HarvardX Data Science
Capstone Project I
A Movielens Recommendation System

Guido D'Alessandro

June 02, 2019

Contents

1 Introduction

This report is a presentation of the process of selecting and evaluating some predictive models to be used in a fictitious post-mortem Netflix's 2006 forecasting challenge, including some of the methods that were used by the winning team, sewn together by series of analyses for the purpose of improving the predictive power of the previous model, in a step-by-step approach, by way of minimizing the loss function, which for this case was chosen to be the root mean square error (RMSE) of the predictions. The report therefore, follows the presentation of a series of forecasting model improvements. Each of which is introduced after some justification based on the current model caveats and additional data inspections.

Recommendation systems do exactly what their name says they do—-they recommend something to clients. Their output is usually in the form of a rating, such as 4 out of 5 and 7.5 out of 10, for example. These ratings are designed to aid clients in making a selection among many choices.

In October 2006, Netflix offered a challenge to the data science community to improve their movie recommendation algorithm by 10% and thus win a million dollars. In September, 2009, the winners were announced. To win the grand prize of $1 million, the winning team achieved a RMSE of approximately 0.8567 ("BellKor's Pragmatic Chaos" team RMSE on the test data). To be included in the top 10 list we would need a RMSE of 0.88 or bettwr. Finally, a RMSE of 0.889 would get us into the leaderboard. Let's see how close we can get to these results.

2 Data Inspections and Analyses

2.1 Introduction

The Netflix data used for the challenge is not publicly available, but the GroupLens research lab generated their own database with over 20 million ratings for over 27,000 movies by more than 138,000 users of which the Data Science staff of Harvard University made a small subset, available for download, which can be retrieved like this:

```
# Set working directory
print( paste("======= Setting working directory to", work_dir) )
## [1] "======= Setting working directory to C:/Users/dax/Documents/datascience/capstone/movielens"
setwd(work_dir)

# Verify working directory
if ( getwd() != work_dir ) {
  stop("Incorrect working directory")
}
print( paste("======= Working directory set to ", getwd(), sep="") )
## [1] "======= Working directory set to C:/Users/dax/Documents/datascience/capstone/movielens"

# check if table files exist
# else download and generate
if ( all(file.exists(dataset_files)) ) {
    f_sourceScript("datasets-load.R")

} else {

####################################
# Download and save Movilens dataset
####################################

# Note: this process could take a couple of minutes

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

# MovieLens 10M dataset:
# Weong Link: https://grouplens.org/datasets/movielens/10m/
# Correct Link: http://files.grouplens.org/datasets/movielens/ml-10m.zip

print("======= Downloading main file...")

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

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

movies <- str_split_fixed(readLines(unzip(dl, exdir = ".\\data", "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")

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

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

```
saveRDS(movielens, file="data/downloaded/movielens.rds")

str(movielens, vec.len=1)
}
## [1] "======= Setting working directory..."
## [1] "======= Working directory set to C:/Users/dax/Documents/datascience/capstone/movielens"
```

Where we see the data set consists of *10000054* rows x 6 columns: *userId, movieId, rating, timestamp, title, genres*

So after this short inspection we continue to separate the dataset into training (edx) and testing or validation (val) parts as follows:

```
# Verify working directory
if ( getwd() != work_dir ) {
  stop("Incorrect working directory")
}
print( paste("======= Working directory set to ", getwd(), sep="") )
## [1] "======= Working directory set to C:/Users/dax/Documents/datascience/capstone/movielens"

# check if table files exist
# else download and generate
if ( all(file.exists(dataset_files)) ) {
    f_sourceScript("datasets-load.R")
} else {

####################################
    # CREATE TRAINING AND TESTING SETS
    ####################################

    ## Validation set will be 10% of MovieLens data

    ## set.seed(1) # if using R 3.6.0: set.seed(1, sample.kind = "Rounding")
    print("======= Partitioning data... 90% train (edx), 10% test (validation)")
    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

    print("======= Creating test (val) section...")
    val <- 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, val)

    print("======= Creating train (edx) section")
    edx <- rbind(edx, removed)

    print("======= Removing temporaries")
    rm(dl, ratings, movies, test_index, temp, removed)

    # Verify working directory
    if ( getwd() != work_dir ) {
```

```
    stop("Incorrect working directory")
}
print("======= Working directory verifies OK")

# Keep a copy of these files for reference
print("======= Saving a copy of the generated files")
saveRDS(edx, file="data/generated/edx.rds")
saveRDS(val, file="data/generated/validation.rds")

