Movielens 10M

Jacopo Trabona

12/22/2020

Introduction

Recommender systems have become one of most popular application of machine learning.

The ubiquity of their implementation in the commercial sphere, together with a wide plethora of data available for training, have gradually led them to reach an emblematic status across both the specialised community and the general audience.

In few words, their main purpose is that of using historical data to **predict** a given user's – or subset of users' – level of satisfaction towards the item at hand.

In this capstone report, we will be creating a movie recommendation system working with **MovieLens**, a widely used movie rating dataset developed in 1997 by the GroupLens Research lab at the University of Minnesota. Specifically, we will be focusing on one of the largest versions of MovieLens, the **MovieLens 10M** Dataset, which is entirely open source, and can be easily downloaded from the GroupLens official website.

Preliminary Considerations

As reported by GroupLens, "the MovieLens 10M Dataset has 10 million ratings and 100,000 tag applications applied by 72,000 users". Presumably, this will make computation rather challenging on consumer-grade machines such as the one used for the purpose of this report. Furthermore, this computational burden is likely to affect both the data analysis and the model development stages.

Bearing this in mind, we now start by downloading the dataset and installing a set of R packages.

Data Preparation

We first load the required packages. We will be making use of widely adopted R libraries, such as **tidyverse** and **caret**, as well as more specific ones, such as the matrix factorization package **recosystem** and the correlation visualization library **ggcorrplot**.

```r
if(!require(tidyverse))  install.packages("tidyverse")
if(!require(caret))      install.packages("caret")
if(!require(lubridate))  install.packages("lubridate")
if(!require(data.table)) install.packages("data.table")
if(!require(hrbrthemes)) install.packages("hrbrthemes")
if(!require(ggcorrplot)) install.packages("ggcorrplot")
if(!require(recosystem)) install.packages("recosystem")
```

```r
library(tidyverse)
library(caret)
library(lubridate)
library(data.table)
library(hrbrthemes)
library(ggcorrplot)
library(recosystem)
```

Then, we download the **dataset**, and split it into two sets, the **edx** and the **validation** set, respectively. The actual project will be developed using **edx**, whereas the **validation** set (10% of the total dataset) will be used to confirm the validity of the final machine learning model.

Here, we decide to include only relevant chunks of our code.

```r
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,]
```

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

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

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

Data Cleansing and Variables Extraction

By looking at the first ten entries of the **edx** set - as well as its structure - we can state that data appear in a **tidy format**, with **6 variables** and **9000055 observations**.

userId	movieId	rating	timestamp	title	genres
1	122	5	838985046	Boomerang (1992)	Comedy|Romance
1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller
1	316	5	838983392	Stargate (1994)	Action|Adventure|Sci-Fi
1	329	5	838983392	Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-Fi
1	355	5	838984474	Flintstones, The (1994)	Children|Comedy|Fantasy

However, we immediately notice that the **timestamp** column, the column where information about the review date and time is stored, needs to be converted into readable format. We decide to keep only the information regarding the **review year**, in that presumably less prone to random variability.

```r
review_date <- as_datetime(edx$timestamp) %>% date() %>% year()
```

We also notice that the characters in the title column include the release date of each respective movie.

We decide to extract them for later use, and include a new **release_date** variable into the **edx** dataset, along with a new **review_date** column.

title	genres	release_date	review_date
Boomerang (1992)	Comedy|Romance	1992	1996
Net, The (1995)	Action|Crime|Thriller	1995	1996
Outbreak (1995)	Action|Drama|Sci-Fi|Thriller	1995	1996
Stargate (1994)	Action|Adventure|Sci-Fi	1994	1996
Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-Fi	1994	1996
Flintstones, The (1994)	Children|Comedy|Fantasy	1994	1996

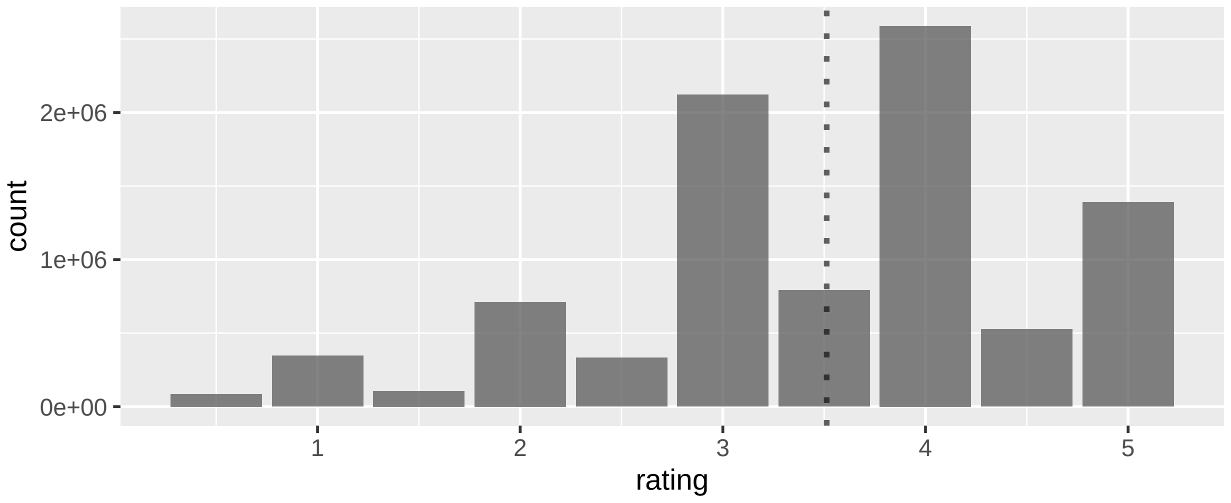
Lastly, we further split the resulting data frame into a **train_set** and a **test_set**, which we will be using to train and test our models, respectively.

```r
#Create index
tst_index <- createDataPartition(y = edx_with_dates$rating, times = 1, p = 0.2,
                                 list = FALSE)

#Create train_set / test_set split
train_set <- edx_with_dates %>% slice(-tst_index)
test_set <- edx_with_dates %>% slice(tst_index)

#Make sure all movies and users in the test_set are also in the train_set
test_set <- test_set %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")
```

Exploratory Data Analysis

Generally speaking, in movie recommendation systems, the main goal is that of predicting how users - or a subset of users - would rate films that have been already rated by others. We will be thereby assuming the **rating** variable as the **outcome Y** in all the models we are going to construct.

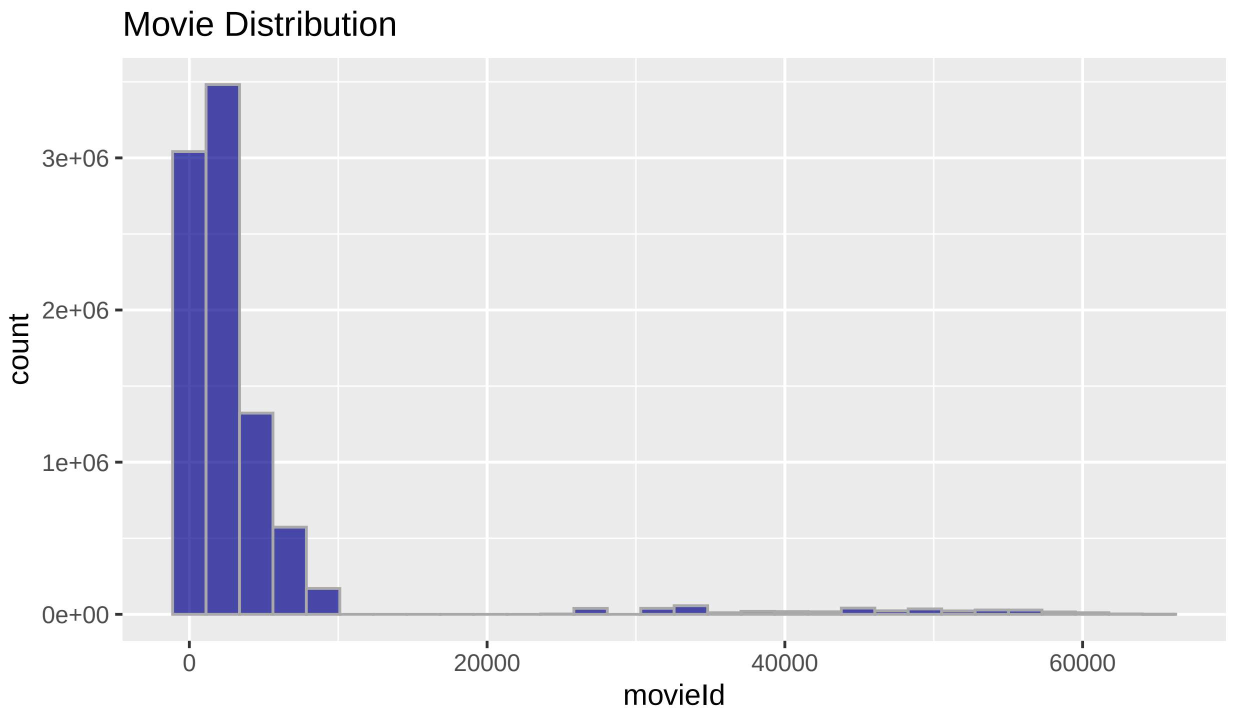
Accordingly, we start our **EDA** calculating the **mean** across all rating values **(3.512465)**. Then we study its distribution with a bar chart, just to confirm that rating values are discrete and span from 0.5 to 5.

