MovieLens Project

Christina Papadopoulou

2022-11-02

Introduction

A recommendation system in machine learning is an algorithm that uses data in order to predict what users are looking for and therefore, to make specific recommendations.There are a lot of things that can be recommended by the recommendation system like products, advertisements, books, movies, jobs etc. For example, Netflix recommends movies and series based on a recommendation system. The same does YouTube when recommending videos, Amazon when recommending products, Linkedin when recommending jobs, Facebook when recommending advertisements or other members etc. Basically, all these companies collect massive datasets based on various criteria, including clicks, likes, past purchases, search history, ratings and other factors, that can be subsequently used for user/customer recommendations. Recommendation systems are considered very useful both for the users/customers and the companies. Users discover services and products that may not have found by their own and companies increase engagement with users and the platform.

In this project, a movie recommendation system is going to be created. Hence, we are going to build a model which will be able to recommend a movie to a specific user for any possible values of the features. Basically, we will predict ratings a user might give to an item or in other words, we will build a model that will be able to predict the preference of a user towards a movie. There are thousands of movies and therefore, each of us has its own preferences. For example some people like genre-specific movies, such as comedy, action, thriller, romance, drama movies. Other people may focus on the actors or the directors. There are many other things that we could take into account. Here, in order to do this, we are going to use the MovieLens 10M Dataset from grouplens. This data set includes almost 10 million movie ratings applied to approximately 10,000 movies by more than 70,000 users. More specifically the attributes, the variables, that will act as inputs in our model are six and will be shown in the following section.

The key steps that were performed are the data wrangling in order to achieve a tidy form of our data. Visualization of different patterns are executed in order to gain some useful insights of our data and finally we proceeded in building our algorithm for movie recommendation.

Methods and analysis

In this section, we are going to explain the techniques and methods used till we get to our final model.

Import data

First of all, we are going to download our data:

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

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

From above, we get a zip file that has three files in it, the movies.dat, the ratings.dat and the tags.dat files. We are going to use two of them, the movies and the ratings files in order to create our final dataset. Therefore, we will do some data wrangling in order to get a tidy form of our data. First of all, we have to unzip the files and get to the form we want.

```
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))
```

Therefore, if we join the above data frames, we get the movielens dataset which we will use to build our movie recommendation algorithm.

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

We will now split our data in order to get the validation set and the edx set. The edx set will be used to develop our algorithm and the validation set to test our final model after completing the training. We will partition our movielens data in a 90:10 ratio for edx and validation set respectively.

```
# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding")
```

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

```
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]
```

We also have to make sure that the users (userId) and the movies (movieId) are in both the validation and the edx set.

```
validation <- temp %>%
      semi_join(edx, by = "movieId") %>%
      semi_join(edx, by = "userId")
```

We will also add the rows removed from validation set back into edx set, in order to include these observations in our training.

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

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

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

Finally, we will remove from the memory the temporary files since, we are not going to use them later on.

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

Data exploration and visualization

First of all, let us see the structure of our edx data

```
str(edx)
```

```
## Classes 'data.table' and 'data.frame':   9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : num  122 185 292 316 329 355 356 362 364 370 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 838984474 838983653 838984885 83
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|A
##  - attr(*, ".internal.selfref")=<externalptr>
```

edx is a data frame of 6 variables(columns) and 9000055 observations(rows). The variables userId (the identification number of a user), movieId (the identification number of a movie), rating, timestamp (the time the rating was made) are quantitative. The rating variable is the one we would like to predict. The title (the title of a movie) and genres (the genre or genres in which the movie belongs) variables are categorical.

Let us now see the number of distinct movies

```
n_distinct(edx$movieId)
```

```
## [1] 10677
```

and the number of distinct users

```
n_distinct(edx$userId)
```

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

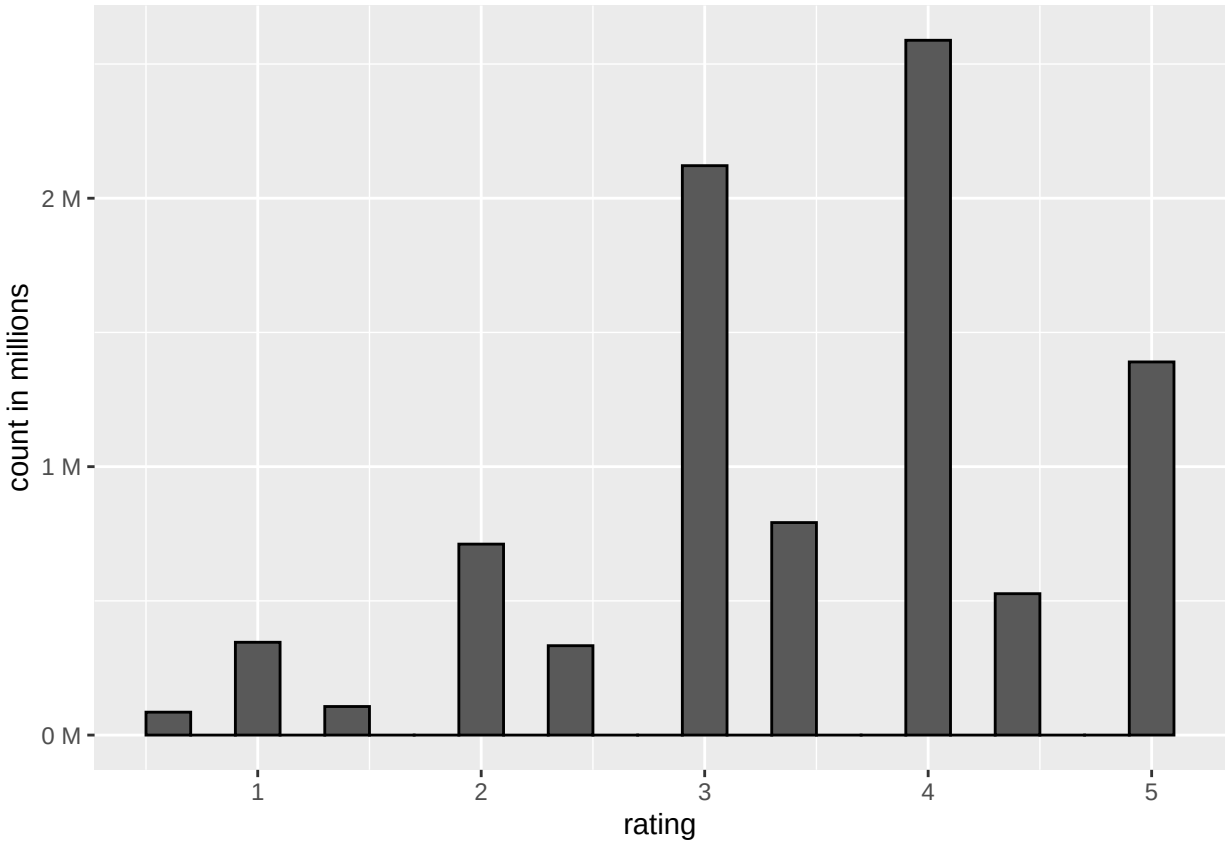
By multiplying these two numbers we get around 746 millions.This means that if each user rated each movie, we should have had around 746 millions observations. Instead we have only around 9 millions, this means that every user has not rated every movie. Therefore, we have a lot of missing values. Our task, in some way, is to fill all these missing values in a very large matrix where rows are the users and columns the movies. Hence, we are trying to predict ratings that users would give to different movies. The difficulty here, is that each outcome has a different set of predictors. Basically, we can use the entire matrix as predictors for each cell.