####################################
# DOWNLOADED TRAINING AND TEST
# VERSION -- IN CASE OF DIFERENCES
# USING R v. 3.6.0
# NOTE THE DOWLOADED VERSIONS WERE
# DOWNLOADED FROM:
# https://drive.google.com/drive/folders/1IZcBBXOOmL9wu9AdzMBFUG8GoPbGQ38D
#
# NOTE 5/6/19: Do not run the code below if you are using R version 3.6.0.
# Instead, download the edx and validation data sets from here
# https://drive.google.com/drive/folders/1IZcBBXOOmL9wu9AdzMBFUG8GoPbGQ38D
# Update 5/12/19: If you are using R version 3.6.0, please use
# set.seed(1, sample.kind = "Rounding") instead of set.seed(1) in the code below.
#
# Later I was told that in case of discrepancies to use the downloaded versions
# which is what much of the code below establishes

print("======= Loading Manually downloaded versions...")

edx_download <- readRDS("data/downloaded/edx.rds")
val_download <- readRDS("data/downloaded/validation.rds")


print("======= Comparing generated and downloaded versions...")

e_same <- TRUE
ne1 <- nrow(edx)
ne2 <- nrow(edx_download)
if ( ne1 != ne2 ) {
    print( paste("======= Downloaded and generated edx.rds files differ by", abs(ne1-ne2), "rows"))
    e_same <- FALSE
} else {
    ej <- inner_join(edx, edx_download)
    nej <- nrow(ej)
    if ( ne1 != nej ) {
        print( paste("======= Downloaded and generated edx.rds files differ after inner_join by", abs(n
        e_same <- FALSE
    }
}
if ( e_same ) {
    print("======= edx.rds downloaded and generated are equal, using generated version")
    saveRDS(edx, file="data/rds/edx.rds")
} else {
    print( "======= Because of edx.rds file differences, using downloaded edx version." )
    edx <- edx_download
    saveRDS(edx, file="data/rds/edx.rds")
}
```

```
    v_same <- TRUE
    nv1 <- nrow(val)
    nv2 <- nrow(val_download)
    if ( nv1 != nv2 ) {
        print( paste("======= Downloaded and generated validation.rds files differ by", abs(nv1-nv2), "rows
        v_same <- FALSE
    } else {
        vj <- inner_join(val, val_download)
        nvj <- nrow(vj)
        if ( nv1 != nvj ) {
            print( paste("======= Downloaded and generated validation.rds files differ after inner_join by"
            v_same <- FALSE
        }
    }
    if ( v_same ) {
        print("======= validation.rds downloaded and generated are equal, using generated version")
        saveRDS(val, file="data/rds/validation.rds")
    } else {
        print( "======= Because of validation.rds file differences, using downloaded version." )
        val <- val_download
        saveRDS(val, file="data/rds/validation.rds")
    }
}
## [1] "======= Setting working directory..."
## [1] "======= Working directory set to C:/Users/dax/Documents/datascience/capstone/movielens"

# edx.rds and validation.rds correct files saved to ./data/used
# objects edx and val contain correct rows
```

2.2 Observations

2.2.1 Date and Time Fields

It is easy to note that the timestamp column is an *integer* that corresponds to an unclassed POSIXct date (seconds since 1970-01-01T00:00:00UTC) and not a datetime class.

2.2.2 Distinct Users and Movies

The number of participating unique users rating movies and the number of unique movies that were rated in this table is given by:

```
library(dplyr)
distinct <- movielens %>%
  summarize(n_users = n_distinct(userId), n_movies = n_distinct(movieId))
distinct %>% knitr::kable()
```

n_users	n_movies
69878	10677

2.2.3 Users and Ratings

```
# Calculating ratings per user,
# arrange in descending order
movie_ratings <- edx %>% group_by(movieId) %>%
  summarize(n_ratings = n(), avg_rating = mean(rating))

movie_ratings <- movie_ratings %>% arrange(desc(n_ratings), desc(avg_rating))
```

The fact that the number of users × the number of movies exceeds the number of rows in the dataset, 10000054, implies that not all movies were rated by all users, which is also supported by the fact that the maximum number of users rating any one movie, 31362, is smaller than the total number of users.

2.2.4 Missing Values

We checked for columns with missing values as follows:

```
colnames(movielens)[colSums(is.na(movielens)) > 0]
## character(0)
```

and found none. Therefore we allowed all movie ratings into this study.

2.2.5 Duplicated Ratings

It is important to verify that there were no duplicated ratings from a distinct user for a particular movie, that is a rating made on one movie more than once by the same user. It is important because some of the analyses that we will conduct assumes that. This is how we checked for duplicates:

```
head( movielens[duplicated(movielens[c("movieId","userId")]),] )
## [1] userId    movieId   rating    timestamp title     genres
## <0 rows> (or 0-length row.names)
```

and again, found there were none.

2.3 Training and Testing Sections

It is customary to divide a dataset in two parts. One that serves as the training arena where we postulate and fit solutions, and in part test them, and another, exclusively reserved to test how really good our solutions are, and in a way, to serve as a safeguard against overfitting. It is therefore a good practice. We already divided the *movielens* table with the help of the *caret* library function *createDataPartition* into a training section (90% of the data), which we called the *edx set* (edx), and a test section (the remaining 10% of the data), which we called the **validation set** (val), right after downloading the dataset, earlier in this study.

2.4 Ratings Distribution

Here we rank movies according to how many times they are rated. The idea for this computation is to visualize the relationship between movie ratings and number of times the movie is rated. We did that earlier, when we created the movie_ratings table. Here we show the average rating on the 5 most rated movies and the average rating on the least 5 rated movies like this:

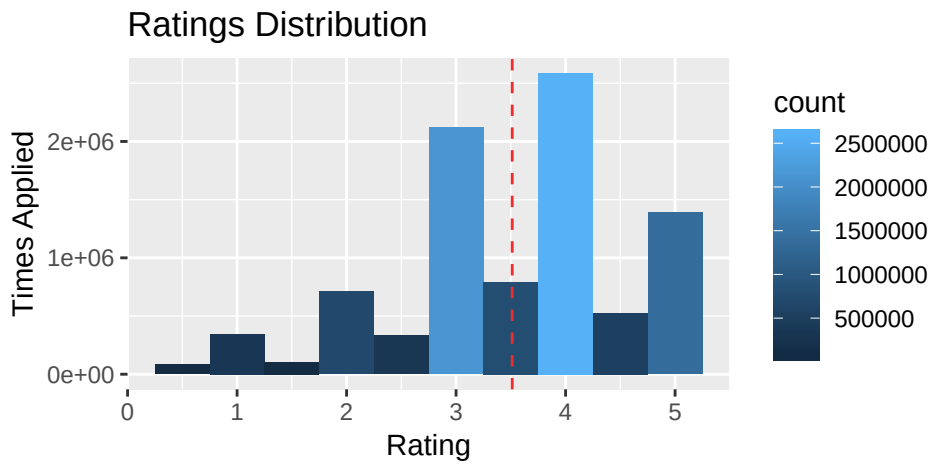
```r
# Movies that are rated the most
movie_ratings %>% head(n=5)%>% knitr::kable()
```

movieId	n_ratings	avg_rating
296	31362	4.154789
356	31079	4.012822
593	30382	4.204101
480	29360	3.663522
318	28015	4.455131