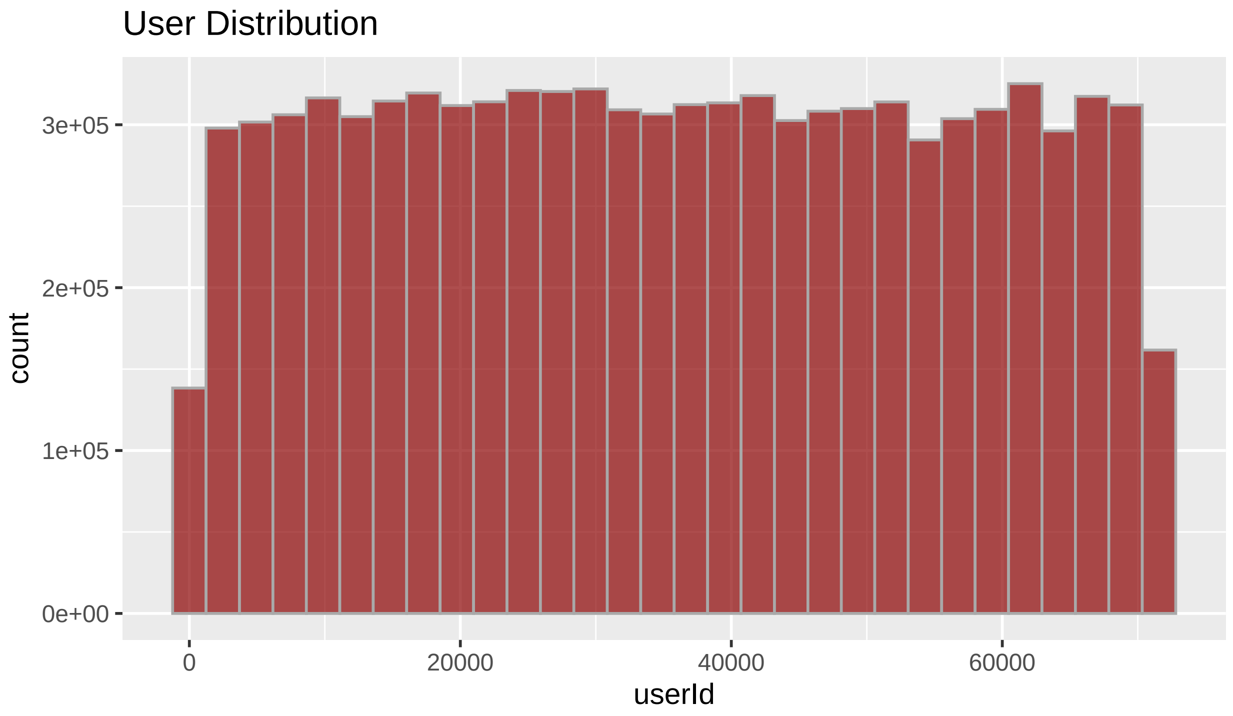


By definition, ratings are necessarily linked to both the users who produced them and the movies to which they were applied.

Thus, we decide to proceed by counting the number of **unique users (69878)** and **unique movies (10677)** in our dataset, and by studying the distribution of both **movieId** and **userId**.

From now on, we will be using **different colours** to plot the different variables we are in the process of exploring.





Additionally, we arrange **most active users** and **most rated movies** in descending order.

```
##   userId count
## 1  59269  6616
## 2  67385  6360
## 3  14463  4648
## 4  68259  4036
## 5  27468  4023
## 6  19635  3771
```

```
##   movieId                            title count
## 1     296              Pulp Fiction (1994) 31362
## 2     356              Forrest Gump (1994) 31079
## 3     593 Silence of the Lambs, The (1991) 30382
## 4     480             Jurassic Park (1993) 29360
## 5     318 Shawshank Redemption, The (1994) 28015
## 6     110                Braveheart (1995) 26212
```

The most relevant findings are that, given the overall number of observations, even **most active users** have rated no more than **60%** of the movies in the dataset, and that, correspondingly, even the **most rated movies** were rated by less than half of the total users.

Methods

Considering the aforementioned findings, our task promptly takes the shape of the typical **collaborative filtering (CF)** problem, where future associations between users and items must be inferred from the observed ones, and where the dataset can be thought of as a **sparse matrix** with a large number of **empty** cells to predict.

In other words, we should expect to deal with an **high dimensional feature space**, where popular kernel methods, such as **k-nearest neighbor** and **local regression** might prove ineffective.

Additionally, in the specific context of our **10 M dataset**, also more suitable approaches, such as **regression trees**, don't seem to be a viable option on our machine, given the sheer volume of the computation to be undertaken.

Thus, building on Professor Rafael Irizarry's take on a similar dataset, we decide to proceed by building a series of increasingly complex **linear models**, where the complexity is given by a growing number of **effects** (biases) that will be progressively added to a base model of the form $Yu, i = \mu + \epsilon i, u$.

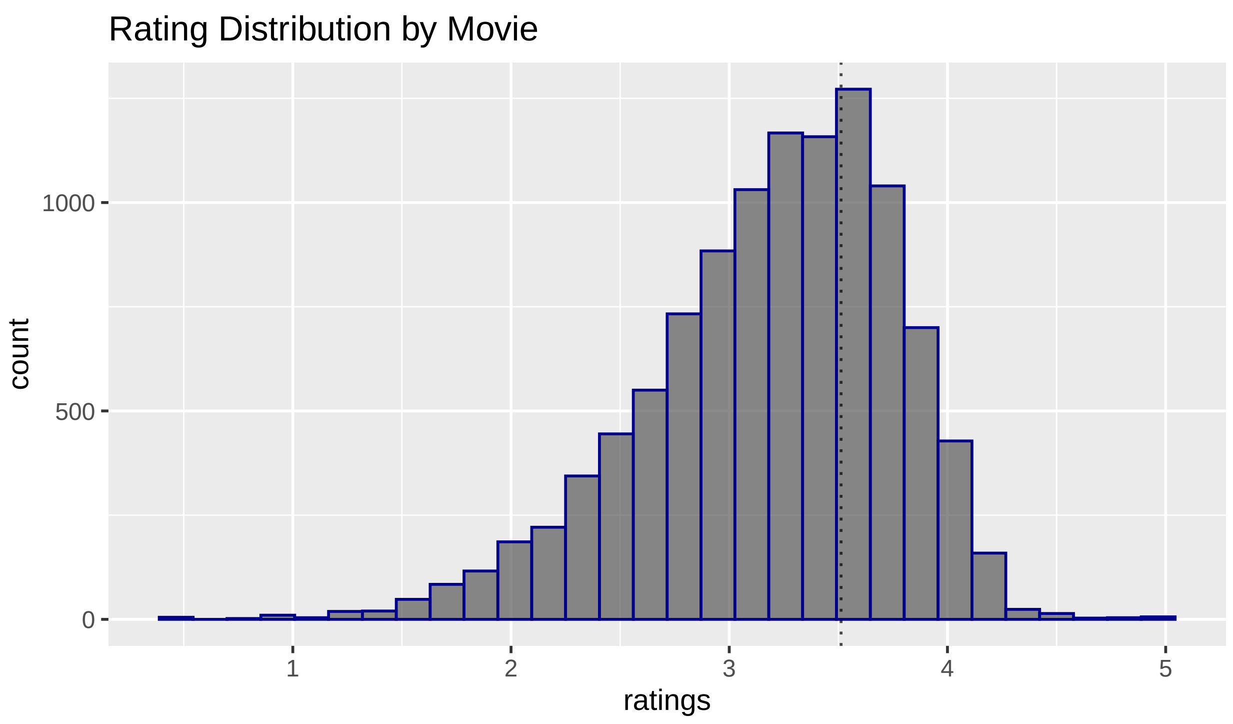
In other words, if we consider Y, ui to be our predictions,   the average rating value, and $\epsilon i, u$ the random error in our data, we will be adding a series of terms $bi, bu, ..., bk$ such that the variability in our observations is explained in an increasingly accurate manner. We will be using **relevant variables** in the edx dataset to do so.

