Data Science: Movelens Capstone

Movielens recommendations model comparison

Marilson Campos
Version: 1.00 - January, 2020

Contents

1 Project Goal & Approach

The objective of this project is to create a machine learning model to make recommendations of movies based on a dataset from the MovieLens project. We are using the dataset 'MovieLens 10M' that contains about 10 Million movie ratings.

The ratings were created by about 72 thousand users on 10 thousand movies and released the dataset to the public in 2009.

We will be using the code provided from the Edx.org site as the starting point to create a dataset used for training and validation.

We are following the sequence of steps as described below:

1. Project setup and dataset load
2. Exploratory Data Analysis
3. Create models and calculate the RMSE
4. Model Comparison & Final Results
5. Possible project improvements

2 Environment Setup

2.1 Loading the R packages used in project.

```
repo_url <- "http://cran.us.r-project.org"
if(!require(kableExtra))
  install.packages("kableExtra", repos = repo_url)
if(!require(tidyverse))
  install.packages("tidyverse", repos = repo_url)
if(!require(data.table))
  install.packages("data.table", repos = repo_url)
if(!require(caret))
  install.packages("caret", repos = repo_url)
```

2.2 Preparing the datasets.

This chunk downloads and prepares the dataset for our project.

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

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

movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))

movielens <- left_join(ratings, movies, by = "movieId")
# Removing unused datasets from memory to avoid out of memory errors.
rm(ratings, movies)
```

2.3 Create the 'Train' and 'Validation' datasets

We use the 'Train' dataset to build the models and the 'Validation' dataset to evaluate how our models are performing.

It's essential not to use any information from the 'Validation' set to make decisions about the model.

We will be taking 10% of the data and use for validation and the remaining 90% to train the models.

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

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

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

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```
train <- rbind(train, removed)

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

3 Basic Exploratory Data Analysis (EDA)

Before starting building models, we should look at the dataset and confirm our assumptions about the data.

Let us verify the sizes of each dataset is correct.

```
num_recs <- nrow(movielens)
num_train <- nrow(train)
num_validation <- nrow(validation)
numbers <- data.frame(Dataset=c('All data', 'Train', 'Validate'),
                      Records=c(
                        format(num_recs, big.mark=","),
                        format(num_train, big.mark=","),
                        format(num_validation, big.mark=",")),
                      PercentTotal=c(
                        format(round(100*num_recs/num_recs, digits = 2), big.mark=","),
                        format(round(100*num_train/num_recs, digits = 2), big.mark=","),
                        format(round(100*num_validation/num_recs, digits = 2), big.mark=","))
                      )
names(numbers) <- c("Dataset", "Number of records", '% Records')
kable(numbers) %>%
  kable_styling(position = "center", full_width = F)
```

Dataset	Number of records	% Records
All data	10,000,054	100
Train	9,000,055	90
Validate	999,999	10

Next, let's make sure that the number of unique users and unique 'movieId' values matches the specification of approximately 72k users and 10k movies.

```
unique_values <- movielens %>%
  summarize(unique_users = format(n_distinct(userId), big.mark=","),
            unique_movies = format(n_distinct(movieId), big.mark=","))
names(unique_values) <- c("Users", "Movies")
uniques_table <- gather(unique_values, key = "Field", value = "# of unique values")
kable(uniques_table) %>%
  kable_styling(position = "center", full_width = F)
```

Field	# of unique values
Users	69,878
Movies	10,677

Finally, we take a look at a few records of the dataset to confirm that the fields make sense.

