A Memory-Based Recommendation System: My Netflix Prize Derivative Model

David Augustine

Contents

Foreword **2**

1 Introduction **3**

2 Methods and Analysis **5**
2.1 Initialization . . . 5
2.2 Data Retrieval . . . 6
2.3 Data Manipulation . . . 8
2.4 Data Analysis . . . 12
2.4.1 Ratings . . . 23
2.4.2 Possible Predictors . . . 24
2.4.2.1 Calculated Importances . . . 24
2.4.2.2 Distributions and Characteristics . . . 25
2.4.2.2.1 Users . . . 25
2.4.2.2.2 Movies . . . 28
2.4.2.2.3 Genres . . . 32
2.4.2.2.4 Rating Lapses . . . 34
2.4.2.2.5 Movie Ages . . . 35
2.4.2.2.6 Rating Ages . . . 37
2.4.3 Applying our Findings . . . 39
2.4.3.1 Included Predictors and Their Order . . . 39
2.4.3.1.1 Users vs Movies - Which is the Real Top Dog? . . . 39
2.4.3.1.2 Genres - Another Favored Variable . . . 41
2.4.3.1.3 Rating Age - Some Variables Just Don't Cut It . . . 41
2.4.3.1.4 Rating Lapse vs Movie Age - OVB? Who Cares! . . . 42
2.4.3.2 Model Specification . . . 43
2.5 Model Construction . . . 44

3 Results **51**

4 Conclusion **52**
4.1 Summary . . . 52
4.2 Limitations . . . 52
4.3 Future Work . . . 52

Foreword

This report has been created as part of an R project requirement for the completion of HarvardX's Data Science Professional Certificate program. HarvardX is a University-wide strategic initiative at Harvard University, designed to facilitate online education by faculty of Harvard for students everywhere.

Chapter 1

Introduction

Recommendation systems constitute a subclass of *information filtering systems* that aims to predict user ratings for items. Some of the most widely recognized applications of such systems are responsible for the suggested playlists, videos and movies presented to users on services such as Spotify, YouTube and Amazon Prime Video.

In this report, we present and analyze a modified subset of the *MovieLens* dataset and then create and test a movie recommendation system based on the information contained within. This dataset, compiled by *GroupLens*, a research lab at the University of Minnesota, consists of roughly 100,000,000 records and was used by various teams for a competition, known as the *Netflix Prize*, that was sponsored by Netflix in 2006. The purpose of the challenge was to substantially improve upon Netflix's then-current recommendation system, called *Cinematch*, with the first team to come up with a recommendation system achieving a *root mean squared error (RMSE)* at least 10% lower to be awarded the grand prize of $1,000,000.

In 2009, the competition came to a close, with a team called *BellKor's Pragmatic Chaos* being announced as the winners of the competition and the grand prize after successfully building a recommendation system resulting in an RMSE of 0.8567, roughly 10.06% lower than Netflix's RMSE of 0.9525.

The dataset we'll be working with consists of roughly 10,000,000 records, with each record representing the rating of a particular movie given by a particular user. In addition to the rating itself, which can take on any non-zero half-integer value up to 5, the fields in the dataset provide us with the corresponding user ID, movie ID, movie title, movie release year, movie genre(s) and rating timestamp.

All work to follow will be done entirely within R and will be delineated as a 5-step process: we initialize our R work environment, retrieve our data, manipulate our data, analyze our data, and lastly construct and test our model.

It's worth noting that the initial main chunks of code corresponding to each of the 5 steps of this process collectively comprise all code used for the analysis I performed for this project. If desired, the reader can run these 5 R scripts in sequence to replicate the work environment of the analysis, although these scripts can also be skipped over without any loss of continuity to

the presentation in this report.

Chapter 2

Methods and Analysis

2.1 Initialization

Before working with any data, we must first initialize our work environment within R by loading several different packages and defining multiple objects to be used throughout the subsequent steps. We do this by executing the following code (note that not all packages loaded are used for this project; those listed are simply the ones I generally load before doing any work in R).

```
# Step0_Initialize #

# Define names of packages to load for scripts. Not all are actually
# necessary.

Packages <- c("bigmemory", "bigstatsr", "broom", "caret", "compiler",
  "caTools", "data.table", "doParallel", "doSNOW", "dplyr", "dslabs",
  "e1071", "fastAdaboost", "foreach", "formatR", "future", "gam",
  "genefilter", "ggplot2", "ggrepel", "gridExtra", "HistData", "kernlab",
  "knitr", "Lahman", "lpSolve", "lubridate", "MASS", "matrixStats",
  "mvtnorm", "naivebayes", "parallel", "pdftools", "promises", "purrr",
  "randomForest", "ranger", "Rborist", "RColorBrewer", "recommenderlab",
  "recosystem", "reshape2", "rlist", "ROSE", "rpart", "rpart.plot", "rtweet",
  "rvest", "scales", "snow", "stringr", "svMisc", "svSocket", "textdata",
  "tibble", "tidyr", "tidytext", "tidyverse", "tree", "zoo")

# Download and install any packages not already installed and then load
# them.

for(p in Packages){
  if(!require(p, character.only = TRUE)){install.packages(p,
    character.only = TRUE, repos = "http://cran.us.r-project.org")}
```

```
  library(p, character.only = TRUE)
}

# Define a function that allows easy viewing of multiple figures
# simultaneously.

Show <- function(v){
  dev.new(noRStudioGD = TRUE)
  print(v)
}

# Define a function that prints a progress bar for time-consuming
# calculations.

ProgressBar <- function(i){
  cat("     Progress: [", strrep(c("|", " "), c(i, 100 - i)), "] ",
    ifelse(i != 100, paste(i, "%"), "Done!"), "\r",
    if(i == 100){c("\n", "\n")}, sep  = "")
  flush.console()
}

# Define a function that assigns a global scope to its input argument (to
# be used when utilizing parallel processing to speed up certain
# calculations).

make_global <- function(task_arg){arg <<- task_arg}

# Assign the number of logical processors to be used for parallel
# processing. For my computer, this is 20. The parallel processing code in
# this project is therefore based on this value.

logical_CPUs <- detectCores(logical = TRUE)

rm(Packages, p)
```

2.2 Data Retrieval

Once we've initialized our R environment, we download the data we'll be working with via a *GroupLens* link (shown in the next script).

Then, after manipulating the downloaded data, we're left with a partition of the data into two datasets: `edx` and `validation`. The `edx` dataset will consist of approximately 90% of the records in the downloaded data and serve as our *training dataset*, and the `validation` dataset will consist of the remaining approximately 10% of records and serve as our *testing*

dataset.

Note that the code for this step is provided to the student and can't be altered in any meaningful way.

```
# Step1_RetrieveData #

# This is the script provided in the course to generate the edx and
# validation datasets, with an additional line of code added at the very
# end.

dl <- tempfile()

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

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

movies <- str_split_fixed(readLines(
  unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)

colnames(movies) <- c("movieId", "title", "genres")

movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
  title = as.character(title), genres = as.character(genres))

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

set.seed(1, sample.kind = "Rounding")

test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1,
  list = FALSE)

edx <- movielens[-test_index, ]

temp <- movielens[test_index, ]

validation <- temp %>% semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)

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

```r
# Delete the "ml-10M100k" directory.

unlink("ml-10M100K", recursive = TRUE)
```

2.3 Data Manipulation

Given our `edx` and `validation` datasets, the next step is to manipulate them (without adding, removing or otherwise fundamentally changing any records) so that they're in a format more conducive to our analysis.

The first few lines of code in the next script add the temporary `rating_year` field and subsequently remove the `timestamp` field in both datasets, in addition to redefining the `title` field so that it excludes the movie's release year, which is instead assigned to the added temporary `release_year` field.

Next, we extract and list all of the unique genres from the `genres` field in the `edx` dataset, since the `genres` field often lists multiple different genres for each movie in the dataset. Note that we exclude from this list the empty string and "(no genres listed)", since both values are uninformative and thus trivial for our purposes of analysis and modeling.

Once we have our list of unique genres, we define and then apply `DefineGenreBernoullis` to both the `edx` dataset and the `validation` dataset. Doing this creates 19 additional fields for each dataset; 1 field for each of the 19 unique non-trivial genres in `edx.UniqueGenres`. Each of these fields takes on a value of either 1 or 0, depending on whether or not the value of `genres` for a given record includes the unique genre represented by that field.

Because applying this function is time-consuming, we include the `ProgressBar` function within its code so that R prints out a progress bar in real time during its execution. We make use of the `ProgressBar` function for a few other functions to be defined later as well.

Note that we also make use of *parallel processing* when applying `DefineGenreBernoullis` to the `edx` dataset. This allows us to simultaneously apply the function to different chunks of the dataset by utilizing multiple logical processors, significantly speeding up the function's runtime and improving the allocation of RAM during its execution. For the `validation` dataset, on the other hand, we don't bother, since the dataset is much smaller and the use of *parallel processing* is unnecessary.

We then define the `movie_age`, `rating_age` and `rating_lapse` fields, which represent the age of each movie as of 2009, the age of each rating as of 2009, and the number of years elapsed between the release date of each movie and the date of its rating. We also remove the temporary `rating_year` and `release_year` fields at this point, as they are now redundant.

We lastly append the 19 additional fields to each dataset and reorder their fields for easier analysis, saving the manipulated datasets to a subfolder (which we create if it doesn't already exist) within the project's home directory.

```r
# Step2_ManipulateData #

# Define the rating_year and release_year variables in both the edx and
# validation datasets.

edx <- edx %>%
  mutate(rating_year = year(as_date(as_datetime(timestamp)))) %>%
  extract(title, c("title", "release_year"), "^(.*)\\s\\((\\d{4})\\)$") %>%
  dplyr::select(-timestamp)

validation <- validation %>%
  mutate(rating_year = year(as_date(as_datetime(timestamp)))) %>%
  extract(title, c("title", "release_year"), "^(.*)\\s\\((\\d{4})\\)$") %>%
  dplyr::select(-timestamp)

# Define a vector containing the unique genre combination values from the
# genres field.

edx.UniqueGenreCombos <- unique(edx$genres)

# Define a vector containing the unique individual genres extracted from the
# various combinations of the genres field. Note that those values of the
# genres field that are empty strings or "(no genres listed)" have been
# ignored.

edx.UniqueGenres <- str_split(edx.UniqueGenreCombos, "\\|{1}",
  simplify = TRUE) %>% as.vector() %>%
  unique() %>% as_tibble() %>%
  filter(value != "" & value != "(no genres listed)") %>% .$value %>% sort()

# Define a function to assign columns of 1s and 0s to each row of its input
# dataset, with each column corresponding to one of the unique genres
# listed in edx.UniqueGenres. A 1 in a given column indicates that row's
# value for the genres field includes the unique genre corresponding to
# that column.

DefineGenreBernoullis <- function(){
  output <- matrix(nrow = length(arg), ncol = length(edx.UniqueGenres))
  r <- 1
  for(x in arg){
    output[r, ] <- ifelse(str_detect(x, edx.UniqueGenres), 1, 0)
    if(r < 0.995*length(arg) + 1){ProgressBar(round(100*(r/length(arg)),
      0))}
    r <- r + 1
```

```
  }
  return(output)
}

# Define a list with 19 components. Each component will be processed
# simultaneously and independently by its own logical processor.

edx.UniqueGenres.Chunks <- vector("list", length = 19)

# Assign chunks of rows of the genres field to the components of the above
# defined list.

for(j in 1:19){
  edx.UniqueGenres.Chunks[[j]] <- edx$genres %>%
    .[((j - 1)*500000 + 1):pmin(j*500000, nrow(edx))]
}

# Define the cluster object to be referenced to utilize parallel processing
# when applying the DefineGenreBernoullis function.

DefineGenreBernoullis.cluster <- makeCluster(logical_CPUs - 1)

registerDoParallel(cl = DefineGenreBernoullis.cluster)

# Make each of the chunks defined above global in scope across all R
# sessions. This therefore makes these chunks global in scope within each of
# the worker R sessions created for parallel processing.

invisible(clusterApply(cl = DefineGenreBernoullis.cluster,
  edx.UniqueGenres.Chunks, make_global))

# Export the "ProgressBar", "str_detect" and "edx.UniqueGenres" objects to
# each of the worker R sessions so the DefineGenreBernoullis function can be
# properly called within each worker R session.

clusterExport(cl = DefineGenreBernoullis.cluster,
  c("ProgressBar", "str_detect", "edx.UniqueGenres"))

# Define a matrix containing the output of the DefineGenreBernoullis
# function.

edx.GenreBernoullis <- clusterCall(cl = DefineGenreBernoullis.cluster,
  DefineGenreBernoullis) %>% as.matrix() %>% do.call("rbind", .)
```

```
# Close the cluster defined to utilize parallel processing when applying the
# DefineGenreBernoullis function.

stopCluster(DefineGenreBernoullis.cluster)

# Assign column names to the edx.GenreBernoullis matrix.

colnames(edx.GenreBernoullis) <- paste(edx.UniqueGenres, "Ind", sep = ".")

# Define the movie_age, rating_age and rating_lapse variables in the edx
# dataset. These variables represent, respectively, the age of each movie as
# of 2009, the age of each rating as of 2009, and the number of years passed
# between the release year of each movie and the years in which each of its
# ratings were submitted.

edx <- edx %>% mutate(release_year = as.numeric(release_year),
  movie_age = pmax(0, 2009 - release_year),
  rating_age = pmax(0, 2009 - rating_year),
  rating_lapse = pmax(0, rating_year - release_year)) %>%
  dplyr::select(!c(release_year, rating_year))

# Append the genre Bernoulli variables to the edx dataset and reorder its
# columns.

edx <- cbind(edx, edx.GenreBernoullis) %>% .[, c(1, 3, 7:8, 2, 4, 6:5, 9:27)]

# The below lines of code repeat the above process of adding and reordering
# the columns, but now for the validation dataset. Note that because this
# dataset is smaller, no parallel processing is used when applying the
# DefineGenreBernoullis function.

validation.GenreBernoullis <- matrix(nrow = nrow(validation),
  ncol = length(edx.UniqueGenres))

arg <- validation$genres

validation.GenreBernoullis <- DefineGenreBernoullis()

colnames(validation.GenreBernoullis) <- paste(edx.UniqueGenres,
  "Ind", sep = ".")

validation <- validation %>% mutate(release_year = as.numeric(release_year),
  movie_age = pmax(0, 2009 - release_year),
  rating_age = pmax(0, 2009 - rating_year),
```

```
  rating_lapse = pmax(0, rating_year - release_year)) %>%
  dplyr::select(!c(release_year, rating_year))

validation <- cbind(validation, validation.GenreBernoullis) %>%
  .[, c(1, 3, 7:8, 2, 4, 6:5, 9:27)]

# Create the "Data" and "Data/R Data" folders if they don't already exist
# in the project's home directory.

if(!dir.exists("Data")){dir.create("Data")}

if(!dir.exists("Data/R Data")){dir.create("Data/R Data")}

# Save the edx and validation datasets within the project's home directory.

save(edx, file = "Data/R Data/edx.rda")

save(validation, file = "Data/R Data/validation.rda")

rm(edx.GenreBernoullis, edx.UniqueGenres.Chunks, validation.GenreBernoullis,
   edx.UniqueGenreCombos, edx.UniqueGenres, arg,
   DefineGenreBernoullis.cluster, j)
```

2.4 Data Analysis

At this point, we're ready to begin analyzing the data in the `edx` table. The below code generates the importances of the dataset's variables as well as several different figures that provide insight into the natures of their distributions. In particular, we are interested in the `movieId`, `userId`, `rating_lapse`, `movie_age`, `rating_age` and `genres` variables.

The information gleaned by reviewing these items will help us decide on an appropriate model specification for our recommendation system.

