MovieLens Project

dsjames

1/2/2021

INTRODUCTION

I'll be creating a recommendation system based off of the MovieLens dataset. It contains millions of movie ratings (from 1 to 5) from thousands of users.

Ideally, this will allow for the creation of an algorithm that can predict the ratings one user might give to a movie they have yet to watch.

METHODS

For this project, I'll be using the following packages:

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.0.5
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1
```

```
## Warning: package 'ggplot2' was built under R version 4.0.5
```

```
## Warning: package 'tibble' was built under R version 4.0.5
```

```
## Warning: package 'tidyr' was built under R version 4.0.5
```

```
## Warning: package 'readr' was built under R version 4.0.5
```

```
## Warning: package 'dplyr' was built under R version 4.0.5
```

```
## Warning: package 'forcats' was built under R version 4.0.5
```

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

```
library(dplyr)
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.0.5
```

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

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

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

```
library(data.table)
```

```
## Warning: package 'data.table' was built under R version 4.0.5
```

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

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

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

```
library(tidyr)
```

The `tidyverse` package will be used for, well, tidying the data.

The same is true for `dplyr`, and you'll see me use it quite often when filtering my predictions.

`caret` is essential for creating the training and test data sets, but I won't be using any built in models.

`data.table` and `tidyr` will be used primarily to organize the data as it goes through the algorithm.

CREATE TRAIN AND VALIDATION SETS

To start with, I'll include the code that was used to generate the `edx` dataset

```
library(tidyverse)
library(caret)
library(data.table)

# keep track of the directory where the file is being downloaded as `dl`
dl <- tempfile()
download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)
```

```
# read and tidy the ratings portion of the file
ratings <- fread(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

# read and tidy the movie IDs, titles and genres
movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres") # readable column names

# Coerce data into classes that are easier to work with
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres))

# merge ratings and movies into one dataset by movie ID
movielens <- left_join(ratings, movies, by = "movieId")

# Create the test set
set.seed(1, sample.kind = "Rounding") # if using R 3.6 or later
```

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

```
# set.seed(1) # if using R 3.5 or earlier, use this one instead
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index]
temp <- movielens[test_index]

# create a validation copy of `temp` containing entries that appear in both `temp` and `edx`
validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

# create an copy of `temp` that does not contain entries that appear in `validation`
removed <- anti_join(temp, validation)
```

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

```
edx <- rbind(edx, removed) # add rows from `removed` to `edx`

# remove unnecessary object/data, leaving only `edx` and `validation`
rm(dl, ratings, movies, test_index, temp, movielens, removed)
```

Create training and test sets from within `edx`

Given that the sample size for `edx` is rather large, I can create multiple training and test sets from that, and will not have to rely on bootstrapping to extend them. I may do so anyway, however, in order to simulate multiple datasets of the same size as edx.

Additionally, by subdividing it into a similarly distributed dataset (90% training, 10% testing), I can emulate the distribution within the `edx` training and test sets.

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

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

```r
# I had to add this one in because when I ran it again using a clear environment,
# it gave inconsistent RMSEs

test_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1, list = FALSE)

train_edx <- edx[-test_index]
temp <- edx[test_index]

test_edx <- temp %>%
  semi_join(train_edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

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

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

```r
train_edx <- rbind(train_edx, removed)

rm(test_index, temp, removed)
```

Lastly, I'll create the RMSE function that I'll be using to test my algorithm's accuracy.

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

ANALYSIS

I'll start off with some data exploration. Firstly, I'll establish a baseline average, `mu_hat`, which is the average of all ratings in my `train_edx` test set.

```r
mu_hat <- mean(train_edx$rating) # start with the mean of all ratings
mu_hat # 3.512456
```

```
## [1] 3.512456
```

I'll also test the predictive power of this using my RMSE function from earlier.

```r
test_edx_ratings <- test_edx$rating

naive_RMSE <- RMSE(test_edx_ratings, mu_hat)
# Call the RMSE function to test `mu_hat` against my test set.
naive_RMSE # 1.060054
```

```
## [1] 1.060054
```

Could be worse, but not passing.