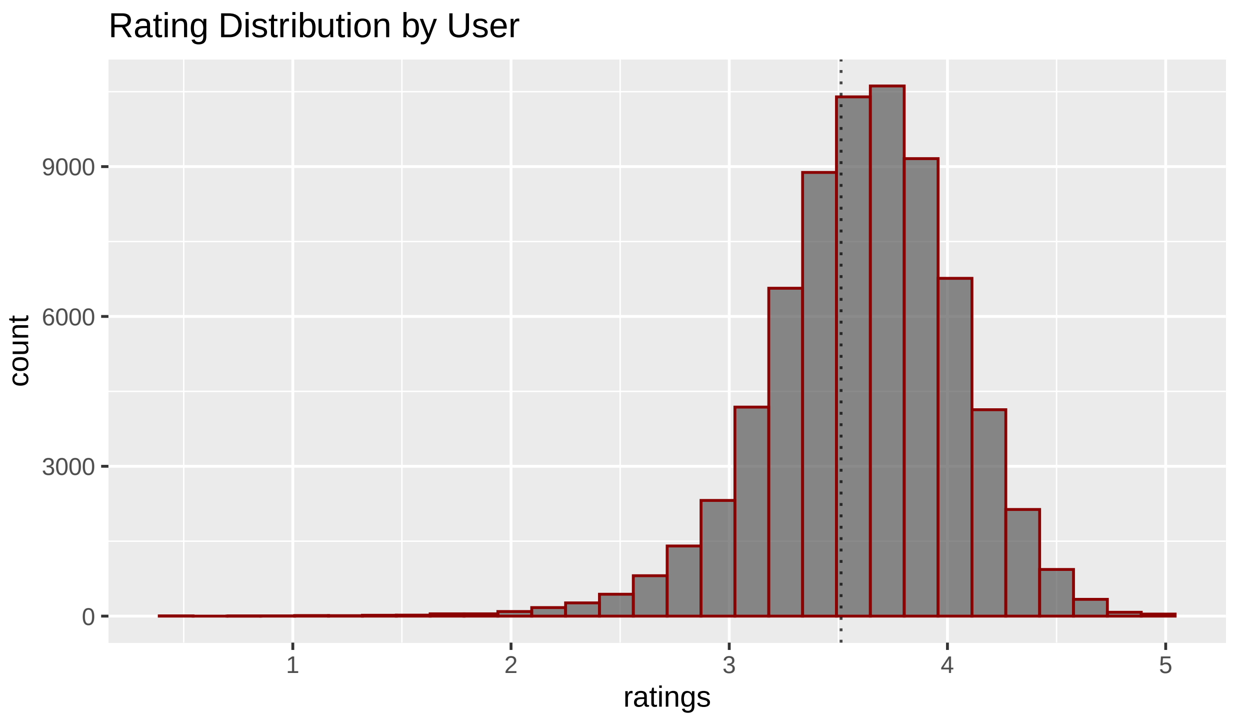
Lastly we will explore one of the most popular approaches in **CF** recommendation systems, **matrix factorization**, by making use of the R package **recosystem** in order to deal with the demanding volume of our data.

Model 1 (Movie + User)

Before we start to build our models, we will need to establish whether a potential effect is actually present in our data.

Thus, we decide to study the distribution of **ratings** when grouped by **movieId** and **userId**.





From the resulting plots, we notice **different rating patterns** for both different movies and different users. As expected, this is probably due to the fact that some movies are generally better received than others, while some users are simply more keen to give higher ratings.

Accordingly, we can easily confirm the presence of both a **movie effect** bi and an **user effect** bu.

Our first linear model will thereby look as follows: $Yu, i = \mu + bi + bu + \epsilon i, u$.

Of course, in order to implement our model, we now need to estimate both effects. However, whereas we would normally use the function **lm()** from the **caret** package to fit our model, applying **lm()** to this specific case would lead R to crash.

Thus, throughout the implementation of the following **linear models**, we will be making use of a set of **tidyverse** functions, in order to estimate $bi, bu, ..., bk$ as the difference between the observed ratings y, the average rating µ, and the previously added $bi, bu, ..., bk$.

We will then repeat this same process for all the **biases** in the respective models.

The code for our first model looks as follows:

```
#Movie + User effects

mu_hat <- mean(train_set$rating)

movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarise(bi = mean(rating - mu_hat))

user_avgs <- train_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  group_by(userId) %>%
  summarise(bu = mean(rating - mu_hat - bi))

preds <- test_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  mutate(preds = mu_hat + bi + bu) %>%
  pull(preds)
```

Once this code is implemented, we use the **Root Mean Squared Error (RMSE)** between our **predictions** and the **observed values** to evaluate our model.

```
RMSE(preds, test_set$rating)
```

```
## [1] 0.865932
```

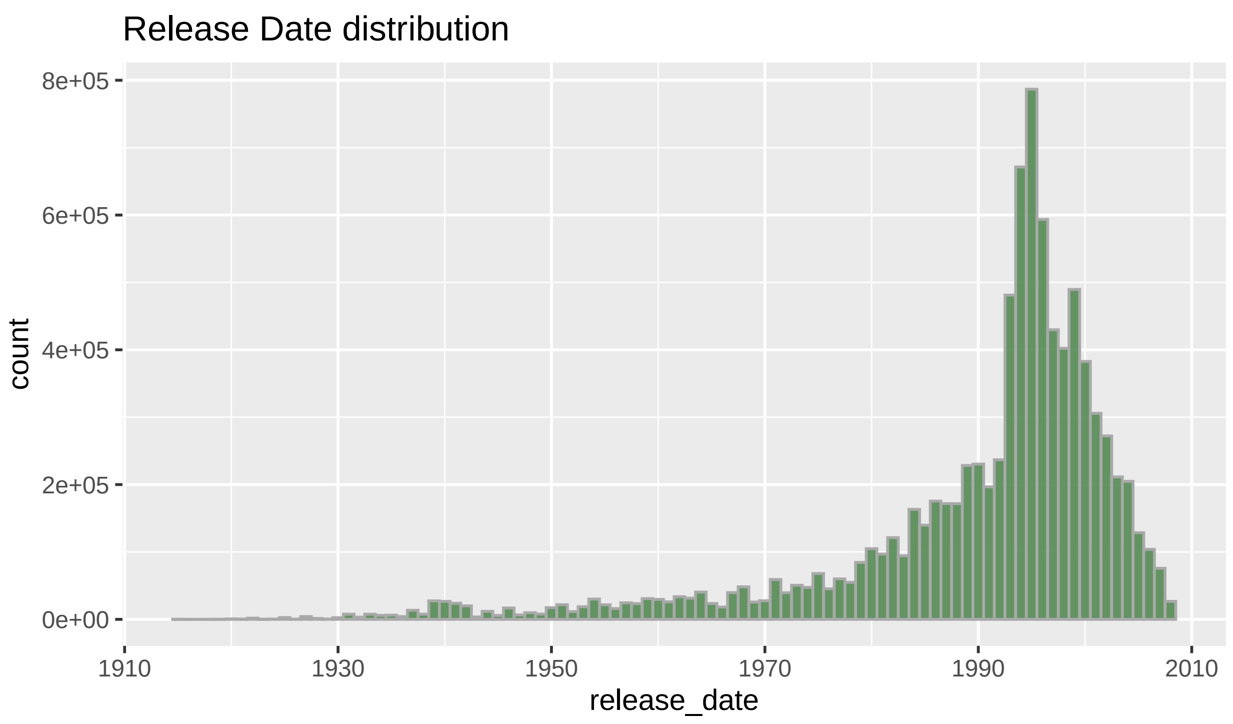
We then create a **data table** to store the resulting **RMSES**, and also include a column named **Regularized** - which will specify whether the model has been regularized or not - and a **Cross Validation (CV)** column - which will specify which type of cross validation has been implemented.

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.865932

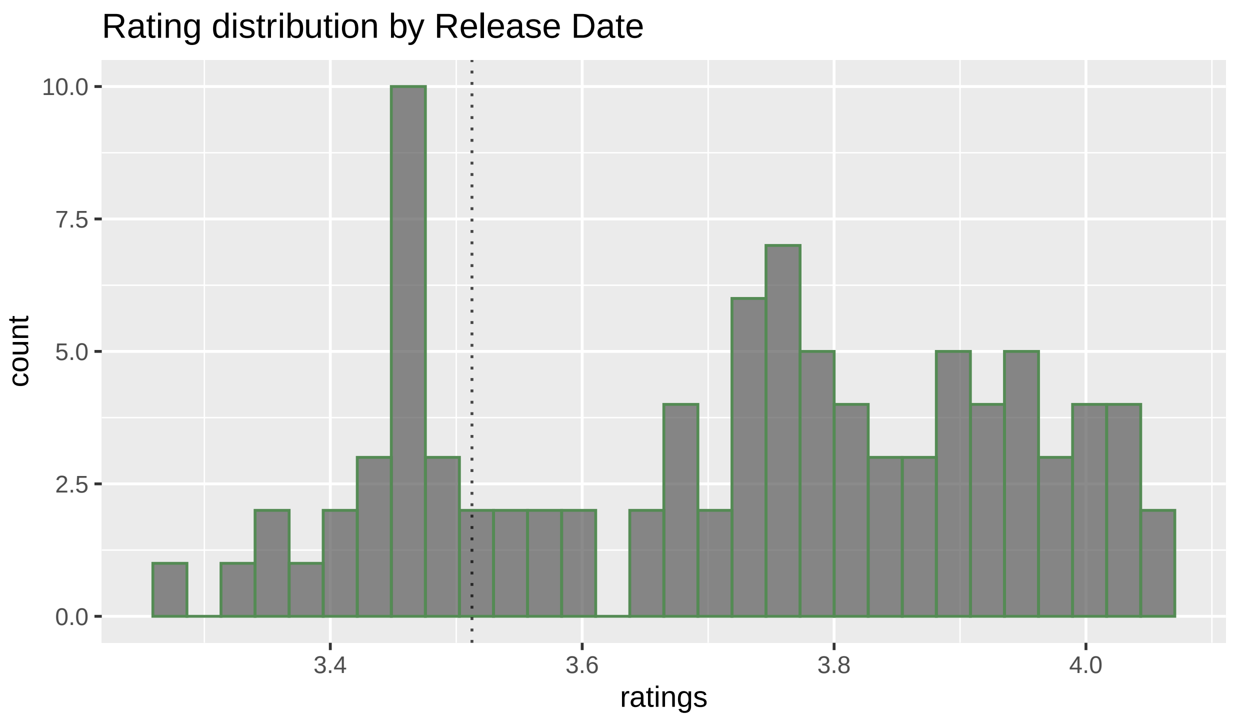
Model 2 (Movie + User + Release date)

Our first model produced rather satisfying results. However, looking at the available data, we can surely examine other **variables** and test them for potentially significant **effects**. We start with the **release_date** variable.

