training set, a testing set and a validation set:

```
# Validation set will be 10% of "movielens" data
set.seed(1, sample.kind="Rounding")

test_index <- createDataPartition(y = movielens$rating,
                                  times = 1, p = 0.1, list = FALSE)
train_test <- movielens[-test_index,]
temp <- movielens[test_index,]

# Make sure "userId" and "movieId" in "validation" set are also in "train_test" set
validation <- temp %>%
  semi_join(train_test, by = "movieId") %>%
  semi_join(train_test, by = "userId")

# Add rows removed from "validation" set back into "train_test" set
removed <- anti_join(temp, validation)
train_test <- rbind(train_test, removed)
```

```r
# Split "train_test" data set between "test_set" and "train_set"
set.seed(100, sample.kind="Rounding")

test_index <- createDataPartition(y = train_test$rating,
                                  times = 1, p = 0.2, list = FALSE)
train_set <- train_test[-test_index,]
temp <- train_test[test_index,]

# Making sure that there are no "userId" or "movieId" in "test_set" which are not in
# "training_set"
test_set <- temp %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")

# If any rows has been discarded from "test_set", add it back to the "train_set"
removed <- anti_join(temp, test_set)
train_set <- rbind(removed, train_set)

# Making sure that no rows has been lost (should return TRUE)
nrow(train_test) == nrow(train_set) + nrow(test_set)

# Do a bit of cleaning:
rm(dl, ratings, movies, test_index, temp, removed, movielens)

# The following datasets now exist: "validation", "test_set", "train_set."
# Save these data frames for future use in "data" directory to avoid repeating the work
# Creates "data" repository in the current directory if doesn't exist
ifelse(!dir.exists(file.path("data")), dir.create(file.path("data")), FALSE)
save("train_test", file = "data/train_test.RData")
save("test_set", file = "data/test_set.RData")
save("train_set", file = "data/train_set.RData")
save("validation", file = "data/validation.RData")
# To use locally stored data on the next run of this rmarkdown, set local_data to TRUE
```

The **golden rule of machine learning** is that the validation data should *never* be used to fit or train the models, in any way, even during data exploration, since it can inform decision making about feature engineering or data modelling. Validation data should only be used to provide an unbiased evaluation of a model, close to how the models would perform on unknown data. If any learning based on the validation data were included in the model, this unbiased estimate would not be possible. Only the `train_set`, `test_set` and `train_test` (which includes both train and test sets) can be used to explore the data, make modelling choices, refine parameters and fit the models.

3.2 Exploration and visualisation

3.2.1 General overview

Let's explore the `train_test` dataset. What is the structure of the dataset? The `str()` function returns the size of the dataset as well as the class of each feature (chr, num, int, factor) and the values of the factors, if any. It's a good place to start.

```r
# View the structure of the dataset
str(train_test)
```

The dataset has 8 columns, also called variables, input variables, features or predictors. These 8 columns are the following:

- `userId`: an integer which identifies the user who made the rating,
- `movieId`: a numerical which identifies the movie the rating refers to,
- `rating`: a numerical between 0.5 and 5, the rating from `userId` on `movieId`,
- `timestamp`: an integer which contains information about the date and time of the rating,
- `title`: a character string, which contains the name of the movie. This column is extracted from the column `ttyr` which is kept in the dataset for verification purposes,
- `year`: an integer, the year of the release data of the movie,
- `genres`: a character string, which lists all the genres of the movie.

Now let's see how many ratings, *unique* movies and *unique* users are included in the dataset.

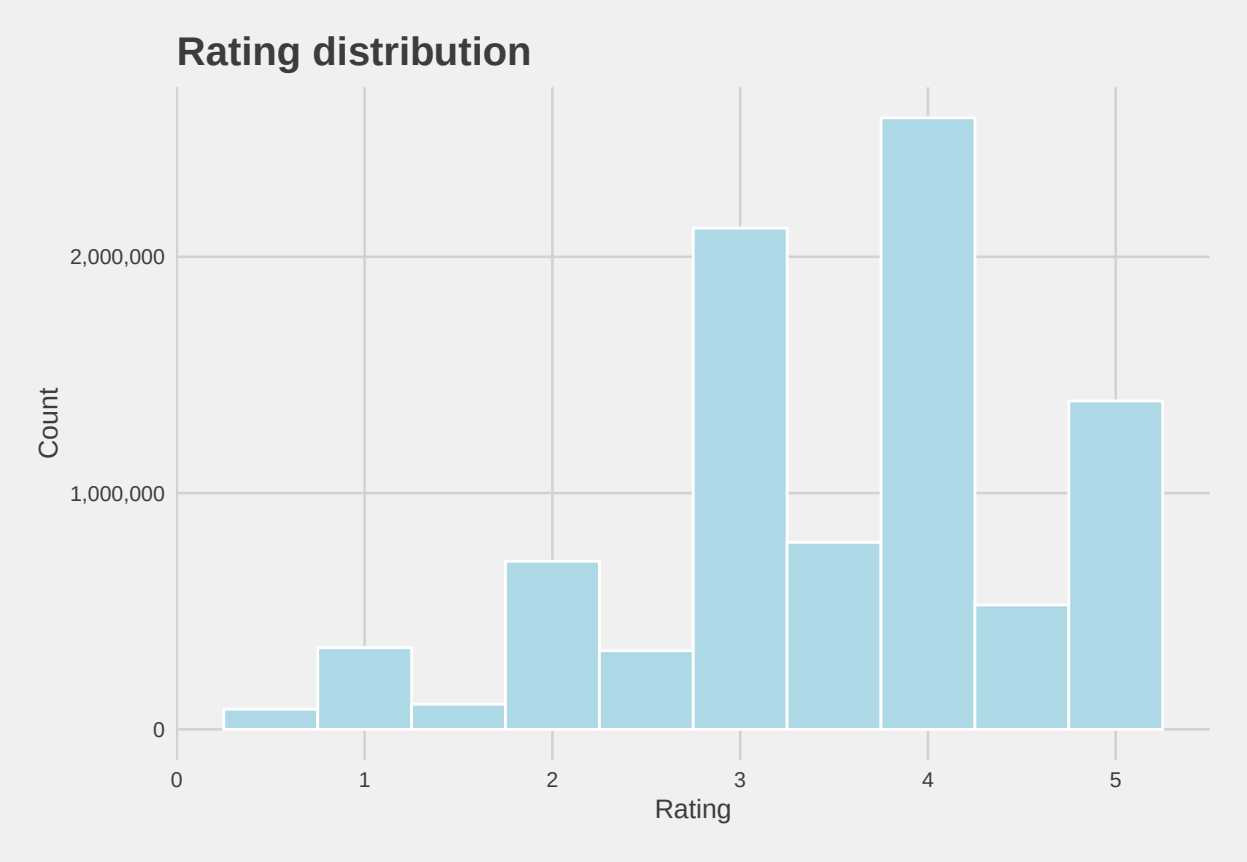
```
# Number of unique movies and users:
kable(train_test %>%
        summarize(n_ratings = n(), unique_users = n_distinct(userId),
                  unique_movies = n_distinct(movieId)),
      caption = "Number of unique movies and users")
```

Table 2: Number of unique movies and users

n_ratings	unique_users	unique_movies
9000055	69878	10677

What is the distribution of the ratings? As the histogram below shows, ratings range from 0.5 to 5 and users tend to give ratings of 3 and above. Exact ratings (1,2,3,4 and 5) are more frequent than intermediate ratings (.5).

```
# Rating distribution
theme_set(theme_fivethirtyeight(base_size = 10))
theme_update(axis.title = element_text())
train_test %>% ggplot(aes(rating)) +
  geom_histogram(color = "white", bins = 10, fill="lightblue") +
  ggtitle("Rating distribution") +
  xlab("Rating") +
  ylab("Count") +
  scale_y_continuous(labels = comma)
```



What is the mean and median values of the ratings? The mean is lower than the median, meaning that the distribution is skewed to the left. Users give on average ratings of 3.5, with as many users giving a rating of 4 or higher than giving a rating below 4.

```
mean(train_test$rating)
```

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

```
median(train_test$rating)
```

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

3.2.2 Distribution of ratings per user and movie

Let's look at the distribution of the ratings per user and movie. How many ratings have the 10 most rated movies received?

```
# Number of ratings received by the 10 most rated movies
movie_ratings <- train_test %>%
  select(movieId, title, rating) %>%
  group_by(movieId, title) %>%
  summarize(n_ratings = n(), average = mean(rating))

kable(head(movie_ratings %>% arrange(desc(n_ratings)), 10),
  caption = "Ratings of the 10 most rated movies")
```

Table 3: Ratings of the 10 most rated movies

movieId	title	n_ratings	average
296	Pulp Fiction	31362	4.154789
356	Forrest Gump	31079	4.012822
593	Silence of the Lambs, The	30382	4.204101
480	Jurassic Park	29360	3.663522
318	Shawshank Redemption, The	28015	4.455131
110	Braveheart	26212	4.081852
457	Fugitive, The	25998	4.009155
589	Terminator 2: Judgment Day	25984	3.927859
260	Star Wars: Episode IV - A New Hope (a.k.a. Star Wars)	25672	4.221311
150	Apollo 13	24284	3.885789

How many ratings have the 10 least rated movies received?

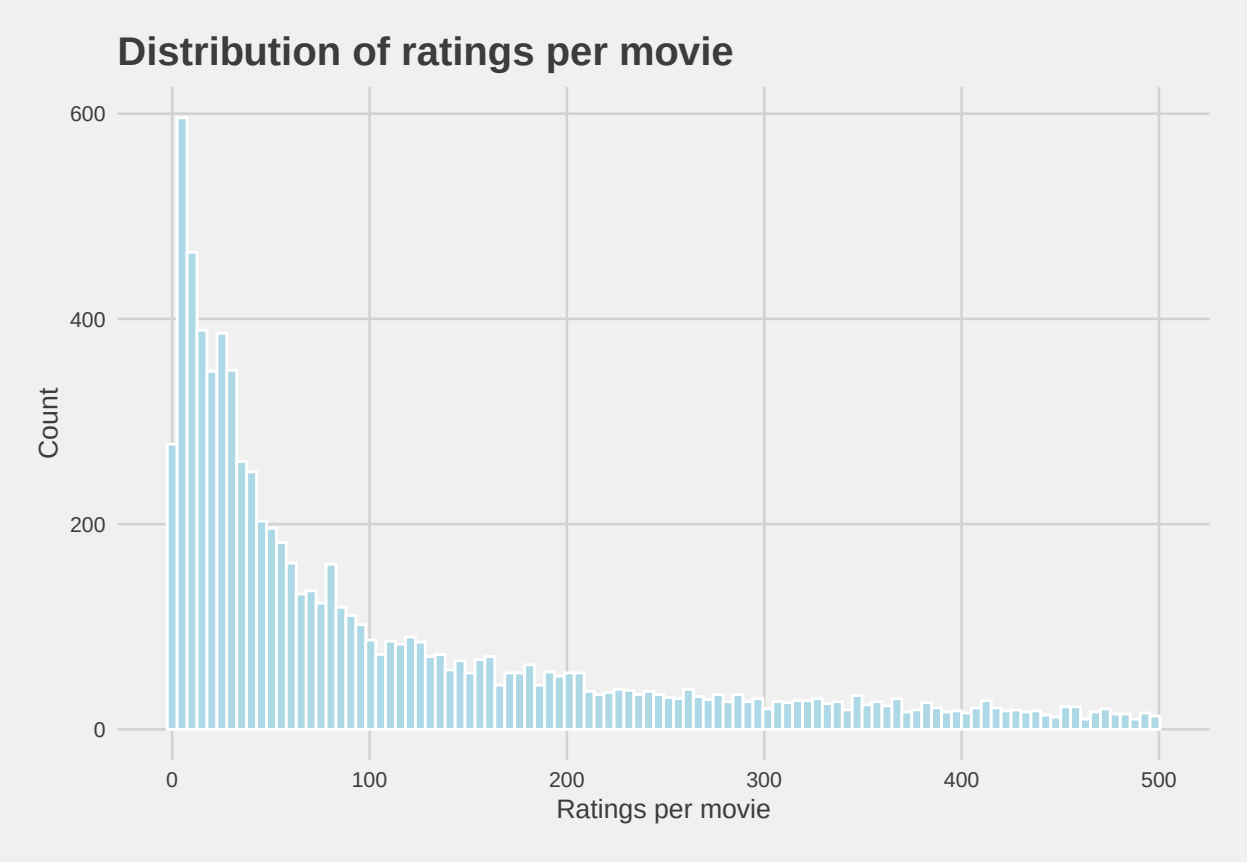
```
# Number of ratings received by the 10 least rated movies
kable(head(movie_ratings %>% arrange(n_ratings), 10),
      caption = "Ratings of the 10 least rated movies")
```