```
# Step3_AnalyzeData #

# If the edx and validation datasets aren't already loaded in the R session,
# load them.

if(!exists("edx")){load("Data/R Data/edx.rda")}

if(!exists("validation")){load("Data/R Data/validation.rda")}

# Create the "Figures" folder if it doesn't already exist in the project's
# home directory.
```

```r
if(!dir.exists("Figures")){dir.create("Figures")}

# Variable Importance metrics and figures:

# Define different sets of indices to be used to define different chunks of
# the edx dataset. Each chunk will correspond to a different worker R
# session in the parallel processing to follow.

set.seed(2020, sample.kind  = "Rounding")

edx.partition.indices <- createFolds(edx$rating, k = logical_CPUs)

# Define the different chunks of the edx dataset.

chunks <- lapply(edx.partition.indices,
  function(data, indices){data[indices, ]}, data = edx)

# Define the RunRandomForest function to be applied to each chunk of the
# edx dataset.

RunRandomForest <- function(tree_num){
  arg <- arg %>% mutate(movieId = as.factor(movieId),
    userId = as.factor(userId), genres = as.factor(genres))
  ranger(rating ~ movieId + userId + genres + rating_lapse + movie_age +
    rating_age, data = arg, num.trees = tree_num, importance = "impurity",
    respect.unordered.factors = TRUE)
}

# Define the cluster object to be referenced to utilize parallel processing
# when applying the RunRandomForest function.

RunRandomForest.cluster <- makeCluster(logical_CPUs)

registerDoParallel(cl = RunRandomForest.cluster)

# Make each of the chunks defined above global in scope across all R
# sessions. This therefore makes these chunks global in scope within each
# of the worker R sessions created for parallel processing.

invisible(clusterApply(cl = RunRandomForest.cluster, chunks, make_global))

# Export the "ranger", "%>%" and "mutate" objects to each of the worker R
# sessions so the RunRandomForest function can be properly called within
# each worker R session.
```

```r
clusterExport(cl = RunRandomForest.cluster, c("ranger", "%>%", "mutate"))

# Define a list containing the output of the RunRandomForest function.

set.seed(2020, sample.kind  = "Rounding")

edx.RandomForest <- clusterCall(cl = RunRandomForest.cluster,
  RunRandomForest, tree_num = 100)

# Close the cluster defined to utilize parallel processing when applying
# the RunRandomForest function.

stopCluster(RunRandomForest.cluster)

# Define the VariableImpsByChunk matrix containing the variable importances
# of the different variables for each chunk of the edx dataset.

for(j in 1:20){
  if(j == 1){
    VariableImpsByChunk <-  edx.RandomForest[[j]]$variable.importance
  }else{
    VariableImpsByChunk <- cbind(VariableImpsByChunk,
      edx.RandomForest[[j]]$variable.importance)
  }
}

# Assign column names to the VariableImpsByChunk matrix.

colnames(VariableImpsByChunk) <- paste("var.imp", seq(1:20), sep = "_")

# Define a sorted vector of average variable importances for the different
# variables.

VariableImps <- rowMeans(VariableImpsByChunk) %>% sort(decreasing = TRUE)

# Metrics and figures of distribution of ratings:

# Define a dataset consisting of the frequency of each rating.

Ratings.Distribution <- edx %>% group_by(rating) %>%
  summarize(Count = n(), .groups = "drop")

# Define, show and save a histogram showing the frequency of each rating.
```

```r
Ratings.Histogram <- Ratings.Distribution %>%
  mutate(rating = factor(rating)) %>% ggplot(aes(rating, Count)) +
  geom_col(fill = "cornflowerblue", color = "white") +
  labs(x = "Rating", y = "Count", title = "Distribution of Ratings") +
  geom_vline(aes(xintercept = 7.02), color = "red") +
  annotate("text", x = 6.75, y = 2.25*10^6,
  label = round(sum(Ratings.Distribution$rating*Ratings.Distribution$Count)/
  sum(Ratings.Distribution$Count), 2), color = "red", size = 3) +
  ggthemes::theme_economist() + theme(plot.title = element_text(hjust = 0.5))

Show(Ratings.Histogram)

ggsave(filename = "Ratings_Histogram.png", plot = Ratings.Histogram,
  path = "Figures")

# Metrics and figures by movie:

# Define a dataset consisting of the frequency, average rating and rating
# standard deviation for each movie.

Ratings_by_Movie.Distribution <- edx %>% group_by(movieId) %>%
  summarize(Count = n(), Mu_gvn_Movie = mean(rating),
  Sd_gvn_Movie = sd(rating), .groups = "drop")

# Define, show and save a histogram showing the distribution of rating
# counts by movie.

Ratings_by_Movie.Counts.Histogram <- Ratings_by_Movie.Distribution %>%
  ggplot(aes(Count)) + geom_histogram(fill = "cornflowerblue",
  color = "white") + labs(title = "Distribution of Movie Rating Counts",
  x = "Movie Rating Count", y = "Movie Count") +
  geom_vline(aes(xintercept = mean(Count)), color = "red") +
  annotate("text", x = 1200, y = 550,
  label = round(mean(Ratings_by_Movie.Distribution$Count), 0),
  color = "red", size = 3) + scale_y_continuous(limits = c(-1, 801)) +
  scale_x_log10() + ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5),
  legend.position = "none")

Show(Ratings_by_Movie.Counts.Histogram)

ggsave(filename = "Ratings_by_Movie_Counts_Histogram.png",
```

```
  plot = Ratings_by_Movie.Counts.Histogram, path = "Figures")

# Define, show and save a density plot showing the distributions of rating
# standard deviations by movie, with one density corresponding to those
# movies with at most the median number of ratings per movie, and the other
# corresponding to those movies with more than the median number of ratings
# per movie.

Ratings_by_Movie.StDevs.Density <- Ratings_by_Movie.Distribution %>%
  mutate(Partition = cut(Count, breaks = c(-Inf, median(Count), Inf),
  labels = c("Rating Count <= 122", "Rating Count > 122"))) %>%
  ggplot(aes(Sd_gvn_Movie, fill = Partition)) + geom_density(alpha = 0.5) +
  labs(title = "Density of Rating Standard Deviation - By Movie Group",
  x = "Rating Standard Deviation", y = "Density") +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_Movie.StDevs.Density)

ggsave(filename = "Ratings_by_Movie_StDevs_Density.png",
  plot = Ratings_by_Movie.StDevs.Density, path = "Figures")

# Define, show and save a scatterplot showing rating counts against rating
# averages by movie, including a smoothed trend line with a confidence
# interval.

Ratings_by_Movie.Counts_vs_Avgs.Scatterplot <-
  Ratings_by_Movie.Distribution %>% ggplot(aes(Count, Mu_gvn_Movie)) +
  geom_point(color = "cornflowerblue", alpha = 0.3) + geom_smooth() +
  labs(title = "Scatterplot of Movie Rating Count vs Average Rating",
  x = "Movie Rating Count", y = "Average Rating") +
  geom_vline(aes(xintercept = mean(Count)), color = "red") +
  annotate("text", x = 2000, y = 5,
  label = round(mean(Ratings_by_Movie.Distribution$Count), 0),
  color = "red", size = 3) +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_Movie.Counts_vs_Avgs.Scatterplot)
```

```r
ggsave(filename = "Ratings_by_Movie_Counts_vs_Avgs_Scatterplot.png",
  plot = Ratings_by_Movie.Counts_vs_Avgs.Scatterplot, path = "Figures")

# Metrics and figures by user:

# Define a dataset consisting of the frequency, average rating and rating
# standard deviation for each user.

Ratings_by_User.Distribution <- edx %>% group_by(userId) %>%
  summarize(Count = n(), Mu_gvn_User = mean(rating),
  Sd_gvn_User = sd(rating), .groups = "drop")

# Define, show and save a histogram showing the distribution of rating
# counts by user.

Ratings_by_User.Counts.Histogram <- Ratings_by_User.Distribution %>%
  ggplot(aes(Count)) + geom_histogram(fill = "cornflowerblue",
  color = "white") + labs(title = "Distribution of User Rating Counts",
  x = "User Rating Count", y = "User Count") +
  geom_vline(aes(xintercept = mean(Count)), color = "red") +
  annotate("text", x = 175, y = 6000,
  label = round(mean(Ratings_by_User.Distribution$Count), 0), color = "red",
  size = 3) + scale_y_continuous(limits = c(-1, 8001)) +
  scale_x_log10() + ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5),
  legend.position = "none")

Show(Ratings_by_User.Counts.Histogram)

ggsave(filename = "Ratings_by_User_Counts_Histogram.png",
  plot = Ratings_by_User.Counts.Histogram, path = "Figures")

# Define, show and save a density plot showing the distributions of rating
# standard deviations by user, with one density corresponding to those
# users with at most the median number of ratings per user, and the other
# corresponding to those users with more than the median number of ratings
# per user.

Ratings_by_User.StDevs.Density <- Ratings_by_User.Distribution %>%
  mutate(Partition = cut(Count, breaks = c(-Inf, median(Count), Inf),
  labels = c("Rating Count <= 62", "Rating Count > 62"))) %>%
  ggplot(aes(Sd_gvn_User, fill = Partition)) + geom_density(alpha = 0.5) +
```

```
  labs(title = "Density of Rating Standard Deviation - By User Group",
  x = "Rating Standard Deviation", y = "Density") +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_User.StDevs.Density)

ggsave(filename = "Ratings_by_User_StDevs_Density.png",
  plot = Ratings_by_User.StDevs.Density, path = "Figures")

# Define, show and save a scatterplot showing rating counts against rating
# averages by user, including a smoothed trend line with a confidence
# interval.

Ratings_by_User.Counts_vs_Avgs.Scatterplot <-
  Ratings_by_User.Distribution %>% ggplot(aes(Count, Mu_gvn_User)) +
  geom_point(color = "cornflowerblue", alpha = 0.3) + geom_smooth() +
  labs(title = "Scatterplot of User Rating Count vs Average Rating",
  x = "User Rating Count", y = "Average Rating") +
  geom_vline(aes(xintercept = mean(Count)), color = "red") +
  annotate("text", x = 400, y = 5,
  label = round(mean(Ratings_by_User.Distribution$Count), 0), color = "red",
  size = 3) + ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_User.Counts_vs_Avgs.Scatterplot)

ggsave(filename = "Ratings_by_User_Counts_vs_Avgs_Scatterplot.png",
  plot = Ratings_by_User.Counts_vs_Avgs.Scatterplot, path = "Figures")

# Metrics and figures by genre:

# Define a dataset consisting of the frequency, average rating and rating
# standard deviation for each unique genre.

Ratings_by_Genre.Distribution <- tibble()

for(j in 1:19){
  temp <- edx %>%
    group_by_(paste("`", colnames(edx)[j + 8], "`", sep = "")) %>%
```

```
    summarize(Count = n(), Mu_gvn_Genre = mean(rating),
    Sd_gvn_Genre = sd(rating), .groups = "drop") %>% slice_max(.[, 1]) %>%
    .[, 2:4] %>% mutate(Genre = colnames(edx)[j + 8]) %>% .[, c(4, 1:3)]
  if(j == 1){
    Ratings_by_Genre.Distribution <- as.matrix(temp)
  }else{
    Ratings_by_Genre.Distribution <-
      rbind(as.matrix(Ratings_by_Genre.Distribution), temp)
  }
}

Ratings_by_Genre.Distribution <- Ratings_by_Genre.Distribution %>%
  mutate(Count = as.numeric(Count), Mu_gvn_Genre = as.numeric(Mu_gvn_Genre),
  Sd_gvn_Genre = as.numeric(Sd_gvn_Genre))

# Define, show and save a scatterplot showing average rating up against
# rating standard deviation for each unique genre.

Ratings_by_Genre.Summary.Scatterplot <- Ratings_by_Genre.Distribution %>%
  ggplot(aes(x = Mu_gvn_Genre, y = Sd_gvn_Genre, label = Genre)) +
  geom_point(aes(x = Mu_gvn_Genre, y = Sd_gvn_Genre, size = Count),
  inherit.aes = FALSE) + geom_point(aes(color = Genre, size = Count),
  show.legend = FALSE) + geom_text_repel() +
  scale_size_continuous(name = "Count:", breaks = 10^6*c(1, 2, 3),
  labels = c("1*10^6", "2*10^6", "3*10^6")) + labs(title =
  "Scatterplot of Average Rating vs Rating Standard Deviation - By Genre",
  x = "Average Rating", y = "Rating Standard Deviation") +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_Genre.Summary.Scatterplot)

ggsave(filename = "Ratings_by_Genre_Summary_Scatterplot.png",
       plot = Ratings_by_Genre.Summary.Scatterplot, path = "Figures")

# Metrics and figures by rating lapse:

# Define a dataset consisting of the frequency, average rating and rating
# standard deviation for each rating_lapse value.

Ratings_by_Rating_Lapse.Distribution <- edx %>% group_by(rating_lapse) %>%
  summarize(Count = n(), Mu_gvn_Rating_Lapse = mean(rating),
```

```
  Sd_gvn_Rating_Lapse = sd(rating), .groups = "drop")

# Define, show and save a histogram showing the frequency of each value of
# the rating_lapse variable.

Ratings_by_Rating_Lapse.Histogram <- Ratings_by_Rating_Lapse.Distribution %>%
  ggplot(aes(rating_lapse, Count)) + geom_col(fill = "cornflowerblue",
  color = "white") + labs(x = "Rating Lapse", y = "Rating Count",
  title = "Distribution of Rating Lapse") +
  scale_x_continuous(limits = c(-1, 100)) + scale_y_log10() +
  ggthemes::theme_economist() + theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5),
  legend.position = "none")

Show(Ratings_by_Rating_Lapse.Histogram)

ggsave(filename = "Ratings_by_Rating_Lapse_Histogram.png",
  plot = Ratings_by_Rating_Lapse.Histogram, path = "Figures")

# Define, show and save a scatterplot showing average ratings against
# number of years lapsed between movie release and rating, including a
# smoothed trend line with a confidence interval.

Ratings_by_Rating_Lapse.Rating_Lapse_vs_Avgs.Scatterplot <-
  Ratings_by_Rating_Lapse.Distribution %>%
  ggplot(aes(rating_lapse, Mu_gvn_Rating_Lapse)) +
  geom_point(color = "steelblue4", alpha = 0.3) +
  geom_smooth() +
  labs(title = "Scatterplot of Rating Lapse vs Average Rating",
  x = "Rating Lapse", y = "Average Rating") +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_Rating_Lapse.Rating_Lapse_vs_Avgs.Scatterplot)

ggsave(filename =
 "Ratings_by_Rating_Lapse_Rating_Lapse_vs_Avgs_Scatterplot.png",
  plot = Ratings_by_Rating_Lapse.Rating_Lapse_vs_Avgs.Scatterplot,
  path = "Figures")

# Metrics and figures by movie age:
```

```r
# Define a dataset consisting of the number of movies, number of ratings,
# average rating and rating standard deviation for each movie_age value.

Ratings_by_Movie_Age.Distribution <- edx %>% group_by(movie_age) %>%
  summarize(Rating_Count = n(), Movie_Count = n_distinct(movieId),
  Mu_gvn_Movie_Age = mean(rating), Sd_gvn_Movie_Age = sd(rating),
  .groups = "drop")

# Define, show and save a histogram showing the number of ratings by movie
# age.

Ratings_by_Movie_Age.Histogram <- Ratings_by_Movie_Age.Distribution %>%
  ggplot(aes(movie_age, Rating_Count)) + geom_col(fill = "cornflowerblue",
  color = "white") + labs(x = "Movie Age", y = "Rating Count",
  title = "Distribution of Movie Age") +
  scale_x_continuous(limits = c(-1, 95)) + scale_y_log10() +
  ggthemes::theme_economist() + theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5),
  legend.position = "none")

Show(Ratings_by_Movie_Age.Histogram)

ggsave(filename = "Ratings_by_Movie_Age_Histogram.png",
  plot = Ratings_by_Movie_Age.Histogram, path = "Figures")

# Define, show and save a scatterplot showing average ratings against
# movie age, including a smoothed trend line with a confidence interval.