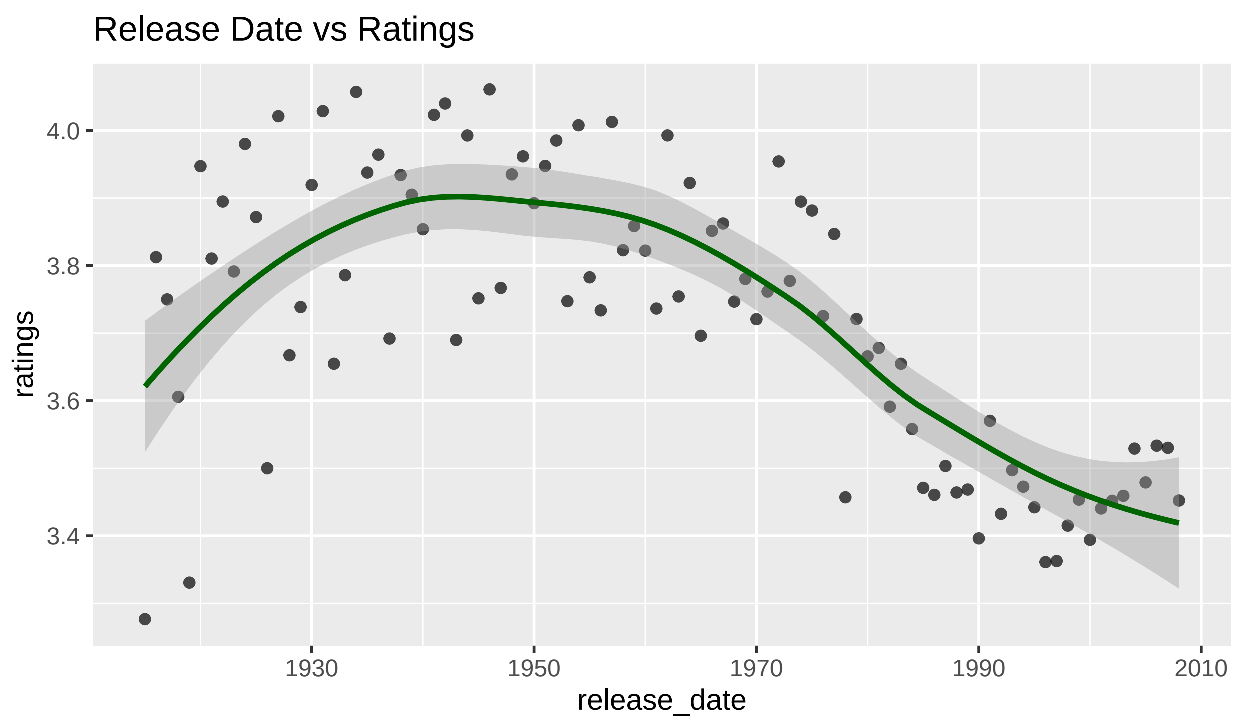
Thus, we first create an **histogram** of its distribution.



Then, we explore the **rating** distribution grouping by **release_date**.



This **histogram** shows evidence of a **release_date effect**. We create a **scatter-plot** to confirm that.



We can see that ratings seem to be generally affected by the year a given movie was released. This might be due to the fact that some users prefer older movies, while some other prefer more recent ones - or, simply, to the fact that, in general, in some good years better movies were release.

Thus, we proceed by adding a **release_date** term *bd* to our model, which now looks as follows: $Yu, i = \mu + bi + bu + bd + \epsilon i, u$.

We implement our **tidyverse** approach to obtain the model predictions and resulting **RMSE**.

```
#Movie + User + Release Date Effects

#Model
date_avgs <- train_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  group_by(release_date) %>%
  summarise(bd = mean(rating - mu_hat - bi - bu))

#Predictions
preds <- test_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  left_join(date_avgs, by = "release_date") %>%
  mutate(preds = mu_hat + bi + bu + bd) %>%
  pull(preds)

RMSE(preds, test_set$rating)
```

```
## [1] 0.8656117
```

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117

Model 3 (Movie + User + Release date) * Regularized

Results further improved with our last model. However, from our previous exploration of the variables' distributions, we can assume the presence of a series of observations that are **significantly distant** from the mean, and which, accordingly, might vex the accuracy of our model.

In other words, movies that were rated by only a few users, users who rated only a few movies, and years in which movies were rated only a few times, all concur to drive up **variability** in a way that is currently not accounted for.

Accordingly, we decide to employ a popular approach, **penalized regression**, whereby we will be **regularizing** the variables by adding a **penalty parameter** λ. In doing so, we proportionally curb the weight of the **observations** as they progress away from the mean.

Our assumption is that the resulting model will be able to explain a **much wider portion of variability**.

In order to facilitate computation and reproducibility, we create a **function** for such purpose, in fact formalizing our **tidyverse** approach.

The **function** takes λ **(l)**, a given **train set**, and a given **test set** as its arguments.

```r
#Write a regularization function for the model
regularization <- function(l, tr, ts) {

  #model
  mu_hat <- mean(tr$rating)

  bi <- tr %>%
    group_by(movieId) %>%
    summarise(bi = sum(rating - mu_hat) / (n() + l))

  bu <- tr %>%
    left_join(bi, by = "movieId") %>%
    group_by(userId) %>%
    summarise(bu = sum(rating - bi - mu_hat) / (n() + l))

  bd <- tr %>%
    left_join(bi, by = "movieId") %>%
    left_join(bu, by = "userId") %>%
    group_by(release_date) %>%
    summarise(bd = sum(rating - bi - bu - mu_hat) / (n() + l))

  #predictions
  predictions <- ts %>%
    left_join(bi, by = "movieId") %>%
    left_join(bu, by = "userId") %>%
    left_join(bd, by = "release_date") %>%
    mutate(preds = mu_hat + bi + bu + bd) %>%
    pull(preds)

  return(RMSE(predictions, ts$rating))
}
```

Since λ can be considered a **tuning parameter** in our tidyverse algorithm, we must asses several values of λ in order to pick the one producing the best results.

Here, we use **cross validation (CV)** to do so. Specifically, we build a **k-fold** cross validation function, which, considering the large size of the dataset, we decide to implement across **5 folds**.

We first create our **folds index**. Since we don't need our **CV** to be **stratified**, we use **caret createDataPartition()**.

```
#Create cross validation folds
set.seed(1, sample.kind = "Rounding")
folds <- createDataPartition(train_set$rating, times = 5, p = 0.2, list = FALSE)
```

Then, we create our **5-fold cross validation function**, including the previously defined **regularization function**. We call it **best_lambda**.

```
#Create a 5 fold cross validation function
best_lambda <- function(l) {
  results <- vector("numeric", 5)
  for (i in 1:5) {
    tr <- train_set[-folds[,i]]
    ts <- train_set[folds[,i]]
    ts <- ts %>%
      semi_join(tr, by = "movieId") %>%
      semi_join(tr, by = "userId")

    errors <- regularization(l, tr, ts)
    results[i] <- errors
    }
  results
}
```