Now I'll test if there's an alternative, with the Median of the possible ratings.

```
preds <- rep(3, nrow(test_edx))
RMSE(preds, test_edx_ratings) #1.17741
```

```
## [1] 1.177464
```

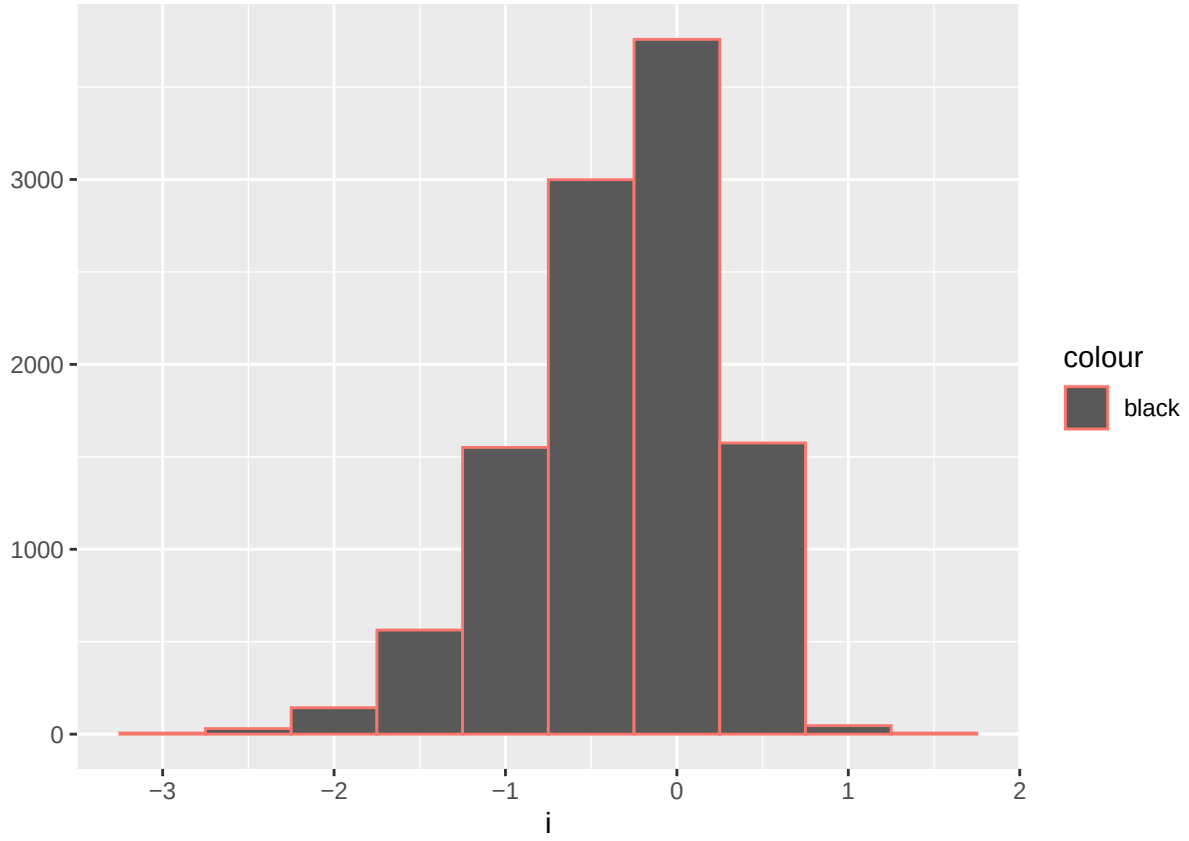
I'll just check the variation of ratings by subtracting `mu_hat`.

```
movies <- train_edx %>%
  group_by(movieId) %>%
  summarize(i = mean(rating - mu_hat))
head(movies)
```

```
## # A tibble: 6 x 2
##   movieId      i
##     <dbl>  <dbl>
## 1       1  0.415
## 2       2 -0.306
## 3       3 -0.361
## 4       4 -0.637
## 5       5 -0.442
## 6       6  0.302
```

Interesting. I'll visualize the data to see if there are any insights that might explain this.

```
qplot(i, data = movies, bins = 10, color = "black")
```



So it would seem that most ratings actually range from 3 to 4, with the most coming up in the 3.5 bin.