Ratings_by_Movie_Age.Movie_Age_vs_Avgs.Scatterplot <-
  Ratings_by_Movie_Age.Distribution %>%
  ggplot(aes(movie_age, Mu_gvn_Movie_Age)) +
  geom_point(color = "cornflowerblue", alpha = 0.3) + geom_smooth() +
  labs(title = "Scatterplot of Movie Age vs Average Rating", x = "Movie Age",
  y = "Average Rating") +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5))

Show(Ratings_by_Movie_Age.Movie_Age_vs_Avgs.Scatterplot)

ggsave(filename = "Ratings_by_Movie_Age_Movie_Age_vs_Avgs_Scatterplot.png",
  plot = Ratings_by_Movie_Age.Movie_Age_vs_Avgs.Scatterplot,
```

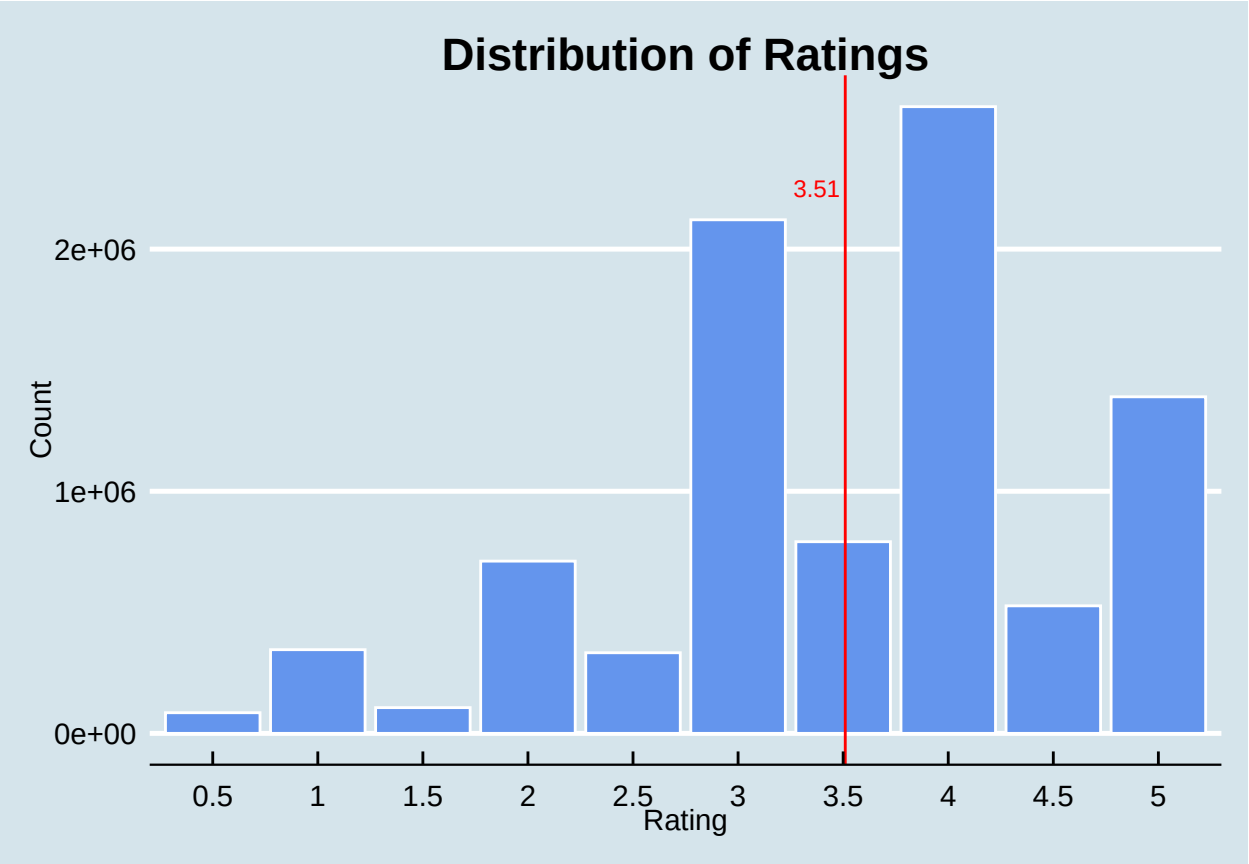
```
  path = "Figures")

# Metrics and figures by rating age:

# Define a dataset consisting of the frequency, average rating and rating
# standard deviation for each rating_age value.

Ratings_by_Rating_Age.Distribution <- edx %>% group_by(rating_age) %>%
  summarize(Count = n(), Mu_gvn_Rating_Age = mean(rating),
  Sd_gvn_Rating_Age = sd(rating), .groups = "drop")

# Define, show and save a bar chart showing the average rating by rating
# age.

Ratings_by_Rating_Age.Bar_Chart <- Ratings_by_Rating_Age.Distribution %>%
  ggplot(aes(rating_age, Mu_gvn_Rating_Age)) +
  geom_col(fill = "cornflowerblue", color = "white") +
  labs(x = "Rating Age", y = "Average Rating",
  title = "Average Rating by Rating Age") +
  geom_hline(aes(yintercept = 3.51), color = "red") +
  annotate("text", x = 4, y = 3.7, label = 3.51, color = "red", size = 3) +
  scale_x_continuous(limits = c(-1, 15)) +
  ggthemes::theme_economist() +
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5),
  legend.position = "none")

Show(Ratings_by_Rating_Age.Bar_Chart)

ggsave(filename = "Ratings_by_Rating_Age_Bar_Chart.png",
  plot = Ratings_by_Rating_Age.Bar_Chart, path = "Figures")

# Define, show and save a histogram showing rating count by rating age.

Ratings_by_Rating_Age.Histogram <- Ratings_by_Rating_Age.Distribution %>%
  ggplot(aes(rating_age, Count)) + geom_col(fill = "cornflowerblue",
  color = "white") + labs(x = "Rating Age", y = "Rating Count",
  title = "Distribution of Rating Age") +
  geom_hline(aes(yintercept = mean(Count)), color = "red") +
  annotate("text", x = 6.5, y = 8*10^5,
  label = round(mean(Ratings_by_Rating_Age.Distribution$Count), 0),
  color = "red", size = 3) + scale_x_continuous(limits = c(-1, 15)) +
  scale_y_log10() + ggthemes::theme_economist() +
```

```
  theme(axis.title.x = element_text(size = 10),
  axis.title.y = element_text(size = 10),
  plot.title = element_text(size = 12, hjust = 0.5),
  legend.position = "none")

Show(Ratings_by_Rating_Age.Histogram)

ggsave(filename = "Ratings_by_Rating_Age_Histogram.png",
  plot = Ratings_by_Rating_Age.Histogram, path = "Figures")

rm(logical_CPUs, RunRandomForest.cluster, edx.partition.indices, chunks, j,
  temp)
```

2.4.1 Ratings

Before covering any of the variables we may want to include in our predictive model for `rating`, we start by showing a histogram of the distribution of `ratings` in the `edx` dataset:

```
Ratings.Histogram
```



We can easily see how half-integer ratings are less frequent than whole-number ratings, and that the mean rating over all observations is roughly 3.51 out of a maximum of 5.

2.4.2 Possible Predictors

2.4.2.1 Calculated Importances

We now review the potentially explanatory variables' importances and their associated figures.

To start, we want to get a rough idea as to how relatively predictive of ratings each of these variables is in the `edx` dataset. To do this, we partition the dataset into 20 subsets of equal size and then simultaneously run a random forest on each of these chunks in parallel. This then allows us to compute the importances of the variables for each of these subsets, which in turn allows us to approximate each variable's importance over the entire dataset by averaging the results.

We note some things regarding this computation:

1. We resort to using parallel processing here because performing a random forest on the entire `edx` dataset would be very time-consuming, if possible at all, given the memory limit R imposes on any one object stored within an R session.
2. The averaged importances here can be relied on because they're of comparable statistical accuracy to the importances that would be calculated by performing a single random forest over the entire dataset.
3. We've chosen to partition the dataset into 20 chunks because that's the number of logical processors in my PC, and having each logical processor operate on its own chunk of data is both convenient and computationally efficient.
4. Rather than using the `randomForest()` function, we use the `ranger()` function for our random forests, since this function is specifically designed to handle large datasets as input.
5. The random forests generated use regression trees, since the `rating` variable is neither categorical nor ordinal, given the numerical distances between each of the possible ratings a movie can be assigned are known, despite the observed values of the `rating` variable not appearing continuous in nature. Consequently, the measure used here to compute variable importance is the *mean decrease in variance* when splitting, with the larger the variable importance, the more important the variable.
6. We pass the `movieId`, `userId` and `genres` variables to the `ranger()` function as factors and specify `respect.unordered.factors = TRUE` within the function's call to reflect the fact that none of these three variables should be interpreted as *ordinal* factors.
7. To estimate the overall importance of movie genre, we use the `genres` variable, rather than all 19 of the Bernoulli genre variables. This is because the Bernoulli variables are inherently positively correlated with each other and their inclusion in the random forest would result in distorted and misleading importance measures for the variables included in the trees.
8. Although we only set the number of regression trees for each random forest to 100, it can be shown that this amount more than suffices for this particular example by increasing

the `tree_num` parameter and observing how the resulting variable importances largely remain unchanged.

To get an idea as to how consistent the generated variable importances are across the different random forests, shown below are the importances calculated from the first 5 random forests:

```
VariableImpsByChunk[, 1:5]
```

```
##              var.imp_1 var.imp_2 var.imp_3 var.imp_4 var.imp_5
## movieId      109002.67 110872.06 110257.09 109380.44 110022.22
## userId       193316.96 194716.61 194869.20 195947.52 197541.00
## genres        39758.54  38491.44  39044.13  38858.46  38269.65
## rating_lapse  27459.27  27643.94  27806.63  28052.25  28153.15
## movie_age     21185.68  20791.67  20811.82  20887.37  20935.14
## rating_age    23191.29  23322.61  23317.81  23387.05  23793.63
```

Moreover, if we compute the minimum, mean and maximum importance for each variable across all 20 random forests, we get:

```
VarImps.Summary <- rbind(rowMaxs(VariableImpsByChunk),
  rowMeans(VariableImpsByChunk),
  rowMins(VariableImpsByChunk))

rownames(VarImps.Summary) <- c("max", "mean", "min")

VarImps.Summary
```

```
##       movieId   userId   genres rating_lapse movie_age rating_age
## max  113830.7 197541.0 40584.73     28153.15  21478.86   23793.63
## mean 110636.9 194648.1 38355.46     27582.90  21042.55   23325.79
## min  106857.4 192680.1 36527.72     27103.68  20723.01   23103.25
```

If we then sort the variables in decreasing order of their mean calculated importance, we see that the estimated order of importance of the variables is given by:

```
##       userId      movieId       genres rating_lapse   rating_age    movie_age
##    194648.11    110636.87     38355.46     27582.90     23325.79     21042.55
```

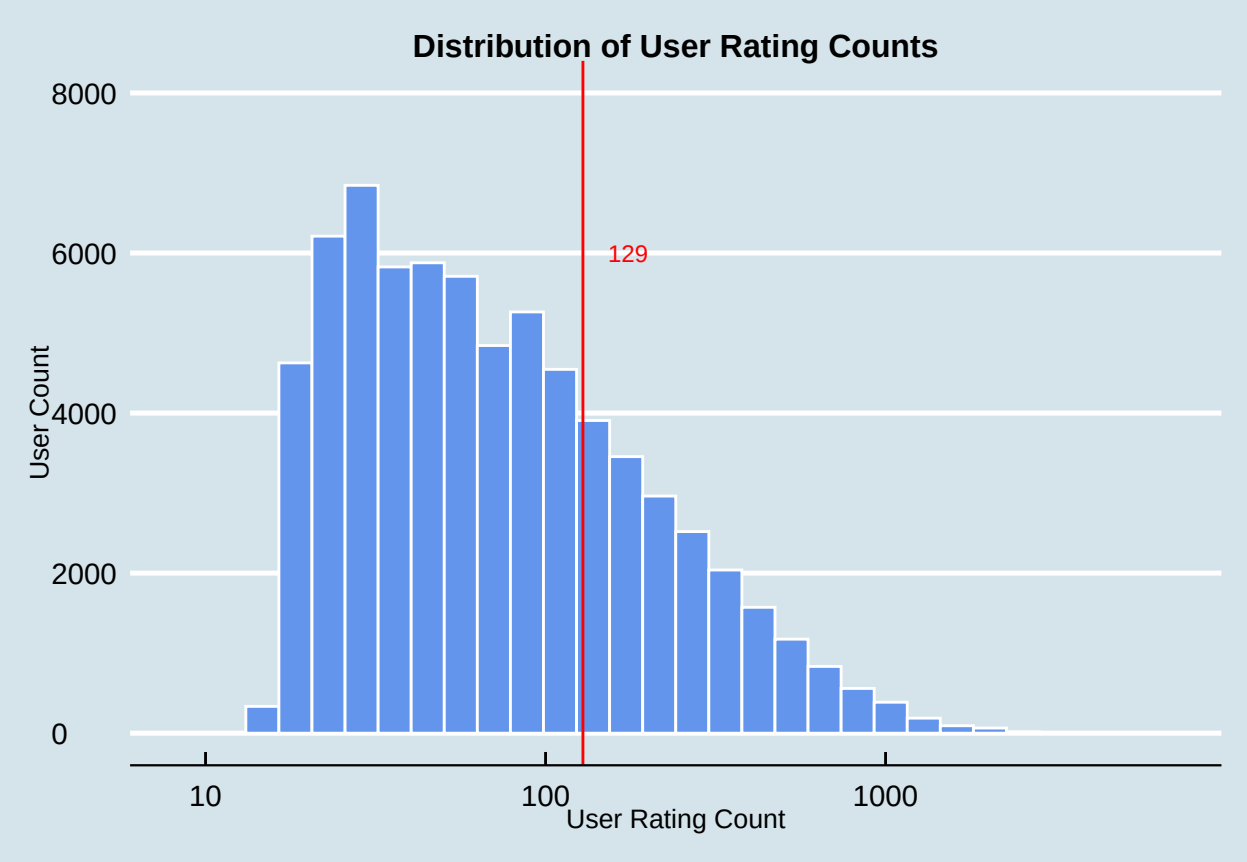
We keep these importances in mind as we now individually review each of these variables.

2.4.2.2 Distributions and Characteristics

2.4.2.2.1 Users Since the `userId` variable has the highest calculated importance, we start with this variable first.

The below histogram shows the distribution of users based on the number of ratings submitted:

```
Ratings_by_User.Counts.Histogram
```

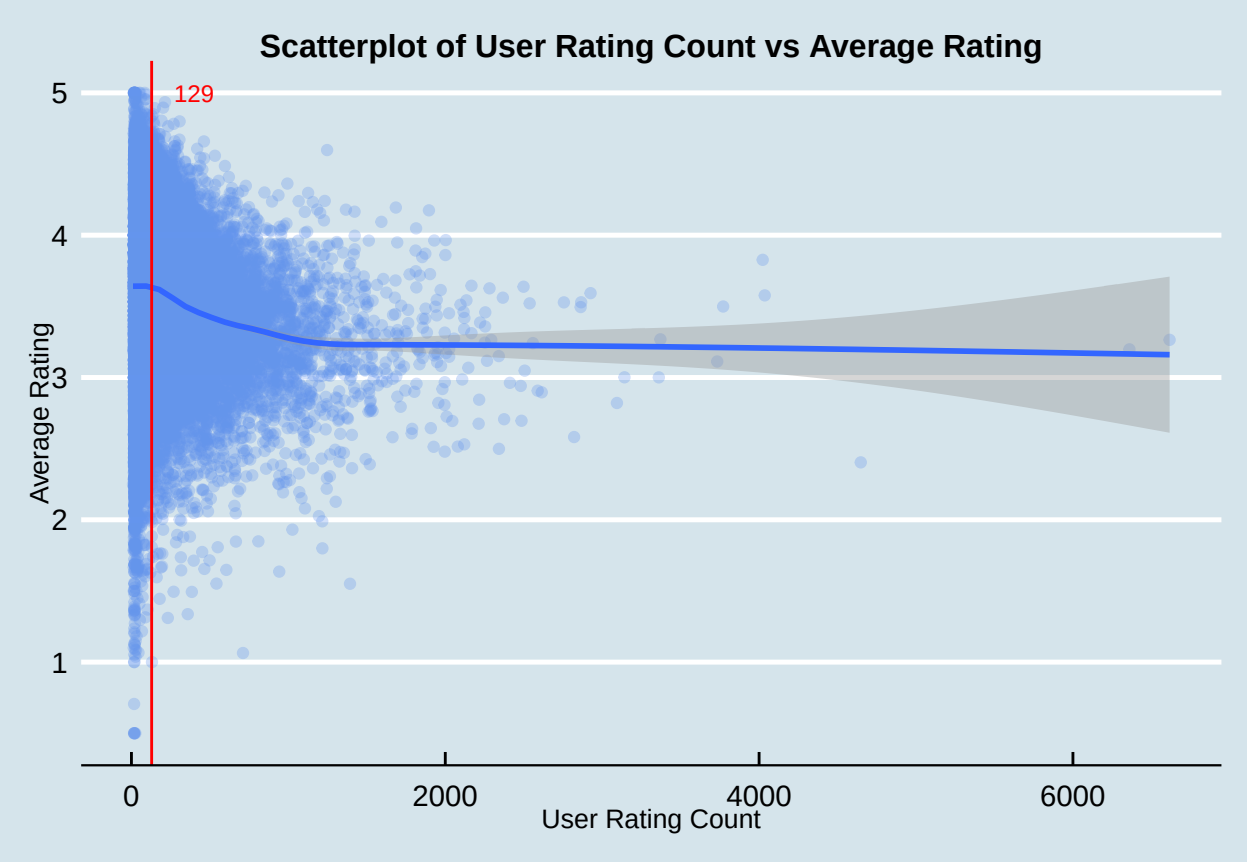


Note that the x-axis of this histogram is on a logarithmic scale.

It's immediately obvious that much more than half of all users in the `edx` dataset submitted at most the average number of ratings per user of 129, even though the maximum number of ratings for a given user was 6,616. If this histogram's x-axis weren't scaled, the distribution of rating counts by user would be even more skewed to the left. We say that the distribution of user rating counts here has negative *skewness*.

We now view a scatterplot of user count vs average rating by user:

```
Ratings_by_User.Counts_vs_Avgs.Scatterplot
```



As expected, this scatterplot also supports the fact that the distribution of user rating count is skewed to the left, despite it being a bit too dense in that region to confirm the fact that more than half of all users submitted less than 129 ratings.

This scatterplot provides us with three additional insights about the users in the `edx` dataset.

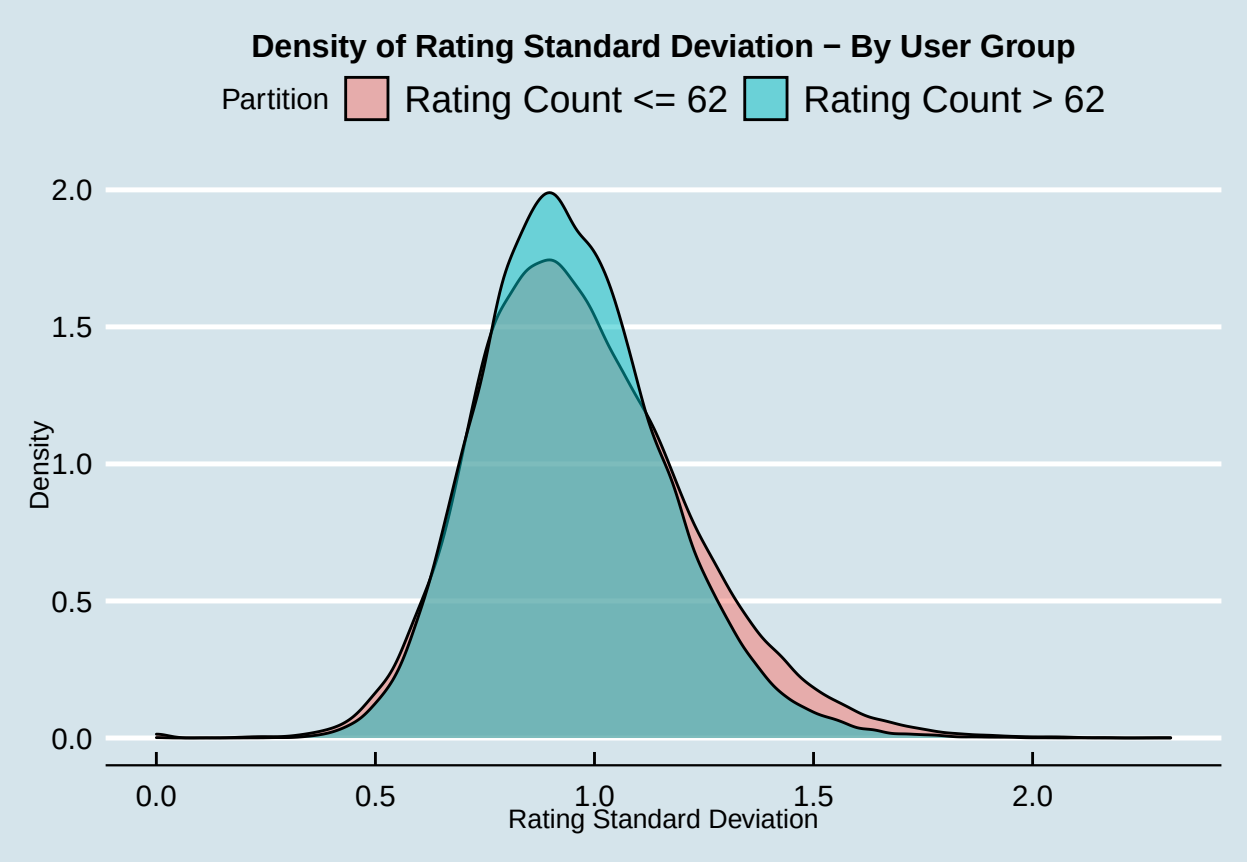
First, we see that our estimate of average rating per user remains more or less constant as user rating count increases from about 1,000 to 6,616.

Second, we see that the 95% confidence interval for expected rating given user rating count widens as the number of ratings per user increases. This is because as the number of ratings per user increases, the number of users (or observations, in this case) decreases, resulting in less credible confidence interval estimates and thus the need to widen these interval estimates to achieve the same level of confidence.

Third, we observe that it's also true that as the number of ratings per user increases, the average rating per user becomes less variable.

However, the fact that the between-user variance of the expected rating per user decreases as the number of ratings increases should NOT be confused with what the next figure reveals about the within-group variance:

```
Ratings_by_User.StDevs.Density
```



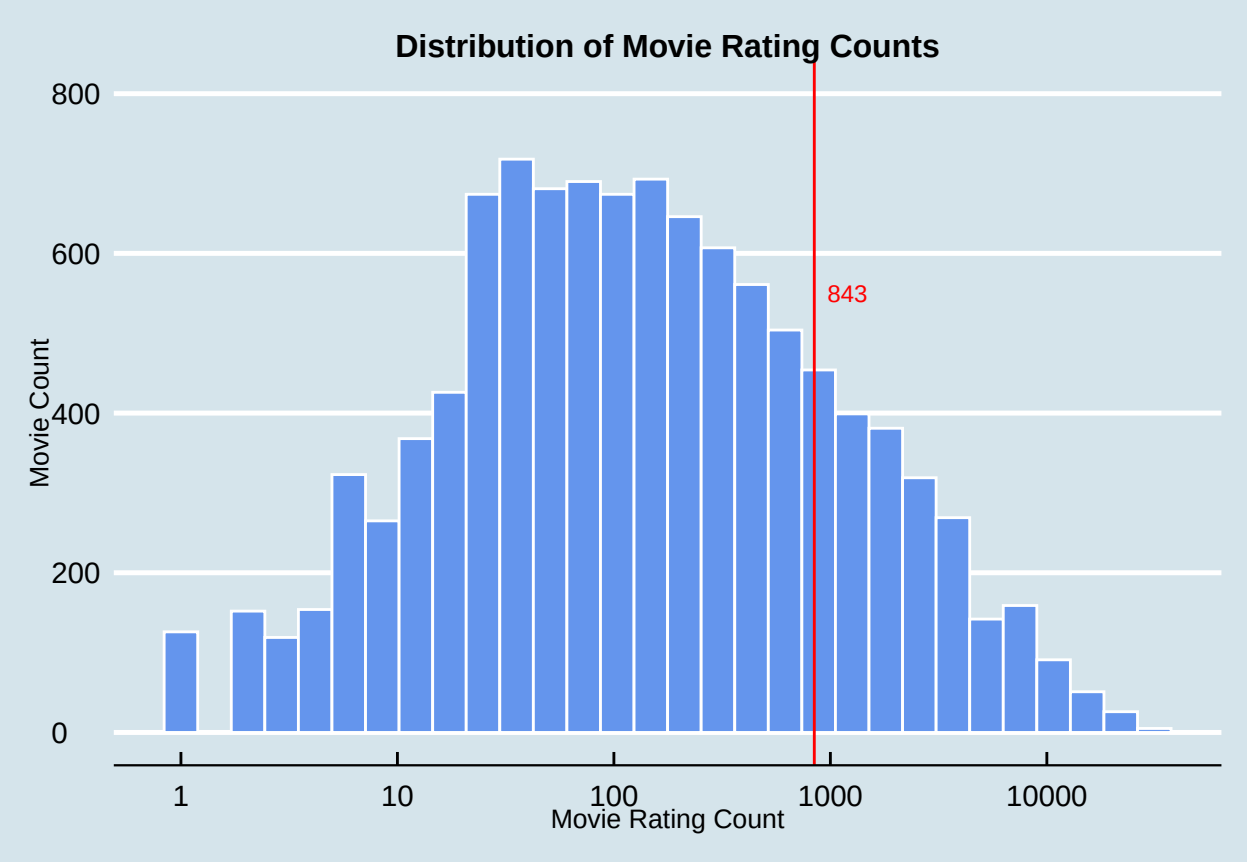
This density reveals that the distributions of rating standard deviation are basically the same between the group of users with at most 62 ratings and the group of users with more than 62 ratings, where 62 is the median number of ratings per user.

The previous scatterplot told us something about the distribution of E[rating|userId], whereas this density plot tells us something about the distribution of SD[rating|userId]. Although E[rating|userId] becomes less variable as the rating count increases for each stratum of users, SD[rating|userId] appears to be about as variable when comparing the stratum of users with at most the median number of ratings and the stratum of users with more than the median number of ratings.

For the purpose of choosing a model, the first insight isn't very useful, but the second is. The fact that SD[rating|userId] appears to have the same distribution for both groups of users tells us that the accuracy of `userId` by itself as a predictor of `rating` barely improves as the number of ratings for the user increases.

2.4.2.2.2 Movies For the `movieId` variable, we will review the same three types of figures as those reviewed for the `userId` variable, and in the same order. We thus start by looking at a histogram of rating counts by movie:

```
Ratings_by_Movie.Counts.Histogram
```

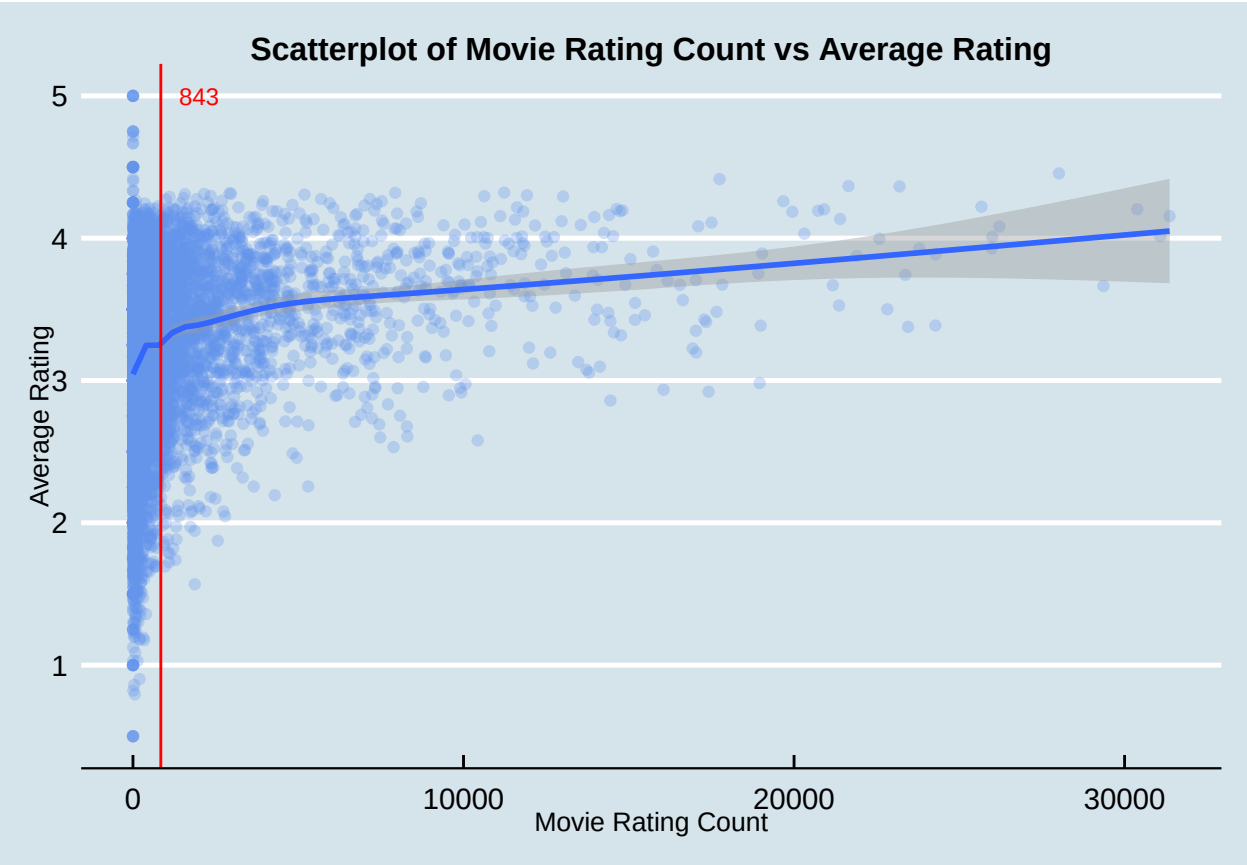


This histogram also has rating counts shown on a logarithmic scale. However, with that being said, this histogram appears to look approximately normal, and in that respect, it differs from the histogram of user rating counts. This means that the distribution of rating counts by movie would look approximately log-normal if we were to show the frequencies on a linear scale.

It's immediately obvious that much more than half of all movies in the `edx` dataset were rated at most the average number of ratings per movie of 843, even though the maximum number of ratings for any one movie was 31,362.

We now view a scatterplot of movie count vs average rating by movie:

```
Ratings_by_Movie.Counts_vs_Avgs.Scatterplot
```



As is the case for users, this scatterplot supports the fact that the distribution of movie rating counts is skewed to the left. Note that these scatterplots have rating counts shown on their true, linear scale (which is why the distribution of movie rating counts appears symmetric in the previous histogram and skewed here).

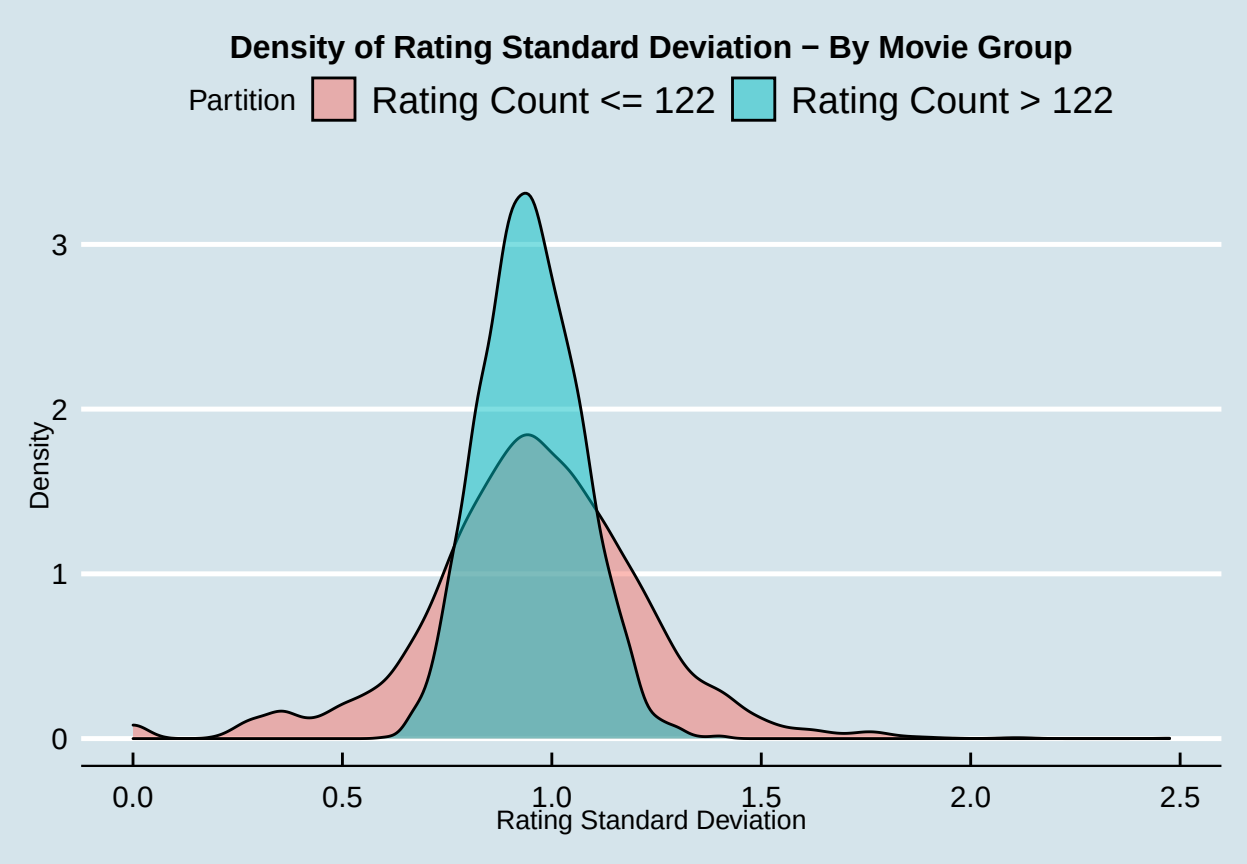
In contrast to the scatterplot for user rating counts, we see that our estimate of average rating per movie seems to increase as movie rating count increases, reflecting the same trend in the underlying data. This observation makes sense: movies that have more ratings are presumably watched by more users, and movies that are watched by more users are more likely to be rated higher, since the higher a movie is rated on average by users, the more likely a given user is to watch that movie. This is an example of *reverse causation*.

For the same reason given for users, we see that the confidence interval for our estimate of expected rating given movie rating count widens as the number of ratings per movie increases.

Lastly, we observe that as the number of ratings per movie increases, the average rating per movie becomes less variable, although it appears as though this effect isn't as pronounced as it is for users.

Again, care should be taken to avoid confusing this last observation about between-group variance with what the following density plot reveals about the within-group variance for movies:

```
Ratings_by_Movie.StDevs.Density
```



Here, we see that the distribution of SD[rating|movieId] differs significantly between the two strata of movies, with the rating standard deviation for the group of movies with more than the median number of ratings per movie (122) distributed much more closely around its mean.

In particular, we note that the rating standard deviation for this group is capped at a maximum of roughly 1.42, whereas the maximum rating standard deviation for the group of movies with rating counts at most the median is capped at a much higher 2.47.

It's then natural to ask if E[rating|movieId] is more accurate as a predictor for the movies in the first group than it is for the movies in the second group.

At first, one might speculate that, in terms of mean squared error, the higher maximum standard deviation is offset by the lower minimum standard deviation, and more generally, that the longer tail to the right of the mean rating standard deviation is offset by the group's longer tail to the left of the mean rating standard deviation, since both distributions are roughly symmetric.

In other words, one might argue that the symmetry of both distributions negates the effect on *mean squared error* that the higher *kurtosis* of the distribution corresponding to movies with at most the median number of ratings per movie might have if one were to predict ratings using E[rating|movieId] for each group separately and compare the results.

However, in theory, this isn't true. To see why, we assume that we'd be predicting the ratings in either the `edx` dataset or in some other dataset with similarly distributed movies and ratings, such as the `validation` dataset that we'll ultimately be testing our model on.

For simplicity, without any loss of generality, let's suppose the mean rating standard deviation for both strata of movies is 1.0, rather than their true means of roughly 0.95, and that both distributions are perfectly symmetric, rather than only roughly symmetric.

Because each distribution is symmetric about a mean of 1.0, we know that each movie in each group can be paired with another movie in the same group such that both movies are equidistant from, but on opposite sides of, 1.0.

Moreover, because each distribution represents 50% of all movies, meaning the two strata of movies contain the exact same number of movies, we know that each movie in the group with few ratings per movie can be paired with a movie in the group with many ratings per movie in such a way that the distance from the mean of the rating standard deviation of the movie in the first group must be greater than or equal to the distance from the mean of the rating standard deviation of the movie in the second group.

This means that, after some algebraic manipulation, the *mean squared error* of the estimator, E[rating|movieId], over the ratings in each group can be expressed as $\frac{2}{N}\sum_j(1+\delta_{i,j}^2)$, with $i = 1$ and $i = 2$ corresponding to the expressions for the MSEs over the first and second group, respectively, and j indexing each mapped equidistant movie pair between both groups, such that $\forall\ j,\ \ 0 \leq \delta_{2,j} \leq \delta_{1,j}$.

Thus, if the above assumptions about these distributions were to hold precisely, the MSE of E[rating|movieId] for the movie group with movies having rating counts less than or equal to the median rating count per movie would be at least as large as the MSE for its counterpart.

Even though we know this isn't the case, these distributions are still largely consistent with the above assumptions, and this is reflected in their MSEs:

```
cat("E[Var(rating|movieId, Count <= 122)]:",
  mean(Ratings_by_Movie.Distribution$Sd_gvn_Movie[
  Ratings_by_Movie.Distribution$Count <= 122]^2, na.rm = TRUE))
```

```
## E[Var(rating|movieId, Count <= 122)]: 0.985583
```

```
cat("E[Var(rating|movieId, Count > 122)]:",
  mean(Ratings_by_Movie.Distribution$Sd_gvn_Movie[
  Ratings_by_Movie.Distribution$Count > 122]^2, na.rm = TRUE))
```

```
## E[Var(rating|movieId, Count > 122)]: 0.916286
```

We now move on to reviewing movie genres.

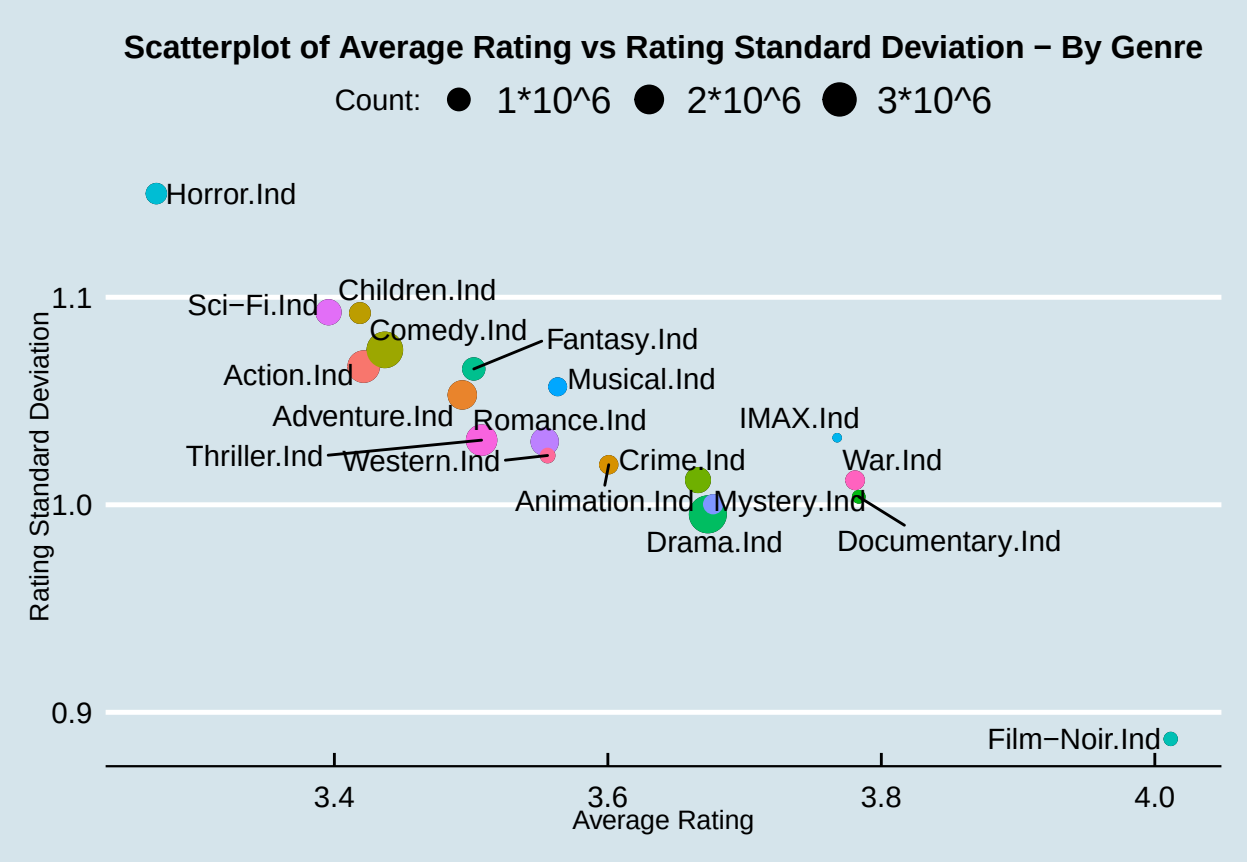
2.4.2.2.3 Genres For movie genres, our analysis will be quick, despite this feature requiring the most special treatment throughout the code of this project.

By taking `genres` in the `edx` dataset and defining 19 Bernoulli variables corresponding to

each of the unique genres appearing throughout the different combinations in the domain of the `genres` feature, we can perform a cleaner analysis of movie genres and build a more robust model including the movie genres.

The following scatterplot summarizes these 19 movie genres and their distributions:

```
Ratings_by_Genre.Summary.Scatterplot
```



Here, we plot a point for each genre based on the average rating for movies in that genre and the rating standard deviation of movies in that genre, and indicate the number of movies within the genre by the size of its plotted point.

Hence, we see that, for example, the `Film-Noir` genre has the highest average rating and lowest rating standard deviation, and is also one of the least popular genres.

At the other extreme, `Horror` films are the lowest rated and most variable of the genres, with a popularity that falls somewhere in the middle of all the genres.

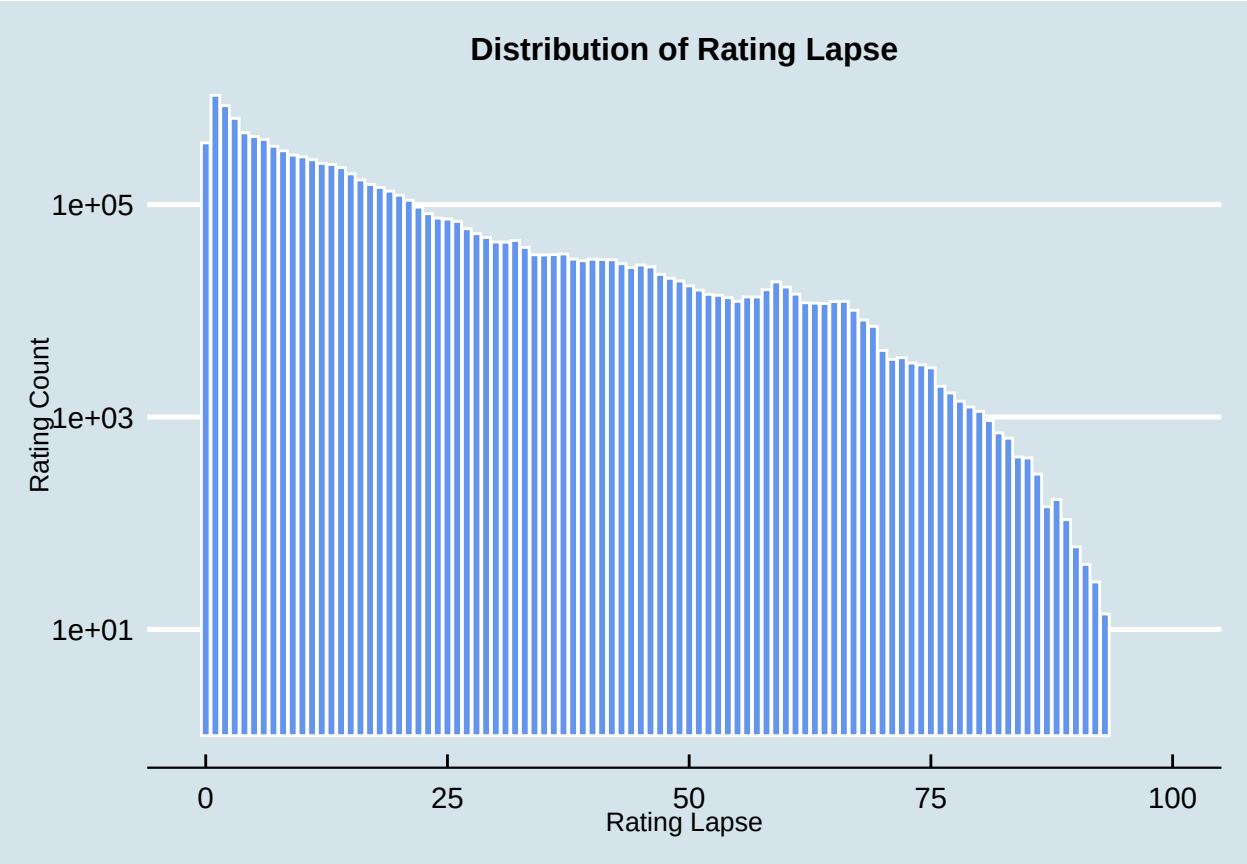
An interesting observation is that all of movie genres appear to follow the same trend: the higher the average rating, the less variable the individual ratings. We can interpret this as telling us that there tends to be agreement between users over the quality of well-liked genres.

We'll later see how movie genres will be incorporated into our model once we get to the next section that covers the training and testing of our model.

2.4.2.2.4 Rating Lapses We now turn to the `rating_lapse` variable, which tells us the number of years that have passed between the release of a movie and the date it was rated.

We start by showing the distribution of this variable in the following histogram:

```
Ratings_by_Rating_Lapse.Histogram
```



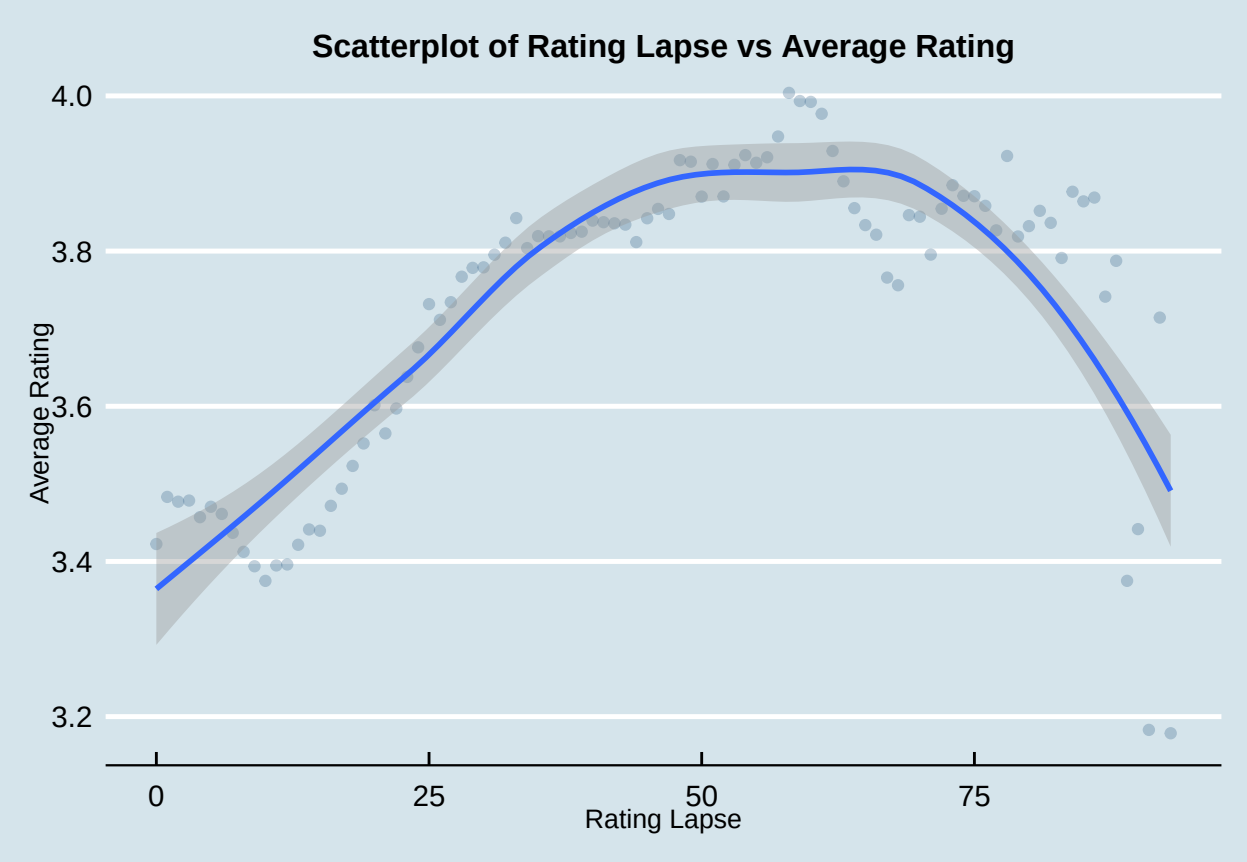
We can see that the domain of this variable ranges from 0 to 93 years, with the number of ratings steadily decreasing at a steeper rate as `rating_lapse` increases.

This makes sense intuitively. The frequency with which a movie is viewed likely typically decreases over time as the movie ages and becomes less relevant to users.

We'll return to this histogram once we review our two other remaining variables of interest.

We now turn to a scatterplot of this variable up against average rating:

```
Ratings_by_Rating_Lapse.Rating_Lapse_vs_Avgs.Scatterplot
```



We note that the line fit to this scatterplot highlights an interesting trend in the data: the average movie rating appears to decrease between the first year and tenth year, and then follows an upside down U-shaped trend thereafter, climaxing around year 58.

If we stop and think for a moment, a sound argument can be made for this.

It's possible that the peak of the trend corresponds to the time window during which previously watched movies are most likely to be watched again, perhaps due to nostalgia. For instance, older users might watch movies they remember watching and enjoying as kids.

The fact that the average rating decreases as we move along this trendline away from its apex in both directions could be justified by arguing that watching old favorites any sooner might be premature, as the effect of nostalgia hasn't yet kicked in, and that users simply might not live long enough to watch such movies for the effect of nostalgia any later than during that time window.

For now, we leave it at that, but we'll also be returning to this figure once we review the `movie_age` and `rating_age` variables.

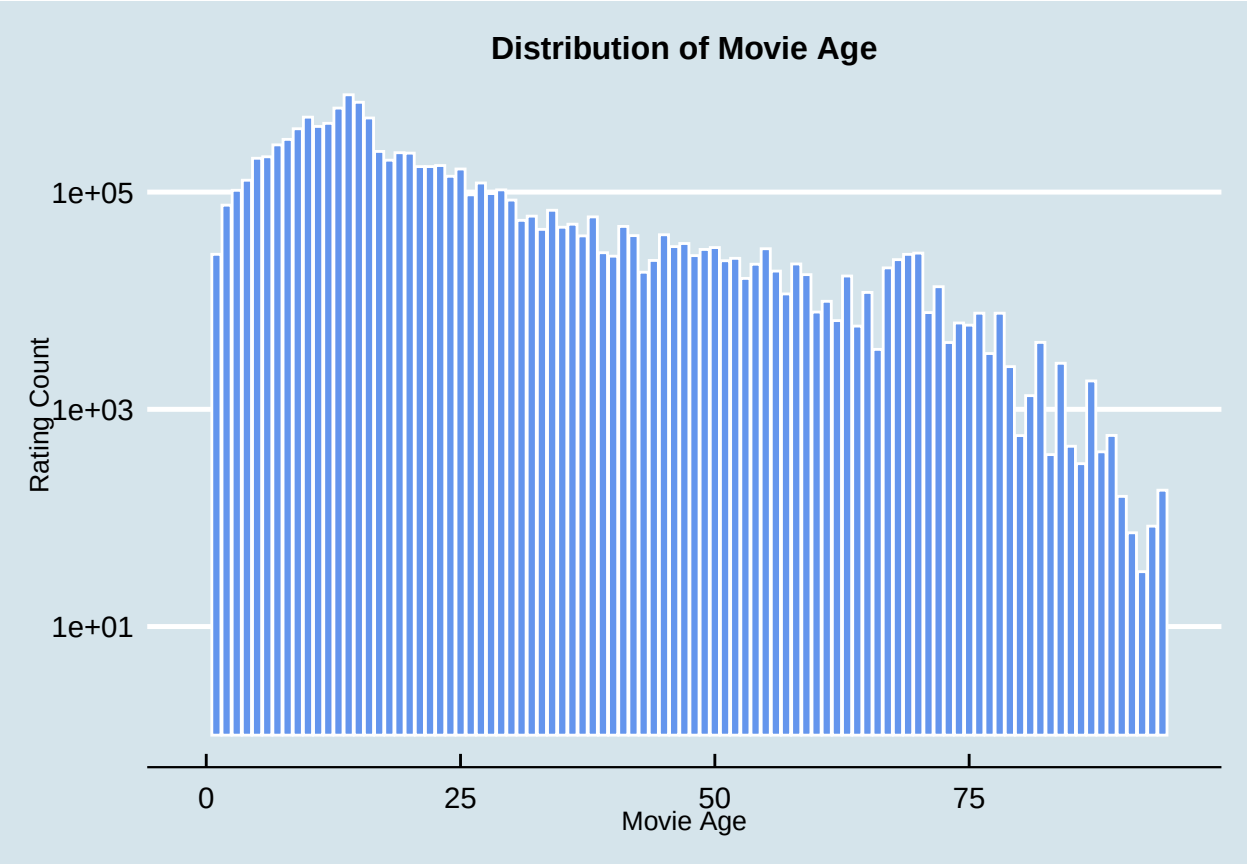
2.4.2.2.5 Movie Ages The `movie_age` variable tells us the age of each movie as of 2009, since 2009 is the most recent year included in the `edx` dataset, between movie release years and rating timestamps.

For this variable, we show the same types of figures that we just showed for the `rating_lapse`

variable, for a reason that will soon become clear.

The distribution of this variable can be seen in the following histogram:

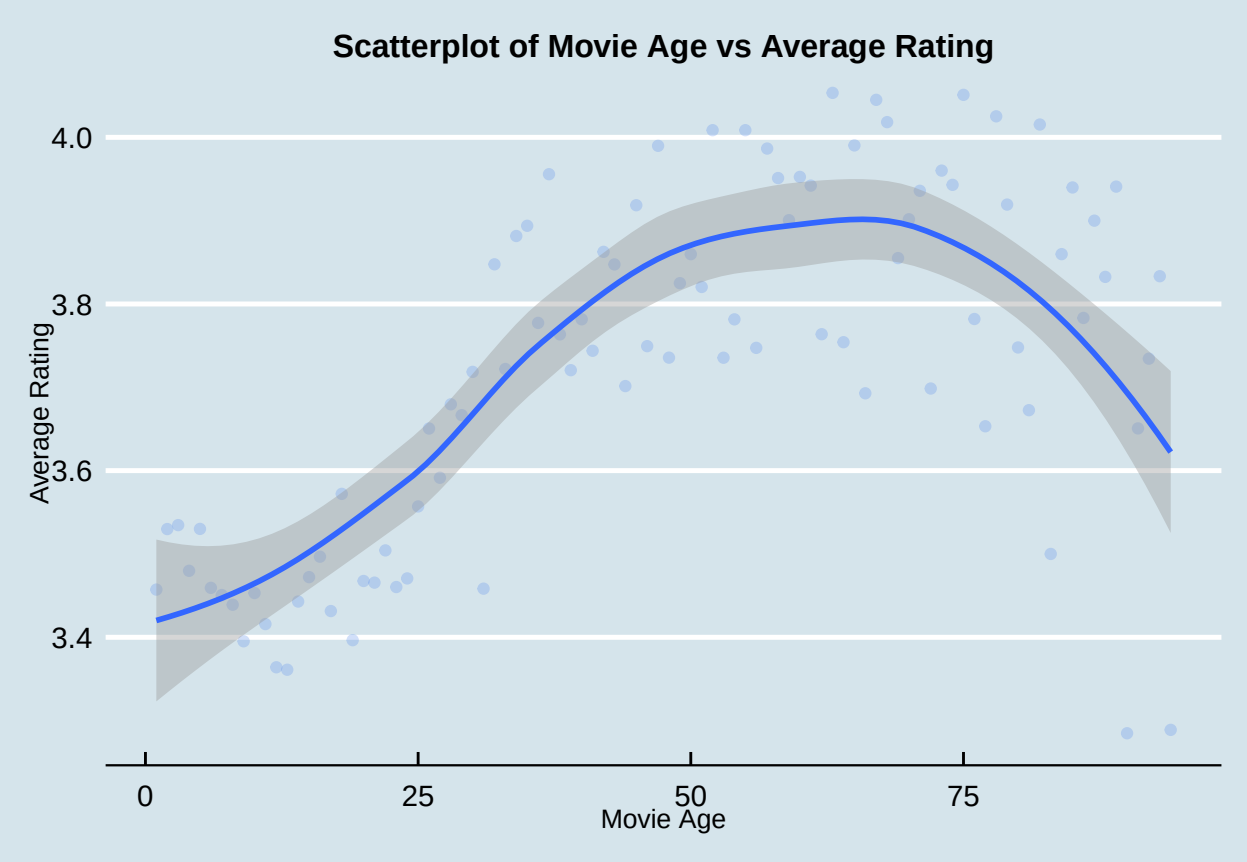
```
Ratings_by_Movie_Age.Histogram
```



We note that the domain of this variable ranges from 1 to 94 years, with the number of ratings roughly decreasing as `movie_age` increases.

Before further commenting on `movie_age`'s distribution, we quickly show the following scatterplot of this variable up against average rating:

```
Ratings_by_Movie_Age.Movie_Age_vs_Avgs.Scatterplot
```



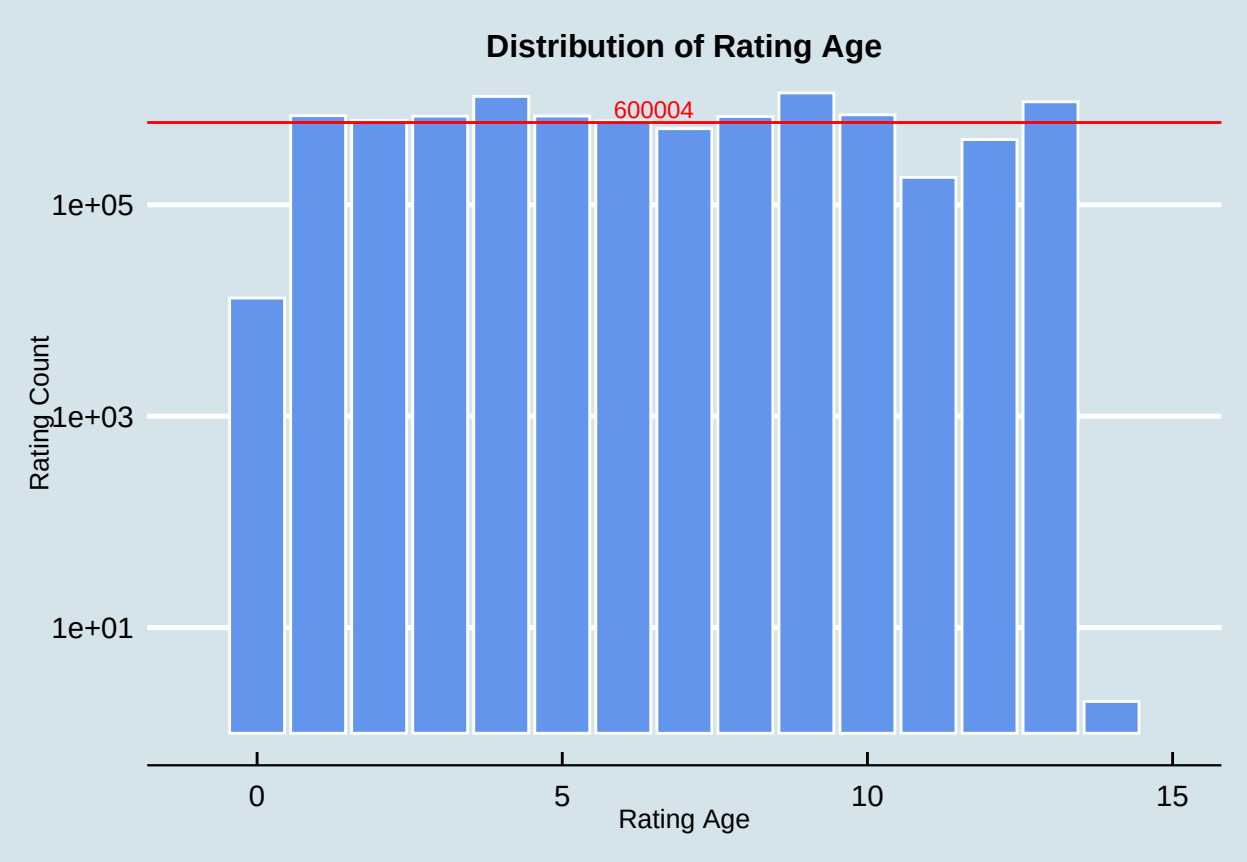
At this point, it occurs to us that the shape of the trendline estimated in this scatterplot is similar to the shape of the trendline we estimated in the scatterplot for `rating_lapse`.

Furthermore, returning to the previous histogram, it occurs to us that the distribution of `movie_age` apppears to have more or less the same shape as the distribution of `rating_lapse`.

Clearly, both variables have very similar distributions and effects on average rating. However, for a complete picture of the relationship between both variables, we'll need to review the `rating_age` variable, which we now cover.

2.4.2.2.6 Rating Ages Analogous to how we defined the `movie_age` variable, we define the `rating_age` variable as the age of each rating as of 2009. The distribution of this variable can be seen below:

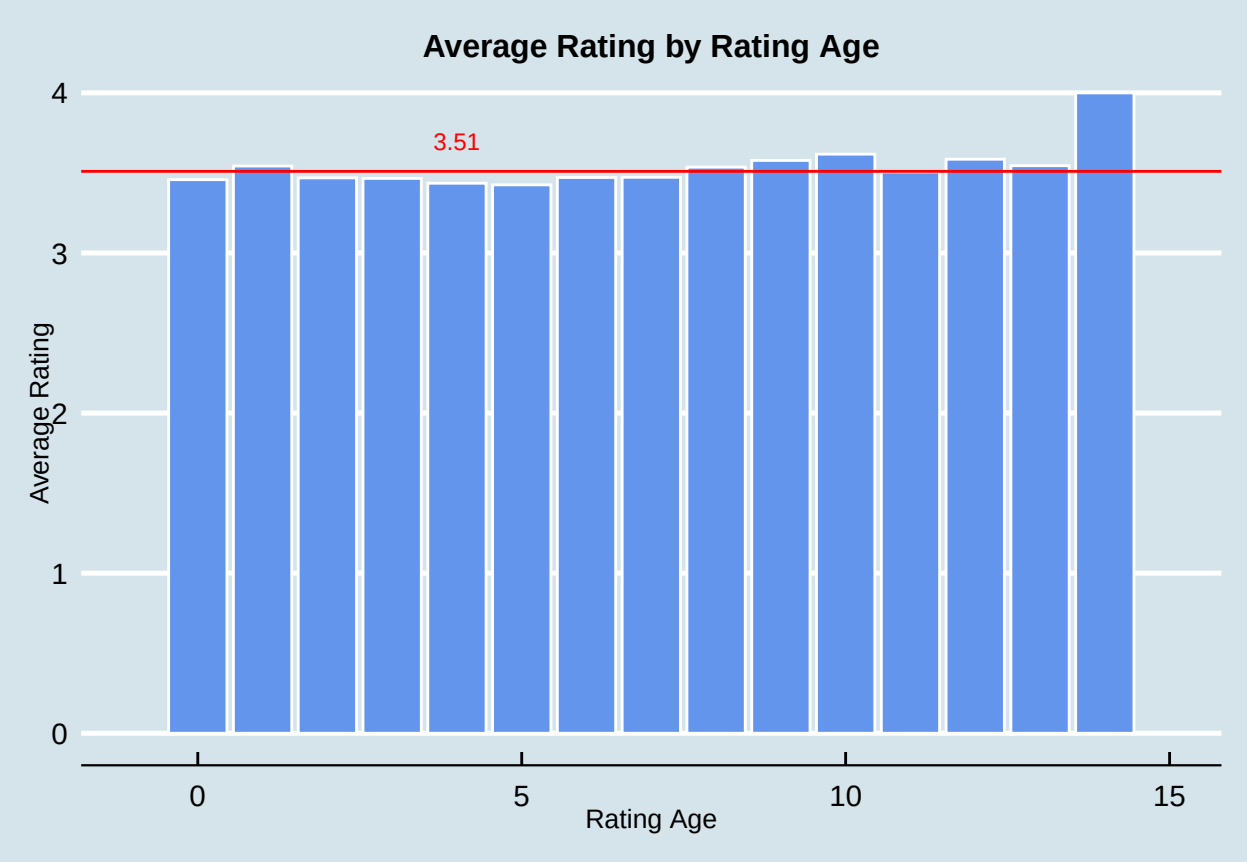
```
Ratings_by_Rating_Age.Histogram
```



The domain of this variable ranges from 0 to 14 years, with the frequency of each of its values being more or less the same, except for the exclusions of years 0 and 14.

As for how the conditional mean of rating with respect to `rating_age` looks, we refer to the following bar chart:

```
Ratings_by_Rating_Age.Bar_Chart
```



We see from this bar chart that the distribution of E[rating|rating_age] is mostly uniform as well.

We will elaborate further on these findings for `rating_age` later, once we summarize our insights and decide on our model.

2.4.3 Applying our Findings

Given what we've learned about the above variables, we now decide which of them will be included in our model, and how.

2.4.3.1 Included Predictors and Their Order

In the next few sections, we justify our decision to include movies, users, rating lapses and genres (in that particular order) in our model.

2.4.3.1.1 Users vs Movies - Which is the Real Top Dog? Recall that the variable importances generated by our random forest indicate that users are the most predictive of movie ratings, followed by the movies themselves, their genres, the number of years until they're rated, their ratings' ages, and then lastly their ages.

Although these importances seem consistent with what we might have guessed, we shouldn't just go ahead and take them at face value. This is because, as a measure of *impurity*, variance

is a biased metric when used to estimate variable importance from generated random forests using regression trees.

In particular, importances of both continuous predictors with many distinct observed values and categorical predictors with a large number of observed levels tend to be overestimated. This favor for such variables follows from the fact that there are many more potential values to split on for these variables than there are for other variables. This means that the trees used to generate these random forests have a much higher propensity to choose such variables as splitting variables at nodes simply by chance, even if these variables have very little true predictive value with respect to their response variables.

Since there are 69,878 unique `userId` values and 10,677 unique `movieId` values in the `edx` dataset, it's likely that the importances of these variables are overestimated to some degree, and that the importance of 194,594 for `userId` is more inflated than the importance of 111,384 for `movieId`. Consequently, the gap between the true importances of these two variables is likely smaller than the estimated 83,210, and it's even possible for the inequity in inflation to be so substantial that `movieId` is actually the most important variable, with `userId` the second most important.

Although it's difficult to quantify just how much these biases contribute to the calculated importances of these variables, if we revisit their distributions, we may be able to accurately qualify these contributions.

Recall how the density plot for `userId` revealed to us that the distributions of the standard deviation of ratings for the two subsets of `edx` split by the median user rating count were almost identical, and how the density plot for `movieId` conversely revealed to us that the distributions of the standard deviation of ratings for the two subsets split by the median movie rating count differed significantly, with one distribution looking just like the two distributions in the density plot for `userId` and the other looking much less variable.

Speculating that this difference in characterization between the two plots might have implications regarding the distributions of these variables, we quickly calculate the MSE of the conditional means of `rating` with respect to these variables:

```
cat("E[Var(rating|userId)]:",
  mean(Ratings_by_User.Distribution$Sd_gvn_User^2))
```

```
## E[Var(rating|userId)]: 0.9728843
```

```
cat("E[Var(rating|movieId)]:",
  mean(Ratings_by_Movie.Distribution$Sd_gvn_Movie^2, na.rm = TRUE))
```

```
## E[Var(rating|movieId)]: 0.9505766
```

It turns out that the MSE of E[rating|movieId] is actually lower than the MSE of E[rating|userId], despite their calculated importances suggesting otherwise!

This is mainly because, in addition to the fact that the MSE of E[rating|movieId] is much smaller over the set of all movies with more than the median number of movie ratings than it

is over its complement set, and is also much smaller than the MSEs of E[rating|userId] over both subsets of `edx` partitioned by the median user rating count, the subset of all movies with more than the median number of movie ratings accounts for nearly all of the ratings in the `edx` dataset:

```
cat(round(100*(sum(Ratings_by_Movie.Distribution$Count[
  Ratings_by_Movie.Distribution$Count > 122 &
  !is.na(Ratings_by_Movie.Distribution$Sd_gvn_Movie)])/
  sum(Ratings_by_Movie.Distribution$Count[
  !is.na(Ratings_by_Movie.Distribution$Sd_gvn_Movie)])), 2), "%", sep  = "")
```

```
## 97.64%
```

Between our observation regarding the likely disproportionate biases in the importances of `userId` and `movieId` and the fact that `movieId` actually serves as the better choice for a baseline predictor for our model, we decide to treat `movieId` as the most important predictive variable of the bunch, and `userId` the second most important.

2.4.3.1.2 Genres - Another Favored Variable As mentioned in the previous section, categorical predictors with a large number of observed levels tend to have inflated variable importances when calculated using random forests.

Even though the 796 unique levels of the `genres` variable pale in comparison to the number of unique levels for `userId` and `movieId`, the calculated importance for this variable is still likely inflated (albeit less much so than that for `userId` and `movieId`).

Considering the calculated importances for `genres` and `rating_lapse` are 38,067 and 27,599, respectively, we reason that it's probable that the true importance for `genres` would fall at or below the calculated importance of `rating_lapse` (which shouldn't be inflated), and proceed by transposing the importance ranks of these two variables, resulting in `rating_lapse` being interpreted as our third most important variable and `genres` as our fourth most importance variable.

2.4.3.1.3 Rating Age - Some Variables Just Don't Cut It Recall our observations regarding `rating_age`. We saw that both this variable's distribution, as well as the distribution of E[rating|rating_age], are mostly uniform.

Although this variable's calculated importance is slightly higher than the calculated importance of `movie_age`, we need to remember that the inclusion of `userId`, `movieId` and `genres` in the regression trees of our random forest introduces some bias towards those variables when calculating importances, which in turn means that the resulting calculated importances of all other variables in the trees might be skewed as well.

For these reasons, we discount the fact that `rating_age` has a slightly higher calculated importance than `movie_age`, and exclude it from our model.

2.4.3.1.4 Rating Lapse vs Movie Age - OVB? Who Cares! Another consideration when deciding which predictive variables to include in a model is how the predictors themselves relate to one another.

Sometimes, predictive variables exhibit *perfect multicollinearity*, meaning they can be expressed as linear combinations of each other, whereas other times, predictive variables are strongly, but not perfectly, correlated, in which case they are sometimes said to exhibit *imperfect multicollinearity*.

In the first case, it's common practice to exclude some of the offending predictive variables so that no *perfect multicollinearity* exists between the predictors used in the model. In the second case, it's common practice to include all of the offending predictive variables so that no *imperfect multicollinearity* exists between the predictors used in the model and those omitted from the model (i.e. - so that there's no *omitted variable bias (OVB)*).

Note, though, that these concepts aren't particularly important in our case, as we won't be making use of any type of linear regression in the derivation of our model from which coefficients that we want to interpret are estimated.

As we previously observed, the distributions of both the `movie_age` and `rating_lapse` are very similar, as are their predictive relationships with the `rating` variable. This isn't due to chance.

Recall that we originally defined `movie_age = pmax(0, 2009 - release_year)`, `rating_age = pmax(0, 2009 - rating_year)`, and `rating_lapse = pmax(0, rating_year - release_year)`.

Disregarding the inclusion of `pmax()` in each expression to correct for potentially bad data, we essentially have `movie_age = 2009 - release_year`, `rating_age = 2009 - rating_year`, and `rating_lapse = rating_year - release_year`.

If we then solve the first two equations for `release_year` and `rating_year` and substitute the resulting identities into the above expression for `rating_lapse`, we see that we end up with `rating_lapse = movie_age - rating_age`.

Between this expression relating these variables and the fact that both `rating_age` and its predictive relationship with `rating` are mostly uniform, we would expect `rating_lapse` and `movie_age` to be very highly correlated.

Calculating the correlation between the two variables, we see that this is indeed the case:

```
cor(edx$rating_lapse, edx$movie_age)
```

```
## [1] 0.9632677
```

We will therefore appear to "break protocol" here by excluding `movie_age` from our model, despite it being highly correlated with `rating_lapse`. We do this to simplify our model for the sake of less demanding computations required to train our model. Moreover, we've chosen to exclude `movie_age` instead of `rating_lapse` simply because the latter has a higher importance.

2.4.3.2 Model Specification

Now that we've decided on the variables to be included in our model and their importances with respect to the statistical significance of their effect on ratings, we wrap up our analysis by quickly describing the form of our model.

The model we'll be using is based largely on the actual winning model of the Netflix Prize, submitted by *BellKor's Pragmatic Chaos*, and describing the process used to construct it will describe its form.

To start, we'll choose a baseline predictor. The conditional mean of `rating` given this variable will be used to directly estimate `rating`.

Then, interpreting this conditional expectation as our "first take" for our model, we'll compute its predictions and residuals. We'll then compute the conditional expectation of these residuals with respect to a second variable and add the results to our "first take", giving us our "second take" for our model.

Given our revised "second take", we'd proceed in a similar manner and compute its predictions and residuals, followed by the conditional expectation of the residuals with respect to a third variable, and add the results to our "second take" to give us our "third take" for our model.

We'd then repeat this process a number of times before choosing to stop after arriving at some "final take", which we'd then use as our model.

Note that this process is essentially *gradient boosting* and is equivalent to repeatedly applying a form of *principal component analysis (PCA)* that doesn't involve *dimension reduction* to the residuals of our model at each step.

Our model will therefore implicitly assign different levels of importance to the variables within it (meaning it would be classified as a type of *hierarchical model*) by choosing the order of the variables to be integrated. The first variable, our baseline predictor, will be assigned the most importance, and the last variable included will be assigned the least importance.

Note that, when we incorporate movie genres into our model, rather than using the bias of the `genres` variable, we'll be using the average of the biases of the 19 Bernoulli variables we defined that correspond to the genres that characterize a given movie. This means that, for a given observation, instead of including a bias term corresponding to the level of the `genres` variable that characterizes that observation's movie, we'll be including the average bias term over all of the unique genres that characterize that movie.

This has an effect similar to that of directly including the bias of the `genres` variable, but results in a less overtained model that better generalizes to external datasets, such as the `validation` dataset that we'll ultimately be testing our model on.

At this point, we have one thing left to mention about our model and how we'll be defining it: it will include a penalty term that we'll determine via the technique of *regularization*.

By including this term, our model's accuracy for strata with insufficient data should improve significantly. Note that regularizing our model is appropriate here since our two most important

predictors, `movieId` and `userId`, lack sufficient data for a significant proportion of their strata (recall that the unique value counts and count medians for these variables are 10,677 and 122 for `movieId`, and 69,878 and 62 for `userId`!).

Regularization works by decreasing the magnitude of biases in the data in such a way that those strata lacking credibility because of an insufficient number of data points are *shrunken* the most. This follows from the fact that the smaller the denominator of a quotient, the greater the percentage decrease in the quotient, if a fixed quantity is added to that denominator.

Below, we show the equations that define the bias terms for the various levels of the variables we'll be including in our model. These definitions are such that, subject to a regularizing penalty term constraint, the MSE of our model at each step of the model-deriving process we described earlier will be minimized.

$$\hat{\beta}_M(m) = \frac{1}{\lambda + n_M(m)} \sum_{u \in U_M(m)} \{y_{u,m} - \hat{\mu}\}$$

$$\hat{\beta}_U(u) = \frac{1}{\lambda + n_U(u)} \sum_{m \in M_U(u)} \{y_{u,m} - [\hat{\mu} + \hat{\beta}_M(m)]\}$$

$$\hat{\beta}_T(t) = \frac{1}{\lambda + n_T(t)} \sum_{m \in M_T(t)} \sum_{u \in U_{T,M}(t,m)} \{y_{u,m} - [\hat{\mu} + \hat{\beta}_M(m) + \hat{\beta}_U(u)]\}; t \equiv f(u,m)$$

$$\hat{\beta}_{G_i}(g) = \frac{1}{\lambda + n_{G_i}(g)} \sum_{m \in M_{G_i}(g)} \sum_{u \in U_M(m)} \{y_{u,m} - [\hat{\mu} + \hat{\beta}_M(m) + \hat{\beta}_U(u) + \hat{\beta}_T(t)]\}; g \equiv f(m)$$

Our model is therefore given by:

$$Y_{u,m} = \hat{Y}_{u,m} + \widetilde{Y}_{u,m} = \hat{\mu} + \hat{\beta}_M(m) + \hat{\beta}_U(u) + \hat{\beta}_T(t) + \overline{\hat{\beta}_G(g)} + \widetilde{Y}_{u,m}$$

2.5 Model Construction

With the form of our recommendation system fully specified, we're finally ready to build our model by training it on the `edx` dataset.

To do this, we first define several functions.

We define the `CalculateRMSE()` function to calculate RMSEs throughout our training and testing.

We define the `DefineGenreEffects()` function to define the genre bias for each of the 19 Bernoulli variables in our model.

We define the `CalculateGenreEffect()` function to take the average genre bias between all genres characterizing a given movie.

We define the `Predict()` function to predict ratings throughout our training and testing.

We define the `GenerateCvRMSEs()` function to generate the RMSE matrix resulting from our cross-validation algorithm.

After defining these functions, we then simply set the number of folds we want to use for our cross-validation, set the range of possible values of λ (the penalty parameter for regularization) we want to consider for our final model, perform cross-validation, choose a tuned value of λ for our model, calculate the various biases/effects our model will use (and thus determine our model), predict the ratings in the `validation` dataset using our model, and lastly calculate the RMSE of our model over the `validation` dataset.

```
# Step4_TrainAndTestModel #

# If the edx and validation datasets aren't already loaded in the R session,
# load them.

if(!exists("edx")){load("Data/R Data/edx.rda")}

if(!exists("validation")){load("Data/R Data/validation.rda")}

# Define a function to calculate RMSE.

CalculateRMSE <- function(Actual, Prediction){
  sqrt(mean((Actual - Prediction)^2))
}

# Define a function to calculate regularized genre effects.

DefineGenreEffects <- function(Lambda.p, train.data.p, Mu_Hat.p,
  Movie.Effects_Hat.p, User.Effects_Hat.p, Rating_Lapse.Effects_Hat.p){
  GenreEffects <- tibble()
  for(j in 1:19){
    temp_data <- train.data.p %>% left_join(Movie.Effects_Hat.p,
      by = "movieId") %>% left_join(User.Effects_Hat.p, by = "userId") %>%
      left_join(Rating_Lapse.Effects_Hat.p, by = "rating_lapse") %>%
      group_by_(paste("`", colnames(train.data.p)[j + 8], "`", sep = "")) %>%
      summarize(Genre.Ct = n(), Genre.Effect_Hat = sum(rating - (Mu_Hat.p +
      Movie.Effect_Hat + User.Effect_Hat +
      Rating_Lapse.Effect_Hat))/(Genre.Ct + Lambda.p), .groups = "drop") %>%
      slice_max(.[, 1]) %>% .[, 2:3] %>%
      mutate(Genre = colnames(train.data.p)[j + 8]) %>% .[, c(3, 1:2)]
    if(j == 1){
      GenreEffects <- as.matrix(temp_data)
    }else{
      GenreEffects <- rbind(as.matrix(GenreEffects), temp_data)
    }
  }
  GenreEffects <- GenreEffects %>% mutate(Genre.Ct = as.numeric(Genre.Ct),
    Genre.Effect_Hat = as.numeric(Genre.Effect_Hat))
```

```
  return(GenreEffects)
}

# Define a function to calculate an average regularized genre effect.

CalculateGenreEffect <- function(G, B){
  ifelse(is.nan(sum(G*B)/sum(G)), 0, sum(G*B)/sum(G))
}

# Define a function to predict ratings for a provided dataset, based on the
# provided model.

Predict <- function(test.data.p, Mu_Hat.p, Movie.Effects_Hat.p,
  User.Effects_Hat.p, Rating_Lapse.Effects_Hat.p, Genre.Effects_Hat.p,
  CalculateGenreEffect){
  Mu_Hat.p + (test.data.p %>%
  inner_join(Movie.Effects_Hat.p, by = "movieId") %>% .$Movie.Effect_Hat) +
  (test.data.p %>% inner_join(User.Effects_Hat.p, by = "userId") %>%
  .$User.Effect_Hat) + (test.data.p %>%
  inner_join(Rating_Lapse.Effects_Hat.p, by = "rating_lapse") %>%
  .$Rating_Lapse.Effect_Hat) + (test.data.p[, 9:27] %>% as.matrix() %>%
  apply(1, function(g){CalculateGenreEffect(g,
    B = Genre.Effects_Hat.p$Genre.Effect_Hat)}))
}

# Define a function that generates a k-fold cross-validation RMSE matrix,
# with each element corresponding to a given fold and given value of the
# Lambda parameter. The contained code first defines the training dataset and
# testing dataset for each fold of the cross-validation, and then, for each
# value of Lambda, fits a model over that training dataset and evaluates that
# fitted model over the corresponding testing dataset.

GenerateCvRMSEs <- function(K.p, Lambdas.p){
  set.seed(2020, sample.kind = "Rounding")
  cv.test.indices <- createFolds(edx$rating, k = K.p)
  cv.RMSEs <- matrix(nrow = K.p, ncol = length(Lambdas.p))
  for(k in 1:K.p){
    train.data.temp <- edx[-cv.test.indices[[k]]]

    test.data.temp <- edx[cv.test.indices[[k]]]

    test.data <- test.data.temp %>%
      semi_join(train.data.temp, by = "movieId") %>%
      semi_join(train.data.temp, by = "userId")
```

```
    test.data.removed_obs <- test.data.temp %>%
      anti_join(test.data, by = c("movieId", "userId"))

    train.data <- rbind(train.data.temp, test.data.removed_obs)

    cv.Mu_Hat <- mean(train.data$rating)

    for(l in Lambdas.p){
      cv.Mu_gvn_Movie.Effects_Hat <- train.data %>% group_by(movieId) %>%
        summarize(Movie.Ct = n(), Movie.Effect_Hat = sum(rating - cv.Mu_Hat)/
        (Movie.Ct + l), .groups = "drop")

      cv.Mu_gvn_User.Effects_Hat <- train.data %>%
        left_join(cv.Mu_gvn_Movie.Effects_Hat, by = "movieId") %>%
        group_by(userId) %>% summarize(User.Ct = n(),
        User.Effect_Hat = sum(rating - (cv.Mu_Hat + Movie.Effect_Hat))/
        (User.Ct + l), .groups = "drop")

      cv.Mu_gvn_Rating_Lapse.Effects_Hat <- train.data %>%
        left_join(cv.Mu_gvn_Movie.Effects_Hat, by = "movieId") %>%
        left_join(cv.Mu_gvn_User.Effects_Hat, by = "userId") %>%
        group_by(rating_lapse) %>% summarize(Rating_Lapse.Ct = n(),
        Rating_Lapse.Effect_Hat =
        sum(rating - (cv.Mu_Hat + Movie.Effect_Hat + User.Effect_Hat))/
        (Rating_Lapse.Ct + l), .groups = "drop")

      cv.Mu_gvn_Genre.Effects_Hat <- DefineGenreEffects(l, train.data,
        cv.Mu_Hat, cv.Mu_gvn_Movie.Effects_Hat, cv.Mu_gvn_User.Effects_Hat,
        cv.Mu_gvn_Rating_Lapse.Effects_Hat)

      cv.Y_Hat <- Predict(test.data, cv.Mu_Hat, cv.Mu_gvn_Movie.Effects_Hat,
        cv.Mu_gvn_User.Effects_Hat, cv.Mu_gvn_Rating_Lapse.Effects_Hat,
        cv.Mu_gvn_Genre.Effects_Hat, CalculateGenreEffect)

      cv.RMSEs[k, which(Lambdas.p == l)] <- CalculateRMSE(test.data$rating,
        cv.Y_Hat)

      ProgressBar(round(100*(((k - 1)*length(Lambdas.p) +
        which(Lambdas.p == l))/(K.p*length(Lambdas.p))), 0))
    }
  }
  return(cv.RMSEs)
}
```

```
# Assign the number of folds to use for cross-validation and the values to
# tune Lambda over.

K <- 5; Lambdas <- seq(4.75, 5.25, 0.05)

# Define the matrix containing the output of the GenerateCvRMSEs function,
# as well as a vector consisting of the average RMS for each of the values of
# the Lambda parameter.

CvRMSEs <- GenerateCvRMSEs(K, Lambdas); Mu_CvRMSEs <- colMeans(CvRMSEs)

# Define a plot showing the average RMSEs against the values of Lambda and
# assign the tuned value of Lambda to Lambda_Hat.

Lambdas_vs_Mu_CvRMSEs.plot <- qplot(Lambdas, Mu_CvRMSEs)

Lambda_Hat <- Lambdas[which.min(Mu_CvRMSEs)]

# Define the average rating to be used for our model.

Mu_Hat <- mean(edx$rating)

# Define the movie effects to be used for our model.

Mu_gvn_Movie.Effects_Hat <- edx %>% group_by(movieId) %>%
  summarize(Movie.Ct = n(), Movie.Effect_Hat =
  sum(rating - Mu_Hat)/(Movie.Ct + Lambda_Hat), .groups = "drop")

# Define the user effects to be used for our model.

Mu_gvn_User.Effects_Hat <- edx %>% left_join(Mu_gvn_Movie.Effects_Hat,
  by = "movieId") %>% group_by(userId) %>% summarize(User.Ct = n(),
  User.Effect_Hat = sum(rating - (Mu_Hat + Movie.Effect_Hat))/
  (User.Ct + Lambda_Hat), .groups = "drop")

# Define the rating_lapse effects to be used for our model.

Mu_gvn_Rating_Lapse.Effects_Hat <- edx %>%
  left_join(Mu_gvn_Movie.Effects_Hat, by = "movieId") %>%
  left_join(Mu_gvn_User.Effects_Hat, by = "userId") %>%
  group_by(rating_lapse) %>% summarize(Rating_Lapse.Ct = n(),
  Rating_Lapse.Effect_Hat =
  sum(rating - (Mu_Hat + Movie.Effect_Hat + User.Effect_Hat))/
  (Rating_Lapse.Ct + Lambda_Hat), .groups = "drop")
```

```
# Define the genre effects to be used for our model.

Mu_gvn_Genre.Effects_Hat <- DefineGenreEffects(Lambda_Hat, edx, Mu_Hat,
  Mu_gvn_Movie.Effects_Hat, Mu_gvn_User.Effects_Hat,
  Mu_gvn_Rating_Lapse.Effects_Hat)

# Define a vector of rating predictions for the validation dataset.

Y_Hat <- Predict(validation, Mu_Hat, Mu_gvn_Movie.Effects_Hat,
  Mu_gvn_User.Effects_Hat, Mu_gvn_Rating_Lapse.Effects_Hat,
  Mu_gvn_Genre.Effects_Hat, CalculateGenreEffect)

# Calculate the RMSE of our model with respect to the true ratings in the
# validation dataset.

Y_Hat.RMSE <- CalculateRMSE(validation$rating, Y_Hat)

save(Y_Hat, file = "Data/R Data/Y_Hat.rda")
```

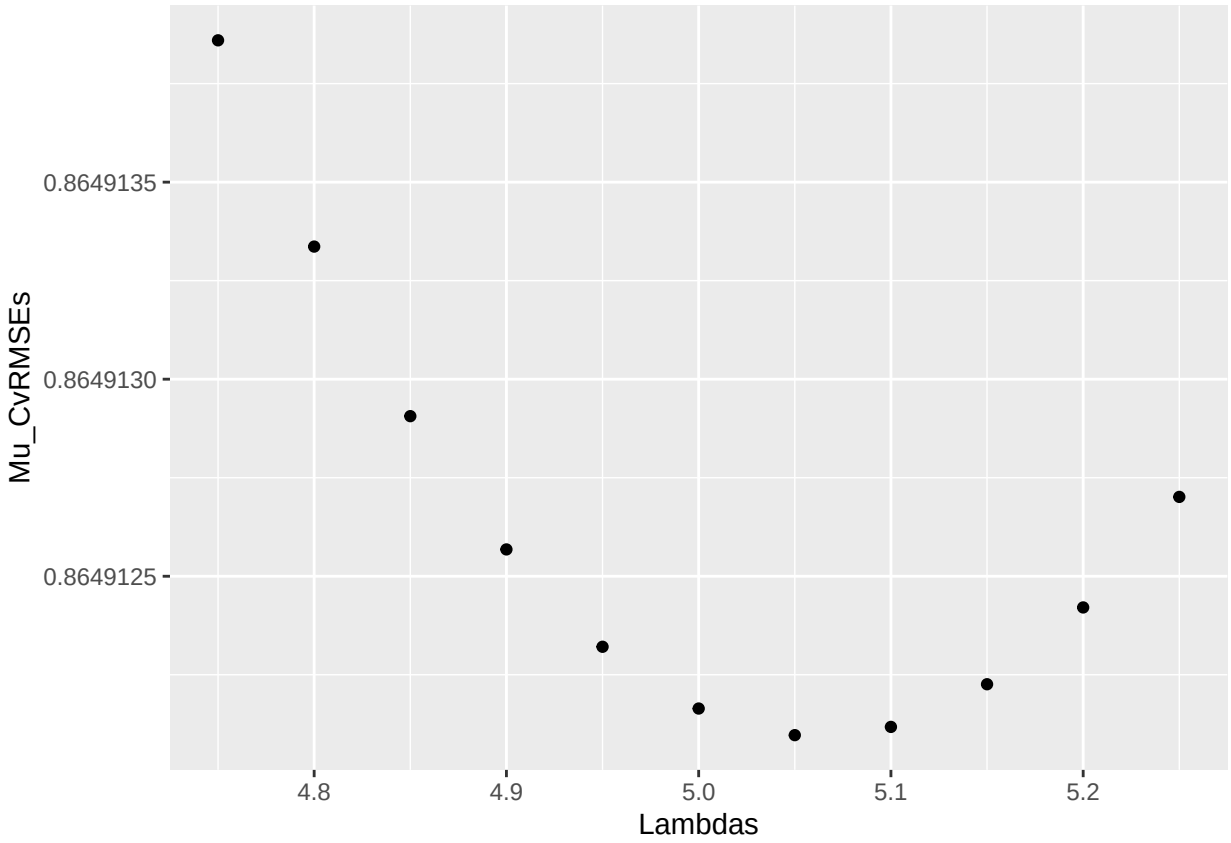
We see that the following choice of $\hat{\lambda}$ minimizes the expected MSE over the test sets of the various folds of our cross-validation procedure:

```
print(Lambda_Hat)
```

```
## [1] 5.05
```

Here is a visual showing the expected MSE as a function of the various candidate values for λ:

```
Lambdas_vs_Mu_CvRMSEs.plot
```



We note a few things about our training algorithm:

1. We set the number of folds for our cross-validation to 5 here. This may seem small, but it can be verified that performing cross-validation in this example on more than 5 folds doesn't change our choice for $\hat{\lambda}$ and improve our model's performance in any meaningful way.

2. It's important to note that each fold in our cross-validation corresponds to a pair of datasets, with one dataset used for training and another used to test the resulting trained model, and that the average of the RMSEs calculated by testing each such trained model on its testing dataset is what's then used to determine the optimal value of λ (which we denote by $\hat{\lambda}$) to use for the penalty parameter of our model (by choosing the value that minimizes this average).

3. We first set the value of $\hat{\lambda}$ so that it minimizes the expected RMSE of our model over all the folds of the `edx` dataset, and then, given this choice for $\hat{\lambda}$, we subsequently set the different bias terms for our model so that they minimize the variance of our model's error (our model's MSE minus the square of our model's bias) over the `edx` dataset.

Chapter 3

Results

We see that the model constructed results in the following RMSE:

```
print(Y_Hat.RMSE)
```

```
## [1] 0.8642294
```

For this project, the performance goal of our model was to achieve an RMSE under 0.86490 over the `validation` dataset. We therefore met our goal, indicating that our model performed satisfactorily.

Chapter 4

Conclusion

4.1 Summary

This report serves as an example walkthrough of the type of analysis typically performed in the context of recommendation system models.

The data on which this report is based was pulled from *MovieLens*, the same dataset used for the *Netflix Prize*, an open competition for the best *collaborative filtering algorithm* for predicting user ratings for films, and perhaps one of the most important catalysts for research in recommendation systems as a distinct area of machine learning.

4.2 Limitations

With that being said, this report barely scratches the surface of such models. For instance, there are numerous statistical models and algorithms that we could have also considered in this report. However, due to computational limitations resulting from the size of the training dataset used for this project, such alternatives weren't explored.

Additionally, we weren't provided with the entire *MovieLens* dataset, which, although consisting of more records and thus demanding more computationally, limited our options for our model.

4.3 Future Work

Recommendation systems constitute an area of active research with existing challenges, and there is a vast library of literature rich in the subject available to researchers as well as the general public.

As user data and the *Internet of Things* continue to play a more important and center role in consumerism and technological advances continue to yield access to greater computational

resources, the research of recommendation systems and their applications is likely to continue growing in both size and scope over the coming years.