Harvard Data Science: Predicting Movie Rating with MovieLens dataset

Maxandre Hebert

2023-08-08

Contents

Introduction

In the rapidly evolving digital landscape, where user-driven content has gained significant prominence, understanding and predicting user behavior is critical for a wide range of applications, from e-commerce to entertainment. The power to accurately predict users' preferences is not just a boon for enhancing user experience but also a strategic advantage for businesses, contributing to their growth and sustainability.

One such area where user preference prediction plays a pivotal role is in the movie industry, especially in online streaming platforms. Personalized movie recommendations can drastically improve user engagement and satisfaction. This study utilizes the MovieLens 10M dataset, a rich trove of user-movie interactions, to predict user ratings, acting as a stepping-stone towards enhanced recommendation systems.

The MovieLens 10M dataset, made available by GroupLens, contains approximately 10 million ratings of 10,000 movies by 72,000 users. These data provide a granular view of user behavior and preferences. By employing modern data analysis and machine learning techniques, we aim to develop a model that accurately predicts a user's rating of a movie, based on their previous ratings and other users' behaviors.

This research paper aims to contribute to the ongoing discussion on user preference prediction and its implications for personalization in digital platforms. It is written to be understandable for a broad audience, whether you are a data science enthusiast, an industry professional, or simply someone interested in how your movie recommendations are shaped.

#Methodology/Analysis In this section we gonna first import the data then clean the data then do the exploratory work.

Importing the data

```
############################################################
# Create edx and final_holdout_test sets
############################################################

# Note: this process could take a couple of minutes

if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")

library(tidyverse)
library(caret)

# MovieLens 10M dataset:
# https://grouplens.org/datasets/movielens/10m/
# http://files.grouplens.org/datasets/movielens/ml-10m.zip

options(timeout = 120)

dl <- "ml-10M100K.zip"
if(!file.exists(dl))
  download.file("https://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

ratings_file <- "ml-10M100K/ratings.dat"
if(!file.exists(ratings_file))
  unzip(dl, ratings_file)

movies_file <- "ml-10M100K/movies.dat"
```

```r
if(!file.exists(movies_file))
  unzip(dl, movies_file)

ratings <- as.data.frame(str_split(read_lines(ratings_file), fixed("::"), simplify = TRUE),
                         stringsAsFactors = FALSE)
colnames(ratings) <- c("userId", "movieId", "rating", "timestamp")
ratings <- ratings %>%
  mutate(userId = as.integer(userId),
         movieId = as.integer(movieId),
         rating = as.numeric(rating),
         timestamp = as.integer(timestamp))

movies <- as.data.frame(str_split(read_lines(movies_file), fixed("::"), simplify = TRUE),
                        stringsAsFactors = FALSE)
colnames(movies) <- c("movieId", "title", "genres")
movies <- movies %>%
  mutate(movieId = as.integer(movieId))

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

# Final hold-out test set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding") # if using R 3.6 or later
# set.seed(1) # if using R 3.5 or earlier
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
validation <- movielens[test_index,]

# Make sure userId and movieId in final hold-out test set are also in edx set
validation_set <- validation %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

# Add rows removed from final hold-out test set back into edx set
removed <- anti_join(validation, validation_set)
edx <- rbind(edx, removed)

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

```r
edx <- readRDS("edx.rds")
validation_set <- readRDS("validation_set.rds")
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
```

```
## Le chargement a nécessité le package : tidyverse

## Warning: le package 'tidyverse' a été compilé avec la version R 4.2.3

## Warning: le package 'ggplot2' a été compilé avec la version R 4.2.3

## Warning: le package 'tibble' a été compilé avec la version R 4.2.3

## Warning: le package 'tidyr' a été compilé avec la version R 4.2.2
```

```
## Warning: le package 'readr' a été compilé avec la version R 4.2.3

## Warning: le package 'purrr' a été compilé avec la version R 4.2.2

## Warning: le package 'dplyr' a été compilé avec la version R 4.2.3

## Warning: le package 'stringr' a été compilé avec la version R 4.2.2

## Warning: le package 'forcats' a été compilé avec la version R 4.2.2

## Warning: le package 'lubridate' a été compilé avec la version R 4.2.3

## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
```

```
## Le chargement a nécessité le package : caret

## Warning: le package 'caret' a été compilé avec la version R 4.2.3

## Le chargement a nécessité le package : lattice
##
## Attachement du package : 'caret'
##
## L'objet suivant est masqué depuis 'package:purrr':
##
##     lift
```

```
library(tidyverse)
library(caret)
```

Data Cleaning

First let's clean the data and check if any missing value are there.

```
  #check if there are any missing data in the trainning set
  print(sum(is.na(edx)))
```

```
## [1] 0
```

```r
#check if there is any missing data in the validation set
print(sum(is.na(validation_set)))
```