Table 4: Ratings of the 10 least rated movies

movieId	title	n_ratings	average
3191	Quarry, The	1	3.5
3226	Hellhounds on My Trail	1	5.0
3234	Train Ride to Hollywood	1	3.0
3356	Condo Painting	1	3.0
3383	Big Fella	1	3.0
3561	Stacy's Knights	1	1.0
3583	Black Tights (1-2-3-4 ou Les Collants noirs)	1	3.0
4071	Dog Run	1	1.0
4075	Monkey's Tale, A (Les ChÃ¢teau des singes)	1	1.0
4820	Won't Anybody Listen?	1	2.0

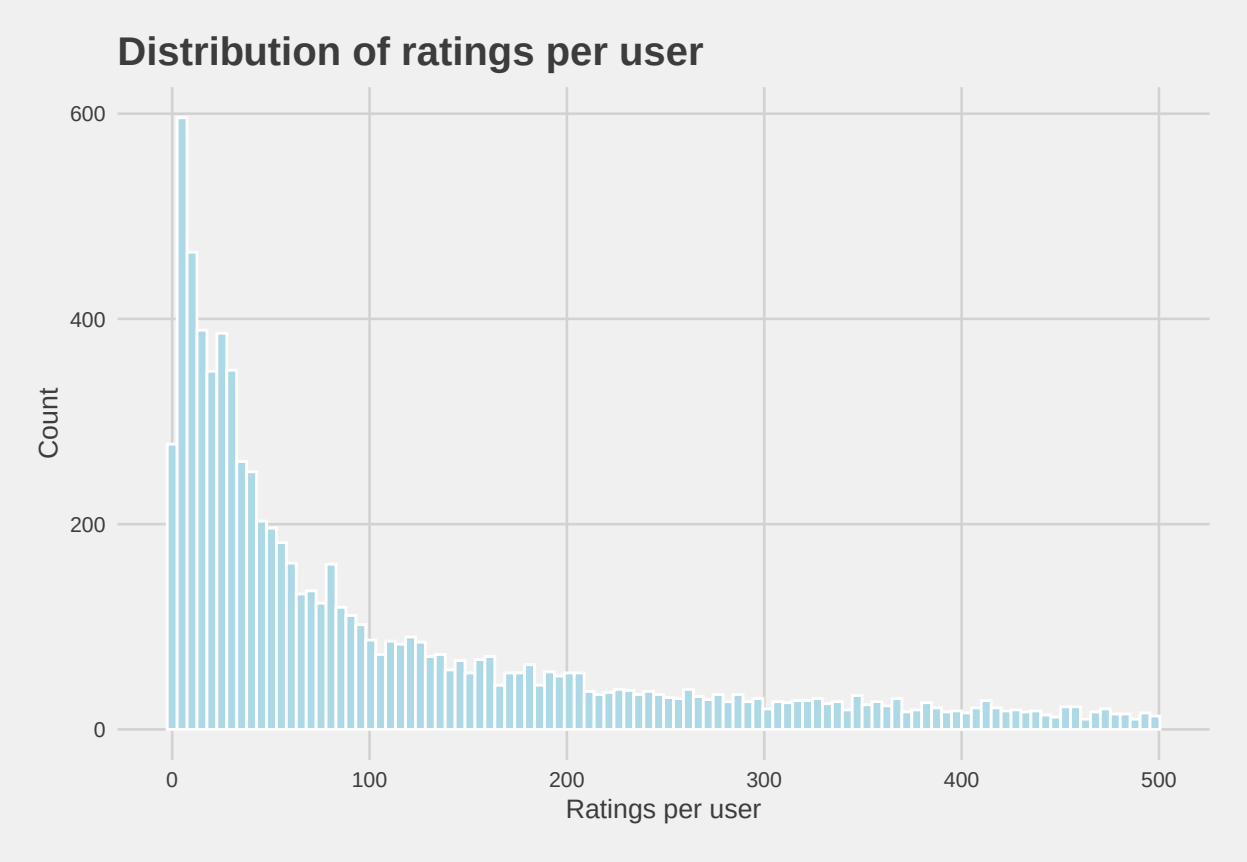
To get a better sense of how many ratings movies have received, let's plot an histogram of the number of ratings per movie. As the histogram shows, most movies have less than 100 ratings, and the number of ratings per movie varies significantly.

```
# Histogram of the number of ratings per movie
# (limiting plotting at 500 ratings per movie max for visibility).
ratings_per_movie <- movie_ratings %>%
  filter(n_ratings < 500)

ggplot(ratings_per_movie, aes(x=n_ratings)) +
  geom_histogram(bins = 100, color = "White", fill="lightblue") +
  ggtitle("Distribution of ratings per movie") +
  xlab("Ratings per movie") +
  ylab("Count") +
  scale_y_continuous(labels = comma)
```



Similarly, the distribution of the number of ratings per users gives a sense of how many ratings viewers have given. Most users made less than 100 ratings, but one viewer made as much as 6616 ratings!

```
# Histogram of the number of ratings per user
# (limiting plotting at 500 ratings per user max for visibility).
user_ratings <- train_test %>%
  select(userId, rating) %>%
  group_by(userId) %>%
  summarize(n_ratings = n(), average = mean(rating))

ratings_per_users <- user_ratings %>%
  filter(n_ratings < 500)

ggplot(ratings_per_movie, aes(x=n_ratings)) +
  geom_histogram(bins = 100, color = "White", fill="lightblue") +
  ggtitle("Distribution of ratings per user") +
  xlab("Ratings per user") +
  ylab("Count") +
  scale_y_continuous(labels = comma)
```



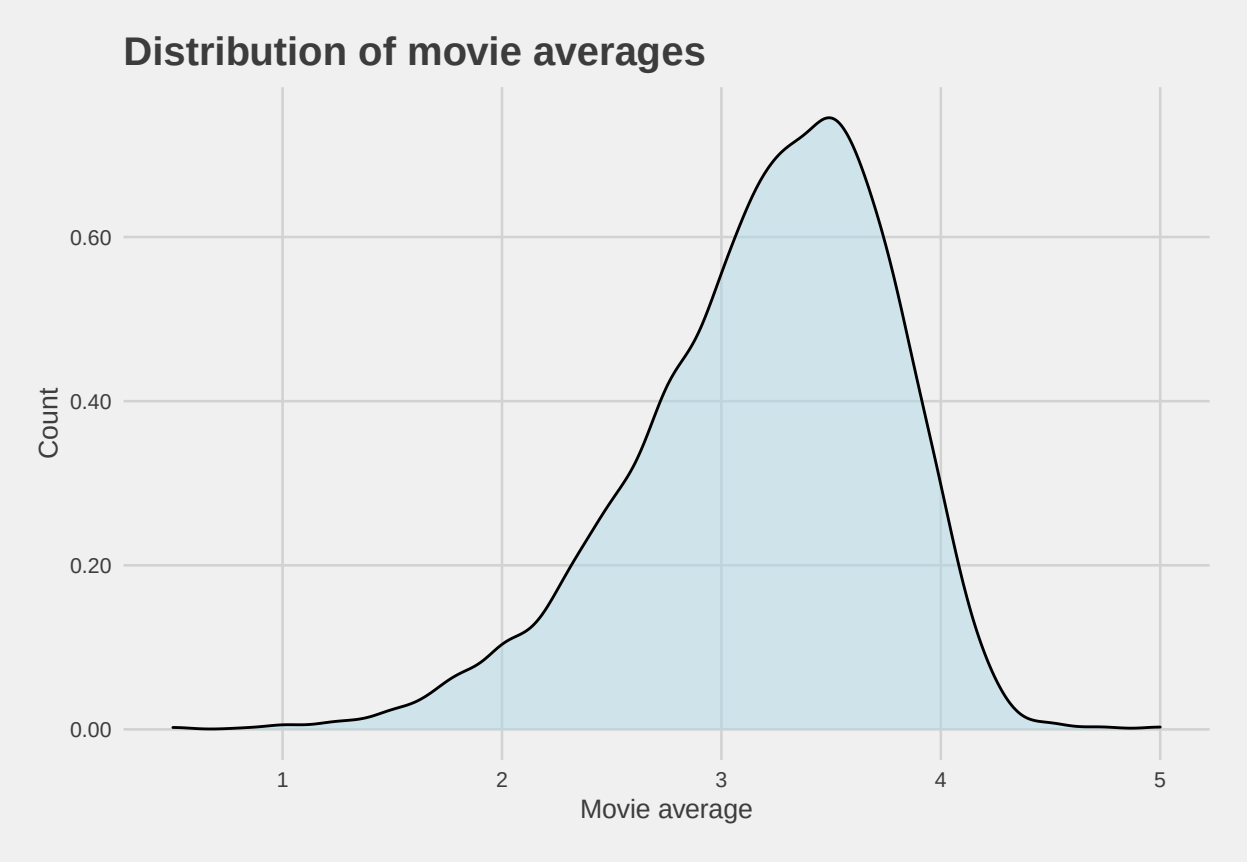

```
# Maximum of number of ratings per user
max(user_ratings$n_ratings)
```

```
## [1] 6616
```