```
kable(head(train, n=12)) %>%
  kable_styling(position = "center")
```

	userId	movieId	rating	timestamp	title	genres
1	1	122	5	838985046	Boomerang (1992)	Comedy|Romance
2	1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
4	1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller
5	1	316	5	838983392	Stargate (1994)	Action|Adventure|Sci-Fi
6	1	329	5	838983392	Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-
7	1	355	5	838984474	Flintstones, The (1994)	Children|Comedy|Fantasy
8	1	356	5	838983653	Forrest Gump (1994)	Comedy|Drama|Romance|Wa
9	1	362	5	838984885	Jungle Book, The (1994)	Adventure|Children|Romance
10	1	364	5	838983707	Lion King, The (1994)	Adventure|Animation|Childre
11	1	370	5	838984596	Naked Gun 33 1/3: The Final Insult (1994)	Action|Comedy
12	1	377	5	838983834	Speed (1994)	Action|Romance|Thriller
13	1	420	5	838983834	Beverly Hills Cop III (1994)	Action|Comedy|Crime|Thrille

Since we are going to start making decisions about our model we are not allowed to peek at the validation records from this point fowward. If we do we would inject bias into our model.

From now on we'll use the 'train' dataset.

3.1 Movie rattings distribution accross movies.

Let's take a look at the minimum and the maximum number of ratings for a movie.

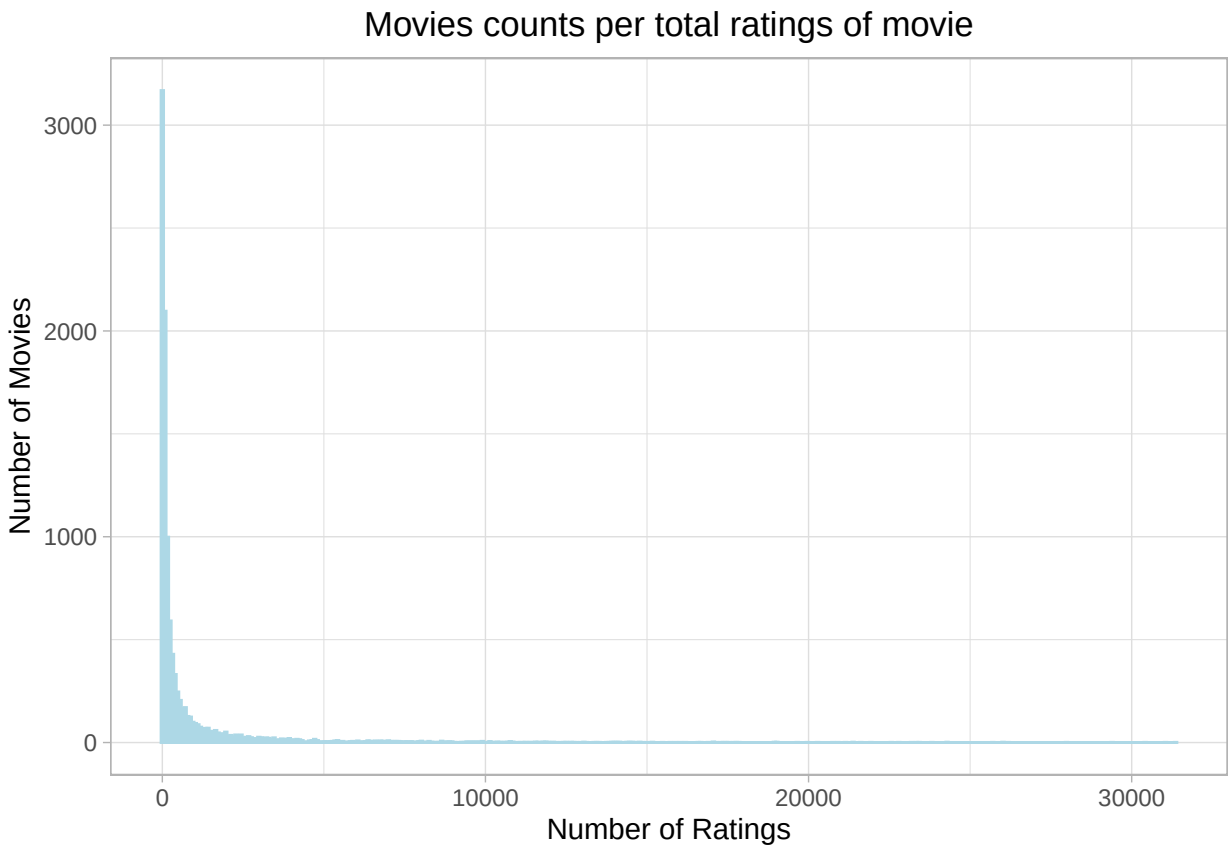
```
ratings_counts <- train %>%
  count(movieId) %>%
  transmute(ct=n)

min_max_ratings <- ratings_counts %>%
  summarize(
            min_ratings = format(min(ct), big.mark=","),
            max_ratings = format(max(ct), big.mark=","))
names(min_max_ratings) <- c("Smallest number of ratings for a film", "Largest number of ratings for a fi
min_max_table <- gather(min_max_ratings, key = "Stat", value = "Value")
kable(min_max_table) %>%
  kable_styling(position = "center", full_width = F)
```

Stat	Value
Smallest number of ratings for a film	1
Largest number of ratings for a film	31,362

As we can see, there there is at least one movie that was rated just once and also at least one movie rated more than 31 thousand times.

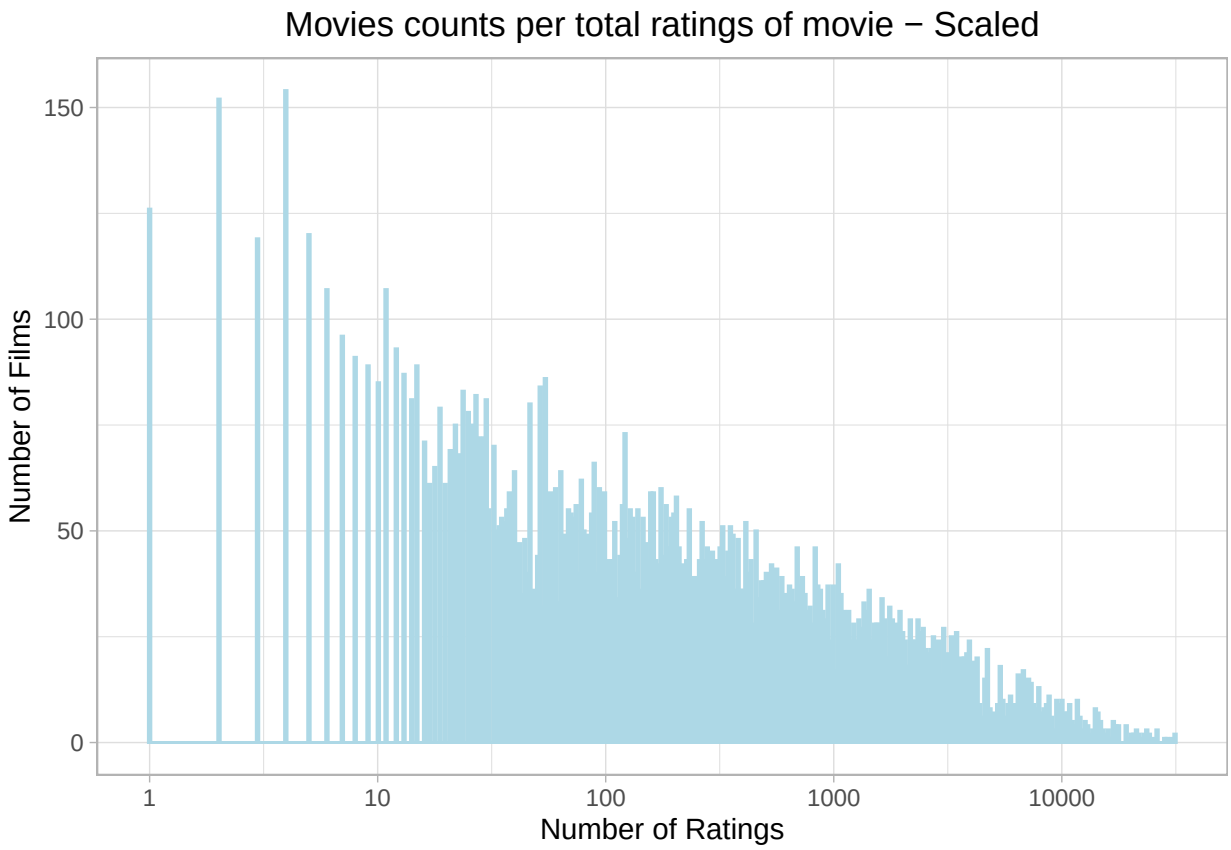
Let's try to plot a histogram showing how the distribution of movies across the number of ratings for it.

```
ggplot(ratings_counts, aes(ct)) +
  geom_histogram(bins = 400, color="light blue") +
  xlab('Number of Ratings') +
  ylab('Number of Movies') +
  theme_light() +
  theme(plot.title = element_text(hjust = 0.5)) +
  ggtitle("Movies counts per total ratings of movie")
```



This graph compressed most of the information on the left side of the x-axis. Let's re-scale the x-axis using the log function to see if we can expose additional information.

