HarvardX - Data Science Professional Certificate - Capstone - Project MovieLens

Werner Alencar Advincula Dassuncao

2021-03-24

Contents

ABSTRACT

This project is the final assignment for the completion of the Professional Certificate in Data Science from Harvard via the EDX online learning platform. Inspired by the *Netflix Prize* where teams competed to achieve the best predicting performance for the movie streaming service Netflix. The goal of the project is to create a movie recommendation system. The *MovieLens* data was made available by Grouplens research lab.

The idea is to predict what movies a particular user will like based on ratings data provided by users regarding movies. Several techniques were employed and the best performing one was Parallel Matrix Factorization from Recosystem.

INTRODUCTION

This project is inspired by the *Netflix Prize* - "an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings without any other information about the users or films" ['https://en.wikipedia.org/wiki/Netflix_Prize']. The video-streaming service Netflix provided the competitors with a training data set of 100,480,507 ratings, provided by 480,189 users regarding 17,770 movies. The competition began on October 2, 2006 and aimed to improve Netflix's *Cinematch* own algorithm by 10%. In summary, the *training* data set was used to train algorithms and the final model for the team's algorithm was used to make predictions on the *qualifying* data set. The quality of the predictions were scored against the true grades in terms of root mean squared error (RMSE).

For the scope of this project, we will gather, explore, visualize, analyze and make predictions over the data from the *MovieLens* data set with 10,000,000 ratings provided by GroupLens, a research lab in the Department of Computer Science and Engineering at the University of Minnesota.

Recommendations can be done using the users own past ratings, but also using a technique *collaborative filtering* can filter out movies that the user might like based on ratings from similar users.

LOAD THE DATA

Setup R environment, load and install packages

During this analysis we will use the following libraries. The code below checks if these are installed, if not, installs the necessary packages.

```
if(!require(tidyverse)) install.packages('tidyverse', repos = 'http://cran.us.r-project.org')
if(!require(recosystem)) install.packages('recosystem', repos='http://cran.us.r-project.org')
if(!require(caret)) install.packages('caret', repos = 'http://cran.us.r-project.org')
if(!require(data.table)) install.packages('data.table', repos = 'http://cran.us.r-project.org')
if(!require(dplyr)) install.packages('dplyr', repos = 'http://cran.us.r-project.org')
if(!require(knitr)) install.packages('knitr', repos = 'http://cran.us.r-project.org')
if(!require(ggplot2)) install.packages('ggplot2', repos = 'http://cran.us.r-project.org')
if(!require(anytime)) install.packages('anytime', repos = 'http://cran.us.r-project.org')
if(!require(recommenderlab)) install.packages('recommenderlab', repos = 'http://cran.us.r-project.org')
if(!require(lsa)) install.packages('lsa', repos = 'http://cran.us.r-project.org')
if(!require(irlba)) install.packages('irlba', repos = 'http://cran.us.r-project.org')
if(!require(tinytex)) install.packages('tinytex', repos = 'http://cran.us.r-project.org')
if(!require(kableExtra)) install.packages('kableExtra', repos = 'http://cran.us.r-project.org')
library(tinytex)
library(tidyverse)
```

```r
library(recosystem)
library(caret)
library(data.table)
library(knitr)
library(dplyr)
library(ggplot2)
library(anytime)
library(recommenderlab)
library(lsa)
library(irlba)
library(kableExtra)
```

Download source files

The MovieLens dataset utilized is available in the following URL: http://files.grouplens.org/datasets/movielens/ml-10m.zip

```r
# Download the source data
dl <- tempfile()
download.file('http://files.grouplens.org/datasets/movielens/ml-10m.zip', dl)
```

After downloading the zip file, we notice there are two data files "movies.dat" and "ratings.dat". The movies file has columns divided by "::" and each line will be split 3 times: movieId, title and genres.

```r
# MovieID::Title::Genres
movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")

# Transform the movies character vector into a data.frame object: # R 4.0 or later:
movies <- as.data.frame(movies) %>%
  mutate(movieId = as.numeric(movieId),
         title = as.character(title),
         genres = as.character(genres))

# Show the first 5 observations:
head(movies, 5) %>% kable()
```

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy

The ratings file will be loaded by nesting the functions gsub() and fread(), substituting the string "::" by "" and adding names the 4 columns respectively 'userId', 'movieId', 'rating', 'timestamp'. Now we have a tab separated object with determined columns names.

```r
# Read and parse ratings.dat, add column names
# UserID::MovieID::Rating::Timestamp
ratings <- fread(text = gsub('::', '\t', readLines(unzip(dl, 'ml-10M100K/ratings.dat'))),
                 col.names = c('userId', 'movieId', 'rating', 'timestamp'))
```

```r
# Show the first 5 observations:
head(ratings,5) %>% kable()
```

userId	movieId	rating	timestamp
1	122	5	838985046
1	185	5	838983525
1	231	5	838983392
1	292	5	838983421
1	316	5	838983392

Now we have two objects movies and ratings and we will join these by the column movieId. The left_join will only add observations with matching movieID in the ratings object and in the movies object.

The result of the left_join will be stored in the 'movielens' object in memory. We can see the first 5 rows bellow:

```r
# join ratings and movie objects
movielens <- left_join(ratings, movies, by = 'movieId')

# Show the first 5 observations:
head(movielens,5) %>% kable()
```

userId	movieId	rating	timestamp	title	genres
1	122	5	838985046	Boomerang (1992)	Comedy|Romance
1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
1	231	5	838983392	Dumb & Dumber (1994)	Comedy
1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller
1	316	5	838983392	Stargate (1994)	Action|Adventure|Sci-Fi

The project requirements we will create a set from the original data to be only used on the final model. For that we are creating the validation set at 10% of the movielens data.We will also remove unnecessary objects from memory with the rm() command:

```r
# Setting the size of the validation set at 10 % of the edx data:
set.seed(1981, sample.kind = 'Rounding') # if using R 3.5 or earlier, use `set.seed(1981)`
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp_set <- movielens[test_index,]

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

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

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

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

# Remove objects no longer required from memory
rm(dl, movielens, ratings, movies, test_index, temp_set, removed)
```

Table 1: Custom summary of the EDX dataset

n_movies	n_users	rating_min	rating_max	release_min	release_max	rate_year_min	rate_year_max
10677	69878	0.5	5	1915	2008	1995	2009

Data preparation and wrangling

The original dataset *Movielens* (10,000,000 rows) has now been split into EDX with 90% of the rows and Validation with 10 % of the rows. From now on we will used solely the EDX data to model and train our machine learning algorithms.

