HarvardX Movielens Project

David Pershall

7/9/2021

Introduction

Recommendation systems have revolutionized e-commerce. Companies like Amazon sell all manner of products to their customers and actively collect reviews which are stored in massive databases. Machine learning tools are then used to recommend products across the full range of customers in a tailored and highly specific way. This process enables data driven companies to increase their pool of satisfied returning consumers. This same process is also used to great effect by media companies such as Apple Music and Netflix.

In 2006, Netflix issued a challenge to data scientists worldwide. It was quite simple, $1 million dollars would be awarded to the team or individual that could improve the recommendation algorithm by 10%. The winners were announced in 2009, and you can read about the prize-winning algorithm by following the link below.

https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf

Unfortunately, Netflix does not make their data publicly available. However, the GroupLens research lab has data which is made available under the Movielens moniker. The goal of this project is to use the knowledge I have gained during HarvardX's Data Science Professional Certificate program to evaluate the strategies used by the team that won the Netflix challenge, and compose a recommendation system on a subset of the Movielens data.

The subset will include 10 million ratings on approximately 10,000 movies from around 70,000 users. In order to evaluate the efficacy of the different methods I will deploy, I will use the Root Mean Squared Error of my predicted ratings against a validation set of actual ratings.

Some of the techniques below were used during our lectures by professor Rafael Irizarry. You can find our class notes in his online book entitled "Introduction to Data Science" by following the link below.

https://rafalab.github.io/dsbook/

For this capstone project, we were instructed to build upon the techniques shown in his book and lectures. It is worth noting that in order to receive an A on our project, our final validation/s must score an RMSE of below .86490.

I will reach for my final validation to achieve a RMSE below 0.80.

Libraries

Install the required packages and load the libraries

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

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

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

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

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

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

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

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

library(tidyverse)
library(caret)
library(data.table)
library(gridExtra)
library(broom)
library(ggrepel)
library(recommenderlab)
library(recosystem)
```

Pulling and cleaning the data

```r
# MovieLens 10M dataset:
# https://grouplens.org/datasets/movielens/10m/
# http://files.grouplens.org/datasets/movielens/ml-10m.zip

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(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres))


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

Exploration

First I want to examine how many unique users and distinct movies are included in the data set.

```
movielens %>% summarize(n_users = n_distinct(userId),
                        n_movies = n_distinct(movieId))
```

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

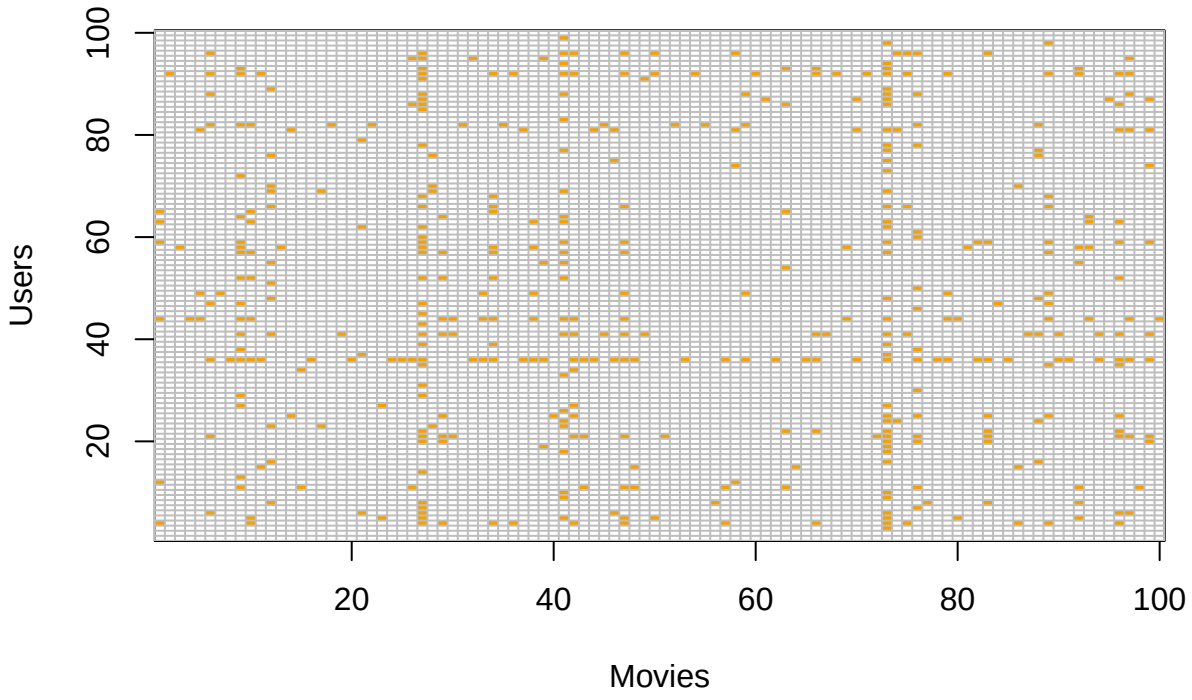
This implies that there could be a total of 746,087,406 ratings if every user watched and rated every movie. However, when I look at the dimensions of the data set, there are only 10,000,054 actual ratings given. This means that there are a great deal of movies certain users have not watched and/or didn't bother rating.

```
dim(movielens)
```

```
## [1] 10000054        6
```

Sometimes it is helpful to visualize the data before beginning to work out a solution. The movielens data could be thought of as a matrix with rows of users, columns of movies, and each cell holding a rating for the specific user-movie combination. I will use some functions to coerce the dataset to this form later in the project, but for now I will use a sample of 100 movies and 100 users from the data. The yellow indicates a movie-user combination for which an actual rating has been given. A recommendation system essentially takes the blank squares and generates a rating that is as close to the rating the actual user would give to the paired movie and fills the empty spaces accordingly.

```
users <- sample(unique(movielens$userId), 100)
movielens %>% 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")
```



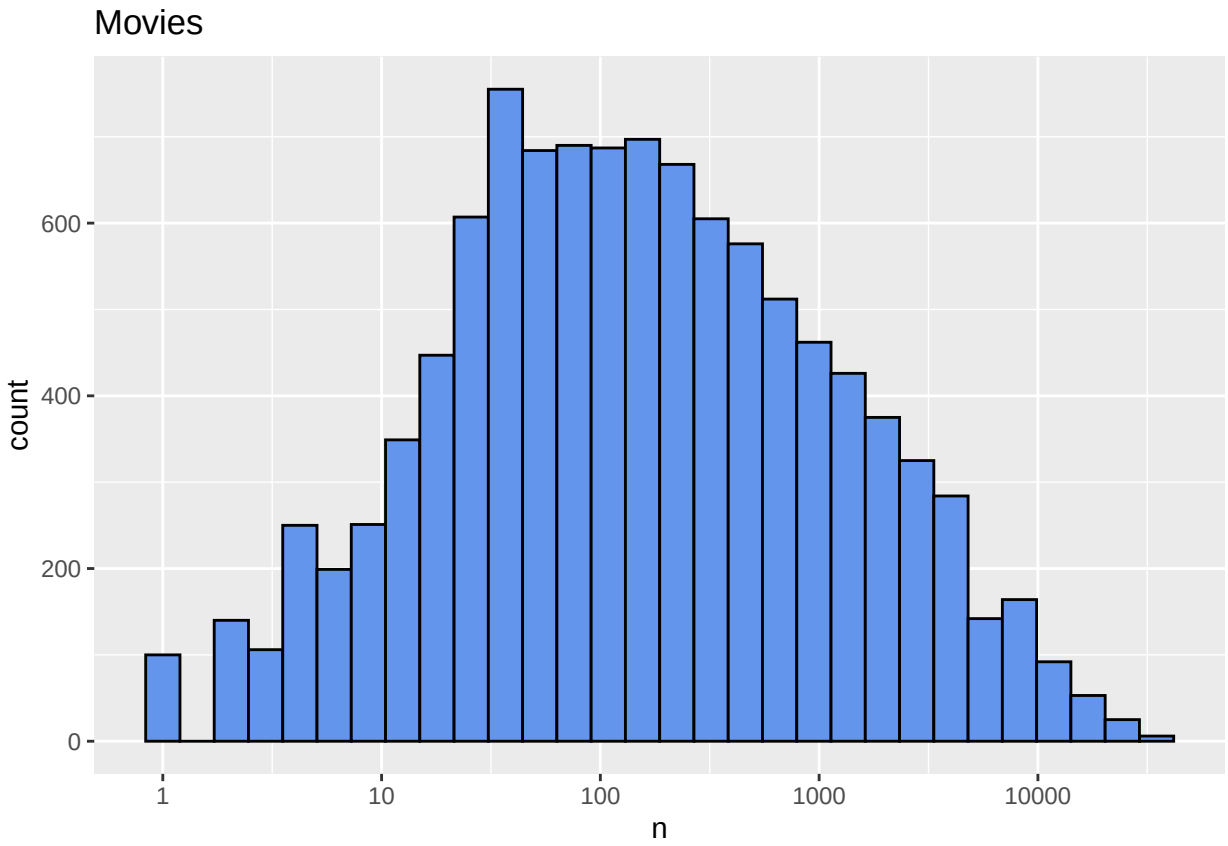
If the goal is to predict a rating y for movie i by user u, then theoretically, all ratings related to movie i by user u can be used as predictors. In addition to this, information from movies that are like movie i or users that are similar to user u can also help in the process. Therefore, it is possible that the entire data set could serve as predictors for each individual cell.

Ok, so where to begin?

Ratings Distribution

Let's have a look at the distribution of the ratings.