Of course there are some outliers, but they seem so few in number. I wonder what kind of movie warrants an average rating of 1 or 5.

```
worstMovies <- train_edx %>% group_by(movieId) %>%
  summarize(rating = mean(rating)) %>%
  filter(rating <= 1)
worstMovies
```

```
## # A tibble: 18 x 2
##    movieId rating
##      <dbl>  <dbl>
##  1     604  1
##  2    2228  1
##  3    3561  1
##  4    4071  1
##  5    4075  1
##  6    5702  1
##  7    5805  0.5
##  8    6189  1
##  9    6483  0.874
## 10    7282  0.909
## 11    8394  0.5
## 12    8856  1
## 13    8859  0.745
## 14   55324  1
```

```
## 15   61348  0.767
## 16   61768  0.5
## 17   63828  0.5
## 18   64999  0.5
```

```
bestMovies <- train_edx %>% group_by(movieId) %>%
  summarize(rating = mean(rating)) %>%
  filter(rating >= 4.5)
bestMovies
```

```
## # A tibble: 26 x 2
##    movieId rating
##      <dbl>  <dbl>
##  1    3226   5
##  2    4454   4.67
##  3    5194   4.75
##  4    5849   4.67
##  5    7452   4.5
##  6    7823   4.5
##  7   25975   4.75
##  8   26048   4.75
##  9   26073   4.75
## 10   33264   5
## # ... with 16 more rows
```

There's only 19 movies with an average rating of 1 or lower, and only 30 that are equal to or above 4.5. That's not entirely impossible, but it's strange that there are so many more that are above 30. Additionally, quite a few of those in `worstMovies` have a rating of exactly 1. That must be truly awful. Though even bad movies might warrant the odd 2-star rating. Perhaps I should take a look at moves whose average rating is exactly 5.

```
bestMovies <- edx %>% group_by(movieId) %>%
  summarize(rating = mean(rating), n = n()) %>%
  filter(rating == 5)
bestMovies
```

```
## # A tibble: 6 x 3
##   movieId rating     n
##     <dbl>  <dbl> <int>
## 1    3226      5     1
## 2   33264      5     2
## 3   42783      5     1
## 4   51209      5     1
## 5   53355      5     1
## 6   64275      5     1
```

```
worstMovies <- edx %>% group_by(movieId) %>%
  summarize(rating = mean(rating), n = n()) %>%
  filter(rating <= 1)
worstMovies
```

```
## # A tibble: 17 x 3
```

```
##    movieId rating     n
##      <dbl>  <dbl> <int>
##  1     604  1         2
##  2    2228  1         2
##  3    3561  1         1
##  4    4071  1         1
##  5    4075  1         1
##  6    5702  1         1
##  7    5805  0.5       2
##  8    6189  1         1
##  9    6483  0.902   199
## 10    7282  0.821    14
## 11    8394  0.5       1
## 12    8859  0.795    56
## 13   55324  1         1
## 14   61348  0.859    32
## 15   61768  0.5       1
## 16   63828  0.5       1
## 17   64999  0.5       2
```

That was informative. Movies with a perfect rating of 5 only have 1 rating, and most of the movies with a rating of 1 or lower only have a few ratings. What's this? An average rating of 0.982 with *199* ratings? It must be truly awful.

```
theWorstMovie <- edx[movieId == 6483]
theWorstMovie[1]
```

```
##    userId movieId rating  timestamp                       title          genres
## 1:   1186    6483    0.5 1134150672 From Justin to Kelly (2003) Musical|Romance
```

I wonder if it's still on Netflix...

It is not.

Oh well. Later on while creating the algorithm, I'll have to come up with a way to compensate for these movies with very few ratings, as they will skew the averages. It's entirely likely that a child might watch a terrible movie but enjoy it and give it a 5, while an absolute grump might see a genuinely good movie and rate it a 1 out of spite for some unknowable reason.

For now, I'll regularize all of the ratings based on their deviation from the mean `mu_hat`.

```
avg_movie <- train_edx %>%
  group_by(movieId) %>%
  summarize(mov_avg = mean(rating - mu_hat))
head(avg_movie)
```

```
## # A tibble: 6 x 2
##   movieId mov_avg
##     <dbl>   <dbl>
## 1       1   0.415
## 2       2  -0.306
## 3       3  -0.361
## 4       4  -0.637
## 5       5  -0.442
## 6       6   0.302
```

Now I can add that on to `mu_hat` as a modifier and create some starting predictions.

```
movie_preds <- mu_hat + test_edx %>%
  left_join(avg_movie, by = "movieId") %>%
  pull(mov_avg)

RMSE(movie_preds, test_edx_ratings) # 0.9429615
```

```
## [1] 0.9429615
```

Clear improvement over the original 1.06. It's a start.

Given how inconsistent ratings could be when only a few people rated something, I should regularize by user as well.

```
avg_user <- train_edx %>%
  group_by(userId) %>%
  summarize(user_avg = mean(rating - mu_hat))
avg_user
```

```
## # A tibble: 69,878 x 2
##    userId user_avg
##     <int>    <dbl>
##  1      1    1.49
##  2      2   -0.512
##  3      3    0.432
##  4      4    0.609
##  5      5    0.455
##  6      6    0.435
##  7      7    0.340
##  8      8   -0.133
##  9      9    0.488
## 10     10    0.334
## # ... with 69,868 more rows
```

```
user_preds <- mu_hat + test_edx %>%
  left_join(avg_user, by = "userId") %>%
  pull(user_avg)

RMSE(user_preds, test_edx_ratings) # 0.977709
```

```
## [1] 0.977709
```

Well, by itself, not great, but better than `mu_hat` by itself.

Now I'll add the regularizations for both `userId` and `movieId`.

```
reg_preds <- test_edx %>%
  left_join(avg_user, by = "userId") %>%
  left_join(avg_movie, by = "movieId") %>%
  mutate(preds = mu_hat + mov_avg + user_avg) %>%
  pull(preds)

RMSE(reg_preds, test_edx_ratings) # 0.8843987
```

```
## [1] 0.8843987
```

considerably better. Within the passing range for this project.

Though I know there's room for improvement. Just because one person gave a movie a 5-star rating doesn't mean it's the perfect movie, it only means that the one person was willing to go out of their way to rate it. That one person may be overly generous with 5-star ratings, or maybe they have very niche tastes, or maybe they were in a great mood at the time.

Whatever the case, I can't account for those kinds of factors, but I can weed them out by limited the impact of movies with so few ratings.

```
train_edx %>% group_by(movieId) %>%
  summarize(rating = mean(rating), n = n()) %>%
  arrange(desc(rating)) %>%
  filter(rating >= 4.5) %>%
  head()
```

```
## # A tibble: 6 x 3
##   movieId rating     n
##     <dbl>  <dbl> <int>
## 1    3226      5     1
## 2   33264      5     1
## 3   42783      5     1
## 4   51209      5     1
## 5   53355      5     1
## 6   64275      5     1
```

Yeah, look at that. None of the movies that received an average rating of 4.5 or higher had more than a few reviews.

I can modify `train_edx` to eliminate these outliers in order to improve accuracy. First, I'll make sure I'm not removing too many ratings.

```
reg_edx <- train_edx %>% group_by(movieId) %>%
  filter(n() > 10)

reg_edx
```

```
## # A tibble: 8,093,844 x 6
## # Groups:   movieId [9,426]
##    userId movieId rating timestamp title                 genres
##     <int>   <dbl>  <dbl>     <int> <chr>                 <chr>
##  1      1     122      5 838985046 Boomerang (1992)      Comedy|Romance
##  2      1     292      5 838983421 Outbreak (1995)       Action|Drama|Sci-Fi|T~
##  3      1     316      5 838983392 Stargate (1994)       Action|Adventure|Sci-~
##  4      1     329      5 838983392 Star Trek: Generation~ Action|Adventure|Dram~
##  5      1     355      5 838984474 Flintstones, The (199~ Children|Comedy|Fanta~
##  6      1     356      5 838983653 Forrest Gump (1994)   Comedy|Drama|Romance|~
##  7      1     362      5 838984885 Jungle Book, The (199~ Adventure|Children|Ro~
##  8      1     364      5 838983707 Lion King, The (1994) Adventure|Animation|C~
##  9      1     370      5 838984596 Naked Gun 33 1/3: The~ Action|Comedy
## 10      1     377      5 838983834 Speed (1994)          Action|Romance|Thrill~
## # ... with 8,093,834 more rows
```

```
nrow(train_edx)
```

```
## [1] 8100065
```

```
nrow(reg_edx)
```

```
## [1] 8093844
```

```
nrow(train_edx) - nrow(reg_edx)
```

```
## [1] 6221
```

So I'm only eliminating 6,221 ratings. I'll still have over 8 million, left over in reg_edx, so no great loss.

