MovieLens Project - Capstone Course (Juan Canon)

Juan Canon

2023-03-31

Introduction

Data science, as an interdisciplinary field, has made remarkable strides in recent years, offering innovative solutions to a wide array of problems across various domains. The power of data science lies in its ability to harness large volumes of data, apply sophisticated algorithms, and generate insights that can drive decision-making and improve user experiences. R programming, with its extensive libraries and versatile capabilities, has become a popular tool for data scientists to implement and experiment with data-driven solutions.

In this context, the movie recommendation system project serves as an exemplary case study to illustrate the practical application of data science tools and R programming techniques in addressing real-world challenges. While the primary objective is not to develop the most accurate movie recommendation model, the project focuses on exploring different aspects of data science and R programming through the implementation of diverse machine learning approaches. Movie recommendation systems, as an area of study, provide a unique opportunity to apply data science techniques to a rich dataset, such as the MovieLens dataset, that contains user preferences and movie features

Data Set

The MovieLens dataset, provided by the GroupLens Research Project at the University of Minnesota, serves as the foundation for our movie recommendation system. For this project, we have utilized the 10M version of the dataset to ease computation. The dataset comprises approximately 10 million ratings from 72,000 users for over 10,000 movies, collected between 1995 and 2021. The data includes user-provided ratings on a scale of 1 to 5, movie titles, genres, and timestamps of the ratings.

Goal of the Project

The primary objective of this project is to create a movie recommendation system using the MovieLens dataset as a fascinating case study to delve into different data science tools and R programming techniques. This system will analyze user preferences, movie features, genres and historical ratings to generate personalized recommendations. By doing so, the project demonstrates the practical application of data science concepts and R programming skills in addressing real-life challenges.

Key Steps Performed

1. Data Preprocessing: The dataset was cleaned and preprocessed using R programming tools to address missing values, inconsistencies, and outliers. Moreover, the data was transformed into a suitable format for the subsequent machine learning algorithms.

2. Feature Engineering: Relevant features were extracted from the dataset to enhance the recommendation system's performance. This included generating new features based on existing data, such as average ratings and user-movie interactions, using R programming techniques.

3. Data Exploration: The dataset was thoroughly examined to uncover insights into rating patterns, popular genres, and trends in user preferences. This step also involved the use of various R programming techniques for data visualization and summary statistics.

4. Model Selection: Various machine learning algorithms, including collaborative filtering, content-based filtering, and hybrid approaches, were evaluated to determine their suitability for the recommendation system. This exploration allowed for a deeper understanding of the strengths and weaknesses of different approaches.

5. Model Training and Evaluation: The selected models were trained using the preprocessed dataset, and their performance was assessed using appropriate evaluation metrics, such as root mean square error (RMSE) and precision at k. R programming techniques played a crucial role in implementing these models and evaluating their performance.

6. Implementation and Deployment: The final recommendation system was implemented and deployed as a user-friendly interface, allowing users to receive personalized movie recommendations based on their preferences and viewing history.

Analysis

Importing the Data to R

To import the dataset into R, we downloaded the ml-10m.zip file from the GroupLens website and extracted the ratings.dat and movies.dat files. The data was then cleaned and preprocessed, including converting data types and joining the ratings and metadata data frames.

Next, we split the data into a training set (edx) and a final hold-out test set (final_holdout_test) for evaluating the performance of a recommendation model. We ensured that the user and movie IDs in the final hold-out test set were also present in the training set (edx) and added any rows removed from the final hold-out test set back into the training set (edx).

Finally, we removed unnecessary variables to free up memory. The code for importing the data is available in the source code for this project but is not shown in the report due to its length.

Data Source: *"F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872"*

Data Wrangling

In this section, we perform the necessary data wrangling steps to prepare the MovieLens data-set for further analysis and modeling. Data wrangling is a crucial step in the data analysis process, as it involves cleaning, reshaping, and transforming the data to facilitate subsequent analysis and modeling tasks.

First, let's inspect the values and column classes of our data-set:

```
#Wrangling the data_____________________________________________________

## We start by inspection the values and column classes
first_inspection <- print(head(edx, 10))
```

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

```
dataset_Classes <- sapply(edx,class)
print(dataset_Classes)
```

```
##      userId     movieId      rating   timestamp       title      genres
##   "integer"   "integer"   "numeric"   "integer" "character" "character"
```

```
#Inspect the number of unique values for title and genres

unique_movieIds <- n_distinct(edx$movieId)
unique_userIds <- n_distinct(edx$userId)
unique_genres <- n_distinct(edx$genres)
tbl <- data.frame(
  'Variable' = c('movieId', 'userId', 'genres'),
  'Unique Values' = c(unique_movieIds, unique_userIds, unique_genres)
)

print(tbl)
```

```
##   Variable Unique.Values
## 1  movieId         10669
## 2   userId         69878
## 3   genres           797
```

```
#inspect for NA values
print(sum(is.na(edx)))
```

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

The analysis provides valuable insights into the nature and complexity of the variables under consideration. Specifically, we observe thousands of unique movies and users, indicating a large and diverse dataset.

Moreover, each movie is associated with multiple genres, highlighting the nuanced and varied nature of the data.

Tidying the Data

In the previous inspection, we observed that the data is already presented in a "tidy" format regarding observations, with only one observation per row. However, some columns contain more than one variable. For example, the "title" column includes the title and the year of release. To address this issue, we perform a simple manipulation to separate them into two distinct columns.

```
#separate the title column into 2 different variables. Then convert the year characters
#into integer values.

edx <- edx %>%
  mutate(year = as.integer(str_extract(title, "(?<=\\()\\d{4}(?=\\))")))

edx <- edx %>%
  mutate(title = str_replace(title, " \\(\\d{4}\\)", ""))
```

We use standard functions from the "dplyr" and "stringr" packages, which are widely known for data wrangling usage.

Finally, we perform three additional changes to the columns "timestamp" and "genres" to make them easier to interpret and analyze:

1. Create a twin dataset where genres are displayed as binary values (associated with 19 additional columns). For example, if a movie contains the genre "comedy", it displays a "1" in the column "comedy". We keep both formats, as it is not fully clear which presentation would be easier to manage.
2. Transform timestamp values from seconds to actual dates (using the lubridate package).
3. Add a new column with the number of times each movie has been rated, as an indication of how popular the movie is.

```
#extract all the unique values of the "genre" vector in 1 column

mini_edx <- edx[1:10000,1:6]

mini_edx2 <- mini_edx %>% separate_rows(genres, sep = "\\|") %>%
  group_by(genres)
genres <- unique(mini_edx2$genres)


# Create an empty data frame with columns for each genre
edx2 <- data.frame(matrix(0, ncol = length(genres), nrow = nrow(edx)))
names(edx2) <- genres

# Loop through each genre and add a 1 to the corresponding column if the genre is present
for (genre in genres) {
  edx2[, genre] <- as.integer(grepl(genre, edx$genres))
}

#add the genre binary columns to edx data-set
```

```
edx2 <- cbind(edx, edx2)

#remove the "genres" column from the "edx2" (twin data-set) data frame.

edx2 <- edx2[, !(names(edx2) %in% c("genres"))]


#Conver "timestamp into a more understandable variable. In this case, dates

edx <- edx %>% mutate(date = as_datetime(timestamp))
edx2 <- edx2 %>% mutate(date = as_datetime(timestamp))

##We add a new column to both edx and edx2 data frames (number of ratings per movie).

edx <- edx %>%
  group_by(movieId) %>%
  mutate(num_ratings = n()) %>%
  ungroup()

edx2 <- edx2 %>%
  group_by(movieId) %>%
  mutate(num_ratings = n()) %>%
  ungroup()

# View the new data frame with the genre columns, as well as the updated edx
head(edx2)
```

```
## # A tibble: 6 x 27
##   userId movieId rating timest~1 title  year Comedy Romance Action Crime Thril~2
##    <int>   <int>  <dbl>    <int> <chr> <int>  <int>   <int>  <int> <int>   <int>
## 1      1     122      5   8.39e8 Boom~  1992      1       1      0     0       0
## 2      1     185      5   8.39e8 Net,~  1995      0       0      1     1       1
## 3      1     292      5   8.39e8 Outb~  1995      0       0      1     0       1
## 4      1     316      5   8.39e8 Star~  1994      0       0      1     0       0
## 5      1     329      5   8.39e8 Star~  1994      0       0      1     0       0
## 6      1     355      5   8.39e8 Flin~  1994      1       0      0     0       0
## # ... with 16 more variables: Drama <int>, `Sci-Fi` <int>, Adventure <int>,
## #   Children <int>, Fantasy <int>, War <int>, Animation <int>, Musical <int>,
## #   Western <int>, Mystery <int>, `Film-Noir` <int>, Horror <int>,
## #   Documentary <int>, IMAX <int>, date <dttm>, num_ratings <int>, and
## #   abbreviated variable names 1: timestamp, 2: Thriller
```

```
genres
```

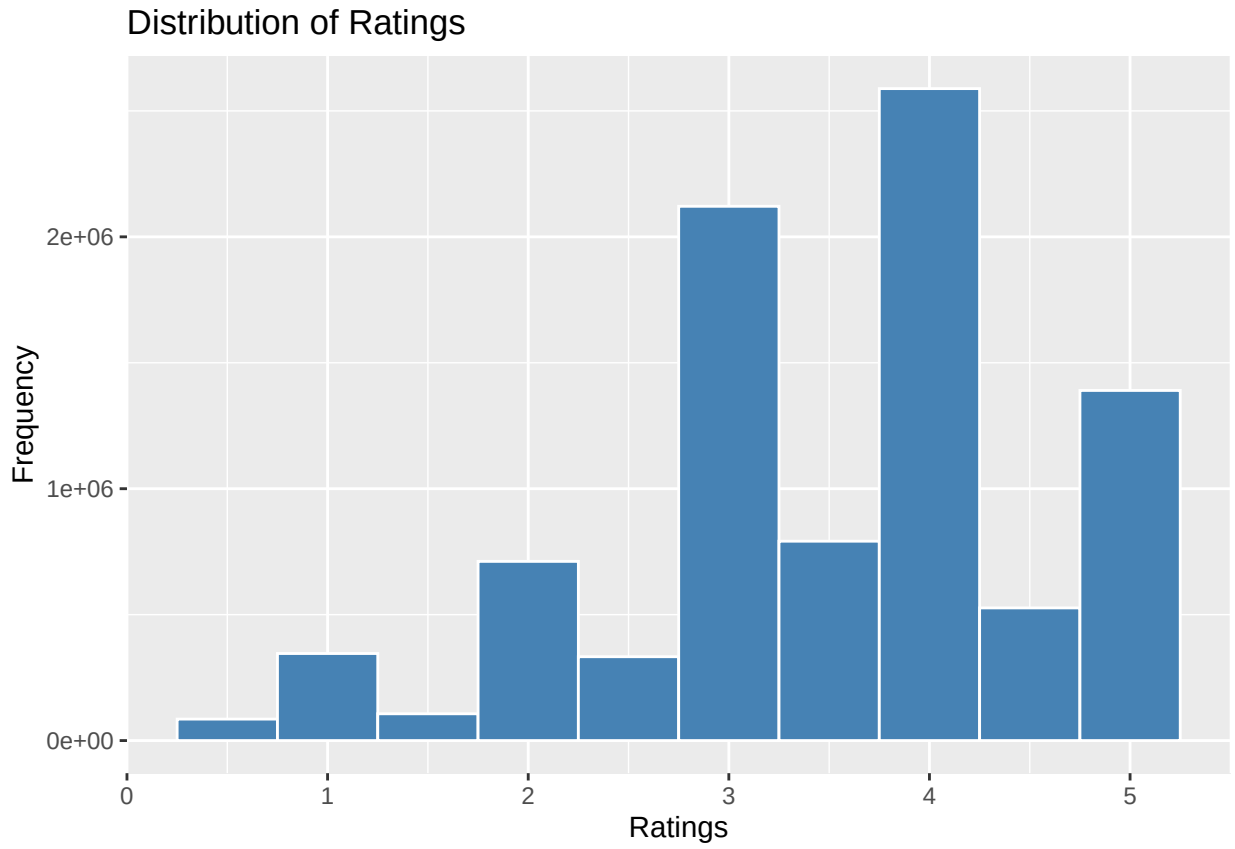
```
##  [1] "Comedy"      "Romance"     "Action"      "Crime"       "Thriller"
##  [6] "Drama"       "Sci-Fi"      "Adventure"   "Children"    "Fantasy"
## [11] "War"         "Animation"   "Musical"     "Western"     "Mystery"
## [16] "Film-Noir"   "Horror"      "Documentary" "IMAX"
```

After implementing these changes, we are satisfied with the data organization and the information displayed in the columns.

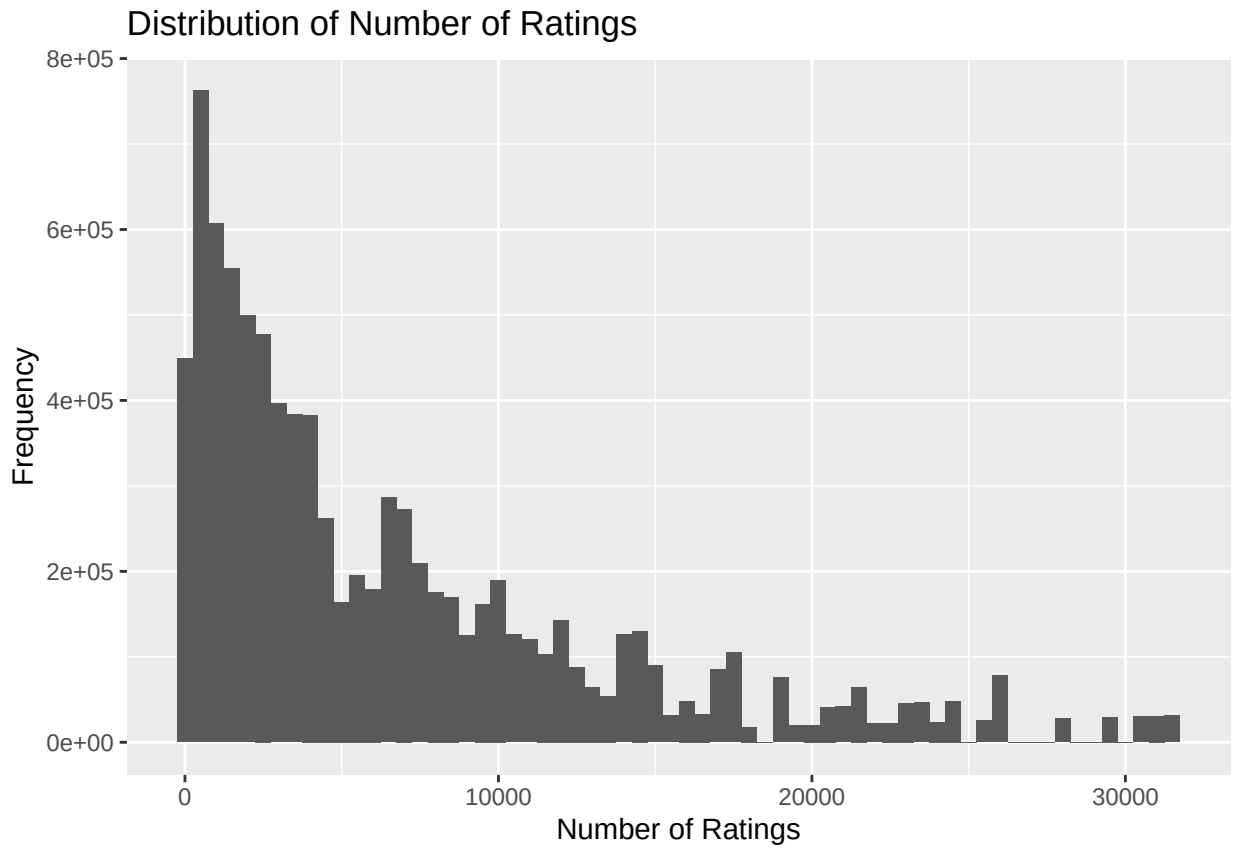
Data Exploration and Visualization

In this section, we explore and visualize our data to gain insights and identify patterns. By examining our data visually, we can better understand relationships and trends that might be hidden in the raw data.

```
#Histogram of frequency for each possible rating (1 to 5)

edx %>% ggplot(aes(x = rating)) +
  geom_histogram(binwidth = 0.5, fill = "steelblue", color = "white") +
  labs(title = "Distribution of Ratings", x = "Ratings", y = "Frequency")
```




```
#Histogram of frequency for the number of ratings

ggplot(data = edx, aes(x = num_ratings)) +
  geom_histogram(binwidth = 500) +
  xlab("Number of Ratings") +
  ylab("Frequency") +
  ggtitle("Distribution of Number of Ratings")
```



From this simple data visualization on the ratings and number of ratings frequencies, we can get at least two insights.

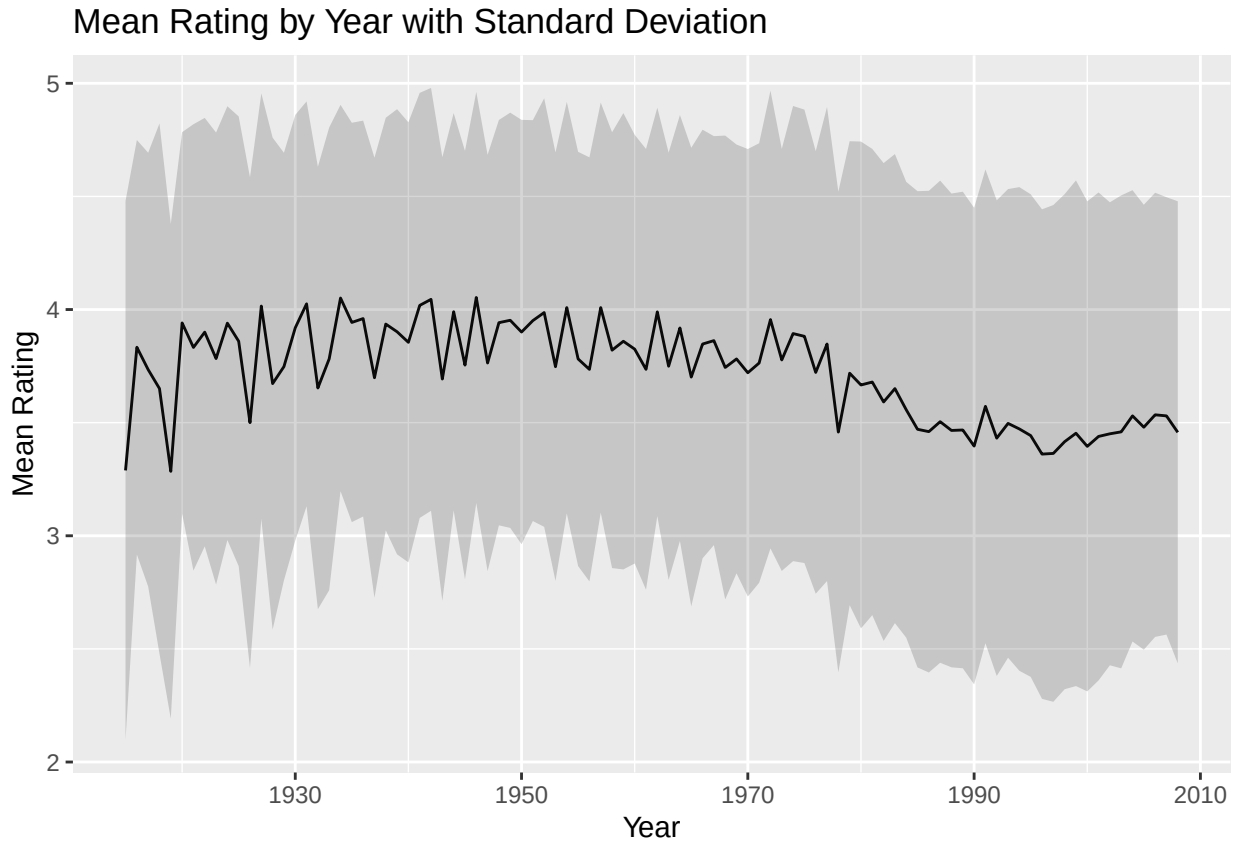
- Users tend to give more positive results to movies, meaning that a rating lower than 3 is rather improbable in comparison with higher ratings. Also, there are almost none "0" ratings present on the data.
- There is a high variability in terms of films popularity. There are blockbusters that have been rated more than 20 thousands of times, as well as movies with only a few hundred ratings.

Then, for the sake of length, and not make the report longer than necessary, we will try to summarize the subsequent analysis in the following way. The plots below, intend to show the effects of different variables on the ratings (as we are intending to create a model to predict it). In this sense, we will mostly present the following analyzes:

- **Ratings vs "year of release".** Mean rating by year, showing an indication of the variability in terms of the standard deviation. This is done through a "line plot".
- **Date/Timestamp vs Rating.** We clustered the rating date on 10 different groups, and show the mean rate and variability for each cluster (organized from earliest to latest date). This is done through a box plot.
- **MovieId/Number of Ratings vs Rating.** A scatter plot shows the number of ratings and average rating for each movie. We consider only movies with 100+ ratings to optimize for computing time and reduce "noise" coming from the data.
- **UserId/Number of Ratings vs Rating.** Same logic of the previous plot, but applied to users with 50+ ratings (have rated more than 50 movies).

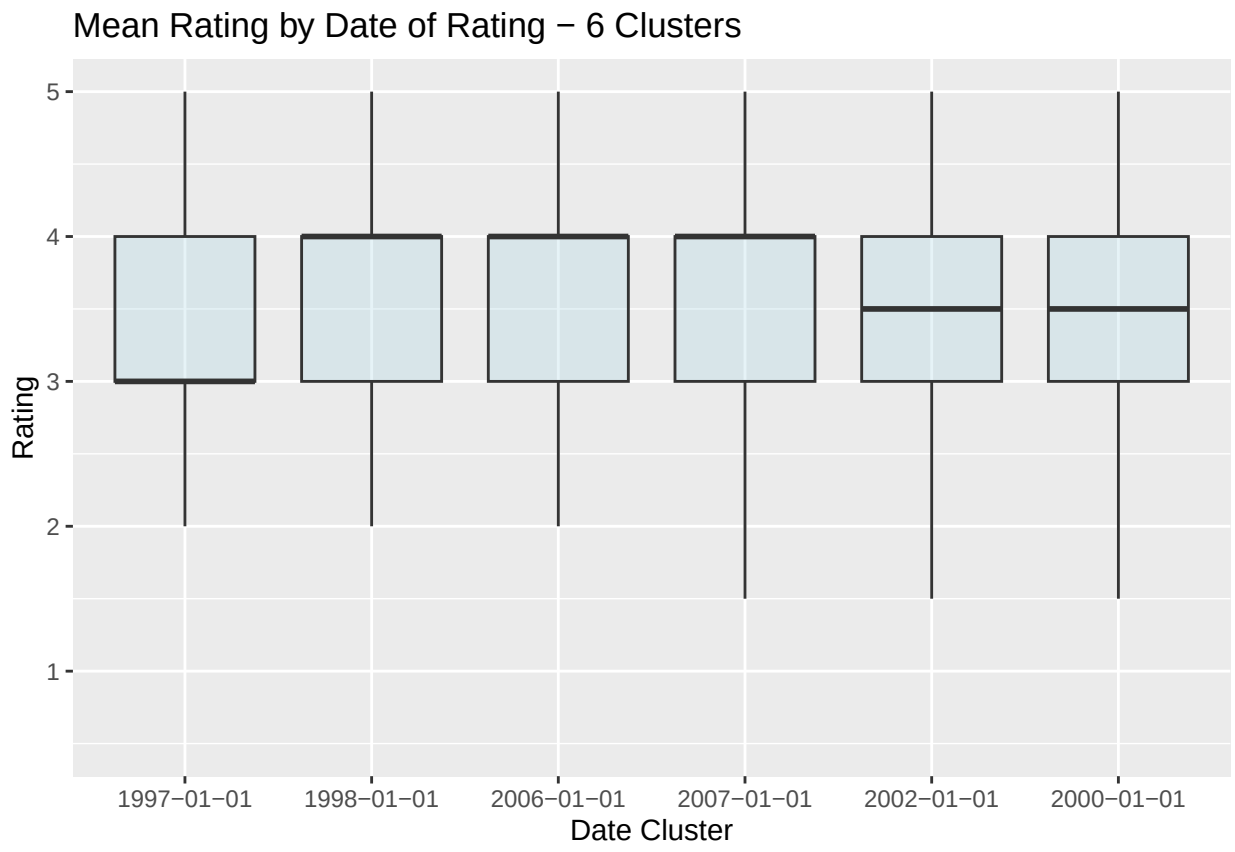
- **Genre vs Rating.** We follow two different approaches. Analyze ratings distribution for each genre (e.g "Comedy", "Horror", etc.), and analyze the distribution for the most popular combinations of genres.
  - Individual genres distribution is shown through a bar plot.
  - Most popular genres combinations are shown through a scatter plot.

```
#mean rating line vs year - showing years variability. Useful to explore effect of year on rating.
edx %>% group_by(year) %>%
  summarise(mean_rating = mean(rating), sd_rating = sd(rating)) %>%
  ggplot(aes(x = year, y = mean_rating)) +
  geom_line() +
  geom_ribbon(aes(ymin = mean_rating - sd_rating, ymax = mean_rating + sd_rating), alpha = 0.2) +
  labs(title = "Mean Rating by Year with Standard Deviation") +
  xlab("Year") +
  ylab("Mean Rating")
```

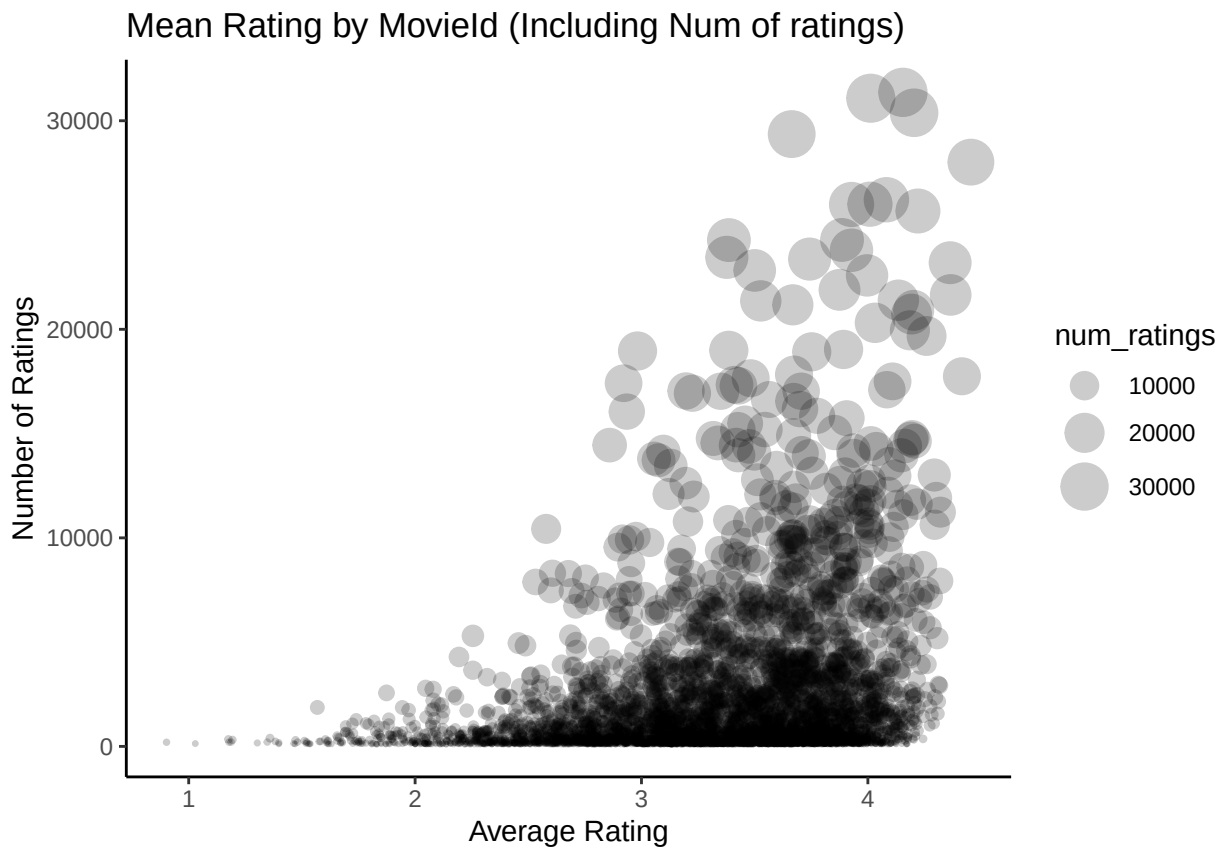



```
#creating 20 different clusters for date and the study it's effect on rating. We do this by plotting on

edx %>%
  mutate(date = round_date(date, unit = "year")) %>%
  mutate(cluster = cut(date, breaks = 6)) %>% #Create 20 clusters of date
  group_by(cluster) %>%
  mutate(cluster_order = min(date)) %>% # get the earliest date in each cluster
  ungroup() %>%
```

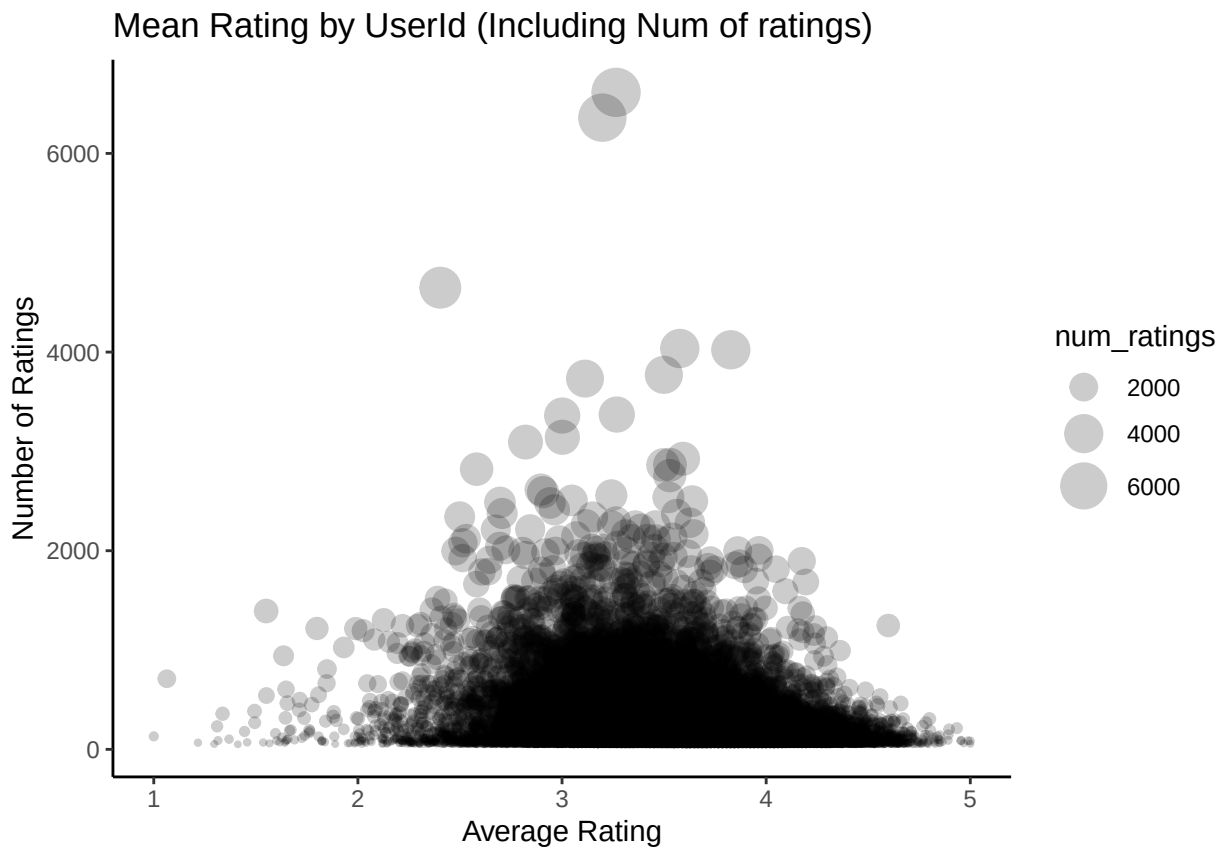
```
  arrange(cluster_order) %>% # sort the clusters based on the earliest date in each cluster
  ggplot(aes(x = cluster, y = rating)) +
  geom_boxplot(fill = "light blue", alpha = 0.3, outlier.alpha = 0) +
  labs(x = "Date Cluster", y = "Rating", title = "Mean Rating by Date of Rating - 6 Clusters") +
  scale_x_discrete(labels = function(x) gsub("\\..*", "", as.character(x)))
```



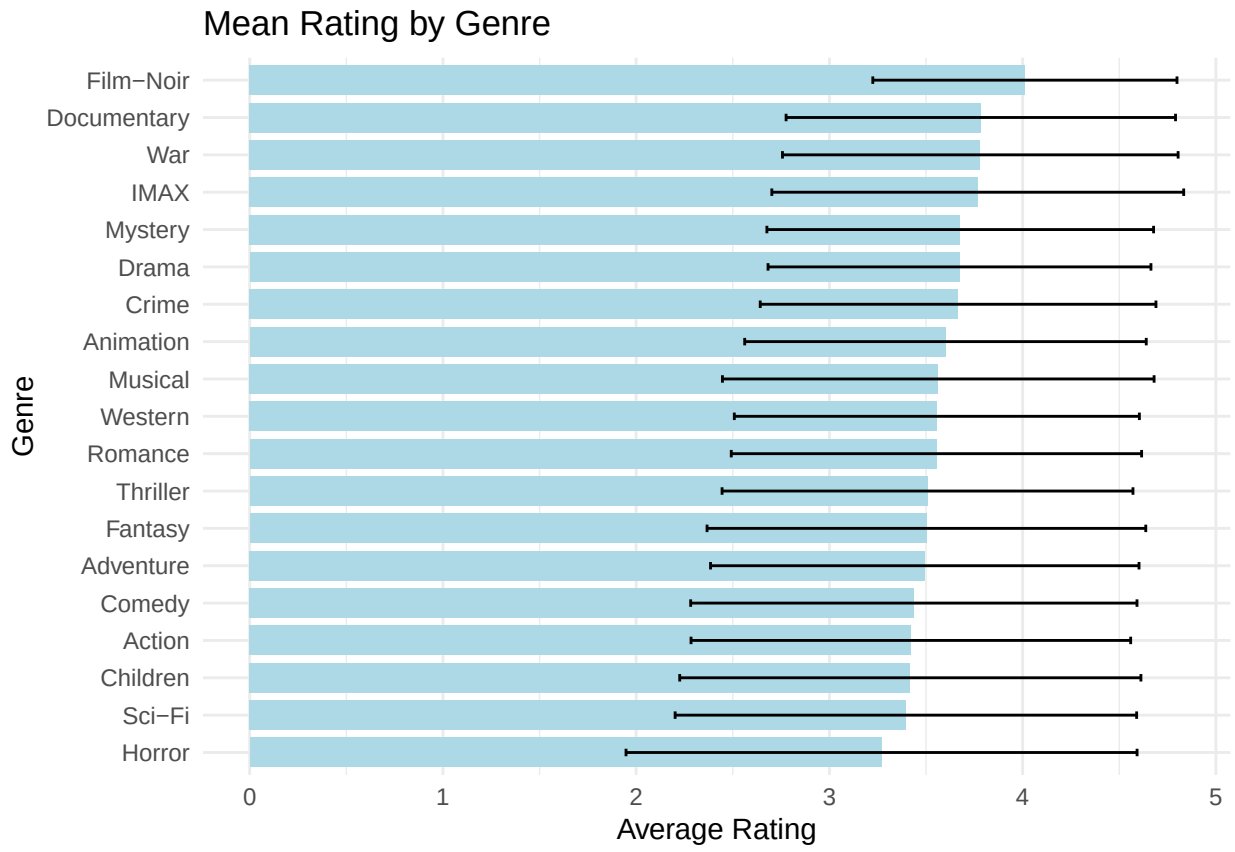

```
#plot of movieID vs ratings. We start by summarizing (mean rate) by MovieID. Then we only keep movies w

edx %>%
  group_by(movieId) %>%
  summarise(avg_rating = mean(rating),
            num_ratings = n()) %>%
  filter(num_ratings > 100) %>% # Only keep movies with more than 50 ratings
  ggplot(aes(x = avg_rating, y = num_ratings, size = num_ratings)) +
  geom_point(alpha = 0.2) +    # we do some trials to find the right alpha and max_size
  scale_size_area(max_size = 8) +
  labs(x = "Average Rating", y = "Number of Ratings", title = "Mean Rating by MovieId (Including Num of
  theme_classic()
```