To make our analysis more interesting we will include a year column that represents the year the rating was registered. At first glance the *timestamp* column seems like a random sequence of numbers, but it actually is a Unix Time (also known as Epoch time) representing the elapsed time in seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

The *timestamp* refers to when the rating was posted, note that it does not necessarily match the release date for the movie. We will convert the timestamp to a POSIXct format and extract the year. We will also drop the date_time and timestamp columns to reduce memory usage

```
# create a movie release year and rating year columns
edx <- edx %>%
  mutate( date_time = anytime(timestamp), # convert column to POSIXct format
          release_year = as.numeric(
                              str_sub(title,-5,-2)), # grab the 4 digits inside the parenthesis
          rating_year = as.numeric(format(date_time,
                                          format = '%Y'))) %>% # grab the year from date_time
  select(userId, movieId, rating, title, genres, release_year, rating_year)
```

Summary of the dataset

The summary() function presents us with a initial assessment of the data quartiles, min, max, mean and median values for each variable. For the character types it displays the vector length, class and mode.

```
##      userId         movieId          rating          title          
##  Min.   :    1   Min.   :    1   Min.   :0.500   Length:9000062    
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   Class :character  
##  Median :35742   Median : 1834   Median :4.000   Mode  :character  
##  Mean   :35871   Mean   : 4122   Mean   :3.512                     
##  3rd Qu.:53609   3rd Qu.: 3624   3rd Qu.:4.000                     
##  Max.   :71567   Max.   :65133   Max.   :5.000                     
##     genres           release_year   rating_year  
##  Length:9000062     Min.   :1915   Min.   :1995  
##  Class :character   1st Qu.:1987   1st Qu.:2000  
##  Mode  :character   Median :1994   Median :2002  
##                     Mean   :1990   Mean   :2002  
##                     3rd Qu.:1998   3rd Qu.:2005  
##                     Max.   :2008   Max.   :2009  
```

Custom summary of the edx data display the number of distinct users and movies, along with minimum and maximum values for rating value, release and rating year.

Data exploration and vizualization

For machine learning purposes, data comes in two forms: the *outcome* and the *features*. Before we start creating our models, we need to determine what inputs we will use as predictors or features) and what output will be our target variable (outcome). Here the rating will be our target, in other words, we will train different models and aim to predict the actual value the user would give to an unknown (unseen or unrated) movie to that particular user.

The EDX data set contains 69,878 users, 10,677 movies and 9,000,062 ratings plus genre classification and timestamp data.

The distribution of the rating column, as displayed on Figure 1:

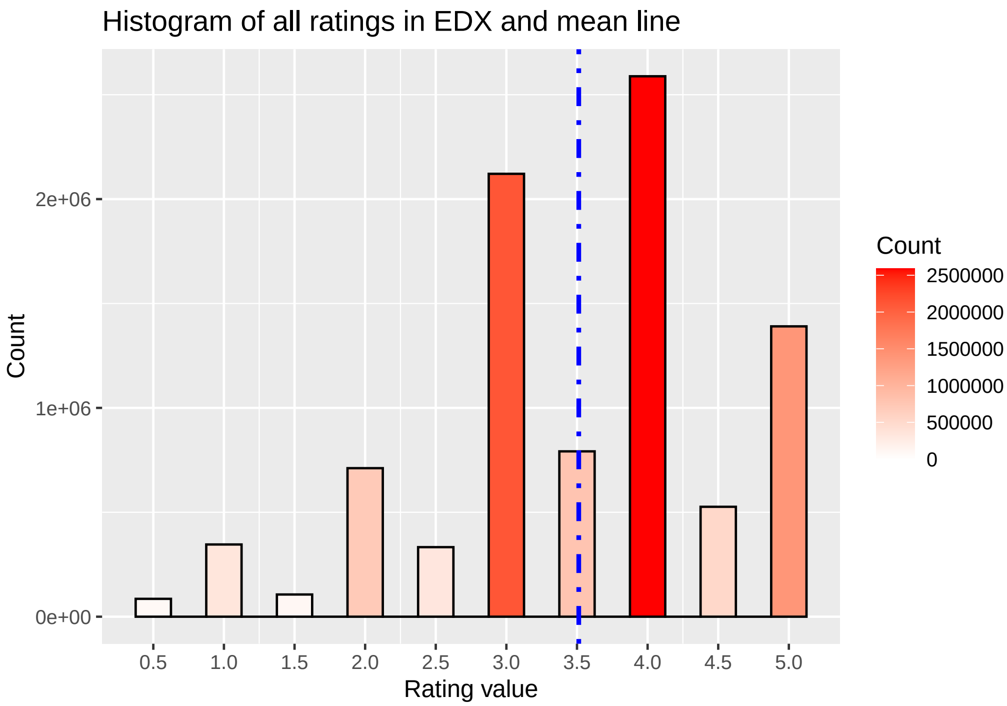


Figure 1: Histogram of all ratings in EDX and average line

The most frequent value 4 stars has 2,588,242 ratings, followed by 3 stars with 2,121,052 occurrences. The least frequent rating was 0.5 with 85,568 count. It is worth noting that no movie received a 0 rating.

Grouping the the dataset per user, we can get the counts for the maximum, minimum and average number of ratings per user:

Similarly we can group the data per movie and observe now what is the maximum, minimum and average number of ratings per movie:

We can see that there is, at least, one rating per movie title and a astonishing 31,323 ratings for one single title. Let's look into the top movies based on number of ratings and compare with top movies based on the actual rating average.

The table 5 displays the top 10 movies according to their average ratings. Note that the titles are different if we sort the data by descending number of ratings (table 6), I also added a column with the corresponding average values for those movies.

On the table bellow, we have the top 10 movies based on the quantity of ratings and also their average rating.

Let us now compare the average rating and count for the worst movies based on rating.

Table 2: Rating counts from most to least frequent

rating	Rating	Count
4.0	4.0	2588242
3.0	3.0	2121052
5.0	5.0	1390474
3.5	3.5	791816
2.0	2.0	711334
4.5	4.5	526377
1.0	1.0	345807
2.5	2.5	333069
1.5	1.5	106323
0.5	0.5	85568

Table 3: Maximum, minium and average number of ratings per user

max	min	avg
6605	11	424.35

Table 4: Maximum, minium and average number of ratings per movie

max	min	avg
31323	1	6786.69

Table 5: Top 10 movies per average rating

title	average_rating
Blue Light, The (Das Blaue Licht) (1932)	5.00
Fighting Elegy (Kenka erejii) (1966)	5.00
Satan's Tango (Sátántangó) (1994)	5.00
Shadows of Forgotten Ancestors (1964)	5.00
Sun Alley (Sonnenallee) (1999)	5.00
Human Condition III, The (Ningen no joken III) (1961)	4.83
Constantine's Sword (2007)	4.75
Human Condition II, The (Ningen no joken II) (1959)	4.75
I'm Starting From Three (Ricomincio da Tre) (1981)	4.75
More (1998)	4.75
Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) (1980)	4.75

Table 6: Top 10 movies per count of ratings, including the average rating

title	count	average_rating
Pulp Fiction (1994)	31323	4.16
Forrest Gump (1994)	30967	4.01
Silence of the Lambs, The (1991)	30329	4.20
Jurassic Park (1993)	29326	3.66
Shawshank Redemption, The (1994)	28041	4.46
Braveheart (1995)	26167	4.09
Fugitive, The (1993)	26070	4.01
Terminator 2: Judgment Day (1991)	26066	3.93
Star Wars: Episode IV - A New Hope (a.k.a. Star Wars) (1977)	25669	4.22
Apollo 13 (1995)	24406	3.89

Table 7: Worst 10 movies per average rating, incl. number of ratings

title	count	average_rating
Accused (Anklaget) (2005)	1	0.50
Alley Cats, The (1966)	1	0.50
Besotted (2001)	2	0.50
Hi-Line, The (1999)	1	0.50
War of the Worlds 2: The Next Wave (2008)	3	0.67
Hip Hop Witch, Da (2000)	14	0.82
SuperBabies: Baby Geniuses 2 (2004)	52	0.84
From Justin to Kelly (2003)	196	0.93
Disaster Movie (2008)	38	0.93
Dischord (2001)	1	1.00

Table 7 displays the worst 10 movies based on the average rating. It is interesting to observe that movies with 0.5 rating have 2 or less registered ratings. It is clear that a recommendation from one user would be too tailored for that user. Ideally we would like to have a much higher number of users who have rated all movies, but this is a desirable scenario, far from the reality in real life. Over the next section we will observe different techniques to deal with such difficulties.