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



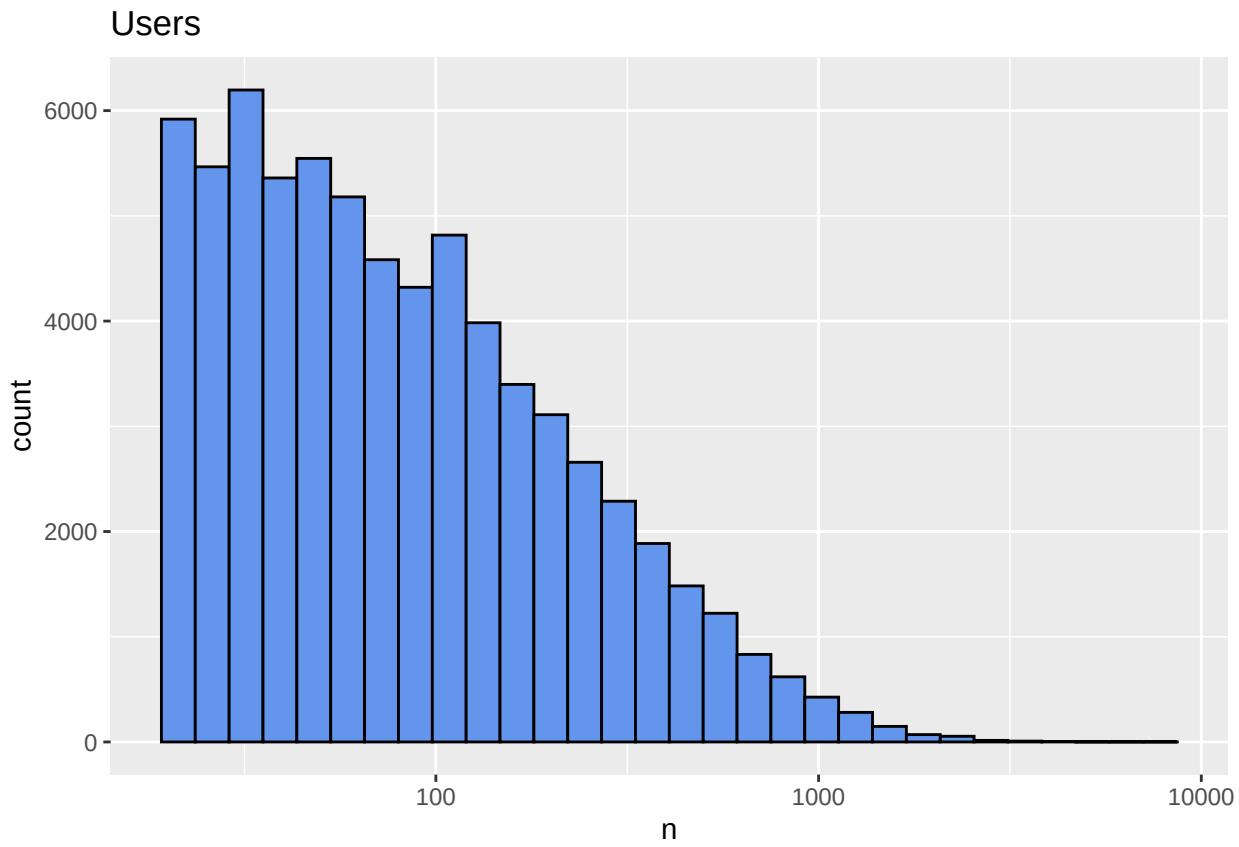
The first thing to notice in the graph above is that some movies are rated quite often, while others are rated just a few times. This distribution of ratings doesn't come as a surprise. After all, many people watch big budget blockbuster films and comparatively very few watch independent films.

Users histogram

Another observation comes from the users. In the histogram below, it is evident that a portion of users have rated over a thousand movies, while others have only rated a few.

```
movielens %>%
  dplyr::count(userId) %>%
```

```
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black", fill = "cornflowerblue") +
  scale_x_log10() +
  ggtitle("Users")
```



Time

Could there be a time effect? The code below pulls the year, month, and day from the timestamp and uses mutate to add a years since release variable to the movielens set.

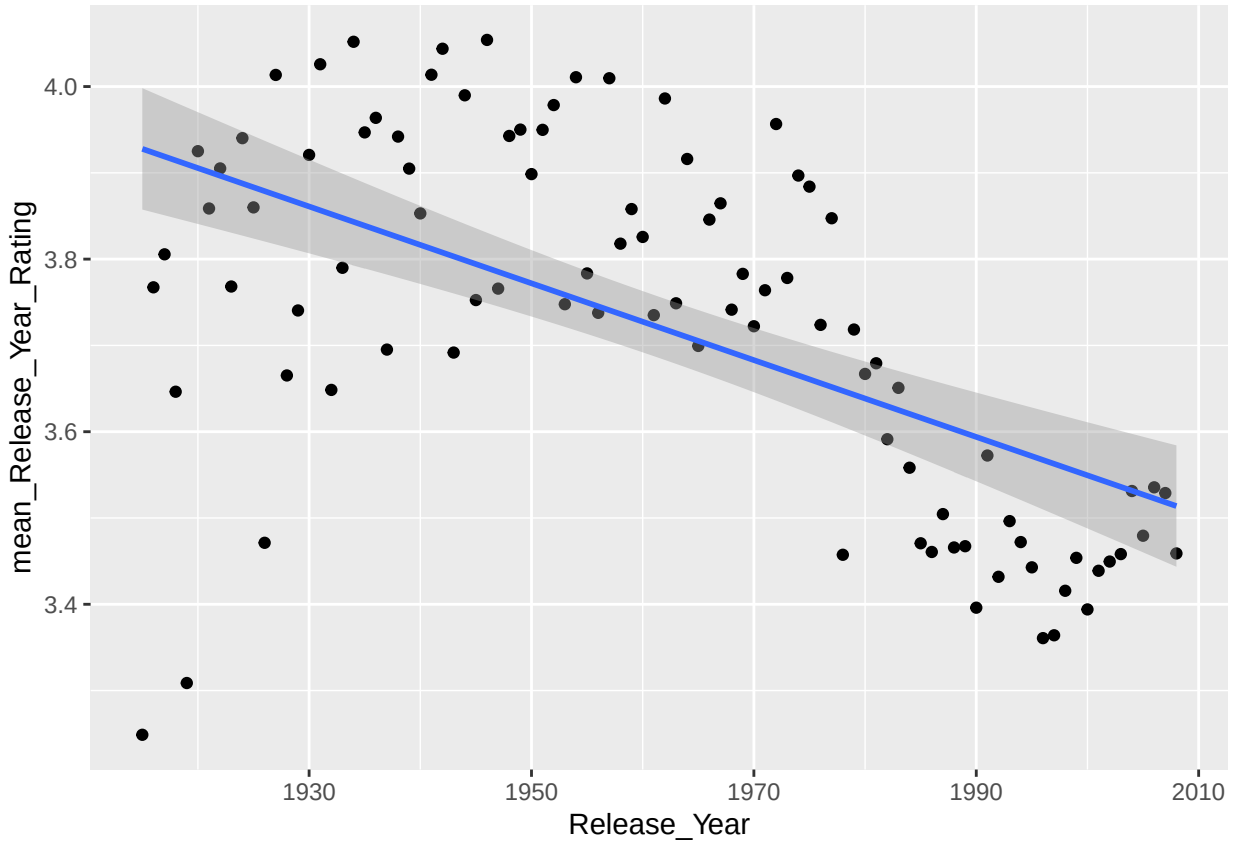
```
movielens <- movielens %>%
  mutate(year = as.numeric(format(as.Date(as.POSIXct(timestamp,
                                                     origin = "1970-01-01"),
                                          format ="%Y/%m/%d"), "%Y")))

pattern <- "\\s[(][0-9]{4}[)]"

# Add Release Year and Years Since Release to the data
movielens <- movielens %>%
  mutate(Release_Year = as.numeric(substr(str_extract(string = title,
                                                      pattern = pattern),3,6)),
         years_since_release = year - Release_Year)

# Double check the above achieved the desired effect
glimpse(movielens)
```

```
## Rows: 10,000,054
```

```
## Columns: 9
## $ userId              <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
## $ movieId             <dbl> 122, 185, 231, 292, 316, 329, 355, 356, 362, 364, ~
## $ rating              <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,~
## $ timestamp           <int> 838985046, 838983525, 838983392, 838983421, 838983~
## $ title               <chr> "Boomerang (1992)", "Net, The (1995)", "Dumb & Dum~
## $ genres              <chr> "Comedy|Romance", "Action|Crime|Thriller", "Comedy~
## $ year                <dbl> 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 19~
## $ Release_Year        <dbl> 1992, 1995, 1994, 1995, 1994, 1994, 1994, 1994, 19~
## $ years_since_release <dbl> 4, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3,~
```

Now I can plot the average rating based on release year with some lm smoothing.

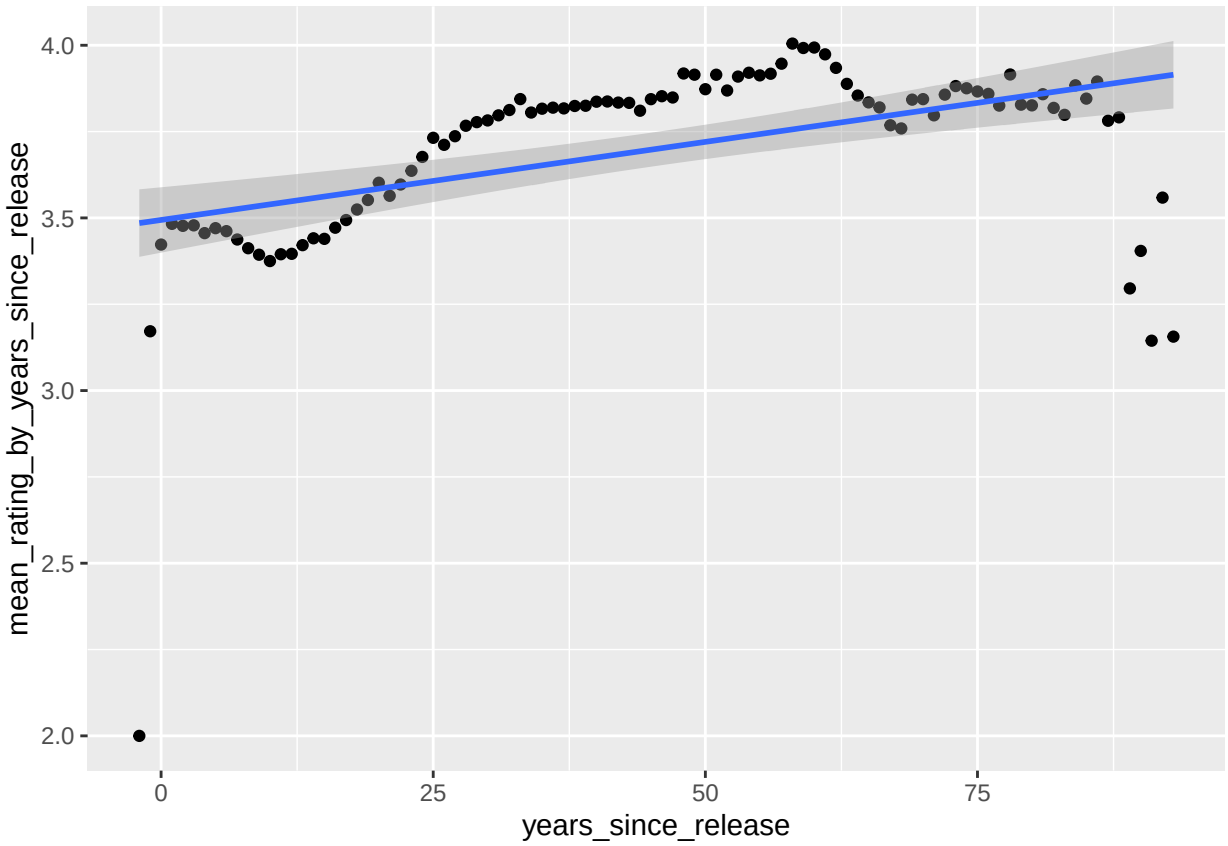
```
movielens %>% group_by(Release_Year) %>%
  summarise(mean_Release_Year_Rating = mean(rating)) %>%
  ggplot(aes(Release_Year,mean_Release_Year_Rating)) + geom_point() + geom_smooth(method = "lm")
```



So, we can observe that there seems to be a relationship between the release year and the average rating given to the movies.

