PH125.9x Data Science Capstone Final Movie Project Report

Marina Ganopolsky

8/24/2020

Contents

Overview **3**

Dataset 4

Methods and Analysis **12**

Data Insights, Cleaning & Visualizations 12

Sparsity 12

Users 14

Movies 18

Ratings 21

Genres 23

Rating Models 27

1. Average movie rating model 27

2. Movie effect model 28

3. Movie and user effect model 30

4. Regularized Models 31

4.1 Movie Effect Regularized Model 31

4.2 Movie + User Bias Effect Regularized Model 36

4.3 Movie + User + Release Year Effect Regularized Model 40

4.4 Movie + User + Release Year + Genre Effect Regularized Model 44

Results **48**

The final RMSE value calculated from the validation set is: 0.8451003. 48

Conclusion **49**

Report Summary 49

Other Options 50

Using Various Regression Functions. 50

Possible Improvements 50

Appendix - Environment **51**

```
knitr::opts_chunk$set(cache=TRUE)
```

```
threshold_value <- 0.86490
```

Overview

This is Marina Ganopolsky's implementation of the **MovieLens Project** of the **Harvard: PH125.9x Data Science, Summer 2020: Capstone**.

The objectives of this project are:

1. To create a movie rating engine in R based on the data set provided by training a number of machine learning algorithms on the data.
2. To choose the best-performing algorithm, and make a final evaluation of said algorithm on the validation dataset using the **RMSE** (Root Mean Square Error) of the predicted ratings, **ideally < 0.8649**, and validate this with the validation data set, as described below.
3. To summarize the findings and provide further recommendations, if needed.

The dataset is provided by the staff, and is wrangled for better use during the project beyond the manipulations suggested in the assignment; the modifications will be shown in the **Dataset** section below.

A quick summary of the data wrangling modifications will be provided, as well as some visual representations of patterns present in the data, in the **Dataset** section. This will inform the reader about the **movie** trends, **user** trends and a variety of **rating** trends present in the dataset.

An exploratory analysis will be performed to generate predicted movie ratings, with various models, to be specified, until a final RMSE calculation fits our requirements. Results of the RMSE calculations will be analyzed and explained, and a conclusion will be provided.

The function used to evaluate algorithm performance is the **Root Mean Square Error**, or RMSE. RMSE is one of the most frequently used measures of the differences between values predicted by a model and the values observed. RMSE is a measure of accuracy, and a lower RMSE is better than a higher one. The effect of each error on RMSE is proportional to the size of the squared error; thus larger errors have a disproportionately large effect on RMSE. Because of this, the RMSE algorithm is sensitive to outliers.

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

With this consideration, I have massaged the data, **removing films that have less then 350 reviews**; the logic behind this is that large outliers make for a larger RMSE, as explained, and movies with only several reviews will be more obscure films reviewed by possibly extra cranky or extra friendly users, creating outliers. At the end of this document, in an addendum, I will present my findings for removing a range of film values - from 0 to 1000.

Seven (7) models will be developed, ranging from the most naive to the most complex. These will be tested and compared using the resulting RMSE in order to assess their quality. The ideal evaluation criteria for this project, as specified by the assignment, is a RMSE that is expected to be lower than **0.8649**.

After evaluating every available algorithm I have come up with using the RMSE, the best-resulting model will be used on the **validation dataset** to evaluate the quality of the predictions of the movie ratings.

This project can be found on GitHub under https://github.com/mganopolsky/r_final_harvard_data_science_project.

Dataset

The MovieLens dataset is automatically downloaded with the code provided; The data sets can be additionally downloaded here, both in zip format and folders:

- https://grouplens.org/datasets/movielens/10m/
- http://files.grouplens.org/datasets/movielens/ml-10m.zip

As specified by the project description (and the provided code) the data is split into a validation set and the "rest". Furthermore, after it was manipulated, the data is again split into **testing** and **training** sets, so as to evaluate the quality of the predictions. The calculations and algorithm verification are done on the testing and training sets, and after the final model is chosen, this algorithm is applied to the validation test to verify the hypothesis.

A fair amount of data manipulation has been done to the datasets created with the provided code. The changes include:

- **New Field** : Adding a ratings average per movie in **avg_rating**
- **New Field** : Extracting the release year of the movie from its title. (Every movie title seems to include the release year in at the end of title, surrounded by parentheses.) and creating the field **release_year** with the information
- **New Field** : Creating a field called **rating_date** , by extracting the rating date from the **timestamp** field
- **New Field** : Creating a field called **age_of_movie**, by subtracting the year the film was released from the year parameter of the current date
- **New Field** : Creating a field called **rating_year** by extracting the rating year from the **rating_date** field.
- Removing the year from the movie title
- Removing films with **350 reviews or less** - this shows to have made a slight difference over using the data as-is.
- Removing the parentheses from the **release_year** and converting it to a numerical field
- As provided, the film genres are provided as a string separated by "|". Therefor, I further manipulate the movielens dataset (and significantly increase its size) by extracting out the film genres, and creating a line per film, per genre
- After the data wrangling is complete, the dataset is split up into :

- **A validation set**, comprised of **10%** of the data complete data; this portion of the data shall not to be analyzed until the final algorithm is found to have an RMSE to be within acceptable range
- **A training dataset** - the data the algorithms will be trained on
- **A testing dataset** - the data the intermediate algorithms will be tested on

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

```
## Loading required package: tidyverse
```

```
## -- Attaching packages ---------------------------------------------------------
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.0
## v tidyr   1.1.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

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

```
## Loading required package: caret
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-proje
```

```
## Loading required package: data.table
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, first, last

## The following object is masked from 'package:purrr':
##
##     transpose
```

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

```
## Loading required package: hexbin
```

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

```
## Loading required package: lubridate

##
## Attaching package: 'lubridate'

## The following objects are masked from 'package:data.table':
##
##     hour, isoweek, mday, minute, month, quarter, second, wday, week,
##     yday, year

## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

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

```
## Loading required package: ggrepel
```

```
if(!require(ggplot2)) install.packages("ggplot2", repos = "http://cran.us.r-project.org
if(!require(cowplot)) install.packages("cowplot", repos = "http://cran.us.r-project.org
```

```
## Loading required package: cowplot

##
## ************************************************************
```

```
## Note: As of version 1.0.0, cowplot does not change the

##   default ggplot2 theme anymore. To recover the previous

##   behavior, execute:
##   theme_set(theme_cowplot())

## ************************************************************

##
## Attaching package: 'cowplot'

## The following object is masked from 'package:lubridate':
##
##     stamp
```

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

```
## Loading required package: here

## here() starts at /Users/marina/Documents/DevProjects/R/projects/r_final_harvard_data_
```

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

```
## Loading required package: formatR
```

```
library(tidyverse)
library(caret)
library(data.table)
library(stringr)
library(lubridate)
library(ggrepel)
library(ggplot2)
library(cowplot)
library(dplyr)
library(here)
library(formatR)

dl <- tempfile()
marinas_directory = "/Users/marina/Documents/DevProjects/R/projects/r_final_harvard_data
```

```
#specifying not needing to download the .zip file if located on personal computer
if (file.exists(marinas_directory)){
  setwd(marinas_directory)
}

if (file.exists("ml-10m.zip")) {

  file.link("./ml-10m.zip", dl)
} else {
  #only download the file if it doesn't exist in the current working directory.
  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")

# if using R 4.0 or later
#mutate to add the "release_year" column, format it as a numeric, and remove the (year

movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres)) %>%
  mutate(release_year =  str_extract(title, "(\\(\\d{4}\\)$)") ) %>%
  mutate(release_year = as.numeric(str_remove(str_remove(release_year, "[\\(]"), "[\\)]
         title = str_trim(str_remove(title, "\\(\\d{4}\\)$")))

#add in ratings average per movie
rating_avgs <- ratings %>% group_by(movieId) %>% summarise(avg_rating = mean(rating), r
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
ratings <- inner_join(ratings, rating_avgs, by = "movieId")

# add the date field , converting the timestamp to an actual date, and rounding to the
#use inner_join instead of provided left_join to make sure the movie titles and genres
movielens <- inner_join(ratings, movies, by = "movieId") %>%
  mutate(rating_date = round_date(as_datetime(timestamp), unit="day"),
         age_of_movie =  year(as_date(Sys.Date())) - release_year,
         rating_year = year(rating_date))
```

I will re-iterate that the algorithm development and tuning will be conducted on one part of the data, and the validation set will be used to evaluate the final chosen data model. **During no other time will the validation dataset be used.**

As a first step, in order to get some basic information about the data we're working with, we examine the **`movielens`** dataset. The original data contains six variables :

- **userID**
- **movieId**
- **rating**
- **timestamp**
- **title**
- **genres**

After massaging the data a bit, the following fields were added, as derived from the existing fields :

- **avg_rating**
- **release_year**
- **rating_date**
- **age_of_movie**
- **rating_year**
- **rating_count**

Per row, this is a representation of one user's rating of a single movie. As you can tell, the **genres** per film are currently combined into **"|"**-delimited string, which I will later separate. **As a first glance, here's a snapshot of dataset:**

```
str(movielens)
```

```
## Classes 'data.table' and 'data.frame':   10000054 obs. of  12 variables:
##  $ userId      : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId     : num  122 185 231 292 316 329 355 356 362 364 ...
##  $ rating      : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp   : int  838985046 838983525 838983392 838983421 838983392 838983392 838
##  $ avg_rating  : num  2.86 3.13 2.94 3.42 3.35 ...
##  $ rating_count: int  2412 14975 17851 16075 18925 16167 5366 34457 4016 20972 ...
##  $ title       : chr  "Boomerang" "Net, The" "Dumb & Dumber" "Outbreak" ...
##  $ genres      : chr  "Comedy|Romance" "Action|Crime|Thriller" "Comedy" "Action|Drama
##  $ release_year: num  1992 1995 1994 1995 1994 ...
##  $ rating_date : POSIXct, format: "1996-08-02" "1996-08-02" ...
##  $ age_of_movie: num  28 25 26 25 26 26 26 26 26 26 ...
##  $ rating_year : num  1996 1996 1996 1996 1996 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

The first few lines of the dataset look like this:

```
head(movielens) %>% print.data.frame()
```

```
##   userId movieId rating timestamp avg_rating rating_count
## 1      1     122      5 838985046   2.861318         2412
## 2      1     185      5 838983525   3.125209        14975
## 3      1     231      5 838983392   2.936950        17851
## 4      1     292      5 838983421   3.418414        16075
## 5      1     316      5 838983392   3.349353        18925
## 6      1     329      5 838983392   3.336271        16167
##                     title                          genres release_year rating_date
## 1              Boomerang                 Comedy|Romance         1992  1996-08-02
## 2               Net, The         Action|Crime|Thriller         1995  1996-08-02
## 3          Dumb & Dumber                         Comedy         1994  1996-08-02
## 4               Outbreak  Action|Drama|Sci-Fi|Thriller         1995  1996-08-02
## 5               Stargate       Action|Adventure|Sci-Fi         1994  1996-08-02
## 6 Star Trek: Generations Action|Adventure|Drama|Sci-Fi         1994  1996-08-02
##   age_of_movie rating_year
## 1           28        1996
## 2           25        1996
## 3           26        1996
## 4           25        1996
## 5           26        1996
## 6           26        1996
```

A summary of the subset confirms that there are no missing values.

```
summary(movielens)
```

```
##      userId         movieId          rating        timestamp
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08
##  1st Qu.:18123   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08
##  Median :35740   Median : 1834   Median :4.000   Median :1.035e+09
##  Mean   :35870   Mean   : 4120   Mean   :3.512   Mean   :1.033e+09
##  3rd Qu.:53608   3rd Qu.: 3624   3rd Qu.:4.000   3rd Qu.:1.127e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
##    avg_rating     rating_count      title              genres
##  Min.   :0.500   Min.   :    1   Length:10000054    Length:10000054
##  1st Qu.:3.219   1st Qu.: 1814   Class :character   Class :character
##  Median :3.591   Median : 4699   Mode  :character   Mode  :character
##  Mean   :3.512   Mean   : 7542
##  3rd Qu.:3.875   3rd Qu.:10928
##  Max.   :5.000   Max.   :34864
##   release_year   rating_date                    age_of_movie     rating_year
##  Min.   :1915   Min.   :1995-01-09 00:00:00   Min.   : 12.00   Min.   :1995
##  1st Qu.:1987   1st Qu.:2000-01-02 00:00:00   1st Qu.: 22.00   1st Qu.:2000
##  Median :1994   Median :2002-10-25 00:00:00   Median : 26.00   Median :2002
##  Mean   :1990   Mean   :2002-09-21 12:05:48   Mean   : 29.78   Mean   :2002
##  3rd Qu.:1998   3rd Qu.:2005-09-15 00:00:00   3rd Qu.: 33.00   3rd Qu.:2005
##  Max.   :2008   Max.   :2009-01-05 00:00:00   Max.   :105.00   Max.   :2009
```

```
data_summary <- movielens %>% summarize(n_users = n_distinct(userId), n_movies = n_dist
```

The total of unique movies and users in the movielens subset is **69,878** unique users and **10,677** different movies:**

```
print(data_summary)
```

```
##   n_users n_movies
## 1   69878    10677
```

If every user rated every movie, we would have **746,087,406** rows; however, we only have this many rows:

```
nrow(movielens)
```

```
## [1] 10000054
```

Therefor, the database is fairly **sparsely populated**. I will visualize this later in the document.

Methods and Analysis

Data Insights, Cleaning & Visualizations

Sparsity

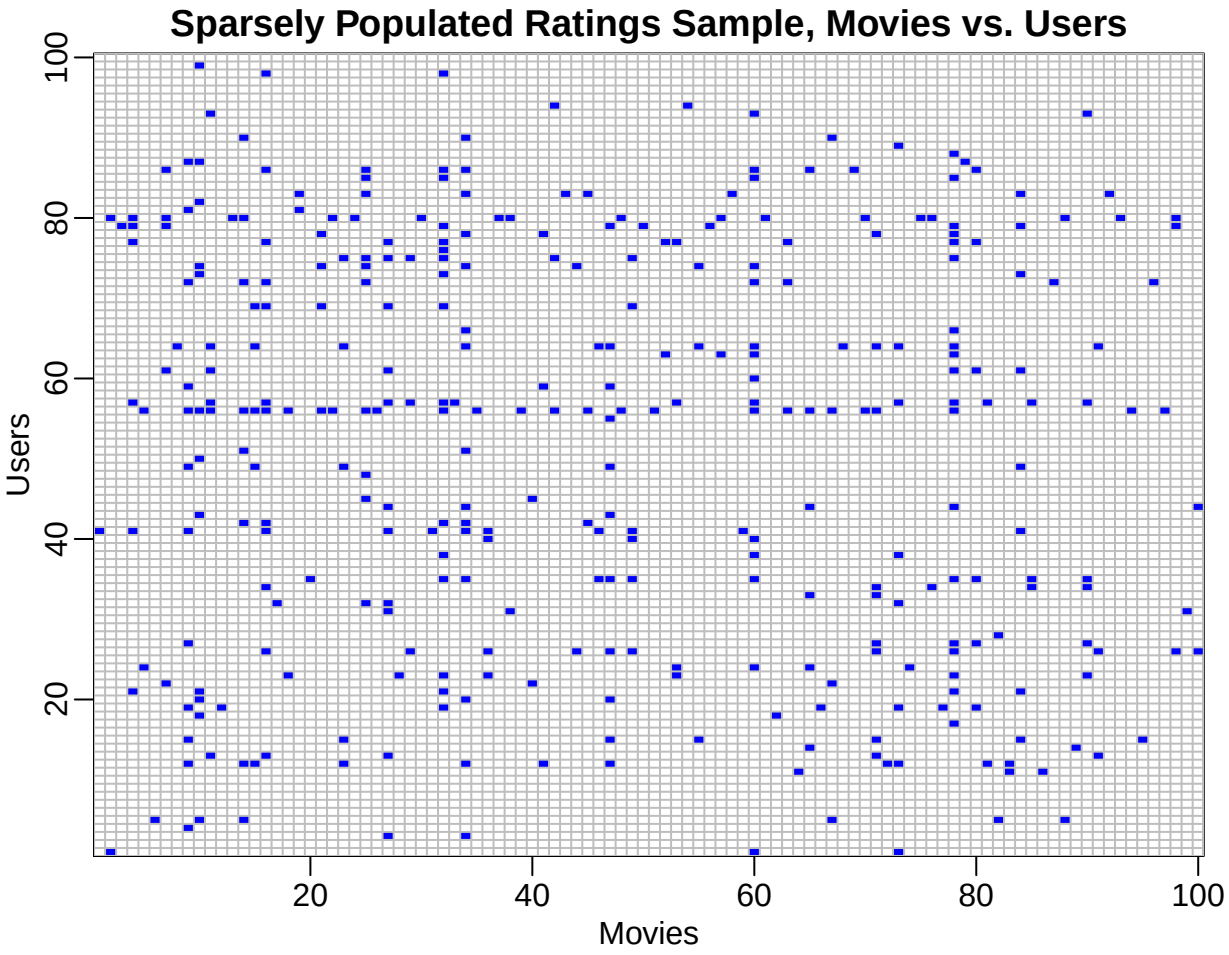
```
percent_populated <- (nrow(movielens)/
                        (data_summary$n_users* data_summary$n_movies)) * 100
print(percent_populated)
```

```
## [1] 1.340333
```

As noted, if every user rated every movie, we would have approximately **746,087,406** entries in the dataset; however, as shown above, we only have **10,000,054** ratings. Apparently, the data set is actually less then **1.34%** populated, which we can see from the calculation above.

Visually, we can show the sparseness on a small sample of the data, with the following image graph:

```
users <- sample(unique(movielens$userId), 100)
rafalib::mypar()
movielens %>% filter(userId %in% users) %>%
  select(userId, movieId, rating) %>%
  mutate(rating = 1) %>%
  spread(movieId, rating) %>% select(sample(ncol(.), 100)) %>%
  as.matrix() %>% t(.) %>%
  image(1:100, 1:100,. ,
        xlab="Movies", ylab="Users", col="blue",
        main="Sparsely Populated Ratings Sample, Movies vs. Users")
abline(h=0:100+0.5, v=0:100+0.5, col = "grey")
```


Sparsely Populated Ratings Sample, Movies vs. Users
Users
Movies
20
40
60
80
100

Users

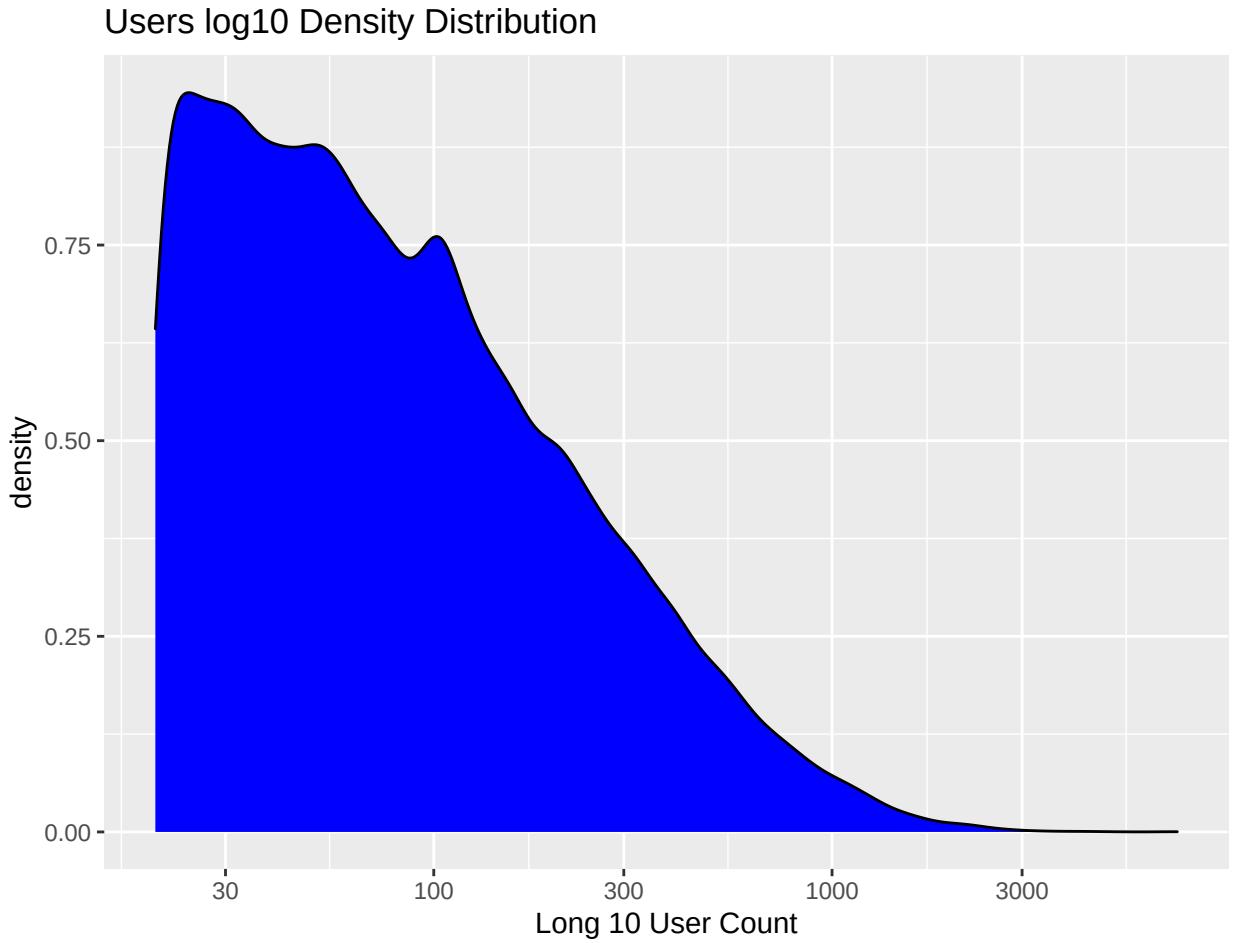
Summarizing the users in the system, we have users with **a minimum of 20 reviews**; however, the maximum is **> 7000 reviews** per user.

```
users <- movielens %>% group_by(userId) %>% summarize(n = n(), avg_user_rating = mean(r

summary(users)
```

```
##      userId            n          avg_user_rating
##  Min.   :    1   Min.   :  20.0   Min.   :0.500
##  1st Qu.:17943   1st Qu.:  35.0   1st Qu.:3.360
##  Median :35798   Median :  69.0   Median :3.635
##  Mean   :35782   Mean   : 143.1   Mean   :3.614
##  3rd Qu.:53620   3rd Qu.: 156.0   3rd Qu.:3.900
##  Max.   :71567   Max.   :7359.0   Max.   :5.000
```

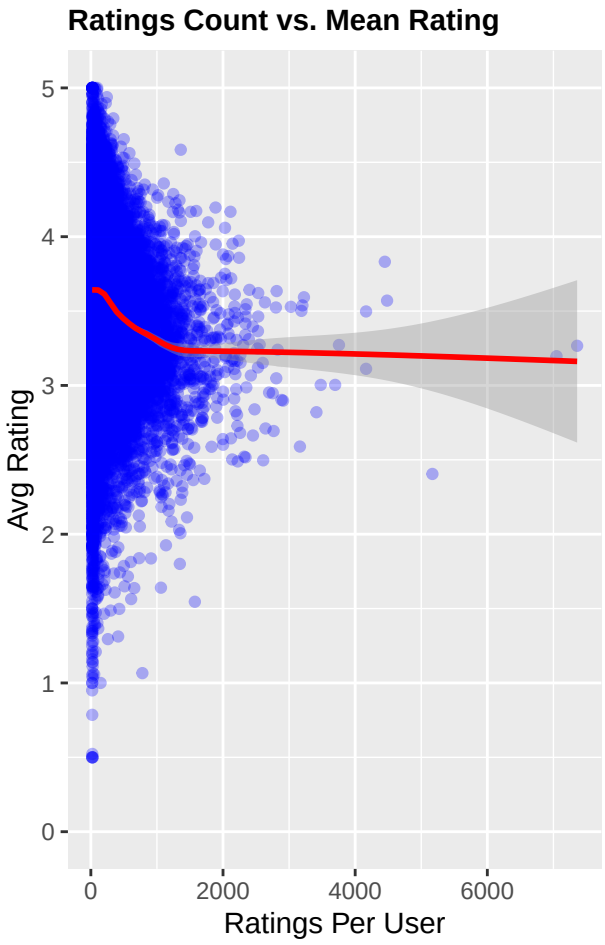
Viewing the ratings' density distribution, **it is obvious that most of the ratings come from a small percentage of users**:

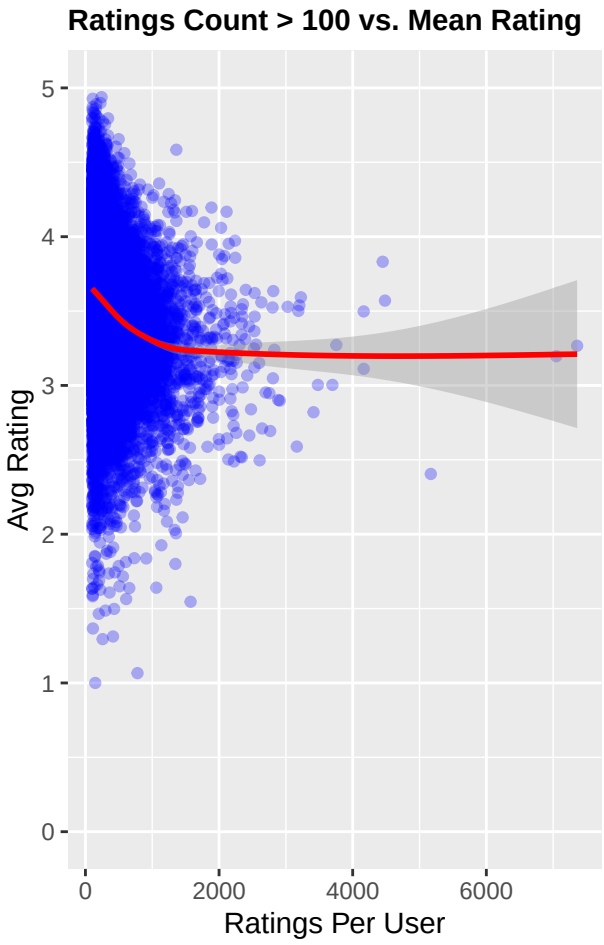
```
users %>%
  ggplot(aes(n)) +
  geom_density(fill = "blue", color="black") +
  scale_x_log10() + xlab("Long 10 User Count") +
  ggtitle("Users log10 Density Distribution")
```


Users log10 Density Distribution
density
0.75
0.50
0.25
0.00
30
100
300
1000
3000
Long 10 User Count

If we examine the trends in the amount of ratings a user submits, versus the average rating, per user, we can clearly see that users with smaller amounts of ratings are much more likely to give extreme ratings on either side of the scale - be either very cranky (and give a rating of .5) or extra optimistic (and give a rating of 5). In the plots below I show the difference in the variability of user's average rating trends vs. the amount of ratings they have submitted. **By omitting users with 100 reviews or less**, we limit the crankiness of the users slightly.

```
g1 <- users %>%
  ggplot(aes(n, avg_user_rating)) +
  geom_point(color="blue", alpha=.3) +
  xlab("Ratings Per User") +ylab("Avg Rating") + geom_smooth(color='red') +
  ggtitle("Ratings Count vs. Mean Rating") + scale_y_continuous(limits = c(0, 5)) +
  theme(legend.position = "none", plot.title = element_text(size = 11, face="bold"))

g2 <- users %>% filter(n > 100) %>%
  ggplot(aes(n, avg_user_rating)) +
  geom_point(color="blue", alpha=.3) +
  xlab("Ratings Per User") +ylab("Avg Rating") + geom_smooth(color='red') +
  ggtitle("Ratings Count > 100 vs. Mean Rating")+ scale_y_continuous(limits = c(0, 5)) +
  theme(legend.position = "none", plot.title = element_text(size = 11, face="bold"))

plot_grid(g1, g2)
```



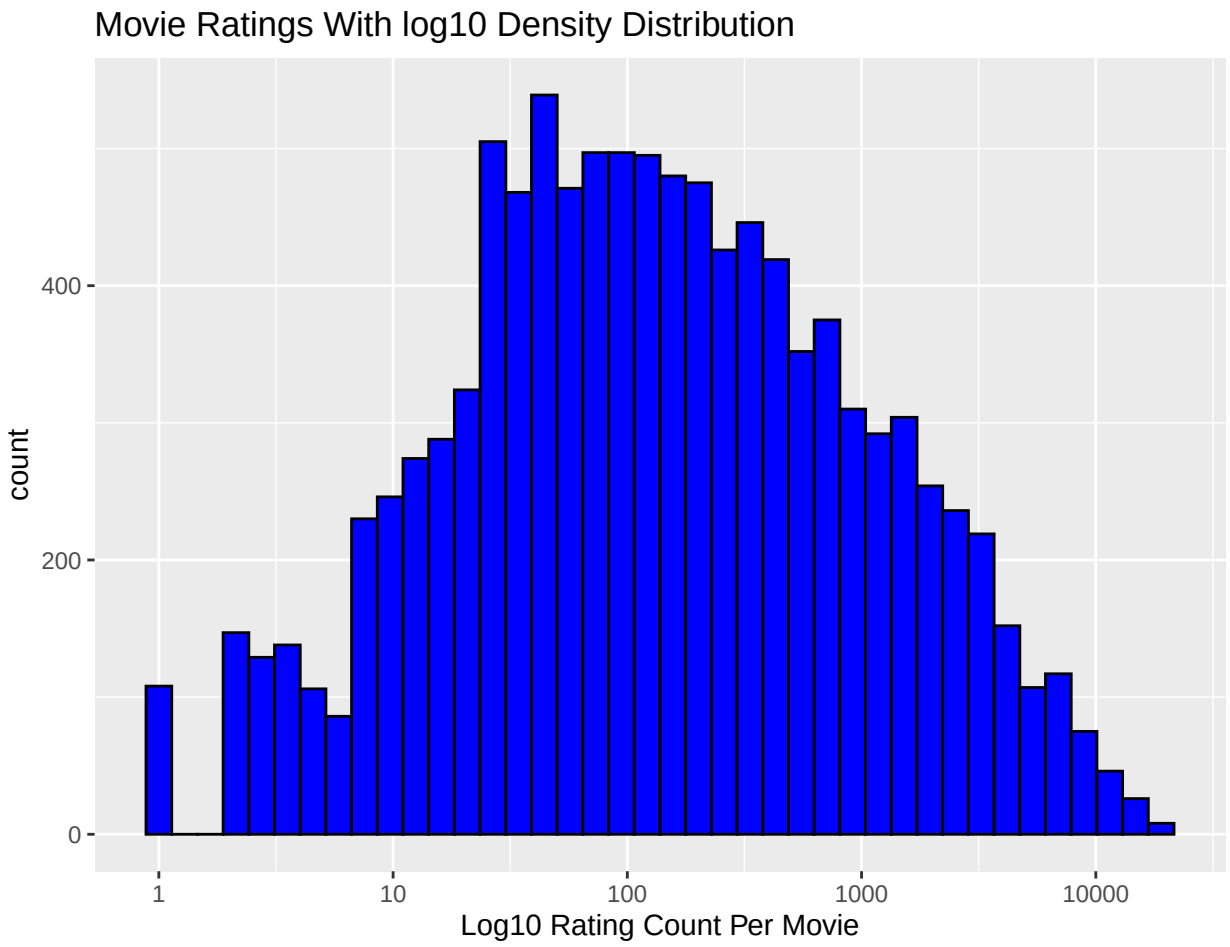



```
bigger_users <- users %>% filter(n > 100)
movielens <- subset(movielens, (userId %in% bigger_users$userId))
```

Movies

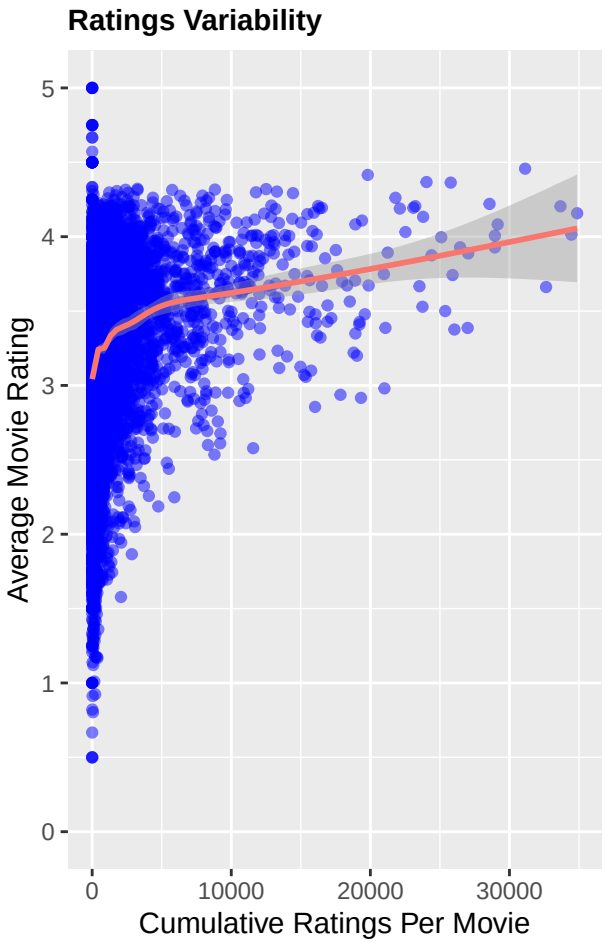
We can also show that **each movie is not rated with the same frequency**; That is, the popular movies will be seen by many more people and will therefor be rated by many more people. Meanwhile, there are also many independent films that will only be seen by a few people, and therefor will only be rated by a small percentage of those.

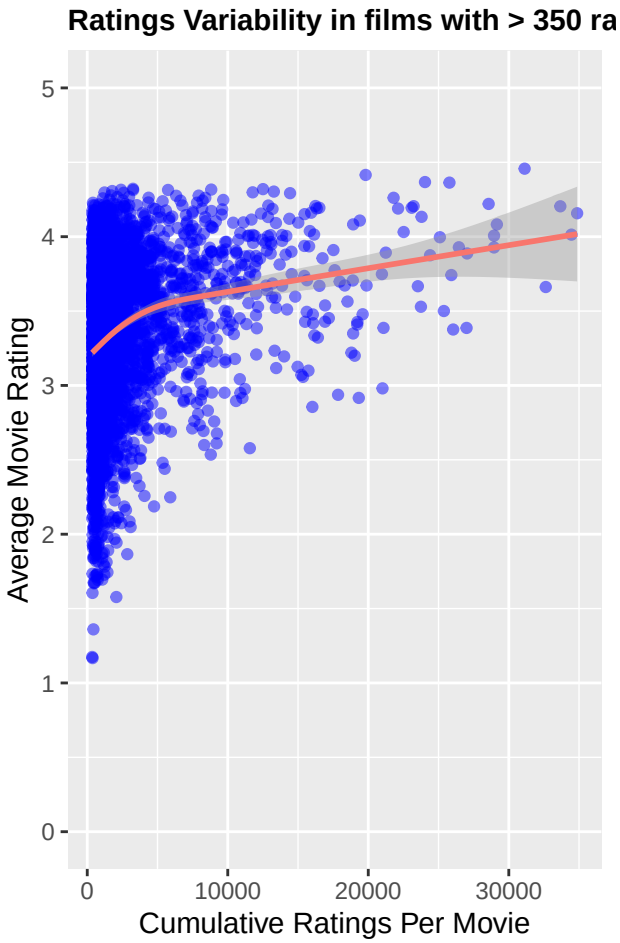
```
movielens %>%  group_by(movieId) %>% summarise(n = n()) %>%
  ggplot(aes(n)) +
  geom_histogram(fill = "blue", color="black", bins = 40) +
  scale_x_log10() + xlab("Log10 Rating Count Per Movie") +
  ggtitle("Movie Ratings With log10 Density Distribution")
```



The **users** data came with a specifications that **only users with 20 or more reviews are included**; however, no similar filtering was done for movies. If we examine the data, we can see that the variability of average film ratings is very high when the number of ratings is low, and it decreases steadily as the rating counts get into the hundreds and then thousands. This trend makes sense if we consider that an obscure film will only be seen by a small audience that may be biased for or against it, making for an an unusual amount of outlier ratings. As the film is seen by more people, it will get rated by more people, and the variability of those ratings will change as well. If we remove films with **< then 350 reviews**, this narrows the variability somewhat. The plots below will show what I mean:

```
g1 <- rating_avgs %>% ggplot(aes(rating_count, avg_rating)) + geom_point(color='blue', a
  geom_smooth(aes(color='red')) + xlab("Cumulative Ratings Per Movie") +
  ylab("Average Movie Rating") + ggtitle("Ratings Variability") +
  scale_y_continuous(limits = c(0, 5)) +
  theme(legend.position = "none", plot.title = element_text(size = 11, face="bold"))

g2 <- rating_avgs %>% filter(rating_count > 350) %>%
  ggplot(aes(rating_count, avg_rating)) + geom_point(color='blue', alpha=.5) +
  geom_smooth(aes(color='red')) + xlab("Cumulative Ratings Per Movie") +
  ylab("Average Movie Rating") + ggtitle("Ratings Variability") +
  ggtitle("Ratings Variability in films with > 350 ratings") +
  scale_y_continuous(limits = c(0, 5))+
  theme(legend.position = "none", plot.title = element_text(size = 11, face="bold"))

plot_grid(g1, g2)
```



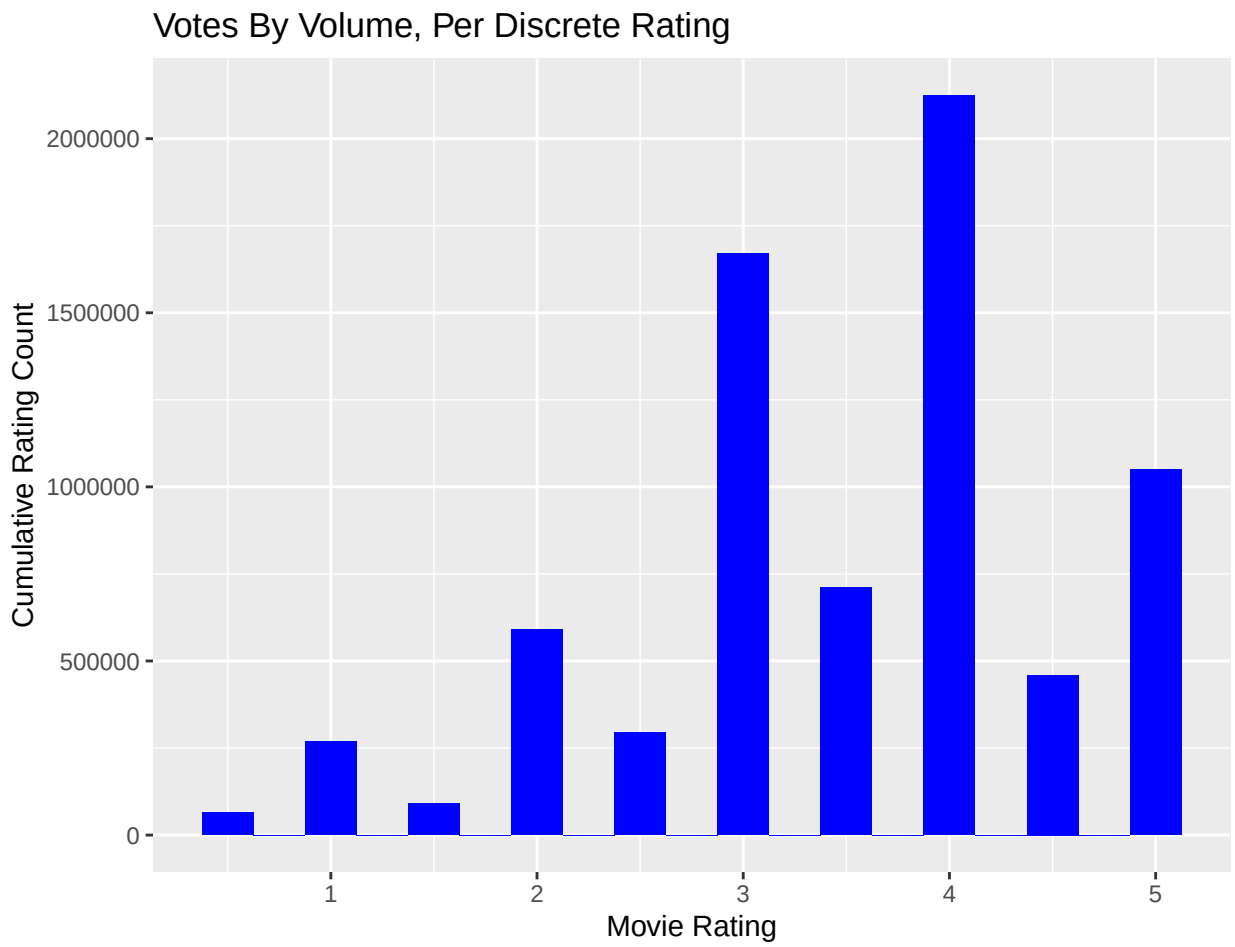


The data shows that by removing the films with fewer reviews, we remove most of the extreme outliers of the **0.5** AND **5** average movie ratings. In general, the data also shows that movies with fewer ratings have a much higher variability in average movie ratings. I believe that movies that have only a few reviews are outliers, and so should be excluded from the training data. Therefor, below, I include only the films with **more then 350 reviews**.

```
bigger_films <- movielens %>% group_by(movieId) %>% summarise(n = n()) %>% filter(n > 3
movielens <- subset(movielens, (movieId %in% bigger_films$movieId))
```

Ratings

```
movielens %>%
  ggplot(aes(rating)) +  xlab("Movie Rating") + ylab("Cumulative Rating Count") +
  geom_histogram(binwidth = 0.25, fill = "blue") +
  scale_y_continuous(breaks = c(seq(0, 3000000, 500000))) +
  ggtitle("Votes By Volume, Per Discrete Rating")
```

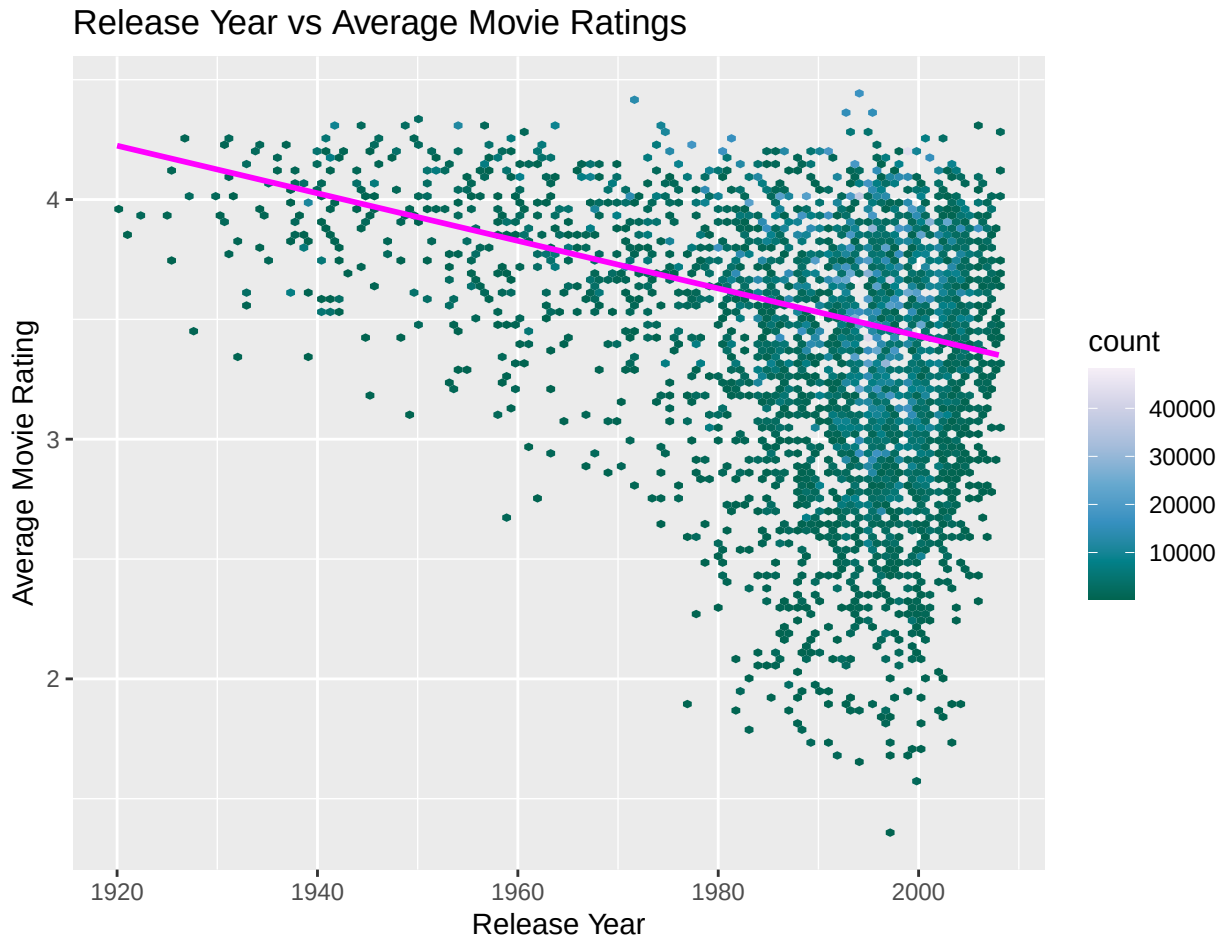


From the data and the bar graph of the ratings, we can make one these conclusions:

1. In terms of ratings, the values are on a **scale of 0.5-5**, in increments of 0.5.
2. There are 10 discrete options, and the ratings are **not** continuous.
3. Full-grade ratings are much more common then the half-grade ratings.
4. **4** is the most common rating.

Plotting release year vs. avg movie ratings - it seems movies are either getting worse with time, or the reviewers are getting pickier.

```
movielens %>%
  ggplot(aes(release_year, avg_rating)) +
  stat_bin_hex(bins = 100) +
  scale_fill_distiller(palette = "PuBuGn") +
  stat_smooth(method = "lm", color = "magenta", size = 1) +
  ylab("Average Movie Rating") + xlab("Release Year") +
  ggtitle("Release Year vs Average Movie Ratings")
```



Genres

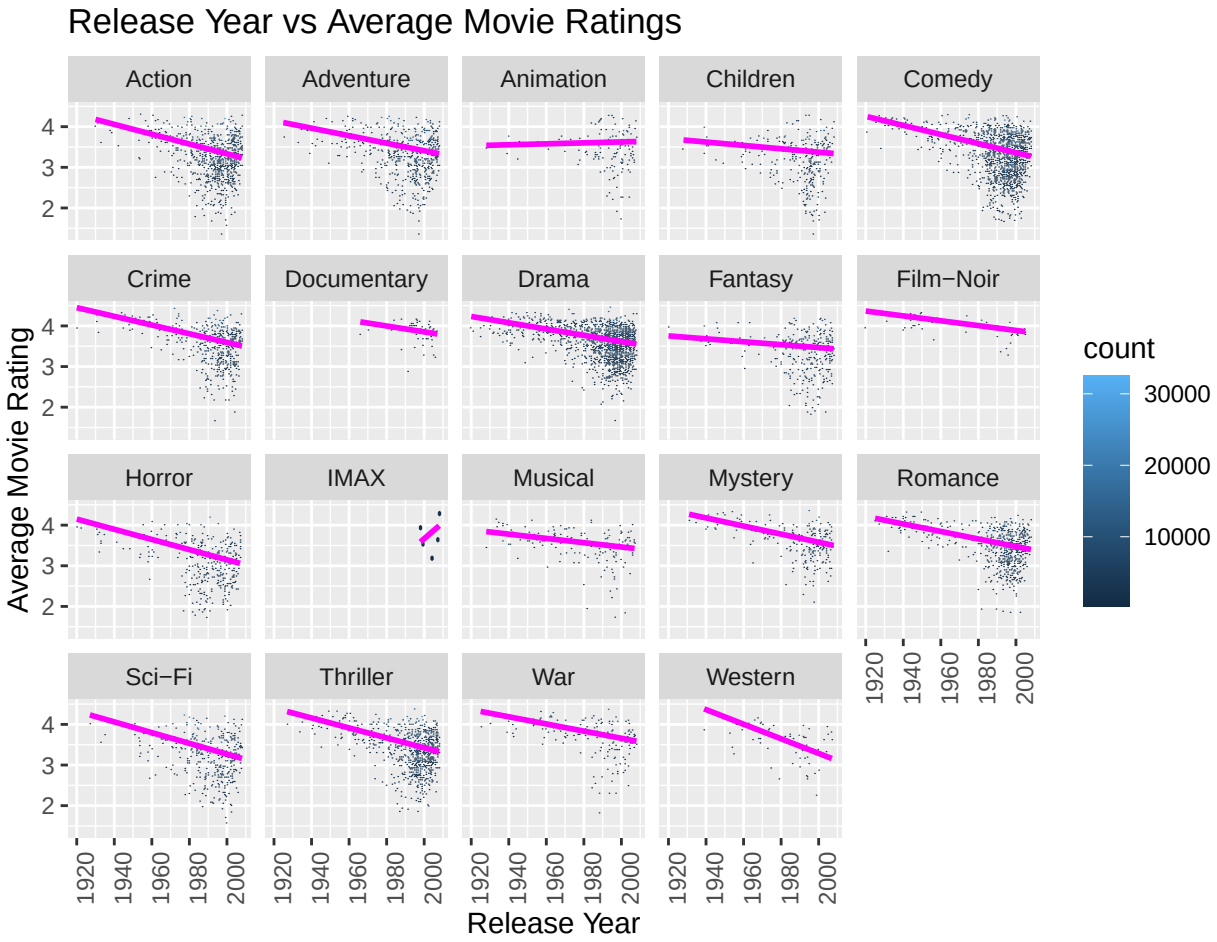
Next in the data wrangling process, I will separate the genres from the combined, "|"-delimited values into separate genres, one entry per rating, per movie, per genre. We can now view the unique genres available in the system:

```
movielens <- movielens %>% separate_rows(genres, sep = "\\|")
genres <- unique(movielens$genres)
print(genres)
```

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

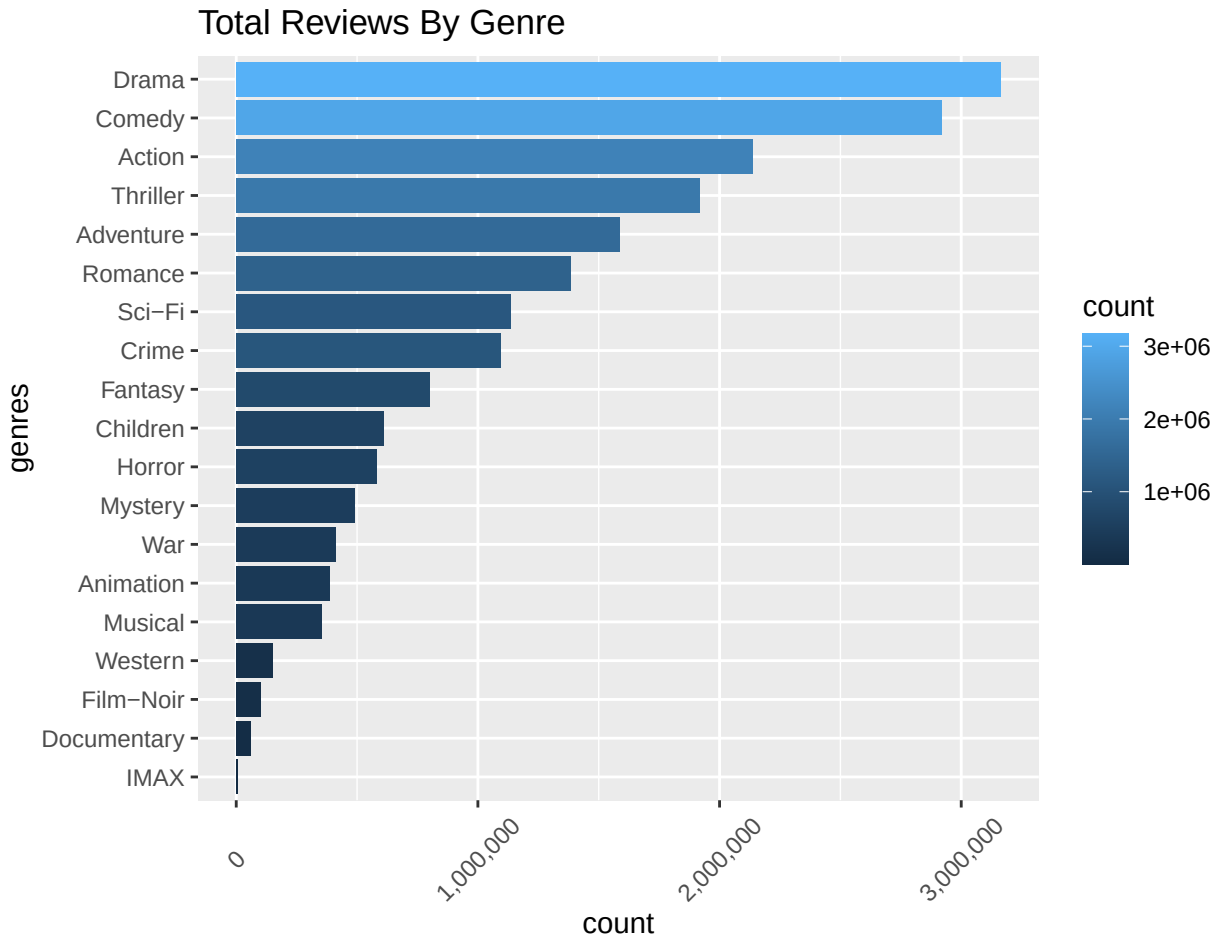
Next, we'll explore how ratings have fared over time by **genre**. Based on the visualizations provided here, it is fairly obvious that **most of the average ratings across genres have declined over the years**. The notable exception to this is **IMAX** movies, and in a small way, **Animation**. These trends make sense, as both IMAX and animated films have benefited greatly from technological advancements over the years.

```
movielens %>% na.omit() %>%
  ggplot(aes(release_year, avg_rating)) + stat_bin_hex(bins = 100) + #scale_fill_distil
  stat_smooth(method = "lm", color = "magenta", size = 1) +
  ylab("Average Movie Rating") + xlab("Release Year") + ggtitle("Release Year vs Average
  facet_wrap(~genres) + theme(axis.text.x = element_text(angle = 90))
```


Release Year vs Average Movie Ratings
Action
Adventure
Animation
Children
Comedy
Crime
Documentary
Drama
Fantasy
Film−Noir
Horror
IMAX
Musical
Mystery
Romance
Sci−Fi
Thriller
War
Western
Average Movie Rating
Release Year
count
30000
20000
10000
1920
1940
1960
1980
2000
2
3
4

The plot below shows the trends of how how many films get rated by genre**.**

```
movies_by_genre <- movielens %>% group_by(genres) %>% summarize(count = n()) %>% arrang

movies_by_genre %>%
  mutate(genres = factor(genres, levels = unique(as.character(genres)))) %>%
  ggplot(aes(x = genres, y= count, fill=count)) + geom_bar( stat = "identity" ) +
  scale_y_continuous(n.breaks = 5, breaks = c(0,  1000000, 2000000, 3000000, 4000000), l
  coord_flip() +
  ggtitle("Total Reviews By Genre") + theme(axis.text.x = element_text(angle = 45, vjus
```



Do people tend to review certain genres more then others?

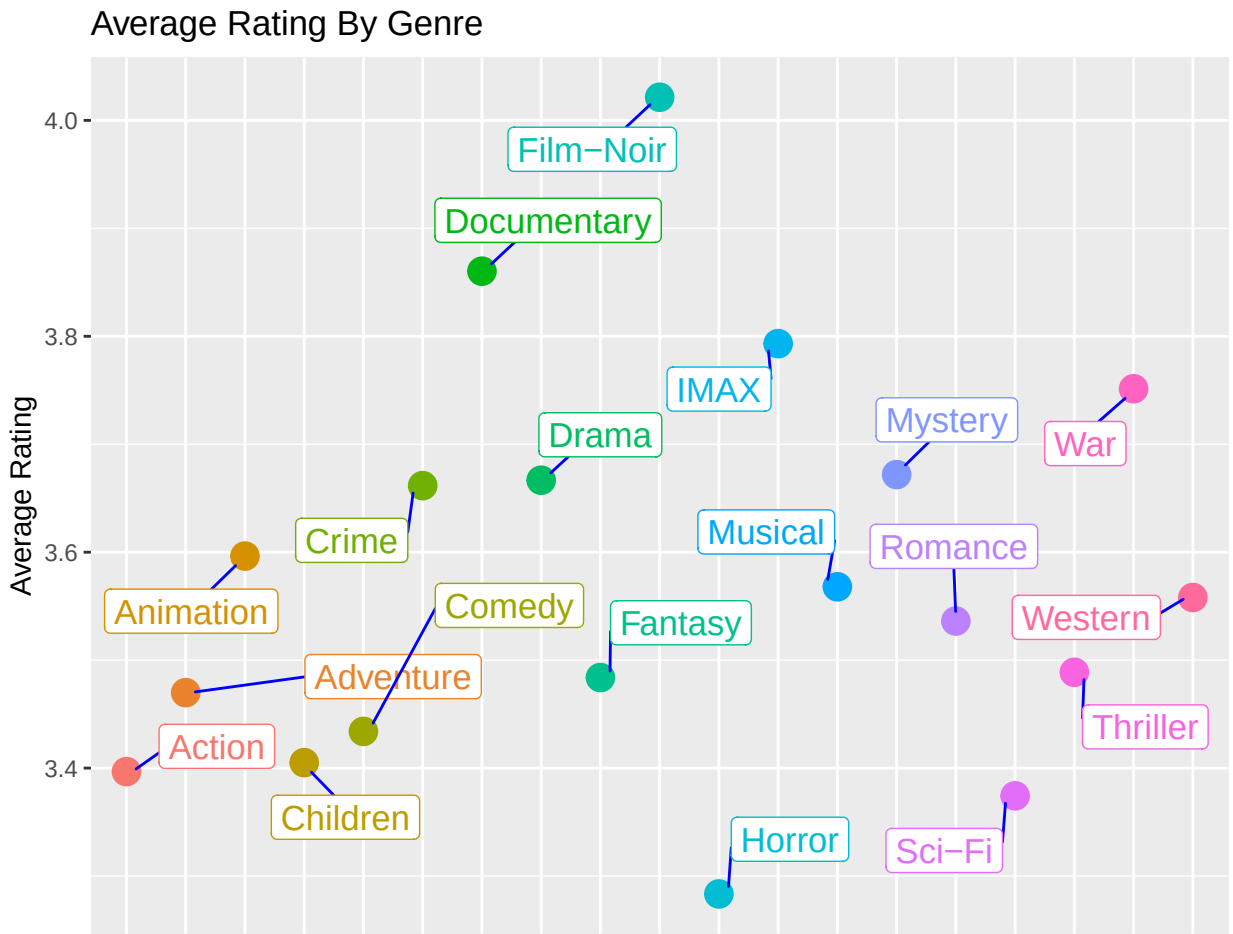
The answer seems to be a resounding **YES**!

Dramas seems to have the most reviews, while **IMAX** has by far the least; this pattern makes sense, since only a small percentage of movies get released in IMAX (although the ones that are are super popular, and will thus get more reviews.)

Ratings By Genre

In general, the average rating by genre can be shown on this plot; **Film Noir** seems to have the highest average rating, while **Horror** films have the lowest.

```
ratings_summary_by_genre <- movielens %>% group_by(genres) %>% summarise(avg_rating = m
ratings_summary_by_genre %>% ggplot(aes(genres, avg_rating, size = 3, col=genres)) +
  geom_point() +   theme(axis.title.x=element_blank(), legend.title = element_blank(),
                         axis.text.x=element_blank(), legend.key= element_blank(),
                         axis.ticks.x=element_blank(), legend.text = element_blank(), l
  geom_label_repel(aes(label = genres),
                   box.padding   = 0.35,
                   point.padding = 0.5,
                   segment.color = "blue") + ggtitle("Average Rating By Genre") + ylab("
```



Rating Models

1. Average movie rating model

```
# Validation set will be 10% of MovieLens data
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,]

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

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

#remove all the objects from memory
rm(dl, ratings, movies, test_index, temp, removed, movielens )

#RMSE function imlpementation
```

Earlier in the document I described the logic behind the RMSE function. Here I am enclosing it's implementation in R.

```
RMSE <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

```
#splitting the edx data set into testing and training , making sure to exclude the val
test_index <- createDataPartition(y = edx$rating, times = 1, p = 0.15, list = FALSE)
train_set <- edx[-test_index,]
test_set <- edx[test_index,]
```

To start out, we will examine the **mean rating**, and use that as the benchmark for evaluations, creating a model that assumes the same rating for all movies and users with all the differences explained by random variation, $\varepsilon_{u,i}$. The formula used for this is:

$$Y_{u,i} = \mu + \varepsilon_{u,i}$$

$Y_{u,i}$ is the predicted rating per user, per movie and μ is the estimate that minimizes the root mean squared error — is the average rating of all movies across all users. The code below calculates μ on the training set of the data, and then uses the RMSE to predict the error of ratings of the test set.

```
mu_hat <- mean(train_set$rating)
print(mu_hat)
```

```
## [1] 3.513859
```

```
naive_rmse <- RMSE(test_set$rating, mu_hat)
rmse_results <- tibble(method = "Just the average", RMSE = naive_rmse)
```

This value will be used with the RMSE function to test the accuracy with most naive version of the algorithm.

```
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.040655

As I continue with the project, I will add the results of the RMSE of every model I test to the **rmse_results** variable, and print out the table as I have above - so that the data can be reviewed as the project progresses.
Taking the average of the existing ratings is a very naive predictor. We can see that the RMSE is > 1.0, which means this isn't a great prediction - as expected. In the rest of the project I will examine several better approaches.

2. Movie effect model

From the data analysis performed earlier, we've surmised that some movies will be rated higher then others - making the naive RMSE above clearly inferior to a more detailed approach. I will now build a model with accounting for bias per movie, b_i (since every movie will have some inherent bias, and some movies are clearly better (or worse) then others).

$$Y_{u,i} = \mu + b_i + \varepsilon_{u,i}$$

b_i is the average of $Y_{u,i}$, minus the overall mean for each movie i. To calculate the movie bias, we transform the formula to :

$$b_i = Y_{u,i} - \mu - \varepsilon_{u,i}$$

And run the RMSE algorithm on the test set again. The predictions improve significantly when we use this model, as shown below.

```r
mu <- mean(train_set$rating)
movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu)) #here we're calculating the b_i, avg bias per mov

predicted_ratings <- mu + test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  pull(b_i)

model_1_rmse <- RMSE(test_set$rating, predicted_ratings)

#get the RMSE results from from the b_i predective model
tmp_rmse_results <- tibble(method = "Movie Effect Model", RMSE = model_1_rmse)

#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, tmp_rmse_results)
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068

Adding a single **movie effect** to the model not only significantly reduces the RMSE to **0.9266068**, but brings it below 1.0, which is a great first step. However, the number is still not low enough, so we continue on.

3. Movie and user effect model

We compute the user bias b_u per user, for those that have rated **over 100 movies**. Some users are cranky, and others are optimistic - so the Effect of the user's crankiness can change the ratings in either direction, positive or negative. The formula for the predicted rating would then be, building on what we have already seen in models 1 and 2 above:

$$Y_{u,i} = \mu + b_i + b_u + \epsilon_{u,i}$$

We compute an approximation by computing μ and b_i, and estimating b_u, as the average of

$$b_u = Y_{u,i} - \mu - b_i - \epsilon_{u,i}$$

```
#Now we will add in the user bias, b_u
user_avgs <- train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i)) #here, we're calculating the bias per movie,


predicted_ratings <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  pull(pred)

model_2_rmse <- RMSE(predicted_ratings, test_set$rating)
```

The RMSE on this third model we're evaluating comes out to **0.8440959**, and is already below the required threshold level of **0.8649**. The results table is now updated with the latest RMSE:

```
tmp_rmse_results <- tibble(method = "User Effect + Movie Effect Model", RMSE = model_2_r

#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, tmp_rmse_results)
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068
3	User Effect + Movie Effect Model	0.8440959

4. Regularized Models

So far we have been computing estimates with certain levels of uncertainty, but we have not been taking into accounts that problems may exist in the data where there is not enough information. There is always a risk of over-fitting the data, and to reduce this risk, we introduce the concept of **regularization**. Regularization will allow us to use a tuning parameter, λ, that when included in the calculation will minimize the mean squared error. This will reduce the b_i and b_u in case of small number of ratings. (Remember that we're using movies with cumulative ratings count of over 100, but this is still not that many, all things considered.)

Since I am attempting to narrow down a λ value quite frequently during this project over various models, I have created a function that will be used to create, calculate, plot, and narrow down the lambda range for each model I attempt to train.

The function takes in parameters for the start and end of the range of the λ, an increment to step through the lambda values, as well as another function that is passed in to do the calculations that vary between the models. The function then recursively calls itself only **once** to get and even finer-grained value of λ, with a smaller, narrower set of lambdas once a rough range has been established in the first run-through. The function returns a minimum lambda for the model in question, and can then be used to calculate the RMSE for the model. The testing set is very large so if we want to evaluate lambdas, we first run the data set with broad intervals, and then zoom in on the best performing section - this is the point of the recursive call of the function below.

The function will also plot both sets of attempted λ's - first, the wide range that is set by the first call of the function, and then, the narrowed range with much more granular steps. Since this function will be called by every regularized model, we will see the λ ranges plotted appropriately.

4.1 Movie Effect Regularized Model

```
#function that will create the loose labmdas, and then come up with a more
#detailed lambda evaluation when we have an idea where the lowest value is
find_generic_lambda <- function(seq_start, seq_end, seq_increment, FUN,
    detailed_flag = FALSE, training_set, testing_set, plot_title="")
{
  lambdas <- seq(seq_start, seq_end, seq_increment)
  RMSE <- sapply(lambdas, FUN)
  #find the smallest lamdba
  print(qplot(lambdas, RMSE, main=plot_title))
  #saving the first round lambda
  min_lambda_first_try <- lambdas[which.min(RMSE)]

  if (detailed_flag) #this flag signifies where the function
    #will call itself recursively
  {
    #if this is the first iteration of the function, continue with taking
    #a slightly lower % and silghtly higher % lambda value range to iterate
```

```
    #through new lambdas that are much more granuluar, with increments at
    #10% of what they were previously.
    new_lambda_range = (seq_end - seq_start)/40
    min_lambda_first_try <- find_generic_lambda(seq_start =
            min_lambda_first_try - new_lambda_range,
            seq_end = min_lambda_first_try + new_lambda_range,
           seq_increment = seq_increment/10, FUN, detailed_flag = FALSE,
           training_set = training_set, testing_set = testing_set,
           plot_title = paste("Narrowed Range: ", plot_title))
  }
  return (min_lambda_first_try)
}
```

The first model to use this function will be the **Regularized Movie Effect Model** - essentially we are re-using the movie-effect model from above with regularization. The general idea of penalized regression is to control the total variability of the movie effects; Specifically, instead of minimizing the least squares equation, we minimize an equation that adds a penalty:

$$\frac{1}{N}\sum_{u,i}\left(y_{u,i}-\mu-b_i\right)^2+\lambda\sum_i b_i^2$$

The first term of this equation is the least squares estimate we've been working with throughout; the second term is basically a penalty that gets larger when many b_i are large. Using calculus, we can re-work this equation to show that the values of b_i that minimize the equation are:

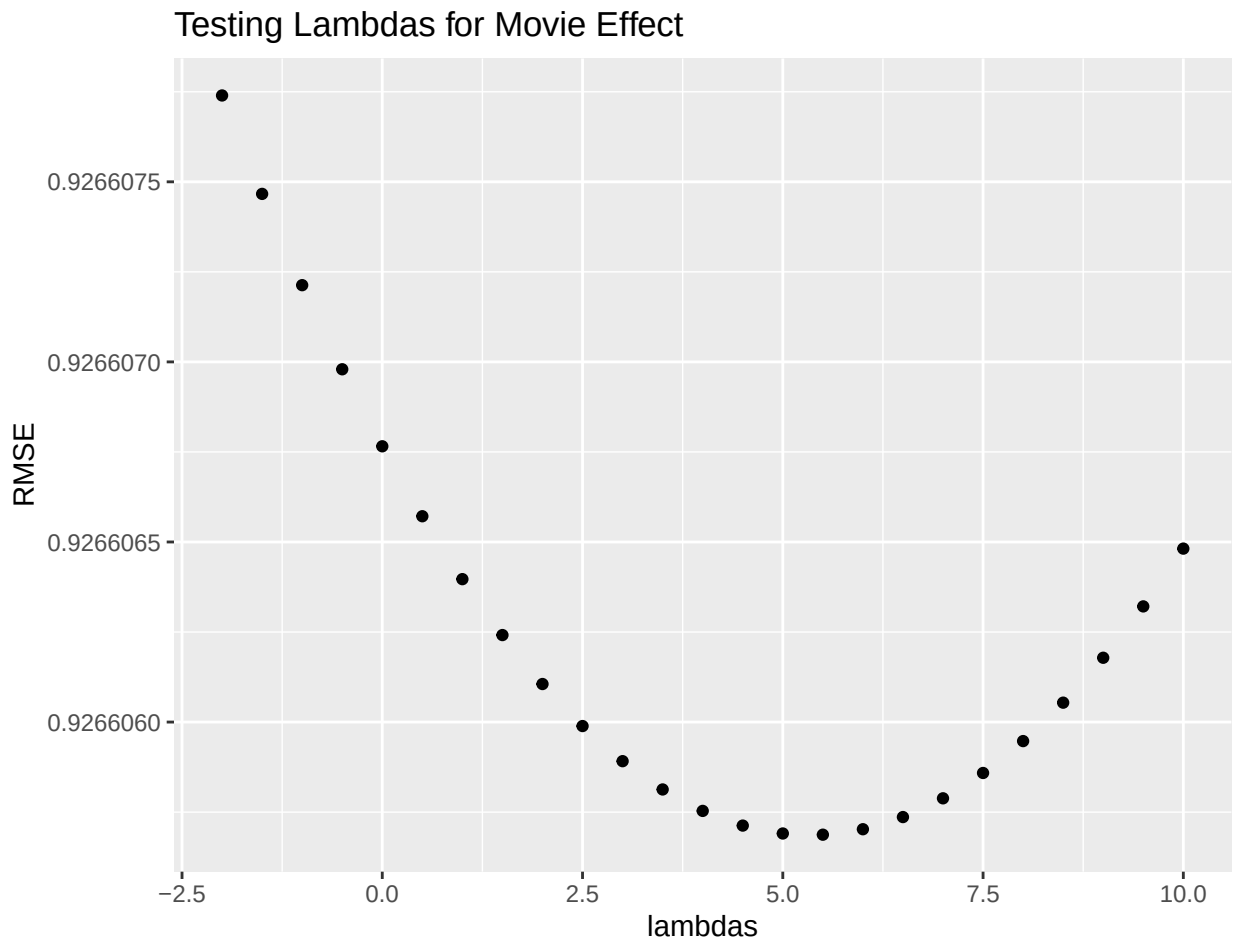
$$\hat{b}_i(\lambda)=\frac{1}{\lambda+n_i}\sum_{u=1}^{n_i}\left(Y_{u,i}-\hat{\mu}\right)$$

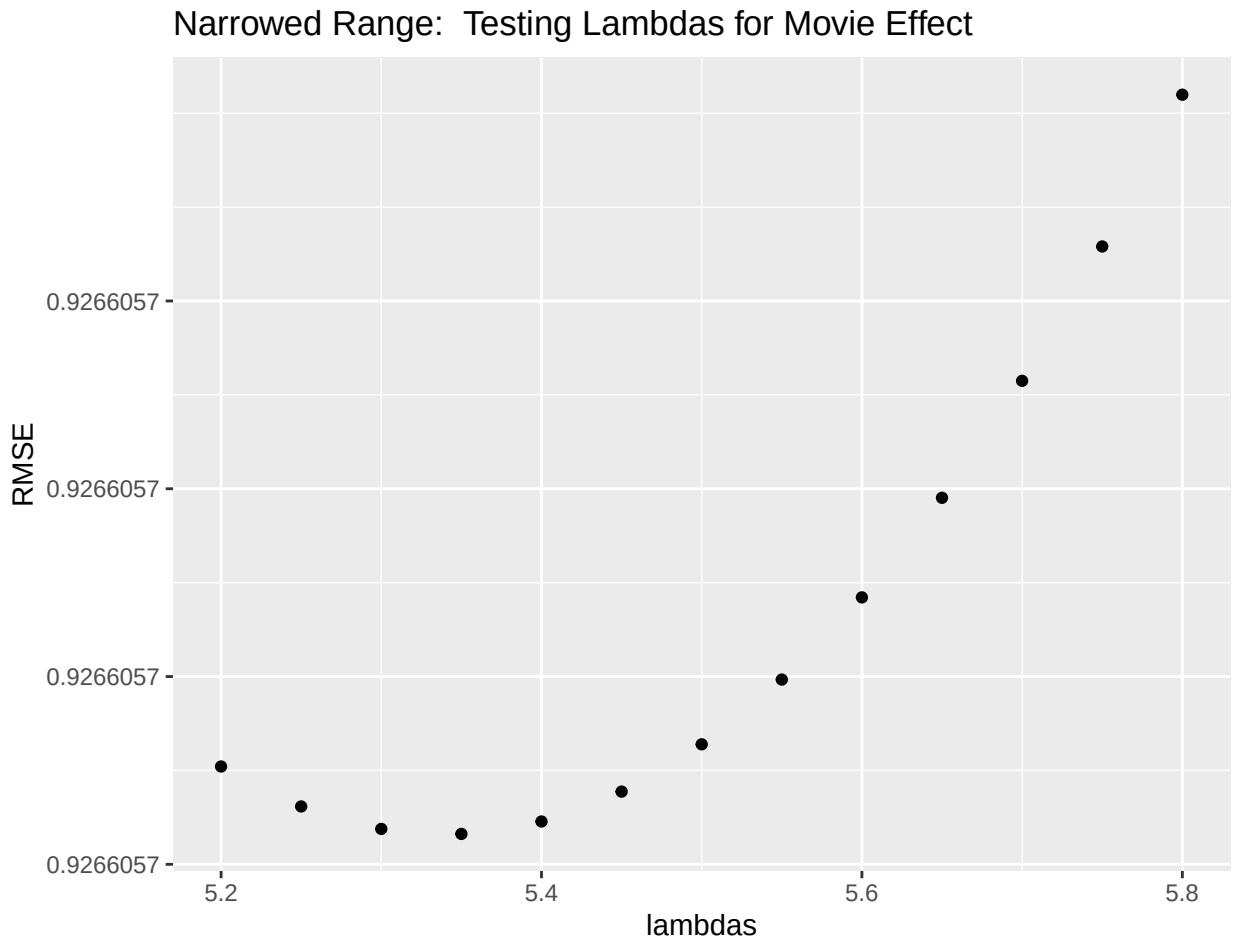
where n_i is the number of ratings made for movie i. When the sample size is large, the penalty λ is so low that it's basically ignored; however, when the sample size is small, the estimate $\hat{b}_i(\lambda)$ diminishes to 0.

λ is a tuning parameter, and we use cross-validation to choose the correct lambda for each case we decide to plug in regularization into. This, in essence, is what the function **find_generic_lambda** shown above is all about.

```
regularized_rmse_3 <- function(l, training_set, testing_set)
{
  #print(l)
  mu <- mean(training_set$rating)
  just_the_sum <- training_set %>%
    group_by(movieId) %>%
    summarize(s = sum(rating - mu), n_i = n())
```

```
    predicted_ratings <- testing_set %>%
    left_join(just_the_sum, by='movieId') %>%
    mutate(b_i = s/(n_i+l)) %>%
    mutate(pred = mu + b_i) %>%
    pull(pred)

    l_rmse <- RMSE(predicted_ratings, testing_set$rating)
    #print(l_rmse)
    return (l_rmse)
}

#testing out the regularization with lamdba -
rmse3_lambda <- find_generic_lambda(seq_start=-2, seq_end=10,
    seq_increment=0.5,
    FUN= function(x) regularized_rmse_3(x, train_set, test_set ),
    detailed_flag = TRUE, training_set=train_set, testing_set=test_set,
    plot_title = "Testing Lambdas for Movie Effect")
```






```
rmse_3 <- regularized_rmse_3(rmse3_lambda, train_set, test_set)
tmp_rmse_results <- tibble(method = "Regularized + Movie Effect", RMSE = rmse_3)
#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, tmp_rmse_results)
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068
3	User Effect + Movie Effect Model	0.8440959
4	Regularized + Movie Effect	0.9266057

The function **regularized_rmse_3** was passed as a parameter into the **find_generic_lambda** function described above, and will produce the RMSE for the regularized movie effect model. It is of note that the result of this is only slightly better then the non-regularized Movie Effect model (model # 2) and significantly worse then the non-regularized model # 3. Moreover, an RMSE of **0.9266057** it no longer fits the criteria of being **< 0.8649**. Therefor, the search for a fitting RMSE

continues.

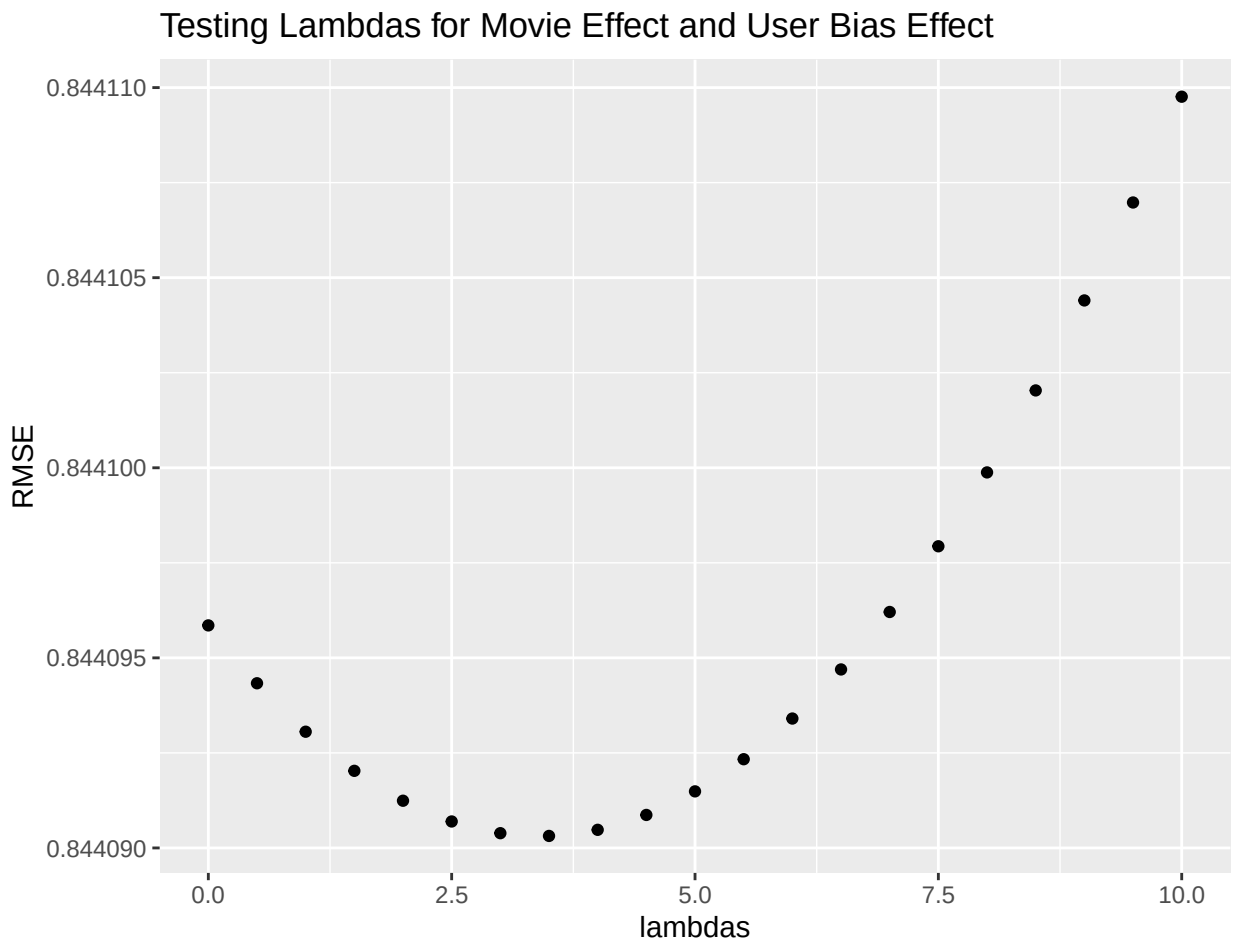
4.2 Movie + User Bias Effect Regularized Model

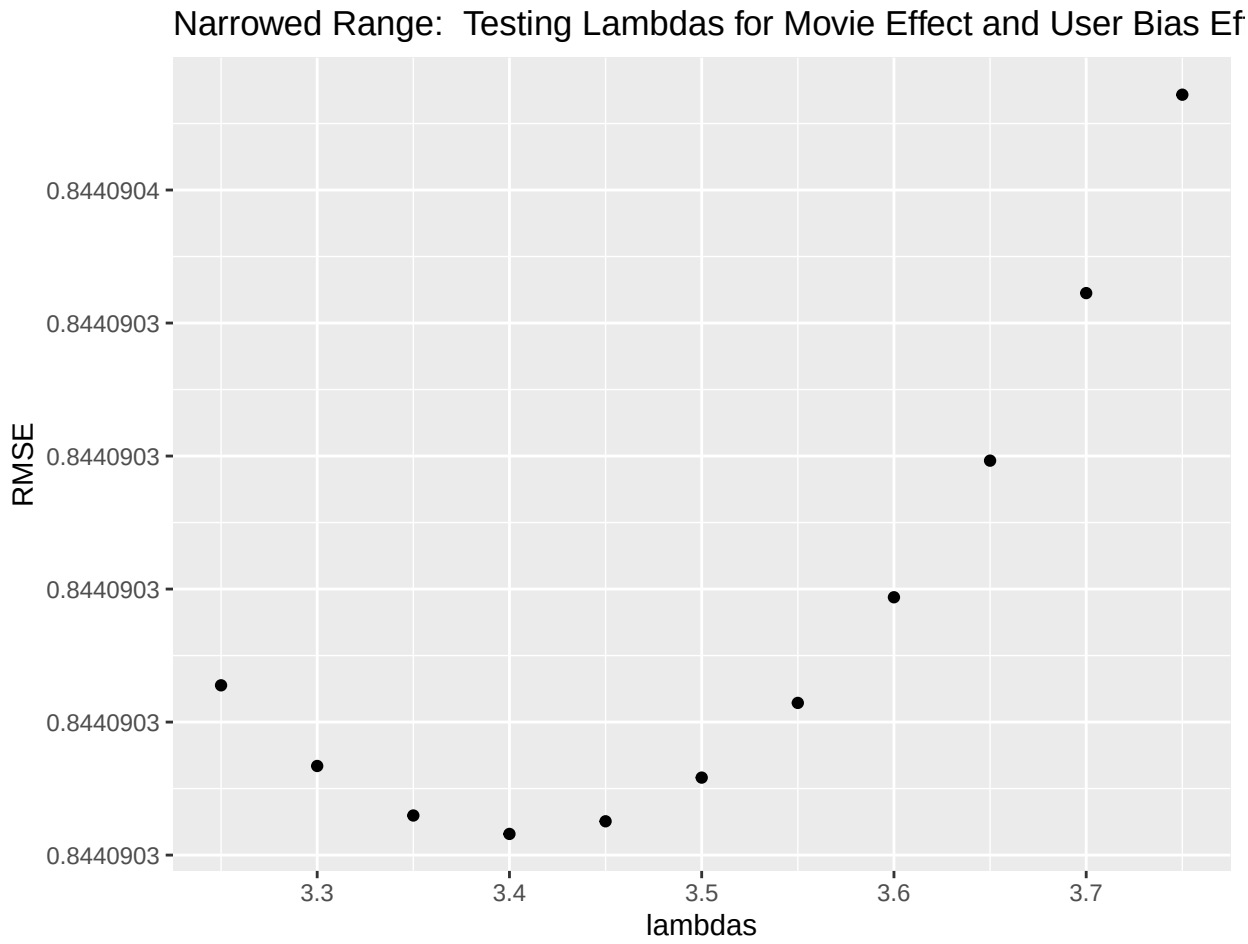
The results from the regularization approach don't seem to be doing any better then the movie Effect model, and significantly worse then the user Effect + movie effect model. We will now try the regularization option with both movie and user bias effect. Therefor, I will next attempt to tune the model more with the both the **Movie Effect** AND the **User Effect** parameter.

This time, the formula we will follow is, with b_u as the user Effect:

$$\frac{1}{N}\sum_{u,i}\left(y_{u,i}-\mu-b_i-b_u\right)^2+\lambda\left(\sum_i b_i^2+\sum_u b_u^2\right)$$

```
regularized_movie_and_user <- function(l, training_set, testing_set)
{

  #print(l)
  mu <- mean(training_set$rating)

  b_i <- training_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))

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

  predicted_ratings <-
    testing_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
    pull(pred)

  return(RMSE(predicted_ratings, test_set$rating))
}

rmse4_lambda <- find_generic_lambda(seq_start=0, seq_end=10,
      seq_increment=0.5,
      FUN= function(x) regularized_movie_and_user(x, training_set=train_set,
                                                  testing_set=test_set ),
      detailed_flag = TRUE, training_set=train_set,
      testing_set=test_set,
      plot_title = "Testing Lambdas for Movie Effect and User Bias Effect")
```


Testing Lambdas for Movie Effect and User Bias Effect
RMSE
0.844110
0.844105
0.844100
0.844095
0.844090
0.0
2.5
5.0
7.5
10.0
lambdas




```
rmse_4 <- regularized_movie_and_user(rmse4_lambda, train_set, test_set)
tmp_rmse_results <- tibble(method = "Regularized Movie + User Effect Model",
                           RMSE = rmse_4)
#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, tmp_rmse_results)
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068
3	User Effect + Movie Effect Model	0.8440959
4	Regularized + Movie Effect	0.9266057
5	Regularized Movie + User Effect Model	0.8440903

It seems like the regularization model with both user and movie effects works very well with the RMSE, and even performed slightly better then the current winner, **the user Effect + movie effect model**. But is an even better approach possible? I will next attempt to add the film's release year

into the mix and examine whether this will help making the predictions more accurate.

4.3 Movie + User + Release Year Effect Regularized Model

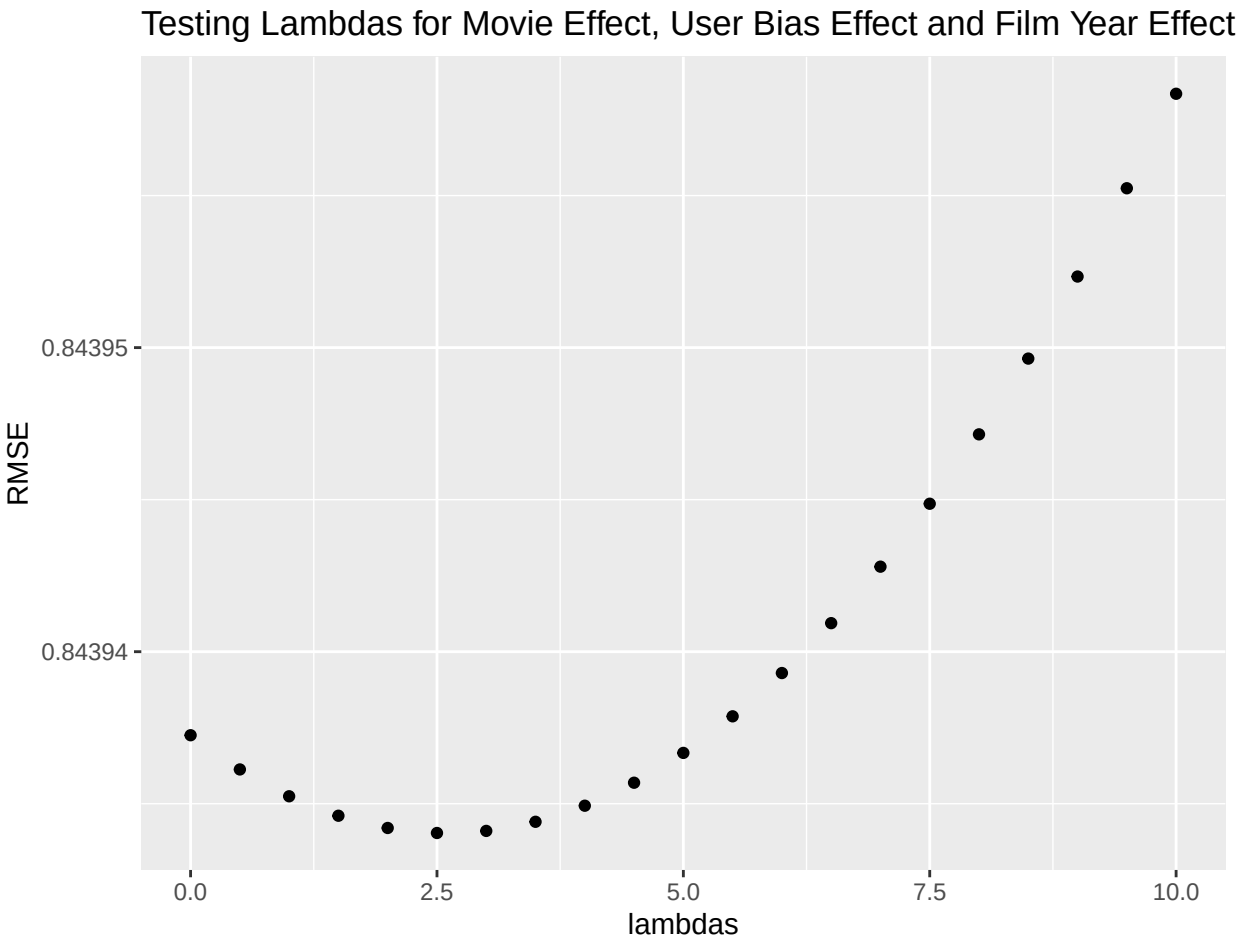
But can we do even better? Next, I will attempt to add in the year into the mix, testing whether the age of the movie makes a difference. For this we have created the field "age_of_movie"

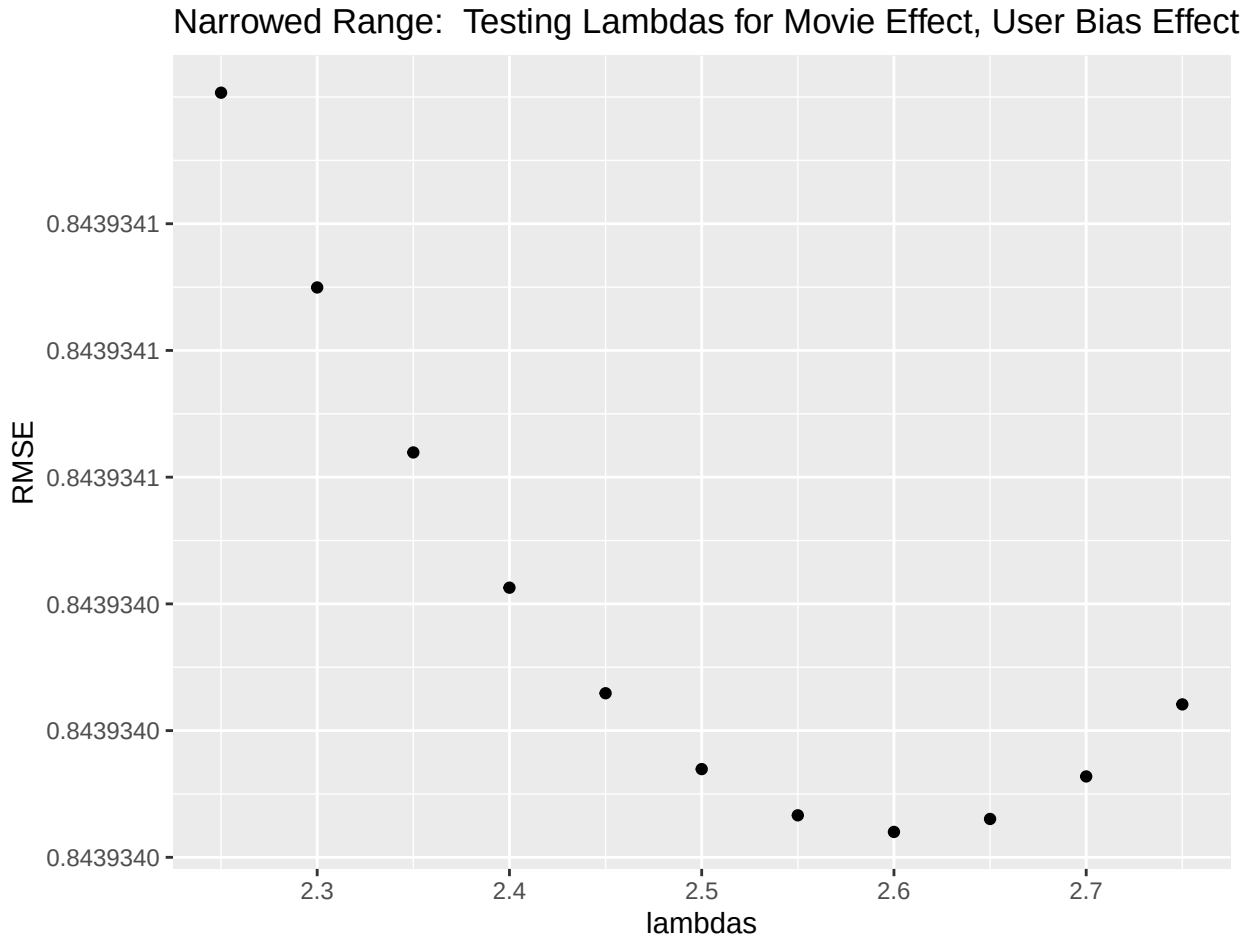
$$\frac{1}{N}\sum_{u,i}\left(y_{u,i}-\mu-b_i-b_u-b_y\right)^2+\lambda\left(\sum_i b_i^2+\sum_u b_u^2+\sum_i b_y^2\right)$$

with b_y as the **release year effect**.

```
regularized_movie_and_user_and_year <- function(l, training_set, testing_set)
{
  mu <- mean(training_set$rating)

  b_i <- training_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))

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

  b_y <- training_set %>%
    left_join(b_i, by='movieId') %>%
    left_join(b_u, by="userId") %>%
    group_by(age_of_movie) %>%
    summarize( b_y = sum(rating - b_i - b_u - mu)/(n()+l))

  predicted_ratings <-
    testing_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(b_y, by = "age_of_movie") %>%
    mutate(pred = mu + b_i + b_u + b_y) %>%
    pull(pred)

  rmse <- RMSE(predicted_ratings, testing_set$rating)
  return(rmse)
}

model_5_lamdba <- find_generic_lambda(seq_start=0, seq_end=10,
    seq_increment=0.5,
    FUN= function(x) regularized_movie_and_user_and_year(x,
              training_set=train_set, testing_set=test_set ),
```

```
    detailed_flag = TRUE, training_set=train_set,
    testing_set=test_set, plot_title =
    "Testing Lambdas for Movie Effect, User Bias Effect and Film Year Effect")
```






```
model_5_rmse <-
  regularized_movie_and_user_and_year(model_5_lamdba, train_set, test_set)
tmp_rmse_results <-
  tibble(method = "Regularized Movie + User + Year Effect Model",
         RMSE = model_5_rmse)
#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, tmp_rmse_results)
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068
3	User Effect + Movie Effect Model	0.8440959
4	Regularized + Movie Effect	0.9266057
5	Regularized Movie + User Effect Model	0.8440903
6	Regularized Movie + User + Year Effect Model	0.8439340

The added year does indeed seem to have made a difference in the calculations; The next item to be tested is the genre.

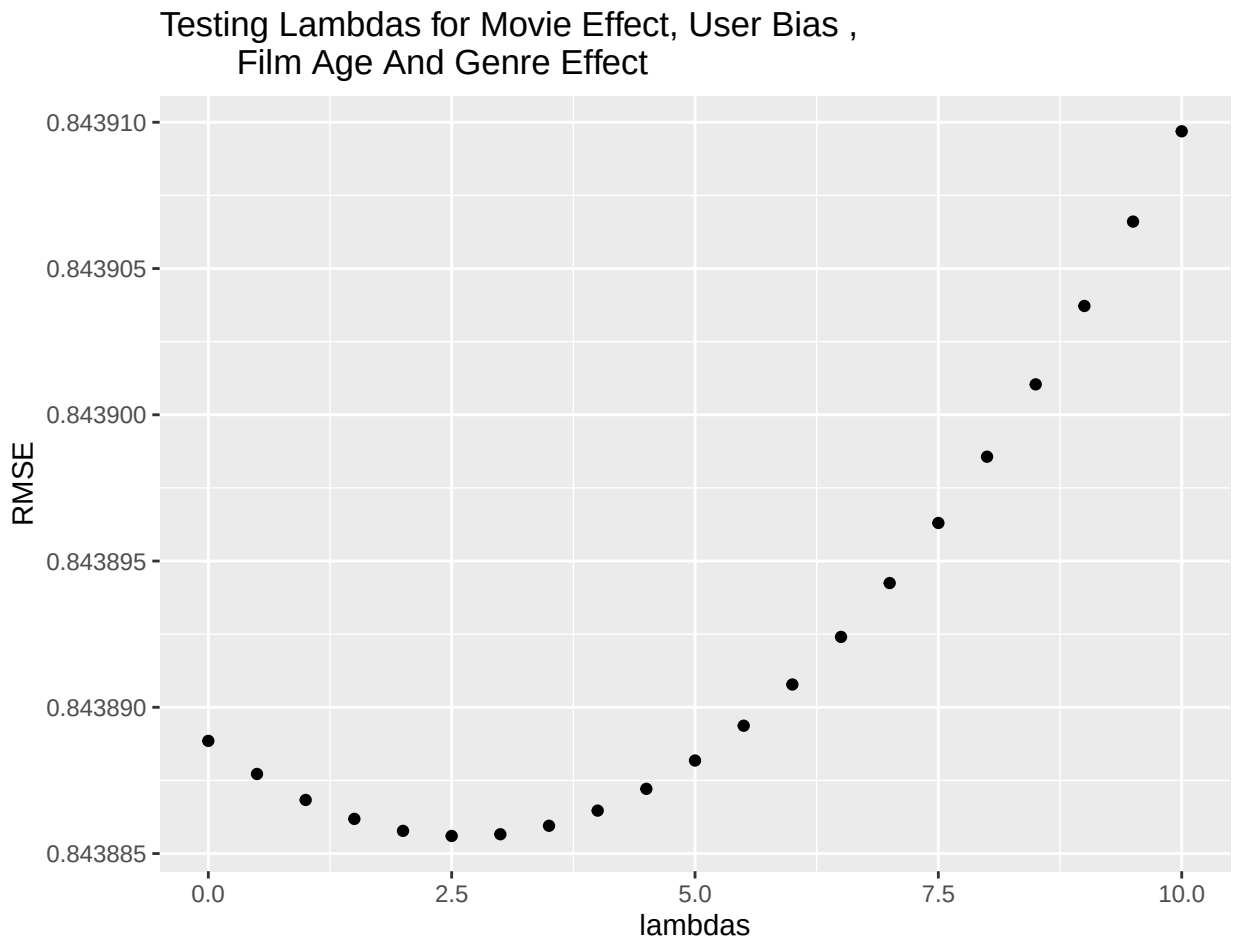
4.4 Movie + User + Release Year + Genre Effect Regularized Model

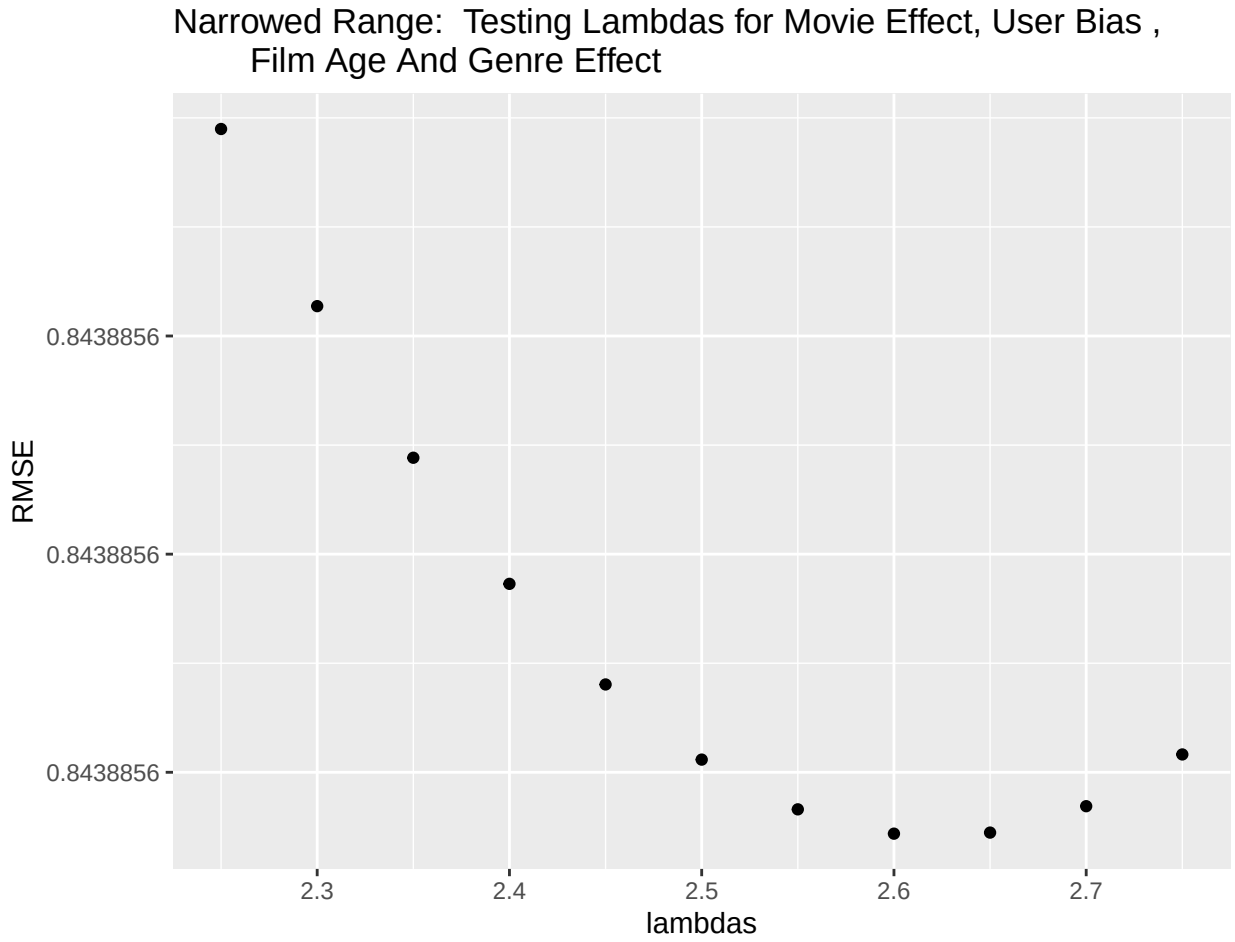
Following the pattern of our previous regularized models, we add in b_g to represent the **genre effect** into the regularization mix, using the following formula:

$$\frac{1}{N}\sum_{u,i}\left(y_{u,i}-\mu-b_i-b_u-b_y-b_g\right)^2+\lambda\left(\sum_i b_i^2+\sum_u b_u^2+\sum_i b_y^2+\sum_i b_g^2\right)$$

```
regularized_movie_and_user_and_year_and_genre <-
  function(l, training_set, testing_set)
{

  mu <- mean(training_set$rating)

  b_i <- training_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))

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

  b_y <- training_set %>%
    left_join(b_i, by='movieId') %>%
    left_join(b_u, by="userId") %>%
    group_by(age_of_movie) %>%
    summarize(b_y = sum(rating - b_i - b_u - mu)/(n()+l))

  #adding in the genre bias here
  b_g <- training_set %>%
    left_join(b_i, by='movieId') %>%
    left_join(b_u, by="userId") %>%
    left_join(b_y, by="age_of_movie") %>%
    group_by(genres) %>%
    summarize( b_g = sum(rating - b_i - b_u - b_y - mu)/(n()+l))


  predicted_ratings <-
    testing_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(b_y, by = "age_of_movie") %>%
    left_join(b_g, by = "genres") %>%
    mutate(pred = mu + b_i + b_u + b_y + b_g) %>%
```

```
    pull(pred)

  rmse <- RMSE(predicted_ratings, testing_set$rating)

  return(rmse)
}

model_6_lambda <- find_generic_lambda(seq_start=0, seq_end=10,
    seq_increment=0.5, FUN= function(x)
      regularized_movie_and_user_and_year_and_genre(x,
            training_set=train_set, testing_set=test_set ),
           detailed_flag = TRUE, training_set=train_set,
          testing_set=test_set,
    plot_title =
      "Testing Lambdas for Movie Effect, User Bias ,
        Film Age And Genre Effect")
```






```
model_6_rmse <- regularized_movie_and_user_and_year_and_genre(
    model_6_lambda,  train_set, test_set)

tmp_rmse_results <- tibble(method =
    "Regularized Movie + User + Year + Genre Effect Model",
    RMSE = model_6_rmse)

#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, tmp_rmse_results)
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068
3	User Effect + Movie Effect Model	0.8440959
4	Regularized + Movie Effect	0.9266057
5	Regularized Movie + User Effect Model	0.8440903
6	Regularized Movie + User + Year Effect Model	0.8439340

	method	RMSE
7	Regularized Movie + User + Year + Genre Effect Model	0.8438856

```
rm(tmp_rmse_results)
```

Results

After creating a number of predictive algorithms, the best model as per the project requirements is the one with the least RMSE : **'Regularized Movie, User, Release Year and Genre Effect Model'**. Now that we've selected this model, it will be applied to the **validation data set** (which has been untouched as of yet).

```
final_rmse <- regularized_movie_and_user_and_year_and_genre(model_6_lambda,
                      train_set, validation)
```

The final RMSE value calculated from the validation set is: 0.8435003.

well below the required threshold of **0.8649**. We can now review the complete set of RMSE calculations :

```
final_rmse_results <- tibble(method = "Final Model Tested on Validation Set",
                             RMSE = final_rmse)

#add to existing rmse results table
rmse_results <- bind_rows(rmse_results, final_rmse_results)

# The final output
rmse_results %>% knitr::kable(row.names=TRUE)
```

	method	RMSE
1	Just the average	1.0406546
2	Movie Effect Model	0.9266068
3	User Effect + Movie Effect Model	0.8440959
4	Regularized + Movie Effect	0.9266057
5	Regularized Movie + User Effect Model	0.8440903
6	Regularized Movie + User + Year Effect Model	0.8439340
7	Regularized Movie + User + Year + Genre Effect Model	0.8438856
8	Final Model Tested on Validation Set	0.8435003

It is apparent from the numbers presented above that without regularization, simply applying Model #3, the **"User Effect + Movie Effect Model"** will actually give us an RMSE low enough to be acceptable, as the output of that model, **0.8440959** is less then the threshold value of **0.8649**. (A notable observation : these numbers were only acceptable after the data has been massaged to remove films that had too few ratings, and remove users who had less then 100 reviews.) However, I wanted to examine other options for improvement and introduced regularization into the models. The simplest regularization model, using only the movie affect with an optimized

lambda, was not good enough for our calculations, and actually regressed from our third model, described above. However, regularization with movie AND user effect beat the previously best-performing model. Adding in the release year in model # 6 increased our performance even further, and finally, using regularization combined with the movie, user, release year and genre effects proved to be the most efficient model. **Applying this final model, #7, to the validation set, gave us a validated RMSE of 0.8435003**.

Conclusion

Report Summary

In this project, I have built a machine learning algorithm to predict movie ratings with MovieLens dataset, and estimate their accuracy with RMSE. The **regularized model including the effect of user, movie, release year and genre** is characterized by the lowest RMSE value of all I have examined, and is hence the optimal model to use for the present project.

The final rating model uses **regularization**, which constrains the total variability of the effect sizes by penalizing large estimates that come from small sample sizes. With a tuning parameter λ to generate a rating $Y_{u,i}$ for movie i by user u. These calculations can be summarized with the following formula:

$$\frac{1}{N}\sum_{u,i}\left(y_{u,i}-\mu-b_i-b_u-b_y-b_g\right)^2+\lambda\left(\sum_i b_i^2+\sum_u b_u^2+\sum_i b_y^2+\sum_i b_g^2\right)$$

where the variables are:

- μ is the average rating across all movies
- b_i is the per-movie movie bias
- b_u is the per-user movie bias
- b_y is the age of movie (year) bias
- b_g is the genre bias

The first term in the equation above is the mean squared error calculation , and the second term is the penalty term that gets larger when all the effects listed above (movie effect, user effect, release year effect, and genre effect) are large. Re-working the equation using calculus we can show that we need a λ value that will minimize the equation that adds the penalty for all of the effects mentioned.

The RMSE arrived at with this model is **0.8435003** - which is sufficiently accurate, as it is lower than the initial evaluation criteria **0.8649** given by the goal of the present project.

It is worth noting that the current RMSE numbers were achieved after editing the data to remove users with less then 100 reviews, and movies with less then 350 reviews. Without removing this data, comprised of mostly outliers, the RMSE was a lot more accurate.

Other Options

Using Various Regression Functions.

During this process I have also attempted to run the predictions using the `train()` R function, with methods that included Linear Regression ("lm"), LASSO regression ("glmnet"), and Generalized Linear Regression ("glm"). When conducted on a small part of the data set, these functions produced an inferior RMSE result, not in line with the threshold required. More importantly, all 3 of these methods crashed the system when attempted to be run on the full dataset - showing that these are not realistic tools to be used for datasets of this size (in the millions).

Possible Improvements

I think improvements on the RMSE could be achieved by evaluating the project with other means. Different machine learning models could also improve the results further, but hardware limitations (memory and processing power) may be a constraint.

The possible evaluation models we can also attempt to get even better results would include:

- Matrix factorization / Single Value Decomposition (SVD) / Principal Component Analysis (PCA)
- Gradient Descent
- SGD Code Walk

Appendix - Environment

```
print("Operating System:")
```

```
## [1] "Operating System:"
```

```
version
```

```
##                _
## platform       x86_64-apple-darwin17.0
## arch           x86_64
## os             darwin17.0
## system         x86_64, darwin17.0
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
## version.string R version 4.0.2 (2020-06-22)
## nickname       Taking Off Again
```