# MovieLens Project Submission

*Juan Pablo Mendoza*

*20 de mayo de 2019*

Greetings dear reviewer. The following is the code I've submitted for the MovieLens Project Submission. The code is divided into two different sections. The first section covers the code provided by the course's staff in the Edx platform.

The second section contains the code I've used looking to diminish the RMSE function's result. I must admit of course that I have not created that code since I used it from the Machine Learning's course content. I do not know if we are supposed to use only certain percentage of it, and in all honesty I did not find a section in which the course covers that thought.

I'm a beginner in R so there may of course be some mistakes in here, it's not a greatly added-value code either from what was seen in the ML course either so I am of course opened to commentary. I will try to clarify each step of the code, as I believe it's the least I can do.

As you may have seen from my user name, English is not my first language so I hope you may excuse me for any language mistakes along the way.

I. Code provided by the staff that downloads and adjusts the 10M MovieLens data set into new "Edx" object:

Installing corresponding packages required to run the code:

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

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

```
## Warning: package 'tidyverse' was built under R version 3.5.3
```

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

```
## v ggplot2 3.1.1       v purrr   0.3.2
## v tibble  2.1.1       v dplyr   0.8.0.1
## v tidyr   0.8.3       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0
```

```
## Warning: package 'ggplot2' was built under R version 3.5.3
```

```
## Warning: package 'tibble' was built under R version 3.5.3
```

```
## Warning: package 'tidyr' was built under R version 3.5.3
```

```
## Warning: package 'readr' was built under R version 3.5.3
```

```
## Warning: package 'purrr' was built under R version 3.5.3
```

```
## Warning: package 'dplyr' was built under R version 3.5.3
```

```
## Warning: package 'stringr' was built under R version 3.5.3
```

```
## Warning: package 'forcats' was built under R version 3.5.3

## -- Conflicts ---------------------------------------------------------------------------- tidyvers
## 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

## Warning: package 'caret' was built under R version 3.5.3

## Loading required package: lattice

##
## Attaching package: 'caret'

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

```
dl <- tempfile()
```

Downloading 10M MovieLens data set grom grouplens.org

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

Replacing string vectors and adding column names

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

Splitting up the text and generating the "Edx" object matrix

```
movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))
movielens <- left_join(ratings, movies, by = "movieId")
```

Creating Validation set with 10% of MovieLens data to make sure that the same userId and movieId are in both sets

```
set.seed(1) # //I have R 3.5.2 so I didn't use the following instructions provided in the code: "if
#using R 3.6.0: 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,]
```

Making sure userId and movieId in validation set are also in edx set

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

Adding rows removed from validation set back into edx set

```
removed <- anti_join(temp, validation)
```

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```
edx <- rbind(edx, removed)
```

Deleting the following objects, as we already have the cleaned "Edx" platform to work with

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

End of Edx's code.

II. Proceeding with method shown in the Machine Learning Course:

Partitioning the Edx data set in order to check predictions. Used a different seed so it's not the same as in the edx's code Given that it's a considerable amount of data to work with, I figured that assigning 60% of the data to the train set would be reasonable trainingwise.

```
set.seed(2)
index <- createDataPartition(y = edx$rating, times = 1, p = 0.4, list = FALSE)
training_set <- edx[-index,]
testing_set <- edx[index,]
```

Making sure that the same movieId and userId data appear in both sets

```
testing_set <- testing_set %>%
  semi_join(training_set, by = "movieId") %>%
  semi_join(training_set, by = "userId")
```

Creating RMSE checking Function

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

Genereting average of ratings for all movies, as it is the first parameter over which to model

```
mu <- mean(training_set$rating)
```

Modeling movie effects. In this part we will establish the "b_i" parameter, which will help us take into account how it is perceived in general over the mean performance of the movies in the data set.

Here we generate a new column called "b_i" in which we take an average of the substraction between all of each movie's ratings, and the absolute average for all ratings in the data set (mu)

```
movie_avgs <- training_set %>% group_by(movieId) %>% summarize(b_i = mean(rating - mu))
```

We willl create an object called "predicted_ratings" in which we will add the values of mu and the b_i column

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

The following is the first model in which we will use the RMSE function, using the list of values contained in predicted_ratings (the sum of mu and b_i) and the testing_set, which has over 40% of the data set content.

```
model_1_rmse <- RMSE(predicted_ratings, testing_set$rating)
```

We see that it doesn't show a great result. There is a great amount of disparity between the real ratings and the ones we predicted, which is understandable since it has few parameters.

```
model_1_rmse
```

```
## [1] 0.9435101
```

This is just a tibble we will use to keep track of our results for each model and see the results of each.

```
rmse_results <- data_frame(method = "Movie Effect Model",
                                    RMSE= model_1_rmse)
```

```
## Warning: `data_frame()` is deprecated, use `tibble()`.
## This warning is displayed once per session.
```

User effects. Since it is only natural to think that it is very hard to satisfy some people with movies, while others are more opened and easily enjoy films, it makes sense to think that some users will have a tendency to rate movies lowly, while others would frequently give high ratings. We will then take into account the "b_u" parameter which will be given to each userId as a way to assess that user's general rating tendency

```
user_avgs <- training_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%
  # //Here we will create the b_u column, thus assigning a b_u value to each userId.
  summarize(b_u = mean(rating - mu - b_i)) # //Each b_u value is computed this way since it shows
#the distance between each user's own perception (rating) and the usual performance of the movie
#(mu and b_i)
```

Now we will adjust the new parameter b_u with our previous predicted_ratings object, making it more comprehensible now taking into mind each user's tendency to rate

```
predicted_ratings <- testing_set %>% # //We will put the new parameters into the testing set since we
  #will compare them to the actual ratings of said test, and check the average disparity using the RMSE
  #function
  left_join(movie_avgs, by='movieId') %>% # // Here we enable the usage of the b_i parameter into the
  #testing set
  left_join(user_avgs, by='userId') %>% # // Here we enable the usage of the b_u parameter into the
  #testing set
  mutate(pred = mu + b_i + b_u) %>% # // We will create a new column called "pred", which will be the
  #sum of the three previous parameters, and our average guess of what each user would rate
  pull(pred) # // Here we take out all the values of the pred column
```

```
model_2_rmse <- RMSE(predicted_ratings, testing_set$rating) # //We will now compare the results from
#the pred column with the actual ratings
```

```
model_2_rmse # //We can see that we already achieve a good estimate according to the course's
```

```
## [1] 0.8675811
```

```
#criteria!! I can't really say if I lucked out in here, perhaps the seed(2) had something to do. I
#don't know if this can be considered enough, so I figured I should continue using the method in the
#ML's course for my submission not to be poorly.
```

Now we will add our results to the previous tibble to compare it with the preceding result.

```
rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Movie + User Effects Model",
                                     RMSE = model_2_rmse))

rmse_results
```

```
## # A tibble: 2 x 2
##   method                     RMSE
##   <chr>                     <dbl>
## 1 Movie Effect Model        0.944
## 2 Movie + User Effects Model 0.868
```

Penalized Least Squares. We have done well but there is still more things to consider in order to achieve a realistic approach. You see, there may be some movies that were not popular at all, and thus ratedby only a minimal amomunt of people.

Now it also makes sense to think that if these people watched some very "specific" (meaning not popular at all) movies, it's probably because there is a really particular taste for them, and we may be thinking of a niche here. So, some movies will be rate greatly by just a tiny amount of people, making their average rating very high, which our code so far would consider to be a good movie, but since they are only for a niche, the majority of the public would not want them to be recommended. So we have to do something about it. A certain form of balance is required

We will use the lambda value to try to add a certain minimal amount of users over which to average the rating of a movie, since it is fair to say that if a considerable amount of people perceive a movie as good, then it is possibly a good movie indeed. However, we don't know which lambda value we should use, so we are gonna have to test several of them and pick the one that gives the lowest RMSE.

The ML course provides some steps before evaluating different lambda values, but they are mostly for theoretical illustrative reasons but since surely you, dear reviewer, have gone through that course as it would just be in the way of picking the actual valuable lambda, so let's proceed straight to it.

```
lambdas <- seq(0, 10, 0.25) # //Here we define a different set of lambda values
rmses <- sapply(lambdas, function(l){ # //We will now test each one along with our previous
  #parameters using the sapply function
  mu <- mean(training_set$rating) # //Nothing new here, but we must put some values into
  #the function
  b_i <- training_set %>%
    group_by(movieId) %>%
```

```
    summarize(b_i = sum(rating - mu)/(n()+l)) # //We will now implement each lambda value
  #and its effect on b_i. We will use each lambda value as a minimum value to devide with
  #when making an average. Same idea as in line 141 and 142.
  b_u <- training_set %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l)) # //We are now using the same procedure
  #with the b_u parameter.
  predicted_ratings <- # // We are now implementing the new "lambdafied" b_i and b_u parameters
    testing_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>% # // We now create a new pred column with the new
    #modified parameters.
    pull(pred) # //We take out the values of the pred column
  return(RMSE(predicted_ratings, testing_set$rating)) # //And we now compare them with the
  #actual ratings and return each result
})

lambda <- lambdas[which.min(rmses)] # //We now want to check which lambda value gave out the
#lowest RMSE

lambda
```

```
## [1] 4.75
```

Now we add it to the table to compare each method

```
rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Regularized Movie + User Effect Model",
                                     RMSE = min(rmses)))
```

And now we make an even fancier table, for fancy reasons I guess.

```
rmse_results %>% knitr::kable()
```

| method | RMSE |
|---|---|
| Movie Effect Model | 0.9435101 |
| Movie + User Effects Model | 0.8675811 |
| Regularized Movie + User Effect Model | 0.8664628 |

As we have seen our final approach is the one most effective one, which makes sense since it has more "balanced" parameters. Surely there may be some even finer approaches out there, but that is beyond the scope of the course, and of my knowledge I must admit.

Thank you dear reviewer for checking my submission. I know it is not a brightly-differentiated approach from what was seen on the course but oh well, I tried to keep it efficient and add value in the only ways I could come out with. Any commentary will be taken into consideration. Thank you for your time and best of lucks for you too on the Capstone course, may we all make the best of it!