3.2.3 Distribution of rating averages per user and movie

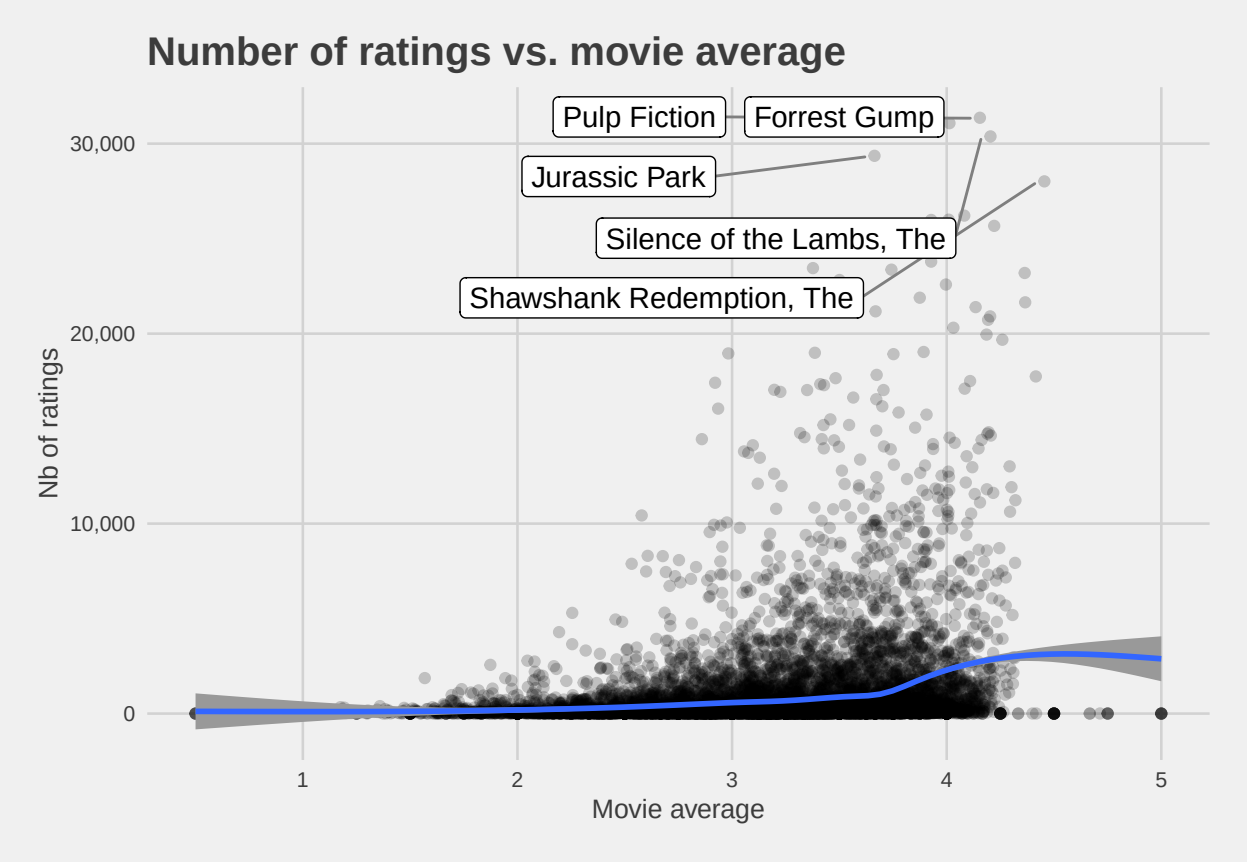
Now let's look at the distribution of the rating average per movie. As the density plot below shows, lots of movies have rating averages between 3 and 4.

```
# Density of movie averages
ggplot(movie_ratings, mapping = aes(x=average)) +
  geom_density(fill="lightblue", alpha=0.5) +
  ggtitle("Distribution of movie averages") +
  xlab("Movie average") +
  ylab("Count") +
  scale_y_continuous(labels = comma)
```



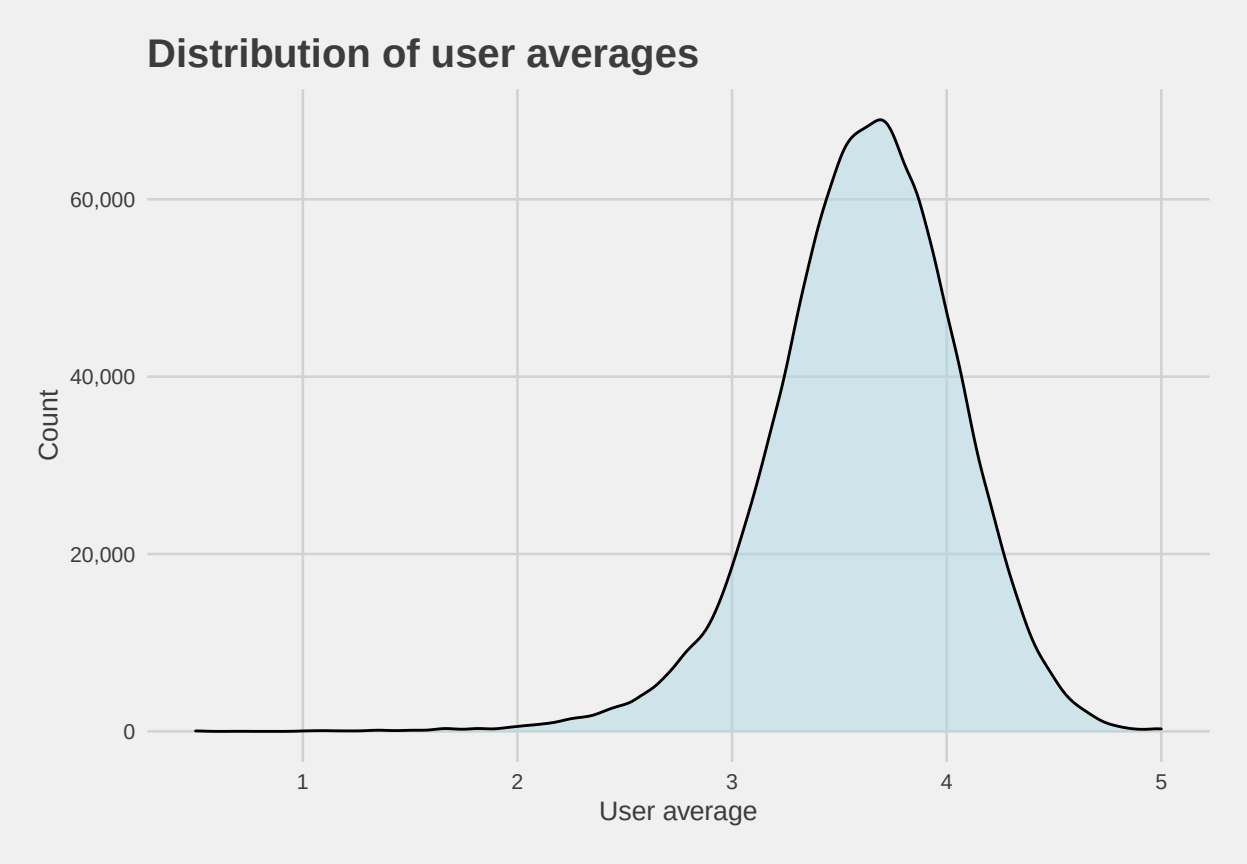
Do movies with higher average have more ratings? Indeed, as the graph below shows, it seems that the higher a movie average, the higher the number of ratings.

```
# Graph of the number of ratings per movie vs. movie average
movie_ratings %>% ggplot(aes(x=average, y=n_ratings)) +
  geom_point(alpha=0.2) +
  geom_smooth(alpha=1) +
  ggtitle("Number of ratings vs. movie average") +
  geom_label_repel(aes(label=ifelse(n_ratings>27000,title,'')),
                   size=4,
                   hjust=0,
                   vjust=0,
                   box.padding   = 0.25,
                   point.padding = 0.5,
                   segment.color = 'grey50') +
  xlab("Movie average") +
  ylab("Nb of ratings") +
  scale_y_continuous(labels = comma)
```



What is the distribution of averages per users?

```
# Density plot of the user averages
ggplot(user_ratings, mapping = aes(x=average, y = after_stat(count))) +
  geom_density(fill="lightblue", alpha=0.5) +
  ggtitle("Distribution of user averages") +
  xlab("User average") +
  ylab("Count") +
  scale_y_continuous(labels = comma)
```



As this density plot shows, a large proportion of the users gives ratings of 3.5 and above. Let's have a look at the means of user averages and movie averages. The results show that the mean of user averages is higher than the mean of movie averages, which can be explained by the fact that movies with higher ratings get rated more.

```
# Comparison of mean of user averages and movie averages
message("Mean of user averages: ", round(mean(user_ratings$average), 2))
```

```
## Mean of user averages: 3.61
```

```
message("Mean of movie averages: ", round(mean(movie_ratings$average), 2))
```

```
## Mean of movie averages: 3.19
```

3.2.4 Exploring the genres, years of release, and time stamps

The previous section has provided great insights about the distribution of ratings and rating averages per users and movies. In this section the genres, years of release and time stamps of the ratings and movies are analysed.

3.2.4.1 Genres Let's start with the genre of the 10 most rated movies.

```
kable(head(train_test %>% group_by(title, genres) %>%
       summarize(n_rating = n()) %>%
       arrange(desc(n_rating)), 10),
      caption = "Genres of the 10 most rated movies")
```

Table 5: Genres of the 10 most rated movies

title	genres	n_rating
Pulp Fiction	Comedy|Crime|Drama	31362
Forrest Gump	Comedy|Drama|Romance|War	31079
Silence of the Lambs, The	Crime|Horror|Thriller	30382
Jurassic Park	Action|Adventure|Sci-Fi|Thriller	29360
Shawshank Redemption, The	Drama	28015
Braveheart	Action|Drama|War	26212
Fugitive, The	Thriller	25998
Terminator 2: Judgment Day	Action|Sci-Fi	25984
Star Wars: Episode IV - A New Hope (a.k.a. Star Wars)	Action|Adventure|Sci-Fi	25672
Apollo 13	Adventure|Drama	24284

As the table shows, some movies have more than one genre. How many different groupings of genres exist in the dataset?

```
n_distinct(train_test$genres)
```

```
## [1] 797
```

Let's see what are the 10 most common genre combinations.

```
genre_comb <- data.frame(sort(table(train_test$genres), decreasing = TRUE))
colnames(genre_comb) <- c("Combination", "Count")
kable(t(t(head(genre_comb, 10))),
      caption = "The 10 most common genre combinations")
```

Table 6: The 10 most common genre combinations

Combination	Count
Drama	733296
Comedy	700889
Comedy|Romance	365468
Comedy|Drama	323637
Comedy|Drama|Romance	261425
Drama|Romance	259355
Action|Adventure|Sci-Fi	219938
Action|Adventure|Thriller	149091
Drama|Thriller	145373
Crime|Drama	137387

It is also possible to extract the list of unique genres and count them.

```
list_genres <- unique(unlist(strsplit(train_test$genres, split = "\\|")))
print(list_genres)
```

```
##  [1] "Comedy"             "Romance"            "Action"
##  [4] "Crime"              "Thriller"           "Drama"
##  [7] "Sci-Fi"             "Adventure"          "Children"
## [10] "Fantasy"            "War"                "Animation"
## [13] "Musical"            "Western"            "Mystery"
## [16] "Film-Noir"          "Horror"             "Documentary"
## [19] "IMAX"               "(no genres listed)"
```

```
print(length(list_genres))
```

```
## [1] 20
```

There are 19 different unique genres. Some movies have no genre listed, and there is a total of 797 different combinations of genres.

