MovieLens Report

Mike Mahon

6/7/2020

```r
library(dplyr)
library(tidyverse)
library(recosystem)
library(lubridate)
library(caret)
edx <- readRDS('./edx.rds')
validation <- readRDS('./validation.rds')
```

Introduction

In this analysis we will be creating a recommendation algorithm that predicts movie ratings using the MovieLens dataset. If you don't have the edx and validation rds files in this directory, this script will not work. You can download them **here**.

The metric we will use to evaluate our model is the RMSE, or root mean squared error.

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

The dataset

The dataset we will be using to train and test our model is called *edx*.

This dataset contains 9,000,055 movie ratings from January 9th, 1995 to January 5th, 2009. There are 10,677 different movies in our dataset and 69,878 different users.
There are six different columns: *userId*, *movieId*, *rating*, *timestamp*, *title*, and *genres*.
Here is a sample of the data:

```r
set.seed(1)
sample_n(edx, 6)
```

```
##   userId movieId rating  timestamp                            title
## 1   7131    1079      5  974646584       Fish Called Wanda, A (1988)
## 2  10963    2011      4 1098645436 Back to the Future Part II (1989)
## 3  38388     552      2 1010900879      Three Musketeers, The (1993)
## 4  37619     380      3  841491195                  True Lies (1994)
## 5  54382    1816      4  971173733        Two Girls and a Guy (1997)
## 6  57125     316      4  829047488                   Stargate (1994)
##                                    genres
## 1                            Comedy|Crime
## 2           Action|Adventure|Comedy|Sci-Fi
## 3                  Action|Adventure|Comedy
## 4 Action|Adventure|Comedy|Romance|Thriller
## 5                             Comedy|Drama
## 6                  Action|Adventure|Sci-Fi
```

The goal of the algorithm will be to minimize the RMSE (root mean squared error) for predicting the ratings in the *validation* set, which has 999,999 ratings, but we will avoid using the validation set until the very end.

In order to do this, the model will be trained and tested exclusively on the *edx* dataset. The *edx* dataset will be split into a training and test set using the following code:

```
# create test set and train set out of edx
set.seed(500)
test_index <- createDataPartition(y = edx$rating, times = 1,
                                  p = 0.2, list = FALSE)
train_set <- edx[-test_index,]
test_set <- edx[test_index,]

# for simplicity only include observations in the test set with movie and user in the train set
test_set <- test_set %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")
```

Data Insights

We have 7,200,043 rows in the training set and 1,799,979 rows in the testing set.

```
nrow(train_set)
```

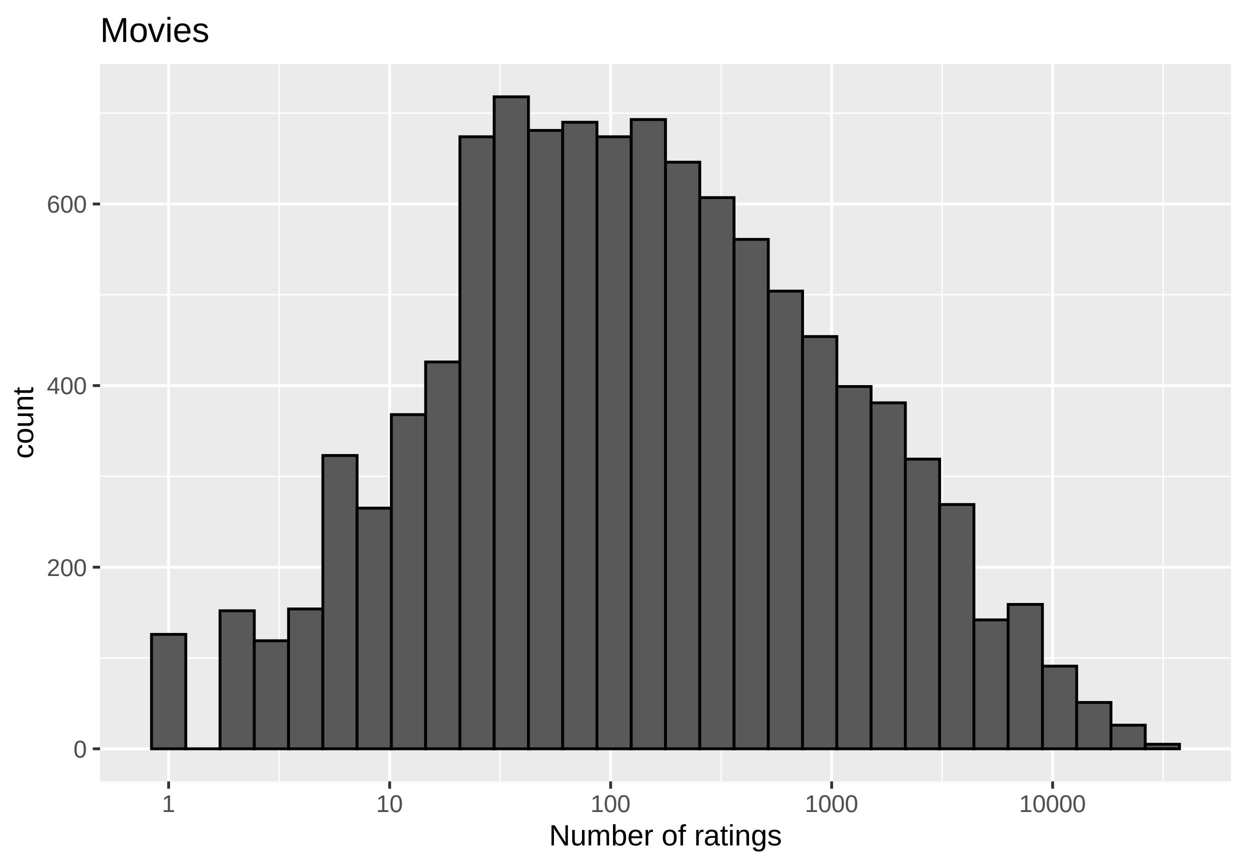
```
## [1] 7200043
```

```
nrow(test_set)
```

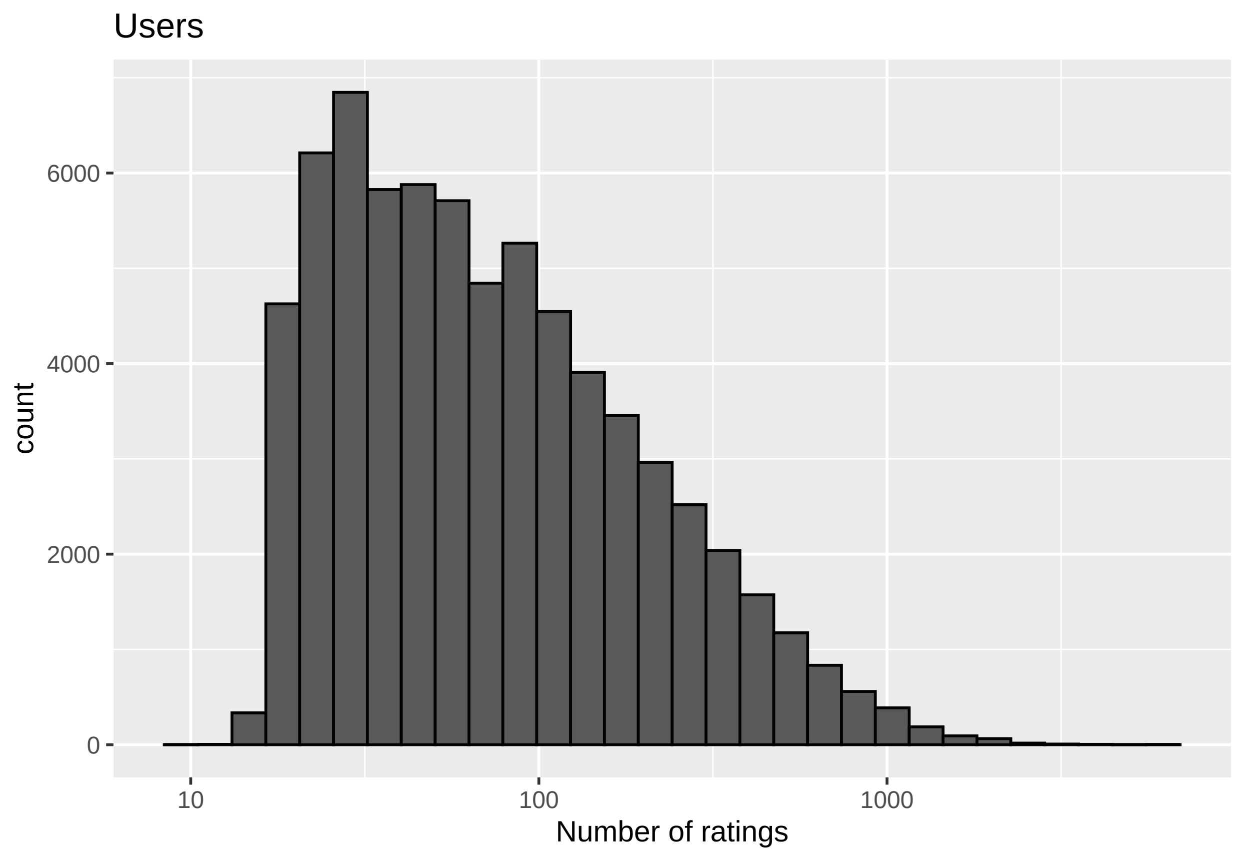
```
## [1] 1799979
```

We see most movies have a significant amount of ratings. From the graph we see that the vast majority of movies have over 10^1.5 (~*32*) ratings.
We can see that practically all users have over ten reviews.

```
# histogram showing number of ratings that movies have
edx %>%
  count(movieId) %>%
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  xlab("Number of ratings") +
  ggtitle("Movies")
```

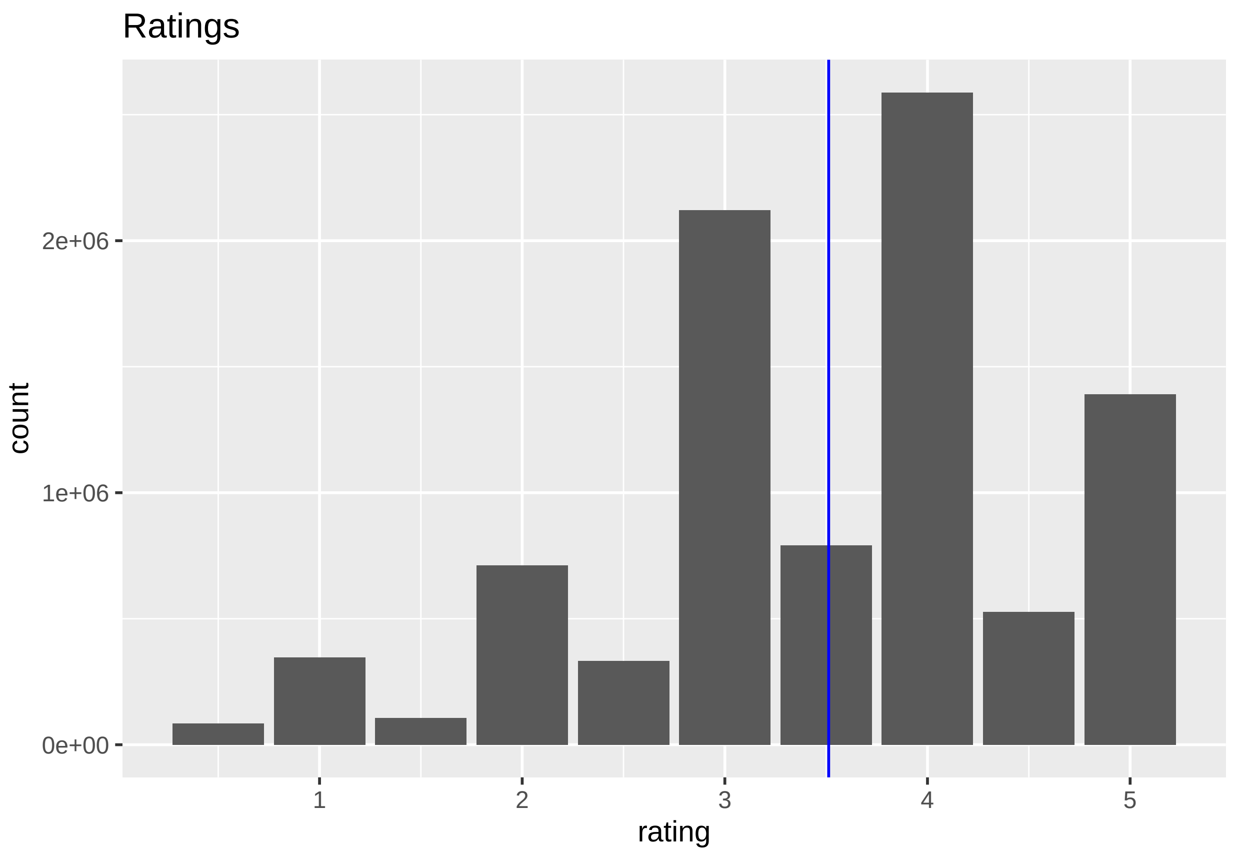


```r
# histogram showing number of ratings users have
edx %>%
  count(userId) %>%
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  xlab("Number of ratings") +
  ggtitle("Users")
```



We can see that half ratings are much less common, and the mean rating, shown by the blue line, is about 3.5.

```r
edx %>%
  ggplot(aes(rating)) +
  geom_bar() +
  ggtitle("Ratings") +
  geom_vline(xintercept = mean(edx$rating), color = "blue")
```



Methods and Analysis

Simplest Model

The most simple model we can make is to predict the same rating every time. The best value we can use for this model would be the mean, which we will be calling mu.

```r
# naive model that predicts average rating everytime
mu <- mean(train_set$rating)
naive_rmse <- RMSE(test_set$rating, mu)
rmse_results <- tibble(method = "Just the average", RMSE = naive_rmse)
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.060024

From this naive model we obtain an RMSE of 1.06. Let's see if we can improve upon that.

The Movie Effect

As we all know, some movies are liked more than others. We can capture this effect by looking at how much each movie typically differs from the mean and adding this amount to the mean.

```r
# get the average amount each movie differs from the mean
movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu))

# predict ratings in the test set using the mean plus the b_i (movie effect)
predicted_ratings <- mu + test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  .$b_i

# compute the RMSE
model_1_rmse <- RMSE(predicted_ratings, test_set$rating)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Movie Effect Model",
                                     RMSE = model_1_rmse ))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0600241
Movie Effect Model	0.9438779

Great! We were able to lower our RMSE down to 0.944!

The User Effect

Just like how different movies have different typical ratings, different users might rate differently than others. Some users may be more generous with their ratings than others, and some may be more stingy. Let's see if we can lower our RMSE by including a user effect.

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

# predict ratings with mean (mu), movie effect (b_i), and user effect (b_u)
predicted_ratings <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  .$pred

# compute rmse
model_2_rmse <- RMSE(predicted_ratings, test_set$rating)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Movie + User Effects Model",
                                     RMSE = model_2_rmse ))
rmse_results %>% knitr::kable()
```

method	RMSE
Just the average	1.0600241
Movie Effect Model	0.9438779
Movie + User Effects Model	0.8660586

Now we're down to 0.866. Not bad at all!

Regularization

Before we move on, there is one problem we need to address. Look at the movies and users with the biases that are highest in magnitude.

```r
# show top 6 movies with largest magnitude of b_i, or largest abs(b_i)
train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(movieId, title) %>%
  summarize(b_i = first(b_i), n = n()) %>%
  arrange(desc(abs(b_i))) %>%
  head()
```

```
## # A tibble: 6 x 4
## # Groups:   movieId [6]
##   movieId title                                  b_i     n
##     <dbl> <chr>                                <dbl> <int>
## 1    5805 Besotted (2001)                      -3.01     1
## 2    8394 Hi-Line, The (1999)                  -3.01     1
## 3    8458 To Each His Own (1946)               -3.01     1
## 4    8764 Under the Lighthouse Dancing (1997)  -3.01     1
## 5   61768 Accused (Anklaget) (2005)            -3.01     1
## 6   64999 War of the Worlds 2: The Next Wave (2008) -3.01     2
```

```r
# do the same for users
train_set %>%
  left_join(user_avgs, by='userId') %>%
  group_by(userId) %>%
  summarize(b_u = first(b_u), n = n()) %>%
  arrange(desc(abs(b_u))) %>%
  head()
```

```
## # A tibble: 6 x 3
##   userId   b_u     n
##    <int> <dbl> <int>
## 1  13496 -3.39    15
## 2  48146 -3.21    17
## 3  49862 -3.14    14
## 4  62815 -2.95    15
## 5  63381 -2.95    15
## 6  42019 -2.88    22
```

As you can see the movies with largest b_i's in magnitude are really obscure movies that we have only one or two reviews for. The users with the largest b_u's in magnitude have only around 15 ratings, which is far less than what we'd expect given the graph from earlier.

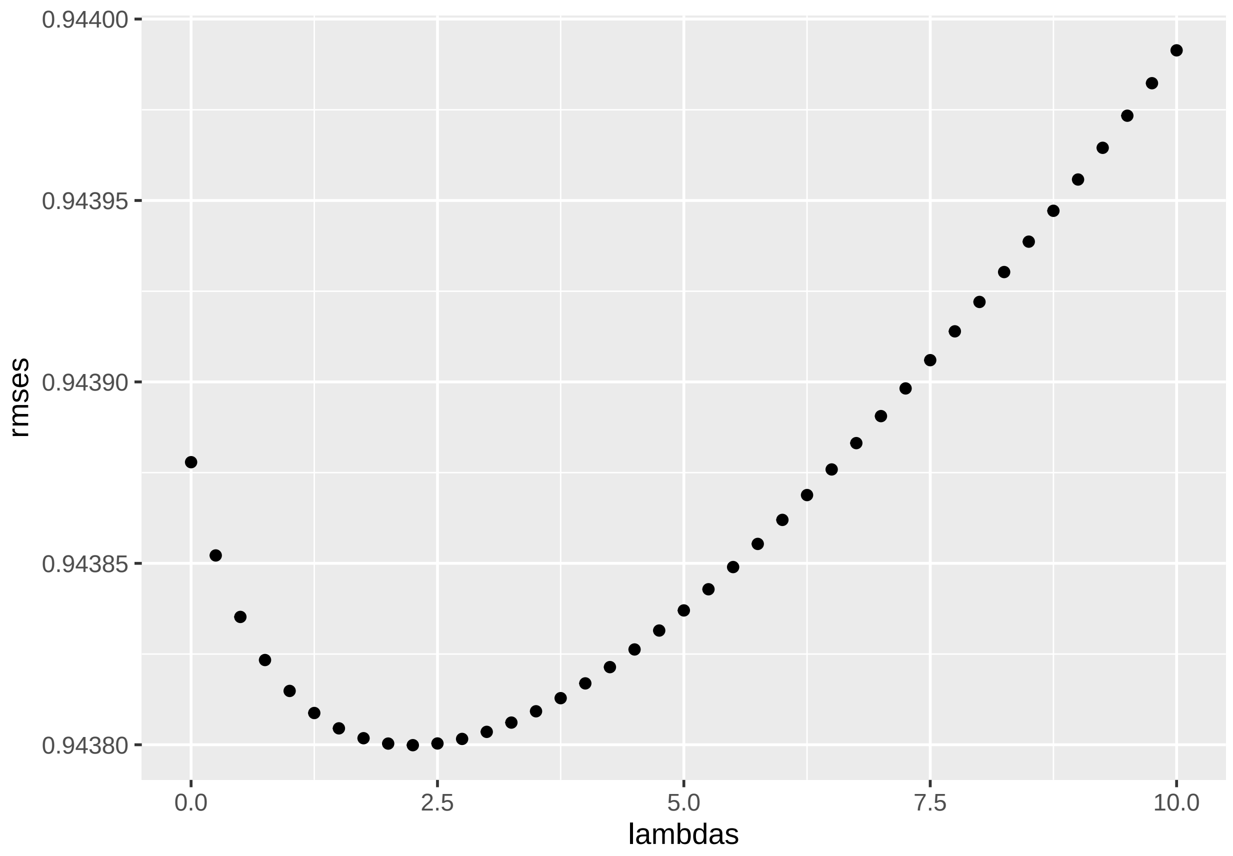
In order to solve this problem we will use regularization, which is a technique for shrinking the effects for features we don't have many observations for. We'll start by doing this for the movie effect.

Regularized Movie Effect

Lambda is the tuning parameter used in regularization. The larger the lambda, the more we shrink. Let's find the lambda that minimizes our RMSE for the movie effect.

```
# function that takes lambda as an argument and computes the regularized movie effect for each movie
get_movie_effect_reg <- function(l) {
  train_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n() + l))
}

# find best lambda for movie effect
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l) {
  movie_avgs <- get_movie_effect_reg(l)
  predicted_ratings <- test_set %>%
    left_join(movie_avgs, by="movieId") %>%
    mutate(pred = mu + b_i) %>%
    .$pred
  RMSE(predicted_ratings, test_set$rating)
})
qplot(lambdas, rmses)
```



```r
# best lambda for calculating movie effect
b_i_lambda <- lambdas[which.min(rmses)]

# save regularized movie biases
movie_avgs_reg <- get_movie_effect_reg(b_i_lambda)

rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularized Movie Effect Model",
                                 RMSE = min(rmses)))
rmse_results %>% knitr::kable()
```

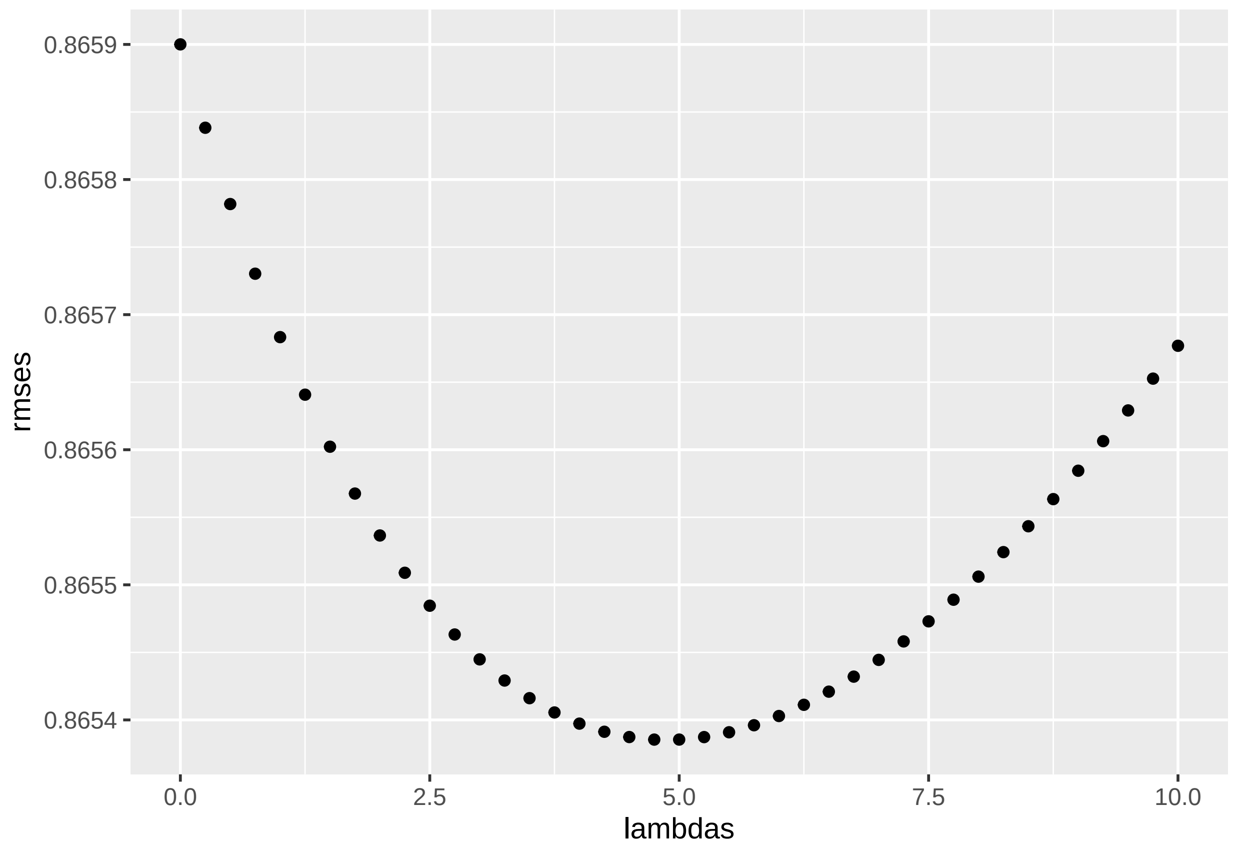
method	RMSE
Just the average	1.0600241
Movie Effect Model	0.9438779
Movie + User Effects Model	0.8660586
Regularized Movie Effect Model	0.9437999

As we can see from the graph, a lambda of 2.25, minimizes our RMSE. Regularization was able to reduce our RMSE by about 0.0001, so not nothing, but not very significant.

Regularized User Effect

Now let's do the same thing for the user effect.

```r
# given lambda, compute regularized user effect for each user
get_user_avgs_reg <- function(l) {
  train_set %>%
    left_join(movie_avgs_reg, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - mu - b_i)/(n() + l))
}
# find best lambda for user effect regularization
rmses <- sapply(lambdas, function(l) {
  user_avgs <- get_user_avgs_reg(l)
  predicted_ratings <- test_set %>%
    left_join(movie_avgs_reg, by="movieId") %>%
    left_join(user_avgs, by="userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
    .$pred
    RMSE(predicted_ratings, test_set$rating)
})
# plot lambdas
qplot(lambdas, rmses)
```



```r
# save best lambda
b_u_lambda <- lambdas[which.min(rmses)]
# save regularized user effects using the best lambda
user_avgs_reg <- get_user_avgs_reg(b_u_lambda)

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

method	RMSE
Just the average	1.0600241
Movie Effect Model	0.9438779
Movie + User Effects Model	0.8660586
Regularized Movie Effect Model	0.9437999
Regularized Movie + User Effect Model	0.8653854

Nice! Using a lambda of 4.75 we got the RMSE down to 0.865!

Residuals

Moving forward we will be trying to estimate the residuals, which in our case is the movie rating minus the regularized user and movie effects. Let's update our training and test set to include a column for the residuals.

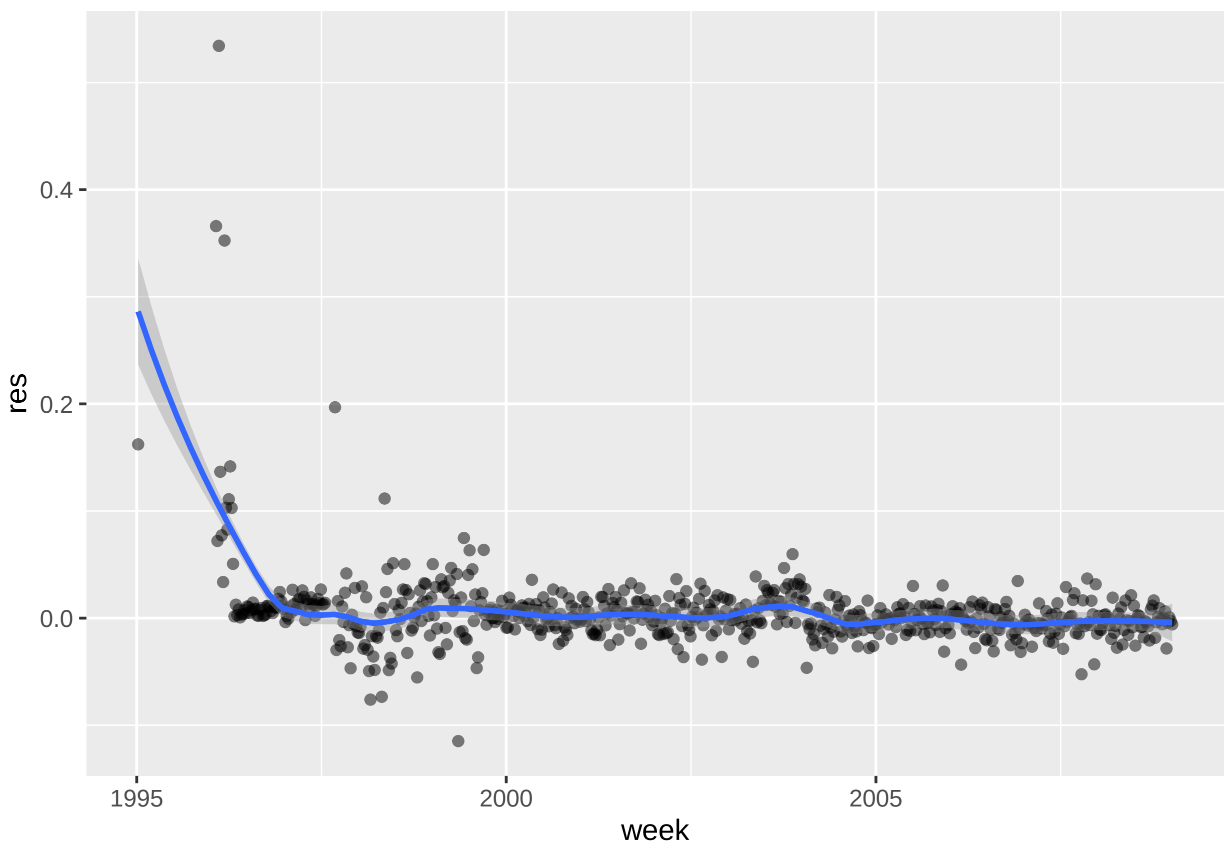
```r
# compute residuals for test and training set
train_set <- train_set %>%
  left_join(user_avgs_reg, by="userId") %>%
  left_join(movie_avgs_reg, by="movieId") %>%
  mutate(res = rating - mu - b_i - b_u)

test_set <- test_set %>%
  left_join(user_avgs_reg, by="userId") %>%
  left_join(movie_avgs_reg, by="movieId") %>%
  mutate(res = rating - mu - b_i - b_u)
```

Date effect

Maybe the date or time of year effects how users rate movies. To explore this, let's plot the residuals of our training set as a function of the date.

```r
# make a loess graph plotting the mean residuals by week
train_set %>%
  mutate(week = round_date(as_datetime(timestamp), unit = "week")) %>%
  group_by(week) %>%
  summarize(res = mean(res)) %>%
  ggplot(aes(week, res)) +
  geom_point(alpha=0.5) +
  geom_smooth(method="loess", span=0.2)
```



We can see a big spike at the beginning, but the data is very sparse at that time, so modeling that won't have a significant effect on our RMSE. The rest of our line looks a little wiggly at certain points, but it never deviates far from zero. It looks like we won't be able to reduce our RMSE by very much by including the date in our model. For simplicity, we will ignore the date in our model.

Matrix Factorization

Before we proceed, let's rethink of our dataset as a matrix. Each row represents a single user, the columns represent a movie, and each cell will contain our residual.

```r
# get number of unique users and movies
length(unique(train_set$userId))
```

```
## [1] 69878
```

```r
length(unique(train_set$movieId))
```

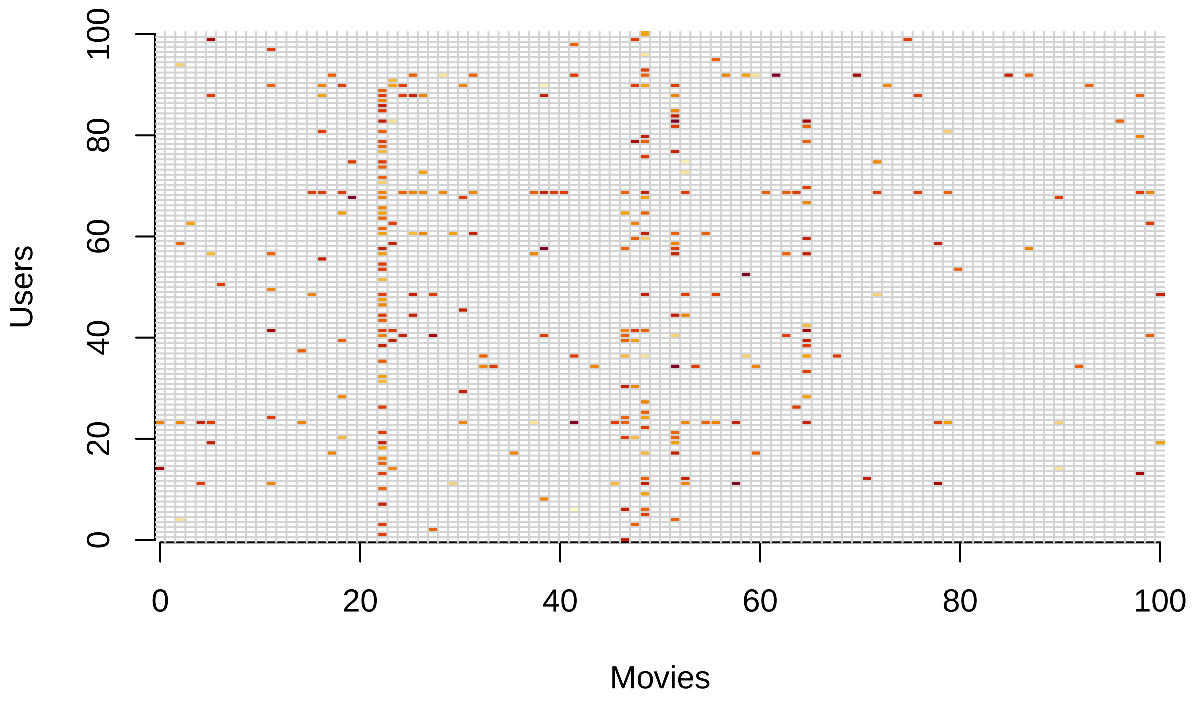
```
## [1] 10650
```

If we turned our training set into this matrix of residuals we would have 69,878 rows and 10,650 columns. This would make a matrix with 744,200,700 (69,878 x 10,650) cells. If we tried to make this using the spread function it would crash R. To better visualize this matrix of residuals let's look at it for a random sample of 100 users and 100 movies.

```r
# sample 100 users
set.seed(100)
userIds <- sample(unique(train_set$userId), 100)

# use spread function to put it in matrix format and select 100 random columns
set.seed(100)
m <- train_set %>%
  filter(userId %in% userIds) %>%
  select(userId, movieId, res) %>%
  spread(movieId, res) %>%
  select(-userId) %>%
  select(sample(ncol(.), 100)) %>%
  as.matrix()

# create image of matrix of residuals
image(m, xlab = "Movies", ylab = "Users", axes = FALSE)
axis(1, at=seq(0,1,0.2), labels=seq(0,100,length = 6))
axis(2, at=seq(0,1,0.2), labels=seq(0,100,length = 6))
grid(100,100, lty = 1)
```



The colored cells represent our residuals, and the empty cells are N/A's. As we can see this is a very sparse matrix. The vast majority of cells are N/A's. Using matrix factorization we can predict the residuals for the N/A's.

Matrix factorization factorizes the matrix into two (or more) lower dimensional rectangular matrices such that when you multiply them you get back the original matrix.

To see how we would model this, let's look at our current model. Right now our model uses the mean (mu) and user and movie effects (b_u & b_i) to predict the rating (Y_u_i). We will add to this model a vector p, which represents latent user factors, and a vector q which represents item or movie factors. Such that our new model looks like this:

Y_u_i = mu + b_u + b_i + p_u * q_i + E_i_j

E_i_j is the error term, which just represents random variability.

We want to find a matrix P and a matrix Q that when multiplied together will approximate our matrix of residuals. In order to do this, we will be using a package called **recosystem**. Recosystem's functions will abstract most of the complexity away from us.

To start, **recosystem** requires us to specify the data set as an object of class *DataSource*. We will use the data_memory() function in order to do this.

```
# specify data set for recosystem's DataSource
train_data <- data_memory(user_index = train_set$userId, item_index = train_set$movieId,
                          rating = train_set$res, index1 = TRUE)

test_data <- data_memory(user_index = test_set$userId, item_index = test_set$movieId, index1 = TRUE)
```

Now we will create the model object for recosystem's computation.

```r
# create model object
r = Reco()
```

Next we will use cross-validation to find the best tuning parameters for our model. (WARNING: This step takes a very long time, so you may want to skip this step and manually enter the parameters we obtain from tuning in the r$train method, which will be printed here.)

```r
# select best tuning parameters
set.seed(123)
opts = r$tune(train_data, opts = list(dim = c(10, 20, 30), lrate = c(0.1, 0.2),
                                      costp_l1 = 0, costq_l1 = 0,
                                      nthread = 1, niter = 10))
print(opts$min)
```

```
## $dim
## [1] 30
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0.01
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.1
##
## $lrate
## [1] 0.1
##
## $loss_fun
## [1] 0.8017479
```

Now we will train our model using our training data and the optimized tuning parameters from the previous step.

```r
# train
set.seed(123)
r$train(train_data,  opts = c(opts$min, nthread = 1, niter = 10))
```

```
## iter      tr_rmse          obj
##    0       0.8595   5.5631e+06
##    1       0.8365   5.1716e+06
##    2       0.8197   5.0289e+06
##    3       0.8030   4.9019e+06
##    4       0.7872   4.7832e+06
##    5       0.7732   4.6788e+06
##    6       0.7616   4.5990e+06
##    7       0.7515   4.5287e+06
##    8       0.7429   4.4726e+06
##    9       0.7354   4.4224e+06
```

Our final step is to predict the residuals of our test data using the matrix factorization computed by **recosystem**. Then we will add them to our mu, and movie and user effects to predict the ratings. Lastly, we should change any values over the maximum and below the minimum to the maximum and minimum values respectively. Note: prior to Febuary 12, 2003 there were no half ratings, so the minimum value before then was 1 and 0.5 afterwards.

```r
# predict
predicted_ratings <- r$predict(test_data, out_memory()) + mu + test_set$b_i + test_set$b_u

# change ratings over max to max
ind <- which(predicted_ratings > 5)
predicted_ratings[ind] <- 5

# find first half rating
half_rating_start <- train_set %>%
  filter(rating %% 1 == 0.5) %>%
  pull(timestamp) %>%
  min()

# change values below one from dates before half ratings to a one.
ind <- which(predicted_ratings < 1 & test_set$timestamp < half_rating_start)
predicted_ratings[ind] <- 1

# change values below 0.5 from dates after half ratings to a 0.5
ind <- which(predicted_ratings < 0.5 & test_set$timestamp >= half_rating_start)
predicted_ratings[ind] <- 0.5

# compute RMSE
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularized Movie + User Effect + Matrix Factorization Model",
                                      RMSE = RMSE(predicted_ratings, test_set$rating)))

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

method	RMSE
Just the average	1.0600241
Movie Effect Model	0.9438779
Movie + User Effects Model	0.8660586
Regularized Movie Effect Model	0.9437999
Regularized Movie + User Effect Model	0.8653854
Regularized Movie + User Effect + Matrix Factorization Model	0.7977048

Excellent! We got the RMSE down to 0.798!

Results

Up until now, we've trained and tested our data exclusively on the edx dataset. Now to compute our final result we will use the validation set just like how we were using the test set before, except we won't be ignoring observations of users and movies which aren't in our training data. This may have an upward effect on our RMSE.

```
# add b_i’s and b_u’s to the validation set
validation <- validation %>%
  left_join(user_avgs_reg, by="userId") %>%
  left_join(movie_avgs_reg, by="movieId") %>%
  setNames(gsub(’\\.x$’, ’’, names(.))) %>%
  mutate(b_i = ifelse(is.na(b_i), 0, b_i), b_u = ifelse(is.na(b_u), 0, b_u))

# make validation data a DataSource for recosystem
validation_data <- data_memory(user_index = validation$userId, item_index = validation$movieId, index1

# predict residuals for validation and add them to the mean and user and movie effects
predicted_ratings <- r$predict(validation_data, out_memory()) + mu + validation$b_i + validation$b_u

# change predictions over max to the max
ind <- which(predicted_ratings > 5)
predicted_ratings[ind] <- 5

# change predictions under min to the min
ind <- which(predicted_ratings < 1 & validation$timestamp < half_rating_start)
predicted_ratings[ind] <- 1

ind <- which(predicted_ratings < 0.5 & validation$timestamp >= half_rating_start)
predicted_ratings[ind] <- 0.5

# compute final RMSE
RMSE(predicted_ratings, validation$rating)
```

```
## [1] 0.7971095
```

Very Good! Our final model obtained an RMSE of 0.7971095! This model, which combined the mean, regularized movie effect, regularized user effect, and matrix factorized features performed very well!

Conclusion

In our final model we used the mean, the regularized movie effect, the regularized user effect, and latent user and movie factors obtained from matrix factorization in order to predict ratings in the validation set. Our final RMSE was about 0.797. If we wanted to improve upon our current methodology we could have tried tuning the matrix factorization further by trying out more parameters in **recosystem**’s r$tune function. However, this would have required even longer compute time. Additionally, **recosystem** uses stochastic gradient methods to solve the factorized matrix, which uses randomization, so the optimal solution will change slightly every time. Perhaps, there is a way we could obtain a more exact result. There was also a lot of information available to us that was ignored from our model. We did not include the genres column in this analysis, and the timestamp was for the most part ignored. Perhaps if we wanted to improve upon this model, we could look for seasonal or yearly effects on ratings. Additionally, we could try to compute the effect different genres have on the ratings for different users.
There are certainly more components that could be added to our model, but in the end we were able to achieve an RMSE below 0.8, which is a long way away from our first RMSE that couldn’t even get below 1. Thank you for taking the time to read through my project, and I hope that you had a good time!