Let's look into the relationship between number of ratings and the average rating per movie.

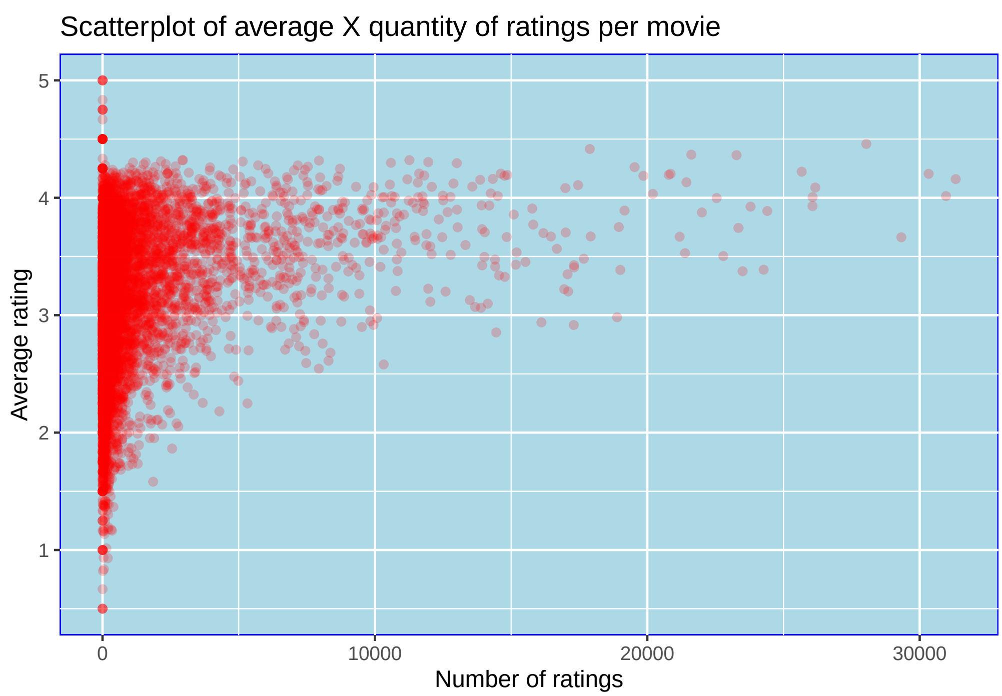


Figure 2: Scatterplot of average X quantity of ratings per movie

The plot depicted on Figure 2 demonstrates that most movies have a small number of ratings. We can also observe that movies with more than 20,000 ratings have average ratings above 3. While movies with lower number of ratings (e.g. less than 100) have a tendency to display higher rating variability, as shown bellow:

Figure 4 helps to expose trends that might be hard to visualize by just looking at a scatter plot, movies were grouped by the release year and presented in a box plot. It becomes easier to notice a higher rating trend between 1940-1950 with an average rating of roughly 3.9 stars.

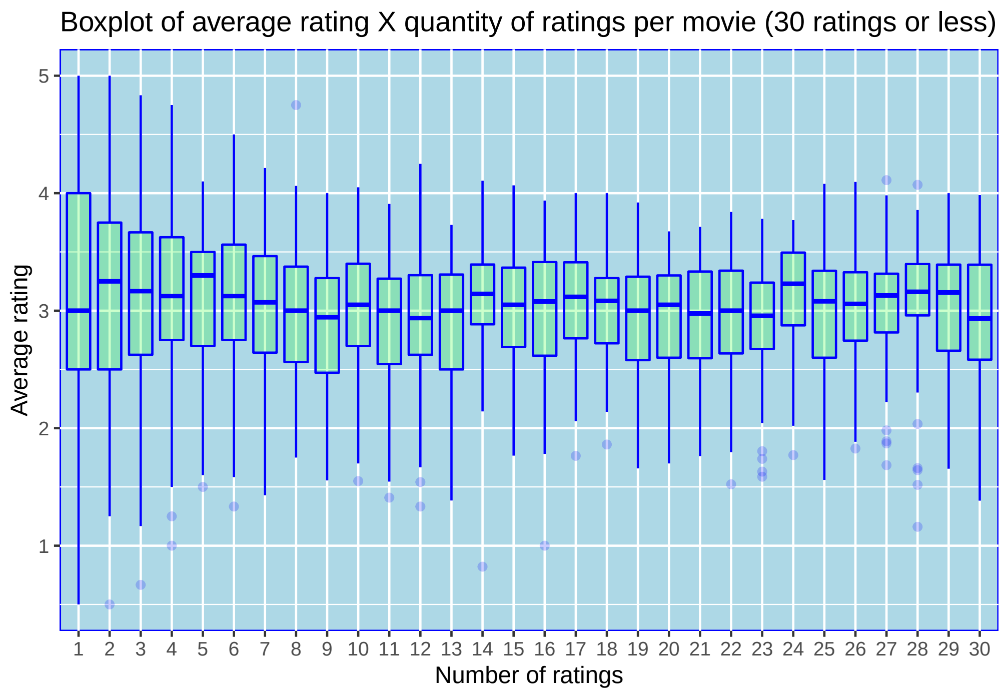


Figure 3: boxplot average rating versus number of ratings

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

METHODOLOGY

After data gathering, cleaning, exploring and visualization, the next step is to look into the methods we would like to implement and compare before analysing its performance on the final hold-out set: validation set. The first 4 methods will be our baseline for comparison with the collaborative methods from Recommenderlab and Recosystem.

Bellow is a list over the techniques we will compare during this project:

1 - Overall rating average
2 - Movie bias
3 - Movie and User biases
4 - Regularized Movie User biases
5 - RecommenderLab IBCF
6 - RecommenderLab UBCF
7 - RecommenderLab POPULAR
8 - Recosystem Parallel Matrix Factorization

The validation set created above will not be used to train or test our algorithms. So, we need to partition the *edx* set in train and test sets. It is important to determine how much data will be used to train versus test. We want enough observations to train but we also want have a decent proportion of *unseen* observations to test with. We also need to ensure that the same movieId and userId also appears in the test set, but not the same observations(rows).

The next step after cleaning and exploring the Movielens data is to train and test sets. We are going to reserve 20% of the edx set as test_set. To create the these sets we will use the function *createDataPartition()* from the *Caret* package. To replicate the same results set the seed to 1981.