```
## [1] 0
```

Alright our data set is good we can move on !

Data Type

```r
str(edx)
```

```
## 'data.frame':    9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : int  122 185 292 316 329 355 356 362 364 370 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 838984474 838983653 838984885 83
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|Ac
```

This trainning dataset contains about 9 million observations with 6 variables each. The variables include 'userId' (integer), 'movieId' (integer), 'rating' (numeric), 'timestamp' (integer), 'title' (character), and 'genres' (character). It appears to be a movie rating dataset where each observation represents a unique rating given by a user to a movie, along with the timestamp of the rating, the title of the movie, and its genres. The 'userId' and 'movieId' suggest that multiple ratings from the same user or for the same movie can be included.

Data Exploration

Descriptive Statistics

Number of unique movies|users|genres

Number.of.Movies	Number.of.Users	Number.of.movies.genres
10677	69878	797

From the given information, we can see that the original dataset is quite extensive, with ratings for over 10,000 different movies provided by nearly 70,000 unique users. Furthermore, the variety of content is noteworthy, with 797 distinct genres represented. The broad user base and diverse range of movie genres imply a comprehensive coverage of user preferences and movie types, making it valuable for movie recommendation systems, audience segmentation, or studying viewer behavior and trends.

Correlation Analysis

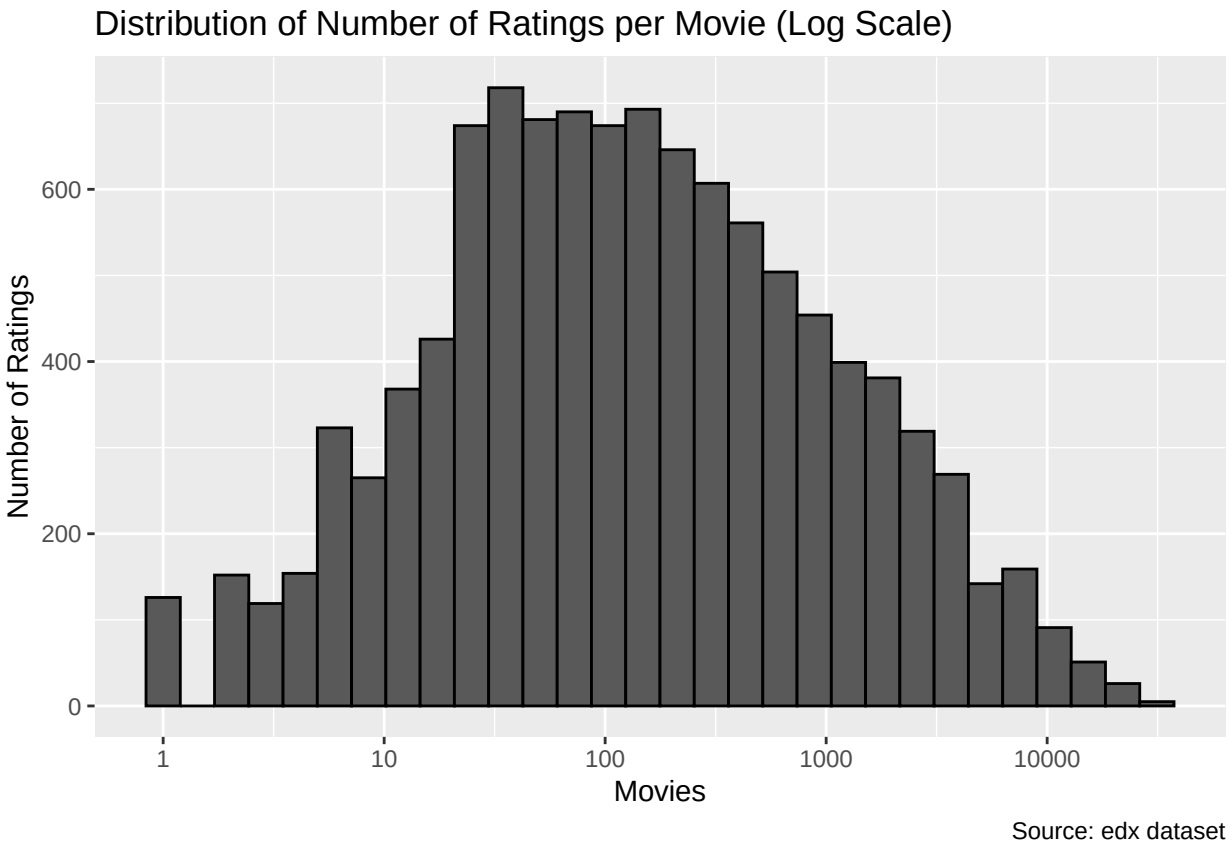
Table 2: Correlation matrix

	rating	timestamp	userId
rating	1.0000000	-0.0290497	0.0025183
timestamp	-0.0290497	1.0000000	0.0159158
userId	0.0025183	0.0159158	1.0000000

Not much to see there, let's go more visual.

Data Visualization

Distribution Plots

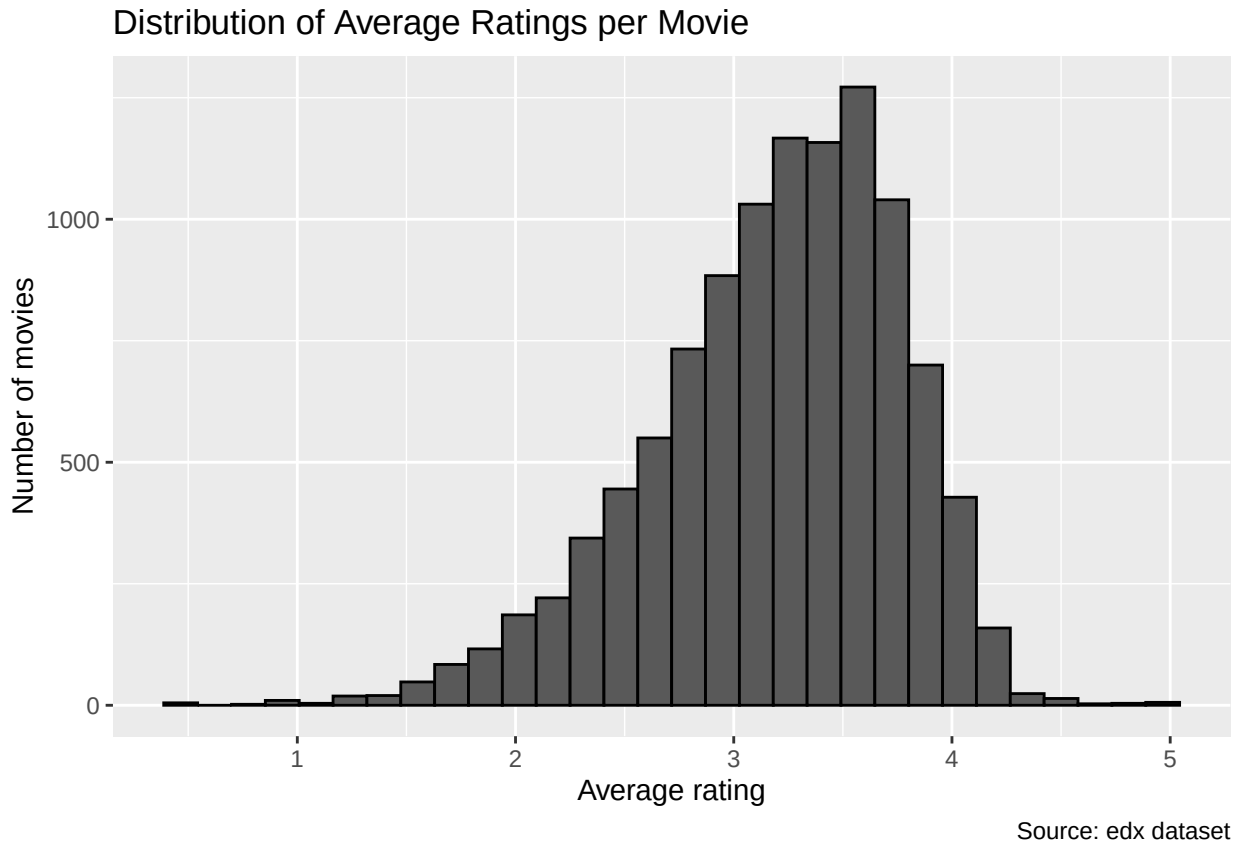


The distribution plot above illustrates the number of ratings per movie in the "edx" dataset, with the x-axis representing the number of ratings and the y-axis indicating the count of movies falling into each rating count range. The use of a logarithmic scale on the x-axis helps to better visualize the data, especially when there is a wide range of rating counts.