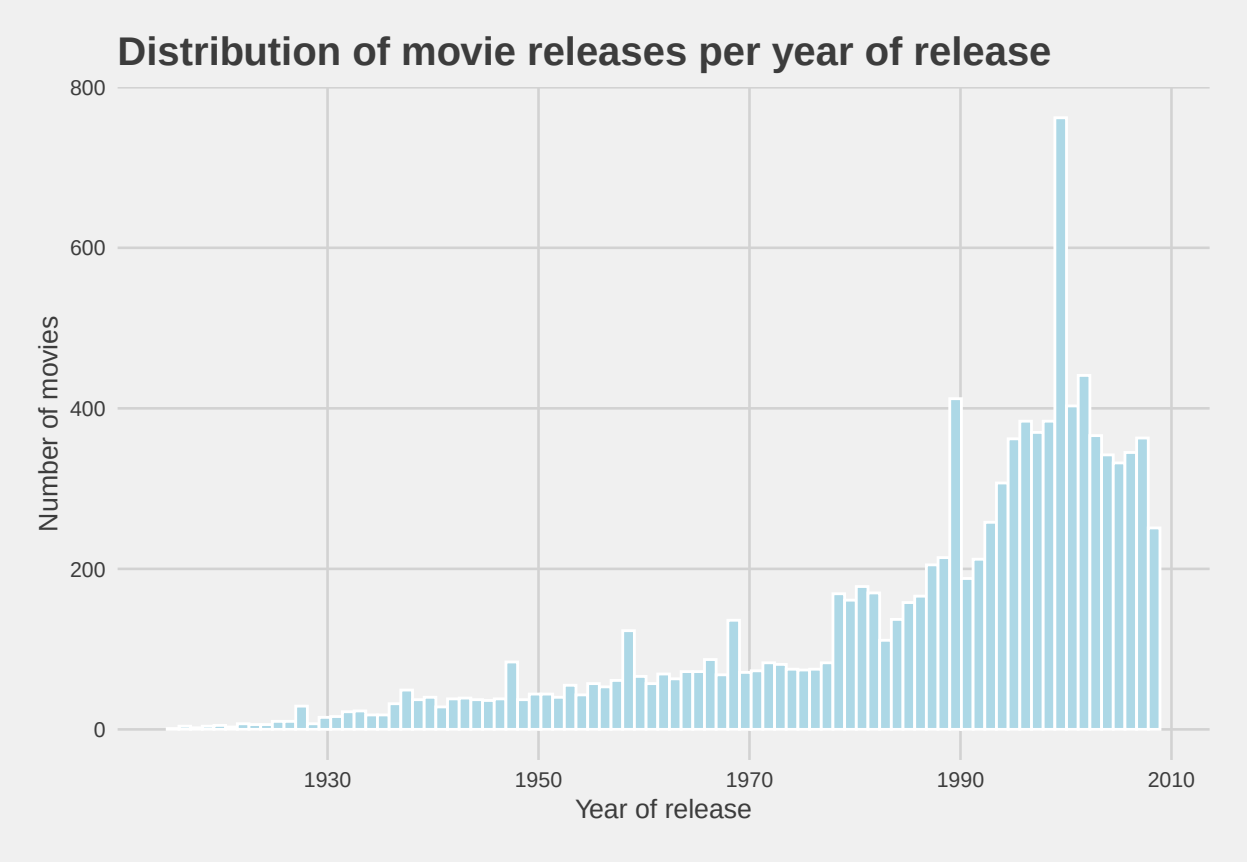
3.2.4.2 Year of release

Let's look at the number of movie releases per year.

```
# Distribution of movie releases per year of release.
year_movie <- train_test %>% distinct(movieId, .keep_all = TRUE)

message("Last movie released in: ", max(year_movie$year))
```

```
## Last movie released in: 2008
```

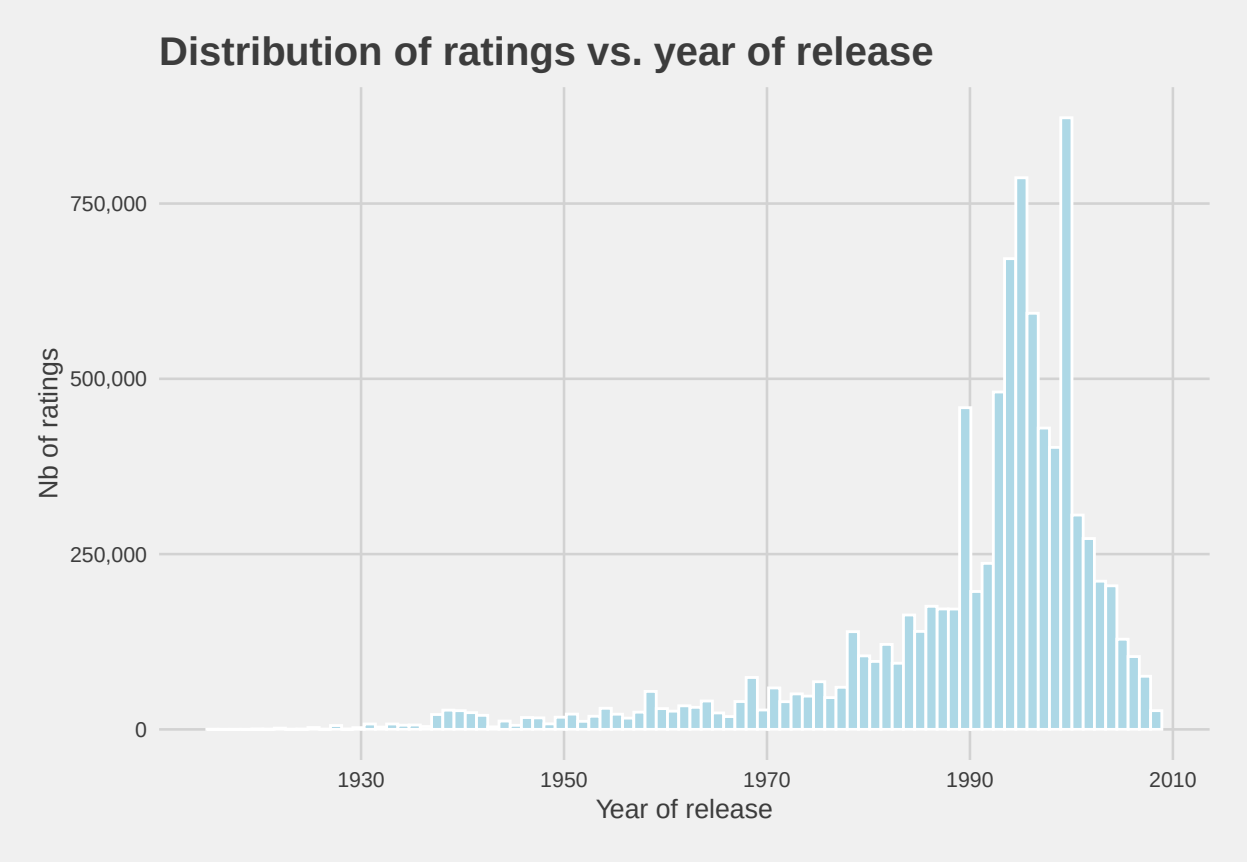
```
ggplot(year_movie, aes(x=year)) +
  geom_histogram(bins = 85, color = "White", fill="lightblue") +
  ggtitle("Distribution of movie releases per year of release") +
  xlab("Year of release") +
  ylab("Number of movies") +
  scale_y_continuous(labels = comma)
```



Most of the releases have taken place after 1995. The latest movie in the dataset was released in 2008. Old movies make only a small part of the dataset.

What is the number of ratings per year of release? As the histogram below shows, older movies have few ratings. This can both mean that the proportion of old movies in the dataset is low, and/or that viewers are less interested in these old movies. It is easy to test which hypothesis is correct by plotting the average number of ratings per movie per year of release.

```
# Number of ratings vs. year of release.
rating_year <- train_test %>%
  group_by(year) %>%
  summarize(n_ratings = n())

ggplot(train_test, aes(x=year)) +
  geom_histogram(bins = 85, color = "White", fill="lightblue") +
  ggtitle("Distribution of ratings vs. year of release") +
  xlab("Year of release") +
  ylab("Nb of ratings") +
  scale_y_continuous(labels = comma)
```



What are the 10 release years with the most ratings?

```
kable(head(train_test %>% group_by(year) %>%
       summarize(n_rating = n()) %>%
       arrange(desc(n_rating)), 10),
      caption = "10 release years with the most ratings")
```

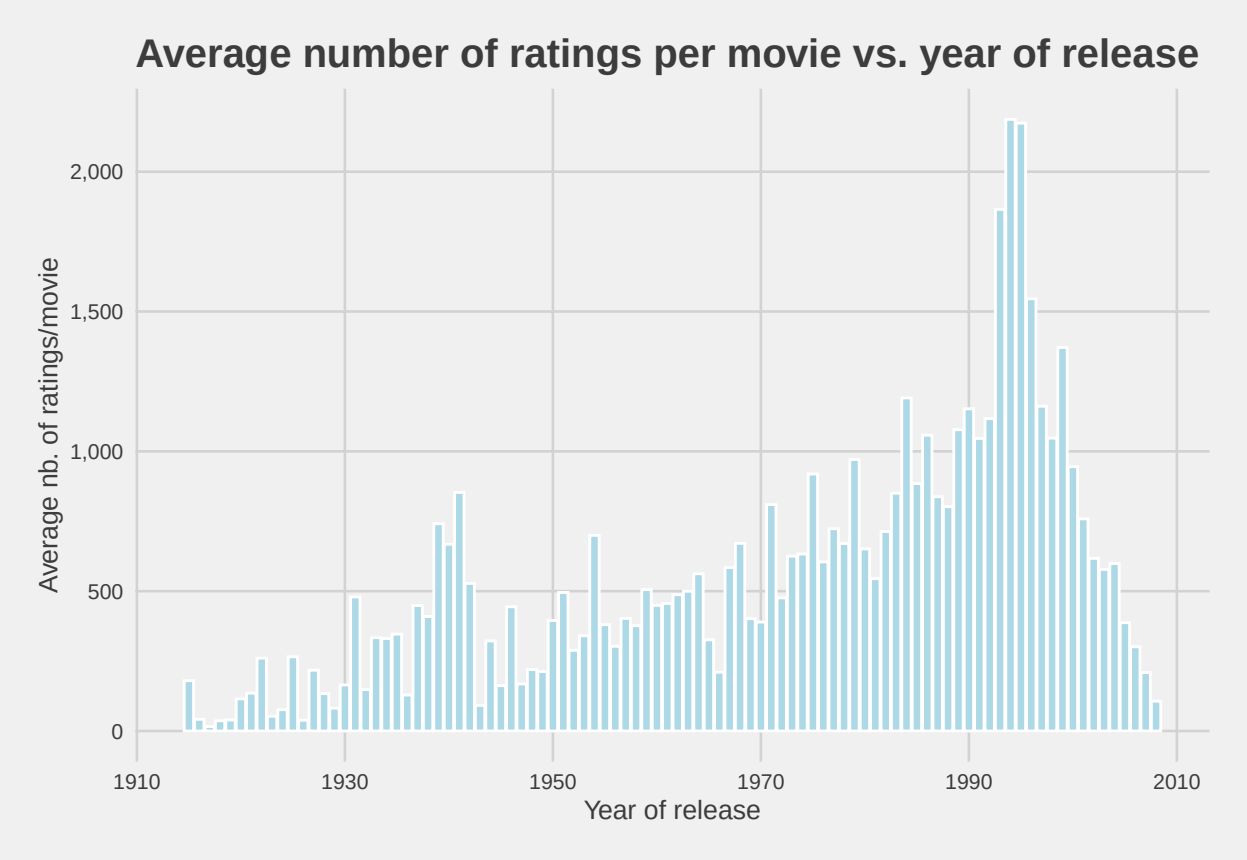
Table 7: 10 release years with the most ratings

year	n_rating
1995	786762
1994	671376
1996	593518
1999	489537
1993	481184
1997	429751
1998	402187
2000	382763
2001	305705
2002	272180