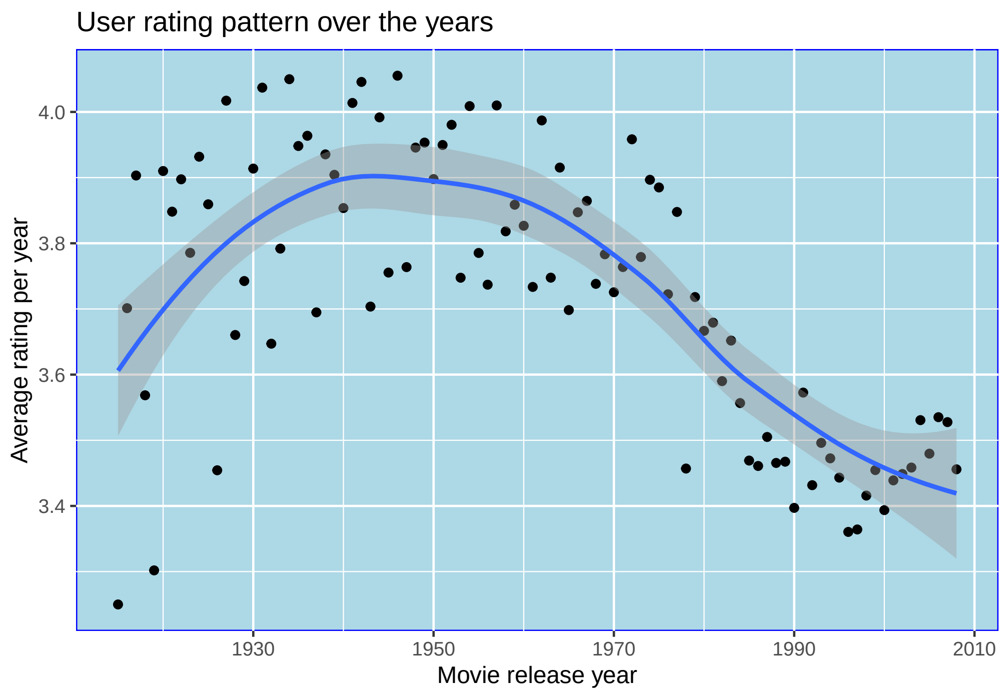


Figure 4: User rating pattern over the years

```
# To increase performance, I will drop all unused columns
edx <- edx %>% select(movieId, userId, rating)

# In order to replicate the results here, you need to set the seed to 1981
set.seed(1981, sample.kind = 'Rounding')

# Reserving 20% of the edx data for testing, train data is 80%:
test_idx <- createDataPartition(y = edx$rating, times = 1, p = 0.2, list = FALSE)
train_set <- edx[-test_idx,]
test_set <- edx[test_idx,]

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

Now let's inspect the dimensions of our sets:

```
dim(train_set)
```

```
## [1] 7200048       3
```

```
dim(test_set)
```

```
## [1] 1799973       3
```

Similarly as the evaluation approach used on the *Netflix Prize* competition, me will use the root mean squared error (RMSE) as the default standard to compare the performance of our models.

Table 8: Overall Rating Average

Method	RMSE
Overall rating average	1.060594

By definition RMSE is:

$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2}$

where N is the sample size, $\hat{y}_i$ are the predicted values and y_i are the corresponding observations.

Let's define our RMSE function:

```
RMSE <- function(true_ratings, predictions) {
  sqrt(mean((true_ratings - predictions)^2,na.rm = TRUE))
}
```

4.1 Overall average rating

We will start with the quickest and most basic way to predict a rating would be to guess the average overall rating from the train dataset. Applying the mean function to the rating column in the train_set we get 3.5123675. The simplest method would be to predict using the the average of the rating column. We can see the resulting RMSE bellow:

```
mu <- mean(train_set$rating)
average_rmse <- RMSE(test_set$rating, mu)

# Create a table to store our RMSE results
models_rmse <- tibble(Method = 'Overall rating average',  RMSE = average_rmse)
models_rmse %>% knitr::kable(caption = 'Overall Rating Average')
```

4.2 Movie bias

Some movies receive better ratings than others. We can include the average rating for a movie to our model. To analyze this further we will calculate the difference between the movie's average rating and the total average rating for all movies. If result is positive, it means that the movie is rated above the mean.

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

pred_movie_bias <- test_set %>%
  left_join(movie_bias, by = 'movieId') %>%
  mutate(prediction = mu + b_i) %>%
  pull(prediction)

movie_bias_rmse <- RMSE(test_set$rating, pred_movie_bias)
models_rmse <- rbind(models_rmse, tibble(Method = 'Movie bias', RMSE = movie_bias_rmse))
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Movie effect')
```

On following plot we can observe that the it is centered slightly to the left of the 0 (the rating received for the movie is equal to the overall average rating). This means that most movies had good ratings (above the average), but some had very low ratings, note the longer tail towards the left.

Table 9: Movie effect

Method	RMSE
Movie bias	0.9442922

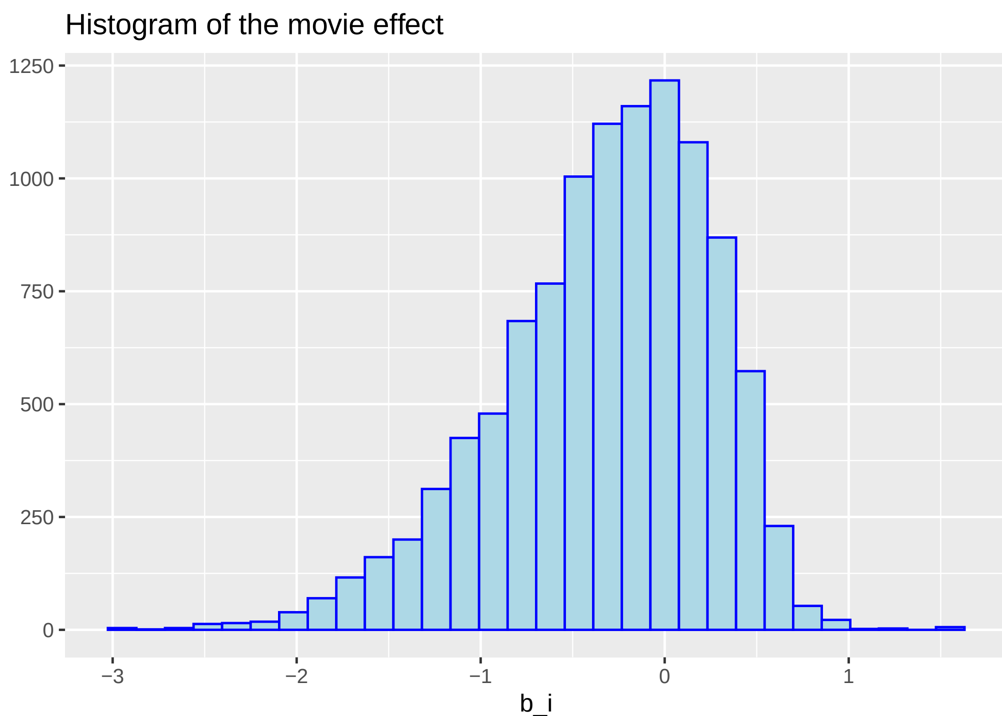


Figure 5: Movie effect histogram

Table 10: Movie-User effect

Method	RMSE
Movie and User biases	0.8669228

4.3 Movie and User biases

We can improve our predictions by adding a user effect to our model. Some users are very rigorous about their ratings and others tend to give great ratings that do not represent the 'quality' of the film.

```
# Calculate movie_user_bias
movie_user_bias <- train_set %>%
  left_join(movie_bias, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))

pred_movie_user <- test_set %>%
  left_join(movie_bias, by = 'movieId') %>%
  left_join(movie_user_bias, by = 'userId') %>%
  mutate(prediction = mu + b_i + b_u) %>%
  pull(prediction)

movie_user_rmse <- RMSE(test_set$rating, pred_movie_user)
models_rmse <- rbind(models_rmse,
                     tibble(Method = 'Movie and User biases',
                            RMSE = movie_user_rmse) )
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Movie-User effect')
```

The RMSE achieved by the Movie and User bias was 0.8669228. Figure 6 displays the distribution of the movie_user:

We have observed that by increasing the number of predicting variables(average, movie/user biases) we were able to reduce the RMSE.