```r
# Movies that are rated the least
movie_ratings %>% tail(n=5) %>% knitr::kable()
```

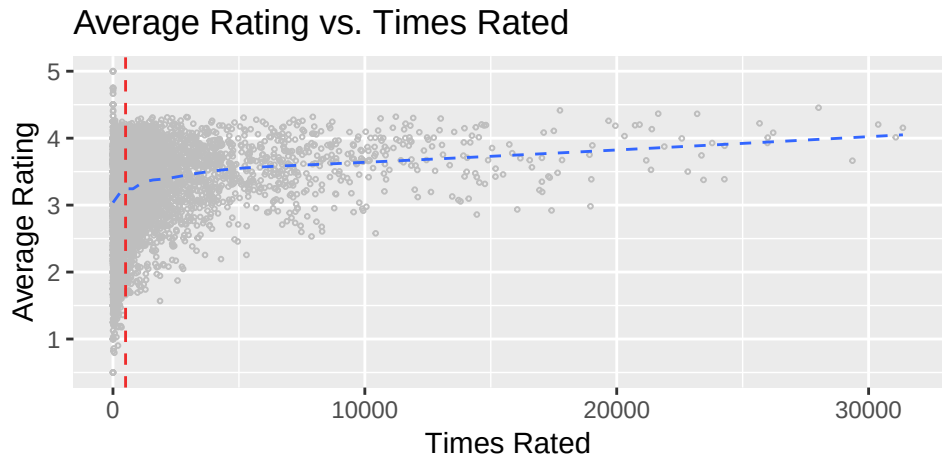
movieId	n_ratings	avg_rating
6189	1	1.0
55324	1	1.0
8394	1	0.5
61768	1	0.5
63828	1	0.5

We can also visualize the frequency of movie ratings like this:



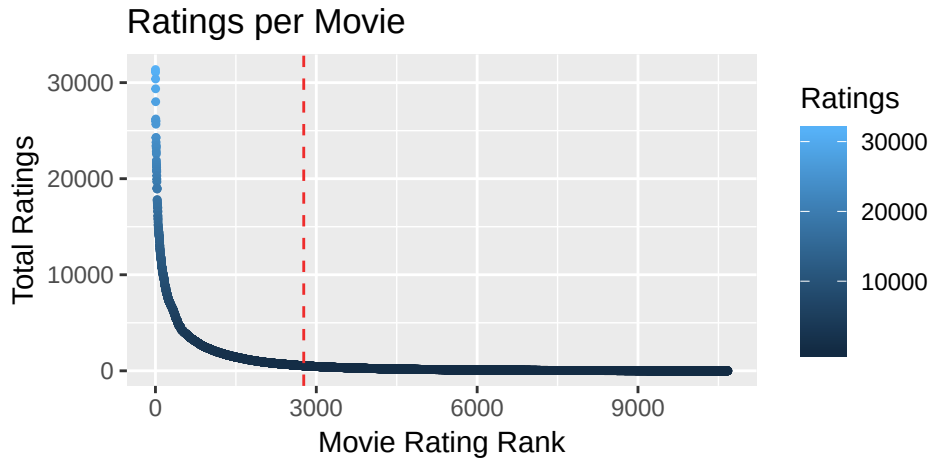
The graph above shows the distribution of the ratings throughout the training data set and the overall mean of the movie ratings (vertical dashed red line).

And also show that most of the top and low rated movies are movies that are rated less than ≈ 500 times (vertical dashed red line), from which we can infer that estimates from these ratings make poor prodictors.



2.5 Ratings per Movie

We saw the number of ratings made on every movie in the movie ratings table where the top and least 5 rated movies were listed. This table can be visualized like this:



The graph above shows that 25.91% of the movies received about 90% of the ratings (area under the curve left of the dashed red line).

2.6 Ratings per User

We can carry the same analysis to understand how users rate movies, some more than others, ranging from 6616 ratings by the most active user to a minimum of 20, by many other least active. Here are the 5 most active and 5 of the least active rating users:

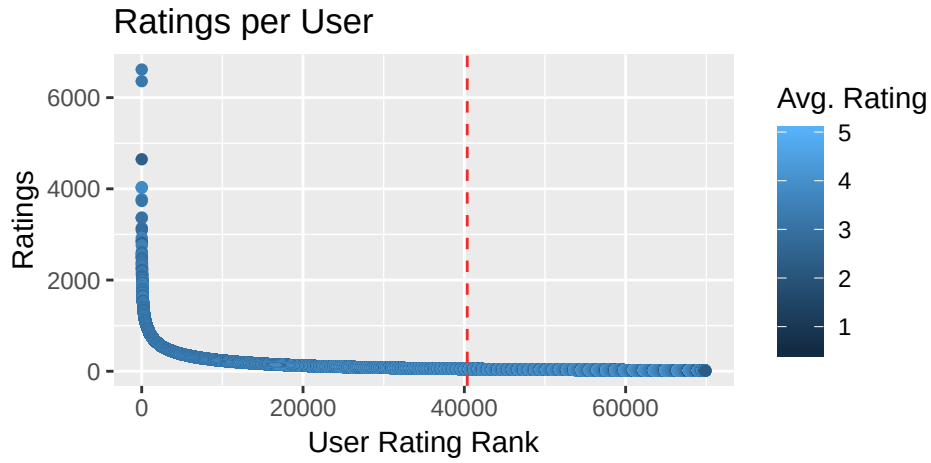
```
# Most active users
user_ratings %>% head(n=5) %>% knitr::kable()
```

userId	n_ratings	avg_rating
59269	6616	3.264586
67385	6360	3.197720
14463	4648	2.403615
68259	4036	3.576933
27468	4023	3.826871