The plot exhibits a right-skewed distribution, also known as a positively skewed distribution. This means that there are a few movies with an exceptionally high number of ratings, while the majority of movies have relatively fewer ratings. In this case, the right tail of the distribution is longer, indicating the presence of a few popular movies that received significantly more ratings compared to the rest of the movies.

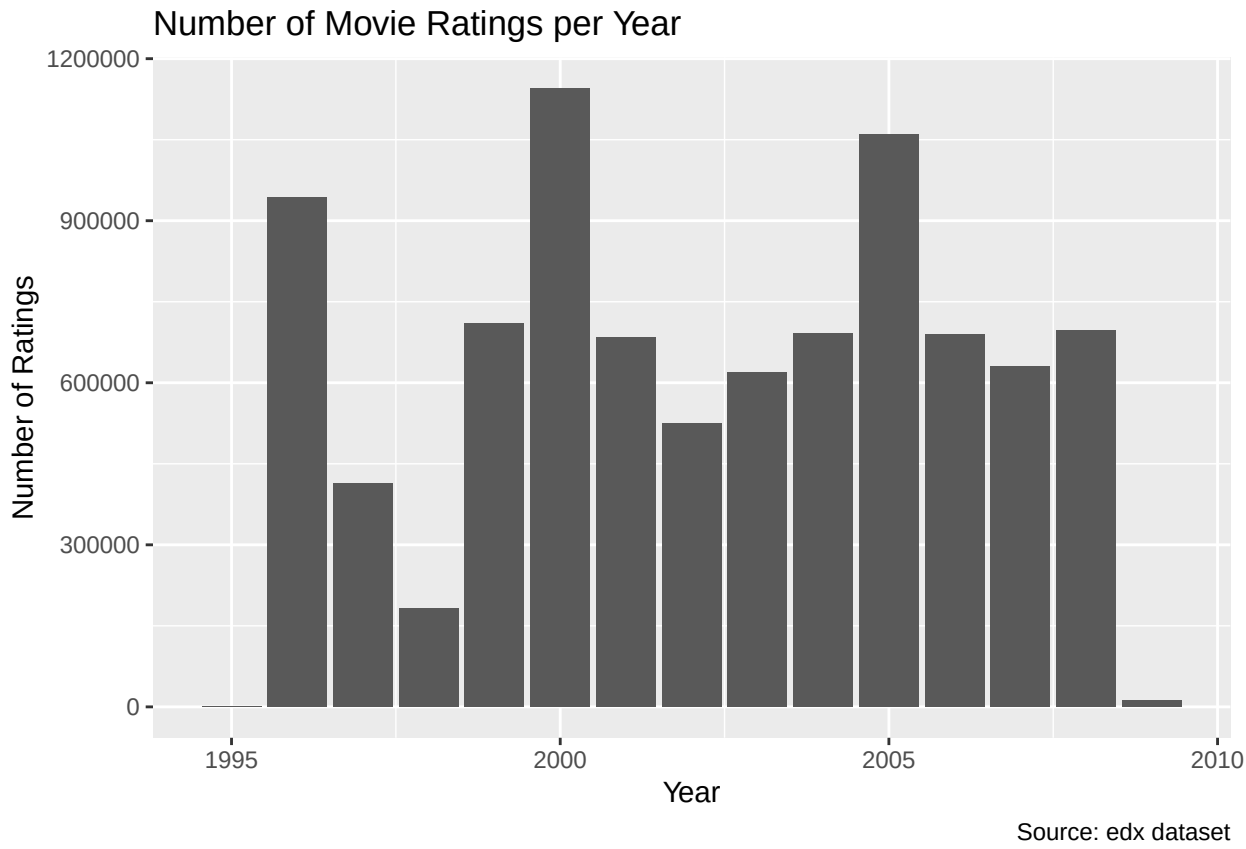
The right-skewed nature of the distribution suggests that some movies are much more popular or well-known among users, leading to a higher number of ratings for those movies. On the other hand, less popular or niche

movies tend to receive fewer ratings. Understanding the distribution of ratings is essential for developing recommendation systems, as it helps identify highly-rated movies that might be appealing to a broader audience, as well as niche movies that could be of interest to specific user segments.

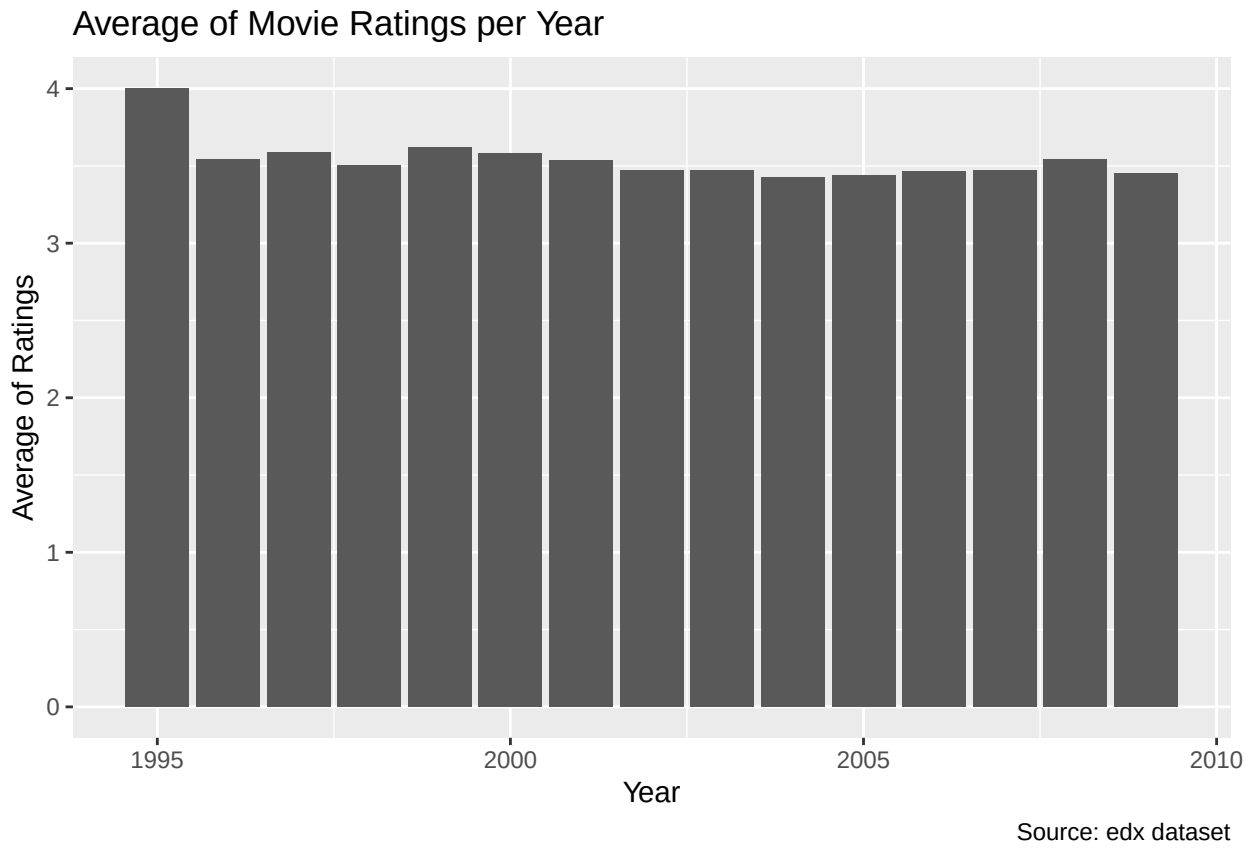