It is worth to note that some movies received a tens of thousands of ratings while others have just a handful. This big discrepancy creates untrustworthy estimates. We can try to account for this by introducing penalties for these occurrences.

4.4 Regularized Bias

In order to find a balance for minimizing the our model's expected error, we will include additional information to prevent overfitting (eg. model has 100% accuracy on train set, but 50% accurate on test set). So here we include an lambda value as independent variable.

```
# Create a sequence of values to test
lambdas <- seq(0, 7, 0.25)
# Calculate the rmses for the lambda values.
rmses <- sapply(lambdas, function(lambda){
  mu <- mean(train_set$rating)
  b_i <- train_set %>% group_by(movieId) %>%
    summarize(b_i = sum(rating - mu) / (n() + lambda))

  b_u <- train_set %>% left_join(b_i, by = 'movieId') %>%
```

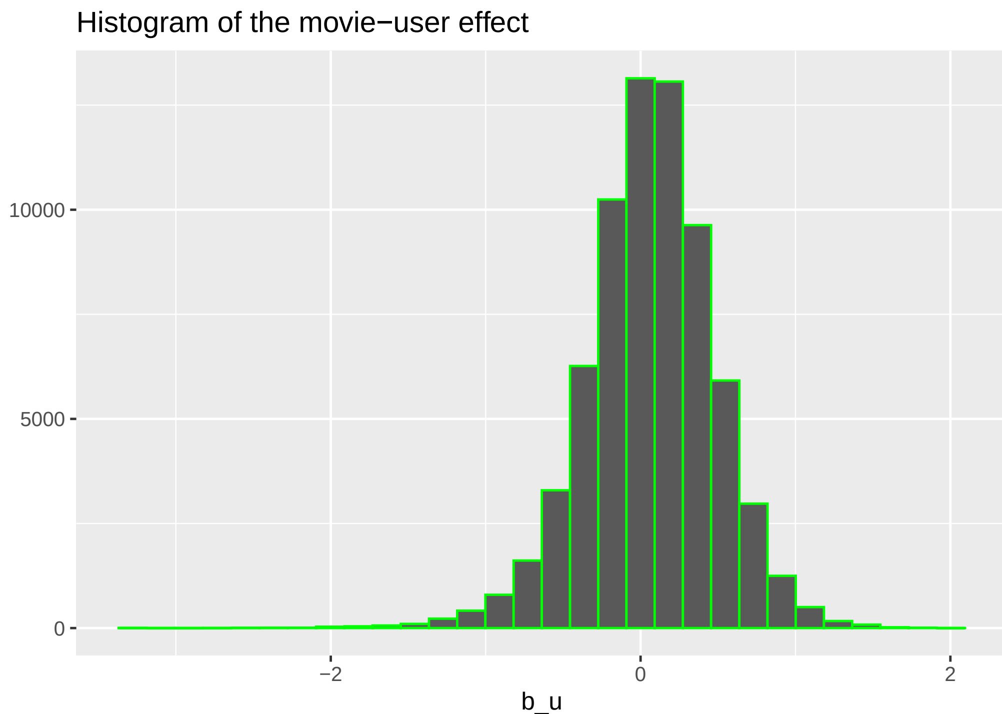


Figure 6: Histogram of Movie-User effect

```
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu) / (n() + lambda))

  predictions <- test_set %>%
    left_join(b_i, by = 'movieId') %>%
    left_join(b_u, by = 'userId') %>%
    mutate(prediction = mu + b_i + b_u) %>%
    pull(prediction)

  RMSE(test_set$rating, predictions)})
```

In order to determine the best value for the independent variable lambda, we initially ran calculations using values from 0 to 20. But to better fit the plot reduced the size of the lambda vector to 0 to 7.

The best value for lambda is 4.75 and the corresponding RMSE is 0.8662604. We will recalculate the model 3 with the new lambda correction.

```
# Train the model
reg_movie_bias <- train_set %>%
  group_by(movieId) %>%
  summarize(r_b_i = sum(rating - mu) / (n() + lambda),
            r_n_i = n())

reg_movie_user_bias <- train_set %>%
  left_join(reg_movie_bias, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(r_b_u = sum(rating - mu - r_b_i) / (n() + lambda),
                              r_n_u = n())
```

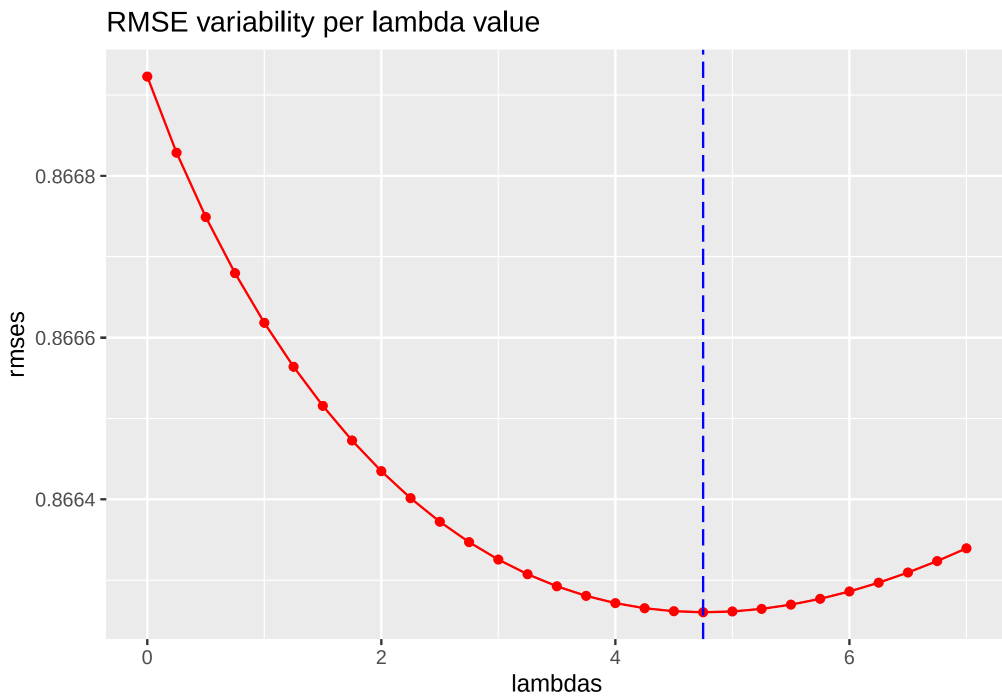


Figure 7: Lambda Versus RMSE plot

Table 11: Regularized Movie User biases

Method	RMSE
Regularized Movie User biases	0.8662604

```r
# Test the model on the test_set
pred_reg_movie_user <- test_set %>%
  left_join(reg_movie_bias, by = 'movieId') %>%
  left_join(reg_movie_user_bias, by = 'userId') %>%
  mutate(prediction = mu + r_b_i + r_b_u) %>%
  pull(prediction)

reg_movie_user_rmse = RMSE(test_set$rating, pred_reg_movie_user)

# Add the latest result to the rmse table
models_rmse <- rbind(models_rmse, tibble(Method = 'Regularized Movie User biases',
                                         RMSE = reg_movie_user_rmse))
# Show the rmses table
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Regularized Movie User biases')
```