```
ggplot(ratings_counts, aes(ct)) +
  geom_histogram(bins = 400, color="light blue") +
  xlab('Number of Ratings') +
  ylab('Number of Films') +
  theme_light() +
  scale_x_log10() +
  theme(plot.title = element_text(hjust = 0.5)) +
  ggtitle("Movies counts per total ratings of movie - Scaled")
```



As we can see, there are around 125 movies with a single rating and a few movies with more than 10 thousand ratings.

3.2 User distribution accross movie ratings.

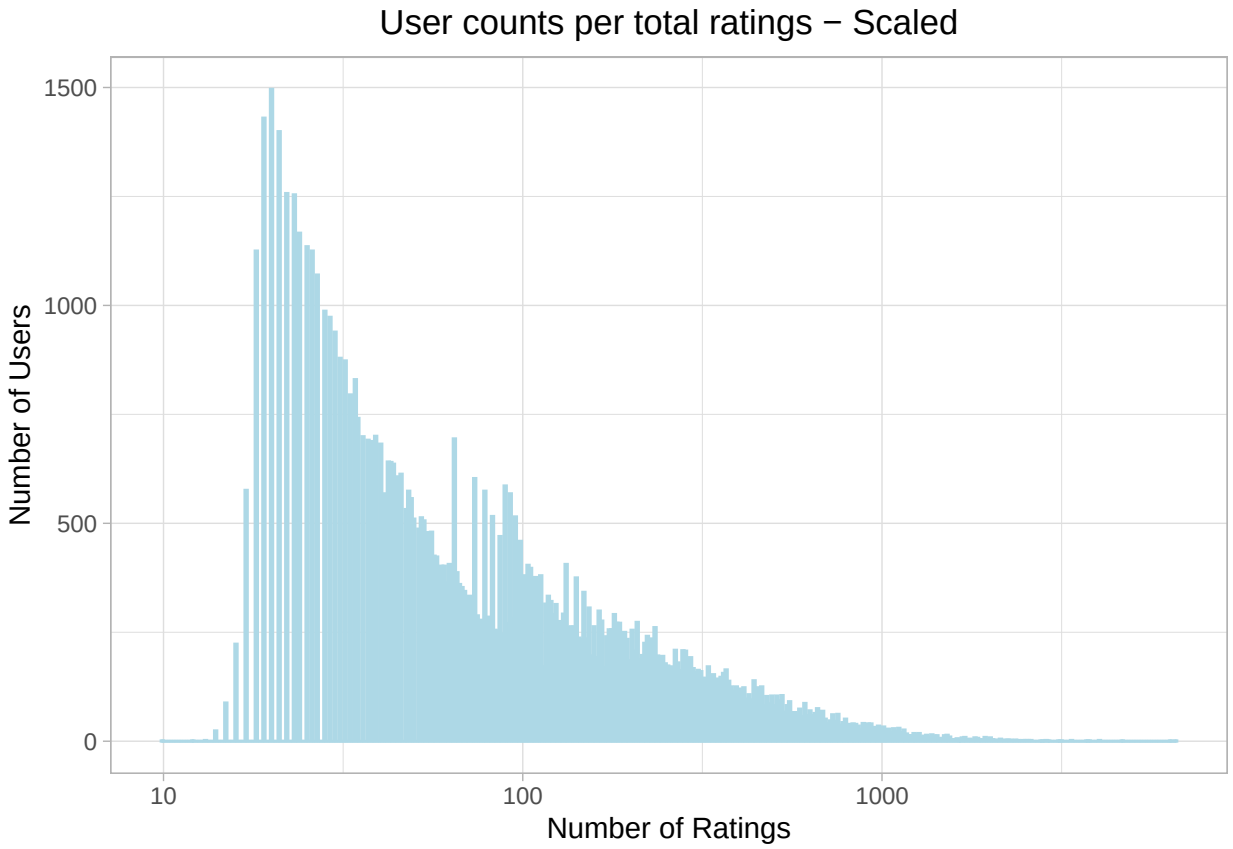
```
user_counts <- train %>%
  count(userId) %>%
  transmute(ct=n)

user_activity_stats <- user_counts %>%
  summarize(
            min_user_act = format(min(ct), big.mark=","),
            max_user_act = format(max(ct), big.mark=","))
names(user_activity_stats) <- c("Minimum number of times a user rates a movie",
                                "Maximum number of times a user rates a movie")
user_activity_table <- gather(user_activity_stats, key = "Stat", value = "Value")
kable(user_activity_table) %>%
  kable_styling(position = "center", full_width = F)
```

Stat	Value
Minimum number of times a user rates a movie	10
Maximum number of times a user rates a movie	6,616

The users also have a wide range of types of users, with some people rating few movies while some users rate thousands of movies.

Since it looks that this distribution also has a long tail, we can plot it using the log scale on the x-axis.

```
ggplot(user_counts, aes(ct)) +
  geom_histogram(bins = 400, color="light blue") +
  xlab('Number of Ratings') +
  ylab('Number of Users') +
  theme_light() +
  scale_x_log10() +
  theme(plot.title = element_text(hjust = 0.5)) +
  ggtitle("User counts per total ratings - Scaled")
```



This confirms the fact that users rates movies very different from each other.

4 Create models and calculate the RMSE

4.1 Building models.

To develop the models, we are going to define some basic terminology used in this document.

We define the variable $y_{u,m}$ as the rating for movie 'm' by user 'u' and the constant N as the total number of ratings.

We also define the predicted value for the $y_{u,m}$ rating as $\hat{y}_{u,m}$.

4.2 Loss function

The *loss function* expresses the error of a single prediction while the *cost function* accounts for all errors in the trainning dataset.

We will be using the square loss function expressed as:

$$loss(\hat{y}_{u,m}, y_{u,m}) = (\hat{y}_{u,m} - y_{u,m})^2$$

4.3 Cost Function

We call the Y and $\hat{Y}$ as the vector of observed ratings and the vector of predictions of the ratings.

We will be using the square root of the average of the loss function value. This model is also called 'RMSE' or residual mean of squared error.

Our cost function (RMSE) is calculated using:

$$cost(\hat{Y}, Y) = \sqrt{\frac{1}{N}\sum_{u,m}(\hat{y}_{u,m} - y_{u,m})^2}$$