```r
#Least active users
user_ratings %>% tail(n=5) %>% knitr::kable()
```

userId	n_ratings	avg_rating
5214	14	1.785714
50608	13	3.923077
15719	13	3.769231
22170	12	4.000000
62516	10	2.250000

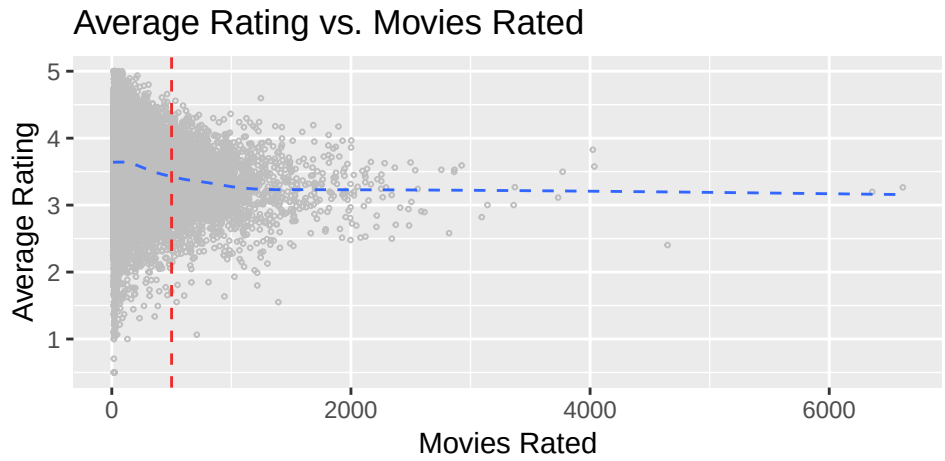
and rank participating users according to the movies they rated as follows:



The graph above shows that 57.76% of the users carried about 90% of the movie ratings (area under the curve left of the dashed red line).

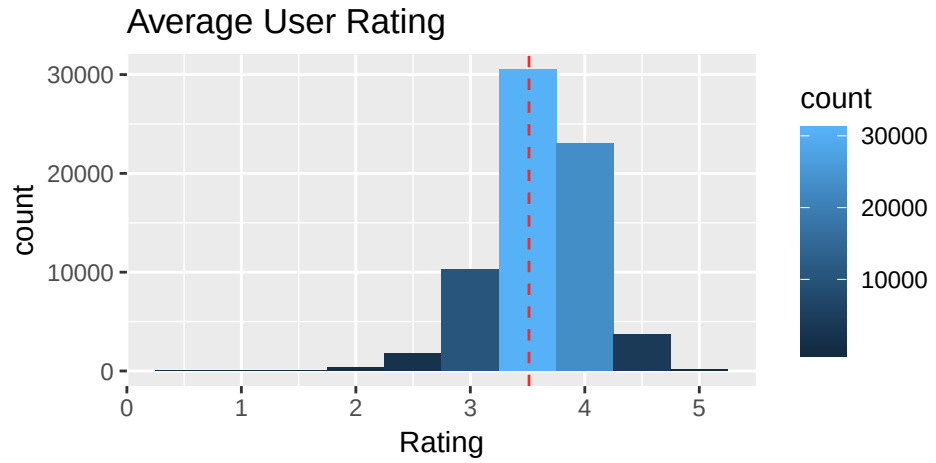
And lastly show that most of the top and low rated movies by users are from users that have rated less than ≈ 500 movies (vertical dashed red line), from which we can infer that estimates from these ratings make poor prodictors.

```r
user_ratings %>%
  ggplot(aes(n_ratings, avg_rating)) +
  geom_point(shape=1, size=0.5, color="gray") +
  geom_smooth(linetype="dashed", size=0.5, se=FALSE) +
  labs(x = "Movies Rated", y = "Average Rating") +
  geom_vline(xintercept = 500, color = "firebrick2", linetype = "dashed") +
  ggtitle("Average Rating vs. Movies Rated")
```



2.7 Average User Rating

When we plot the histogram of the average movie ratings by user, we can see that on the average, users tend to rate movies higher, that is there are more users rating movies higher than lower:



The figure above shows the distribution of the average ratings by individual users and the overall mean of the movie ratings (vertical dashed red line).

3 Building the Models

3.1 Introduction

The Netflix challenge decided on a winner based on the residual RMSE over a test set. So if we define $y_{u,i}$ as the rating for movie i by user u and $\hat{y_{u,i}}$ as our prediction, then the residual root mean square error is defined as follows:

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

Where N is the total number of user-movie combinations on which the sum in the equation above takes place. These residuals are treated as the standard deviation of the predictions. If this number is large, say one (1) for example,

we would be missing by one or more stars the target rating about 67% of the time (assuming a normal distribution of the errors), which for the Netflix challenge case was not a good reduction of the error.

Because we will be computing the RMSE often in this study, it is a good idea to create a function to that purpose so we would need to only include it once in the code:

```
f_RMSE <- function(actual_rating, predicted_rating) {
  round( sqrt(mean((actual_rating - predicted_rating)^2, na.rm=TRUE)), rnd_rmse)
}
```

3.2 Baseline Models

3.2.1 The Naive Model

From lessons in statistics, we know that the mean of a sample is the estimate that minimizes the residual errors acros the sample. Therefore, we apply the same principle and assume a rating system based on the mean of all the movie ratings by all users and explain the differences by random variation:

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

Where $\varepsilon_{u,i}$ represents independent errors from the given sample distribution centered at zero–the random variation, and μ represents the mean rating among all movies and users, which we compute like this:

```
mu <- mean(edx$rating)
mu
## [1] 3.512465
```

which should yield a RMSE residual to approximately the standard error of the validation sample:

```
sd(val$rating)
## [1] 1.061202
```

that we can verify when applying this prediction to the RMSE function we created earlier:

```
m1_rmse <- f_RMSE(val$rating, mu)
m1_rmse
## [1] 1.0612
```

to conclude, this model yields a RMSE of 1.0612, which we already explained to be the result of a poor estimate. Therefore, we will continue improving our model until we can reduce its RMSE as close to 0.8567 as possible, as explained in the models introduction of this study.

To keep tabs on the performance of our models, we build a table where we add each model's RMSE as we go along:

```
result_tabs <- data.frame(Method = "Naive", RMSE = m1_rmse, Improvement="NA")
result_tabs %>% knitr::kable()
```

Method	RMSE	Improvement
Naive	1.0612	NA

3.2.2 Movie Effects (ME) Model

We continuing to build our model on top the previous naive model taking to account that some movies are generally rated higher than others, so we add another term to our previous model and call it b_i, b for bias:

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

where b_i stands for the average rating of movie i after deducting the prediction of the previous naive model:

```
avg_movie_ratings <- edx %>% group_by(movieId) %>%
  summarize(b_i = mean(rating - mu) )
```

which we use for predicting our movie ratings like this:

```
y_hat <- val %>% left_join(avg_movie_ratings, by='movieId') %>%
  mutate(pred = mu + b_i) %>% .$pred
m2_rmse <- f_RMSE(val$rating, y_hat)
improv <- round(100*(m1_rmse-m2_rmse)/m1_rmse, rnd_pct)
m2_rmse
## [1] 0.9439
```

where we see an improvement of 11.05% over the 1.0612 achieved previously with the naive method. Better, but not enough, so we continue improvement and add this result to our RMSE tabs table, as we previously indicated:

Method	RMSE	Improvement
Naive	1.0612	NA
Naive + ME	0.9439	11.05%

3.2.3 User Effects (UE) Model

This model also builds on the Movie Effects Model taking to account that different users rate movies differently and so we add another term to the previous model and call it b_u, for user bias:

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

where b_u is the user-specific effect, which we compute like this:

```
 avg_user_rating <- edx %>%
   left_join(avg_movie_ratings, by='movieId') %>%
   group_by(userId) %>%
   summarize(b_u = mean(rating - mu - b_i))
```

which we use to predict the the movie ratings like this:

```
y_hat <- val %>%
   left_join(avg_movie_ratings, by='movieId') %>%
   left_join(avg_user_rating, by='userId') %>%
   mutate(pred = mu + b_i + b_u) %>% .$pred
m3_rmse <- f_RMSE(val$rating, y_hat)
improv <- round(100*(m2_rmse-m3_rmse)/m2_rmse, rnd_pct)
```

```
overall <- round(100*(m1_rmse-m3_rmse)/m1_rmse, rnd_pct)
m3_rmse
## [1] 0.8653
```

Here we see yet another improvement of 8.33% over the 0.9439 achieved with the previous naive + me method and a whooping overal 18.46% over the original naive method. Even if this amount is lower than our target error 0.8567 we can improve residual reduction, so we continue by adding this result to our tabs table and proceed to improve our model:

```
result_tabs <- bind_rows(result_tabs, data.frame(Method="Naive + ME + UE",
  RMSE=m3_rmse,
  Improvement=paste(overall, "%", sep="")))
result_tabs %>% knitr::kable()
```

Method	RMSE	Improvement
Naive	1.0612	NA
Naive + ME	0.9439	11.05%
Naive + ME + UE	0.8653	18.46%

Note: Improvements displayed in this table are in relation to the Naive method alone.

3.3 Regularized Models

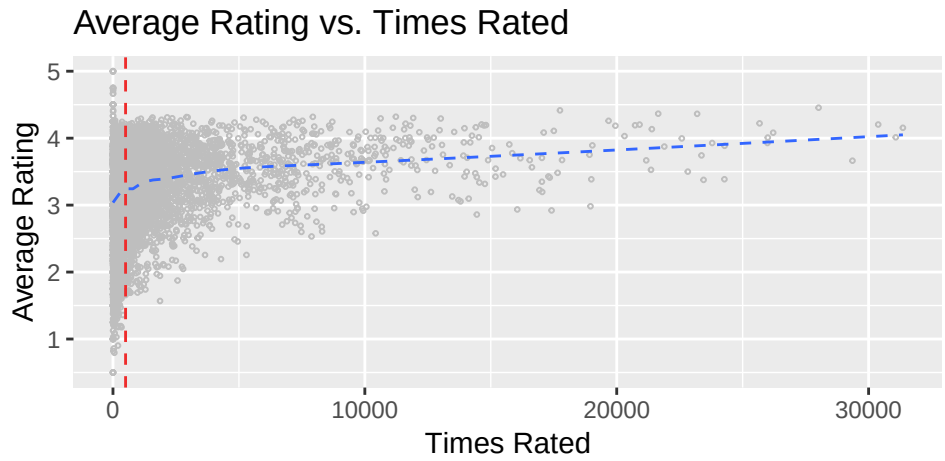
3.3.1 Introduction

In *Machine Learning*, **regularization** is a principle implying techniques that for most of the time are used to solve overfitting problems found in statistical models. We find an overfitting problem in a statistical model as the result of optimizing a model to perform better according to some *loss metric* within the scope of a training data section, just to find later it performs worse in the test section. However, some other times, as in this case, *regularization* is introduced to existing models with the hope that it will improve the models' performance in both the training and testing data sections. Lets see if we can do this.

3.3.2 Analysis

Despite the large variation in movie to movie ratings, our improvement in the residual RMSE when we added the previous movie effects model was only about 6.58%. Let's investigate why wasn't this contribution larger and what we can do to improve it.

We saw in the ratings distribution section that most of the best and worst rated movies were rated by very few users, in many cases by just one user, as reproduced below:



Note: This plot shows how most of the top and low rated movies are rated less than 50 times or so (vertical dashed red line). Confirming the suscition that larger ratings, positive or negative, are most likely when fewer users rate a movie, and often by just one. Therefore, we should not fully trust, specially for prediction.

3.3.3 Regularized Movie Effects (RME) Model

As metioned in the analysis section, we should not fully trust predictions from ratings that result from just a few users, so we need a way to penalize these predictions as to minimize their impact to our model. To do this, we appeal to the principle of *regularization*, where the idea is to add a penalty to large values of b_i with few ratings by minimizing this equation:

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

where the first term is the residual sum of squares, and the second, $\lambda\sum_i b_i^2$, the penalty term. It can be proven through calculus that the b_i's that minimize this equation are given by:

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

where n_i is the number of ratings b for movie i. Here we can see that when n_i is large, the penalty term becomes small, and when small, the penalty term becomes significant, which is the effect we want.

Despite of this, the equations above do not prescribe a value for λ. For this reason we will have to select it from a cross validation inside the prediction code like this:

```
## Iterate through values of lambda
## and choose best performance
lambdas <- seq(0, 10, 0.1)
rmses <- sapply(lambdas, function(lambda) {

   # b_i by movie
   b_i <- edx %>%
      group_by(movieId) %>%
      summarize(b_i = sum(rating - mu) / (lambda + n()))

   # Calculate prediction ratings on test dataset
   pred <- val %>%
      left_join(b_i, by='movieId') %>%
```

```
      mutate(pred = mu + b_i) %>% .$pred

   return( f_RMSE(val$rating, pred) )
})

# Best Lambda
lambdas[ which.min(rmses) ]
## [1] 0

# Min RMSE
m4_rmse <- min(rmses)
m4_rmse
## [1] 0.9439

# Save improvements
improv <- round(100*(m2_rmse-m4_rmse)/m2_rmse, rnd_pct)
overall <- round(100*(m1_rmse-m4_rmse)/m1_rmse, rnd_pct)
```

Where we see the error that is returned by the regularized movie effects prediction has an improvement of 0% over the previous movie effects model. Despite this, we accept this model and add its result to the tabs table as we did before:

```
result_tabs <- bind_rows(result_tabs, data.frame(Method="Naive + RME",
  RMSE=m4_rmse,
  Improvement=paste(overall, "%", sep="")))

# Add to RMSE tabs table
result_tabs %>%  knitr::kable()
```

Method	RMSE	Improvement
Naive	1.0612	NA
Naive + ME	0.9439	11.05%
Naive + ME + UE	0.8653	18.46%
Naive + RME	0.9439	11.05%

Note: Improvements displayed in this table are in relation to the Naive method alone.

3.3.4 Regularized User Effects (RUE) Model

We are going to build in the previous regularized movie effects model and we are going to apply a similar argument, though this time the function we have to minimize is the following:

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

As before, we also use cross validation inside the predicting algorithm to select the most suitable λ as follows:

```
## Iterate through values of lambda
## and choose best performance
lambdas <- seq(0, 10, 0.1)
rmses <- sapply(lambdas, function(lambda) {

   # b_i by movie
```

```
   b_i <- edx %>%
      group_by(movieId) %>%
      summarize(b_i = sum(rating - mu) / (lambda + n()))

   # b_u by users
   b_u <- edx %>%
     left_join(b_i, by="movieId") %>%
      group_by(userId) %>%
      summarize(b_u = sum(rating - mu - b_i) / (lambda + n()))

   # Calculate prediction ratings on test dataset
   pred <- val %>%
      left_join(b_i, by='movieId') %>%
      left_join(b_u, by='userId') %>%
      mutate(pred = mu + b_i + b_u) %>% .$pred

   # Retrieve the RMSE from the test set
   return( f_RMSE(val$rating, pred) )
})

# Best Lambda
lambdas[ which.min(rmses) ]
## [1] 3.7

# Min RMSE
m5_rmse <- min(rmses)
m5_rmse
## [1] 0.8648

# Save improvements
improv <- round(100*(m4_rmse-m5_rmse)/m4_rmse, rnd_pct)
overall <- round(100*(m1_rmse-m5_rmse)/m1_rmse, rnd_pct)
```

where we see a marked improvement of 8.38% error reduction over the RMSE of 0.9439 achieved by the previous regulirized movie effects model and of 18.51% over the RMSE of 1.0612 achieved by the naive model.

So better than the previous models, and therefore add it to our RMSE tabs table:

```
result_tabs <- bind_rows(result_tabs, data.frame(Method="Naive + RME + RUE",
  RMSE=m5_rmse,
  Improvement=paste(overall, "%", sep="")))

# Arrange because RME model's RMSE is larger than ME+UE
result_tabs %>% knitr::kable()
```

Method	RMSE	Improvement
Naive	1.0612	NA
Naive + ME	0.9439	11.05%
Naive + ME + UE	0.8653	18.46%
Naive + RME	0.9439	11.05%
Naive + RME + RUE	0.8648	18.51%

Note: Improvements displayed in this table are in relation to the Naive method alone.

4 Results

Here are the orginized saved results of our predictive models:

```
# Arrange in case some model's RMSE is larger than a previous one
result_tabs %>% arrange(desc(RMSE)) %>% knitr::kable()
```

Method	RMSE	Improvement
Naive	1.0612	NA
Naive + ME	0.9439	11.05%
Naive + RME	0.9439	11.05%
Naive + ME + UE	0.8653	18.46%
Naive + RME + RUE	0.8648	18.51%

Note: Improvements displayed in this table are in relation to the Naive method alone.

Where it is easy to see that our residual errors were reduced as we improved on the models, except for the regularized movie effects model (which did not perform any better than its plain version, the movie effects model), as we went from the basic naive model to the more complex regularized movie and user effects model.

We achieved an RMSE of 0.8648, which would have not been enough to have us win the challenge, but low enough to have us included in the top 10 list. Althogh we did not beat the winning RMSE of 0.8567, we must say to our credit that in their solution, the Netflix challenge winning team described using gradient boosted decision trees to combine over 500 models, compared to our simple 5 models. (see Chen.)

5 Conclusion

In this study we used a stepwise refinement process on the predictive models that were proposed for exploration in the implementation of a simple, yet optimal, system to predict a movie rating for rating users. We built, tested, and improved the models as we went along, from the naive model, throgh the inclusion of movie effects, movie and user effects, regulirized movie effects, and up to the inclusion of both the regulirized movie and user effects.

It is worth mentioning that many other predictive methods, such as KNN, Random Forests, Matrix Factorization, Singular Value Decomposition (SVD), Principal Components Analysis (PCA), among others, could have been used to minimize the residuals even further, though exploration of these methods was prohibitive considering the amount of time reserved for this study.

I was surprised by the fact that the regularized movie effects model did not offer any improvement over the more simplemovie effects model version, and of how little improvement the regularized user effects model offered over its simple counterpart version.

Finally, as we may have noticed from the inspection and analysis section of this study, the *movielens* dataset contains many other features that may have been worth exploring as potential predictors, such as the movie release year, the date of rating, and the movie genre, that may have served to reduce the remaining residuals even lower. However, again because time constraints, we limited this study to the inclusion of the *movieId* and *userId* predictors alone.