As the table shows, movies from the 90s have the most ratings, followed by movies in the 2000s. In order to understand better the interest of viewers in old and new movies, let's see what is the average number of

ratings per movie for each year of release. As the histogram below shows, older movies have, on average, a lower count of ratings than newer movies, except for the most recent ones (movies released after 2003). This is likely due to the fact that very old movies are less interesting for contemporary viewers, and the most recent movies have been seen by fewer people because they have only been on the market for a short period of time.

```
# Nb or ratings per movie vs. release year of the movies.
rating_movie_year <- train_test %>%
       group_by(movieId, year) %>%
       summarise(n_ratings = n()) %>%
       group_by(year) %>%
       summarise(movie_average = mean(n_ratings))

ggplot(rating_movie_year, aes(x=year, y=movie_average)) +
  geom_col(color = "White", fill="lightblue") +
  ggtitle("Average number of ratings per movie vs. year of release") +
  xlab("Year of release") +
  ylab("Average nb. of ratings/movie") +
  scale_y_continuous(labels = comma)
```



3.2.4.3 Time stamp The `timestamp` value gives the time and date when a rating was made. How has the frequency of the ratings changed with time?
In what year were the youngest and oldest rating made?

```r
# Finding the youngest and oldest ratings.
birthyear_rating <- train_test  %>%
  mutate(date = as_datetime(timestamp)) %>%
  mutate(ts_year = floor_date(date, unit = "year"))

message("Year of the oldest rating: ", year(min(birthyear_rating$ts_year)))
```

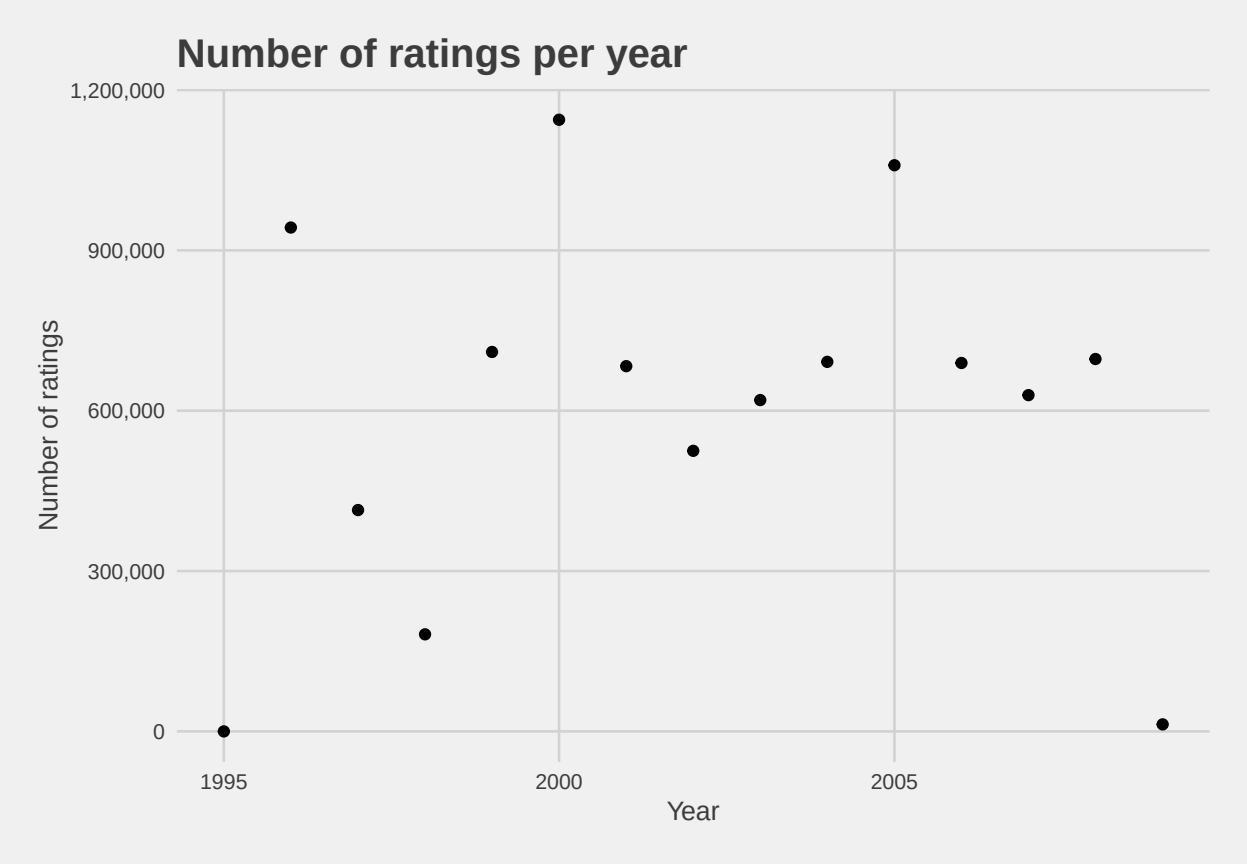
```
## Year of the oldest rating: 1995
```

```r
message("Year of the youngest rating: ", year(max(birthyear_rating$ts_year)))
```

```
## Year of the youngest rating: 2009
```

How has the number of ratings per year evolved over time? As the figure below shows, year 2000 has seen the most ratings.

```r
# Number of ratings done per year
ratings_year <- birthyear_rating %>%
  group_by(ts_year) %>%
  summarize(n_rating = n())

ratings_year %>% ggplot(aes(ts_year, n_rating)) +
  geom_point() +
  ggtitle("Number of ratings per year") +
  xlab("Year") +
  ylab("Number of ratings") +
  scale_y_continuous(labels = comma)
```



This concludes the data exploration and visualisation section. Let's move to building the models.

4 Modelling

4.1 Important functions

First let's define a few important functions which are used throughout the rest of the analysis to measure performance, store performance information and compare models.

- The **RMSE() function**: this function will calculate the RMSE based on $\hat{Y}$ and Y.

```
# Declare RMSE function:
RMSE <- function(y_predic, y_real){
  sqrt(mean((y_predic-y_real)^2))
}
```

- The **add_row_results() function** to populate the data frame which will store the various RMSE corresponding to the different models:

```
# Create data frame to store the results of the various models
model_results <- data.frame(
  model_nb = integer(),
  type = character(),
```

```
  lambda =  numeric(),
  rmse = numeric(),
  stringsAsFactors = FALSE)

# Since model type will be factor we can create a function to make it
# easier to add a new row and a new factor to model_results:
add_row_results <- function(row, model_nb, new_type, opt_lambda, new_rmse){
  model_results[row,  "model_nb"] <- model_nb
  model_results[row, "type"] <- new_type
  model_results[row, "lambda"] <- opt_lambda
  model_results[row, "rmse"] <- new_rmse
  return (model_results)
}
```

- Finally let's create the **improvement() function** that will calculate the improvement (in %) between two models:

```
# Returns the variation in % from model_1 rmse to model_2 rmse.
improvement <- function(model_1, model_2){
  model_results$rmse[model_2]
  model_results$rmse[model_1]
  return (-(model_results$rmse[model_2]-model_results$rmse[model_1])
          /model_results$rmse[model_1]*100)
}
```

4.2 Linear model

In this section we will model one bias after another using regularisation and check, each time, what is the corresponding RMSE. This will give us a good sense of how each bias contributes to the accuracy of the model.

4.2.1 Naive mean (base model)

Let's start with a very simple algorithm, `naive_mean` which consists in taking the average rating of the training dataset and applying it to the test set. As the next table shows, the performance of this base model is low. Hopefully adding the different biases will help improve this performance.

```
# First initial models, naive_rmse
model_nb <- 1
mu <- mean(train_set$rating)
# Calculate the RMSE
rmse <- RMSE(test_set$rating, mu)
# Store result in the summary dataframe model_results
model_results <- add_row_results(model_nb, model_nb, "naive mean", NA, rmse)
# Display results
kable(model_results, caption = "Summary of results")
```

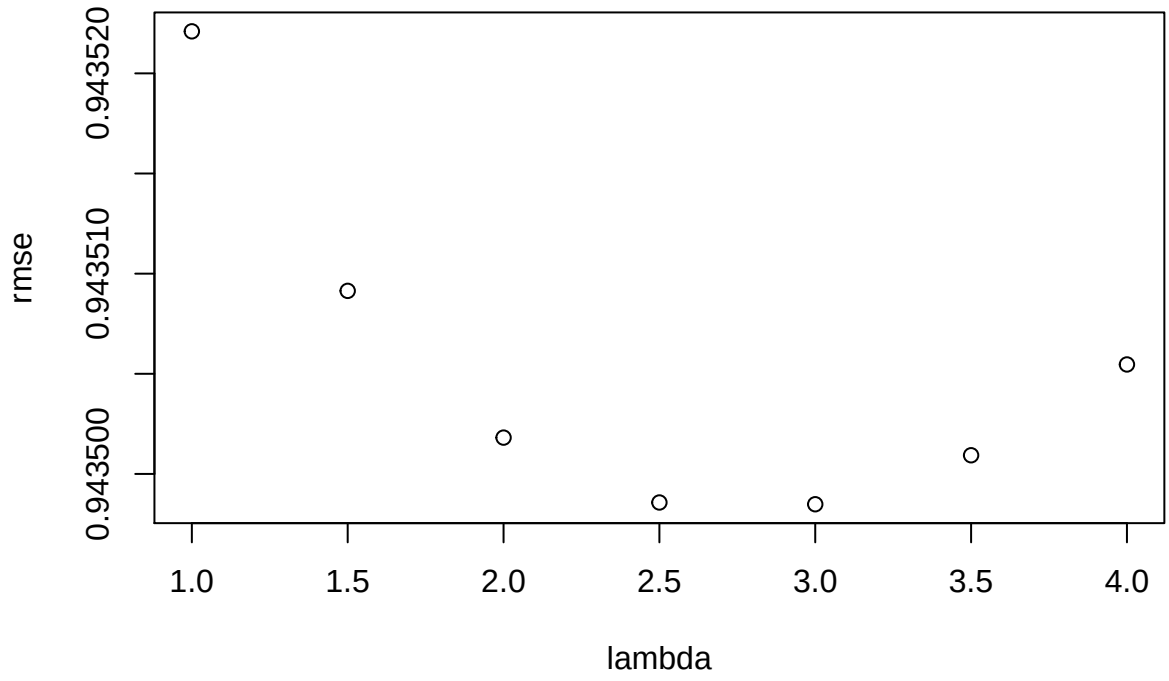
Table 8: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.059899

4.2.2 Movie effect

The next step is to improve the model by adding the regularised average per movie to the overall average. For each movie the code below calculates the difference between the movie average and the overall average (mu). This difference is the movie bias or movie effect b_m of the model.

```
model_nb <- 2
# Define a sequence of penalisation factor (lambda) for the regularization.
lambda <- seq(1, 4, 0.5)

# Declare the function that calculates the RMSE corresponding to each
# regularised average per movie based on the argument lbd.
rmse_lambda <- function(lbd){
  movie_reg_avg <- train_set %>%
    group_by(movieId) %>%
    summarize(bm = sum(rating - mu)/(lbd + n()))

  # Run the model on the test_set, compare the predicted ratings with the real one
  # and return the RMSE.
  movie_preds <- test_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    pull(bm)
  return(RMSE(test_set$rating, mu + movie_preds))
}

# Apply the previous function to each item in the lambda sequence.
rmse <- sapply(lambda, rmse_lambda)

# Plot the different RMSE for the different values of 'lbd'.
plot(lambda, rmse)
```



Let's get the lambda which minimises the RMSE and update the summary table `model_results`.

```
best_lbd <- lambda[which.min(rmse)]
# Store result in the summary dataframe model_results.
model_results <- add_row_results(
  model_nb, model_nb, "movie effect with regularisation", best_lbd, min(rmse))
# Display results
kable(model_results, caption = "Summary of results")
```

Table 9: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3	0.9434985

Adding the movie effect has increased the accuracy of the model by almost 10%.