Collaborative Filtering

Collaborative filtering (CF) uses given rating data by many users for many items as the basis for predicting missing ratings and/or for creating a top-N recommendation list for a given user, called the active user. Formally, we have a set of users U = $\{u_1, u_2, ..., u_m\}$ and a set of items I = $\{i_1, i_2, ..., i_n\}$. Ratings are stored in a m × n user-item rating matrix R = (r_{jk}) where each row represents a user u_j with $1 \leq j \leq m$ and columns represent items i_k with $1 \leq k \leq n$. r_{jk} represents the rating of user u_j for item i_k. Typically only a small fraction of ratings are known and for many cells in R the values are missing. Many algorithms operate on ratings on a specific scale (e.g., 1 to 5 (stars)) and estimated ratings are allowed to be within an interval of matching range (e.g., [1, 5]). From this point of view recommender systems solve a regression problem.

The package Recommenderlab has the following alternatives available.

```
##  [1] "HYBRID_realRatingMatrix"       "ALS_realRatingMatrix"
##  [3] "ALS_implicit_realRatingMatrix" "IBCF_realRatingMatrix"
##  [5] "LIBMF_realRatingMatrix"        "POPULAR_realRatingMatrix"
##  [7] "RANDOM_realRatingMatrix"       "RERECOMMEND_realRatingMatrix"
##  [9] "SVD_realRatingMatrix"          "SVDF_realRatingMatrix"
## [11] "UBCF_realRatingMatrix"
```

The following code creates a matrix object from the EDX data:

```
# creating a copy of edx data and change data types.
edx_copy <- edx

# Coercing the values to numeric
edx_copy$userId <- as.numeric(as.factor(edx_copy$userId))
edx_copy$movieId <- as.numeric(as.factor(edx_copy$movieId))
edx_copy$rating <- as.numeric(edx_copy$rating)

# Create a sparseMatrix object
ratings_matrix <-
  sparseMatrix(i = edx_copy$userId,
               j = edx_copy$movieId,
               x = edx_copy$rating,

               dims = c(length(unique(edx_copy$userId)),
                        length(unique(edx_copy$movieId))),

               dimnames = list(paste('user_', unique(edx_copy$userId), sep = ''),
                               paste('movie_', unique(edx_copy$movieId), sep = '')))

# Show the ratings_matrix structure
str(ratings_matrix)
```

```
## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:9000062] 4 13 16 19 20 25 26 27 28 29 ...
##   ..@ p       : int [1:10678] 0 23790 34560 41603 43187 49574 61920 69196 70023 72309 ...
##   ..@ Dim     : int [1:2] 69878 10677
##   ..@ Dimnames:List of 2
##   .. ..$ : chr [1:69878] "user_1" "user_2" "user_3" "user_4" ...
##   .. ..$ : chr [1:10677] "movie_121" "movie_184" "movie_290" "movie_326" ...
##   ..@ x       : num [1:9000062] 1 3 3 5 5 5 4 3 3 4 ...
##   ..@ factors : list()
```

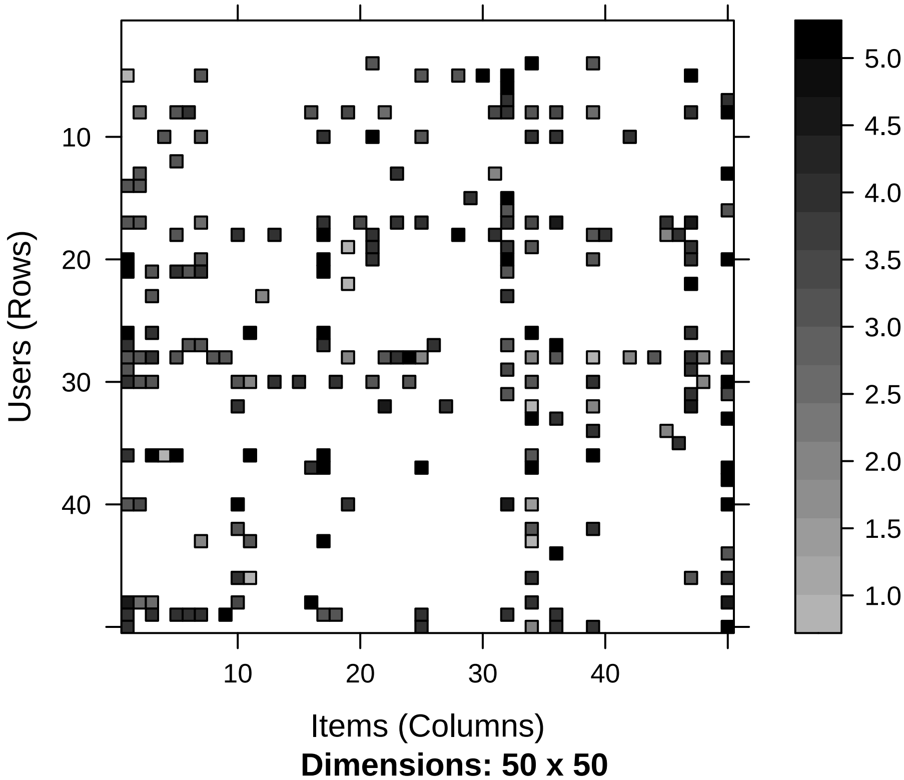
The now the ratings_matrix object needs to be converted to a *realRatingMatrix* object:

Create a *realRatingMatrix* object:

```r
# Create a realRatingMatrix object
recom_matrix <- new('realRatingMatrix', data = ratings_matrix)
```

The dimensions of the matrix are 69,878 rows (Movies) and 10,677 columns (Users)

Let's look at a heatmap for the first 100 movies and users in the recom_matrix object:



We can see that the matrix is very sparse. This is the main challenge to be solved: finding the right value to fill in the 'blanks'.

Using the *quantiles()* function over the columns and rows, we determine these numbers for movies and users respectively. We will keep enough users and movies to retain 90% of the original data variability and optimize resources.

```r
movies_min <- quantile(rowCounts(recom_matrix), 0.9);movies_min
```

```
## 90%
## 302
```

```r
users_min <- quantile(colCounts(recom_matrix), 0.9);users_min
```

```
##    90%
## 2152.8
```

```r
recom_matrix <- recom_matrix[rowCounts(recom_matrix) > movies_min,
                             colCounts(recom_matrix) > users_min]
```

After reducing the size of our matrix even more by applying the movie and user cut-offs, the new matrix has 6968 rows (Movies) and 1068 columns (Users).

```
## Formal class 'realRatingMatrix' [package "recommenderlab"] with 2 slots
##   ..@ data     :Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   .. .. ..@ i       : int [1:2311857] 1 2 3 5 6 8 9 11 12 13 ...
##   .. .. ..@ p       : int [1:1069] 0 4921 8149 9776 11162 14265 15903 16376 19706 22284 ...
##   .. .. ..@ Dim     : int [1:2] 6968 1068
##   .. .. ..@ Dimnames:List of 2
##   .. .. .. ..$ : chr [1:6968] "user_8" "user_17" "user_28" "user_30" ...
##   .. .. .. ..$ : chr [1:1068] "movie_121" "movie_184" "movie_290" "movie_352" ...
##   .. .. ..@ x       : num [1:2311857] 3 3 4 4.5 3 4 3 5 5 3 ...
##   .. .. ..@ factors : list()
##   ..@ normalize: NULL
```