```
cost <- function(observed, predicted){
  sqrt(mean((observed - predicted)^2))
}
draw_rmse_table <- function(ds){
  temp <- ds
  names(temp)  <- c("ML Model", "RMSE")
  kable(temp) %>%
    kable_styling(position = "center", full_width = F)
}
```

4.4 ML Model - Simple Model - Constant Average Rating

We'll start with a model that always predicts the rating as the 'average of ratings on the training dataset'.

We define μ as the average rating and our model as:

$$Y_{u,m} = \mu + \epsilon_{u,m}$$

Where $\epsilon_{u,m}$ is the error. The assumption of the model is that error is random and has a distribution with mean 0.

```
simple_model_predicted <- mean(train$rating)
```

If we predict all unknown ratings with μ we obtain the following RMSE:

```
ml_simple_rmse <- cost(validation$rating, simple_model_predicted)
ml_simple_df <- data.frame(model = "Simple Model (Avg Rating)", rmse = round(ml_simple_rmse,digits=5))
draw_rmse_table(ml_simple_df)
```

ML Model	RMSE
Simple Model (Avg Rating)	1.0612

4.5 ML Model - Movie

Next, we look if we can use the movie information to reduce the prediction errors obtained in the previous model.

We will expand our model and add the movie bias.

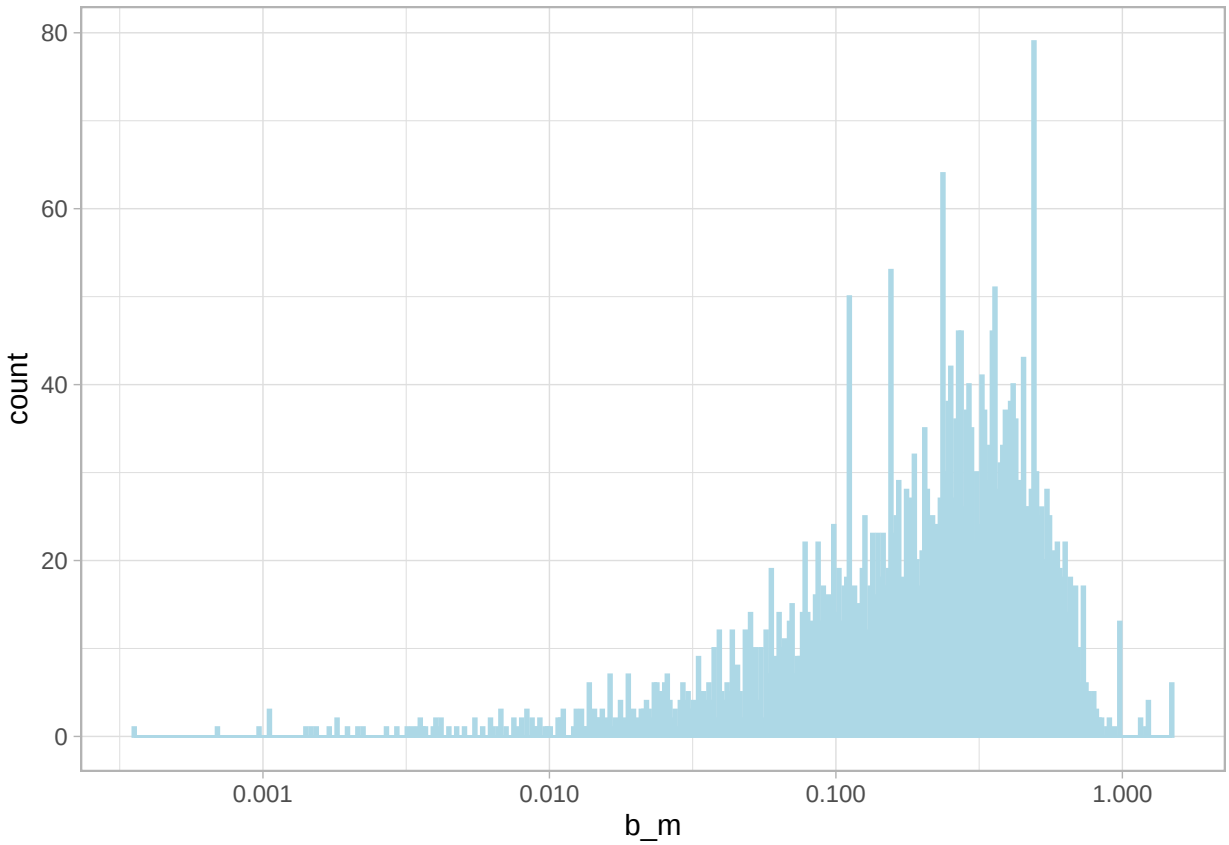
$$Y_{u,m} = \mu + b_m + \epsilon_{u,m}$$

Where b_m represents the bias for the movie 'm'.

```
mu <- mean(train$rating)
movie_averages <- train %>%
  group_by(movieId) %>%
  summarize(b_m = mean(rating - mu))
```

Let's confirm that these averages vary substantially for different movies:

```
ggplot(movie_averages, aes(b_m)) +
  geom_histogram(bins = 400, color="light blue") +
  theme_light() +
  scale_x_log10() +
  theme(plot.title = element_text(hjust = 0.5))
```



We can see that there is a component of movie bias. The next step is to calculate the RMSE for the new model, so we have a sense of how much better this model is when compared with the previous one:

```
movie_bias  <- validation %>%
    left_join(movie_averages, by='movieId') %>%
    pull(b_m)
movie_predicted <- mu + movie_bias

ml_movie_rmse <- cost(movie_predicted, validation$rating)
ml_movie_df <- data.frame(model = "Movie Model", rmse = round(ml_movie_rmse,digits=5))
draw_rmse_table(ml_movie_df)
```

ML Model	RMSE
Movie Model	0.94391

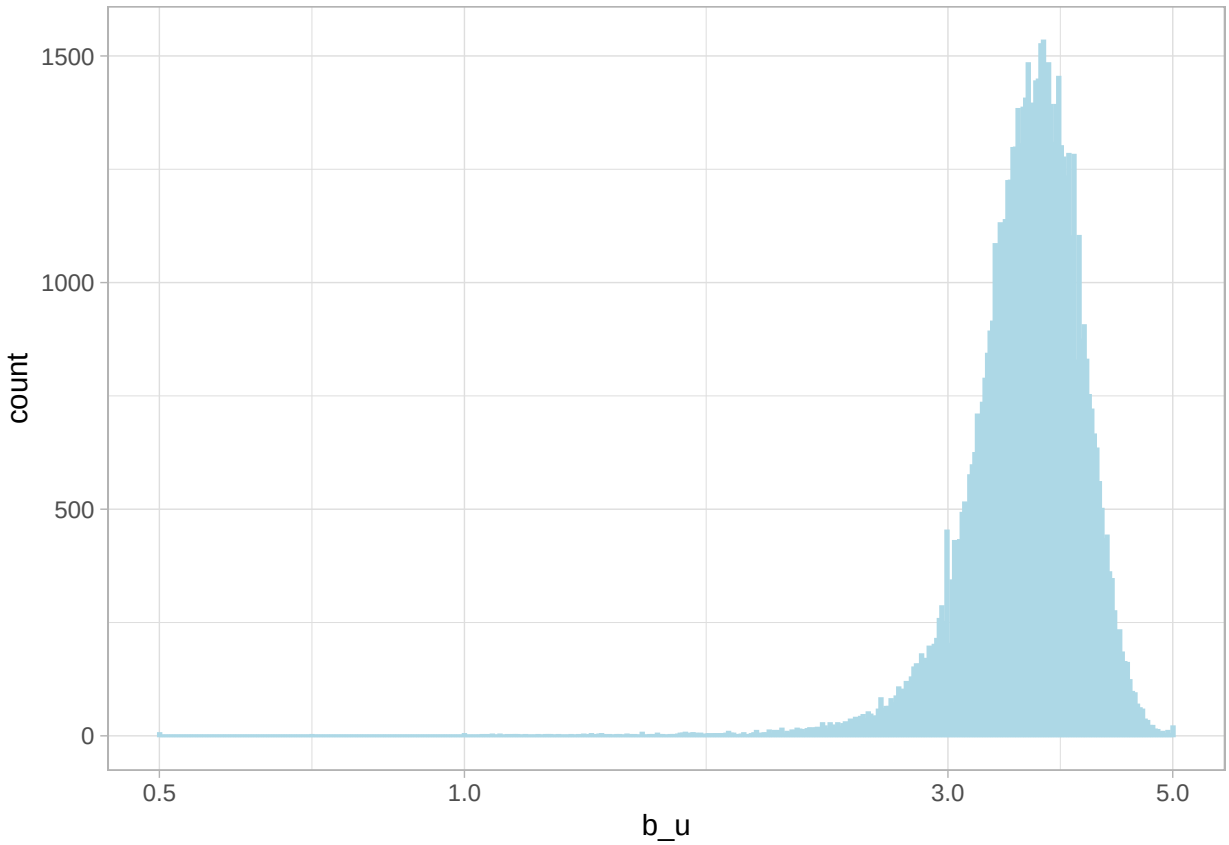
We are on the right track. After adding the movie effect, our error rate improved.

4.6 Looking at User Bias

Besides the movieId, e can explore the possibility of using the unserId in our model. Let's confirm that user rating averages vary substantially for different users:

```
user_averages_plain <- train %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating))
ggplot(user_averages_plain, aes(b_u)) +
  geom_histogram(bins = 400, color="light blue") +
  theme_light() +
```

```
  scale_x_log10() +
  theme(plot.title = element_text(hjust = 0.5))
```



We can see users rating the movies on substantially different on average, so adding the userId its most likely to improve the model.

4.7 ML Model - Movie + User

Let's then expand our model to include the effects both of movies and the users like the one defined below:

$$Y_{u,m} = \mu + b_m + b_u + \epsilon_{u,m}$$

Now we are considering the effects from users and movies in our recommendations.

```
user_averages <- train %>%
    left_join(movie_averages, by='movieId') %>%
    group_by(userId) %>%
    summarize(b_u = mean(rating - mu - b_m))
```

```
movie_user_predicted <- validation %>%
    left_join(movie_averages, by='movieId') %>%
    left_join(user_averages, by='userId') %>%
    mutate(y_hat = mu + b_m + b_u) %>%
    pull(y_hat)

ml_movie_user_rmse <- cost(movie_user_predicted, validation$rating)
ml_movie_user_df <- data.frame(model = "Movie & User Model", rmse = round(ml_movie_user_rmse,digits=5))
draw_rmse_table(ml_movie_user_df)
```

ML Model	RMSE
Movie & User Model	0.86535

4.8 ML Model - Movie Bias + User Bias using regularization

The concept of regularization adds a 'tax' on model complexity and allows these models to generalize better. These models avoid learning the noise created by few observations.

The most common type of regularization is called 'Ridge regularization' and uses the following formula:

$$loss = \frac{1}{N}\sum_{u,m}(y_{u,m} - \mu - b_m - b_u)^2 + \lambda\left(\sum_m b_m^2 + \sum_u b_u^2\right)$$

Where the λ is called the regularization parameter and defines how big of a 'tax' we are placing on the biases.

We can then prerfom a 'grid search' and calculated the RMSE for several values of lambda.

I performed an initial run using the values from 0 to 8 with increments of 0.25. This initial result produced an optimal $\lambda = 5$. However, after finding the neighborhood of the best lambda, we can adjust the step size and obtain a more precise value.

The code below shows the grid search after reducing the step size from 0.25 to 0.01.

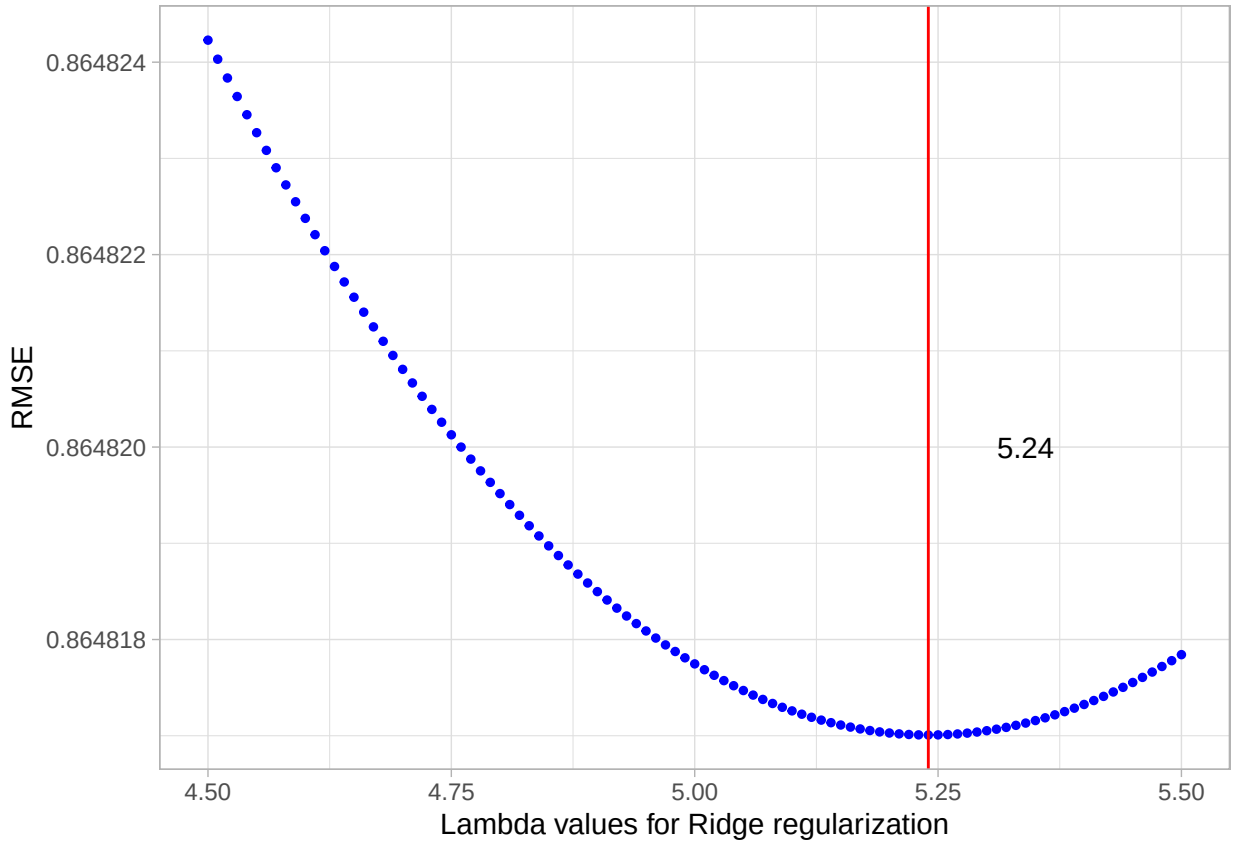
Our approach to search the initial range to find the approximated value. Then, we scan the neighborhood of the approximated value with a smaller step.

This approach produces a more precise result and is less computationally expensive than scanning a broad range using small steps.

```
lambda_grid <- seq(4.5, 5.5, 0.01)

rmse_vector <- sapply(lambda_grid, function(l){
  mu <- mean(train$rating)
  b_m <- train %>%
    group_by(movieId) %>%
    summarize(b_m = sum(rating - mu)/(n()+l))

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

  predicted_ratings <-
      validation %>%
      left_join(b_m, by = "movieId") %>%
      left_join(b_u, by = "userId") %>%
      mutate(y_hat = mu + b_m + b_u) %>%
      pull(y_hat)

  return(cost(predicted_ratings, validation$rating))
})
```

```
rmse_lambda <- data.frame(rmse = rmse_vector, lambda_grid = lambda_grid)
best_lambda <- lambda_grid[which.min(rmse_vector)]
```

```
ggplot(rmse_lambda, aes(x=lambda_grid, y=rmse)) +
  geom_point(color="blue", size = 1) +
  geom_vline(xintercept = best_lambda, linetype="solid",
                color = "red", size=0.5) +
  annotate("text", x = best_lambda + 0.1, y = 0.86482, label = best_lambda) +
  theme_light() +
  xlab('Lambda values for Ridge regularization') +
  ylab('RMSE') +
  theme(plot.title = element_text(hjust = 0.5))
```



For the full model, the optimal λ is: 5.24. This is the value produces the smallest RMSE.

```
ml_movie_user_reg_df <- data.frame(model = "Movie & User with regularization",
                                   rmse = round(min(rmse_vector),digits=5))
draw_rmse_table(ml_movie_user_reg_df)
```

ML Model	RMSE
Movie & User with regularization	0.86482

5 Model Performance Results

Let's present the RMSE from all models we built so far in a single table so we can compare our results.

```
results <- bind_rows(
  ml_simple_df,
  ml_movie_df,
  ml_movie_user_df,
  ml_movie_user_reg_df)
draw_rmse_table(results)
```

ML Model	RMSE
Simple Model (Avg Rating)	1.06120
Movie Model	0.94391
Movie & User Model	0.86535
Movie & User with regularization	0.86482

We can see that the regularized model using Movies and Users is the best performing model.

6 Possible project improvements

There are several possible changes we could make to our model to perform even better. Here is a short description of each item:

a. Adding the 'Year' of the rating. This change could allow the model to account for years where ratings were higher or lower than the average.

b. Adding the 'Month' of the rating. This change could capture the seasonality effects like people being more positive in some months of the year. (like Christmas for example)

c. Adding the 'movie genre' to the model. Despite being an obvious option, this requires to expand the dataset where each record on the current dataset would produce several records in the new dataset (one for each genre on the new one.) This change would require the use of a powerful server to handle the processing instead of a laptop.