Now to think of this time effect in another way, I will categorize by the years that have passed since the release of the movie and look at the average rating for the time differences.

```
movielens %>% group_by(years_since_release) %>%
  summarise(mean_rating_by_years_since_release = mean(rating)) %>%
  ggplot(aes(years_since_release, mean_rating_by_years_since_release)) + geom_point() + geom_smooth(meth
```



Interesting, the time since a movie was released seems to have a net positive effect on the average rating. I will keep this in mind for the predictive algorithms later in the project.

The splits

Now that I have a general idea of the challenge ahead it is time to split the data. To prevent over training, the following code creates a final hold out set called "validation" which will be 10% of the Movielens data. The validation set will not be used until the final test.

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

# Make sure userId and movieId in validation set are also in edx set
validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

# Add rows removed from validation set back into edx set
removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)

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

Now it is now time to split the edx set into train and test sets so that I can evaluate different algorithms before performing my final test on the hold out "validation" set.

```
set.seed(755, sample.kind = "Rounding")
test_index2 <- createDataPartition(y = edx$rating, times = 1,
                                   p = 0.1, list = FALSE)
train_set <- edx[-test_index2,]
test_set <- edx[test_index2,]

# Make certain that users and movies in the test set are included in the train set
test_set <- test_set %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")
```

Model Evaluation Metric

To evaluate the different models or to see how well each strategy performs comparatively, I need a loss function. The Netflix challenge chose a winner based on the residual mean squared error of the predicted ratings against the actual ratings in the test set. If one defines $y_{u,i}$ as the rating by user u for movie i and $\hat{y}_{u,i}$ as the predicted rating by user u for movie i, then the residual mean squared error can be calculated with this formula.

$$RMSE = \sqrt{\frac{1}{N}\sum_{u,i}(\hat{y}_{u,i} - y_{u,i})^2}$$

In the formula above, N is a number of user movie combinations and the sum occurs over all these combinations. Remember that RMSE can be interpreted similarly to standard deviation. It is emblematic of the error made when a rating is predicted. If the RMSE is over 1 then the predicted rating is typically off by one or more stars.

The following is a function that computes RMSE for a vector of ratings and their corresponding predictors. This will serve as the evaluation metric, and as previously stated, my goal is an RMSE below 0.8.

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

Models

The simplest model would predict the same rating for all movie-user combinations and would look like this.

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

Here, ϵ stands for independent errors sampled from the same zero-centered distribution and μ is the actual rating for all movies and users. The least squares estimate of μ minimizes the residual mean squared error, and for this instance, that is the overall ratings average. The code below computes this average.

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

```
## [1] 3.512418
```

Naive RMSE - Just the Average

So, the RMSE of the naive random squares model can be computed like this.

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

```
## [1] 1.060007
```

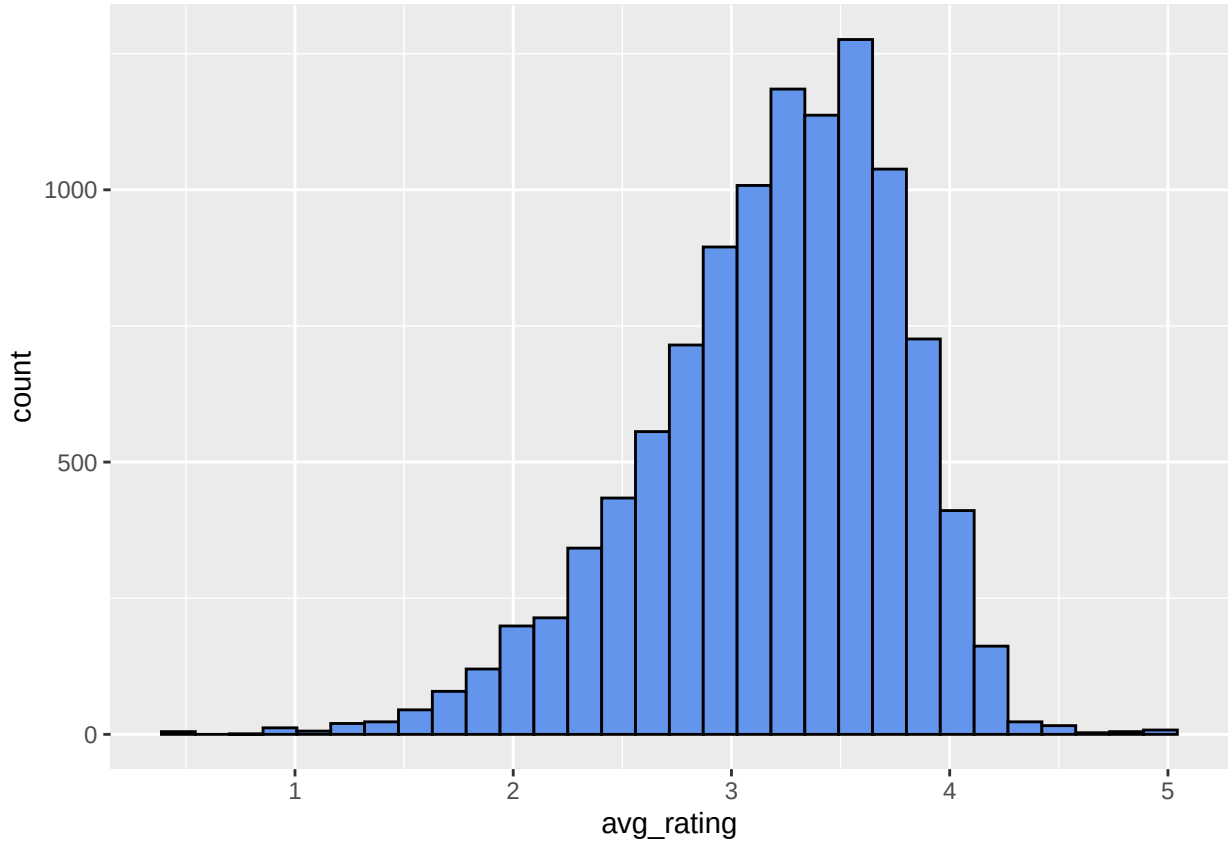
When all unknown ratings are predicted with this average, the result is a residual mean squared error of ≈ 1.06. The code below creates a table to keep track of the RMSE on all models that I will examine in this project.

```
RMSE_results <- tibble(Method = "The Average", RMSE = naive_rmse)
RMSE_results %>% knitr::kable()
```

Method	RMSE
The Average	1.060007

Now, not all movies are created equally. Or put another way, some movies average a higher rating across all users than others. The histogram below is the distribution of the average rating for movies that have been rated by more than 100 users.

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



This confirms the idea that not all movies are created equal.

The Movie Effect Model

A new model can now be written which will include the term b_i to represent the average rating for movie i. The "b's effect model" or movie effect model looks like this.

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

The lm function would use least squares to estimate the b's, but the vector would be too large to run on my computer and take too much time to run via cloud computing.

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

In this instance, $\hat{b}_i$, the least squares estimate, is the average of $y_{u,i}$ minus the overall mean for each movie i. So it can be computed in a way that saves on hardware use and run time.

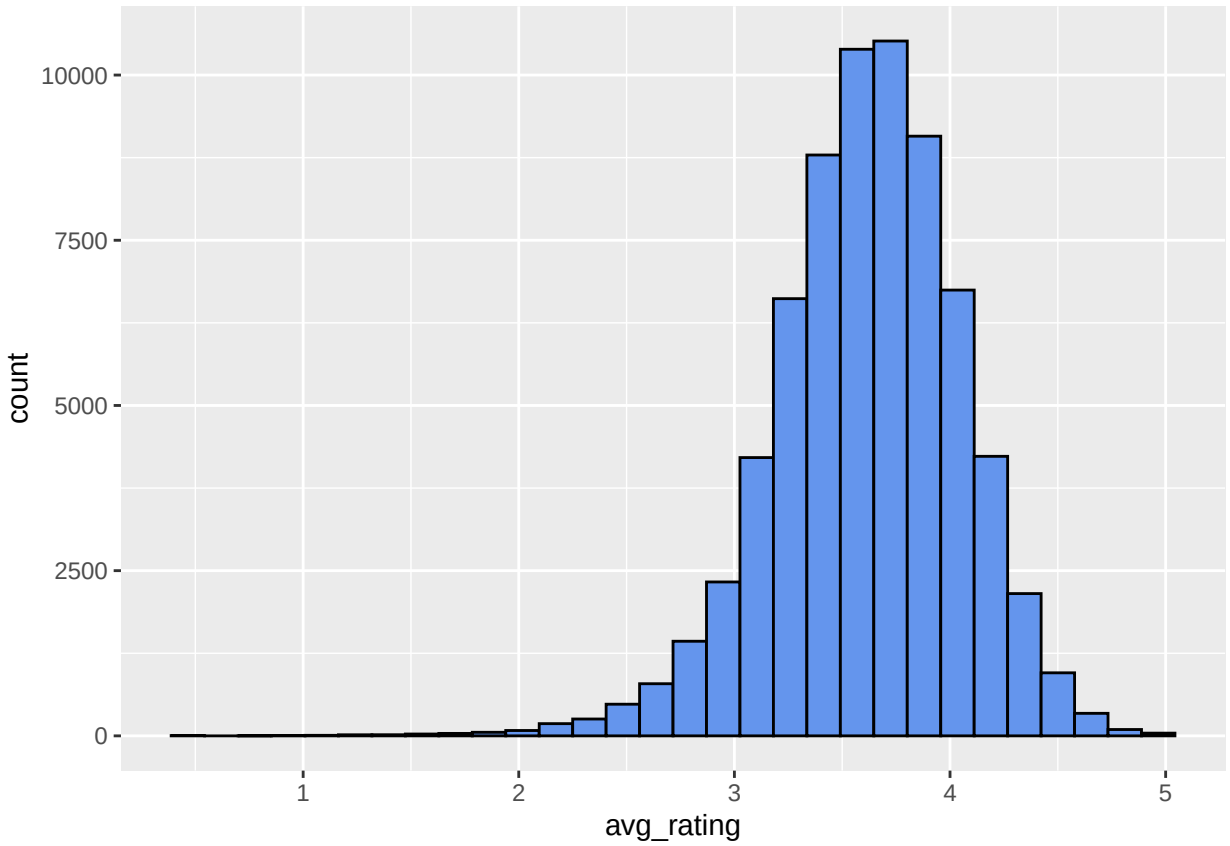
```
# In order to keep the code as clean as possible, _hat notation has been dropped

mu <- mean(train_set$rating)

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

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

model_2_rmse <- RMSE(predicted_ratings, test_set$rating)
RMSE_results <- bind_rows(RMSE_results,
                          tibble(Method = "Movie Effect Model", RMSE = model_2_rmse ))
RMSE_results %>% knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103

Movie + User Effects Model

Now to explore the users. It would be helpful to see a distribution of the average rating all active users give to movies they watch. The code below displays a histogram of the average rating from users that have rated more than 100 movies.

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



So, there is substantial variability across users that have rated at least 100 movies. The following mathematical model takes this into account.

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

Because this is a large model, the lm() function is not advisable. The code below will compute the approximation and calculate the RMSE of the new model which accounts for movie effects and user effects.

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

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

model_3_rmse <- RMSE(predicted_ratings, test_set$rating)

RMSE_results <- bind_rows(RMSE_results,
                          tibble(Method="Movie + User Effects Model",
                                 RMSE = model_3_rmse ))
RMSE_results %>% knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103

Method	RMSE
Movie + User Effects Model	0.8658570

Movie + User + Time + Genre Model

Following the logic of the previous two models, we can expand the formula with terms for time and genre effect like so

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

The code below will compute the approximation and calculate the RMSE of the new model which accounts for movie, user, time, and genre effects.

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

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

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

model_4_rmse <- RMSE(predicted_ratings, test_set$rating)

RMSE_results <- bind_rows(RMSE_results,
                          tibble(Method=
                                   "Movie + User + Time + Genre Effects Model",
                                 RMSE = model_4_rmse ))
RMSE_results %>% knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103
Movie + User Effects Model	0.8658570
Movie + User + Time + Genre Effects Model	0.8650181

Regularization

Another method the Netflix challenge winners deployed is called regularization. To learn why I need to use it in my own recommendation system, I will examine the results of the residuals in the movie effect model with the code below.

```
train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  mutate(residual = rating - (mu + b_i)) %>%
  arrange(desc(abs(residual))) %>%
  select(title, residual) %>% slice(1:10) %>% knitr::kable()
```

title	residual
From Justin to Kelly (2003)	4.093220
From Justin to Kelly (2003)	4.093220
Pokémon Heroes (2003)	4.016949
Pokémon Heroes (2003)	4.016949
Shawshank Redemption, The (1994)	-3.955395
Shawshank Redemption, The (1994)	-3.955395
Shawshank Redemption, The (1994)	-3.955395
Shawshank Redemption, The (1994)	-3.955395
Shawshank Redemption, The (1994)	-3.955395
Shawshank Redemption, The (1994)	-3.955395

Why would *Pokémon Heroes* and *From Justin to Kelly* receive a weightier residual than *Shawshank Redemption*? To see what's happening I will dig a little deeper. The code below will show the 10 best and 10 worst movies based on the estimates of the movie effect $\hat{b}_i$.

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

ten_best <- movie_avgs %>%
  left_join(movie_titles, by="movieId") %>%
  arrange(desc(b_i)) %>%
  select(title, b_i) %>%
  slice(1:10,) %>% knitr::kable(name="Best")

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

ten_best
```

title	b_i
Hellhounds on My Trail (1999)	1.487582
Satan's Tango (Sátántangó) (1994)	1.487582
Shadows of Forgotten Ancestors (1964)	1.487582
Money (Argent, L') (1983)	1.487582
Fighting Elegy (Kenka erejii) (1966)	1.487582
Sun Alley (Sonnenallee) (1999)	1.487582
Along Came Jones (1945)	1.487582
Blue Light, The (Das Blaue Licht) (1932)	1.487582
Human Condition III, The (Ningen no joken III) (1961)	1.320915
Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) (1980)	1.237582

```
ten_worst
```

title	b_i
Besotted (2001)	-3.012418
Grief (1993)	-3.012418
Accused (Anklaget) (2005)	-3.012418
Confessions of a Superhero (2007)	-3.012418
War of the Worlds 2: The Next Wave (2008)	-3.012418
SuperBabies: Baby Geniuses 2 (2004)	-2.723956
Hip Hop Witch, Da (2000)	-2.637418
Disaster Movie (2008)	-2.615866
From Justin to Kelly (2003)	-2.605638
Pokémon Heroes (2003)	-2.529367

So, movies that receive the highest and the lowest b's effect are all uncommon movies. Below are the same two tables, but this time with a column that includes the number of times each movie has been rated.

```
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()
```

title	b_i	n
Hellhounds on My Trail (1999)	1.487582	1
Satan's Tango (Sátántangó) (1994)	1.487582	2
Shadows of Forgotten Ancestors (1964)	1.487582	1
Money (Argent, L') (1983)	1.487582	1
Fighting Elegy (Kenka erejii) (1966)	1.487582	1
Sun Alley (Sonnenallee) (1999)	1.487582	1
Along Came Jones (1945)	1.487582	1
Blue Light, The (Das Blaue Licht) (1932)	1.487582	1
Human Condition III, The (Ningen no joken III) (1961)	1.320915	3
Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) (1980)	1.237582	4

```
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()
```

title	b_i	n
Besotted (2001)	-3.012418	2
Grief (1993)	-3.012418	1
Accused (Anklaget) (2005)	-3.012418	1
Confessions of a Superhero (2007)	-3.012418	1
War of the Worlds 2: The Next Wave (2008)	-3.012418	2

title	b_i	n
SuperBabies: Baby Geniuses 2 (2004)	-2.723956	52
Hip Hop Witch, Da (2000)	-2.637418	12
Disaster Movie (2008)	-2.615866	29
From Justin to Kelly (2003)	-2.605638	177
Pokémon Heroes (2003)	-2.529367	118

All of these movies were rated by very few users and in some cases only 1 user. Larger estimates of b_i are more common when fewer users rate the movie. Regularization helps account for this problem by penalizing large estimates of b_i, whether positive or negative, when they come from a small sample size.

To estimate the b's, the task is now to minimize this equation.

$$\frac{1}{N}\sum_{u,i}(y_{u,i}-\mu-b_i)^2+\lambda\sum_i b_i^2$$

This is the residual sum of squares plus a penalty term that gets larger when many of the b's are large. After doing a bit of calculus, it is possible to show that the values of b that minimize this equation can be given by the following formula:

$$\hat{b}_i(\lambda)=\frac{1}{\lambda+n_i}\sum_{u=1}^{n_i}(Y_{u,i}-\hat{\mu})$$

where n_i is the number of ratings for movie i and λ is the penalty term. The effect of this equation is such that when the number of ratings, n_i, is small then the value of the estimate, $\hat{b}_i(\lambda)$ is "shrunken" towards 0, and when n_i is very large then the penalty term, λ, is all but ignored because $n_i+\lambda$ is essentially equal to n_i.

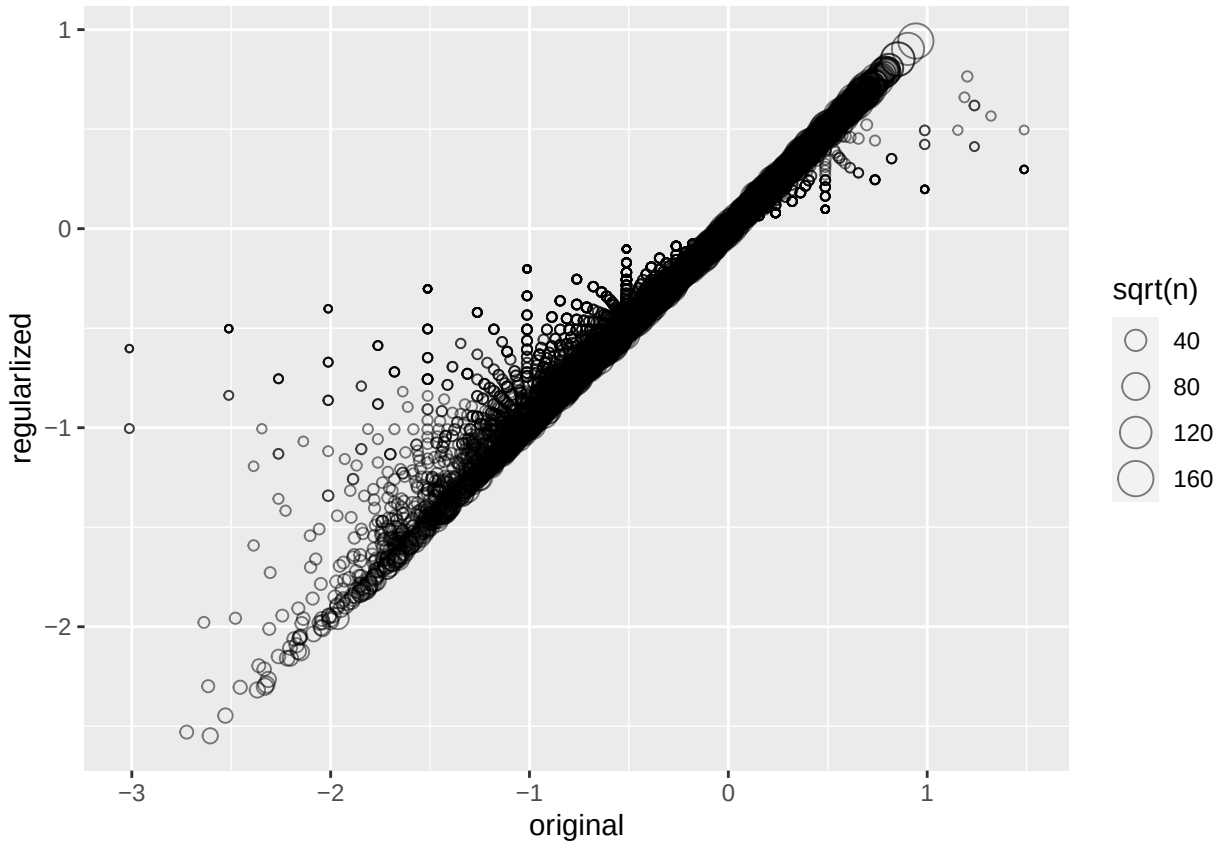
Regularized Movie Model

For the regularized movie model below, I will arbitrarily set the penalty term λ equal to 4. It will become apparent why and how I chose this number as you read the subsequent model.

```
lambda <- 4

mu <- mean(train_set$rating)

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

Now I can make a plot to examine the effect of regularization on movies that were rated only a few times.

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



This shows that indeed the movies with only a few ratings were moved towards zero on both ends. Now I can examine what the new top ten list of movies will be after 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()
```

title	b_i	n
Shawshank Redemption, The (1994)	0.9428276	25188
Godfather, The (1972)	0.9031418	15989
Schindler's List (1993)	0.8515183	20916
Usual Suspects, The (1995)	0.8514632	19497
Rear Window (1954)	0.8081017	7172
Casablanca (1942)	0.8051444	10117
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)	0.8012216	2617
Double Indemnity (1944)	0.7985016	1942
Seven Samurai (Shichinin no samurai) (1954)	0.7975160	4702
Third Man, The (1949)	0.7959245	2675

This makes much more sense, as does the ten worst movies shown below.

```
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()
```

title	b_i	n
From Justin to Kelly (2003)	-2.548055	177
SuperBabies: Baby Geniuses 2 (2004)	-2.529388	52
Pokémon Heroes (2003)	-2.446437	118
Pokemon 4 Ever (a.k.a. Pokémon 4: The Movie) (2002)	-2.317555	181
Carnosaur 3: Primal Species (1996)	-2.303962	61
Glitter (2001)	-2.300905	313
Disaster Movie (2008)	-2.298792	29
Gigli (2003)	-2.290793	285
Barney's Great Adventure (1998)	-2.264788	186
Yu-Gi-Oh! (2004)	-2.213072	73

Now I can calculate the RMSE of the regularized movie effect model.

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

model_3_rmse <- RMSE(predicted_ratings, test_set$rating)

RMSE_results <- bind_rows(RMSE_results,
                          tibble(Method="Regularized Movie Effect Model",
                                 RMSE = model_3_rmse ))
RMSE_results %>% knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103
Movie + User Effects Model	0.8658570
Movie + User + Time + Genre Effects Model	0.8650181
Regularized Movie Effect Model	0.9434773

The Regularized Movie Effect model is an improvement over the least squares Movie Effect Model. Now that the brief exploration and proof of improvement using regularization is behind us, we can move forward by expanding our model with all the possible terms.

Regularized Movie + User + Time + Genre Model

The previous model can be expanded to include regularization terms for the user, time, and genre with the following formula:

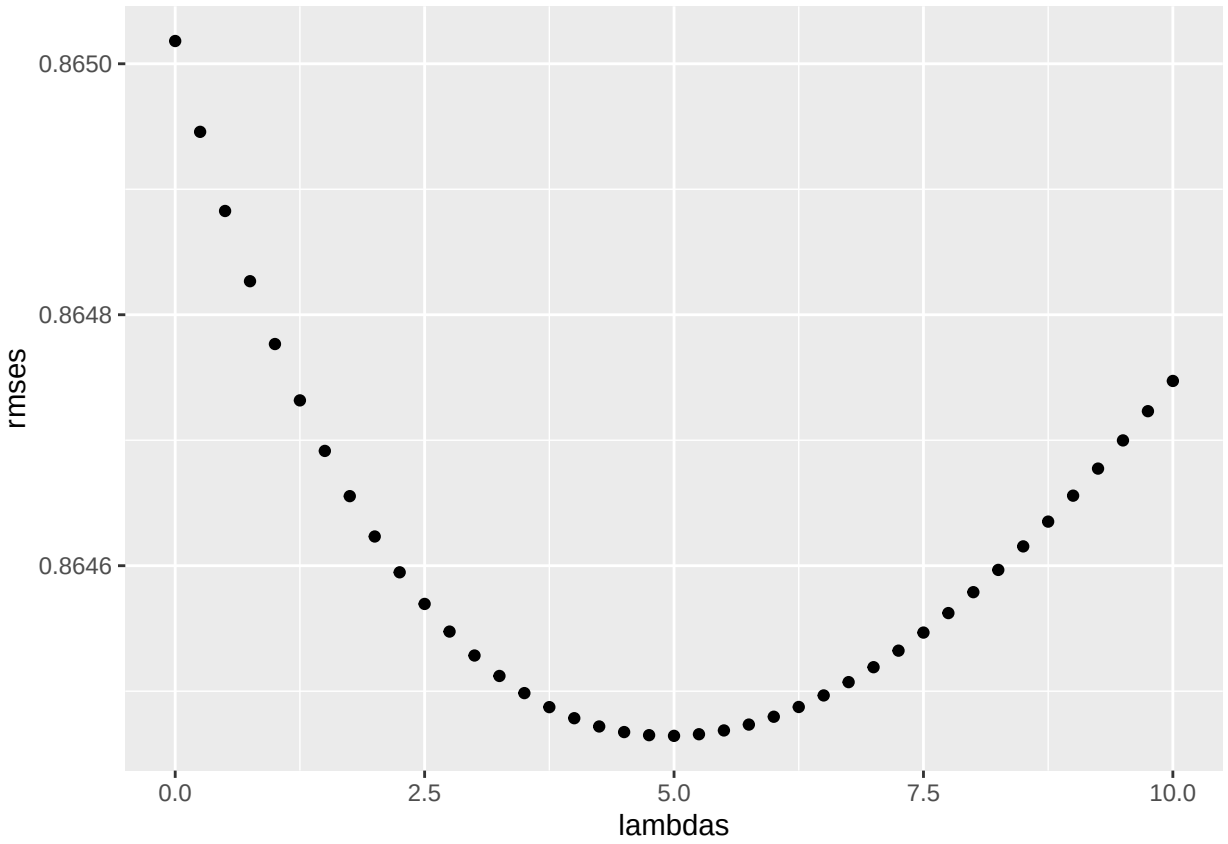
$$\frac{1}{N}\sum_{u,i}(y_{u,i}-\mu-b_i-b_u-b_t-b_g)^2+\lambda(\sum_i b_i^2+\sum_u b_u^2+\sum_t b_t^2+\sum_g b_g^2)$$

Now I would like to note that because the penalty term, λ, is a tuning parameter, I can use cross-validation to select the value of λ that minimizes RMSE with the code below.

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

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

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

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

Below is a plot of the rmse based on the value of λ over the range 0 to 10 and the effect it has on reducing the root mean squared error.

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



Now I can select the best value of λ with the code below.

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

```
## [1] 5
```

Time to show the results of the regularized movie + user + time + genre effect model.

```
RMSE_results <- bind_rows(RMSE_results,
                          tibble(Method="Regularized Movie + User + Time + Genre Model",
                                 RMSE = min(rmses)))
RMSE_results %>% knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103
Movie + User Effects Model	0.8658570
Movie + User + Time + Genre Effects Model	0.8650181
Regularized Movie Effect Model	0.9434773
Regularized Movie + User + Time + Genre Model	0.8644644

The Regularized Movie + User + Time + Genre Effect Model has brought me to an RMSE of below the class goal for an A!

Why Matrix Factorization

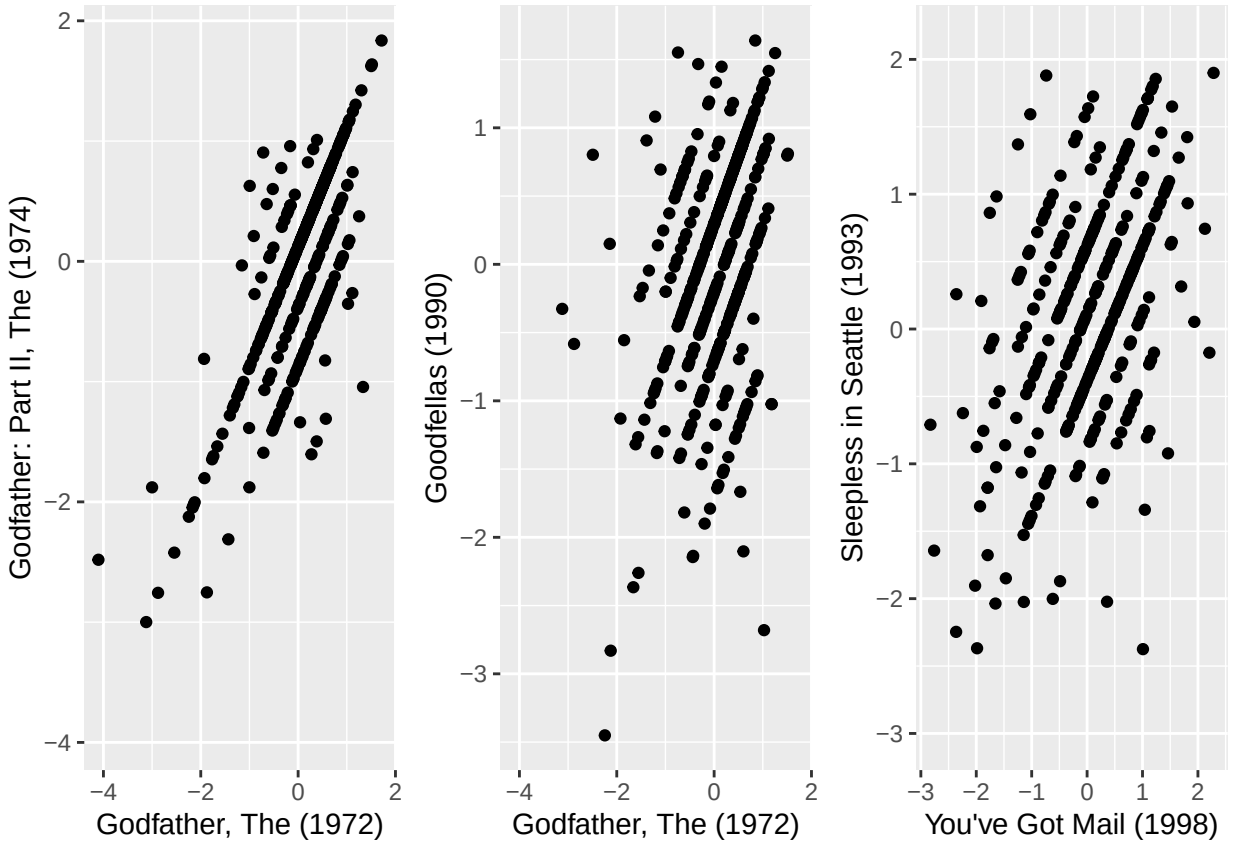
The previous model leaves out an important source of variation related to the fact that groups of movies have similar rating patterns and groups of users have similar rating patterns as well. We will discover these patterns by studying the residuals obtained after fitting my model. Once I convert the data into a matrix, I can see these residuals:

$$r_{u,i} = y_{u,i} - \hat{b}_i - \hat{b}_u$$

, where $y_{u,i}$ is the entry in row u for user and column i for movie. For illustration purposes we will only consider a small subset of movies with many ratings and users that have rated many movies.

```
# Sample the train set for movies that have received 600 or more
# ratings and users who users who have rated 600 or more movies

train_small <- train_set %>%
  group_by(movieId) %>%
  filter(n() >= 600) %>%
  ungroup() %>%
  group_by(userId) %>%
  filter(n() >= 600) %>% ungroup()

# Convert to matrix
y <- train_small %>%
  select(userId, movieId, rating) %>%
  spread(movieId, rating) %>%
  as.matrix()

# keep rownames
rownames(y)<- y[,1]

y <- y[,-1]

# select distinct titles
movie_titles <- train_set %>%
  select(movieId, title) %>%
  distinct()

# add colnames
colnames(y) <- with(movie_titles, title[match(colnames(y), movieId)])

# calc vectorization
y <- sweep(y, 2, colMeans(y, na.rm=TRUE))
y <- sweep(y, 1, rowMeans(y, na.rm=TRUE))

# display correlation between times of movies
m_1 <- "Godfather, The (1972)"
m_2 <- "Godfather: Part II, The (1974)"
p1 <- qplot(y[ ,m_1], y[,m_2], xlab = m_1, ylab = m_2)

m_1 <- "Godfather, The (1972)"
m_3 <- "Goodfellas (1990)"
p2 <- qplot(y[ ,m_1], y[,m_3], xlab = m_1, ylab = m_3)

m_4 <- "You've Got Mail (1998)"
m_5 <- "Sleepless in Seattle (1993)"
p3 <- qplot(y[ ,m_4], y[,m_5], xlab = m_4, ylab = m_5)
```

```
gridExtra::grid.arrange(p1, p2 ,p3, ncol = 3)
```



This plot says that users that liked *The Godfather* more than what the model expects them to, based on the movie and user effects, also like *The Godfather II* more than expected. A similar relationship is seen when comparing *The Godfather* and *Goodfellas*. Although not as strong, there is still correlation. There are also correlations between *You've Got Mail* and *Sleepless in Seattle*.

By looking at the pairwise correlation for these 5 movies, I can see a pattern.

```
cor(y[, c(m_1, m_2, m_3, m_4, m_5)], use="pairwise.complete") %>%
  knitr::kable()
```

	Godfather, The (1972)	Godfather: Part II, The (1974)	Goodfellas (1990)	You've Got Mail (1998)	Sleepless in Seattle (1993)
Godfather, The (1972)	1.0000000	0.7788633	0.4004764	-0.0514765	-0.0693291
Godfather: Part II, The (1974)	0.7788633	1.0000000	0.4506023	0.0156077	-0.0810283
Goodfellas (1990)	0.4004764	0.4506023	1.0000000	-0.0946919	-0.0798672
You've Got Mail (1998)	-0.0514765	0.0156077	-0.0946919	1.0000000	0.4556652
Sleepless in Seattle (1993)	-0.0693291	-0.0810283	-0.0798672	0.4556652	1.0000000

The correlations across genres informs us that the data has some structure that up to now hasn't been accounted for in the models.

Using matrix factorization here allows the model to be adjusted to explain more of the observed variance. Matrix r_u, i will be factorized it into two vectors p_u and q_i. This is defined as $r_{u,i} \approx p_u q_i$. Then a new model can be written like so.

$$Y_{u,i} = \mu + b_i + b_u + p_u q_i + \epsilon_{i,j}$$

Now the structure in our movie data seems to be much more complicated than mobster versus romantic comedies. In order to discover all the structure, I will use principal component analysis or singular value decomposition. Singular value decomposition can be thought of as an algorithm that finds the vectors p and q which can be used to create a matrix of residuals r with m rows and n columns in the following way:

$$r_{u,i} = p_{u,1} q_{1,i} + p_{u,2} q_{2,i} + ... + p_{u,m} q_{m,i}$$

The variability of each term is decreasing, and the p's are uncorrelated. The algorithm also computes this variability so that we can know how much of the matrices total variability is explained as we add new terms. This may permit us to see that, with just a few terms, we can explain most of the variability.

Let's see an example with the movie data. To compute the decomposition, I will make the residuals with NA's equal to 0:

```
y[is.na(y)] <- 0
y <- sweep(y, 1, rowMeans(y))
pca <- prcomp(y)
```

The q vectors are called the principal components and they are stored in this matrix:

```
dim(pca$rotation)
```

```
## [1] 2452 1006
```

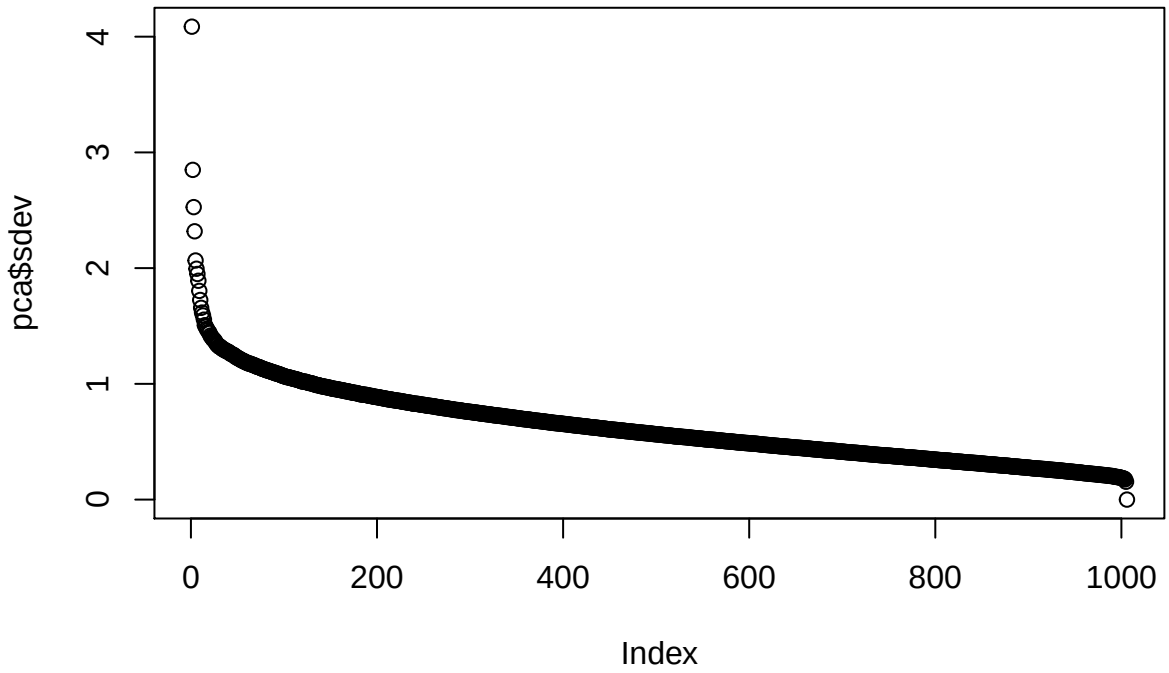
While the p, or the user effects, are stored here:

```
dim(pca$x)
```

```
## [1] 1006 1006
```

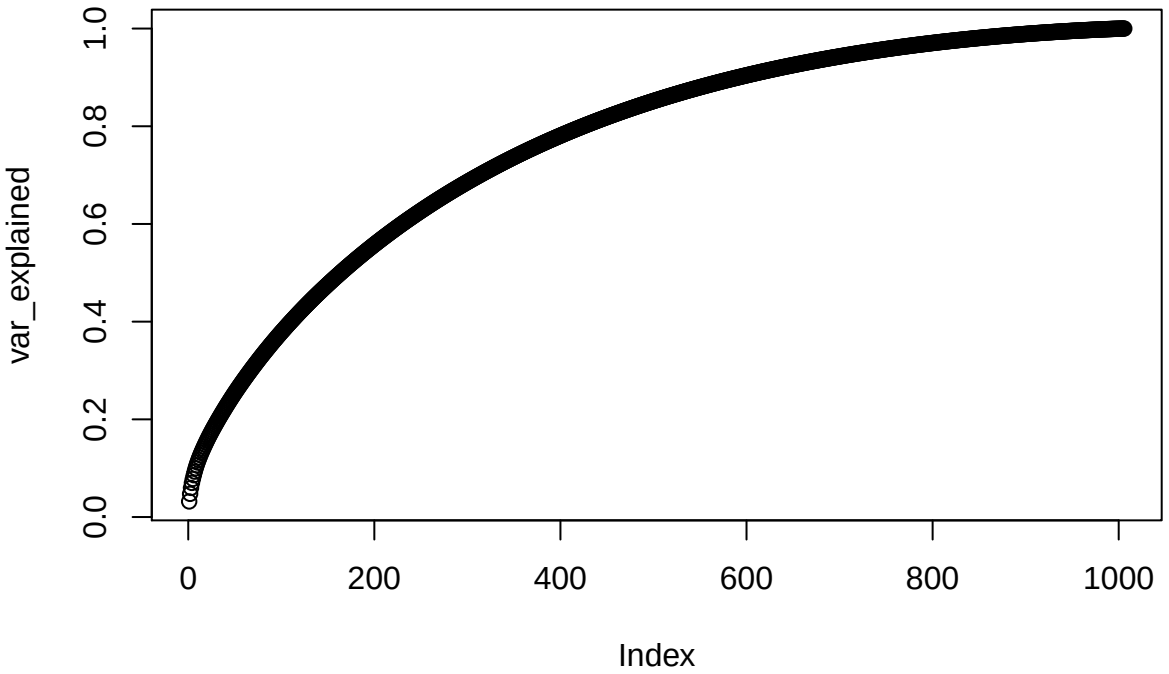
The PCA function returns a component with the variability of each of the principal components and it can be accessed and plotted with the following code.

```
plot(pca$sdev)
```



We can also see that with less than 200 of these variables the majority of the variance is explained.

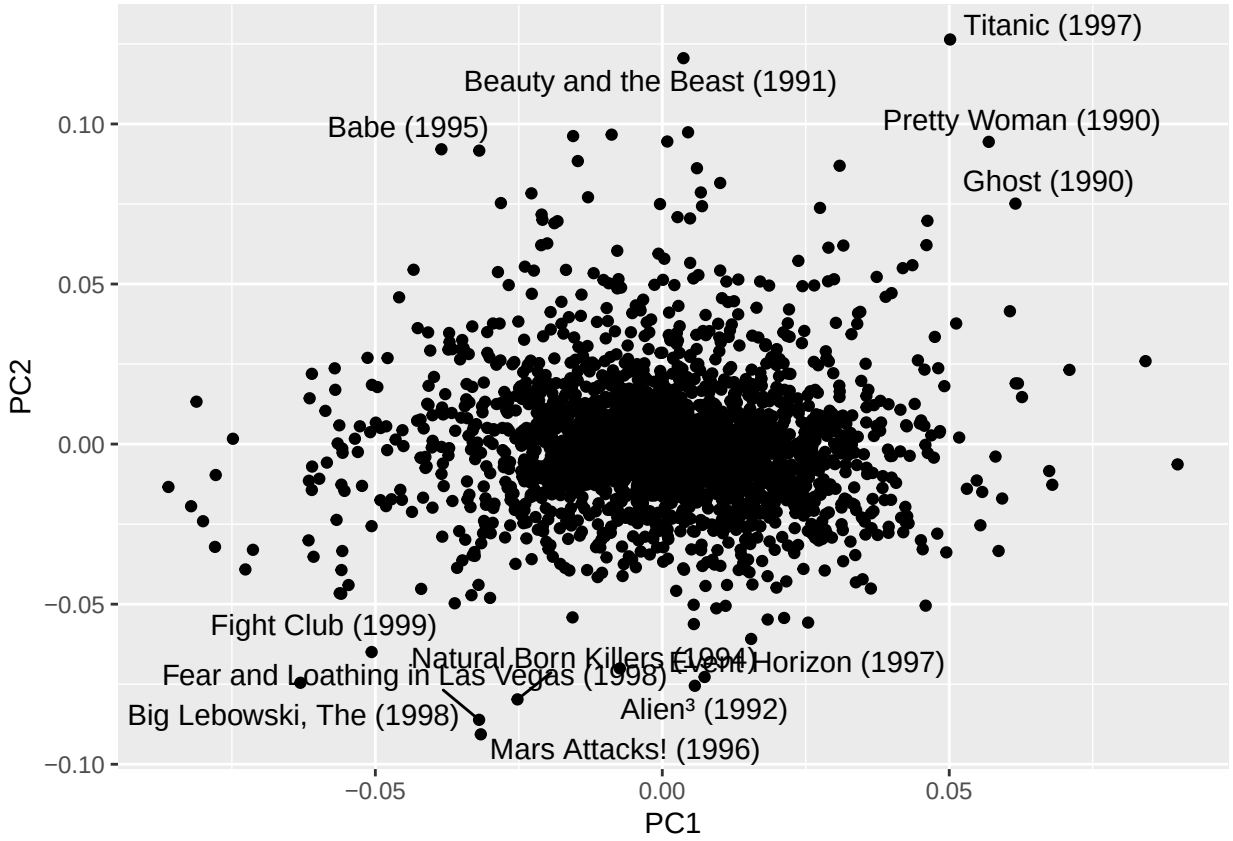
```
var_explained <- cumsum(pca$sdev^2/sum(pca$sdev^2))
plot(var_explained)
```



To see that the principal components are actually capturing something important about the data, I can make a plot of them. For example, the first two principal components are plotted below. The points represent individual movies and I have provided the titles for few in order to provide us with a quick overview.

```
pcs <- data.frame(pca$rotation, name = colnames(y))
pcs %>%
  ggplot(aes(PC1, PC2)) +
```

```
  geom_point() +
  geom_text_repel(aes(PC1, PC2, label=name),
                  data = filter(pcs,
                                PC1 < -.01 | PC1 > .01 | PC2 < -0.075 | PC2 > .01))
```



Just by looking at these, we see some meaningful patterns. The first principal component shows the difference between critically acclaimed movies on one side and blockbusters on the other.

These are one extreme of the principal component.

```
pcs %>% select(name, PC1) %>% arrange(PC1) %>% slice(1:10) %>% tibble() %>%
  knitr::kable()
```

name	PC1
2001: A Space Odyssey (1968)	-0.0860530
Taxi Driver (1976)	-0.0821018
Fargo (1996)	-0.0811645
Rushmore (1998)	-0.0800214
Royal Tenenbaums, The (2001)	-0.0779362
Being John Malkovich (1999)	-0.0778082
Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964)	-0.0748130
Clockwork Orange, A (1971)	-0.0726414
Pulp Fiction (1994)	-0.0713262
Big Lebowski, The (1998)	-0.0630542

And these are the other extreme of the first principal component.

```
pcs %>% select(name, PC1) %>% arrange(desc(PC1)) %>% slice(1:10) %>% tibble() %>%
  knitr::kable()
```

name	PC1
Armageddon (1998)	0.0897940
Independence Day (a.k.a. ID4) (1996)	0.0842063
Twister (1996)	0.0709471
Patch Adams (1998)	0.0679575
Batman & Robin (1997)	0.0674209
Pearl Harbor (2001)	0.0626811
Patriot, The (2000)	0.0619665
Star Wars: Episode I - The Phantom Menace (1999)	0.0615800
Ghost (1990)	0.0615392
Top Gun (1986)	0.0605724

If we look at the extremes of PC2 , we see more structure in the data.

```
pcs %>% select(name, PC2) %>% arrange(PC2) %>% slice(1:10) %>% tibble() %>%
  knitr::kable()
```

name	PC2
Mars Attacks! (1996)	-0.0906492
Fear and Loathing in Las Vegas (1998)	-0.0861183
Natural Born Killers (1994)	-0.0797422
Alien³ (1992)	-0.0754929
Big Lebowski, The (1998)	-0.0745489
Starship Troopers (1997)	-0.0727399
From Dusk Till Dawn (1996)	-0.0700621
Fight Club (1999)	-0.0649497
Event Horizon (1997)	-0.0608562
Dune (1984)	-0.0562172

```
pcs %>% select(name, PC2) %>% arrange(desc(PC2)) %>% slice(1:10) %>% tibble() %>% knitr::kable()
```

name	PC2
Titanic (1997)	0.1263799
Beauty and the Beast (1991)	0.1205415
Little Mermaid, The (1989)	0.0973924
When Harry Met Sally... (1989)	0.0966556
E.T. the Extra-Terrestrial (1982)	0.0962091
Sound of Music, The (1965)	0.0945134
Pretty Woman (1990)	0.0944073
Babe (1995)	0.0920892
Wizard of Oz, The (1939)	0.0916665
Shakespeare in Love (1998)	0.0884054

Remember earlier when I set the residuals that were NA equal to 0? This is the point that true SVD for a recommendation system fails to be the best approach. I need an algorithm that accounts for the NA's and not merely ignores them, this is where the Matrix Factorization becomes necessary.

Matrix Factorization Model - Recommenderlab

The idea is to approximate the whole rating matrix $R_{m\times n}$ by the product of two matrices of lower dimensions, $P_{k\times m}$ and $Q_{k\times n}$, such that $R \approx PQ$. Let p_u be the uth column of P, and q_i be the ith column of Q, then the rating given by user u on item i would be predicted as $p_u q_i$.

A typical solution for P and Q is given as follows:

$$\min_{P,Q} \sum_{(u,i)\in R} \left[f(p_u, q_i; r_{u,i}) + \mu_P ||p_u||_1 + \mu_Q ||q_i||_1 + \frac{\lambda_P}{2}||p_u||_2^2 + \frac{\lambda_Q}{2}||q_i||_2^2 \right]$$

where (u, i) are locations of observed entries in R, $r_{u,i}$ is the observed rating, $f(p_u, q_i; r_{u,i})$ is the loss function, and $\mu_P||p_u||_1$, $\mu_Q||q_i||_1$, $\frac{\lambda_P}{2}||p_u||_2^2$, and $\frac{\lambda_Q}{2}||q_i||_2^2$ are penalty parameters to prevent over-fitting.

The process of solving the matrices P and Q is called model training, and the selection of penalty parameters is parameter tuning. After obtaining P and Q, you can then do the prediction of $\hat{R}_{u,i} = p_u q_i$.

Our instructor mentioned in the video that we should look into recommenderlab for matrix factorization. I have written the code below for the recommenderlab version of the LIBMF library (library for matrix factorization). However, it takes quite some time to run because recommenderlab is by default extremely memory based for computations.

```
##############################################################################
#                                                                            #
#                                  WARNING                                   #
#                                                                            #
##############################################################################
# THE FOLLOWING CODE TAKES 30 MINUTES TO RUN AND REQUIRES A MINIMUM of 32GB of
# RAM

# I USED CLOUD COMPUTING ON AWS TO KNIT THE PDF REPORT BECAUSE MY MACHINE
# CANNOT HANDLE THE MEMORY LOAD

########################         WARNING          ##############################
# IF YOUR SYSTEM DOESN'T HAVE AT LEAST 32GB OF RAM DO NOT RUN THIS CHUNK LOCALLY


# First step is to free unused memory space
gc()
```

```
##             used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells   3199186  170.9   10039763  536.2  22895559 1222.8
## Vcells 275874175 2104.8  690884223 5271.1 690884223 5271.1
```

```
# A quick read of the help files states that I must have matrices in the
# required realRatingMatrix type. So first up I will have to coerce the
# movielens data to a matrix
y <- movielens %>%
  select(userId, movieId, rating) %>%
  spread(movieId, rating) %>%
  as.matrix()

# setting names properly
rownames(y)<- y[,1]

y <- y[,-1]
```

```
# Finally coerce to the recommenderlab specific "realRatingMatrix"
movielenseM <- as(y, "realRatingMatrix")

# split the data by a 80/20 split which mimics the second split of the edx set
# to train and test sets
e <- evaluationScheme(movielenseM, method="split",
                      train=0.8, k=1, given=-10)



# create a libmf recommender using training data
r <- Recommender(getData(e, "train"), "LIBMF")


# create predictions for the test data using known ratings (the given param
# from the evaluation scheme)
p <- predict(r, getData(e, "known"), type="ratings")
```

```
## iter      tr_rmse          obj
##    0       0.9423   9.5706e+06
##    1       0.8575   8.1049e+06
##    2       0.8257   7.5733e+06
##    3       0.8166   7.4215e+06
##    4       0.8119   7.3442e+06
##    5       0.8082   7.2840e+06
##    6       0.8049   7.2327e+06
##    7       0.8015   7.1811e+06
##    8       0.7981   7.1301e+06
##    9       0.7947   7.0793e+06
##   10       0.7913   7.0310e+06
##   11       0.7882   6.9875e+06
##   12       0.7852   6.9455e+06
##   13       0.7824   6.9061e+06
##   14       0.7797   6.8687e+06
##   15       0.7773   6.8351e+06
##   16       0.7749   6.8026e+06
##   17       0.7727   6.7723e+06
##   18       0.7706   6.7438e+06
##   19       0.7686   6.7167e+06
```

```
# calculate the average RMSE when User specific

acc <- calcPredictionAccuracy(p, getData(e, "unknown"), byUser=TRUE)

acc <- mean(acc[,1])

# Append to the RMSE results tibble
RMSE_results <- bind_rows(RMSE_results,
                    tibble(Method = "Matrix Factorization - Recommenderlab",
                           RMSE = acc))
# Show the RMSE improvement
RMSE_results %>%
  knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103
Movie + User Effects Model	0.8658570
Movie + User + Time + Genre Effects Model	0.8650181
Regularized Movie Effect Model	0.9434773
Regularized Movie + User + Time + Genre Model	0.8644644
Matrix Factorization - Recommenderlab	0.8106222

```
# use gc() one last time to free unused memory
gc()
```

```
##             used   (Mb) gc trigger    (Mb)   max used    (Mb)
## Ncells    3301567  176.4   10039763   536.2   22895559  1222.8
## Vcells 1283255267 9790.5 2900710724 22130.7 3625888404 27663.4
############################## END WARNING  #####################################
```

This is a great result! However, the time and machine requirements are not feasible for my peers to review on their own machines. This problem is a common one with large data sets eating too much ram for computation. The solution is quite simple however. That is to to compute the calculations in parallel on most modern CPU's. Indeed, this problem is so common that it has a name and a wikipedia page devoted to it. It is called "Embarrassingly Parallel"

Matrix Factorization Model - Recosystem

My search for a better implementation of the LIBMF library lead me to Recosystem. Recosystem is a completely parallel wrapper for the LIBMF library which saves on the usage of RAM. The most important benefit to me personally is that it saves on time!

One important note: Recosystem's Makevars can be adjusted to take advantage of modern CPUs that have SSE3 and AVX. You just need a C++ compiler that supports the C++11 standard. Then you can edit src/Makevars (src/Makevars.win for Windows system) according to the following guidelines:

1) If your CPU supports SSE3, add the following

```
PKG_CPPFLAGS += -DUSESSE
PKG_CXXFLAGS += -msse3
```

2) If SSE3 and also AVX is supported, add the following

```
PKG_CPPFLAGS += -DUSEAVX
PKG_CXXFLAGS += -mavx
```

After editing the Makevars file, run R CMD INSTALL recosystem to install recosystem with AVX and SSE3.

This is the process I used on my desktop. However, the default Makevars provides generic options that should apply to most CPUs. Recosystem is a wrapper for the Library for Parallel Matrix Factorization, which is known as LIBMF. You can read more about LIBMF by following the documentation links below.

https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/libmf_journal.pdf

https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/mf_adaptive_pakdd.pdf

https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/libmf_open_source.pdf

OK, now that the documentation links are over with for the LIBMF library. I would like to site the help pages authored by Yixuan Qiu. This is where I found information about how to transform the data sets into the appropriate object type.

You can read them for yourself the standard way by running the code below in R.

```
??recosystem
```

You can also follow the link below to Yixuan's blog post which includes the same helpful information.

https://statr.me/2016/07/recommender-system-using-parallel-matrix-factorization/

```
set.seed(1985, sample.kind = "Rounding") # This is a randomized algorithm

# This takes four minutes to run and should not be a concern for anyone on
# a modern laptop.

# Convert the train and test sets into recosystem input format
train_data <- with(train_set, data_memory(user_index = userId,
                                          item_index = movieId,
                                          rating = rating))
test_data <- with(test_set, data_memory(user_index = userId,
                                        item_index = movieId,
                                        rating = rating))


# Create the model object
r <- recosystem::Reco()

# Select the best tuning parameters using cross-validation
opts <- r$tune(train_data, opts = list(dim = c(10, 20, 30),
                                       costp_l2 = c(0.01, 0.1),
                                       costq_l2 = c(0.01, 0.1),
                                       costp_l1 = 0,
                                       costq_l1 = 0,
                                       lrate    = c(0.01, 0.1),
                                       nthread  = 8,
                                       niter    = 10,
                                       verbose  = FALSE))
# Train the algorithm
r$train(train_data, opts = c(opts$min,
                             nthread = 8,
                             niter = 100,
                             verbose = FALSE))

# Calculate the predicted values
reco_pred <- r$predict(test_data, out_memory())

RMSE_results <- bind_rows(RMSE_results,
                    tibble(Method = "Parallel Matrix Factorization - Recosystem",
                           RMSE = RMSE(test_set$rating, reco_pred)))
# Show the RMSE improvement
RMSE_results %>%
  knitr::kable()
```

Method	RMSE
The Average	1.0600069
Movie Effect Model	0.9435103
Movie + User Effects Model	0.8658570
Movie + User + Time + Genre Effects Model	0.8650181

Method	RMSE
Regularized Movie Effect Model	0.9434773
Regularized Movie + User + Time + Genre Model	0.8644644
Matrix Factorization - Recommenderlab	0.8106222
Parallel Matrix Factorization - Recosystem	0.7877075

Matrix Factorization has taken me to my goal of an RMSE below 0.80! Now it is time to run the final validations.

Final Validations

I will include three of the models for final validations, all of which surpass the minimum RMSE of .86490 to receive an A on the project. The first is the regularized movie, user, time, and genre model, the second will be the recommenderlab Matrix Factorization model, and the third will be the recosystem matrix factorization model.

Final Validation - Regularized Movie + User + Time + Genre Model

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

rmses <- sapply(lambdas, function(l){
  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_t <- edx %>%
    left_join(b_i, by='movieId') %>%
    left_join(b_u, by='userId') %>%
    group_by(years_since_release) %>%
    summarize(b_t = sum(rating - b_i - b_u - mu)/(n()+l))

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

  predicted_ratings <- validation %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(b_t, by = 'years_since_release') %>%
    left_join(b_g, by = 'genres') %>%
    mutate(pred = mu + b_i + b_u + b_t + b_g) %>%
```

```
    .$pred

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

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

```
## [1] 5.25
```

```
Final_Validation_RMSE_results_table <- tibble(Method =
"Regularized Movie + User + Time + Genre Model", RMSE = min(rmses))

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

Method	RMSE
Regularized Movie + User + Time + Genre Model	0.8639782

Final Validation - Recommenderlab - Matrix Factorization Model

```
##############################################################################
#                                                                            #
#                                  WARNING                                   #
#                                                                            #
##############################################################################
# THE FOLLOWING CODE TAKES 30 MINUTES TO RUN AND REQUIRES A MINIMUM of 32GB of
# RAM

# I USED CLOUD COMPUTING ON AWS TO KNIT THE PDF REPORT BECAUSE MY MACHINE
# CANNOT HANDLE THE MEMORY LOAD

#########################         WARNING          ##############################
# IF YOUR SYSTEM DOESN'T HAVE AT LEAST 32GB OF RAM DO NOT RUN THIS CHUNK LOCALLY


# Free unused memory space
gc()
```

```
##             used   (Mb) gc trigger    (Mb)   max used    (Mb)
## Ncells    3302336  176.4   10039763   536.2   22895559  1222.8
## Vcells 1276531270 9739.2 2900710724 22130.7 3625888404 27663.4
```

```
# I have already coerced the movielens data into the required format and it is
# stored as movielenseM which is a realRatingMatrix so there is no need to
# repeat those steps. However, I do need to repeat the split. The recommenderlab
# way is evaluationScheme and the code below shows a 90/10 split which
# is the same proportion as the original edx / validation split, which is also
# 90/10.
e <- evaluationScheme(movielenseM, method="split",
                      train=0.9, k=1, given=-10)


# create a libmf recommender using training data
r <- Recommender(getData(e, "train"), "LIBMF")
```

```
# create predictions for the test data using known ratings (the given param
# from the evaluation scheme)
p <- predict(r, getData(e, "known"), type="ratings")
```

```
## iter      tr_rmse          obj
##    0       0.9423   9.6369e+06
##    1       0.8556   8.1242e+06
##    2       0.8278   7.6585e+06
##    3       0.8194   7.5169e+06
##    4       0.8152   7.4448e+06
##    5       0.8122   7.3947e+06
##    6       0.8097   7.3537e+06
##    7       0.8072   7.3138e+06
##    8       0.8046   7.2738e+06
##    9       0.8020   7.2344e+06
##   10       0.7991   7.1911e+06
##   11       0.7961   7.1480e+06
##   12       0.7930   7.1034e+06
##   13       0.7899   7.0604e+06
##   14       0.7869   7.0174e+06
##   15       0.7840   6.9779e+06
##   16       0.7813   6.9399e+06
##   17       0.7787   6.9036e+06
##   18       0.7763   6.8705e+06
##   19       0.7740   6.8392e+06
```

```
# calculate the average RMSE when User specific

acc <- calcPredictionAccuracy(p, getData(e, "unknown"), byUser=TRUE)

acc <- mean(acc[,1])

# Append to the RMSE results tibble
Final_Validation_RMSE_results_table <- bind_rows(Final_Validation_RMSE_results_table,
                                                 tibble(
                                                   Method =
                                                     "Final Validation - Recommenderlab - Matrix Factori
                                 RMSE = acc))

# Show the Results
Final_Validation_RMSE_results_table %>% knitr::kable()
```

Method	RMSE
Regularized Movie + User + Time + Genre Model	0.8639782
Final Validation - Recommenderlab - Matrix Factorization	0.8161826

```
# use gc() one last time to free unused memory
gc()
```

```
##             used   (Mb) gc trigger    (Mb)   max used    (Mb)
## Ncells    3303971  176.5   10039763   536.2   22895559  1222.8
## Vcells 1179512028 8999.0 2900710724 22130.7 3625888404 27663.4
```

```
######################### END WARNING  ########################################
```

Final Validation - Recosystem - Matrix Factorization Model

Below is the matrix factorization method via Recosystem performed on the final validation set.

```
# Final Validation

set.seed(1986, sample.kind="Rounding")

train_data <- with(edx, data_memory(user_index = userId,       #EDX
                                    item_index = movieId,
                                    rating = rating))
test_data <- with(validation, data_memory(user_index = userId, #VALIDATION
                                          item_index = movieId,
                                          rating = rating))

r <- recosystem::Reco()

# Select the best tuning parameters using cross-validation
opts <- r$tune(train_data, opts = list(dim = c(10, 20, 30),
                                       costp_l2 = c(0.01, 0.1),
                                       costq_l2 = c(0.01, 0.1),
                                       costp_l1 = 0,
                                       costq_l1 = 0,
                                       lrate    = c(0.01, 0.1),
                                       nthread  = 8,
                                       niter    = 10,
                                       verbose  = FALSE))
# Train the algorithm
r$train(train_data, opts = c(opts$min,
                             nthread = 8,
                             niter = 100,
                             verbose = FALSE))

# Calculate the predicted values
reco_final_pred <- r$predict(test_data, out_memory())

Final_Validation_RMSE_results_table <- bind_rows(Final_Validation_RMSE_results_table,
                          tibble(Method = "Final Validation - Recosystem - True Parallel Matrix Factori
                             RMSE = RMSE(validation$rating, reco_final_pred)))
Final_Validation_RMSE_results_table %>% knitr::kable()
```

Method	RMSE
Regularized Movie + User + Time + Genre Model	0.8639782
Final Validation - Recommenderlab - Matrix Factorization	0.8161826
Final Validation - Recosystem - True Parallel Matrix Factorization	0.7823174

Summary

While this is just a glimpse into the world of recommendation systems using machine learning, it has been an informative one! I have certainly enjoyed working through the different models and examining the results.

While no recommendation system is perfect, I am pleased to have two final validations that surpass the class goal of an RMSE below .86490, and happy to have met my personal goal of an RMSE below that of 0.80.

Obviously, the results from Recosystem's implementation were the best in terms of RMSE results and capacity for the data set. In the future, I plan on learning more about parallelization and the benefits it brings to working with "big" data. While this data set is large enough to present challenges, it is important to remember that it is still only a subset of the movielense data. It is also worth noting that as time progresses, the size of data sets like this can increase dramatically from the addition of new users and movies.