MovieLens Report

Jose Leonardo Ribeiro Nascimento

28/10/2019

1 INTRODUCTION

1.1 Overview

Harvard University offers, through edx.org platform, a *Professional Certificate in Data Science*. In this program, it is possible to learn:

- Fundamental R programming skills
- Statistical concepts such as probability, inference, and modeling and how to apply them in practice
- Gain experience with the tidyverse, including data visualization with ggplot2 and data wrangling with dplyr
- Become familiar with essential tools for practicing data scientists such as Unix/Linux, git and GitHub, and RStudio
- Implement machine learning algorithms
- In-depth knowledge of fundamental data science concepts through motivating real-world case studies

The program includes nine courses, beginning with R Basics, visualization and probability, going through inference and modeling, wrangling data, linear regression and culminating with machine learning. As a final course, **Data Science: Capstone** presents two challenges, being the first one the object of this report: **to create a movie recommendation system using the MovieLens dataset**. To compare different models or to see how well I'm doing compared to a baseline, I will use Residual Mean Squared Error (RMSE) as my loss function.

1.2 MovieLens dataset

GroupLens Research (http://grouplens.org) has collected and made available rating data sets from the MovieLens web site (http://movielens.org). There are several data sets, from huge ones, with up to 27M movie ratings and more than 1M tag applications, to small ones, with "only" 100m movie ratings and 3,6m tag applications. In this project, I will use an older dataset: MovieLens 10M Dataset (https://grouplens.org/datasets/movielens/10m/). It is a stable benchmark dataset, released in 1/2009, with 10 million ratings and 100,000 tag applications applied to 10,000 movies by 72,000 users. The dataset README.txt points more useful information:

```
"This data set contains 10000054 ratings and 95580 tags applied to 10681
movies by 71567 users of the online movie recommender service MovieLens.
Users were selected at random for inclusion. All users selected had rated
at least 20 movies. Unlike previous MovieLens data sets, no demographic i
nformation is included. Each user is represented by an id, and no other i
nformation is provided."
```

1.3 Goal

As explained in **Data Science: Machine Learning Course, Section 6, 6.2: Recommendation Systems**, these are "more complicated machine learning challenges because each outcome has a different set of predictors". According to Prof. Rafael Irizarry's Introduction to Data Science book, "recommendation systems use ratings that users have given items to make specific recommendations". In this project, I'm supposed to create a recommendation system somewhat like the one Netflix uses. This system, therefore, will be able to predict how many stars a user will give a specific movie. In MovieLens data, ratings are made on a 5-star scale, with half-star increments. One star suggests it is not a good movie, whereas five stars suggests it is an excellent movie. Movies for which a high rating is predicted for a given user are then recommended to that user. I took two factors into consideration when deciding which approach to choose: 1) This is my first recommendation system project, so I must be conservative and use what I've learned through the course. 2) MovieLens dataset size makes it impossible to me (considering my hardware and my limited knowledge of machine learning and R language) to test some approaches, like Linear Models, k-nearest neighbors (kNN) or Random Forests. Thus, I've decided to develop my project based on the steps described in the Recommendation Systems chapter of the Machine Learning Course. Since the example provided in that chapter works with only two predictors, based on movies and users, my goal was to implement two other predictors, in order to reach better results for RMSE. I will explain the approach in detail in the **Methods** section.

---

2 METHODS

2.1 Generating Train and Validation Sets

The first step is to generate Train (edx) and Validation Set, using the initial code provided in the Section "**Project Overview: MovieLens -> Create Train and Validation Sets**". Algorithm must be developed using *edx* set, whereas *validation* set will be used only for a final test of the algorithm. The initial code is the following:

```
## Loading required package: tidyverse

## -- Attaching packages ------------------------------- tidyverse 1.2.1
--
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

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

## Loading required package: caret

## Loading required package: lattice

##
## Attaching package: 'caret'

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

## 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

## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Roundin
g'
## sampler used

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

2.2 Information about datasets

Before beginning to think about a recommendation system, let's examine our datasets. Edx and Validation are both data.frames which share the same structure. Using the following code we can see edx has 9,000,055 rows and 6 columns and validation has 999,999 rows and 6 columns:

```
#This function returns dimensions (number of rows and number of columns)
of a data.frame.

dim(edx)
```

```
## [1] 9000055       6
```

```
dim(validation)
```

```
## [1] 999999      6
```

Both edx and validation have columns named userId, movieId, rating, timestamp, title and genres.

```
#Head returns the first part of an object.
```

```
head(edx)
```

```
##   userId movieId rating timestamp                         title
## 1      1     122      5 838985046              Boomerang (1992)
## 2      1     185      5 838983525               Net, The (1995)
## 4      1     292      5 838983421               Outbreak (1995)
## 5      1     316      5 838983392               Stargate (1994)
## 6      1     329      5 838983392 Star Trek: Generations (1994)
## 7      1     355      5 838984474       Flintstones, The (1994)
##                           genres
## 1                 Comedy|Romance
## 2          Action|Crime|Thriller
## 4  Action|Drama|Sci-Fi|Thriller
## 5       Action|Adventure|Sci-Fi
## 6 Action|Adventure|Drama|Sci-Fi
## 7       Children|Comedy|Fantasy
```

```
head(validation)
```

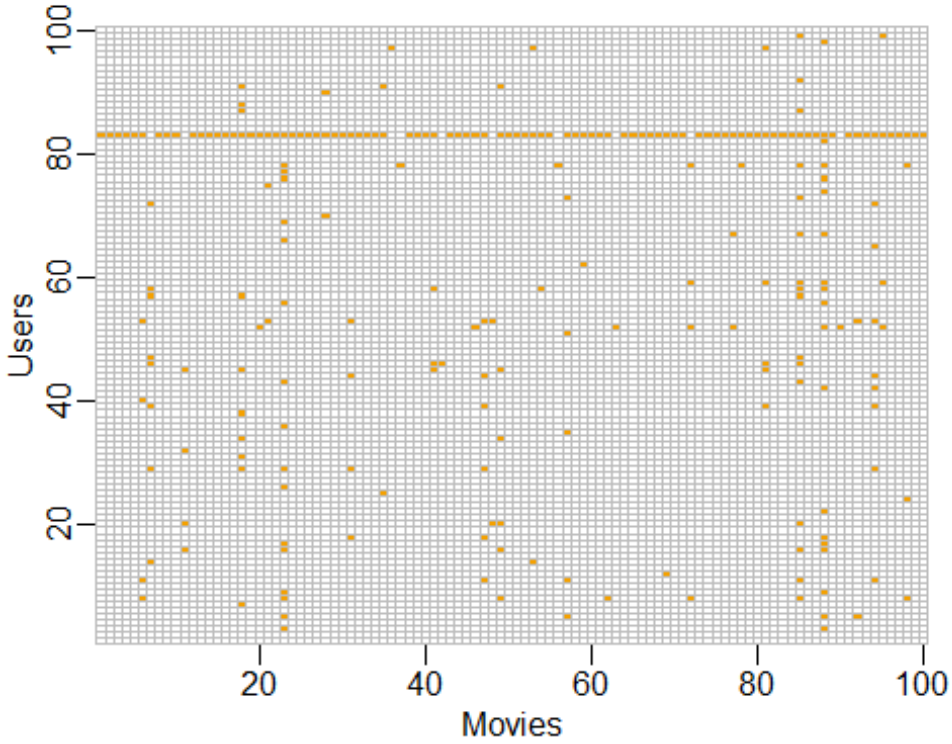
```
##   userId movieId rating timestamp
## 1      1     231      5 838983392
## 2      1     480      5 838983653
## 3      1     586      5 838984068
## 4      2     151      3 868246450
## 5      2     858      2 868245645
## 6      2    1544      3 868245920
##                                                  title
## 1                                     Dumb & Dumber (1994)
## 2                                     Jurassic Park (1993)
## 3                                        Home Alone (1990)
## 4                                           Rob Roy (1995)
## 5                                     Godfather, The (1972)
## 6 Lost World: Jurassic Park, The (Jurassic Park 2) (1997)
##                                     genres
## 1                                   Comedy
## 2        Action|Adventure|Sci-Fi|Thriller
## 3                          Children|Comedy
## 4                 Action|Drama|Romance|War
## 5                              Crime|Drama
## 6 Action|Adventure|Horror|Sci-Fi|Thriller
```

Although edx set has 9M movie ratings, using the code below we can see that there are 69,878 unique users and 10,677 unique movies.

```
#This code returns how many unique users and unique movies are in edx set
.

edx %>%
  summarise(n_users = n_distinct(userId),
      n_movies = n_distinct(movieId))

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

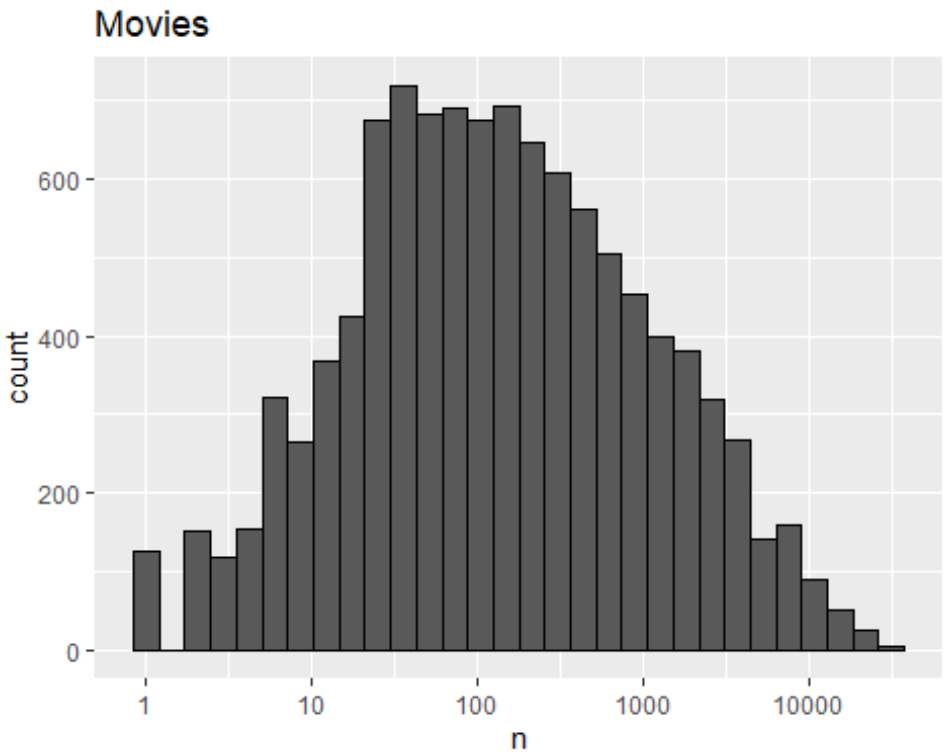
If we multiply these two numbers, we get a number greater than 746 million, what implies that not every user rated every movie. Of course, we shouldn't expect that every user watched more than ten thousand movies. The code below, which generates an image, gives us an idea of how sparse the distribution of the users and movies ratings is:

```
#This code returns a matrix for a random sample of 100 movies and 100 use
rs, with an image with yellow indicating a user movie combination for whi
ch we have a rating.

users <- sample(unique(edx$userId), 100)
rafalib::mypar()
edx %>% 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")
abline(h=0:100+0.5, v=0:100+0.5, col = "grey")
```

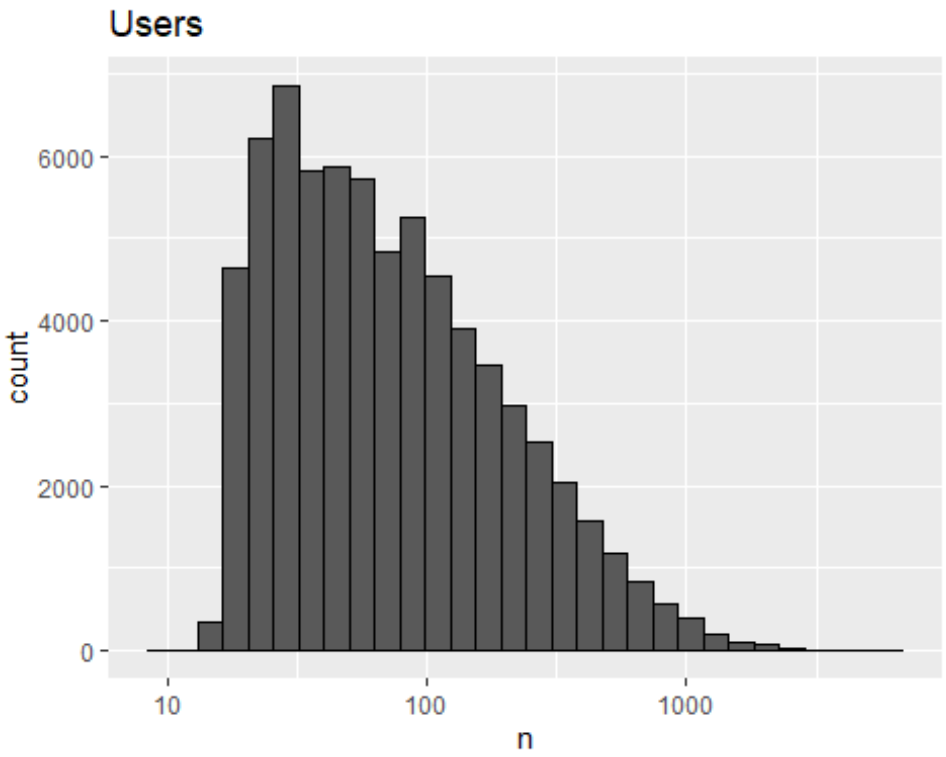


In the image above, yellow indicates a user movie combination for which we have a rating. We can think of the task of recommendation system as filling in the NA's (the grey spots) in the image. In order to predict movies, it's essential to know that some movies get more reviews than others and some users review more movies than others, as shown by the following codes and plots:

```
#This code groups edx dataset by movieId and returns a graph showing the
distribution of movie rates by movie.

edx %>%
   dplyr::count(movieId) %>%
   ggplot(aes(n)) +
   geom_histogram(bins = 30, color = "black") +
   scale_x_log10() +
   ggtitle("Movies")
```




```
#This code groups edx dataset by userId and returns a graph showing the d
istribution of movie rates by user.

edx %>%
   dplyr::count(userId) %>%
   ggplot(aes(n)) +
   geom_histogram(bins = 30, color = "black") +
   scale_x_log10() +
   ggtitle("Users")
```



In my algorithm, I must consider the fact that there are blockbusters and obscure movies and users who sees a lot of movies while others are “occasional movie reviewers”. Later I will evaluate how this can cause impact on my predictions. Now I know the basic about my dataset, I will split edx set into train and test set:

```
#In order to test the predictors, I've decided to split the EDX set into
train_set and test_set, using the following code:


#First, I must load caret package, which includes the tools for data spli
tting.

library(caret)


#Then I set the seed to 1:

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

## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Roundin
g'
## sampler used

#Finally, I split the edx set, with 80% of the data to train_set and 20%
for test_set:
```

```
test_index <- createDataPartition(y = edx$rating, times = 1,
            p = 0.2, list = FALSE)
train_set <- edx[-test_index,]
test_set <- edx[test_index,]
```

To make sure we don't include users and movies in the test set that do not appear in the train set, I removed these using the semi_join function:

```
#To make sure we don't include users and movies in the test_set that do n
ot appear in the training set, I removed these using the semi_join functi
on:

test_set <- test_set %>%
   semi_join(train_set, by = "movieId") %>%
   semi_join(train_set, by = "userId")
```

To compare the different models and to see how well we're doing compared to some baseline, we need a loss function. In this project, we're required to use **residual mean squared error (RMSE)**. As stated at Prof. Rafael Irizarry's Introduction to Data Science book, "if N is the number of user-movie combinations, yu,i is the rating for movie i by user u, and y^u,i is our prediction, then RMSE is defined as follows:

```
#This function computes the residual mean squared error for a vector of r
atings and their corresponding predictors.

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

2.3 Starting our model

The first model I will try assumes the same rating for all movies and users, with all the differences explained by random variation. Thus, our equation will be this one:

$$Yu,i = \mu + \epsilon u,i$$

In this case, μ represents the true rating for all movies and users and ϵ represents independent errors sampled from the same distribution centered at zero. The estimate that minimizes the root mean squared error is simply the average rating of all movies across all users, which can be computed like this:

```
#This function returns the average rating of all movies across all users
in the train set.

mu_hat <- mean(train_set$rating)
mu_hat

## [1] 3.512482
```

Now that I computed the average on the training data, I will predict all unknown ratings with the average and then compute the residual mean squared error on the test set data.

```
#This function returns the RMSE of the first model applied on the test se
t data.

naive_rmse <- RMSE(test_set$rating, mu_hat)
naive_rmse

## [1] 1.059904
```

We can see that with *RMSE = 1.059904*, I'm far from the desired RMSE, which is, for the purpose of obtain max score, lower than *0.8649*. Since I'm going to improve my model, I will create a table that's going to store the results that I obtain as I go along.

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

The next step considers the fact that some movies are generally rated higher than others. This implies that I'll have to add the term bi, that represents the average rating for movie i, to my model:

$Yu, i = \mu + bi + \epsilon u, i$

One way to estimate this is using the *lm* function, like this:

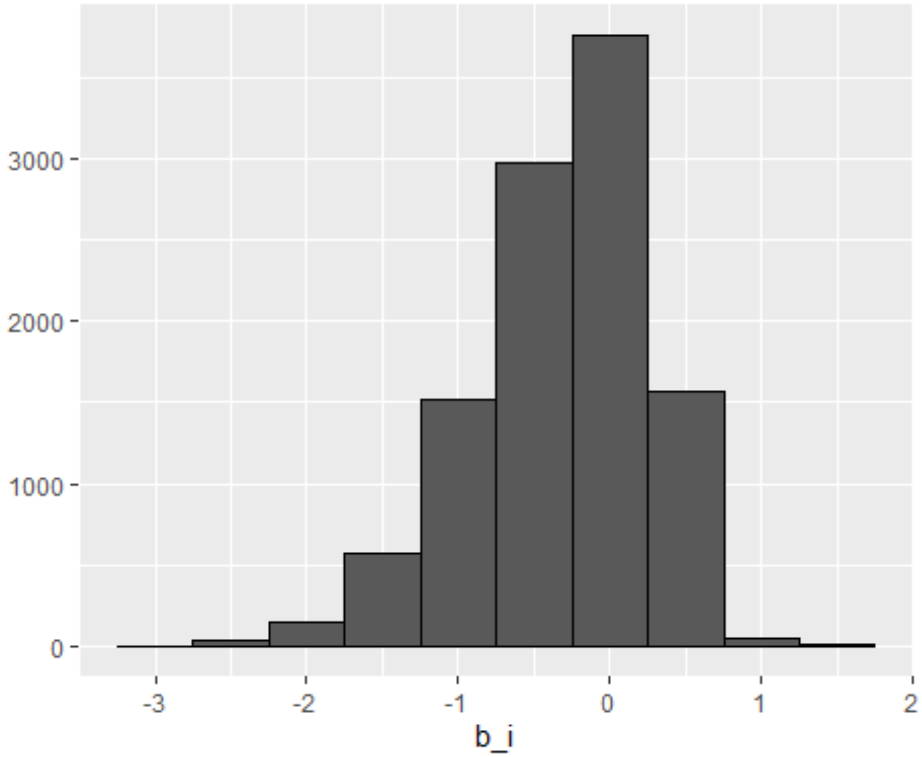
```
#fit <- lm(rating ~ as.factor(MovieId), data = train_set)
```

However, since there are thousands of b's, each movie gets one parameter, what would cause the *lm* function get very slow, possibly crashing R. In this specific case, though, the least squares estimate, bi, is just the average of Yu,i minus the overall mean for each movie, i. So, we can use this function instead of *lm*:

```
#This code first calculates average rating for every movie and then calcu
lates b_i, which is the average of Y_u_i minus the overall mean for each
movie

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

We can see, plotting the estimates, that the rates for movies vary substantially. Some movies are good, others are bad. Since the overall average is about 3.5, a b_i of 1.5 implies that a movie received a 5-star rating.

```
#This code plots the b_i effect.

movie_avgs %>% qplot(b_i, geom ="histogram", bins = 10, data = ., color =
I("black"))
```



Now I've included b_i in my model, let's see how my prediction improves:

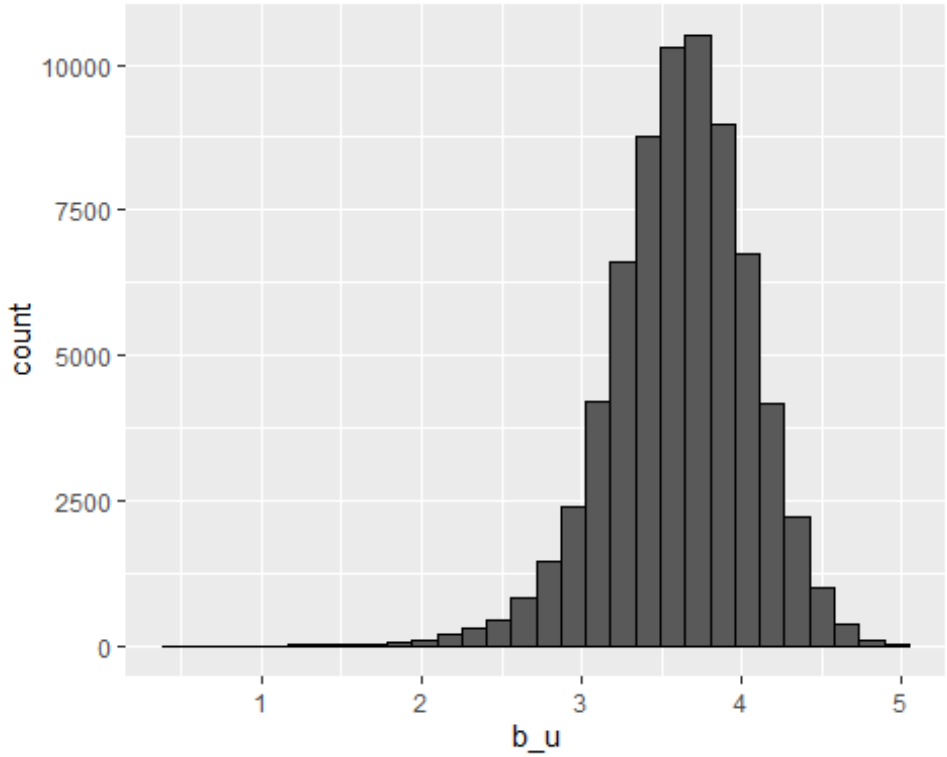
```
#This code applies the model with the b_i effect to test set

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

#Calculates the RMSE for the model

model_1_rmse <- RMSE(predicted_ratings, test_set$rating)

#Add the result of the new model to the rmse_results table
rmse_results <- bind_rows(rmse_results,
          data_frame(method="Movie Effect Model",
             RMSE = model_1_rmse ))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429

2.4 Applying user effect

We can see a good improvement, but I'm still far from the goal. Let's see how we can improve the model. The next step is a little bit intuitive: as movies vary from great to bad, users vary in their preferences: some users tend to be more critic, giving lower rating, while others love all the movies they watch, giving high ratings. We can see this using the following code:

```
#This code computes the average rating for users who have rated over than
100 movies and make a histogram of those values

train_set %>%
   group_by(userId) %>%
   summarize(b_u = mean(rating)) %>%
   filter(n()>=100) %>%
   ggplot(aes(b_u)) +
   geom_histogram(bins = 30, color = "black")
```



Considering the differences among the users, we can include the term *bu*, which refers to user-specific effect, in our model.

```
                **$Yu,i=μ+bi+bu+ϵu,i$**
```

To add *bu*, once more we can't use lm function, since this would crash R. Similarly to *bi*, now we can simply compute the average of *Yu,i* minus the overall mean for each movie, *i*, minus *bi*:

```
#This code calculates b_u, which is the average of Y_u_i minus the overal
l mean for each movie, minus b_i


user_avgs <- train_set %>%
   left_join(movie_avgs, by='movieId') %>%
   group_by(userId) %>%
   summarize(b_u = mean(rating - mu - b_i))
```

Now we apply the new model, with *bu*, to the test set, and then calculate the RMSE, to see if our model results are better than the previous one.

```
#This code applies the model with the b_u effect to test set

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

#Calculates the RMSE for the model

model_2_rmse <- RMSE(predicted_ratings, test_set$rating)

#Add the result of the new model to the rmse_results table
rmse_results <- bind_rows(rmse_results,
          data_frame(method="Movie + User Effects Model",
             RMSE = model_2_rmse ))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320

Now RMSE result (0.8659320) is within the score reference, but it is still too high, since with this result I would get 10 points out of 25.

2.5 Regularization

The next step, according to **Data Science: Machine Learning Course, Section 6, 6.3**, is to use regularization to improve results. Regularization constrains the total variability of the effect sizes by penalizing large estimates that come from small sample sizes. We can use regularization for movie and user effect. We will do this one

at a time. In order to see what exactly regularization does, let's see our top ten movies and the ten worst movies according to our first model. Top ten movies:

```
#This code creates a simple table with movieId and title, without repetit
ion

movie_titles <- edx %>%
  select(movieId, title) %>%
  distinct()

#Shows the top 10 movies ordered by rating, according to the first model,
which only considers movie effects

movie_avgs %>% left_join(movie_titles, by="movieId") %>%
  arrange(desc(b_i)) %>%
  select(title, b_i) %>%
  slice(1:10) %>%
  knitr::kable()
```

title	b_i
Hellhounds on My Trail (1999)	1.487518
Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) (1980)	1.487518
Satan's Tango (SÃ¡tÃ¡ntangÃ³) (1994)	1.487518
Shadows of Forgotten Ancestors (1964)	1.487518
Money (Argent, L') (1983)	1.487518
Fighting Elegy (Kenka erejii) (1966)	1.487518
Sun Alley (Sonnenallee) (1999)	1.487518
Aerial, The (La Antena) (2007)	1.487518
Blue Light, The (Das Blaue Licht) (1932)	1.487518
More (1998)	1.404184

Worst ten movies:

```
#Shows the worst 10 movies ordered by rating, according to the first mode
l, which only considers movie effects

movie_avgs %>% left_join(movie_titles, by="movieId") %>%
  arrange(b_i) %>%
  select(title, b_i) %>%
  slice(1:10) %>%
  knitr::kable()
```

title	b_i
Besotted (2001)	-3.012482

Confessions of a Superhero (2007)	-3.012482
War of the Worlds 2: The Next Wave (2008)	-3.012482
SuperBabies: Baby Geniuses 2 (2004)	-2.749982
From Justin to Kelly (2003)	-2.667244
Legion of the Dead (2000)	-2.637482
Disaster Movie (2008)	-2.637482
Hip Hop Witch, Da (2000)	-2.603391
Criminals (1996)	-2.512482
Mountain Eagle, The (1926)	-2.512482

We can see that both top and worst ten movies have something in common: all of them are quite obscure movies. We can run the following code to check how often they were rated:

```
#This code returns the top ten movies, but adding number of reviews for e
ach movie

train_set %>% dplyr::count(movieId) %>%
   left_join(movie_avgs) %>%
   left_join(movie_titles, by="movieId") %>%
   arrange(desc(b_i)) %>%
   select(title, b_i, n) %>%
   slice(1:10) %>%
   knitr::kable()

## Joining, by = "movieId"
```

title	b_i	n
Hellhounds on My Trail (1999)	1.487518	1
Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) (1980)	1.487518	3
Satan's Tango (SÃ¡tÃ¡ntangÃ³) (1994)	1.487518	2
Shadows of Forgotten Ancestors (1964)	1.487518	1
Money (Argent, L') (1983)	1.487518	1
Fighting Elegy (Kenka erejii) (1966)	1.487518	1
Sun Alley (Sonnenallee) (1999)	1.487518	1
Aerial, The (La Antena) (2007)	1.487518	1
Blue Light, The (Das Blaue Licht) (1932)	1.487518	1
More (1998)	1.404184	6

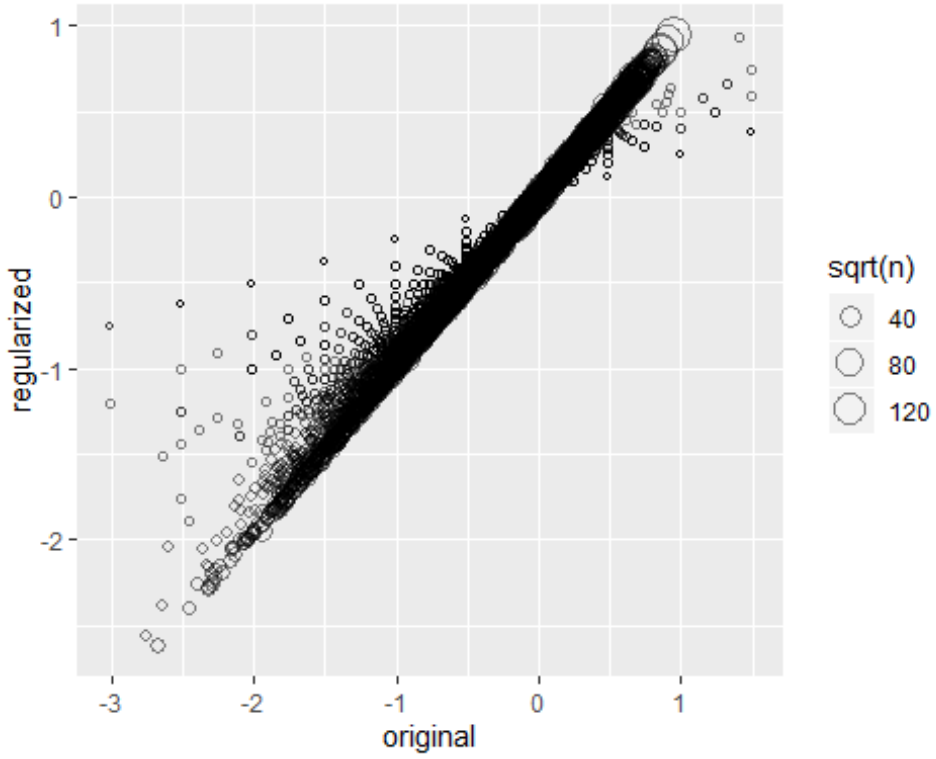
```
#This code returns the worst ten movies, but adding number of reviews for
each movie

train_set %>% dplyr::count(movieId) %>%
```

```
     left_join(movie_avgs) %>%
     left_join(movie_titles, by="movieId") %>%
     arrange(b_i) %>%
     select(title, b_i, n) %>%
     slice(1:10) %>%
     knitr::kable()

## Joining, by = "movieId"
```

title	b_i	n
Besotted (2001)	-3.012482	1
Confessions of a Superhero (2007)	-3.012482	1
War of the Worlds 2: The Next Wave (2008)	-3.012482	2
SuperBabies: Baby Geniuses 2 (2004)	-2.749982	40
From Justin to Kelly (2003)	-2.667244	168
Legion of the Dead (2000)	-2.637482	4
Disaster Movie (2008)	-2.637482	28
Hip Hop Witch, Da (2000)	-2.603391	11
Criminals (1996)	-2.512482	1
Mountain Eagle, The (1926)	-2.512482	1

We can see that the supposed best and worst movies were rated by very few users, in most cases just one. Regularization permit us to penalize large estimates that come from small sample sizes. It works as a tuning parameter, which we will call **λ** (lambda). To apply **λ**, the code is the following:

```
#This code apply regularization to the first model, considering lambda va
lue as 3, which is already the optimized value

lambda <- 3
mu <- mean(train_set$rating)
movie_reg_avgs <- train_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+lambda), n_i = n())

#This code plots original estimates vs regularized estimates, showing tha
t as ni as small, values are shrinking more towards zero

data_frame(original = movie_avgs$b_i,
           regularized = movie_reg_avgs$b_i,
           n = movie_reg_avgs$n_i) %>%
     ggplot(aes(original, regularized, size=sqrt(n))) +
     geom_point(shape=1, alpha=0.5)
```



We can see in the plot that when n is small, the values are shrinking more towards zero, what represents the effect we want with regularization. Now we can check the top 10 movies based on the estimates we got when using regularization.

```
#Top 10 movies using regularization

train_set %>%
     dplyr::count(movieId) %>%
     left_join(movie_reg_avgs) %>%
     left_join(movie_titles, by="movieId") %>%
     arrange(desc(b_i)) %>%
     select(title, b_i, n) %>%
     slice(1:10) %>%
     knitr::kable()

## Joining, by = "movieId"
```

title	b_i	n
Shawshank Redemption, The (1994)	0.9447090	22363
More (1998)	0.9361230	6
Godfather, The (1972)	0.9048132	14107
Usual Suspects, The (1995)	0.8567890	17315
Schindler's List (1993)	0.8514402	18567
Rear Window (1954)	0.8086363	6325

Casablanca (1942)	0.8045763	9027
Double Indemnity (1944)	0.7970496	1711
Third Man, The (1949)	0.7960352	2420
Dark Knight, The (2008)	0.7957351	1900

```
#Worst 10 movies using regularization

train_set %>%
     dplyr::count(movieId) %>%
     left_join(movie_reg_avgs) %>%
     left_join(movie_titles, by="movieId") %>%
     arrange(b_i) %>%
     select(title, b_i, n) %>%
     slice(1:10) %>%
     knitr::kable()

## Joining, by = "movieId"
```

title	b_i	n
From Justin to Kelly (2003)	-2.620450	168
SuperBabies: Baby Geniuses 2 (2004)	-2.558123	40
PokÃ©mon Heroes (2003)	-2.396076	109
Disaster Movie (2008)	-2.382242	28
Glitter (2001)	-2.296561	282
Barney's Great Adventure (1998)	-2.281269	168
Gigli (2003)	-2.264318	249
Carnosaur 3: Primal Species (1996)	-2.261789	51
Pokemon 4 Ever (a.k.a. PokÃ©mon 4: The Movie) (2002)	-2.243257	168
Son of the Mask (2005)	-2.218303	128

Now we got two lists with movies more frequently rated. Therefore, we can expect our RMSE results will be better.

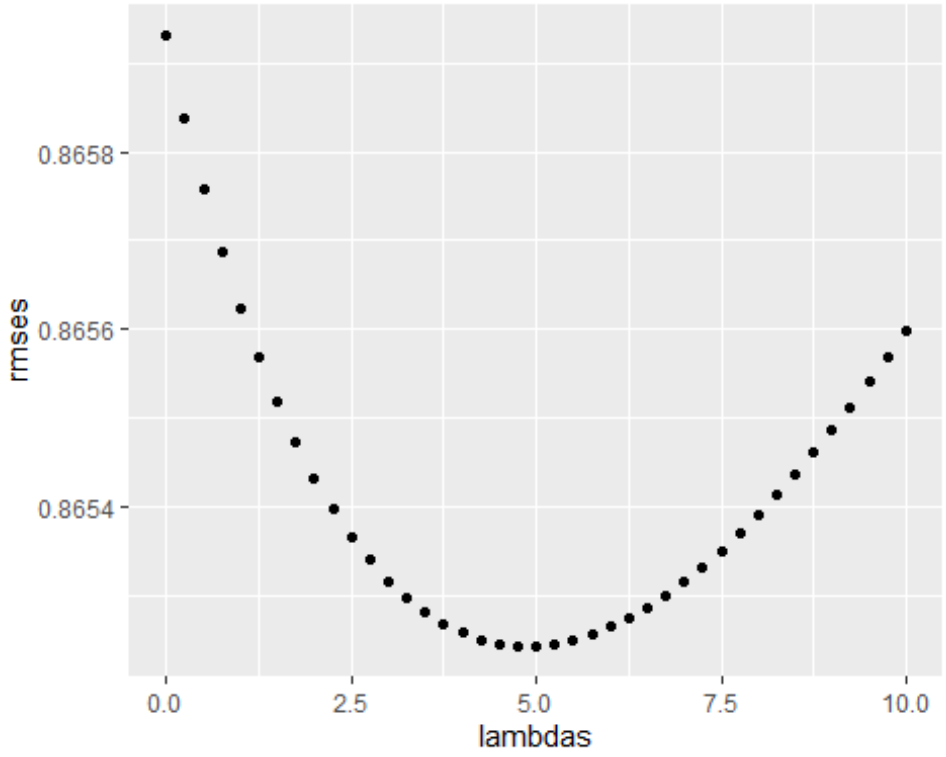
```
#This code applies the model with the b_i effect and regularization to te
st set

predicted_ratings <- test_set %>%
     left_join(movie_reg_avgs, by='movieId') %>%
     mutate(pred = mu + b_i) %>%
     .$pred

#Calculates the RMSE for the model

model_3_rmse <- RMSE(predicted_ratings, test_set$rating)

#Add the result of the new model to the rmse_results table
```

```
rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Regularized Movie Effect Mod
el",
                                     RMSE = model_3_rmse ))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320
Regularized Movie Effect Model	0.9436762

With RMSE of 0.9436762, we can see our result didn't improve that much. Let's see how we can do when implementing regularization on the user effect model:

```
#This code apply regularization to the second model (user effect), using
cross-validation to choose the best value for lambda

lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
     mu <- mean(train_set$rating)
     b_i <- train_set %>%
          group_by(movieId) %>%
          summarize(b_i = sum(rating - mu)/(n()+l))
     b_u <- train_set %>%
          left_join(b_i, by="movieId") %>%
          group_by(userId) %>%
          summarize(b_u = sum(rating - b_i - mu)/(n()+l))
     predicted_ratings <-
          test_set %>%
          left_join(b_i, by = "movieId") %>%
          left_join(b_u, by = "userId") %>%
          mutate(pred = mu + b_i + b_u) %>%
          .$pred
     return(RMSE(predicted_ratings, test_set$rating))
})

#Plot lambda and rmses value to demonstrate the best value for lambda

qplot(lambdas, rmses)
```




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

As demonstrated in the plot, the better value for lambda is the one which generates the smallest value for RMSE. In this case, 4.75. Now we add the best RMSE result to the table:

```
#Add the result of the new model to the rmse_results table

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Regularized Movie + User Eff
ect Model",
                                     RMSE = min(rmses)))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320
Regularized Movie Effect Model	0.9436762
Regularized Movie + User Effect Model	0.8652421

Working with the example from the Data Science: Machine Learning Course, we got RMSE of 0.8652, which is not enough to obtain the best score. It's time to improve my model by myself!

2.6 Genre effect

If movies vary one from another and users vary one from another, we can think the same about genres. One can prefer adventure and action movies rather than horror and musical. We can so imagine that ratings of this specific user for adventure and action movies tend to be higher than to horror and musical movies. MovieLens README file points that movies can be classified in the following genres:

1. Action
2. Adventure
3. Animation
4. Children's
5. Comedy
6. Crime
7. Documentary
8. Drama
9. Fantasy
10. Film-Noir
11. Horror
12. Musical
13. Mystery
14. Romance
15. Sci-Fi
16. Thriller
17. War
18. Western

Occurs that most movies are tagged with more than one genre. The following code shows that edx set has 797 distinct combination of genres:

```
n_distinct(edx$genres)
```

```
## [1] 797
```

As an example, I picked The Lion King, animation from 1994, which is classified as Adventure|Animation|Children|Drama|Musical. Although each one of this is a genre, for the purpose of my predictor, this combination counts as a unique genre. That's why there are 797 unique combination of genres, while there are only 18 "pure" genres. To add the genre predictor (bg), the procedure is like the bi and bu predictors. With bg, our model looks like this:

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

To calculate bg, we simply compute the average of Yu,i minus the overall mean for each movie, i, minus bi, minus bu:

```
#This code calculates b_g, which is the average of Y_u_i minus the overal
l mean for each movie, minus b_i, minus b_u


genre_avgs <- train_set %>%
left_join(movie_avgs, by='movieId') %>%
left_join(user_avgs, by = 'userId') %>%
group_by(genres) %>%
summarize(b_g = mean(rating - mu - b_i - b_u))
```

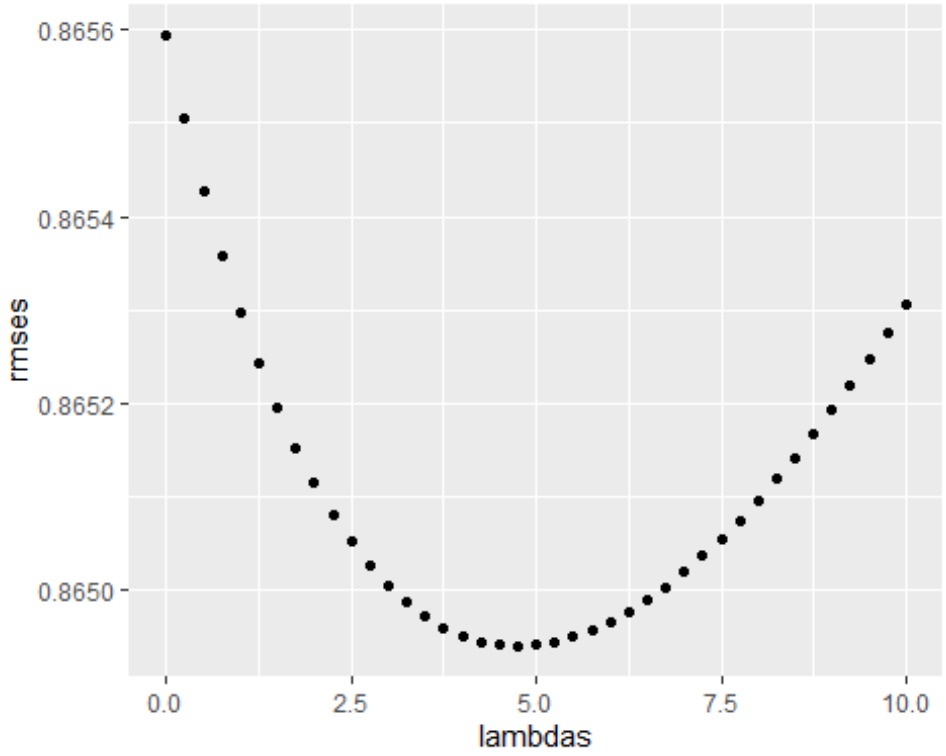
Now we apply the new model, with bg, to the test set, and then calculate the RMSE, to see if our model results are better than the previous one.

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320
Regularized Movie Effect Model	0.9436762
Regularized Movie + User Effect Model	0.8652421
Movie + User + Genre Effects Model	0.8655941

Now, in order to improve this model, I will regularize it:

```
#This code apply regularization to the third model (movie + user + genres
effect), using cross-validation to choose the best value for lambda

lambdas <- seq(0, 10, 0.25)

rmses <- sapply(lambdas, function(l){

mu <- mean(train_set$rating)

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

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

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

```
predicted_ratings <-
test_set %>%
left_join(b_i, by = "movieId") %>%
left_join(b_u, by = "userId") %>%
left_join(b_g, by = "genres") %>%
mutate(pred = mu + b_i + b_u + b_g) %>%
.$pred
return(RMSE(predicted_ratings, test_set$rating))
})

#Plot lambda and rmses value to demonstrate the best value for lambda

qplot(lambdas, rmses)
```




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

As demonstrated in the plot, the better value for lambda is the one which generates the smallest value for RMSE. In this case, 4.75. Now we add the best RMSE result to the table:

```
#Add the result of the new model to the rmse_results table

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Regularized Movie + User + G
enre Effect Model",
                                     RMSE = min(rmses)))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320
Regularized Movie Effect Model	0.9436762
Regularized Movie + User Effect Model	0.8652421
Movie + User + Genre Effects Model	0.8655941
Regularized Movie + User + Genre Effect Model	0.8649406

We can see now RMSE is within the second-best score, but I still want to improve it.

2.7 Year effect

I want to know if the movie ratings can vary depending on the year the movie was reviewed. Maybe if the movie was just released, people who were anxious waiting for it are more likely to give better grades. Or maybe there is another factor related to the consolidation of the movie. As time passes by, some movies reach the "classic" status, and people would tend to see that movie with more reverent eyes. The first step to do this analysis is to add the year column to test set and train set. This can be done using lubridate package to convert timestamp column and date and extract the year, as follows:

```
#Load lubridate package (install if needed with #install.packages("lubridate")

library(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 object is masked from 'package:base':
##
##     date

#Add date and year columns to train and test set using lubridate package command as_datetime

train_set<-mutate(train_set, date = as_datetime(timestamp),year=year(date))

test_set<-mutate(test_set, date = as_datetime(timestamp),year=year(date))
```

```
Now I will add the genre predictor (by) to our model, which will look lik
e this:
                    *Yu,i=μ+bi+bu+bg+by+ϵu,i*
To calculate by, we simply compute the average of Yu,i minus the overall
mean for each movie, i, minus bi, minus bu, minus bg:

#This code calculates b_y, which is the average of Y_u_i minus the overal
l mean for each movie, minus b_i, minus b_u, minus b_g


year_avgs <- train_set %>%
    left_join(movie_avgs, by='movieId') %>%
    left_join(user_avgs, by='userId') %>%
    left_join(genre_avgs, by='genres') %>%
    group_by(year) %>%
summarize(b_y = mean(rating - mu - b_i - b_u - b_g))
```

Now we apply the new model, with by, to the test set, and then calculate the RMSE, to see if our model results are better than the previous one.

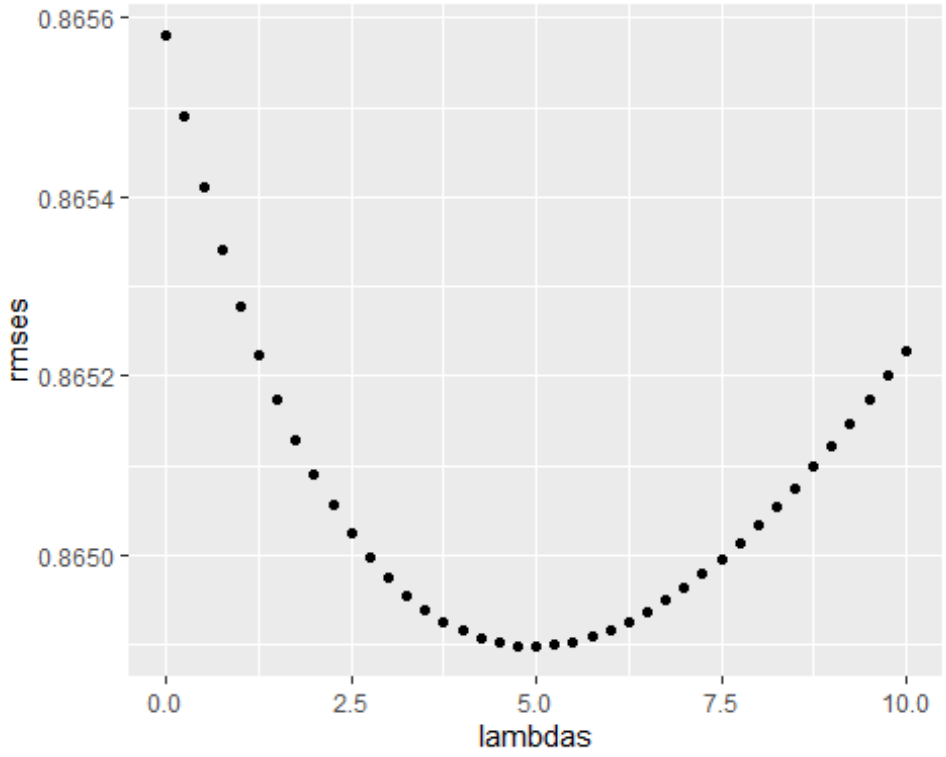
```
#This code applies the model with the b_y effect to test set

predicted_ratings <- test_set %>%
    left_join(movie_avgs, by='movieId') %>%
    left_join(user_avgs, by='userId') %>%
    left_join(genre_avgs, by='genres') %>%
    left_join(year_avgs, by='year') %>%
    mutate(pred = mu + b_i + b_u + b_g + b_y) %>%
    .$pred

#Calculates the RMSE for the model

model_5_rmse <- RMSE(predicted_ratings, test_set$rating)

#Add the result of the new model to the rmse_results table

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Movie + User + Genre + Year
Effects Model",
                                     RMSE = model_5_rmse ))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320
Regularized Movie Effect Model	0.9436762
Regularized Movie + User Effect Model	0.8652421

Movie + User + Genre Effects Model	0.8655941
Regularized Movie + User + Genre Effect Model	0.8649406
Movie + User + Genre + Year Effects Model	0.8655806

Now, in order to improve this model, I will regularize it:

```
#This code apply regularization to the fourth model (movie + user + genre
s + year effect), using cross-validation to choose the best value for lam
bda

lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
    mu <- mean(train_set$rating)
    b_i <- train_set %>%
        group_by(movieId) %>%
        summarize(b_i = sum(rating - mu)/(n()+l))
    b_u <- train_set %>%
        left_join(b_i, by="movieId") %>%
        group_by(userId) %>%
        summarize(b_u = sum(rating - b_i - mu)/(n()+l))
    b_g <- train_set %>%
        left_join(b_i, by="movieId") %>%
        left_join(b_u, by="userId") %>%
        group_by(genres) %>%
        summarize(b_g = sum(rating - b_i - b_u- mu)/(n()+l))
    b_y <- train_set %>%
        left_join(b_i, by="movieId") %>%
        left_join(b_u, by="userId") %>%
        left_join(b_g, by="genres") %>%
        group_by(year) %>%
        summarize(b_y = sum(rating - mu - b_i - b_u- b_g)/(n()+l))


    predicted_ratings <-
        test_set %>%
        left_join(b_i, by = "movieId") %>%
        left_join(b_u, by = "userId") %>%
        left_join(b_g, by = "genres") %>%
        left_join(b_y, by = "year") %>%
        mutate(pred = mu + b_i + b_u + b_g + b_y) %>%
        .$pred
    return(RMSE(predicted_ratings, test_set$rating))
})

#Plot lambda and rmses value to demonstrate the best value for lambda

qplot(lambdas, rmses)
```




```
#Shows the lambda which generates minimum RMSE

lambda <- lambdas[which.min(rmses)]
lambda

## [1] 5
```

As demonstrated in the plot, the better value for lambda is the one which generates the smallest value for RMSE. In this case, 5. Now we add the best RMSE result to the table:

```
#Add the result of the new model to the rmse_results table

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Regularized Movie + User + G
enre + Year Effect Model",
                                     RMSE = min(rmses)))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0599043
Movie Effect Model	0.9437429
Movie + User Effects Model	0.8659320
Regularized Movie Effect Model	0.9436762
Regularized Movie + User Effect Model	0.8652421

Movie + User + Genre Effects Model	0.8655941
Regularized Movie + User + Genre Effect Model	0.8649406
Movie + User + Genre + Year Effects Model	0.8655806
Regularized Movie + User + Genre + Year Effect Model	0.8648982

And finally, with regularization applied to my final model, which includes movie, user, genres and year effects, I reached a **Residual Mean Squared Error (RMSE) of 0.8648982**, below the **0.8649**, which is the one required to get the max score of 25 points.

---

3 RESULTS

Now there is only one step left: apply the final model to validation set and check the final RMSE. First, I must add date and year columns to edx and validation sets:

```
#Add date and year columns to edx and validation set using lubridate pack
age command as_datetime

edx<-mutate(edx, date = as_datetime(timestamp),year=year(date))

validation<-mutate(validation, date = as_datetime(timestamp),year=year(da
te))
```

And now I apply my final model, considering lambda (l=5):

```
#And now I apply my final model, considering lambda (l=5):
#This code applies the final model (Regularized movie + user + genres + y
ear effect) to edx and validation set

l<-5

mu <- mean(edx$rating)

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

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

b_g <- edx %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
```

```
  group_by(genres) %>%
  summarize(b_g = sum(rating - b_i - b_u- mu)/(n()+l))

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


predicted_ratings <-
  test_set %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  left_join(b_g, by = "genres") %>%
  left_join(b_y, by = "year") %>%
  mutate(pred = mu + b_i + b_u + b_g + b_y) %>%
  .$pred

# Calculate the predicted values for the validation data set

predicted_ratings <- validation %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  left_join(b_g, by = "genres") %>%
  left_join(b_y, by = "year") %>%
  mutate(pred = mu + b_i + b_u + b_g + b_y) %>%
  pull(pred)


# Final RMSE
Final_RMSE <- RMSE(predicted_ratings, validation$rating)
Final_RMSE

## [1] 0.8644101
```

As we can see, applying the final model to validation set resulted in an even smaller value for RMSE: **0.8644101**.

---

4 CONCLUSIONS

Using the example provided in the Data Science: Machine Learning Course, I was able to get RMSE result of 0.8652. To reach my goal (the max score), I had to test other predictors. Both predictors tested improved my model and I got a RMSE result of 0.8644101, which is within the interval for the best score. It is important to note, however, that I could use other methods to get better results, like Linear Models, k-

nearest neighbors (kNN) or Random Forests. These methods, though, couldn't be used in this case for two correlated reasons: dataset size and my computer specifications. During the Capstone Project course, I had to buy an SSD disk and 8GB RAM memory for my laptop, since it was taking too long to process some training tests. I then formatted my laptop and installed OS on SSD disk, what improved significantly my work. It wasn't enough, though, to train using knn or rf methods. If dataset was smaller, I could have tested more advanced machine learning algorithms. The basic approach learned in the Machine Learning course and throughout the other courses of the program was enough for reaching the goal. The recommendation system model has an acceptable accuracy and can be improved as I advance in my studies about machine learning.