Prepare the train and test sets to use with the Recommenderlab package:

```
# split the edx data in train and test sets
set.seed(1981, sample.kind = 'Rounding')

# Create train and test sets, with 80% and 20% of the edx data set, respectively.
evaluation <- evaluationScheme(recom_matrix,
                               method='split',
                               train = 0.8,
                               given=-5,
                               goodRating = 3,
                               k = 1)
evaluation
```

```
## Evaluation scheme using all-but-5 items
## Method: 'split' with 1 run(s).
## Training set proportion: 0.800
## Good ratings: >=3.000000
## Data set: 6968 x 1068 rating matrix of class 'realRatingMatrix' with 2311857 ratings.
```

```
# Show training_set info
getData(evaluation, 'train')
```

```
## 5574 x 1068 rating matrix of class 'realRatingMatrix' with 1851980 ratings.
```

```
getData(evaluation,'known')
```

```
## 1394 x 1068 rating matrix of class 'realRatingMatrix' with 452907 ratings.
```

```
# Show test set info
getData(evaluation, 'unknown')
```

```
## 1394 x 1068 rating matrix of class 'realRatingMatrix' with 6970 ratings.
```

4.5 Recommenderlab Item-based Collaborative Filtering - IBCF

The idea behind the Item-based Collaborative Filtering is to measure the similarity between the items that target users rates/interacts with and other items. The similarity can be calculated using Pearson Correlation or Cosine Similarity.

Table 12: Recommenderlab IBCF

Method	RMSE
RecommenderLab IBCF	1.149819

```
# Show the tune parameters for the IBCF model
rec_models$IBCF_realRatingMatrix$parameters
```

```
## $k
## [1] 30
##
## $method
## [1] "Cosine"
##
## $normalize
## [1] "center"
##
## $normalize_sim_matrix
## [1] FALSE
##
## $alpha
## [1] 0.5
##
## $na_as_zero
## [1] FALSE
```

```
# set seed
set.seed(1981, sample.kind = 'Rounding')

# Generate the model
IBCF_method <- Recommender(getData(evaluation, 'train'), method = 'IBCF',
                           param=list(normalize = 'center',
                                      method='Cosine', k = 30)) # k = 350
# Make predictions
pred_IBCF_method <- predict(IBCF_method,
                            getData(evaluation, 'known'),
                            type = 'ratings')
# Test the method
IBCF_method_rmse <- calcPredictionAccuracy(pred_IBCF_method, getData(evaluation, 'unknown'))

# Update the rmse table
models_rmse <- rbind(models_rmse, tibble(Method = 'RecommenderLab IBCF',
                                         RMSE = IBCF_method_rmse[1]))
# Show the table
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Recommenderlab IBCF')
```

4.6 Recommenderlab User-based Collaborative Filtering (UBCF)

The User-based Collaborative Filtering is a technique where in order to predict items (movies) that a *user* might based on what ratings were given to that movie by other users that have similar taste to the target

Table 13: Recommenderlab UBCF

Method	RMSE
RecommenderLab UBCF	0.8243024

user. The idea is to attribute a higher weight to ratings given by more similar users then those ratings given by users that are less similar. The algorithm uses a similarity factor to make these adjustments.

```
# set seed
set.seed(1981, sample.kind = 'Rounding')

# Create UBCF model
UBCF_method <- Recommender(getData(evaluation, 'train'), method = 'UBCF',
                           param=list(normalize = 'center', method = 'Cosine', nn = 50))

pred_UBCF_method <- predict(UBCF_method, getData(evaluation, 'known'), type = 'ratings')

# Test the models accuracy
UBCF_method_rmse <- calcPredictionAccuracy(pred_UBCF_method, getData(evaluation, 'unknown'))

# Update the table
models_rmse <- rbind(models_rmse, tibble(Method = 'RecommenderLab UBCF',
                                         RMSE = UBCF_method_rmse[1]))
# Show the table
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Recommenderlab UBCF')
```

4.7 Recommenderlab Popular Items

```
set.seed(1981, sample.kind = 'Rounding')
popular_method <- Recommender(recom_matrix, method = 'POPULAR',
                              param = list(normalize = 'center'))

# Evaluating the rmse for the popular_method
set.seed(1981, sample.kind = 'Rounding')
popular_method <- Recommender(getData(evaluation, 'train'),
                              method = 'POPULAR')
pred_popular_method <- predict(popular_method,
                               getData(evaluation, 'known'),
                               type = 'ratings')

popular_method_rmse <- calcPredictionAccuracy(pred_popular_method, getData(evaluation,'unknown'))

models_rmse <- rbind(models_rmse, tibble(Method = 'RecommenderLab POPULAR',
                                         RMSE = popular_method_rmse[1]))
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Recommenderlab Popular Items')
```

```
#prediction example on the first 10 users
pred_popular_method <- predict(popular_method, recom_matrix[1:10], type = 'ratings')
as(pred_popular_method, 'matrix')[1:10,1:10]
```

Table 14: Recommenderlab Popular Items

Method	RMSE
RecommenderLab POPULAR	0.8481354

```
##          movie_121 movie_184 movie_290 movie_352 movie_353 movie_359 movie_367
## user_8    3.874858        NA  2.934670        NA        NA  3.098055  2.608229
## user_17         NA        NA  3.005867  2.882770  3.841006        NA  2.679426
## user_28         NA        NA        NA        NA  3.130653  2.458899        NA
## user_30         NA        NA        NA  2.811691  3.769927  3.098173  2.608347
## user_43   4.668703  3.749353  3.728515  3.605418  4.563653        NA  3.402073
## user_48         NA        NA        NA  3.491521  4.449757  3.778003  3.288177
## user_57         NA        NA  2.660380  2.537283  3.495519  2.823764  2.333938
## user_70   4.463139  3.543789  3.522951  3.399854  4.358090  3.686336  3.196509
## user_88         NA  3.041259  3.020420  2.897323        NA  3.183805  2.693979
## user_103        NA  2.785971  2.765133  2.642036        NA  2.928518  2.438692
##          movie_374 movie_463 movie_581
## user_8    3.335325  3.473643  3.442235
## user_17   3.406521  3.544840  3.513432
## user_28   2.696169  2.834487  2.803080
## user_30         NA        NA  3.442353
## user_43   4.129169        NA  4.236080
## user_48         NA  4.153591  4.122184
## user_57   3.061034        NA  3.167945
## user_70   3.923605  4.061923  4.030516
## user_88   3.421075  3.559393  3.527986
## user_103  3.165787  3.304106        NA
```

