Movie Recommendation System

Samuel Velez Arango

07/08/2019

Introduction

The Netflix challenge offered $1 million dollars to the team that could provide a movie reccomendation system that improved theirs by 10%. While the Netflix data set isn't public, MovieLens provides a similar one. We are going to attempt to provide a solution to the challenge (through the MovieLens data set) using linear regression, trying out three different approaches.

Each row in the MovieLens database corresponds to the rating one user assigned to a movie, and there's and ID that identifies each individual user.

The following snippets download the data set en prepare the data

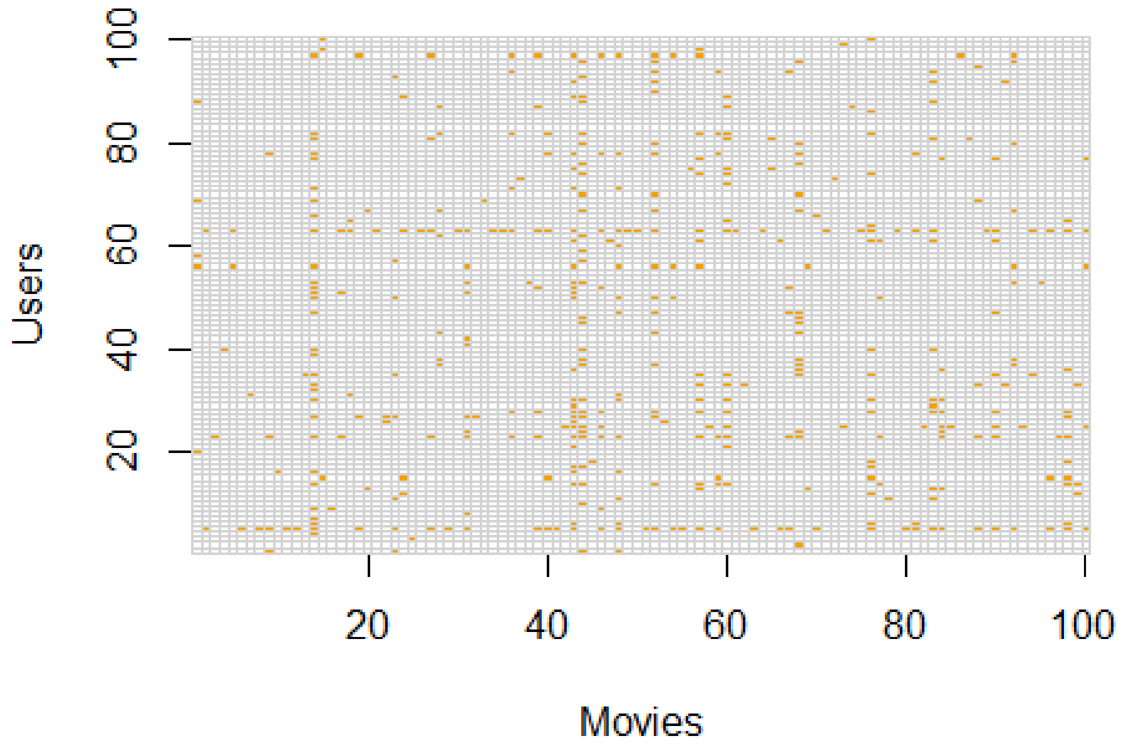
```r
library(dslabs)
library(tidyverse)
library(data.table)
library(caret)
data("movielens")

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

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

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

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

What follows is the total number of users who rated and the number of movies rated:

```r
library(dplyr)
```

```
count_users <-  length(unique(movielens$userId))
count_movies <- length(unique(movielens$movieId))
data.frame(count_users,count_movies)

##   count_users count_movies
## 1       69878        10677
```

However not every user rated every movie, and therefore many fields in the matrix will be empty, as we can see in the following sample plot highlighting the ratings we do have:



Pre-processing

We are going to partition the data into a training and a test data subset, the first containing 90% of the data and the latter the remaining.

```
library(caret)

set.seed(1)
data_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.
1, list = FALSE)
train_set <- movielens[-data_index,]
temp <- movielens[data_index,]
```

To make sure we don't have users or movies in one set that don't appear in the other we use the semi_join function

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

The rows that were removed from test_set are put back into train_set

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

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

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

The winners of the challenge were defined based on the Residual Sum of Squared Errors, RMSE. To win, the resulting RMSE had to be lower than 0.85. The RMSE was stated like this:

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

Recommendation system

To create the recommendation system, we will use an iterative process, increasing on the complexity of the previous attempt until the data set is fully exploited.

Simplest solution: disregarding movie ratings and user preferences.

The simplest possible solution is to suggest the same movie for all users, regardless of their preferences. This would be represented by a model that chalks up all variations in preferences to the standard error:

```
average_rating <- mean(train_set$rating)
average_rating

## [1] 3.512464
```

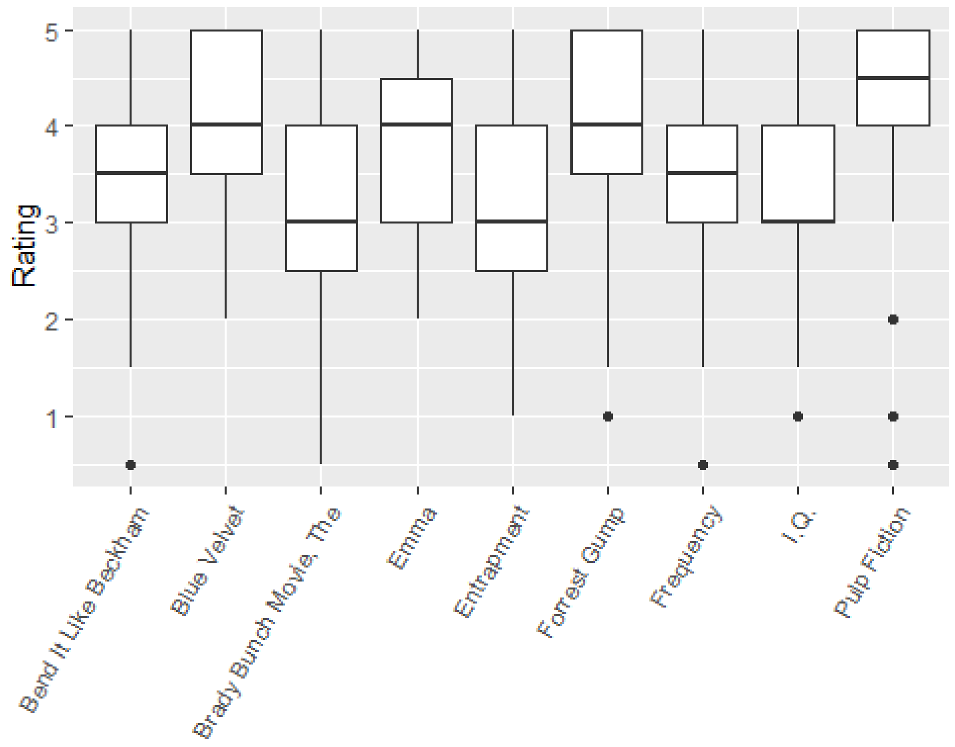
Through the previous code, our model will only use the average rating and an error term. The root mean squared-error can be calculated as follows:

```
simple_rmse <- RMSE(test_set$rating,average_rating)
simple_rmse

## [1] 1.060651
```

And so, the simplest possible solution provides an RMSE of a little over 1.

Recommendation based on movie ratings

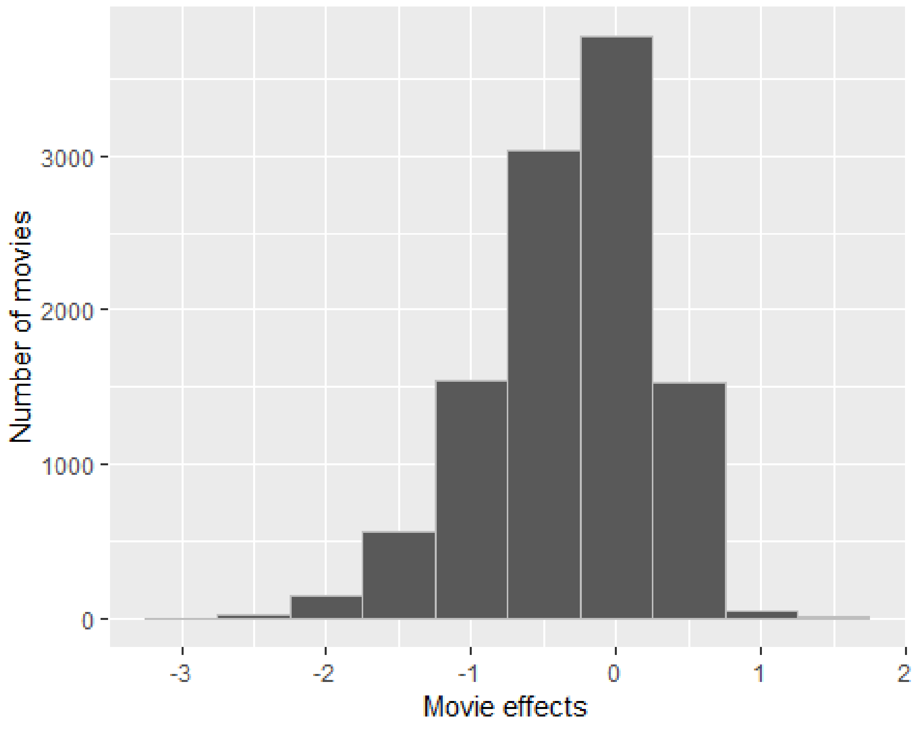
Another approach would imply taking into account movie ratings. Some movies have a better average rating than others:



And we can use this for the recommendation system by accounting for individual movie ratings. In the Netflix challenge, the difference between the overall average rating and the average rating of each movie was called the bias, and we will implement it with the following code, that groups by movie and takes the average of its ratings, to later substract the total average:

```
movie_average <- train_set %>%
  group_by(movieId) %>%
  summarize(movie_effects = mean(rating - average_rating), times_rated =
n())
```

And we can see the distribution of this bias:



The expressed model would be of the form Prediction = Overall Movie Average + Movie Bias + Random Error, and its RMSE is the following:

```
predicted_ratings <- average_rating + test_set %>%
  left_join(movie_average, by='movieId') %>%
  pull(movie_effects)

movie_effects_rmse <- RMSE(predicted_ratings, test_set$rating)
movie_effects_rmse

## [1] 0.9437046
```

Taking each movie's average rating into account improves our RMSE slightly.

Recommendation based on individual preference and movie rating

Introducing the individual preference of each user will add the last layer of information provided by the MovieLens data set:

The following code is similar to the one used to find the movie bias; we are now finding each user's preference bias by removing both the movie effect and the mean overall rating effect.

```
user_average <- train_set %>%
  left_join(movie_average, by='movieId') %>%
  group_by(userId) %>%
  summarize(user_effects = mean(rating - average_rating - movie_effects))
```

Afterwards, with the user bias, we create a formula of the form: Prediction = Overall Movie Average + Movie Bias + User Bias + Error, so when a hard to please user who rates all movies below their average rating needs a suggestion, the effect of his manifested preferences will be negated against the movie's rating, and we can also identify correctly when he isn't bashing a movie but giving a personally above average review:

```
predicted_ratings <- test_set %>%
  left_join(movie_average, by='movieId') %>%
  left_join(user_average, by='userId') %>%
  mutate(prediction = average_rating + movie_effects + user_effects) %>%
  pull(prediction)
```

Our results so far:

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

rmse_results <- data_frame(method = "Average of all ratings", RMSE = simp
le_rmse)

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

rmse_results <- bind_rows(rmse_results, data_frame(method="Movie Effects
Model", RMSE = movie_effects_rmse))
rmse_results <- bind_rows(rmse_results, data_frame(method="Movie and User
Effects Model", RMSE = user_effects_rmse))
rmse_results

## # A tibble: 3 x 2
##   method                         RMSE
##   <chr>                         <dbl>
## 1 Average of all ratings        1.06
## 2 Movie Effects Model           0.944
## 3 Movie and User Effects Model  0.866
```

Each time we have incorporated more information into the model our results have improved, but so far won't win the Netflix prize.

Introducing regularization

The biggest problem with our previous result is how skewed our movie effect prediction is with movies that receive very few ratings. As we can see in the following tables, most of our best and worst rated movies were rated less than 3 times, and most frequently just once. (Remember our movie bias is the difference between the movie's average rating and the average rating for all movies, and so it ranges from -3.04 to 1.45).

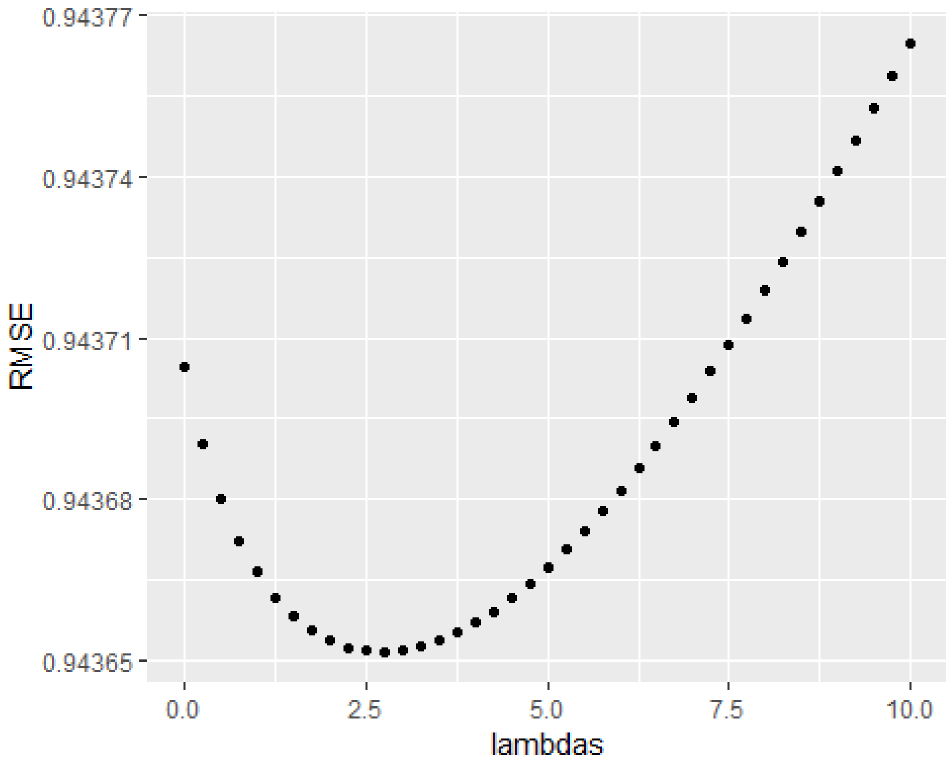
```
## # A tibble: 5 x 4
##   movieId title                                      movie_effects times_
rated
##     <dbl> <chr>                                              <dbl>
<int>
## 1    3226 <NA>                                                1.49
1
## 2   33264 <NA>                                                1.49
2
## 3   42783 Shadows of Our Forgotten Ancestors (Ti~             1.49
1
## 4   51209 <NA>                                                1.49
1
## 5   53355 <NA>                                                1.49
1

## # A tibble: 5 x 4
##   movieId title movie_effects times_rated
##     <dbl> <chr>         <dbl>       <int>
## 1    5805 <NA>          -3.01           2
## 2    8394 <NA>          -3.01           1
## 3    8707 <NA>          -3.01           1
## 4   61768 <NA>          -3.01           1
## 5   64999 <NA>          -2.76           2
```

while the previous attempts were based on minimizing the residual sum of squares, using Regularization we will penalize these large estimates born of small sample sizes.

To do this, we create a term that penalizes an estimate depending on its sample size, but will be negated with larger ones. How much we punish depends on the parameter lambda, the larger it is, the more we push estimates with small samples towards zero, but, the larger the sample is, the more lambda will be ignored.

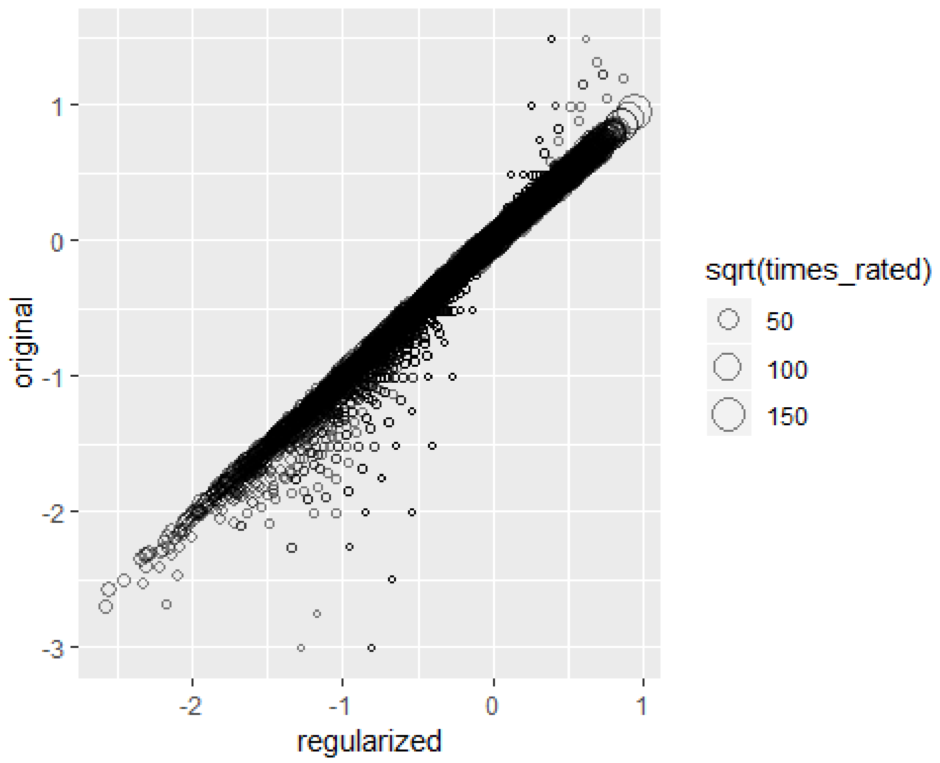
To choose the best lambda to calculate the the movie bias we will use cross validation, testing a series of lambda values and extracting the one that minimizes the RMSE:

```r
lambdas <- seq(0, 10, 0.25)

rating_minus_average_rating <- train_set %>%
  group_by(movieId) %>%
  summarize(s = sum(rating - average_rating), n_i = n())

RMSE <- sapply(lambdas, function(l){
  predicted_ratings <- test_set %>%
    left_join(rating_minus_average_rating, by='movieId') %>%
    mutate(regularized_movie_effects = s/(n_i+l)) %>%
    mutate(pred = average_rating + regularized_movie_effects) %>%
    pull(pred)
  return(RMSE(predicted_ratings, test_set$rating))
})
qplot(lambdas, RMSE)
```



```
lambdas[which.min(RMSE)]

## [1] 2.75

optimal_lambda <- lambdas[which.min(RMSE)]

regularized_movie_effects <- train_set %>%
     group_by(movieId) %>%
     summarize(movie_effects = sum(rating - average_rating)/(n()+optimal_
lambda), times_rated = n())
```

And here is a comparison between the regularized estimates and the least square estimates, with dot size representing sample size (in square root to facilitate visualization):



As we can see, the previous extremes with small samples now gravitate around the center.

These are our new best and worst movies, which make more sense:

```
movie_titles_best <- movielens %>%
  select(movieId, title) %>%
  distinct()

regularized_movie_effects %>% left_join(movie_titles_best, by="movieId")
%>%
  arrange(desc(movie_effects)) %>%
  select(movieId,title,movie_effects,times_rated) %>%
  slice(1:5)

## # A tibble: 5 x 4
##   movieId title                          movie_effects times_rated
##     <dbl> <chr>                                  <dbl>       <int>
## 1     318 Shawshank Redemption, The              0.944       27988
## 2     858 Godfather, The                         0.906       17776
## 3    4454 More                                   0.863           7
## 4      50 Usual Suspects, The                    0.857       21533
## 5     527 Schindler's List                       0.853       23234

movie_titles_worst <- movielens %>%
  select(movieId, title) %>%
  distinct()
```

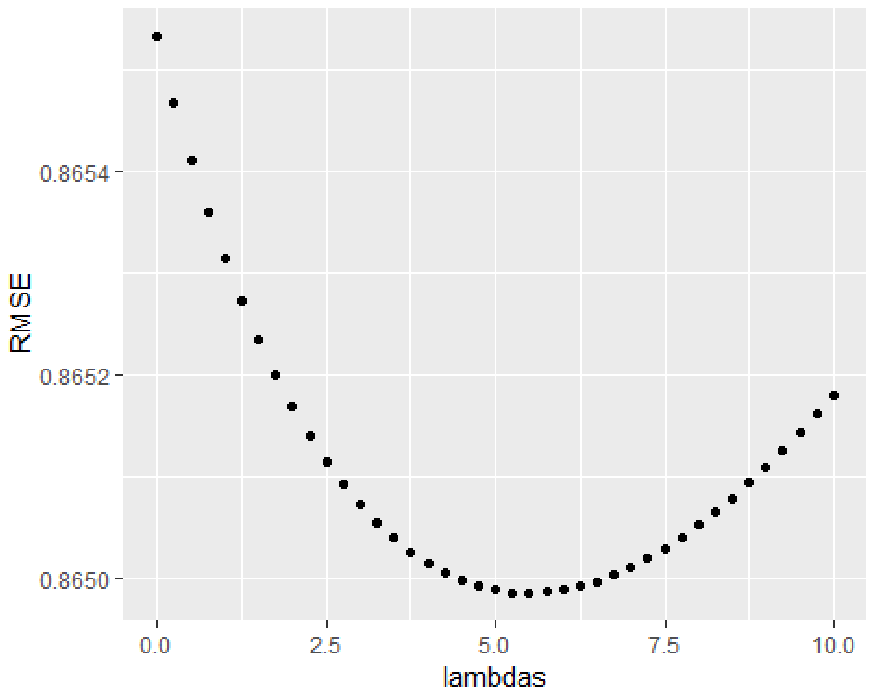
```
regularized_movie_effects %>% left_join(movie_titles_worst, by="movieId")
%>%
  arrange((movie_effects)) %>%
  select(movieId,title,movie_effects,times_rated) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 4
##   movieId title                        movie_effects times_rated
##     <dbl> <chr>                                <dbl>       <int>
## 1    8859 SuperBabies: Baby Geniuses 2         -2.58          59
## 2    6483 <NA>                                 -2.55         198
## 3    6371 <NA>                                 -2.46         145
## 4    4775 Glitter                              -2.34         340
## 5   61348 Disaster Movie                       -2.32          31
```

While the aforementioned examples are of the movie effects part of our model, the user effects also suffer the same effect, and so this regularization will be applied to it as well.

To choose lambda for both the movie and user biases we will use cross validation to minimize the RMSE again:

```
lambdas <- seq(0, 10, 0.25)

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

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

  user_effect_u <- train_set %>%
    left_join(movie_effect_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(user_effect_u = sum(rating - movie_effect_i - average_ratin
g)/(n()+l))

  predicted_ratings <-
    test_set %>%
    left_join(movie_effect_i, by = "movieId") %>%
    left_join(user_effect_u, by = "userId") %>%
    mutate(pred = average_rating + movie_effect_i + user_effect_u) %>%
    pull(pred)

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

qplot(lambdas, RMSE)
```



```
lambdas[which.min(RMSE)]

## [1] 5.5

regularized_rmse <- min(RMSE)
```

And so, we use this optimal lambda for our regularized estimation, and obtain the following RMSE:

```
regularized_rmse

## [1] 0.8649857
```

Result and conclusions

With each iteration our results got progressively better, and while this result inched us closer to the $1 million dollars prize, it fell short of it.

This is a fun exercise that only needs linear models and regularization, that proved to be easy to understand and implement. The limitations of this approach come from linear regressions, and the fact that they aren't learning models, and are therefore very strict on their use, as any input outside of the initial "training" will provide a theoretically erroneous result.

```
      Method                                                RMSE
Average of all ratings                                    1.060650
Movie Effects Model                                       0.943704
Movie and User Effects Model                              0.865532
Regularized Movie Effects and Regularized User Effects    0.864985
```