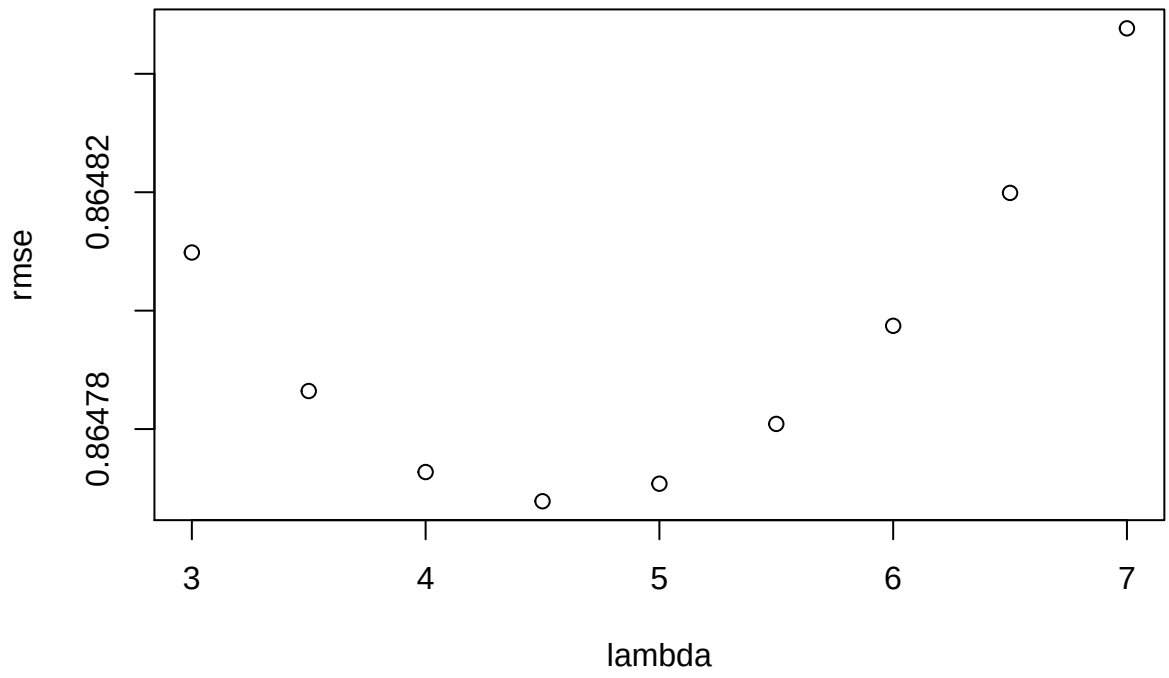
```
# Calculate the improvement between this model and the previous one.
improvement(1,2)
```

```
## [1] 10.9822
```

4.2.3 User effect

The model can be improved one step further by adding the regularised average per user. For each user the code below calculates the difference between the user average and the overall average corrected by the regularised movie average (mu and bm). This difference is the user bias or user effect b_u of the model.

```
model_nb <- 3
# Define a sequence of penalisation factor (lambda) for the regularization.
lambda <- seq(3, 7, 0.5)

# Calculate the movie effect using the best lambda identified previously.
movie_reg_avg <- train_set %>%
  group_by(movieId) %>%
  summarize(bm = sum(rating - mu)/(3 + n()))

# Declare the function that calculates the RMSE corresponding to each
# regularised average per movie based on the argument lbd.
rmse_lambda <- function(lbd){
  user_reg_avg <- train_set %>%
    left_join(movie_reg_avg, by='movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - bm - mu)/(lbd + n()))

  # Run the model on the test_set, compare the predicted ratings with
  # the real one and return the RMSE.
  movie_user_preds <- test_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    mutate(pred = mu + bu + bm) %>%
    pull(pred)
  return(RMSE(test_set$rating, movie_user_preds))
}

# Apply the previous function to each item in the lambda sequence.
rmse <- sapply(lambda, rmse_lambda)

# Plot the different RMSE for the different values of 'lbd'.
plot(lambda, rmse)
```



Let's get the lambda which minimises the RMSE and update the summary table `model_results`.

```
best_lbd <- lambda[which.min(rmse)]
# Store results in the summary dataframe model_results.
model_results <- add_row_results(
  model_nb, model_nb, "movie and user effects with reg.", best_lbd, min(rmse))
# Display results
kable(model_results, caption = "Summary of results")
```

Table 10: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678

Adding the user effect has further increased the accuracy of the model by more than 8%.

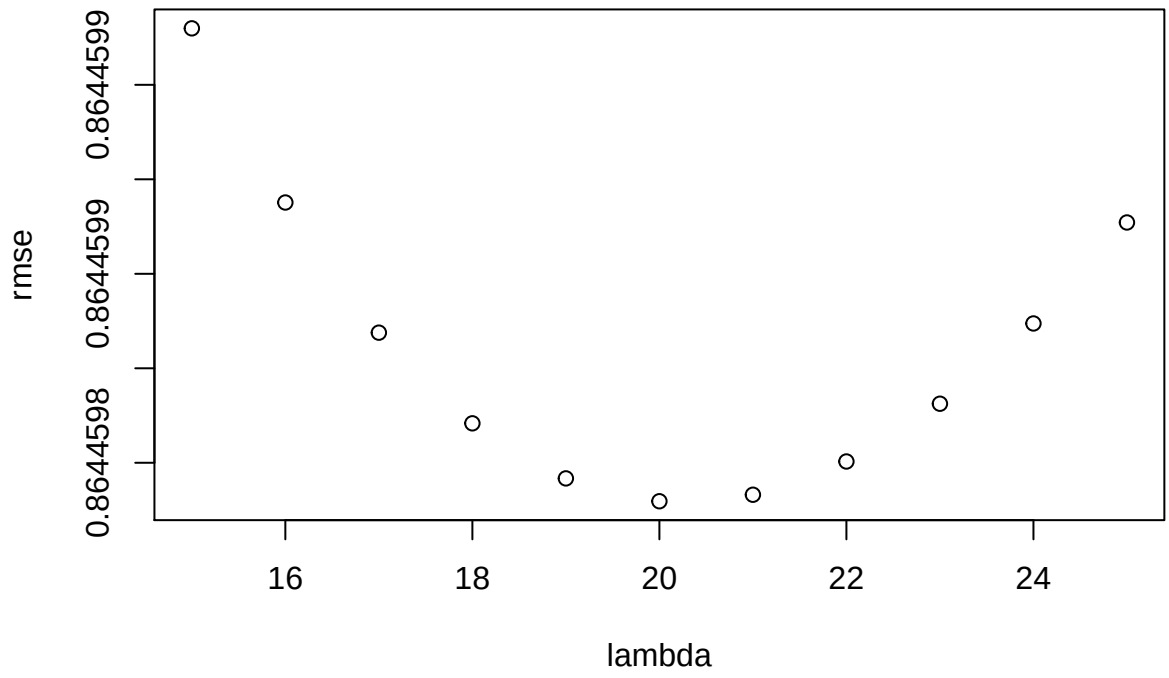
```
# Calculate the improvement between this model and the previous one.
improvement(2,3)
```

```
## [1] 8.344547
```

4.2.4 Genre effect

To improve the model further, the regularised average per genre is added. For each genre the code below calculates the difference between the genre average and the overall average corrected by the regularised movie and user averages (mu, bm and bu). This difference is the genre bias or genre effect b_g of the model.

```
model_nb <- 4
# Define a sequence of penalisation factor (lambda) for the regularization.
lambda <- seq(15, 25, 1)

# Calculates the user effect using the best lambda identified previously.
user_reg_avg <- train_set %>%
  left_join(movie_reg_avg, by="movieId") %>%
  group_by(userId) %>%
  summarize(bu = sum(rating - bm - mu)/(4.5 + n()))

# Declares the function that calculates the RMSE corresponding to each
# regularised average per movie based on the argument lbd.
rmse_lambda <- function(lbd){
  genre_reg_avg <- train_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    group_by(genres) %>%
    summarize(bg = sum(rating - bu - bm - mu)/(lbd + n()))

  # Run the model on the test_set, compare the predicted ratings with
  # the real one and return the RMSE.
  mug_preds <- test_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    left_join(genre_reg_avg, by="genres") %>%
    mutate(pred = mu + bg + bu + bm) %>%
    pull(pred)
  return(RMSE(test_set$rating, mug_preds))
}

# Apply the previous function to each item in the lambda sequence.
rmse <- sapply(lambda, rmse_lambda)

# Plot the different RMSE for the different values of 'lbd'.
plot(lambda, rmse)
```



Let's get the lambda which minimises the RMSE and update the summary table `model_results`.

```
best_lbd <- lambda[which.min(rmse)]
# Store result in the summary dataframe model_results.
model_results <- add_row_results(
  model_nb, model_nb, "movie, user and genre effects with reg.", best_lbd, min(rmse))
# Display results
kable(model_results, caption = "Summary of results")
```

Table 11: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598

As the change in RMSE indicates, the genre bias does not provide a significant improvement in accuracy (less than 0.05%). This is likely due to the fact that most of the impact is already captured in the movie effect.

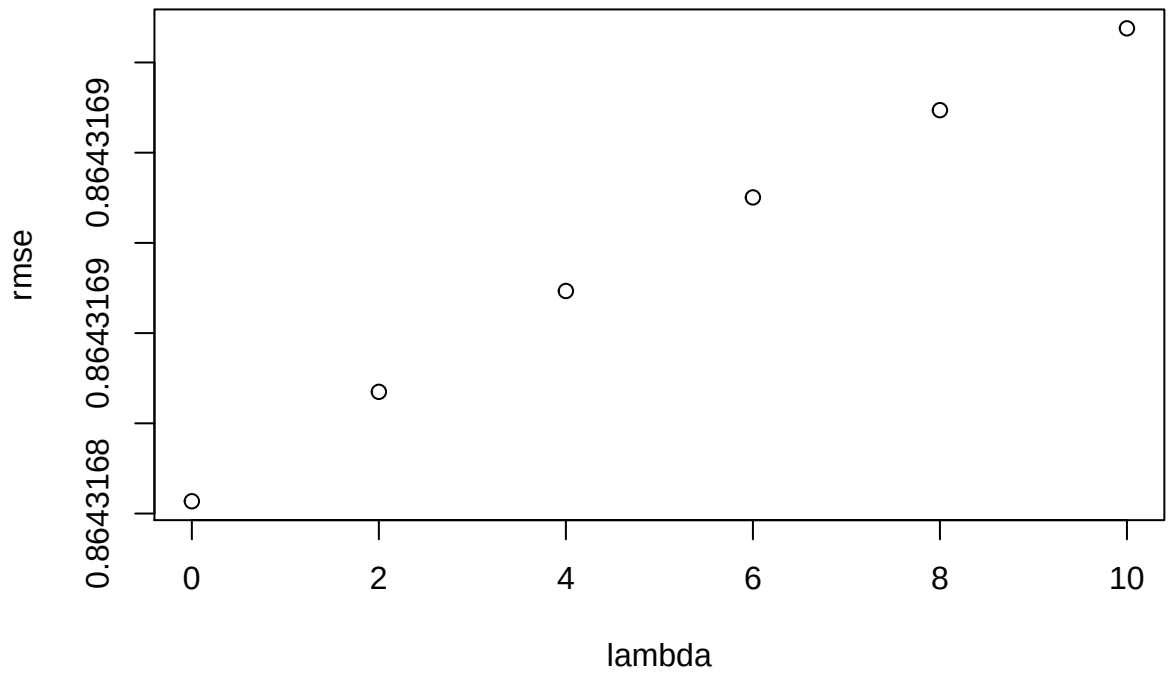
```
# Calculate the improvement between this model and the previous one.
improvement(3,4)
```

```
## [1] 0.03561359
```

4.2.5 Release date effect (year)

Let's add now the next bias, the regularised average per year of the date of release. For each year, the code below calculates the difference between the year average and the overall average corrected by the regularised movie, user and genre averages (mu, bm, bu, bg). This difference is the year bias or year effect b_y of the model.

```
model_nb <- 5
# Define a sequence of penalisation factor (lambda) for the regularization.
lambda <- seq(0, 10, 2)

# Calculate the genre effect using the best lambda identified previously.
genre_reg_avg <- train_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    group_by(genres) %>%
  summarize(bg = sum(rating - bm - bu - mu)/(20 + n()))

# Declare the function that calculates the RMSE corresponding to each
# regularised average per movie based on the argument lbd.
rmse_lambda <- function(lbd){
  year_reg_avg <- train_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    left_join(genre_reg_avg, by="genres") %>%
    group_by(year) %>%
    summarize(by = sum(rating - bm - bu - bg - mu)/(lbd + n()))

  # Run the model on the test_set, compare the predicted ratings with
  # the real one and return the RMSE.
  mugy_preds <- test_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    left_join(genre_reg_avg, by="genres") %>%
    left_join(year_reg_avg, by="year") %>%
    mutate(pred = mu + by + bg + bu + bm) %>%
    pull(pred)
  return(RMSE(test_set$rating, mugy_preds))
}

# Apply the previous function to each item in the lambda sequence.
rmse <- sapply(lambda, rmse_lambda)

# Plot the different RMSE for the different values of 'lbd'.
plot(lambda, rmse)
```



For the year effect, the penalisation does not improve the prediction. Let's update the summary table `model_results`.

```
best_lbd <- lambda[which.min(rmse)]
# Store result in the summary dataframe model_results
model_results <- add_row_results(
  model_nb, model_nb, "m, u, g and year effects with reg.", best_lbd, min(rmse))
# Display results
kable(model_results, caption = "Summary of results")
```

Table 12: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168

The contribution of the year effect is not important (less than 0.02%). The improvement for model 4 to 5 is marginal. This is likely due to the fact that most of this effect is already captured in the movie effect.

```
# Calculate the improvement between this model and the previous one.
improvement(4,5)
```

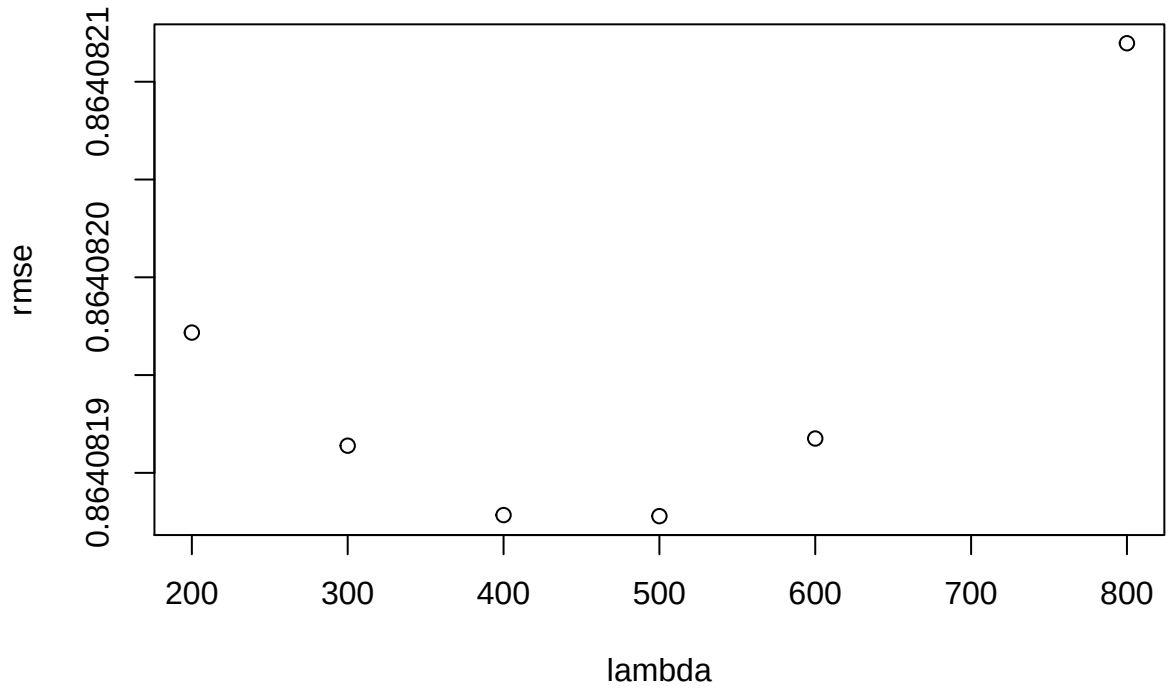
```
## [1] 0.01654365
```

4.2.6 Date of rating effect

Next the regularised date (`timestamp`) of the ratings is added to the model. In this case, the time stamps need to be grouped by period (`week`, `month` or `year`) in order to compute the regularised average. For each period, the code below calculates the difference between the period average and the overall average corrected by the regularized movie, user, genre and year averages (mu, bm, bu, bg, by). This difference is the date bias or effect, b_t, of the model. The code below runs with a weekly period. Running it with a month or year period would show that the `week` period has the most impact.

```
model_nb <- 6
# Define a sequence of penalisation factor (lambda) for the regularisation.
lambda <- c(200, 300, 400, 500, 600, 800)

unit_date = "week"

# Update the train_set and test_set datasets with a new column, date,
# which contains the conversion of timestamp to date object.
train_set <- train_set %>% mutate(date = as_datetime(timestamp))
test_set <- test_set %>% mutate(date = as_datetime(timestamp))

year_reg_avg <- train_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    left_join(genre_reg_avg, by="genres") %>%
    group_by(year) %>%
    summarize(by = sum(rating - bm - bu - bg - mu)/(n()))

# Declare the function that calculates the RMSE corresponding to each
# regularised average per movie based on the argument lbd.
rmse_lambda <- function(lbd){
  date_reg_avg <- train_set %>%
    left_join(movie_reg_avg, by='movieId') %>%
    left_join(user_reg_avg, by="userId") %>%
    left_join(genre_reg_avg, by="genres") %>%
    left_join(year_reg_avg, by="year") %>%
    group_by(unit_date = round_date(date, unit = unit_date)) %>%
    summarize(bts = sum(rating - bm - bu - bg - by - mu)/(lbd + n()))

  # Run the model on the test_set, compare the predicted ratings
  # with the real one and return the RMSE.
  mugyd_preds <- test_set %>%
    left_join(movie_reg_avg, by="movieId") %>%
    left_join(user_reg_avg, by="userId") %>%
    left_join(genre_reg_avg, by="genres") %>%
    left_join(year_reg_avg, by="year") %>%
    mutate(unit_date = round_date(date, unit = unit_date)) %>%
    left_join(date_reg_avg, by="unit_date") %>%
    mutate(pred = mu +  bts + by + bg + bu + bm) %>%
    pull(pred)
  return(RMSE(test_set$rating, mugyd_preds))
}
```

```
# Calculate the date effect using the best lambda identified
# previously for future use.
date_reg_avg <- train_set %>%
  left_join(movie_reg_avg, by='movieId') %>%
  left_join(user_reg_avg, by="userId") %>%
  left_join(genre_reg_avg, by="genres") %>%
  left_join(year_reg_avg, by="year") %>%
  group_by(week = round_date(date, unit = "week")) %>%
  summarize(bts = sum(rating - bm - bu - bg - by - mu)/(500 + n()))

# Apply the previous function to each item in the lambda sequence.
rmse <- sapply(lambda, rmse_lambda)

# Plot the different RMSE for the different values of 'lbd'.
plot(lambda, rmse)
```



For the date effect, the penalisation improves marginally the prediction. Let's update the summary table `model_results`.

```
best_lbd <- lambda[which.min(rmse)]
# Store result in the summary dataframe model_results
model_results <- add_row_results(
  model_nb, model_nb, "m, u, g, y and date effects with reg.", best_lbd, min(rmse))
# Display results
kable(model_results, caption = "Summary of results")
```

Table 13: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168
6	m, u, g, y and date effects with reg.	500.0	0.8640818

The comparison between the RMSE from models 5 to 6 shows little effect on the accuracy of the model (less than 0.03%).

```
# Calculate the improvement between this model and the previous one.
improvement(5,6)
```

```
## [1] 0.02718851
```

Altogether, adding the genre, year and time has improved the model of about 0.08%, most of the effect being already captured in the movie effect.

4.2.7 Conclusion

By adding successively different biases or effects in such a way that all the information available in the dataset was used, the accuracy of the model has been improved. Some biases have more impact on the accuracy of the model than others. The biases with the most impact are the movie and user biases (improvement of more than 18% on the naive mean model).

```
round(improvement(1,3),2)
```

```
## [1] 18.41
```

The genre, year and time stamp biases had little impact on the accuracy of the model (less than 0.1%).

```
round(improvement(3,6),2)
```

```
## [1] 0.08
```

4.3 Matrix factorisation

Now let's see how matrix factorisation performs. Matrix factorisation relies only on the ratings and on user and movie data (the genre, time stamp and release date are not taken into account).

The workflow in r to run a matrix factorisation algorithm with the `recosystem` package is as follows:

1. Model training (building matrix P and Q)
2. Parameter tuning (estimating the best penalty factor λ_P and λ_Q)
3. Exporting the model

4. Making the predictions
5. Validation of the model

To operate, the `recosystem` requires as input a 3-columns matrix corresponding to the users, movies, and ratings. Let's build this. The function `data_memory()` is used to provide input to the recommender system. Help on how to use this function is available using the command `?recosystem::data_memory` in the R console.

```
# Load the corresponding package
library(recosystem)

reco_train <- data_memory(user_index = train_set$userId,
                          item_index = train_set$movieId,
                          rating = train_set$rating,
                          index1 = TRUE)

reco_test <- data_memory(user_index = test_set$userId,
                          item_index = test_set$movieId,
                          index1 = TRUE)
```

Next the model object must be created using the function `Reco()` which returns a object of class RecoSys which makes the methods `$train()`, `$tune()` and `$predict()` accessible. Help on this function is available using the command `?recosystem::Reco`.

```
reco_obj <- Reco()
```

It is now time to train the model with the method `$train()`. Helps on this function (`?recosystem::train`) indicates that the methods uses as input the following arguments:

- `train_data`: an object of class `DataSource` usually returned by `data_memory()`. That's good, since this input is available (`reco_train`),
- `out_model`: the path to the output model file that will be created,
- `opts`: a number of parameters and options for the model training. The algorithm will be run twice, once without prior tuning, and a second time after tuning the model using the `$tune` method.

4.3.1 Training without parameter tuning

For the first attempt, the default options `opts` are used, except for the number of threads, which is set at 4 to lower computing time, and the number of iteration, `niter`, sets at 10 instead of the default value of 20.

```
# Train the algorithm with the default options except for niter
# and nthread, without tuning
reco_obj$train(reco_train, opts = c(nthread = 4, niter = 10))
```

```
## iter      tr_rmse          obj
##    0       0.9706  1.1938e+007
##    1       0.8835  1.0679e+007
##    2       0.8693  1.0595e+007
##    3       0.8500  1.0402e+007
##    4       0.8422  1.0329e+007
##    5       0.8372  1.0286e+007
```

```
##    6       0.8331  1.0260e+007
##    7       0.8301  1.0242e+007
##    8       0.8276  1.0221e+007
##    9       0.8258  1.0208e+007
```

```
# Calculate predicted values with 10 iterations
reco_preds <- reco_obj$predict(reco_test, out_memory())
rmse <- RMSE(reco_preds, test_set$rating)
```

Let's update the summary table.

```
model_nb <- 7
model_results <- add_row_results(
  model_nb, model_nb, "Matrix factorisation, 10 iterations, no tuning", NA, rmse)
# Display results
kable(model_results, caption = "Summary of results")
```

Table 14: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168
6	m, u, g, y and date effects with reg.	500.0	0.8640818
7	Matrix factorisation, 10 iterations, no tuning	NA	0.8371155

Out of curiosity, let's make another attempt with 20 iterations.

```
model_nb <- 8
# Train the algorithm with the default options except for nthread, without tuning
reco_obj$train(reco_train, opts = c(nthread = 4, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9727  1.1960e+007
##    1       0.8820  1.0701e+007
##    2       0.8606  1.0516e+007
##    3       0.8479  1.0377e+007
##    4       0.8427  1.0322e+007
##    5       0.8391  1.0296e+007
##    6       0.8355  1.0271e+007
##    7       0.8320  1.0252e+007
##    8       0.8290  1.0230e+007
##    9       0.8266  1.0212e+007
##   10       0.8249  1.0198e+007
##   11       0.8236  1.0193e+007
##   12       0.8226  1.0182e+007
##   13       0.8217  1.0176e+007
##   14       0.8211  1.0171e+007
##   15       0.8206  1.0169e+007
```

```
##   16       0.8201  1.0164e+007
##   17       0.8197  1.0160e+007
##   18       0.8193  1.0157e+007
##   19       0.8190  1.0153e+007
```

```
# Calculate predicted values
reco_preds <- reco_obj$predict(reco_test, out_memory())
rmse <- RMSE(reco_preds, test_set$rating)
# Update model_results
model_results <- add_row_results(
  model_nb, model_nb, "Matrix factorisation, 20 iterations, no tuning", NA, rmse)
# Display results
kable(model_results, caption = "Summary of results")
```

Table 15: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168
6	m, u, g, y and date effects with reg.	500.0	0.8640818
7	Matrix factorisation, 10 iterations, no tuning	NA	0.8371155
8	Matrix factorisation, 20 iterations, no tuning	NA	0.8338880

By increasing the number of iteration from 10 to 20 the accuracy of the model is increased by about 1%

```
# Calculate the improvement between this model and the previous one.
improvement(7, 8)
```

```
## [1] 0.3855549
```

4.3.2 Training with parameter tuning

This time the `$tune` method is used to try and improve the accuracy of the model. For detailed information about the parameters of the function, type `?recosystem::tune`. This step takes time to compute, so be patient.

```
# Run the tuning function to generate optimised parameters. This takes time.
opts <- reco_obj$tune(reco_train, opts = list(nthread  = 4, niter = 10))

# Save the opts object for future use. Creates "data" repository
# in the current directory if doesn't exist.
ifelse(!dir.exists(file.path("data")), dir.create(file.path("data")), FALSE)
saveRDS(opts, file = "data/reco-opts.rds")
```

The model is trained with the resulting tuning parameters in `opts$min`.

```
model_nb <- 9
# Train the algorithm with the default options except for nthread, with tuning
reco_obj$train(reco_train, opts = c(opts$min, nthread = 4, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9861  9.8949e+006
##    1       0.8794  8.0777e+006
##    2       0.8498  7.5286e+006
##    3       0.8313  7.2102e+006
##    4       0.8160  6.9765e+006
##    5       0.8036  6.8008e+006
##    6       0.7936  6.6652e+006
##    7       0.7856  6.5606e+006
##    8       0.7788  6.4733e+006
##    9       0.7730  6.4005e+006
##   10       0.7681  6.3413e+006
##   11       0.7637  6.2879e+006
##   12       0.7599  6.2440e+006
##   13       0.7564  6.2037e+006
##   14       0.7533  6.1713e+006
##   15       0.7505  6.1402e+006
##   16       0.7481  6.1146e+006
##   17       0.7458  6.0902e+006
##   18       0.7437  6.0690e+006
##   19       0.7418  6.0488e+006
```

```
# Calculate predicted values
reco_preds <- reco_obj$predict(reco_test, out_memory())
rmse <- RMSE(reco_preds, test_set$rating)
# Update model_results
model_results <- add_row_results(
  model_nb, model_nb, "Matrix factorisation, 20 iterations, tuning", NA, rmse)
# Display results
kable(model_results, caption = "Summary of results")
```

Table 16: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168
6	m, u, g, y and date effects with reg.	500.0	0.8640818
7	Matrix factorisation, 10 iterations, no tuning	NA	0.8371155
8	Matrix factorisation, 20 iterations, no tuning	NA	0.8338880
9	Matrix factorisation, 20 iterations, tuning	NA	0.7950266

By tuning parameters the accuracy of the model is increased by more than 4.6%.

```r
# Calculate the improvement between this model and the previous one.
improvement(8, 9)
```

```
## [1] 4.660269
```

5 Results

This section presents the results and discusses the performance of the two models. The **validation set** is now used to estimate these performances. Because this set has not been used so far, these performances can be considered as unbiased estimates of how the models would perform on new data.

5.1 Linear model

Let's apply the linear model to the varaibles in the validation set and compare the prediction $\hat{Y}$ with the real ratings Y of the dataset.

```r
validation <- validation %>% mutate(date = as_datetime(timestamp))

# Apply all the biases to the validation set.
mugyd_preds <- validation %>%
  left_join(movie_reg_avg, by="movieId") %>%
  left_join(user_reg_avg, by="userId") %>%
  left_join(genre_reg_avg, by="genres") %>%
  left_join(year_reg_avg, by="year") %>%
  mutate(week = round_date(date, unit = "week")) %>%
  left_join(date_reg_avg, by="week") %>%
  mutate(pred = mu +  bts + by + bg + bu + bm) %>%
  pull(pred)

# Calculate the RMSE.
rmse_LM <- RMSE(validation$rating, mugyd_preds)
rmse_LM
```

```
## [1] 0.8649501
```

The linear model provides a RMSE of 0.86495 on the validation set.

```r
# Update model_results
model_nb <- 10
model_results <- add_row_results(
  model_nb, model_nb, "Full linear model, validation set", NA, rmse_LM)
kable(model_results, caption = "Summary of results")
```

Table 17: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678

model_nb	type	lambda	rmse
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168
6	m, u, g, y and date effects with reg.	500.0	0.8640818
7	Matrix factorisation, 10 iterations, no tuning	NA	0.8371155
8	Matrix factorisation, 20 iterations, no tuning	NA	0.8338880
9	Matrix factorisation, 20 iterations, tuning	NA	0.7950266
10	Full linear model, validation set	NA	0.8649501

5.2 Matrix factorisation

```
# Prepare the validation data in the suitable format.
reco_validation <- data_memory(user_index = validation$userId,
                               item_index = validation$movieId,
                               index1 = TRUE)

# Run predictions using the MF model.
reco_preds <- reco_obj$predict(reco_validation, out_memory())

# Calculate RMSE
rmse_MF <- RMSE(reco_preds, validation$rating)
rmse_MF
```

```
## [1] 0.7951512
```

The RMSE of the matrix factorisation model, when applied to the validation set, is 0.79515.

```
# Update model_results
model_nb <- 11
model_results <- add_row_results(
  model_nb, model_nb, "Matrix factorisation, validation set", NA, rmse_MF)
kable(model_results, caption = "Summary of results")
```

Table 18: Summary of results

model_nb	type	lambda	rmse
1	naive mean	NA	1.0598986
2	movie effect with regularisation	3.0	0.9434985
3	movie and user effects with reg.	4.5	0.8647678
4	movie, user and genre effects with reg.	20.0	0.8644598
5	m, u, g and year effects with reg.	0.0	0.8643168
6	m, u, g, y and date effects with reg.	500.0	0.8640818
7	Matrix factorisation, 10 iterations, no tuning	NA	0.8371155
8	Matrix factorisation, 20 iterations, no tuning	NA	0.8338880
9	Matrix factorisation, 20 iterations, tuning	NA	0.7950266
10	Full linear model, validation set	NA	0.8649501
11	Matrix factorisation, validation set	NA	0.7951512

The difference between the two models applied to the validation set is significant, at 8.1%.

```
round(improvement(10, 11),3)
```

```
## [1] 8.07
```

6 Conclusion

In this tutorial, the typical workflow of a data-science project has been introduced and followed. Two models have been developed, trained, tuned and applied to the publicly available "MovieLens 10M" dataset to make movie predictions.

The first model, a **linear regression** with regularisation applied to categorical features (movie, user, genre, year of movie release, date of rating), has shown that the variables `userId` and `movieId` were the most influencial to estimate ratings. The genre, year and date contributed to a much smaller extent. This can be explained by the fact that the movie variable already includes most of the effect of year and genres (the year and genre effects are already included in the movie effect which has been calculated sooner). An easy way to test this hypothesis is to re-run the linear model by starting with the genre bias instead of the movie bias, and to measure the improvement from the naive model. Doing so would show that introducing the genre bias first causes a 4% improvement, confirming this hypothesis.

The second model, based on **matrix factorisation**, proved more accurate at predicting ratings. However, it is a black box and this approach did not provide any insight about how the various features in the dataset influence ratings. The main reason why this model is more accurate is largely because every observation in the dataset is considered an input for any prediction. In the linear models, only aggregated values (averages) where used for the predictions.

Both models used only historical data. It is possible to improve further the prediction of a recommender model by adding other information about the viewers, such as their gender, ethnicity, age and location, which most current recommender models do.

In the section "reminders", two important functions of machine learning models have been introduced: *inference* and *prediction.* From the point of view of error comparison, it is clear that matrix factorisation does a much better job at predicting movie ratings. However, it is a complete black box: the model provides no easily accessible indication of what influences these predictions. On the other hand the linear model is not as good at predicting ratings, but it is much easier to interpret. The user can easily see what impacts movie ratings the most: movie, user, or any of the other predictors. In most cases prediction accuracy is the most important criteria, but this is not always the case. In the banking industry, for example, where machine learning algorithms can be used to predict the likelyhood of default before authorising a loan to a client, the legislations of some countries impose that these machine learning algorithms be easily understandable by any human intelligence, for transparency and fairness reason: the bankers must be in a position to explain clearly why a client has not be offered a loan.