The distribution plot displayed above represents the average ratings for each movie in the "edx" dataset. The x-axis shows the average rating values, while the y-axis indicates the count of movies falling into each average rating range. In this case, the plot demonstrates a left-skewed distribution, also known as a negatively skewed distribution.

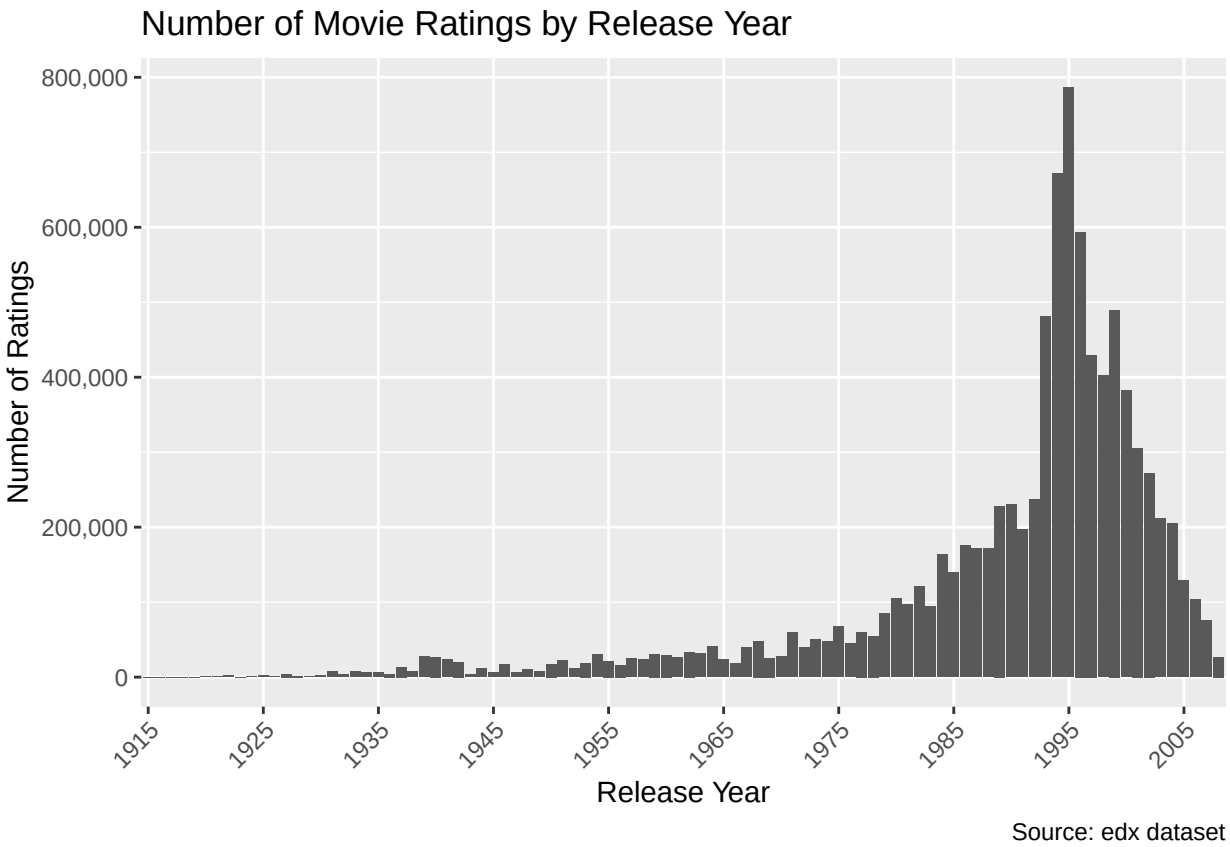
A left-skewed distribution suggests that the majority of movies have higher average ratings, with only a few movies receiving lower average ratings. The left tail of the distribution is elongated, indicating the presence of a small number of movies that received significantly lower average ratings compared to the rest. Such a distribution implies that most movies are generally well-received by users, with only a few exceptions being less favored.



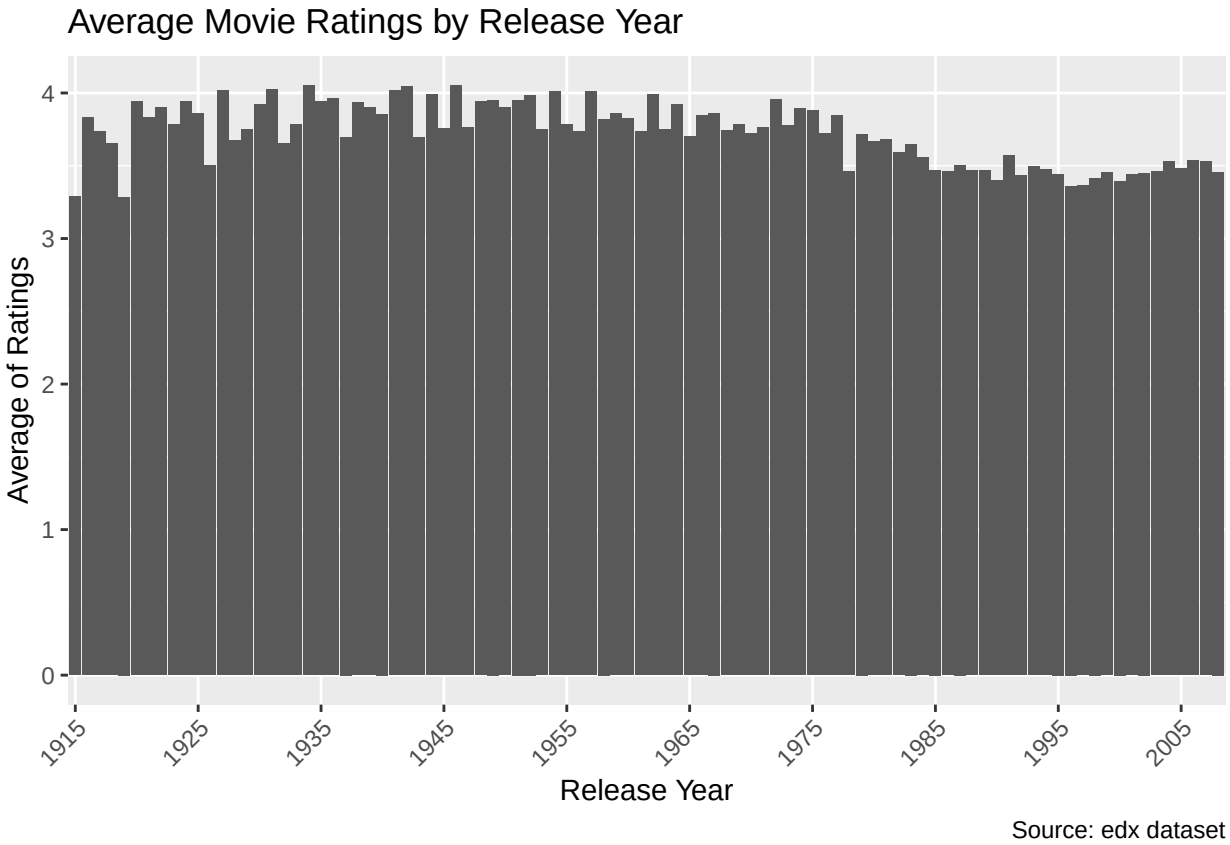
This is a plot showing number of rating per year. Not much to see there.



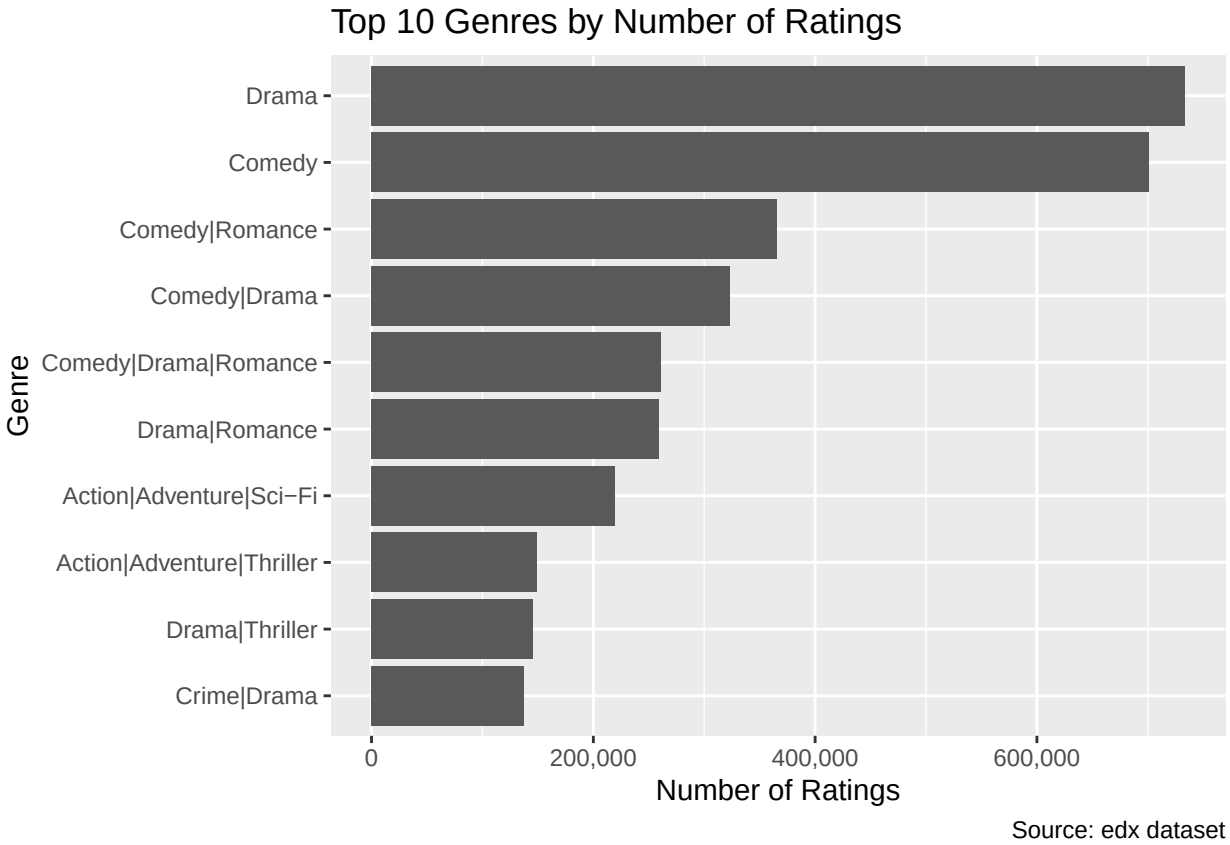
This plot show the average rating per year, it is pretty stable !



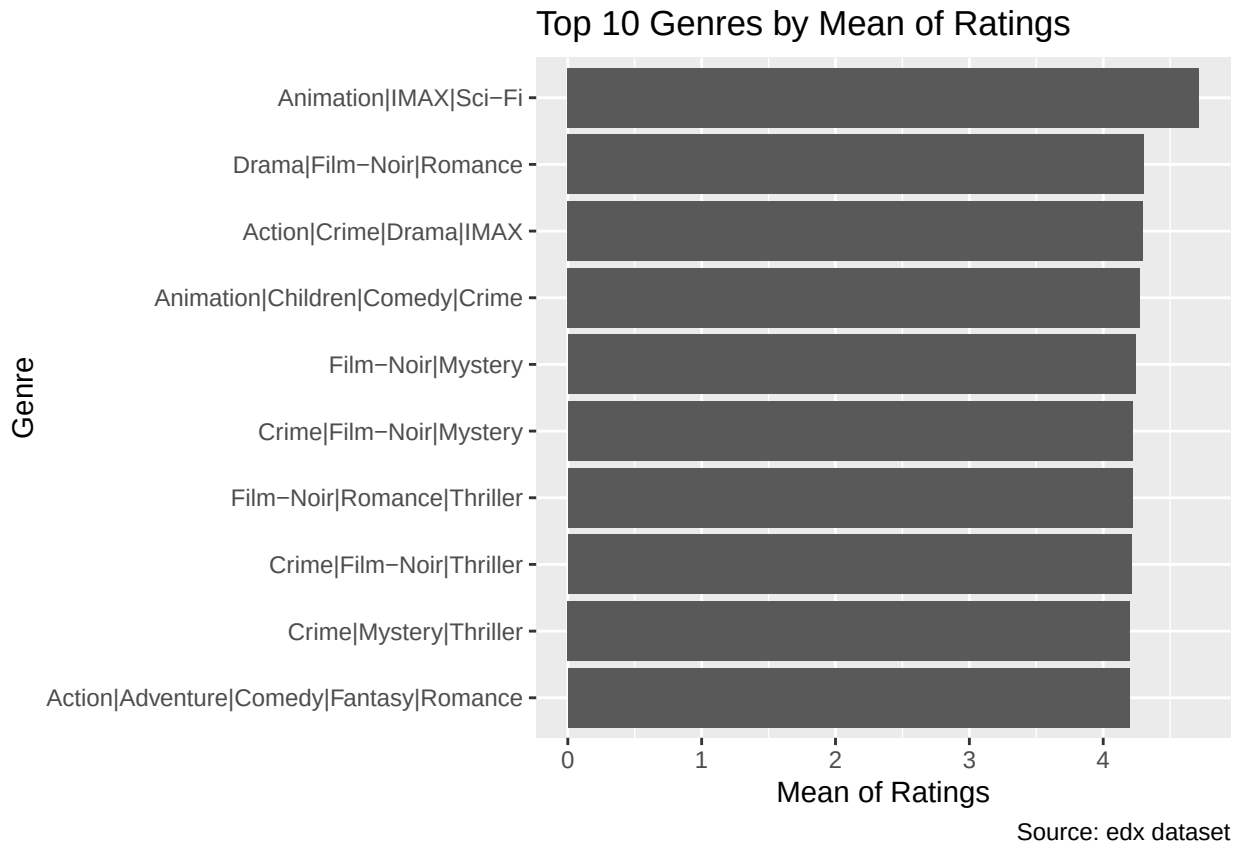
This plot show the number of ratings received movie release year. We can see that the 90s are the most popular movie.



In this plot, we see that when the release year is below 1975 the average ratings is higher, but we also know that those movies below 1975 were less rated so their average ratings could be less reliable.



Finally, in this plot here are the most rated genre.



Result

Modeling Approach

Choice of Model

For this project, RMSE model have been choose for its simplicity.

RMSE is a commonly used metric to measure the performance of regression models, including movie rating prediction models. It calculates the square root of the average of the squared differences between the predicted values and the actual values.

The formula for RMSE is as follows:

$RMSE = \sqrt{\frac{1}{n}\sum(predicted - actual)^2}$

Where:

- n is the number of data points (movie ratings) in your dataset.
- Σ represents the sum of the squared differences between the predicted and actual values for each data point.

A lower RMSE value indicates better model performance.

For this project we aim an RMSE below 0.8649

The formula

```
#This is the formula for the RMSE
rmse <- function(true_ratings, predicted_ratings) {
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

Model 1: Uniform Prediction and Least Squares Estimation for Movie Ratings

In the simplest possible recommendation system, we make a uniform prediction of the same rating for all movies, regardless of the user. This approach assumes that any differences in ratings are solely due to random variations. Mathematically, we express our prediction for a user's rating on a movie as follows:

Predicted Rating (Yu,i) = Average Rating (μ) + Random Variation (u,i)

To minimize the Root Mean Square Error (RMSE) and find the best estimate for the user's ratings, we employ the least squares method, which involves averaging all the ratings in the dataset.

```
#First model the mean of all ratings
mean_rating <- mean(edx$rating)
mean_rating
```

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

```
# Now if we test it on the validation test
only_mean_rmse <- rmse(validation_set$rating,mean_rating)
only_mean_rmse
```

```
## [1] 1.061202
```

```
rmse_table_results <- data.frame(Model="Model 1",RMSE=only_mean_rmse)
rmse_table_results %>% knitr::kable(caption="Table of all model result")
```

Table 3: Table of all model result

Model	RMSE
Model 1	1.061202

We can see that the rmse from the last piece of code, using a naive approach didn't work quite well. Actually we are off the track by 1.06 which is not what we want. We want the rmse below 0.8649.

Let's move to the second model.

Model 2: Incorporating Movie Effects for Improved Movie Rating Predictions

Model 2 introduces the concept of "Movie Effect," represented by the term b_i. The rationale behind this approach is that certain movies tend to receive higher ratings in general, irrespective of individual user preferences. To account for this, we incorporate an additional term, b_i, in the movie rating prediction equation.

The updated prediction equation now becomes Yu,i = μ + b_i + u,i, where:

Yu,i is the predicted rating by user u for movie i. μ is the overall average rating across all movies and users. b_i is the bias term representing the average ranking for movie i. u,i is the individual rating deviation for user u and movie i. By incorporating the movie bias (b_i) into the model, we can capture the inherent differences in movie ratings and better tailor the recommendations. Movies that tend to be rated higher will have a positive b_i value, while movies with lower average ratings will have a negative b_i value. This allows the model to adjust predictions based on the general popularity or appeal of each movie.

```
b_i <- edx %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mean_rating))

model_2_pred_ratings <- validation_set %>%
  left_join(b_i, by='movieId') %>%
  mutate(pred = mean_rating + b_i) %>%
  pull(pred)

RMSE_model_2 <- rmse(model_2_pred_ratings, validation_set$rating)
RMSE_model_2
```

```
## [1] 0.9439087
```

```
rmse_table_results <- bind_rows(rmse_table_results,
                                data.frame(Model="Model 2",RMSE=RMSE_model_2))

rmse_table_results %>% knitr::kable()
```

Model	RMSE
Model 1	1.0612018
Model 2	0.9439087

We can see our RMSE is improving but still note quite what we want.

Let's move to the next model.

Model 3: Incorporating User Effects and Enhanced Movie Rating Predictions

In Model 3, we introduce the concept of "User Effect," represented by the term b_u, along with the existing "Movie Effect" (b_i) from Model 2. The motivation behind this approach is that users exhibit substantial variability in their movie preferences and ratings due to individual tastes and preferences.

The updated prediction equation is now $Yu, i = + b_i + b_u + \epsilon u, i,$

where:

- Yu,i is the predicted rating by user u for movie i.
- μ is the overall average rating across all movies and users.
- b_i is the movie bias term representing the average ranking for movie i.
- b_u is the user bias term representing the individual rating behavior of user u.

- u,i is the individual rating deviation for user u and movie i.

By incorporating both the user and movie biases into the model, we can account for the personalized preferences of users and the inherent differences in movie ratings. Users with generally higher ratings will have a positive b_u value, while users with lower average ratings will have a negative b_u value. This allows the model to adapt predictions based on both individual user preferences and the popularity of each movie, leading to more accurate and tailored movie recommendations.

```
b_u <- edx %>%
  left_join(b_i, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mean_rating - b_i))

model_3_pred_ratings <- validation_set %>%
  left_join(b_i, by='movieId') %>%
  left_join(b_u, by='userId') %>%
  mutate(pred = mean_rating + b_i + b_u) %>%
  pull(pred)

RMSE_model_3 <- rmse(model_3_pred_ratings, validation_set$rating)
RMSE_model_3
```

```
## [1] 0.8653488
```

```
rmse_table_results <- bind_rows(rmse_table_results,
                                data.frame(Model="Model 3",RMSE=RMSE_model_3))

rmse_table_results %>% knitr::kable()
```

Model	RMSE
Model 1	1.0612018
Model 2	0.9439087
Model 3	0.8653488

We are improving and really close to our objective.

Model 4: Regularized Movie and User Effects for Improved Recommendation Predictions

Let's start first to understand what is regularization.

Imagine you are trying to hit a target with a bow and arrow. You're practicing, and you're getting really good at hitting the target exactly where you aim. This is like creating a model that fits your training data perfectly.

But then the wind starts blowing. You shoot the arrow exactly the way you've been practicing, but the wind carries it off-course, and you miss. This is like applying your model to new data and finding out it doesn't perform well. This is an example of overfitting - your model was too perfectly fitted to the specific conditions of your training data and failed when those conditions changed.

Regularization is like deciding not to aim exactly at the target, but a little off to one side, taking into account that there might be some wind that will push your arrow towards the target. You're making your shot a little "worse" on purpose, because it makes you better able to handle unexpected conditions.

In the world of machine learning, "making your shot a little worse" means preventing your model from getting too complex. A complex model can fit the training data perfectly, but fail on new data, just like the perfect shot fails when the wind starts blowing. Regularization does this by adding a penalty to the model for complexity.

Here what the code do :

lambdas is a vector of lambda values, which controls the level of regularization in the model. Higher lambda values imply a simpler model (more regularization) and lower values imply a more complex model (less regularization).

For each lambda value, the code calculates two bias terms: b_i (movie effect) and b_u (user effect). These terms are calculated using a method that takes into account regularization. The lambda value in the denominator serves as the regularization parameter, controlling the influence of the number of ratings each movie or user has.

The b_i term is calculated by grouping the training data by movieId, and for each movie, calculating the average of the deviations of the movie's ratings from the overall average rating (mu), adjusted by the regularization term.

The b_u term is calculated in a similar way, but it's grouped by userId and also takes into account the already-calculated b_i term. This means it's the average deviation of the user's ratings from the average ratings of the movies they rated, again adjusted by the regularization term.

Then, the code calculates the predicted ratings for the test data by adding up mu, b_i, and b_u for each rating. It's essentially creating a very simple model where the predicted rating is the overall average rating plus a movie effect and a user effect.

The last step is to evaluate the model by calculating the RMSE between the predicted ratings and the actual ratings in the test data.

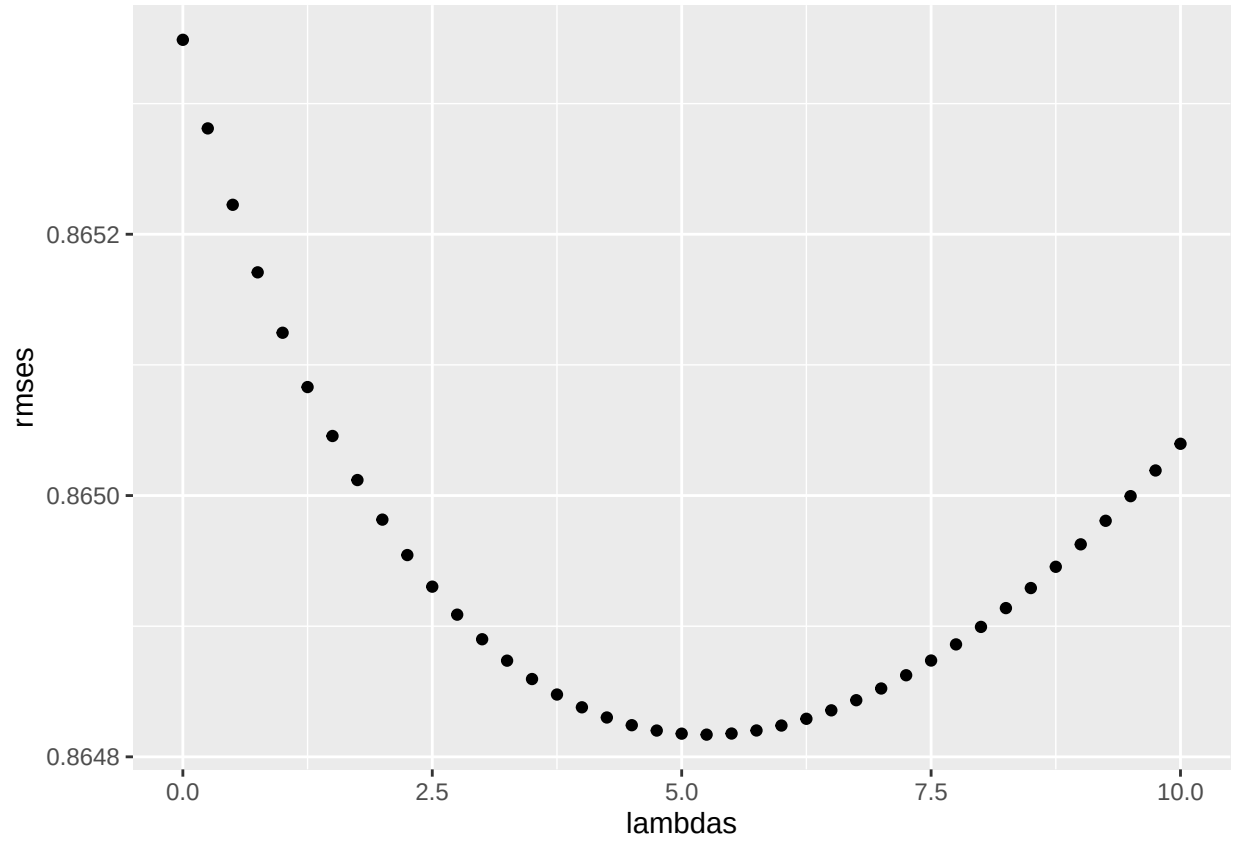
Here is the code :

```
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(lambda){

  b_i <- edx %>% # Movie Effect (Regularized)
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mean_rating)/(n()+lambda))

  b_u <- edx %>% # Movie and User Effect (Regularized)
    left_join(b_i, by = "movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mean_rating)/(n()+lambda))

  model_4_pred_ratings <- validation_set %>% # Predicted Ratings based on Model 4
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mean_rating + b_i + b_u) %>%
    pull(pred) # We want to use predictions based on Model 4

  return(rmse(model_4_pred_ratings, validation_set$rating)) # We want RMSE
})


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

```
## [1] 5.25
```

```
qplot(lambdas,rmses)
```

```
## Warning: `qplot()` was deprecated in ggplot2 3.4.0.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```




```
RMSE_model_4 <- min(rmses)
RMSE_model_4
```

```
## [1] 0.864817
```

```
rmse_table_results <- bind_rows(rmse_table_results,
                                data.frame(Model="Model 4",RMSE=RMSE_model_4))
```

BAM we got what we want !

Model Performance

Model RMSE score

Model	RMSE
Model 1	1.0612018
Model 2	0.9439087
Model 3	0.8653488
Model 4	0.8648170

We can see the best model was model 4.

Conclusion

In this movie recommendation algorithm project, an iterative approach was taken to achieve a low Root Mean Square Error (RMSE). Starting with a basic approach, different models were developed, incorporating MovieLens features like movie and user effects. Regularization techniques were also applied to improve the RMSE value.

Despite the progress made, there are still opportunities for improvement. By considering additional predictors such as "genres" and "releaseYear" and using a larger dataset like MovieLens 25M, further RMSE reduction is achievable.

Though the Regularized Movie + User Effects Model has been successful in achieving the primary goal of the MovieLens Project, certain limitations need addressing. The hardware constraints and processing power have restricted exploring more advanced machine learning models like k-NN. Moreover, a deeper understanding of the dataset could have allowed for better fine-tuning of the models.

Looking ahead, overcoming these limitations with the support of peer reviewers and more powerful hardware would lead to more refined movie recommendation algorithms, taking us closer to the project's full potential.