Now I just need to come up with a way that gives relative weights to the values based on the number of ratings.

```
avg_movie_n <- train_edx %>%
  group_by(movieId) %>%
  summarize(avg_mov = sum(rating - mu_hat) / n())
head(avg_movie_n)
```

```
## # A tibble: 6 x 2
##   movieId avg_mov
##     <dbl>   <dbl>
## 1       1   0.415
## 2       2  -0.306
## 3       3  -0.361
## 4       4  -0.637
## 5       5  -0.442
## 6       6   0.302
```

```
avg_user_n <- train_edx %>%
  group_by(userId) %>%
  summarize(avg_user = sum(rating - mu_hat) / n())
head(avg_user_n)
```

```
## # A tibble: 6 x 2
##   userId avg_user
##    <int>    <dbl>
## 1      1    1.49
## 2      2   -0.512
## 3      3    0.432
## 4      4    0.609
## 5      5    0.455
## 6      6    0.435
```

```
reg_preds_n <- test_edx %>%
  left_join(avg_user_n, by = "userId") %>%
  left_join(avg_movie_n, by = "movieId") %>%
  mutate(preds = mu_hat + avg_mov + avg_user) %>%
  pull(preds)

RMSE(reg_preds_n, test_edx_ratings) # 0.8843987
```

```
## [1] 0.8843987
```

Hmmm. No change. Perhaps I should introduce a lambda value to offset the divisor on each `avg_` in order to adjust for movies with fewer (but not an inconsequential number of) ratings.

```
avg_movie_10 <- train_edx %>%
  group_by(movieId) %>%
  summarize(avg_mov = sum(rating - mu_hat) / (n() + 10))
head(avg_movie_10)
```

```
## # A tibble: 6 x 2
##   movieId avg_mov
##     <dbl>   <dbl>
## 1       1   0.415
## 2       2  -0.306
## 3       3  -0.361
## 4       4  -0.633
## 5       5  -0.441
## 6       6   0.302
```

```
avg_user_10 <- train_edx %>%
  group_by(userId) %>%
  summarize(avg_user = sum(rating - mu_hat) / (n() + 10))
head(avg_user_10)
```

```
## # A tibble: 6 x 2
##   userId avg_user
##    <int>    <dbl>
## 1      1    0.956
## 2      2   -0.290
## 3      3    0.315
## 4      4    0.467
## 5      5    0.392
## 6      6    0.344
```

```
reg_preds_10 <- test_edx %>%
  left_join(avg_user_10, by = "userId") %>%
  left_join(avg_movie_10, by = "movieId") %>%
  mutate(preds = mu_hat + avg_mov + avg_user) %>%
  pull(preds)

RMSE(reg_preds_10, test_edx_ratings) # 0.8810978
```

```
## [1] 0.8810978
```

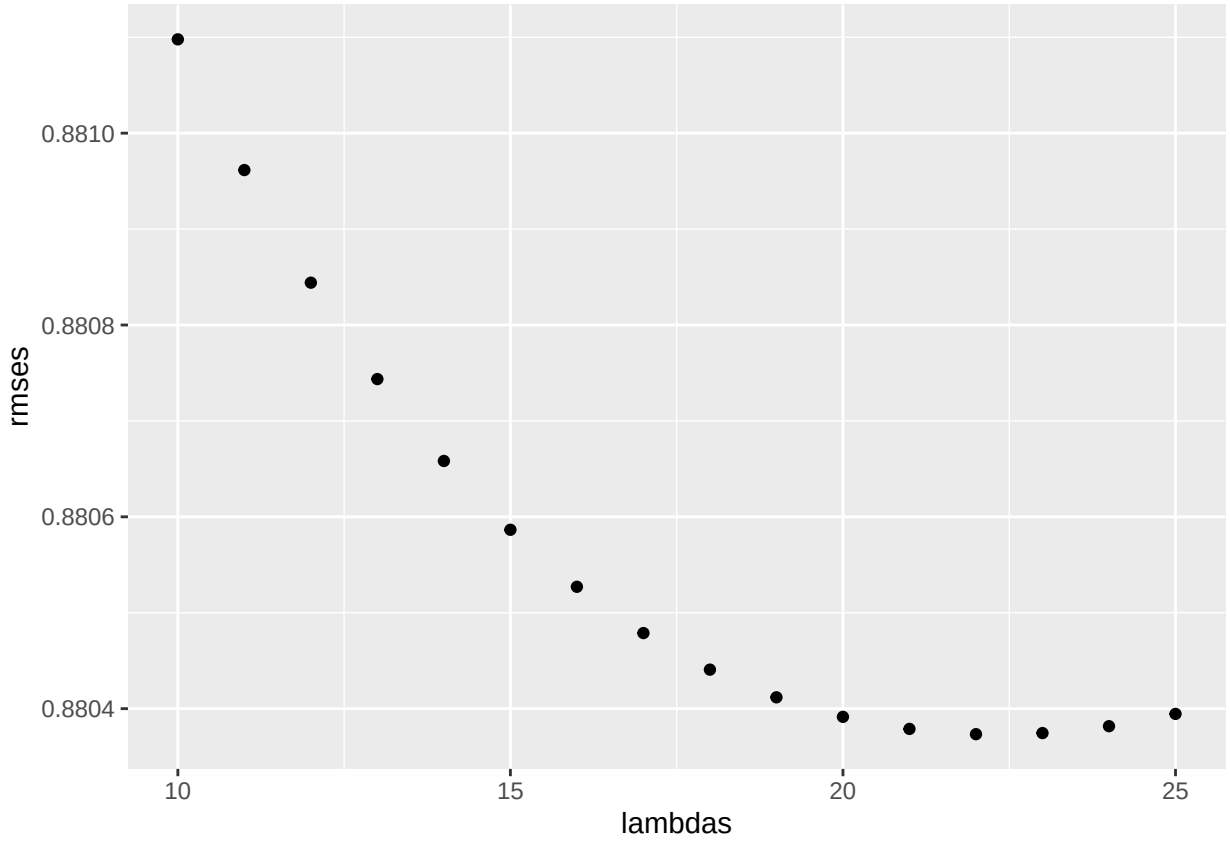
Fantastic. Not a huge difference, but I'll take it.

Maybe 10 isn't the best possible lambda for this purpose so I'll try out a series of lambda using `sapply()` to see if maybe there's a better one.

```
lambdas <- seq(10, 25, 1) # note: I originally ran this with `seq(0, 100, 1)`,
                          # but for now I'm going to reduce it to the range to
                          # save time for anyone trying to run this.
rmses <- sapply(lambdas, function(l) {
  mu_hat <- mean(train_edx$rating)

  avg_movie <- train_edx %>%
    group_by(movieId) %>%
    summarize(avg_mov = sum(rating - mu_hat) / (n() + l))

  avg_user <- train_edx %>%
    group_by(userId) %>%
    summarize(avg_user = sum(rating - mu_hat) / (n() + l))

  preds <- test_edx %>%
    left_join(avg_movie, by = "movieId") %>%
    left_join(avg_user, by = "userId") %>%
    mutate(preds = mu_hat + avg_mov + avg_user) %>%
    pull(preds)

  return(RMSE(preds, test_edx_ratings))
})
head(rmses)
```

```
## [1] 0.8810978 0.8809615 0.8808441 0.8807436 0.8806582 0.8805865
```

```
qplot(lambdas, rmses)
```




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

```
## [1] 22
```

Now I'll take that newly discovered best lambda and rebuild my algorithm using that.

```
avg_movie_l <- train_edx %>%
  group_by(movieId) %>%
  summarize(avg_mov = sum(rating - mu_hat) / (n() + lambda))

avg_user_l <- train_edx %>%
  group_by(userId) %>%
  summarize(avg_user = sum(rating - mu_hat) / (n() + lambda))


reg_preds_l <- test_edx %>%
  left_join(avg_movie_l, by = "movieId") %>%
  left_join(avg_user_l, by = "userId") %>%
  mutate(preds = mu_hat + avg_mov + avg_user) %>%
  pull(preds)

RMSE(reg_preds_l, test_edx_ratings) # 0.8803734
```

```
## [1] 0.8803734
```

Huzzah! One additional increment of improvement.

Now I'll try to regularize by genre as well.

```
train_genres <- train_edx %>%
  group_by(genres) %>%
  summarize(n = n(), rating = mean(rating - mu_hat)) %>%
  filter(n >= 10)

train_genres
```

```
## # A tibble: 760 x 3
##    genres                                                n   rating
##    <chr>                                             <int>    <dbl>
##  1 Action                                            22096 -0.574
##  2 Action|Adventure                                  61787  0.148
##  3 Action|Adventure|Animation|Children|Comedy          6705  0.449
##  4 Action|Adventure|Animation|Children|Comedy|Fantasy   165 -0.519
##  5 Action|Adventure|Animation|Children|Comedy|IMAX       55 -0.176
##  6 Action|Adventure|Animation|Children|Comedy|Sci-Fi    534 -0.435
##  7 Action|Adventure|Animation|Children|Fantasy          655 -0.803
##  8 Action|Adventure|Animation|Children|Sci-Fi            41 -0.549
##  9 Action|Adventure|Animation|Comedy|Drama             1719  0.00645
## 10 Action|Adventure|Animation|Drama|Fantasy            3917  0.432
## # ... with 750 more rows
```

```
avg_genres_l <- train_edx %>%
  group_by(genres) %>%
  summarize(avg_genres = sum(rating - mu_hat) / (n() + lambda))

avg_genres_l %>% arrange(desc(avg_genres))
```

```
## # A tibble: 797 x 2
##    genres                                  avg_genres
##    <chr>                                        <dbl>
##  1 Drama|Film-Noir|Romance                      0.785
##  2 Action|Crime|Drama|IMAX                      0.779
##  3 Animation|Children|Comedy|Crime              0.764
##  4 Film-Noir|Mystery                            0.724
##  5 Crime|Film-Noir|Mystery                      0.707
##  6 Film-Noir|Romance|Thriller                   0.698
##  7 Crime|Film-Noir|Thriller                     0.695
##  8 Crime|Mystery|Thriller                       0.688
##  9 Action|Adventure|Comedy|Fantasy|Romance      0.683
## 10 Crime|Thriller|War                           0.652
## # ... with 787 more rows
```

Now to add that to the algorithm

```
reg_preds_lg <- test_edx %>%
  left_join(avg_genres_l, by = "genres") %>%
  left_join(avg_movie_l, by = "movieId") %>%
  left_join(avg_user_l, by = "userId") %>%
```

```
  mutate(preds = mu_hat + avg_genres + avg_mov + avg_user) %>%
  pull(preds)

head(reg_preds_lg)
```

```
## [1] 3.750509 4.024731 3.773589 3.387594 3.803433 5.278149
```

```
RMSE(reg_preds_lg, test_edx_ratings) # 0.9361367
```

```
## [1] 0.9361367
```

Hmm, that's less than ideal. Perhaps I should quit while I'm ahead.

After hitting this roadblock, I've spent several weeks trying to figure out a way to make matrix factorization fit into this model. While I'm sure there's a way, at a certain point it's better to move forward with satisfactory results than spend ages trying to work out something perfect. 0.8803734 is good enough, and sometimes "good enough" today is better than "better" some time down the road.

RESULTS

I'll go ahead and put the full cleaned-up algorithm here, minus the exploration I did in the previous section, explaining my results in the notes. I'll also discuss it a bit more afterwards.

```
RMSE <- function(true_ratings, predicted_ratings) { # RMSE function in order to
                                                    # test the results against
                                                    # the test set I set aside.
  sqrt(mean((true_ratings - predicted_ratings)^2))
}

mu_hat <- mean(train_edx$rating)  # the basic mean of all ratings to serve as a
                                  # baseline for the algorithm.

lambdas <- seq(10, 25, 1) # note: I originally ran this with `seq(0, 100, 1)`,
                          # but for I'm going to reduce it to the range to save
                          # time for anyone trying to run this.

rmses <- sapply(lambdas, function(l) {  # repeatedly evaluating the RMSE of each
                                        # individual lambda 10:25.
  mu_hat <- mean(train_edx$rating)  # including this within the function just in
                                    # case

  avg_movie <- train_edx %>%  # creates a vector of avg_movie containing the
                              # regularization values for each movie for the
                              # lambda required for the current run
    group_by(movieId) %>%
    summarize(avg_mov = sum(rating - mu_hat) / (n() + l))
  # subtracting each individual rating for a given movie from the mean of all
  # ratings centers the results around 0 which distinguishes the deviation of
  # each individual rating from mu_hat, making it easier to work into the
  # algorithm.
```

```
  avg_user <- train_edx %>%
    # creates a vector of avg_user containing the regularization values for each
    # user for the lambda required for the current run
    group_by(userId) %>%
    summarize(avg_user = sum(rating - mu_hat) / (n() + l))
  # performs the same regularization function as with avg_movie, but centered
  # around each `userId` instead.

  preds <- test_edx %>%
    # collates the results of the avg_movie and avg_user equations for the
    # current lambda and creates a table of predictions.
    left_join(avg_movie, by = "movieId") %>%
    # I use left_join to ensure that the table of predictions align with the
    # movies and `userId`'s contained within the test set
    left_join(avg_user, by = "userId") %>%
    # and also to ensure that there aren't any predictions in the `preds` set
    # that aren't in the test set.
    mutate(preds = mu_hat + avg_mov + avg_user) %>%
    # The real bread-and-butter of the algorithm. It actually regularizes the
    # naive mean using the appropriate adjustments
    pull(preds)
    # from the prediction set.

  return(RMSE(preds, test_edx_ratings))
  # returns the RMSE of the set created for the lambda of the current run as
  # objects in the table `rmses`
})

lambda <- lambdas[which.min(rmses)]
# picks out which specific `lambda` has the best RMSE

lambda # 22
```

```
## [1] 22
```

```
avg_movie_l <- train_edx %>%
  # avg_movie_l is the average rate by which a movie's rating deviates from
  # mu_hat, regularized by the lambda determined to produce the best RMSE
  group_by(movieId) %>%
  summarize(avg_mov = sum(rating - mu_hat) / (n() + lambda))
  # using the ideal lambda to adjust the divisor when determining the mean for
  # each individual value, I can adjust for movies or users that have very few
  # ratings in the dataset, as they're just going to introduce noise that will
  # inhibit the accuracy of my model.

avg_user_l <- train_edx %>%
  # avg_user_l is the same as avg_movie_l, except it's centered around `userId`
  group_by(userId) %>%
  summarize(avg_user = sum(rating - mu_hat) / (n() + lambda))


reg_preds_l <- test_edx %>%
  # puts together the final results of the lambda-adjusted regularization
```

```
  # factors.
  left_join(avg_movie_l, by = "movieId") %>%
  left_join(avg_user_l, by = "userId") %>%
  mutate(preds = mu_hat + avg_mov + avg_user) %>%
  pull(preds)

RMSE(reg_preds_l, test_edx_ratings) # 0.8803734
```

```
## [1] 0.8803734
```

Finally, I'll test `reg_preds_l` against the validation set.

```
reg_preds_final <- validation %>%
  left_join(avg_movie_l, by = "movieId") %>%
  left_join(avg_user_l, by = "userId") %>%
  mutate(preds = mu_hat + avg_mov + avg_user) %>%
  pull(preds)

RMSE(reg_preds_final, validation$rating) # 0.8814447
```

```
## [1] 0.8814447
```

Well, it's worse than with my original test set, but at least it's "good enough".

CONCLUSION

The short version: This is harder than I thought it would be, and I'm not satisfied with the results.

The long version: Creating a recommendation system using the method I employed is a lot harder than I thought it would be, and during my research I discovered that it's a lot harder than it *needs* to be.

The RMSE isn't exactly stellar, but it's passable. If I were going to actually create a recommendation system, the package I came across near the end of this project, `recosystem` would likely make up more of the algorithm I'd use. But I can certainly see the value in understanding this method.

First and foremost, it's hard to compensate for all of the very human factors that influence something as large and complex as a recommendation system for a streaming site. I've learned a lot from this project, and my future attempts will likely be more successful than this.

I've also discovered that there are a ton of resources available out there for me to use. I stuck to what I learned in this course for this particular project because I didn't want to take the risk, but ended up wasting a lot of time researching how to force the old model into working well. Given the grading curve, I know that it's possible to do better, but the margins are quite small. Given that the difference between a perfect score and the minimum passing is 0.0351, the margins are very tight.

I'm not sure what else I could have done to make this older algorithm work much better. Perhaps I should have abandoned it altogether and made something completely new using the packages and methods I discovered online.

Maybe next time.