4.7 Parallel Matrix Factorization

This method is provided by the Recosystem package. To take advantage of the configuration options we selected number of threads (nthreads = 4) to match the 4 virtual cores on a dual-core mac computer with hyper-threading technology, we also limited the number of iteration at 10.

```
set.seed(1981, sample.kind='Rounding')

train_set_reco_mf <- train_set %>% select(userId, movieId, rating)
#train_set_reco_mf <- as.matrix(train_set_reco_mf)

test_set_reco_mf <- test_set %>% select(userId, movieId, rating)
#test_set_reco_mf <- as.matrix(test_set_reco_mf)


# Create sample and test sets for the edx:
train_mf <- with(train_set_reco_mf, data_memory(user = userId, item = movieId, rating = rating))

test_mf <- with(test_set_reco_mf, data_memory(user = userId, item = movieId, rating = rating))

# Create the recosystem model
reco <- recosystem::Reco()

# Select tuning parameters:
```

```r
opts <- reco$tune(train_mf, opts = list(dim = c(10, 20, 30), lrate = c(1.0, 0.2),
                                        costp_l1 = 0, costq_l1 = 0,
                                        nthread = 4, niter = 10))
opts
```

```
## $min
## $min$dim
## [1] 20
##
## $min$costp_l1
## [1] 0
##
## $min$costp_l2
## [1] 0.01
##
## $min$costq_l1
## [1] 0
##
## $min$costq_l2
## [1] 0.1
##
## $min$lrate
## [1] 0.2
##
## $min$loss_fun
## [1] 0.8097404
##
##
## $res
##    dim costp_l1 costp_l2 costq_l1 costq_l2 lrate  loss_fun
## 1   10        0     0.01        0     0.01   1.0 1.0594623
## 2   20        0     0.01        0     0.01   1.0 1.0595512
## 3   30        0     0.01        0     0.01   1.0 1.0603033
## 4   10        0     0.10        0     0.01   1.0 1.0597516
## 5   20        0     0.10        0     0.01   1.0 1.0606808
## 6   30        0     0.10        0     0.01   1.0 1.0605604
## 7   10        0     0.01        0     0.10   1.0 1.0605747
## 8   20        0     0.01        0     0.10   1.0 1.0613920
## 9   30        0     0.01        0     0.10   1.0 1.0586138
## 10  10        0     0.10        0     0.10   1.0 1.0602472
## 11  20        0     0.10        0     0.10   1.0 1.0593165
## 12  30        0     0.10        0     0.10   1.0 1.0599054
## 13  10        0     0.01        0     0.01   0.2 0.8261620
## 14  20        0     0.01        0     0.01   0.2 0.9262328
## 15  30        0     0.01        0     0.01   0.2 0.9785192
## 16  10        0     0.10        0     0.01   0.2 0.8260972
## 17  20        0     0.10        0     0.01   0.2 0.9170379
## 18  30        0     0.10        0     0.01   0.2 0.9193607
## 19  10        0     0.01        0     0.10   0.2 0.8291398
## 20  20        0     0.01        0     0.10   0.2 0.8097404
## 21  30        0     0.01        0     0.10   0.2 0.8098713
## 22  10        0     0.10        0     0.10   0.2 0.8423553
## 23  20        0     0.10        0     0.10   0.2 0.8289382
```

```
## 24  30        0     0.10        0     0.10   0.2 0.8279250
```

```
# Train the model
MF_model <- reco$train(train_mf, opts = c(opts$min, nthread = 4, niter = 10))
```

```
## iter      tr_rmse          obj
##    0       0.9705   8.9309e+06
##    1       0.8684   7.4940e+06
##    2       0.8253   6.9419e+06
##    3       0.7990   6.6420e+06
##    4       0.7810   6.4538e+06
##    5       0.7685   6.3225e+06
##    6       0.7591   6.2291e+06
##    7       0.7522   6.1614e+06
##    8       0.7467   6.1092e+06
##    9       0.7421   6.0658e+06
```

```
MF_model
```

```
## [=== Fitted Model ===]
##
## Path to model file   = /var/folders/wj/cxyjtyk92d16s890xxjkvvwr0000gn/T//RtmpljwGCw/model.txt
## Number of users      = 71568
## Number of items      = 65134
## Number of factors    = 20
##
##
## [=== Training Options ===]
##
## Loss function        = Squared error (L2-norm)
## L1 penalty for P     = 0
## L2 penalty for P     = 0.01
## L1 penalty for Q     = 0
## L2 penalty for Q     = 0.1
## Learning rate        = 0.2
## NMF                  = FALSE
## Number of iterations = 10
## Number of threads    = 4
## Verbose              = TRUE
```

```
# calculate the predictions
predictions_mf <- reco$predict(test_mf, out_memory())

head(predictions_mf)
```

```
## [1] 4.513809 3.962354 4.659603 2.684497 3.186404 4.307757
```

```
MF_method_rmse <- RMSE(predictions_mf, test_set_reco_mf$rating)

models_rmse <- rbind(models_rmse, tibble(Method = 'Recosystem Parallel Matrix Factorization',
                                         RMSE = MF_method_rmse[1]))
models_rmse[nrow(models_rmse):nrow(models_rmse),] %>% kable(caption = 'Recosystem Parallel Matrix Facto
```

Table 15: Recosystem Parallel Matrix Factorization

Method	RMSE
Recosystem Parallel Matrix Factorization	0.7993393

Table 16: Performance comparison of the methods on the test set

Method	RMSE
Overall rating average	1.0605944
Movie bias	0.9442922
Movie and User biases	0.8669228
Regularized Movie User biases	0.8662604
RecommenderLab IBCF	1.1498192
RecommenderLab UBCF	0.8243024
RecommenderLab POPULAR	0.8481354
Recosystem Parallel Matrix Factorization	0.7993393

Table 6 lists all the models and their performance results when making predictions on the test_set:

```
models_rmse %>% knitr::kable(caption = 'Performance comparison of the methods on the test set')
```

VALIDATION

As wee can observe from the table above, the best performing method was the Recosystem - Parallel Matrix Factorization with a 0.7993393 RMSE. This was our best performing model. We will use it for the final test: how it performs on the validation set (unseen/unknown data).

Firstly we need to prepare the validation set to the format needed:

Making predictions on the validation data:

```
# Select the userId, movieId and rating columns from the validation set
validation_reco_mf <- validation %>% select(userId, movieId, rating)

# Load the validation data using the data_memory() function from recosystem
valid_mf <- with(validation_reco_mf, data_memory(user = userId,
                                                  item = movieId,
                                                  rating = rating))

# Create predictions using the Parallel Matrix Factorization model trained above
pred_validation_mf <- reco$predict(valid_mf, out_memory())

# Calculate the RMSE for the model on the validation set
MF_validation_rmse <- RMSE(validation_reco_mf$rating, pred_validation_mf)
```

The Parallel Matrix Factorization method performed with a RMSE of 0.7978153 on the validation set.

CONCLUSION

It has been very interesting to explore this new data set from the Grouplens Research Group. One of my favorite sections of this project was the vizualizations, for me personally conveying the information with visual aids greatly improves how well we understand its variability and relations to other variables in the data. The Movielens data was very interesting to work with and the size of the data set has proven to be more than my MacBook Air (i5, 8Gb RAM) could handle for some other machine learning packages like the H2O.

The final performance of our Recosystem Parallel Matrix Factorization algorithm was 0.7993393 (on test set) versus 0.7978153 (on validation set) shows good stability of the prediction precision over unknown data. I am pleased with the result and eager to check the performance of more computationally intensive methods and their performances in the future.

REFERENCES:

- The MovieLens Datasets:

  F. Maxwell Harper and Joseph A. Konstan. 2015. History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI='http://dx.doi.org/10.1145/2827872'.

- Recosystem:

  https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html

- Recommenderlab:

  https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf