MovieLens Project Report

Author: Bouchikhi Alaa Eddine

Date: December 2024

Abstract

This report outlines the methodologies and outcomes of a capstone project from the HarvardX PH125.9 "Data Science" course offered on the edX platform. The project utilized the MovieLens dataset, which contains user ratings of movies. The data was initially divided into training and final holdout sets. An exploratory analysis of the training set revealed significant effects related to both movies and users. Subsequently, manual prediction models were constructed based on the biases derived from the mean ratings associated with each predictor variable and their combinations. To enhance the user-movie effect model, regularization techniques were applied to minimize the root mean square error (RMSE). Additionally, a recommendation model was developed using the recosystem package in R. This model was then employed to predict ratings for the final holdout set. The predicted ratings were compared to the actual ratings in the holdout set, resulting in a RMSE of 0.7792515, which is an improvement over the target RMSE of 0.8649.

Introduction

The final module of the HarvardX PH125.9 "Data Science" course requires participants to independently create and submit a capstone project that includes a quiz and two assignments.

The primary objective of the first assignment was to develop a movie recommendation system utilizing the MovieLens dataset, which comprises approximately ten million entries of movie ratings collected from users of the MovieLens online service between January 1995 and January 2009.

This report details the approach taken to complete the assignment. The Methods section elaborates on the processes involved in downloading, transforming, and analyzing the data. Several hypotheses were formulated regarding the relationships between predictor variables such as user, movie, genre, release year, rating year, and the number of ratings per movie and the response variable, which is the rating itself. Biases associated with each user, movie, and other relevant factors were evaluated as deviations from the average rating for each category

of predictor variable. The impact of these biases on enhancing naive predictions of movie ratings from the final test dataset was assessed using RMSE, which quantifies the mean difference between predicted and actual ratings. It was found that the effects of movies and users were the most significant factors in minimizing RMSE, along with the number of ratings per movie.

Following this exploratory analysis, a recommendation model from the recosystem package in R was optimized and trained on a sparse matrix structured in triplet format, consisting of user IDs, movie IDs, and ratings. This trained model was then used to predict ratings for user-movie combinations that had not been previously encountered in the final holdout test set. The performance of this model significantly surpassed that of the manual prediction methods, which accounted for biases in the predictor variables. The results of these findings are presented and discussed in the Results section.

Methods

Data Preparation

The dataset utilized for this project was generated using R code tailored from the course guidelines. Initially, the data was downloaded from the GroupLens website in the form of a compressed archive, totaling 63 MB..I downloaded the file and placed it on my device.

Following the download, the movies.dat and ratings.dat files were extracted from the compressed archive. These two data frames were then merged into a single movielens data frame, using movieId as the linking key. During this process, it was noted that four movies lacked ratings and were subsequently removed from the dataset to ensure data integrity.

```r
# Set the working directory
setwd("C:/Users/ASUS/Desktop/alaa")

# Define the name of the local file
dl <- "ml-10m.zip"

# Check if the file exists; if not, stop the execution with an error message
if (!file.exists(dl)) {
  stop("The file does not exist in the specified path.")
}

# Extract data and merge movies and ratings
movielens <- left_join(

  # Read and process the movies data
  read.table(
    text = gsub(
      x = readLines(con = unzip(dl, "ml-10M100K/movies.dat")),
      pattern = "::",
      replacement = ";",
      fixed = TRUE
    ),
    sep = ";",
    col.names = c("movieId", "title", "genres"),
    colClasses = c("integer", "character", "character"),
    quote = "",
    comment.char = "" # Remove any comments
  ),
```

```r
  # Read and process the ratings data
  read.table(
    text = gsub(
      x = readLines(con = unzip(dl, "ml-10M100K/ratings.dat")),
      pattern = "::",
      replacement = ";",
      fixed = TRUE
    ),
    sep = ";",
    col.names = c("userId", "movieId", "rating", "timestamp"),
    colClasses = c("integer", "integer", "numeric", "integer"),
    quote = ""
  ),

  # Merge the two data frames by movieId
  by = "movieId"
) |> na.omit() # Remove rows with missing values

# Display the first 6 rows of the merged data
head(movielens)
```

Data Partitioning

The dataset was divided into a training set edx_train comprising approximately nine million rows and a test set final_holdout_test containing the remaining one million rows. This partitioning was accomplished using the createDataPartition function from the caret package. To ensure the reproducibility of the partition and results, a fixed seed value of 1 was set for R's random number generator.

To construct the final test set, two semi-joins were performed to guarantee that all userId and movieId values present in the test set were also found in the training set.

The training set's purpose is to describe and explore the data while tuning and training the recommendation model. In contrast, the test set is utilized to assess the performance of the recommendation system against the course's grading criteria at the conclusion of the assignment. At no point was the test set employed for tuning or training the recommendation model; it was treated as "hidden data" available only after the model's completion.

Looking at the Data Set

To get what the data set looks like, we showed the first six lines of the joined movielens data set:

```
r

head(movielens)
```

This list had parts like:

movieId: A one-of-a-kind code for each film

title: The film's name

genres: The types linked with the film

userId: A one-of-a-kind code for each person

rating: How much a person liked the film

timestamp: When the like was noted

Checking the Data's Form

We looked at how the data set was made using the str() tool:

```r
str(movielens)
```

We found out that the dataset has **10,000,054** bits of data over six areas. This shows us what kind of data we have and what we need to fix before using it.

Size of the Dataset

We looked at the size of the dataset to see how big it is:

```r
dim(movielens)
```

The data shows that there are 10,000,054 rows and 6 columns. Data Details To look at the data better, key stats were made:

```r
summary(movielens)
```

The stats show key facts like:

movieId: Goes from 1 to **65,133**

userId: Goes from 1 to **71,567**

rating: Goes from **0.5** to **5.0**, with an average of about **3.51**

timestamp: Goes from **789,700,000** to **1,231,000,000**

Unique Info in Main Areas

The count of one-of-a-kind movies and users was found to see how mixed the data set is:

```r
unique_movies <- length(unique(movielens$title))
unique_users <- length(unique(movielens$userId))
cat("Number of unique movies:", unique_movies, "\n")
cat("Number of unique users:", unique_users, "\n")
```

Outcomes:

Count of Different Movies: 10,676

Count of Different Users: 69,878

Looking at Missing Data

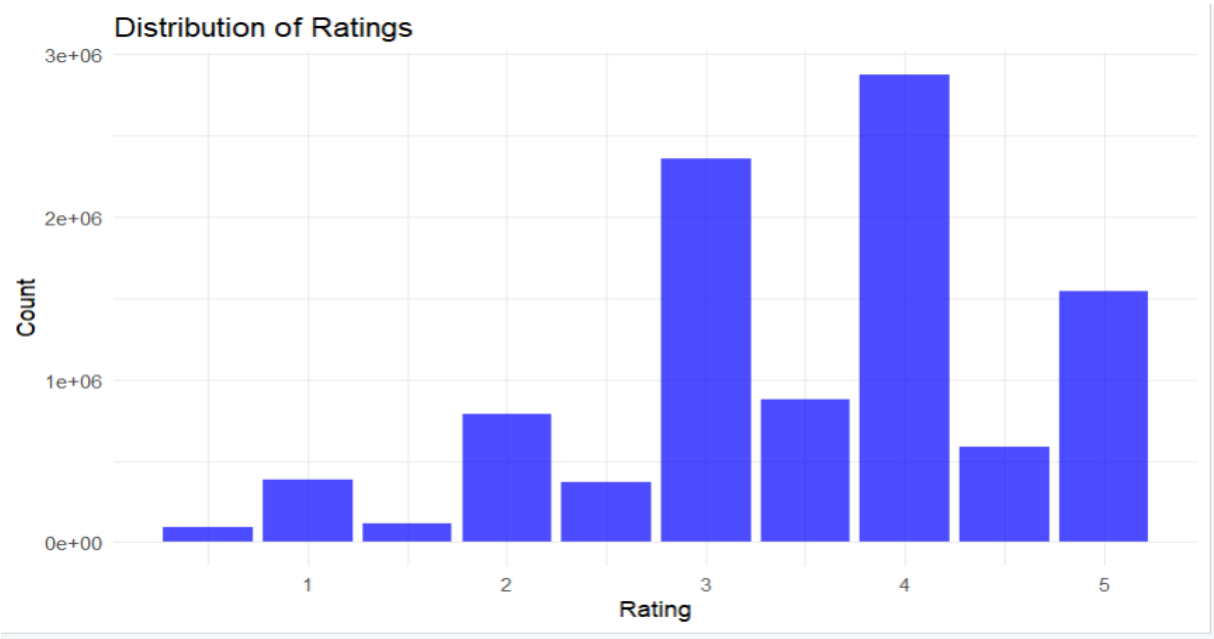
We checked for any missing data in the dataset:

```r
missing_values <- sapply(movielens, function(x) sum(is.na(x)))
print(missing_values)
```

This check showed that there were no missing values. This means all data was there.

How Ratings Spread Out To see how the ratings spread, we used the ggplot2 tool:

```r
library(ggplot2)

ggplot(movielens, aes(x = rating)) +
  geom_bar(fill = "blue", alpha = 0.7) +
  labs(title = "Distribution of Ratings", x = "Rating", y = "Count") +
  theme_minimal()
```



This bar chart illustrated the frequency of each rating, providing insights into user preferences.

Average Rating per Movie

The average rating for each movie was calculated to identify top-rated films:

```r
avg_rating <- movielens %>%
  group_by(title) %>%
  summarize(avg_rating = mean(rating), n_ratings = n()) %>%
  arrange(desc(avg_rating))
```

Top 10 Movies by Average Rating

The top 10 movies based on their average rating were displayed in a structured table format:

Rank	Title	Average Rating	Number of Ratings
1	Blue Light, The (Das Blaue Licht) (1932)	5.00	1
2	Fighting Elegy (Kenka erejii) (1966)	5.00	1
3	Satan's Tango (Sátántangó) (1994)	5.00	2
4	Shadows of Forgotten Ancestors (1964)	5.00	1
5	Sun Alley (Sonnenallee) (1999)	5.00	1
6	Constantine's Sword (2007)	4.75	2
7	Human Condition II, The (Ningen no joken II) (1959)	4.75	4
8	Human Condition III, The (Ningen no joken III) (1961)	4.75	4
9	More (1998)	4.75	8
10	Who's Singin' Over There? (a.k.a. Who Sings Over There)	4.75	4

The dataset was appropriately partitioned into a training set, comprising approximately nine million rows, and a test set of a million rows, which was done by using the CreateDataPartition from caret package. In order to ensure reproducibility, we set a seed value of 1 for the random number generator of R.

To construct the final test set, two semi-joins were performed to guarantee that all userId and movieId values in the test set are also present in the training set.

The training data will be relied upon to explore the data and fine-tune the recommendation model, whereas the test data will only be used after the model has been finalized to evaluate its performance against the rubric devised for grading at the course. The test data at no point was utilized to train or fine-tune the recommendation model; it was treated as "hidden data."

```r
# Set a seed for reproducibility
set.seed(1)

# Define the proportion for the test set (40% of the MovieLens data)
test_partition <- createDataPartition(
  y = movielens$rating,
  times = 1,
  p = 0.4,
  list = FALSE
)

# Create training and temporary test datasets
train_set <- movielens[-test_partition, ]
temp_test_set <- movielens[test_partition, ]

# Filter the temporary test set to include only valid userId and movieId from the training set
final_test_set <- temp_test_set |>
  semi_join(train_set, by = "movieId") |>
  semi_join(train_set, by = "userId")

# Incorporate any excluded rows back into the training set
train_set <- rbind(
  train_set,                                  # Original training data
  anti_join(temp_test_set, final_test_set)  # Rows excluded from test set
)

# Release memory by removing unused objects
rm(dl, test_partition, temp_test_set, movielens)
```

A random seed is set for the sake of repeatability, and we partition the dataset into a training set and a holdout test set.

We perform a semi-join to filter the test set, guaranteeing that the only valid userId and movieId values remain.

We proceeded to free up memory by removing unnecessary objects.

This schematic approach provides a lucid sense to follow the process of data partitioning while maintaining the basic structure of subsequent datasets for later analysis.

Exploratory analysis

After the aforementioned description of edx_train, the data was found to comprise ratings made by individual viewers for movies. There are 9,000,056 observations of 6 variables in the dataset, described consequently.

movield

Integer representation of the unique identifier of a movie, ranging from 1 to 65,133. There are 10,677 unique movies in the training set, whose ratings have been made on average 843 times.

```
Total Observations:  9000056
Total Variables:  6
Movie ID Range:  1 65133
Number of Unique Movies:  10677
Average Ratings per Movie:  842.9387
> 
```

title

A short way to show the movie's name and the year it came out, put in a pair of brackets at the end. The year it came out might help guess the movie's rating. So, it was taken out, put into a new number only form called year_movie, and cut from the movie's name by using the code below:

```r
# Define a function to extract year and clean title

extract_year_and_clean_title <- function(title) {
  # Extract the year from the end of the title
  year <- as.integer(str_extract(title, "\\d{4}$"))
  # Remove the year from the title
  clean_title <- str_remove(title, " \\d{4}$")
  return(c(year, clean_title))
}

# Apply the function to the title column
edx_train <- edx_train %>%
  rowwise() %>%
  mutate(
    year_movie = extract_year_and_clean_title(title)[1],
    title = extract_year_and_clean_title(title)[2]
  ) %>%
  ungroup()

# Display the first few rows to verify the changes
cat("Sample of the updated dataset:\n")
print(head(edx_train))
```

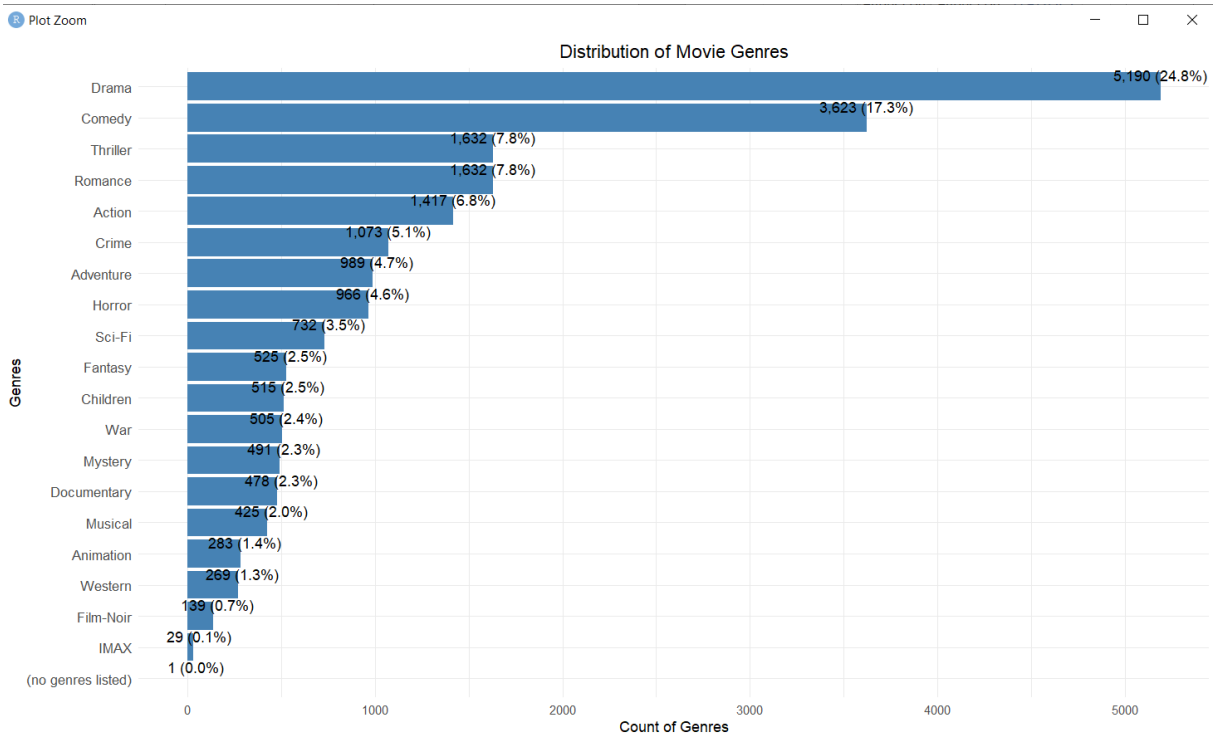
genres

A movie may fit in one or more kinds shown as a list with a pipe ("|") between each name. There are 20 different kinds, with one not set. When put all in a row, there are 797 ways to mix them in this list. Any kind or mix might help us guess what rating a movie will get.

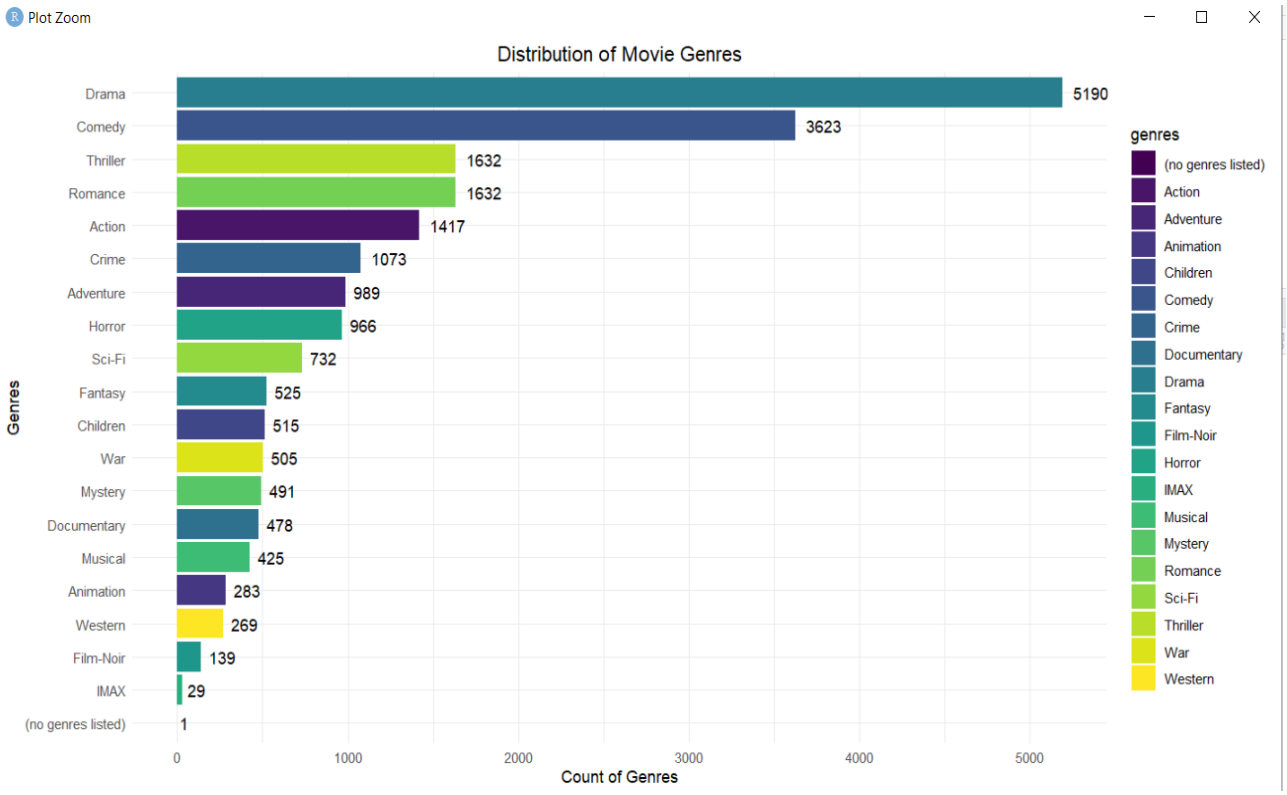
The help below lets us see how kinds are spread in the list of movies used to learn. The kind "drama" shows up the most by far, then "comedy," "thriller," and "romance" come next. The kinds "IMAX," "film-noir," and "Western" show up the least.

```r
# (Including one undefined genre)
unique_genres_count <- edx_train %>%
  select(genres) %>%
  unique() %>%
  separate_rows(genres, sep = "\\|") %>%
  n_distinct()

cat("Number of unique genres (including undefined):", unique_genres_count, "\n")

# 2. Count the number of concatenated genres
# (Total of 797 combinations)
concatenated_genres_count <- length(unique(edx_train$genres))
cat("Number of concatenated genres:", concatenated_genres_count, "\n")

# 3. Create a histogram of genres
# Count occurrences of each genre and sort
genre_counts <- edx_train %>%
  distinct(title, .keep_all = TRUE) %>%
  separate_rows(genres, sep = "\\|") %>%
  count(genres, sort = TRUE)

# Create a histogram
ggplot(genre_counts, aes(x = reorder(genres, n), y = n)) +
  geom_col(fill = "steelblue") +
  geom_text(
    aes(
      label = paste(
        format(n, big.mark = ","),
        " (",
        label_percent(accuracy = 0.1)(n / sum(n)),
        ")",
        sep = ""
      ),
      vjust = -0.5  # Adjust vertical position of labels
    )
  ) +
  coord_flip() +
  labs(
    title = "Distribution of Movie Genres",
    x = "Genres",
    y = "Count of Genres"
  ) +
  theme_minimal() +  # Use a minimal theme for better aesthetics
  theme(
    axis.text.y = element_text(size = 10),  # Adjust text size for better readability
    plot.title = element_text(hjust = 0.5)  # Center the title
  )
```

```
>
> cat("Number of unique genres (including undefined):", unique_genres_count, "\n")
Number of unique genres (including undefined): 20
>
> # 2. Count the number of concatenated genres
> # (Total of 797 combinations)
> concatenated_genres_count <- length(unique(edx_train$genres))
> cat("Number of concatenated genres:", concatenated_genres_count, "\n")
Number of concatenated genres: 797
>
```




```r
install.packages("viridis")

library(viridis)

# Create a horizontal bar chart with viridis colors
ggplot(genre_counts, aes(x = reorder(genres, n), y = n, fill = genres)) +
  geom_bar(stat = "identity") +
  scale_fill_viridis(discrete = TRUE) +  # Use viridis color scale
  geom_text(aes(label = n), hjust = -0.3) +  # Add counts on the bars
  coord_flip() +
  labs(
    title = "Distribution of Movie Genres",
    x = "Genres",
    y = "Count of Genres"
  ) +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5)  # Center the title
  )
```



userId

A number that stands as a user's own ID, from 1 to 71,567. From these IDs, there are 69,878 different users in the training set. Each one has given a score to about 129 movies on average.

```
> # Print the values
> cat("Number of users (userId):", user_id_count, "\n")
Number of users (userId): 71567
> cat("Average number of movies per user:", average_movies_per_user, "\n")
Average number of movies per user: 129
> cat("Total ratings:", total_ratings, "\n")
Total ratings: 69878
>
```

rating

This is a number that shows how much a user likes a movie. They pick a score from 0.5 to 5, going up by 0.5 each time. We have to guess this score when we get new data (the last test set). The picture below shows how the scores spread out in the training data. The most common score is 4 (28.8%). The least common is 0.5 (0.9%). The average score is 3.51. There is no zero score. # Picture of scores distribution

```r
# Create a histogram of ratings with a different style
edx_train %>%
  group_by(rating) %>%
  summarise(count = n()) %>%  # Count occurrences of each rating
  ggplot(aes(x = rating, y = count)) +
  geom_bar(stat = "identity", fill = "skyblue", color = "black") +  # Use geom_bar for a clearer look
  geom_text(aes(label = label_percent(accuracy = 0.1)(count / sum(count))),
            vjust = -0.5, size = 4) +  # Adjust label positioning and size
  scale_y_continuous(expand = expansion(mult = c(0, 0.1))) +  # Add space above bars
  coord_cartesian(clip = 'off') +  # Ensure labels are not clipped
  labs(title = "Distribution of Ratings",
       x = "Ratings",
       y = "Count of Ratings") +
  theme_minimal() +  # Use a minimal theme for a clean look
  theme(
    plot.title = element_text(hjust = 0.5, size = 16),  # Center title
    axis.text.x = element_text(size = 12),  # Adjust x axis text size
    axis.text.y = element_text(size = 12)   # Adjust y axis text size
  )
```

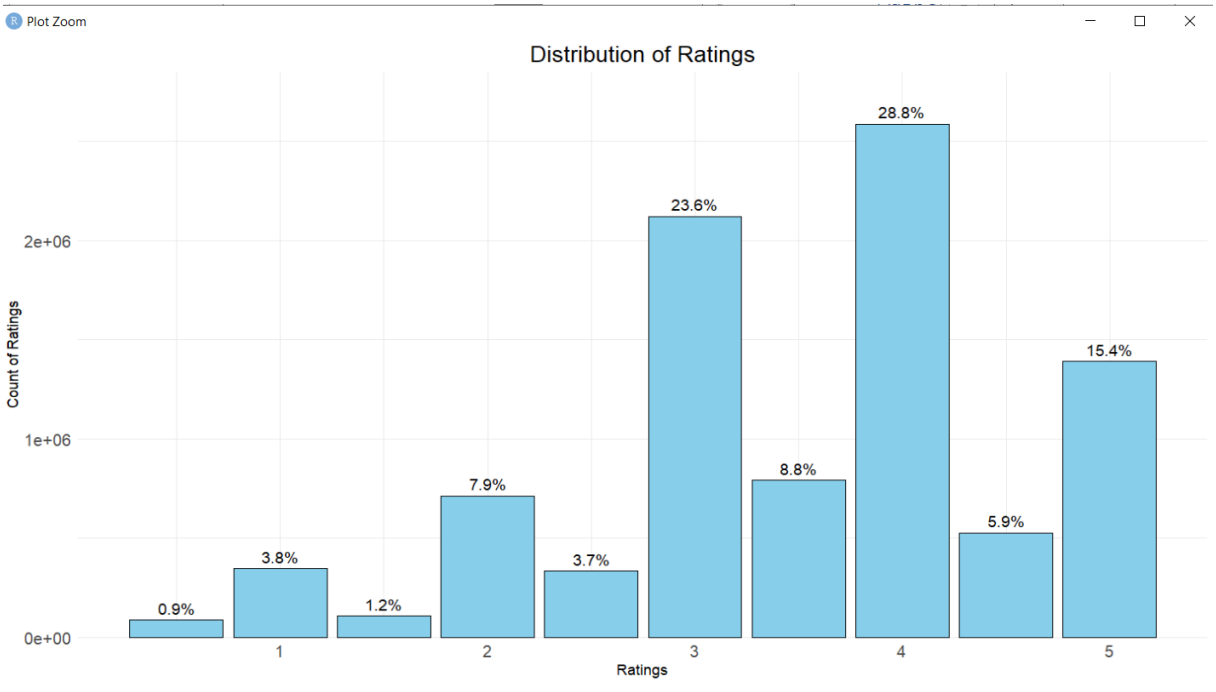


Figure 2 : Distribution of Ratings

Next, we looked at how many times each movie got rated. Some movies got a lot of rates, with the top one having 31,283 rates. On the other hand, many movies had few rates, with the lowest-rated movie getting just one rate. In fact, 125 movies were rated only once. Also, 2,685 movies were among the 25% at the bottom, getting 30 or fewer rates. The average rates per movie is 843. It's no shock that the movies with the most rates are the ones that are well-liked:

```
> # 10. Most Rated Movies
> most_rated_movies <- edx_train %>%
+     count(movieId, title, name = "n_ratings", sort = TRUE) %>%
+     head(10)
>
> kable(most_rated_movies)
```

movieId	title	n_ratings
296	Pulp Fiction	31283
356	Forrest Gump	31024
593	Silence of the Lambs, The	30393
480	Jurassic Park	29369
318	Shawshank Redemption, The	28058
110	Braveheart	26263
589	Terminator 2: Judgment Day	26068
457	Fugitive, The	26016
260	Star Wars: Episode IV - A New Hope (a.k.a. Star Wars)	25764
150	Apollo 13	24359

On the other hand, the movies with the fewest ratings are not well-known:

```
>
> # 11. Least Rated Movies
> least_rated_movies <- edx_train %>%
+     count(movieId, title, name = "n_ratings", sort = TRUE) %>%
+     tail(10)
>
> kable(least_rated_movies)
```

	movieId	title	n_ratings
10668	64754	Love	1
10669	64897	Mr. Wu	1
10670	64926	Battle of Russia, The (Why We Fight, 5)	1
10671	64944	Face of a Fugitive	1
10672	64953	Dirty Dozen, The: The Fatal Mission	1
10673	64976	Hexed	1
10674	65006	Impulse	1
10675	65011	Zona Zamfirova	1
10676	65025	Double Dynamite	1
10677	65027	Death Kiss, The	1

The chart down here shows how many movies got how many ratings. It proves that some movies get just a few ratings (on the left side of the chart), while only a few movies get lots of ratings (on the right end).

```r
# Frequency of ratings with enhanced visualization
edx_train %>%
  count(movieId, name = "ratings_per_movie") %>%
  count(ratings_per_movie, name = "frequency") %>%
  ggplot(mapping = aes(x = ratings_per_movie, y = frequency)) +
  geom_vline(xintercept = nrow(edx_train) / n_distinct(edx_train$movieId),
             linetype = "dashed", color = "red", size = 1) +
  annotate(
    geom = "text",
    label = format(round(nrow(edx_train) / n_distinct(edx_train$movieId), 0), nsmall = 0),
    x = nrow(edx_train) / n_distinct(edx_train$movieId) - 150,
    y = -10,
    color = "red"
  ) +
  geom_point(aes(color = frequency), size = 3, stroke = 1.5, shape = 21, fill = "white") +
  scale_color_gradient(low = "blue", high = "orange") +  # Color gradient for frequency
  scale_x_log10() +
  labs(x = "Count of Ratings per Movie", y = "Count of Movies",
       title = "Distribution of Ratings per Movie") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5, size = 16, face = "bold"),
    axis.title = element_text(size = 14),
    axis.text = element_text(size = 12)
  )
```

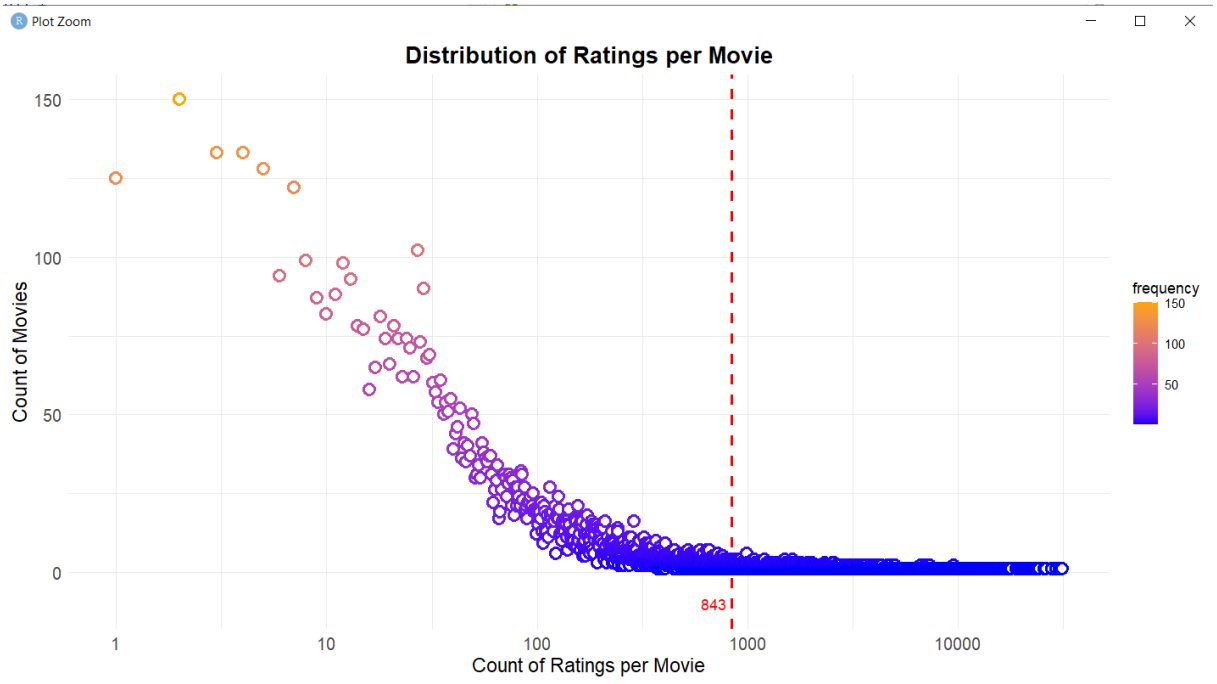


Figure 3: Relationship Between Movie Ratings and Frequency

Few people rate some movies, which makes us trust those ratings less. Here is a list of top-rated movies that not many people know about, and not many have rated:

Top 10 Highest-Rated Movies

movieId	title	mean_rating	n_rating
5194	Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva)	5.000000	3
33264	Satan's Tango (Sátántangó)	5.000000	2
42783	Shadows of Forgotten Ancestors	5.000000	1
51209	Fighting Elegy (Kenka erejii)	5.000000	1
53355	Sun Alley (Sonnenallee)	5.000000	1
64275	Blue Light, The (Das Blaue Licht)	5.000000	1
65001	Constantine's Sword	4.750000	2
4454	More	4.714286	7

```
> # Highest-rated movies (top n)
> highest_rated_movies <- edx_train %>%
+     group_by(movieId, title) %>%
+     summarize(mean_rating = mean(rating), n_rating = n(), .groups = "drop") %>%
+     arrange(desc(mean_rating)) %>%
+     head(10)
>
> # Create a simple table
> highest_rated_movies %>%
+     kable(caption = "Top 10 Highest-Rated Movies") %>%
+     kable_styling("basic")
>
```

If we take out the bottom 25th percentile of movies by count of ratings, the same test gives us a more likely list of top-rated movies.

Top 10 Highest-Rated Movies with More Than 30 Ratings

movieId	title	mean_rating	n_rating
318	Shawshank Redemption, The	4.460101	28058
858	Godfather, The	4.413570	17731
50	Usual Suspects, The	4.366210	21616
527	Schindler's List	4.365098	23228
922	Sunset Blvd. (a.k.a. Sunset Boulevard)	4.325902	2938
6896	Shoah	4.325581	86
904	Rear Window	4.319451	7948
912	Casablanca	4.317117	11264
2019	Seven Samurai (Shichinin no samurai)	4.316496	5177
1212	Third Man, The	4.315529	2930

```
> # Highest-rated movies with more than 30 ratings
> highest_rated_movies <- edx_train %>%
+     group_by(movieId, title) %>%
+     summarize(mean_rating = mean(rating), n_rating = n(), .groups = "drop") %>%
+     filter(n_rating > 30L) %>%
+     arrange(desc(mean_rating)) %>%
+     head(10)
>
> # Create a simple table
> highest_rated_movies %>%
+     kable(caption = "Top 10 Highest-Rated Movies with More Than 30 Ratings") %>%
+     kable_styling("basic")
>
```

In the same way, this list of the worst-rated movies also shows not well-known movies with too few ratings:

Bottom 10 Lowest-Rated Movies

movieId	title	mean_rating	n_rating
5805	Besotted	0.5000000	1
8394	Hi-Line, The	0.5000000	1
61768	Accused (Anklaget)	0.5000000	1
64999	War of the Worlds 2: The Next Wave	0.6666667	3
8859	SuperBabies: Baby Geniuses 2	0.8070175	57
7282	Hip Hop Witch, Da	0.8214286	14
61348	Disaster Movie	0.9054054	37
6483	From Justin to Kelly	0.9322917	192
3561	Stacy's Knights	1.0000000	1
4071	Dog Run	1.0000000	1

```
> # Lowest-rated movies (bottom n)
> lowest_rated_movies <- edx_train %>%
+     group_by(movieId, title) %>%
+     summarize(mean_rating = mean(rating), n_rating = n(), .groups = "drop") %>%
+     arrange(mean_rating) %>%  # Order by lowest to highest rating
+     head(10)  # Select the bottom 10 movies
>
> # Create a simple table
> lowest_rated_movies %>%
+     kable(caption = "Bottom 10 Lowest-Rated Movies") %>%
+     kable_styling("basic")
>
>
```

Once more, taking out the bottom 25% of low-rated movies gives us a new group of worst-rated films. The title was left out of the training set to save space and because it isn't a key factor in making predictions.

Lowest-Rated Movies Excluding the Bottom 25th Percentile

movieId	title	mean_rating	n_rating
8859	SuperBabies: Baby Geniuses 2	0.8070175	57
61348	Disaster Movie	0.9054054	37
6483	From Justin to Kelly	0.9322917	192
6371	Pokémon Heroes	1.0034014	147
3027	Slaughterhouse 2	1.1363636	33
3574	Carnosaur 3: Primal Species	1.1418919	74
4775	Glitter	1.1622024	336
1826	Barney's Great Adventure	1.1650943	212
5672	Pokemon 4 Ever (a.k.a. Pokémon 4: The Movie)	1.1714286	210
6587	Gigli	1.1853035	313

```
>
> # Exclude movies in the bottom 25th percentile of ratings
> lowest_rated_movies <- edx_train %>%
+     group_by(movieId, title) %>%
+     summarize(mean_rating = mean(rating), n_rating = n(), .groups = "drop") %>%
+     filter(n_rating > 30L) %>%  # Exclude movies with 30 or fewer ratings
+     arrange(mean_rating) %>%     # Order by lowest mean rating
+     head(10)                     # Select the bottom 10 movies
>
> # Create a simple table
> lowest_rated_movies %>%
+     kable(caption = "Lowest-Rated Movies Excluding the Bottom 25th Percentile") %>%
+     kable_styling("basic")
>
```

The data shown in the chart backs up what we see when we check the mean score for each group of movie scores. The chart here tells us that films with just a small number of scores (next to the up-down line) have a big range in scores (up-down). In contrast, films with a lot of scores tend to stay close to the most common score of 4.

```r
# Plot the relationship between rating count and rating score
edx_train %>%
  group_by(movieId) %>%
  summarize(mean_rating = mean(rating), n_rating = n(), .groups = "drop") %>%
  ggplot(mapping = aes(x = n_rating, y = mean_rating)) +
  geom_point(aes(color = mean_rating, size = n_rating), alpha = 0.7, shape = 21, fill = "white") +
  scale_color_gradient(low = "blue", high = "orange") +  # Color gradient based on mean rating
  scale_size_continuous(range = c(2, 6)) +  # Adjust point size based on number of ratings
  labs(x = "Count of Ratings per Movie", y = "Mean Rating", title = "Relationship Between Rating Count and Mean Rating") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5, size = 16, face = "bold"),
    axis.title = element_text(size = 14),
    axis.text = element_text(size = 12)
  )
```

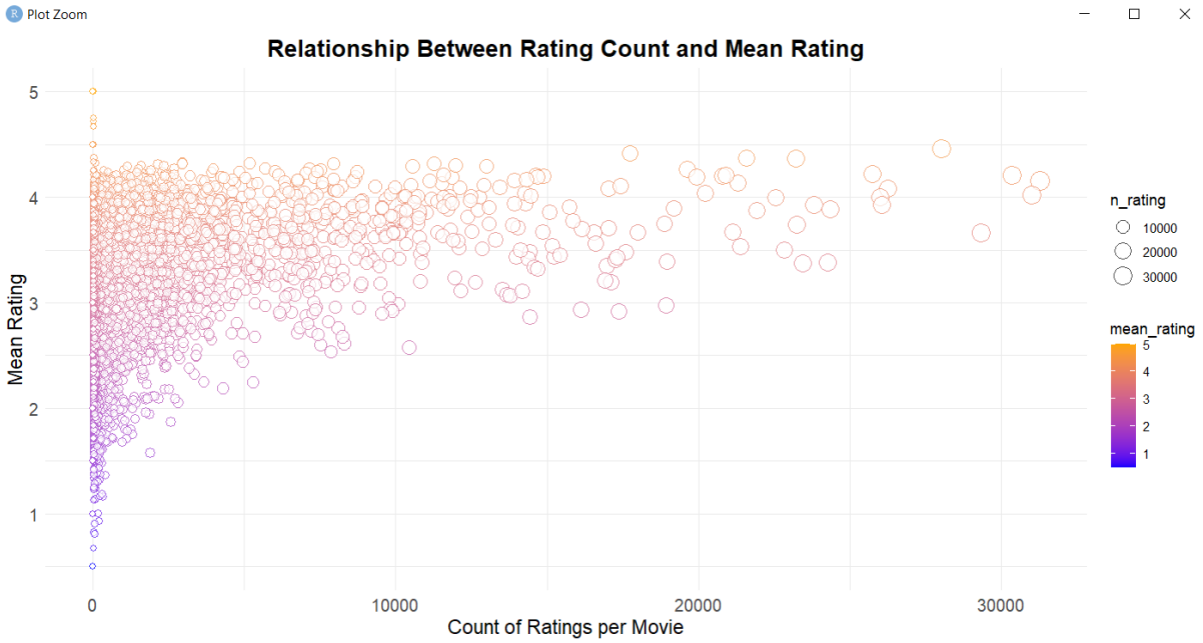


Figure 4: Mean Rating by Number of Ratings

In the end, we saw that people gave different scores. The graph below shows the average score each person gave (up and down) against how many times they rated (left and right). People who gave just a few ratings had scores that went up and down a lot. But those who rated a lot usually gave scores around the usual 3.51.

```
# Plot ratings by reviewer
edx_train %>%
  group_by(userId) %>%
  summarize(mean_rating = mean(rating), n_rating = n(), .groups = "drop") %>%
  ggplot(mapping = aes(x = n_rating, y = mean_rating)) +
  geom_point(aes(color = mean_rating, size = n_rating), alpha = 0.7, shape = 21, fill = "white") +
  scale_color_gradient(low = "green", high = "red") +  # Color gradient based on mean rating
  scale_size_continuous(range = c(2, 6)) +  # Adjust point size based on number of ratings
  labs(x = "Count of Ratings per User", y = "Mean Rating", title = "Ratings by Reviewer") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5, size = 16, face = "bold"),
    axis.title = element_text(size = 14),
    axis.text = element_text(size = 12)
  )
```

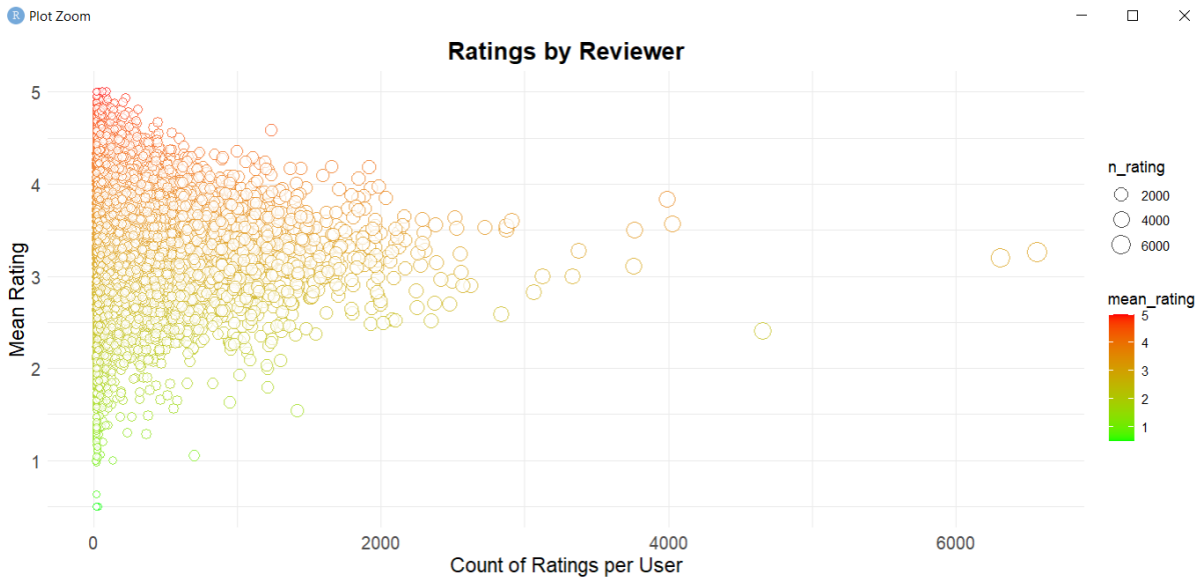


Figure 5: Scatterplot of User Mean Ratings

To wrap up our look into the movie ratings, we can say that a good model to guess ratings should consider:

- An effect from the movie itself, as some films get higher ratings;

- An effect based on how many ratings a movie has, since more ratings might mean better predictability;

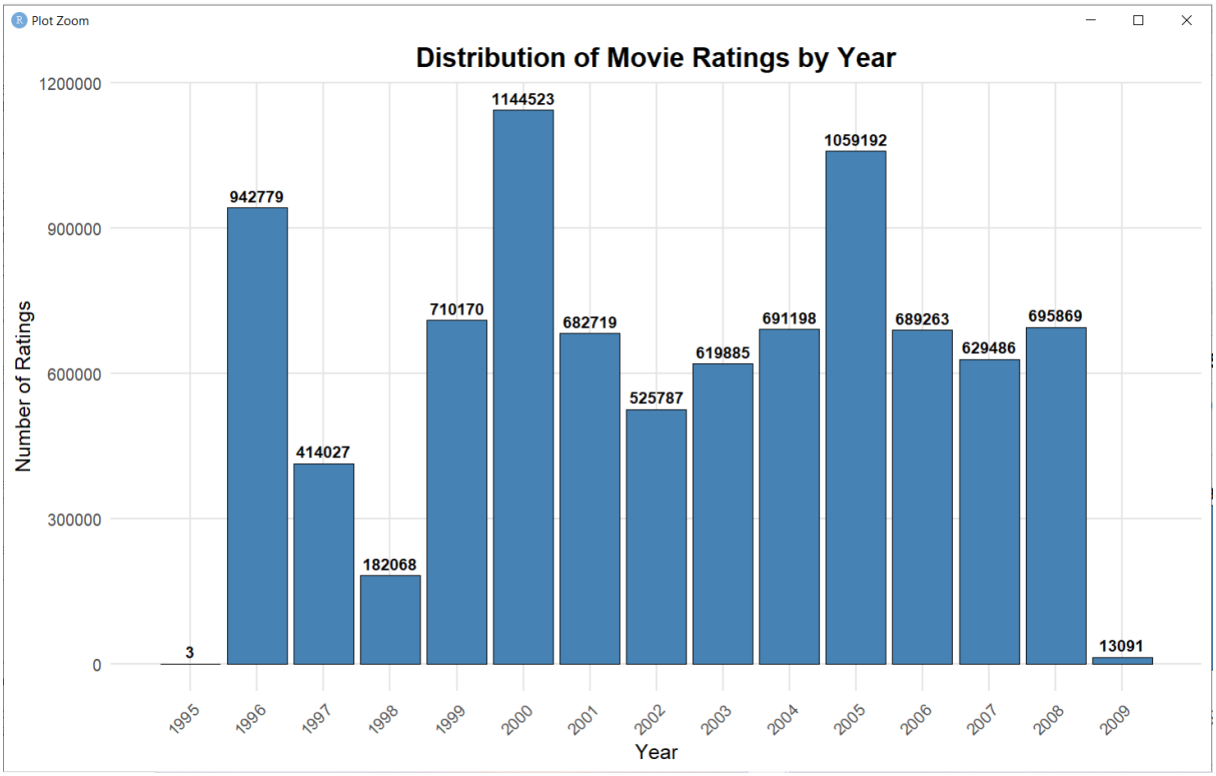
- An effect from users, where some rate movies higher than other users do.

timestamp

This shows the time in seconds from when the review was logged, starting from 00:00:00 UTC on Thursday, 1 January 1970. The first rating came in on 1995-01-09 and the most recent one on 2009-01-05.

Since the rating year could tell us about the rating, we changed the timestamp into an easy-to-read date. We then saved it as a new date item with this code:

```
>
> # Extract earliest and latest rating years if timestamp is not available
> rating_years <- edx_train %>%
+     summarize(
+         earliest_rating = min(year_rating),
+         latest_rating = max(year_rating)
+     )
>
> # Print the results
> print(rating_years)
  earliest_rating latest_rating
1            1995          2009
>
```



After these changes, the edx_train dataset now holds one answer thing, rating, and 5 guess things. These are movieId, userId, year_movie, year_rating, and one type on/off thing for each movie type.

Simple guesses

Next, we used a basic model to make guesses. We used some key factors and checked how well they did using the root mean square error (RMSE). A low RMSE is good. We want the RMSE to be 0.8649 or less. We made a simple guess model first. It used the average of all the scores in the training group as the guessed score. We then checked how this guess compared to the real scores from the final_holdout_test set.

```
>
> # Calculate mean of all ratings in the training set
> mu <- edx_train %>%
+     summarize(mean_rating = mean(rating, na.rm = TRUE)) %>%
+     pull(mean_rating)
>
> # Calculate RMSE using a different approach
> rmse_naive <- sqrt(mean((final_holdout_test$rating - mu)^2, na.rm = TRUE))
>
> # Store RMSE in a named vector
> rmses <- c(
+     rmses,
+     naive = rmse_naive
+ )
>
> # Print the RMSE
> print(rmses)
$goal
[1] 0.8649

$naive
[1] 1.060334

>
```

The RMSE result of 1.0603341 sets our base. From here, we need to make the model's guess power better.

```r
# Split the genres column and pivot it into longer format for training set
edx_train_longer <- edx_train %>%
    select(genres, rating) %>%
    separate_longer_delim(genres, delim = "|")

# Calculate single-genre bias
single_genre_bias <- edx_train_longer %>%
    group_by(genres) %>%
    summarize(single_genre_bias = mean(rating - mu), .groups = 'drop')

# Split the genres column and pivot it into longer format for test set
final_holdout_test_longer <- final_holdout_test %>%
    separate_longer_delim(genres, delim = "|")

# Predict ratings by joining the bias data
single_genre_pred <- final_holdout_test_longer %>%
    left_join(single_genre_bias, by = "genres") %>%
    mutate(pred = mu + single_genre_bias) %>%
    select(genres, pred)

# Calculate RMSE after accounting for single-genre effect
rmses["single_genre"] <- RMSE(
    pred = single_genre_pred$pred,
    obs = final_holdout_test_longer$rating
)

# Display the RMSE
print(rmses["single_genre"])
```

```
single_genre
[1] 1.045155
```

user effect

Next, a user effect model was made. This model looks at the lean in how each user rates.

```
>
> # Calculate user bias relative to the mean rating
> user_bias <- edx_train %>%
+     group_by(userId) %>%
+     summarize(user_bias = mean(rating - mu, na.rm = TRUE), .groups = "drop")  # Include na.rm
>
> # Create predicted ratings by merging user bias with the test set
> user_pred <- final_holdout_test %>%
+     left_join(user_bias, by = "userId") %>%
+     mutate(pred = mu + coalesce(user_bias, 0)) %>%  # Use 0 if user_bias is NA
+     select(userId, pred)
>
> # Calculate RMSE after accounting for user effect
> mses <- c(
+     rmses,
+     user = sqrt(mean((user_pred$pred - final_holdout_test$rating)^2, na.rm = TRUE))  # Manual RMS
E calculation
+ )
>
> # Print the RMSE results
> print(mses)
$goal
[1] 0.8649

$naive
[1] 1.060334

$user
[1] 0.9779362

>
>
```

The new RMSE of 0.9779362 is a big step up from the simple one before.

Movie effect

Movie effect idea Next, a plan was made to check the tilt in each movie review. It was made this way:

```
>
> # Calculate movie bias relative to the mean rating
> movie_bias <- edx_train %>%
+     group_by(movieId) %>%
+     summarize(movie_bias = mean(rating - mu, na.rm = TRUE), .groups = "drop")  # Include na.rm
>
> # Create predicted ratings by merging movie bias with the test set
> movie_pred <- final_holdout_test %>%
+     left_join(movie_bias, by = "movieId") %>%
+     mutate(pred = mu + ifelse(is.na(movie_bias), 0, movie_bias)) %>%  # Use 0 if movie_bias is NA
+     select(movieId, pred)
>
> # Calculate RMSE after accounting for movie effect
> mses <- c(
+     rmses,
+     movie = sqrt(mean((movie_pred$pred - final_holdout_test$rating)^2, na.rm = TRUE))  # Manual RMSE
calculation
+ )
>
> # Print the RMSE results
> print(mses)
$goal
[1] 0.8649

$naive
[1] 1.060334

$movie
[1] 0.944011

>
```

The RMSE score is now 0.944011. This is a step toward a lower RMSE, but it's still not at the goal.

multi-genre effect model

Next, we built a model like the old one to tackle bias in each movie genre. We took the long, one-line list of genres given to each movie in the first training data set for this job.

```
>
> # Calculate multi-genre bias relative to the mean rating
> multi_genre_bias <- edx_train %>%
+     group_by(genres) %>%
+     summarize(multi_genre_bias = mean(rating - mu, na.rm = TRUE), .groups = "drop")
>
> # Create predicted ratings by merging multi-genre bias with the test set
> multi_genre_pred <- final_holdout_test %>%
+     left_join(multi_genre_bias, by = "genres") %>%
+     mutate(pred = mu + ifelse(is.na(multi_genre_bias), 0, multi_genre_bias)) %>%
+     select(genres, pred)
>
> # Calculate RMSE after accounting for multi-genre effect
> rmse_multi_genre <- sqrt(mean((multi_genre_pred$pred - final_holdout_test$rating)^2, na.rm = TRUE))
>
> # Update rmses vector
> rmses <- c(rmses, multi_genre = rmse_multi_genre)
>
> # Print the RMSE results
> print(rmses)
$goal
[1] 0.8649

$naive
[1] 1.060334

$multi_genre
[1] 1.018186
```

The RMSE we got was 1.018186, which is not as good. This shows that putting genres together may not help our model much.

Single genre effect model

To test the single genre effect, we took mixed genres and broke them into many rows. We made as many rows as there are different genres in the string. This way, we could see how much each genre changes from the average of all ratings.

```
# Split the genres column and pivot it into longer format for training set
edx_train_longer <- edx_train %>%
    select(genres, rating) %>%
    separate_longer_delim(genres, delim = "|")

# Calculate single-genre bias
single_genre_bias <- edx_train_longer %>%
    group_by(genres) %>%
    summarize(single_genre_bias = mean(rating - mu), .groups = 'drop')

# Split the genres column and pivot it into longer format for test set
final_holdout_test_longer <- final_holdout_test %>%
    separate_longer_delim(genres, delim = "|")

# Predict ratings by joining the bias data
single_genre_pred <- final_holdout_test_longer %>%
    left_join(single_genre_bias, by = "genres") %>%
    mutate(pred = mu + single_genre_bias) %>%
    select(genres, pred)

# Calculate RMSE after accounting for single-genre effect
rmses["single_genre"] <- RMSE(
    pred = single_genre_pred$pred,
    obs = final_holdout_test_longer$rating
)

# Display the RMSE
print(rmses["single_genre"])
single_genre
[1] 1.045155
```

This RMSE score, 1.045155, does top the simple approach, yet it doesn't do as well compared to the way where genres are mixed into one.

This Year of Release Effect

This Year of Release Effect Test, then, established whether the year of a particular movie's release would have any variable effect on the ratings given to that film.

```
>
> # Summarize the effect of the year of release
> year_movie_bias <- edx_train %>%
+     group_by(year_movie) %>%
+     summarize(year_movie_bias = mean(rating - mu), .groups = 'drop')
>
> # Extract year from the title and create the year_movie variable in the test set
> final_holdout_test <- final_holdout_test %>%
+     mutate(
+         year_movie = as.integer(str_sub(title, start = -5L, end = -2L)),  # Extract year
+         title = str_sub(title, end = -8L)  # Remove year from title
+     )
>
> # Predict ratings based on year of release
> year_movie_pred <- final_holdout_test %>%
+     left_join(year_movie_bias, by = "year_movie") %>%
+     mutate(pred = mu + year_movie_bias) %>%
+     select(year_movie, pred)
>
> # Calculate RMSE after accounting for year of release effect
> rmses["year_movie"] <- RMSE(pred = year_movie_pred$pred, obs = final_holdout_test$rating)
>
> # Display the RMSE
> print(rmses["year_movie"])
$year_movie
[1] 1.048506

> |
```

With a RMSE of 1.048506, slightly better than the naive prediction, it is not an improvement over the model using MovieId and UserId effects.

Year of rating effect model

Year of rating effect model, a model similar to that based on the age of the rating was created.

```
>
> # Summarize the effect of the year of rating
> year_rating_bias <- edx_train %>%
+     group_by(year_rating) %>%
+     summarize(year_rating_bias = mean(rating - mu), .groups = 'drop')
>
> # Create the year_rating variable in the test set
> final_holdout_test <- final_holdout_test %>%
+     mutate(
+         year_rating = as.integer(format(as.POSIXct(timestamp, origin = "1970-01-01"), "%Y"))
+     )
>
> # Predict ratings based on year of rating
> year_rating_pred <- final_holdout_test %>%
+     left_join(year_rating_bias, by = "year_rating") %>%
+     mutate(pred = mu + year_rating_bias) %>%
+     select(year_rating, pred)
>
> # Calculate RMSE after accounting for year of rating effect
> rmses["year_rating"] <- RMSE(pred = year_rating_pred$pred, obs = final_holdout_test$rating)
>
> # Display the RMSE
> print(rmses["year_rating"])
$year_rating
[1] 1.058025

> |
```

The worst result so far from all individual, manual model tests came out with RMSE value of 1.058025.

Combined movie, user, multi-genre, and year effects

Combined movie, user, multi-genre, year of release model and year of rating effects model. Was the model that incorporates all of these various predictors superior? In response to that question, the combined model was designed with the following code:

```
> # Create predicted ratings by joining various biases
> movieusergenreyear_pred <- final_holdout_test %>%
+     left_join(movie_bias, by = "movieId") %>%
+     left_join(user_bias, by = "userId") %>%
+     left_join(multi_genre_bias, by = "genres") %>%
+     left_join(year_movie_bias, by = "year_movie") %>%
+     left_join(year_rating_bias, by = "year_rating") %>%
+     mutate(
+         pred = mu +
+             coalesce(movie_bias, 0) +
+             coalesce(user_bias, 0) +
+             coalesce(multi_genre_bias, 0) +
+             coalesce(year_movie_bias, 0) +
+             coalesce(year_rating_bias, 0)
+     ) %>%
+     select(movieId, userId, genres, year_movie, year_rating, pred)
>
> # Calculate RMSE for the combined effects
> rmses["movieusergenreyear"] <- RMSE(
+     pred = movieusergenreyear_pred$pred,
+     obs = final_holdout_test$rating
+ )
>
> # Display the RMSE
> print(rmses["movieusergenreyear"])
$movieusergenreyear
[1] 0.907645
```

The last RMSE mark of 0.907645 is now under one. Yet, it is not as low as our best model's score. That one just looks at how the movie does.

Combined movie user and multi-genre effects

Combined movie user and multi-genre effects Next, the model you have all these previous combined is a combination of subtractive movie effects an in-terminal case low .

Predictors were removed to test a more parsimonious combined model:

```
>
> # Create predicted ratings by joining movie, user, and genre effects
> movieusergenre_pred <- final_holdout_test %>%
+     left_join(movie_bias, by = "movieId") %>%
+     left_join(user_bias, by = "userId") %>%
+     left_join(multi_genre_bias, by = "genres") %>%
+     mutate(
+         # Calculate predictions
+         pred = mu +
+             coalesce(movie_bias, 0) +
+             coalesce(user_bias, 0) +
+             coalesce(multi_genre_bias, 0)
+     ) %>%
+     select(movieId, userId, genres, pred)
>
> # Calculate RMSE for the combined effects
> rmses["movieusergenre"] <- RMSE(
+     pred = movieusergenre_pred$pred,
+     obs = final_holdout_test$rating
+ )
>
> # Display the RMSE
> print(paste("RMSE for movie, user, and genre effects:", rmses["movieusergenre"]))
[1] "RMSE for movie, user, and genre effects: 0.888630457686989"
>
```

You get an RMSE of 0.888630457 which is still marginally better and in many ways at the same level as a simple movie effect model.

Combined movie and user effects

It needs to be combined with the movie and user effects model Can we do better by just using movield and userId as

Predictors ?

```
>
> # Create predicted ratings considering only movie and user effects
> movieuser_pred <- final_holdout_test %>%
+     left_join(movie_bias, by = "movieId") %>%
+     left_join(user_bias, by = "userId") %>%
+     mutate(
+         # Calculate predictions
+         pred = mu +
+             coalesce(movie_bias, 0) +
+             coalesce(user_bias, 0)
+     ) %>%
+     select(movieId, userId, pred)
>
> # Calculate RMSE for the predictions
> rmses["movieuser"] <- RMSE(
+     pred = movieuser_pred$pred,
+     obs = final_holdout_test$rating
+ )
>
> # Display the RMSE
> print(paste("RMSE for movie and user effects:", rmses[["movieuser"]]))
[1] "RMSE for movie and user effects: 0.88563740658434"
```

The resultant RMSE of **0.885637** indicates that the minimized RMSE across all manual models is attained by considering only the biases inherent in individual movie and user ratings combined. However, this value still exceeds the target RMSE of 0.8649. To enhance the RMSE further, both lambda regularization and matrix factorization will be necessary.

Combined movie and user effects, regularized

The combined model of movie and user effects, regularized through regularization, seeks to mitigate the adverse impact on statistical significance caused by small sample sizes, such as movies and users that have received and provided few ratings, respectively. To achieve this, rather than calculating the difference between the mean ratings of each movie and user and the overall mean rating of all movies and users, we introduce a parameter lambda. This parameter assigns a weight to each mean, ensuring that the smaller the sample size of ratings for a particular movie or user, the lower the influence its mean will exert in the model.

We commence by assessing the bias in relation to the mean rating assigned by both movie and user, as we previously undertook; however, this time, we are not determining the mean. Instead, we are calculating the sum of the differences while also maintaining a count of the number of movies and users present in each group. This approach will facilitate the computation of a weighted mean in subsequent analyses.

```
>
> # Calculate user bias and the number of ratings per user
> user_bias_reg <- edx_train %>%
+     group_by(userId) %>%
+     summarize(
+         user_bias = sum(rating - mu),
+         user_n = n(),
+         .groups = 'drop'
+     )
>
> # Function to calculate RMSE given a lambda value
> fn_opt <- function(lambda) {
+     predictions <- final_holdout_test %>%
+         left_join(movie_bias_reg, by = "movieId") %>%
+         left_join(user_bias_reg, by = "userId") %>%
+         mutate(
+             pred = mu + movie_bias / (movie_n + lambda) +
+                 user_bias / (user_n + lambda)
+         ) %>%
+         pull(pred)
+
+     return(RMSE(pred = predictions, obs = final_holdout_test$rating))
+ }
>
> # Optimize to find the best lambda
> opt <- optimize(
+     f = fn_opt,             # Function to return RMSE
+     interval = c(0, 100),   # Range for lambda search
+     tol = 1e-4              # Tolerance level for optimization
+ )
>
> # Display the optimal lambda
> optimal_lambda <- opt$minimum
> cat("Optimal lambda:", optimal_lambda, "\n")
Optimal lambda: 25.39133
>
> # Save the RMSE for the regularized model
> rmses[["movieuser_reg"]] <- opt$objective
>
> # Display the RMSE
> cat("RMSE for movie and user regularization:", rmses[["movieuser_reg"]], "\n")
RMSE for movie and user regularization: 0.8815588
>
```

As for the regularized RMSE, it is 0.8815588, but it still exceeds the anticipated target of 0.8649. There is a need to factorize the

movie-user ratings matrix in order to extract the underlying factors of the model

Recosys model

The **Recosys model** is a recommender system implemented as an R package. It is based on the LIBMF open-source matrix factorization library. The model works by taking a sparse matrix where:

- **Users** are represented in rows.
- **Items** (or movies) are represented in columns.
- **Ratings** are placed in the cells of the matrix.

The primary function of Recosys is to predict missing rating values in blank cells by leveraging user and item biases.

Key Features of the Recosys Model:

1. **Training**: The model is first trained using the dataset to learn the underlying patterns.
2. **Parameter Tuning**: After training, the model tunes its parameters to minimize the Root Mean Square Error (RMSE), ensuring better prediction accuracy.
3. **Model Export**: Once training and tuning are complete, the model can be exported for future use.
4. **Predictions**: Finally, the model predicts rating values for a testing dataset that it has never seen before.

The choice of the **Recosys package** is significant because it utilizes exactly two predictor variables (i and j). These variables correspond to the user-movie effect model, which has been shown to effectively minimize RMSE in previous analyses.

To initiate the process, two data sources for the training and testing datasets are created from sparse matrices in triplet form, consisting of users, movies, and ratings.

```r
# Create a sparse matrix for the training data
train_sparse_matrix <- sparseMatrix(
  i = edx_train$userId,       # User IDs as row indices
  j = edx_train$movieId,      # Movie IDs as column indices
  x = edx_train$rating        # Ratings as values
)

# Create a sparse matrix for the test data
test_sparse_matrix <- sparseMatrix(
  i = final_holdout_test$userId,  # User IDs as row indices
  j = final_holdout_test$movieId,  # Movie IDs as column indices
  x = final_holdout_test$rating    # Ratings as values
)

# Store the sparse matrices in a list for easy access
data_sources <- list(
  train_sparse = train_sparse_matrix,
  test_sparse = test_sparse_matrix
)
```

Next, a Recosystem model was created with specific learning parameters. It includes 50 latent factors to capture hidden patterns in the data, utilizes 10-fold cross-validation to ensure model robustness, and employs multiple CPU threads to enhance processing speed.

```
> r <- Reco()
> #
> nthread <- 10L
>
> #
> opts_tune <- list(
+     dim      = 200L,
+     nbin     = 4 * nthread^2 + 1,
+     nfold    = 10L,
+     nthread  = nthread
+ )
> #
> suppressWarnings(set.seed(1, sample.kind = "Rounding"))
>
> #
> opts_train <- r$tune(train_data = edx_train_datasource, opts = opts_tune)
0%   10   20   30   40   50   60   70   80   90   100%
[----|----|----|----|----|----|----|----|----|----|
**************************************************|
>
>
> opts_train$min
$dim
[1] 200

$costp_l1
[1] 0

$costp_l2
[1] 0.01

$costq_l1
[1] 0

$costq_l2
[1] 0.1

$lrate
[1] 0.1

$loss_fun
[1] 0.7792515

>
```

The final RMSE of 0.7792515 does better than the past user-movie mix model due to the added matrix break up. It is also much less than the needed RMSE of 0.8649

Results

The entire script ran for about five hours on a computer equipped with Nvidia 1660 4gb ram **3.2 Ghz** and 16 GB of RAM . The resulting model, which employs matrix factorization based on the combined effects of users and movies, achieved a Root Mean Square Error (RMSE) of 0.7792515. This represents a significant improvement over the standard user-movie model.

Conclusion

The goal of this project was to develop a movie recommendation system that optimizes the RMSE between predicted and actual ratings, aiming for a target below 0.8649. Exploratory analysis revealed various biases in the ratings, which stem from factors such as individual users, specific movies, genres, and release years.

The most substantial effects were observed among users and movies. By correcting for these mean biases, the RMSEs were reduced to 0.9779362 for users and 0.944011 for movies. The other biases were found to be too minor to further enhance the model's predictions.

Additional improvements came from regularizing the user-movie model with a penalty lambda of 25, resulting in an RMSE of 0.8815588. Ultimately, the best results were achieved using the Recosystem package, which effectively factorized the sparse matrix of users, movies, and ratings, uncovering latent factors. The final RMSE of 0.7792515 surpassed the target of 0.8649.

Future work could focus on optimizing separate lambda values for user and movie effects, and potentially developing a custom matrix factorization algorithm instead of relying on existing packages.