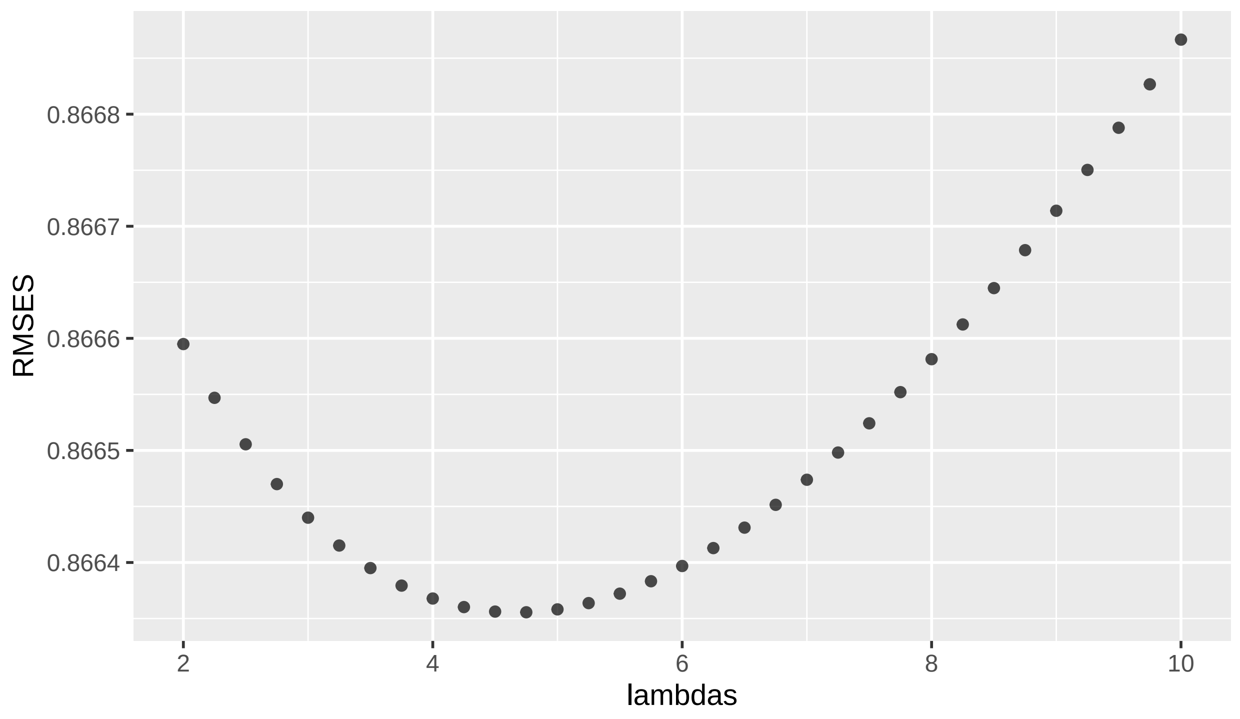
Now we are ready to apply the **best_lambda()** to several different values of λ, which we define as follows:

```
#Define a range of possible lambdas
lambdas <- seq(2, 10, 0.25)
```

We run **best_lambda()** for all defined values of λs of using **sapply()**, and then build a **data frame** containing the **means** of the resulting **RMSES** computed in each fold.

```
#Apply cross validation to models with varying lambda
res <- sapply(lambdas, best_lambda)

#Results data frame
pick_lambda <- data.frame(lambdas = lambdas, RMSES = colMeans(res))
```

Here is a **plot** summarizing how different λs behaved.



Finally, we use the initial **train set** and **test set** to re-run our **regularized model** with the best value of λ we have just obtained.

```
##   lambdas     RMSES
## 1    4.75 0.8663556
## 2    4.50 0.8663562
## 3    5.00 0.8663582
## 4    4.25 0.8663602
## 5    5.25 0.8663638
## 6    4.00 0.8663679
```

```
#Re-run model with the best lambda
regularization(4.75, train_set, test_set)
```

```
## [1] 0.8649761
```

And we add the resulting **RMSE** to the **results table**.

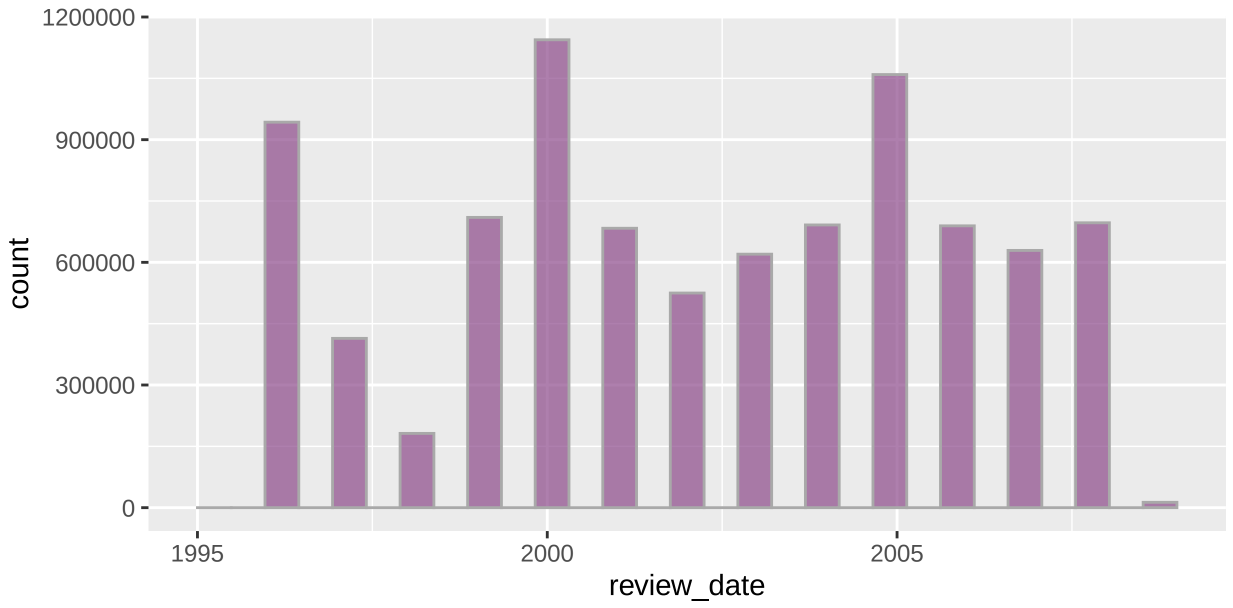
method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117
3 Movie + User + Release Date	YES	5-Folds (Lambda)	0.8649761

We can safely state that **regularization** definitely improved how the model fits our data. Accordingly, from now on we will be using **regularization** for **all the models** and biases we are going to explore.

Model 4 (Movie + User + Release Date + Review Date)

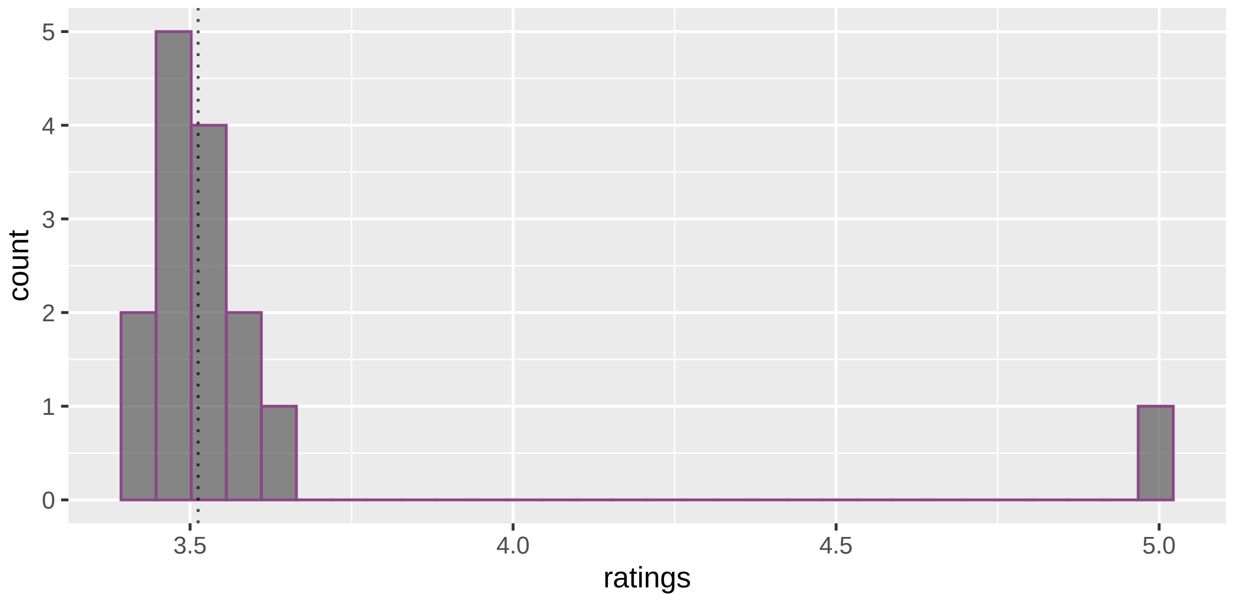
We proceed by studying the variable **review_date**, which we previously obtained converting the **timestamp** column in the original dataset.

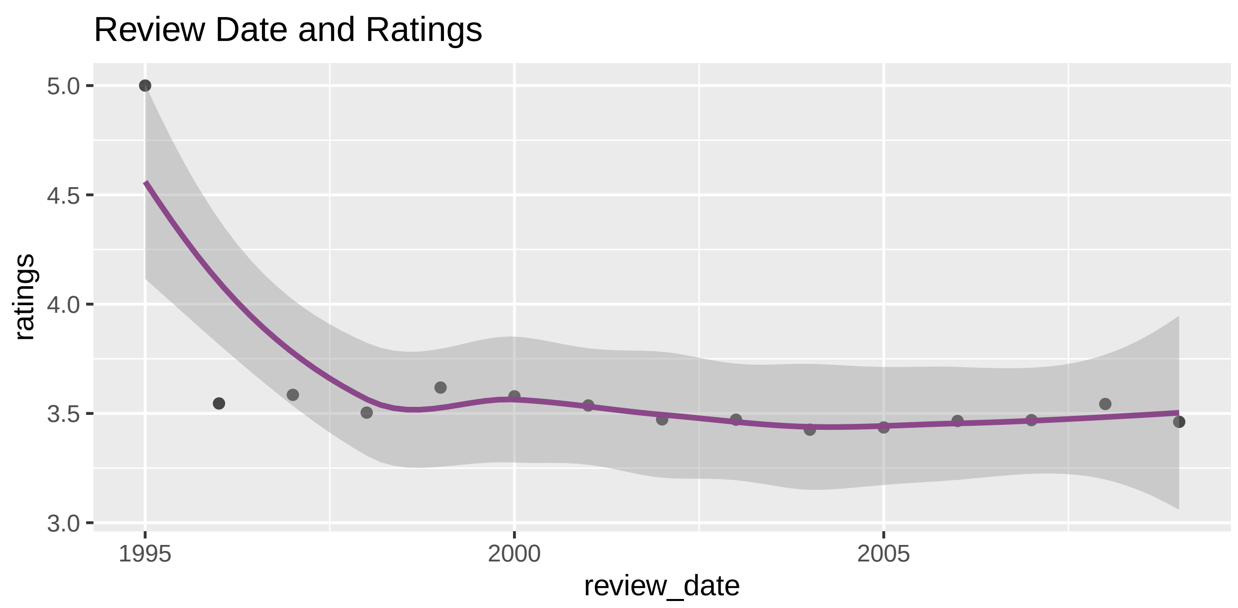
First, we **plot** its own **distribution**.



We see that there is **variability** across the number of reviews redacted in different years - with some significantly distant observations that make **regularization** a suitable approach.

Thus, we plot **ratings distribution** grouping by **review_date** to check for a **review_date effect**, and then confirm that by using a **scatter-plot**.





Evidence of a **review_date** effect are provided, although **not as strongly** as we might have expected. Nevertheless, we decide to include **review_date** in our regularized model to see whether it marginally helps improve our results.

Thus, we adjust our **regularization function** and then use **5-fold cross validation** to pick the best value of λ.

Since the process is virtually a repetition of what we have already shown above, here we won't include the full code.

Lastly, once we have determined the best suiting value of λ, we employ it to fit our **regularized model** and obtain the resulting **RMSE**.

```
#Re-run model with the best lambda
regularization_rw(5, train_set, test_set)
```

```
## [1] 0.8648612
```

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117
3 Movie + User + Release Date	YES	5-Folds (Lambda)	0.8649761
4 Movie + User + Release Date + Review Date	YES	5-Folds (Lambda)	0.8648612

In fact, our **RMSE** did **improve** by a small - yet non-negligible - margin.

Model 5 (Movie + User + Release Date + Review Date + Genres)

The **genres** column in our dataset is more peculiar than the **release_date** and **review_date** ones we have just explored.

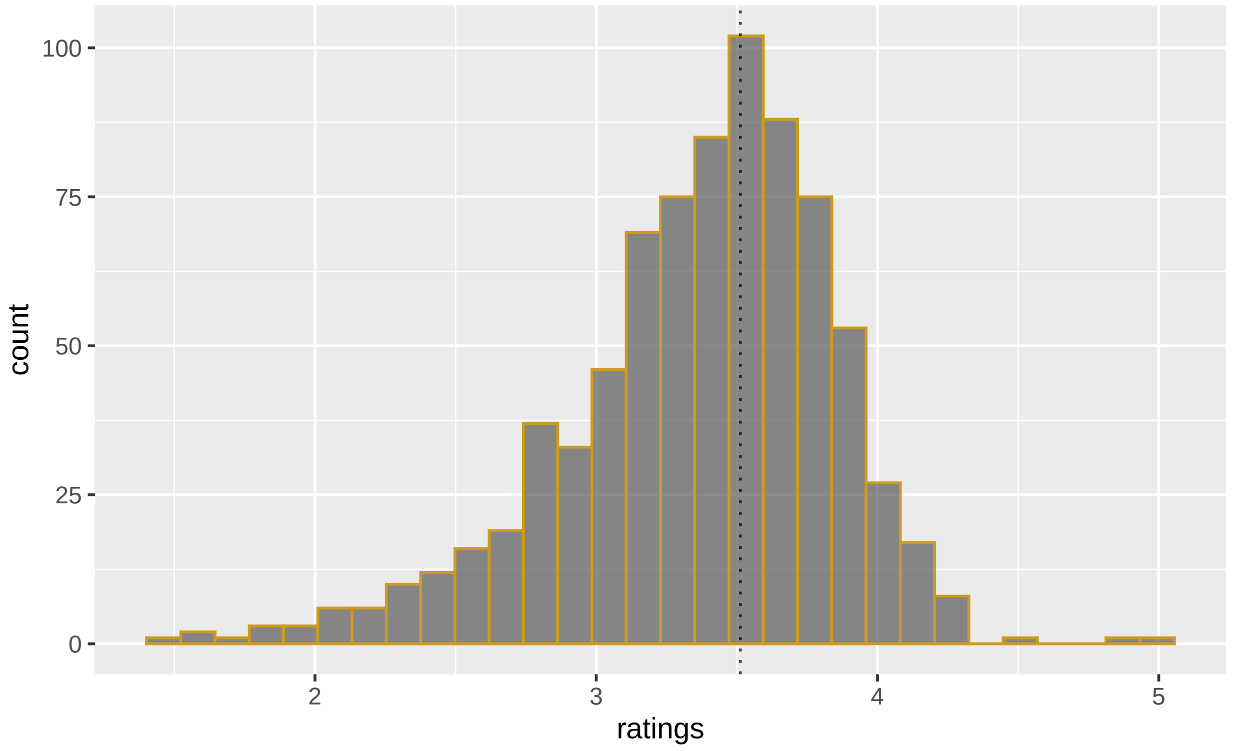
```r
head(train_set$genres)
```

```
## [1] "Action|Crime|Thriller"
## [2] "Action|Adventure|Sci-Fi"
## [3] "Action|Adventure|Drama|Sci-Fi"
## [4] "Children|Comedy|Fantasy"
## [5] "Adventure|Animation|Children|Drama|Musical"
## [6] "Action|Romance|Thriller"
```

Firstly, by its very nature, it is more subject to the **arbitrary decisions** made by those compiling the original data - and assigning a given genre to a given movie. Most probably, that is why the GroupLens research team decided to include small sets of **multiple genres** to describe each movie. But did it actually reduce the arbitrariness?

Secondly, there are many more **unique observations**, namely **797**, which are as many as - roughly - **7%** of all unique movies in our **train set**. On the other hand, **Release_date** and **review_date** have only **94** and **15** unique entries, respectively.

Thus, we start by studying a potential **effect** on **ratings** grouping by **genre**.



We see that our model might benefit from a **genre_effect** term, and then proceed by adjusting our **regularization** and **cross validation functions**. Again, we decide not to show the full code.

```r
#Re-run model with the best lambda
regularization_genres(5, train_set, test_set)
```

```
## [1] 0.8645972
```

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117
3 Movie + User + Release Date	YES	5-Folds (Lambda)	0.8649761
4 Movie + User + Release Date + Review Date	YES	5-Folds (Lambda)	0.8648612
5 Movie + User + Release date + Review Date + Genres	YES	5-Folds (Lambda)	0.8645972

Our **RMSE** decreased significantly as compared to the improvement obtained in **model 3** and **model 4**. This means that **genres** is a rather **descriptive** variable. However, as previously stated, it is one that is prone to the influence of the authors who labeled the movies.

Accordingly, we want to check whether **combining genres** actually reduced arbitrariness - and therefore, presumably, random variability.

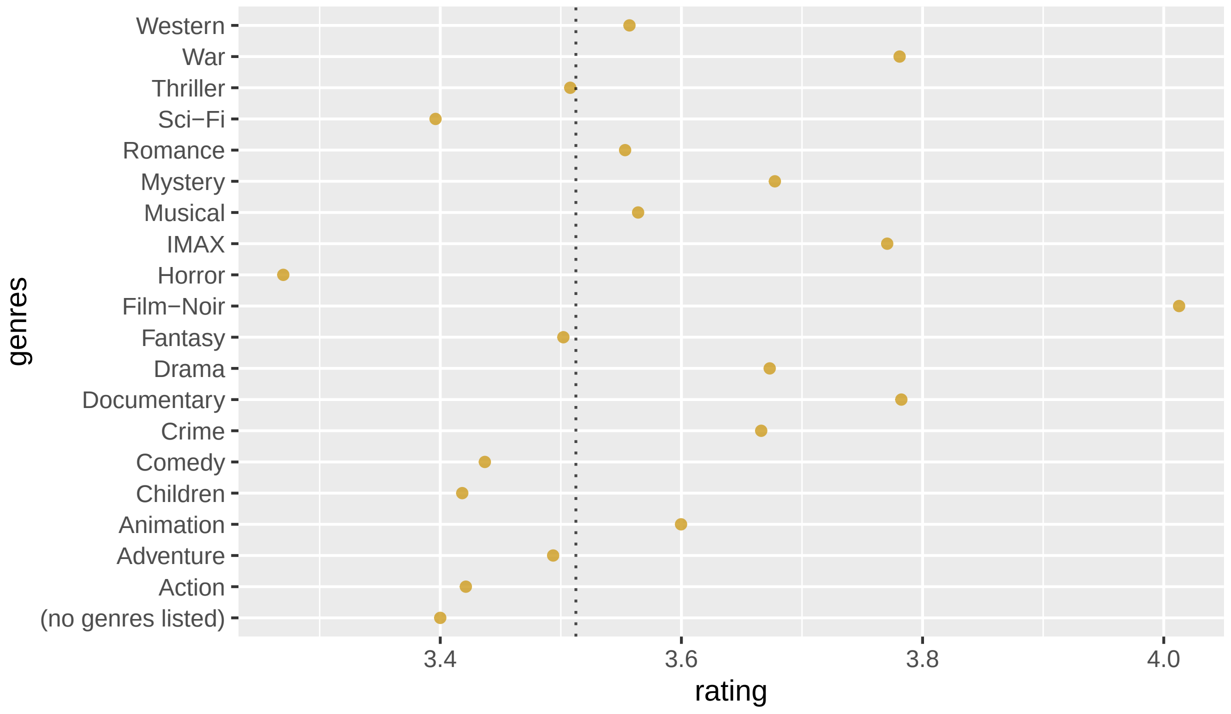
Model 6 (Genres Split)

As a matter of fact, in some versions of the MovieLens dataset the **genres** variable appears **split into many columns**, one for each movie genre. We will try to mimic this setting using our own train set.

Thus, we start by **splitting genres** by dint of **separate_rows()** and a short regular expression.

```r
#Split genres
train_split <- train_set %>% separate_rows(genres, sep = "\\|")

test_split <- test_set %>% separate_rows(genres, sep = "\\|")
```

Then, we confirm the presence of an **individual genres effect** by creating a scatter-plot.



However, now we are troubled by the fact that, using **separate_rows()**, we actually **increased the number of observations** in our train set. This will prove problematic if we were to eventually evaluate this model using the validation set.

Accordingly, we decide to spread **genres**, thereby creating an **individual column for each genre** - we use the column timestamp as the key argument since we no longer need it.

```
#Create a column for each genre in the dataset
train_split <- spread(train_split, key = genres, value = timestamp)

test_split <- spread(test_split, key = genres, value = timestamp)
```

Now we have to **remove missing values** from each observation that doesn't belong to a given genre, and **binarize** our data to make them available for computation.

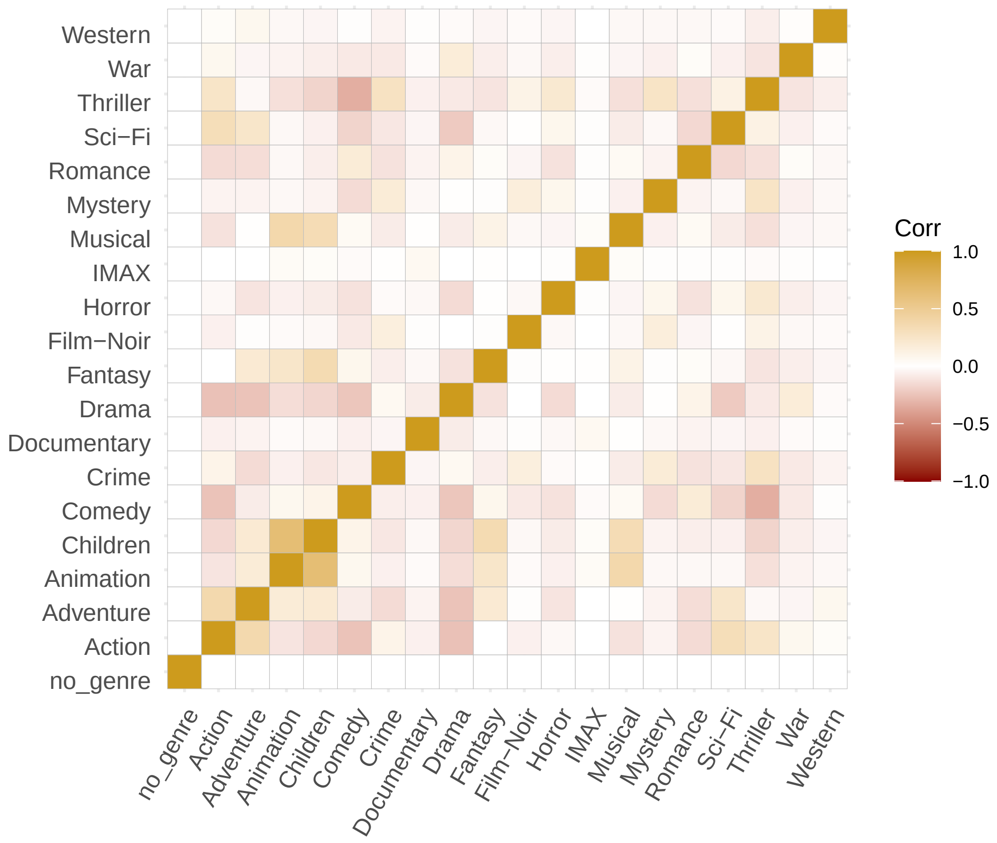
```
#Binarize genre observations
#(0 = movie doesn't belong to a given genre, 1 = it does)
train_split[is.na(train_split)] <- 0
train_split[,7:26]<- ifelse(train_split[,7:26] > 0, 1, 0)
train_split <- as.data.frame(train_split)

test_split[is.na(test_split)] <- 0
test_split[,7:26]<-ifelse(test_split[,7:26] > 0, 1, 0)
test_split <- as.data.frame(test_split)
```

We also need to rename the **(no genre listed)** column, since it appears in a potentially problematic format.

```
#Rename no genre column
train_split <- rename(train_split, "no_genre" = `(no genres listed)`)

test_split <- rename(test_split, "no_genre" = `(no genres listed)`)
```

We then compute - and **plot** - the **correlation coefficients** between each **genre variable**, in order to make sure not to add **collinearity** to our model.

```
#Correlation between individual genres variables
genres_cor<- cor(train_split[,7:26], use = "pairwise.complete")
```



We observe only **weak correlations** - if not no correlation at all - between the variables. Accordingly we resume building our model, being potentially able to add all individual genre effects.

We start with **Horror**, which, on average, is the worst rated genre. Then, we tweak our **regularization function** and, as a trial, run it with the best λ from the previous model.

```
#Run function with best lambda from last model
regularization_genres_Horror(5, train_split, test_split)
```

```
## [1] 0.8648612
```

To our initial surprise - with the current digit approximation - the resulting **RMSE doesn't change at all** from the one we obtained with **model 4**, the model we run before adding the genres effect.

This might be due to the fact that our **regularization** formula penalizes **Horror** to the point that has **virtually no effect** on the way the model fits our train set.

This seems to be confirmed by the fact that **Horror** has very few entries **(552622)**, as compared to other genres such as - for instance - **Action**, which has **2048436**.

We thereby make another attempt, using **Action** as the only genres variable.

```
#Run function
regularization_genres_Action(5, train_split, test_split)
```

```
## [1] 0.8648347
```

The resulting **RMSE** has indeed **improved**, confirming our assumption.

We then decide to build a model including only variables with a **relevant number of observations**. On the basis of **Horror** behaviour, we decide to filter them using a cut-off value of **999999**.

```
#Build a model only with genres with relevant number of observation

#Build a genres data frame
genres <- train_split[,8:25]

#Number of observations per genre
genres_sum <- sapply(genres, function(n){
  res <- sum(n)
})

#Order genres by number of observations and filter out non-significant genres
data.frame(genres_sum) %>%
  filter(. > 999999) %>%
  arrange(desc(.))
```

```
##           genres_sum
## Drama        3128642
## Comedy       2832972
## Action       2048436
## Thriller     1859510
## Adventure    1527769
## Romance      1369135
## Sci-Fi       1073697
## Crime        1061689
```

Thus, we build our **regularization function** by progressively adding each genre column (from **Drama** to **Crime**) using our **tidyverse approach**. Since this **function** has a very **large number of lines of code**, we don't show it here.

Then, we proceed by using **cross validation** to pick the best value of λ - which is, again, **5** - and by fitting the resulting model to obtain our **predictions** and resulting **RMSE**.

```
#Run model with best lambda
regularization_genres_split(5, train_split, test_split)
```

```
## [1] 0.8647848
```

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117
3 Movie + User + Release Date	YES	5-Folds (Lambda)	0.8649761
4 Movie + User + Release Date + Review Date	YES	5-Folds (Lambda)	0.8648612
5 Movie + User + Release date + Review Date + Genres	YES	5-Folds (Lambda)	0.8645972
6 Movie + User + Release Date + Review Date + Genres Split	YES	5-Folds (Lambda)	0.8647848

The resulting **RMSE** is rather low, although **not as much** as the one from **model 5**, where **multiple genres** were used to describe each movie, and which, accordingly, we assume to be a better and more specific approach. To definitively confirm this, we could try not to regularize the individual genres variable we didn't include in the model, and see whether the **RMSE** improves.