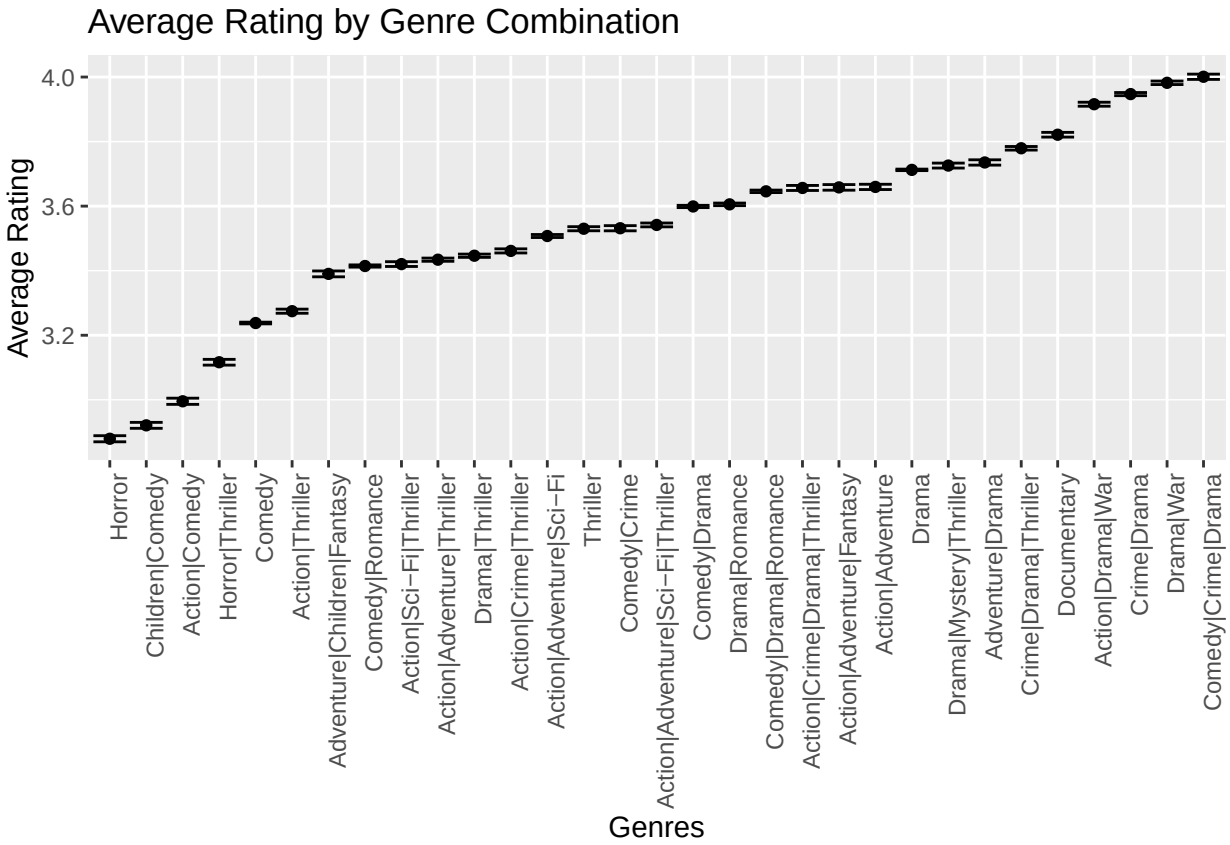

```
#Similarly, we plot of userID vs rating. We start by summarizing (mean rate) by userID.
edx %>% group_by(userId) %>%
  summarise(avg_rating = mean(rating),
            num_ratings = n()) %>%
  filter(num_ratings > 50)  %>% # Only keep users with more than 50 ratings
ggplot(aes(x = avg_rating, y = num_ratings, size = num_ratings)) +
  geom_point(alpha = 0.2) +
  scale_size_area(max_size = 8) +
  labs(x = "Average Rating", y = "Number of Ratings", title = "Mean Rating by UserId (Including Num of
  theme_classic()
```




```
#summarize the average rating per genre and analyze it

# create a data frame with the average rating and count per genre
genre_data <- edx2 %>%
  select(-c(userId, movieId, timestamp, title, year, num_ratings, date)) %>%
  pivot_longer(cols = -rating, names_to = "genre", values_to = "has_genre") %>%
  filter(has_genre == 1) %>%
  select(-has_genre)

genre_summary <- genre_data %>%
  group_by(genre) %>%
  summarise(avg_rating = mean(rating),
            var_rating = var(rating),
            count = n()) %>%  filter(count >= 1000) %>%
  mutate(genre = reorder(genre, avg_rating))

# Plot the mean rating and variance for each genre
ggplot(genre_summary, aes(x = genre, y = avg_rating, ymin = avg_rating - var_rating, ymax = avg_rating +
  geom_col(fill = "lightblue", width = 0.8, position = position_dodge()) +
  geom_errorbar(width = 0.2, position = position_dodge(0.8)) +
  labs(x = "Genre", y = "Average Rating", title = "Mean Rating by Genre") +
  coord_flip() +
  theme_minimal()
```




```
##Alternatively, we can do a similar exercise understanding "genres" as 2500+ unique factors

edx %>%
  group_by(genres) %>%
  summarize(n = n(), avg = mean(rating), se = sd(rating)/sqrt(n())) %>%
  filter(n >= 50000) %>%
  mutate(genres = reorder(genres, avg)) %>%
  ggplot(aes(x = genres, y = avg, ymin = avg - 2*se, ymax = avg + 2*se)) +
  geom_point() +
  geom_errorbar() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  labs(x = "Genres", y = "Average Rating",
       title = "Average Rating by Genre Combination")
```



Having visualized the effects of each variable on ratings we can draw the following insights.

- We confirm that movies that have been rated more often, tend to have a significant higher rating. Movies with ratings below 2 are usually considerably less popular.
- Both, year of release and date of rating have some effect on rating average, but the overall mean is always close to 3.5. However, it is debatable how much of the variability could be explain by those factors (year and date).
- Overall, variables as UserId, MovieId and Genres seem to have the stronger influence over rating. Certain users tend to give lower or higher reviews in general. Similarly, genres as "Horror" and "Children" movies tend to have a lower rating than genres as "Drama" or "War".
- When analyzing the effect of genres combinations, it is clear the strong effect on the reviews. Nonetheless, it might have a correlation with MovieId, as certain combinations might apply to specific *Blockbuster movies.*

Modeling / Machine Learning

Movie recommendation systems are well-documented case studies, as streaming platforms like Netflix, Amazon Prime, Disney+, and others use machine learning tools to provide relevant content to their users. Some of the most common R packages for such analyses are "Recommenderlab" and "Recosystem", which we will revisit later in the report.

To explore more intuitive tools, we will start by creating a simplified model using the variables UserId, MovieId, and Genres, as these are the most influential factors based on previous visualizations.

Linear Model - User, Movie and Genres Effect

In the book "Irizarry, R. A. (2019). Introduction to data science: Data analysis and prediction algorithms with R. CRC Press", provided in the EdX - Data Science Program (in collaboration with HarvardX), a simplified mathematical formulation is suggested in the form of:

$$u, i = \mu + b_i + b_u + \sum_{k=1}^{K} x_{ku,i}\beta_k + \epsilon_{u,i}, \quad \text{with } x_{ku,i} = \begin{cases} 1, & \text{if } g_{u,i} \text{ is genre } k \\ 0, & \text{otherwise} \end{cases}$$

Where i represents each specific movie, u the user, and k the different genres. We start by checking the accuracy of our prediction if we assume the global mean for all ratings. This will be used as a *benchmark* for future model comparisons.

Only Global Mean as Predictor

```
#We start by dividing our data-set into train and test data.
test_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1, list = FALSE)
edx_train <- edx[-test_index,] #train data
edx_test <- edx[test_index,] #test data

global_mean <- mean(edx_train$rating) #estimate the global mean of train data
n <- nrow(edx_test)
predicted_ratings_onlyMean <- rep(global_mean, n) #all predictions are the global mean
actual_ratings <- edx_test$rating
rmse_onlyMean <- RMSE(predicted_ratings_onlyMean, actual_ratings)
print(rmse_onlyMean)
```

```
## [1] 1.060295
```

Global Mean + MovieId Correction

We now add an additional parameter where we correct based on the mean of each individual movie. We subtract the global mean from every movieId rating average, and then use it to correct our predictions on the test data.

```
# Calculate movie biases
movie_biases <- edx_train %>%
  group_by(movieId) %>%
  summarize(movie_bias = mean(rating - global_mean))

# Make predictions on test data using MovieId bias correction
test_predictions_movieBias <- edx_test %>%
  left_join(movie_biases, by = "movieId") %>%
  mutate(movie_bias = if_else(is.na(movie_bias), 0, movie_bias),
         predicted_rating = global_mean + movie_bias)

rmse_movieBias <- RMSE(edx_test$rating, test_predictions_movieBias$predicted_rating)
print(rmse_movieBias)
```

```
## [1] 0.9438299
```

We observe a clear improvement in our predictions using RMSE. RMSE (Root Mean Square Error) is a commonly used metric to measure the difference between predicted and actual values. It calculates the square root of the average squared difference between the predicted and actual values. In simple words, the lower it is, the more accurate our prediction is.

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

We implement a similar approach now for the userId. Subtract global mean from userId rating averages

```
#Calculate user bias following the previous approach
user_biases <- edx %>%
  group_by(userId) %>%
  summarize(user_bias = mean(rating - global_mean))
# Make predictions on test data using MovieId bias correction
test_predictions_UserMovie <- edx_test %>%
  left_join(movie_biases, by = "movieId") %>%
  left_join(user_biases, by = "userId") %>%
  mutate(movie_bias = if_else(is.na(movie_bias), 0, movie_bias),
         user_bias = if_else(is.na(user_bias), 0, user_bias),
         predicted_rating = global_mean + movie_bias + user_bias)

rmse_movieUserBias <- RMSE(edx_test$rating, test_predictions_UserMovie$predicted_rating)
print(rmse_movieUserBias)
```

```
## [1] 0.8783062
```

There is a substantial improvement just by making use of the movie and user averages to correct our initial prediction. Nonetheless, we have a small issue when correcting for movieId bias. the same weight to the bias term for every movie, regardless of the number of ratings each movie has. This can lead to overfitting, especially for movies with a limited number of ratings, as their bias term is heavily influenced by a small number of observations. To address this issue, we can employ regularization techniques for the movie bias term.

Regularizing the movie bias term is essential to prevent overfitting and improve the model's reliability, especially for movies with a limited number of ratings. Regularization adds a penalty term during optimization, which shrinks the bias coefficients towards zero. This makes the model less sensitive to small fluctuations in the data.

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

```
# Create a function to calculate movie biases with regularization
movie_biases_regularized <- function(train_data, lambda) {
  train_data %>%
    group_by(movieId) %>%
    summarize(movie_bias = sum(rating - global_mean) / (n() + lambda))
}

# Define the range of lambda values to test
lambda_seq <- seq(0.1, 20, by = 0.1)
```

```r
# Initialize an empty data frame to store lambda and corresponding RMSE values
lambda_rmse <- data.frame(lambda = numeric(), rmse = numeric())

# Loop through the lambda values
for (lambda in lambda_seq) {
  # Calculate movie biases with regularization
  movie_biases_reg <- movie_biases_regularized(edx_train, lambda)

  # Make predictions on test data
  test_predictions_regul <- edx_test %>%
    left_join(movie_biases_reg, by = "movieId") %>%
    left_join(user_biases, by = "userId") %>%
    mutate(movie_bias = if_else(is.na(movie_bias), 0, movie_bias),
           user_bias = if_else(is.na(user_bias), 0, user_bias),
           predicted_rating = global_mean + movie_bias + user_bias)
}

rmse_regul <- RMSE(edx_test$rating, test_predictions_regul$predicted_rating)
print(rmse_regul)
```

```
## [1] 0.8772606
```

Again, we can see some modest improvement in the accuracy of our predictions. This happens because we incrementally improve our predictions for less popular movies. If we don't have much data, we assume a more conservative estimation (closer to the mean).

Finally, to implement the whole model explained at the beginning of the section, we will calculate the genre bias. To calculate the genre bias, we first determine the "bias" towards each genre and then perform a correction as the sum of all genre biases for each observation in the dataset. The code block provided calculates genre biases and estimates the sum of all genre biases for each observation in the dataset.

```r
#genre bias calculation
min_genre_ratings <- 2000  # Set the minimum number of ratings for a genre combination to avoid over fi

# Calculate genre biases ------------------------------------------------------
genre_biases <- edx2 %>%
  select(c("userId", "movieId", "rating", genres)) %>%
  gather("genre", "genre_present", genres) %>%
  filter(genre_present == 1) %>%
  group_by(genre) %>%
  filter(n() >= min_genre_ratings) %>%
  summarize(genre_bias = mean(rating - global_mean)) %>%
  select(all_of("genre"), genre_bias)
```

```
## Warning: Using an external vector in selections was deprecated in tidyselect 1.1.0.
## i Please use `all_of()` or `any_of()` instead.
##   # Was:
##   data %>% select(genres)
##
##   # Now:
##   data %>% select(all_of(genres))
##
## See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
```

```r
#this function takes each value of the "genres" column and breaks it into a genres vector. Then estimat

calculate_total_genre_bias <- function(genres_str, genre_biases_df) {
  genre_list <- strsplit(genres_str, "\\|")[[1]]
  genre_bias_values <- genre_biases_df$genre_bias[genre_biases_df$genre %in% genre_list]
  sum(genre_bias_values, na.rm = TRUE)
}
#follow the same approach to calculate the prediction. But this time we make use of the above function

test_predictions_withGenreBias <- edx_test %>%
  left_join(movie_biases_reg, by = "movieId") %>%
  left_join(user_biases, by = "userId") %>%
  mutate(movie_bias = if_else(is.na(movie_bias), 0, movie_bias),
         user_bias = if_else(is.na(user_bias), 0, user_bias),
         genre_bias = map_dbl(genres, calculate_total_genre_bias, genre_biases),
         predicted_rating = global_mean + movie_bias + user_bias + genre_bias)

rmse_genre <- RMSE(edx_test$rating, test_predictions_withGenreBias$predicted_rating)
print(rmse_genre)
```

```
## [1] 0.9060254
```

Upon running the code, we find that, contrary to our expectations, our estimation has not improved but has actually moved further away from the real values. Two main reasons for this result could be:

1. **Noise**: The genre biases might be introducing noise into the prediction model, which reduces its accuracy. This could happen if there is a high variance in the genre biases or if they do not capture relevant patterns in the data.
2. **Over fitting**: It is possible that the model is over fitting the data by trying to capture too many genre biases, leading to a more complex model that does not generalize well to the test dataset. This can result in a higher RMSE

Collaborative Filtering Methodologies / Recosystem and Recommenderlab

Recommenderlab and Recosystem are R packages designed to facilitate the implementation of collaborative filtering methods for building recommendation systems. In simple terms, these packages provide tools to create models that can recommend items to users based on the preferences of similar users.

Collaborative filtering methods work by leveraging the past behavior and preferences of users to make recommendations. However, a full exploration of these tools is beyond the scope of this report. Below are some sources that explain their application in R:

- *Fernández, A. (2016). Recosystem: Recommender System Using Parallel Matrix Factorization. R-bloggers. Retrieved from* ***https://www.r-bloggers.com/2016/07/recosystem-recommender-system-using-parallel-matrix-factorization/***
- *Vargas, S. (2021). Introduction to the recosystem package. R Foundation for Statistical Computing. Retrieved from* ***https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html****.*
- *Dasgupta, A. (2020). Building a Shiny Application: Step-by-Step Guide to Analytics Application Development. Retrieved from* ***https://ashgreat.github.io/analyticsAppBook/index.html***

We now implement a simple attempt using SVD (Matrix Factorization) through the Recommenderlab package. The provided code block demonstrates the process of creating a rating matrix, filtering the data for improved computing time, and implementing standard parameters based on literature.

```
#First we create the rating matrix (necessary input for using the recommender standard function). We fi

min_ratings <- 100

filtered_data <- edx %>%
  group_by(movieId) %>%
  filter(n() >= min_ratings) %>%
  ungroup() %>%
  group_by(userId) %>%
  filter(n() >= min_ratings) %>%
  ungroup()


y <- select(filtered_data, movieId, userId, rating) %>%
  pivot_wider(names_from = movieId, values_from = rating)
y <- as.matrix(y[,-1])

colnames(y) <- filtered_data %>% select(movieId, title) %>%
  distinct(movieId, .keep_all = TRUE) %>%
  right_join(data.frame(movieId=as.integer(colnames(y))), by = "movieId") %>%
  pull(title)

ratings_matrix <- as(y, "realRatingMatrix")

#We take standard parameters provided on literature.
given <- 15
n_folds <- 15

#translate the rating matrix in a format that can be used by the Recommender function
evaluation_scheme <- evaluationScheme(
  ratings_matrix,
  method='split',
  train= 0.9,
  k=n_folds,
  given= given)
```

```
## as(<dgCMatrix>, "dgTMatrix") is deprecated since Matrix 1.5-0; do as(., "TsparseMatrix") instead
```

```
ratings_train <- getData(evaluation_scheme, 'train')

ratings_test_known <- getData(evaluation_scheme, 'known')

ratings_test_unknown <- getData(evaluation_scheme, 'unknown')
#We proceed to train our SVD model with the rating matrix above.

SVD_model <- Recommender(
  data=ratings_train,
  method='SVD')
#Finally, we apply the model using the standard function predict()
```

```
svd_prediction <- predict(object = SVD_model, newdata = ratings_test_known, n = 10, type = "ratings")
#Now we calculate the accuracy of the model using the "unkown" data.
rmse_sdv <- calcPredictionAccuracy(x = svd_prediction, data = ratings_test_unknown, byUser = FALSE, meas

rmse_sdv <- as.numeric(rmse_sdv[1])
print(rmse_sdv)
```

```
## [1] 0.8902255
```

After running the code, we observe that our SVD model also performs worse than the simplified model. The reason is likely related to the chosen parameters, which may not be the most suitable for our data. Finding optimal parameters for an SVD model can be challenging, as it usually involves searching the hyperparameter space through methods like grid search or random search, followed by evaluating the model's performance using cross-validation or a separate validation set.

In this case, we proceed with the recosystem solution. This package includes an embedded function for parameter optimization, allowing us to obtain the "best" possible model for our data using the recosystem tools.

The code block demonstrates the process of converting the train and test sets into the recosystem input format, creating the model object, selecting the best tuning parameters, training the algorithm, and predicting the ratings for the test data.

```
# Convert the train and test sets into recosystem input format
train_data <- with(edx_train, data_memory(user_index = userId, item_index = movieId, rating = rating))
test_data <- with(edx_test, data_memory(user_index = userId, item_index = movieId, rating = rating))

# Create the model object
r <- recosystem::Reco()

# Select the best tuning parameters
opts <- r$tune(train_data,
               opts = list(dim = c(10, 20, 30),
# dim is number of factors
                           lrate = c(0.1, 0.2),
# learning rate
                           costp_l2 = c(0.01, 0.1),      #regularization for P factors
                           costq_l2 = c(0.01, 0.1),      #regularization for Q factors
                           nthread = 4, niter = 10))     # convergence controlled by number of iteratio

# Train the algorithm
r$train(train_data, opts = c(opts$min, nthread = 4, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9823   1.1049e+07
##    1       0.8761   8.9955e+06
##    2       0.8434   8.3478e+06
##    3       0.8210   7.9601e+06
##    4       0.8044   7.6925e+06
##    5       0.7918   7.4970e+06
##    6       0.7816   7.3527e+06
##    7       0.7730   7.2380e+06
##    8       0.7656   7.1429e+06
```

```
##    9       0.7592   7.0631e+06
##   10       0.7538   6.9977e+06
##   11       0.7487   6.9405e+06
##   12       0.7443   6.8908e+06
##   13       0.7403   6.8485e+06
##   14       0.7365   6.8082e+06
##   15       0.7331   6.7760e+06
##   16       0.7300   6.7453e+06
##   17       0.7272   6.7174e+06
##   18       0.7244   6.6912e+06
##   19       0.7220   6.6696e+06
```

```
# Predict the ratings for the test data
predicted_ratings <- r$predict(test_data)


# You can calculate the RMSE or other evaluation metrics using the true and predicted ratings
rmse_reco <- RMSE(edx_test$rating, predicted_ratings)
```

Results

We have now implemented various models and calculated the RMSE for each of them. The results are summarized in the provided Models_Performance data frame.

```
Models_Performance <- data.frame(
  Model = c(
    "Global Mean as Prediction",
    "Movie Bias",
    "Movie and User Bias",
    "Regularized Movie Bias + User Bias",
    "SVD Model",
    "Recosystem Model"
  ),
  RMSE = c(
    rmse_onlyMean,
    rmse_movieBias,
    rmse_movieUserBias,
    rmse_regul,
    rmse_sdv,
    rmse_reco
  )
)

# Print the results
print(Models_Performance)
```

```
##                               Model      RMSE
## 1          Global Mean as Prediction 1.0602947
## 2                         Movie Bias 0.9438299
## 3                Movie and User Bias 0.8783062
## 4 Regularized Movie Bias + User Bias 0.8772606
## 5                          SVD Model 0.8902255
## 6                   Recosystem Model 0.7870370
```

Upon examining the results, it is clear that the model created using the recosystem package is superior to our previous.

```
# Convert the train and test sets into recosystem input format
train_data <- with(edx, data_memory(user_index = userId, item_index = movieId, rating = rating))
test_data <- with(final_holdout_test, data_memory(user_index = userId, item_index = movieId, rating = ra

# Create the model object
r <- recosystem::Reco()

#Please notice that we avoid the calculation again for the optimal parameters due to computational timi

r$train(train_data, opts = c(opts$min, nthread = 4, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9724   1.2019e+07
##    1       0.8733   9.8873e+06
##    2       0.8396   9.1844e+06
##    3       0.8171   8.7553e+06
##    4       0.8011   8.4701e+06
##    5       0.7889   8.2685e+06
##    6       0.7790   8.1162e+06
##    7       0.7708   7.9950e+06
##    8       0.7638   7.8960e+06
##    9       0.7578   7.8159e+06
##   10       0.7527   7.7497e+06
##   11       0.7481   7.6931e+06
##   12       0.7438   7.6411e+06
##   13       0.7402   7.5980e+06
##   14       0.7368   7.5616e+06
##   15       0.7336   7.5256e+06
##   16       0.7308   7.4947e+06
##   17       0.7282   7.4681e+06
##   18       0.7257   7.4426e+06
##   19       0.7236   7.4214e+06
```

```
# Predict the ratings for the test data
predicted_ratings <- r$predict(test_data)


# You can calculate the RMSE or other evaluation metrics using the true and predicted ratings
rmse_final_prediction <- RMSE(final_holdout_test$rating, predicted_ratings)


# Add the final prediction to the Models_Performance dataframe
Models_Performance <- rbind(Models_Performance,
                            data.frame(Model = "** Recosystem Model - Final **",
                                       RMSE = rmse_final_prediction))

# Print the results
print(Models_Performance)
```

```
##                                 Model      RMSE
```

```
## 1          Global Mean as Prediction 1.0602947
## 2                          Movie Bias 0.9438299
## 3                Movie and User Bias 0.8783062
## 4 Regularized Movie Bias + User Bias 0.8772606
## 5                           SVD Model 0.8902255
## 6                    Recosystem Model 0.7870370
## 7     ** Recosystem Model - Final ** 0.7824847
```

Conclusion

In conclusion, we explored various models to predict movie ratings, starting with a global mean as a baseline, followed by the incorporation of movie bias, user bias, and regularized movie bias. We also experimented with an SVD model using the Recommenderlab package and a Recosystem model. Our analysis showed that the Recosystem model significantly outperformed all other approaches, achieving an RMSE of 0.7865 compared to the next best-performing model, the regularized movie bias with user bias model, which had an RMSE of 0.8770. The final prediction results from the Recosystem model indicate that it converges to an optimal solution, further validating its effectiveness as a recommendation system tool. Overall, the Recosystem package emerges as the most suitable choice for our dataset, providing superior prediction accuracy and offering embedded functions for parameter optimization.

Throughout the analysis, we observed that various variables have differing levels of influence on the predictions. For instance, incorporating movie bias and user bias into the model helped reduce the RMSE, suggesting that these variables play a significant role in determining a user's rating for a movie. However, the genre bias did not improve the model's performance, which might be due to the noise introduced or potential overfitting from trying to capture too many genre biases.