However, not only this trial and error approach is very unlikely to reach better results, but is also very time consuming, and will probably make computation very challenging.

Instead, we decide to rely on one of the most popular methods used in movie recommendation systems, **matrix factorization**.

Model 6 (Matrix Factorization)

As previously stated, if we consider that each user rated only a fraction of the movies we examined, we can think of our dataset as a **sparse matrix** with many **missing values** - that is why in scientific literature our task is often referred to as **matrix completion problem**.

In order to fill these **missing values**, correlations between different **rating patterns** must be leveraged.

In machine learning, this is generally done either by predicting user-movie combinations on the basis of the **neighboring cells** in the matrix (**memory based approaches**), or by assuming that preferences can be inferred by **modeling the structure** of the examined data (**model based approaches**). Considering the size of our dataset, as well as the large number of cells to predict, thus far we have used a set of relatively simple **linear models** in order to approximate this structure.

A better - and widely adopted - method is **matrix factorization**, an approach whereby we conceive of our matrix as the **product of two matrices**, a user-feature matrix and a movie-feature matrix, each of which is constructed by identifying a number of **K latent factors** in the original one.

In R, **matrix factorization** is relatively simple to implement by making use of packages such as **recosystem**.

Developed by Yixuan Qiu et al. at the National Univeristy of Taiwan, **recosystem** is specifically suitable to our task, and can be operated as follows:

First, we need convert the data into a format **recosystem** can process. This is done by mean of the function, **data_memory()** which saves the formatted data as an R object - in recosystem this data could also be saved directly into the hard drive using the function data_file().

```
#Set seed for randomized process
set.seed(1, sample.kind = "Rounding")

#Format data for package usage
#Specify index1 = TRUE since our train_set userId first entry is 1
train_reco <- data_memory(train_set$userId, train_set$movieId, train_set$rating,
                          index1 = TRUE)

test_reco <- data_memory(test_set$userId, test_set$movieId, index1 = TRUE)
```

Then we need to call the function **Reco()**.

```
#Call to Reco function
r <- Reco()
```

Now, we could simply proceed by training our model. However, since a series of **hyperparameters** are necessarily employed in the factorization process, **recosytem** has a specific function that uses **cross validation** to choose the best value for all of them.

Here, for our first try, we only **tune** across different values of **K latent factors** (through the argument **dim**), and **learning rate**. We use default tuning values for the other **hyperparameters**, as well as the default 5 folds to use in cross validation. We also specify the number of cores we want to use for computation, and the number of iteration for the **tuning process**.

```
#Tune hyper-parameters
opts <- r$tune(train_reco, opts = list(dim = c(10, 20, 30), lrate = c(0.1, 0.2),
                                       nthread = 2, niter = 5))
```

Now we are ready to **train** our algorithm with the best parameters obtained from the tuning process.

```
#train algorithm
r$train(train_data = train_reco, opts = c(opts$min, nthread = 2, niter = 100))
```

And to **predict** the outcomes.

```
#predict
predictions_reco <- r$predict(test_reco, out_pred = out_memory())
```

```
#Rmses
RMSE(predictions_reco, test_set$rating)
```

```
## [1] 0.8018538
```

The **RMSE** dropped by nearly **0.06** as compared to **model 5** and is by far the lowest we have achieved.

However, this was only our first attempt. We can probably do better by going through **tuning** again, and testing out different values for other **hyperparameter**, specifically if we do so by building on the best ones we obtained from the first tuning.

```
#Re-tune with different hyper-parameters
opts1 <- r$tune(train_reco, opts = list(dim = 30, lrate = c(0.1, 0.2),
                                        costp_l1 = 0, costp_l2 = 0.01,
                                        costq_l1 = 0, costq_l2 = c(0.1, 0.2),
                                        nthread = 2, niter = 20))
```

```
#Train with new best hyper-parameters
r$train(train_reco, opts = c(opts1$min, nthread = 2, niter = 100))
```

```
#New predictions
predictions_reco1 <- r$predict(test_reco, out_pred = out_memory())
```

```
#New RMSES
RMSE(predictions_reco1, test_set$rating)
```

```
## [1] 0.7943937
```

It is easy to see how **RMSE** further improved, dropping to **0.7943479**, which is a value far below the others we previously obtained as well as our project target of **0.86490**.

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117
3 Movie + User + Release Date	YES	5-Folds (Lambda)	0.8649761
4 Movie + User + Release Date + Review Date	YES	5-Folds (Lambda)	0.8648612
5 Movie + User + Release date + Review Date + Genres	YES	5-Folds (Lambda)	0.8645972
6 Movie + User + Release Date + Review Date + Genres Split	YES	5-Folds (Lambda)	0.8647848
7 Matrix Factorization	YES	5-Folds (Tuned)	0.7943479

Thus we select this very last model, **model 7** , as our **final model**, and thereby proceed by evaluating it on the **validation set**.

Validation

```r
#Set seed
set.seed(1, sample.kind = "Rounding")

#Format train set
edx_reco <- data_memory(edx$userId, edx$movieId, edx$rating,
                        index1 = TRUE)
#Format test set
validation_reco <- data_memory(validation$userId, validation$movieId,
                               index1 = TRUE)
```

```r
#Call to reco function
r_validation <- Reco()
```

```r
#Train model
r_validation$train(train_data = edx_reco,
                   opts = c(opts1$min, nthread = 2, niter = 100))
```

```r
#Predictions on the validation set
predictions_validation <- r_validation$predict(validation_reco,
                                               out_pred = out_memory())
```

```r
#RMSE from final matrix factorization on the validation set
RMSE(predictions_validation, validation$rating)
```

```
## [1] 0.7827345
```

Our final **validated RMSE** is even lower than what we predicted using the **train_set** / **test_set** split.

Rather than to the randomness involved in the data partitioning process, this is probably due to the fact that **edx** is a larger set where to train our **matrix factorization** algorithm, and that the **validation set** is a bigger set where to eventually test it.

We can consider ourselves happy with the results we obtained from **matrix facorization**, our **final model**.

Results

The **RMSE** resulting from **matrix factorization** is, by far, the lowest we have obtained, having it dropped by **0.0704369** from our best performing **linear model**.

That comes as no surprise, considering that **matrix factorization** methods are capable of explaining portions of variability that **linear models** can only **roughly approximate**.

Throughout our project we tried to tackle this **lack of complexity** by progressively adding **biases** (or effects) to our models, but this approach, eventually, suffered from **two major deficiencies**: the **lack of further employable variables** in our dataset (e.g. demographics, or more context-specific information), and the **computational burden** that our tidyverse approach necessarily implied. By the time **model 5** was implemented, sequentially adding **biases** had become a cumbersome, time-consuming process to which **recosystem** provided a lightly elegant alternative.

In fact, not only did **recosystem** produce more accurate models, it also efficiently succeeded where other methods would have failed.

Because of the sheer size of our datatset, we could not have - for instance - converted the whole dataframe into a sparse matrix format, nor - accordingly - reduced its **high dimensionality** by manually performing **principal component analysis (PCA)** or other type of transformations. Not being able to convert our data, additionally, would have prevented us to make use of a popular R package such as **recommenderlab**.

Finally, here we present the **table** summarizing the **results** we have obtained from all the **models** we build and tested over the course of this report.

method	Regularized	CV	RMSE
1 Movie + User	NO	Tr/Tst	0.8659320
2 Movie + User + Release Date	NO	Tr/Tst	0.8656117
3 Movie + User + Release Date	YES	5-Folds (Lambda)	0.8649761
4 Movie + User + Release Date + Review Date	YES	5-Folds (Lambda)	0.8648612
5 Movie + User + Release date + Review Date + Genres	YES	5-Folds (Lambda)	0.8645972
6 Movie + User + Release Date + Review Date + Genres Split	YES	5-Folds (Lambda)	0.8647848
7 Matrix Factorization	YES	5-Folds (Tuned)	0.7943479
8 Matrix Factorization (Validated)	YES	5-Folds (Tuned)	0.7825019

Conclusion

Despite the satisfying results obtained by implementing **matrix factorization** with **recosystem**, some final considerations are to be made.

Firstly, we need to remind ourselves that, however much flawed and somewhat unpractical, our **tidyverse approach** led us to achieve an acceptable **RMSE**. This cannot be mindlessly discarded, since, as previously stated, we simply couldn't count on any other tool to help us fit linear regression models. Thus, we could generalize, and state that such approximations might prove useful even in the future, if we were to face similar instances, where computational power is fairly limited. Additionally, in order to speed up its very implementation, we could eventually explore other R packages, such as **CrossValidate**, to improve cross validation performances.

Secondly, our **recosystem** models were highly effective, and yet they may still prove to be **sub-optimal**. In machine-learning literature, there is a wide plethora of articles and researches that aim to refine **matrix factorization** for recommendation systems. For instance, interesting elaborations might reside in combining our **model-based** approaches to **memory-based** ones, or exploring alternative options to optimize our models' **hyperparmeters**.

Lastly, it would be interesting to work with more **comprehensive datasets**, where more specific elements, such as **context aware** information, might provide added value to the **predictive capability** of our models.