Let us now have a look at our data with the summary statistics

```
summary(edx)
```

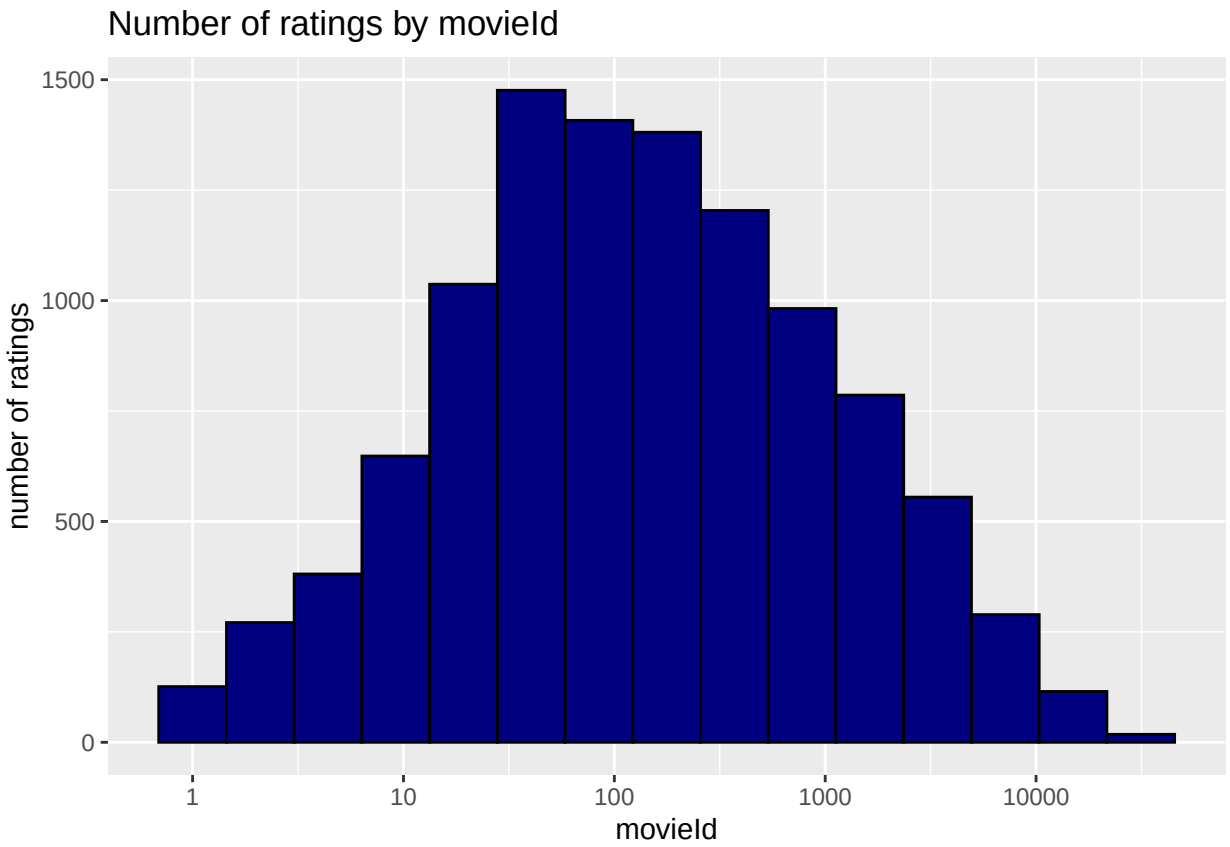
```
##      userId         movieId          rating        timestamp
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08
##  Median :35738   Median : 1834   Median :4.000   Median :1.035e+09
##  Mean   :35870   Mean   : 4122   Mean   :3.512   Mean   :1.033e+09
##  3rd Qu.:53607   3rd Qu.: 3626   3rd Qu.:4.000   3rd Qu.:1.127e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
##     title              genres
##  Length:9000055     Length:9000055
##  Class :character   Class :character
##  Mode  :character   Mode  :character
##
##
##
```

From the summary statistics, we can see that the minimum of a rating is 0.5,the mean is 3.512, the median 4.0 and the maximum is 5.Therefore there is a negative skewness. The users tend to rate above the average. This is something that we can also see in our following plot:

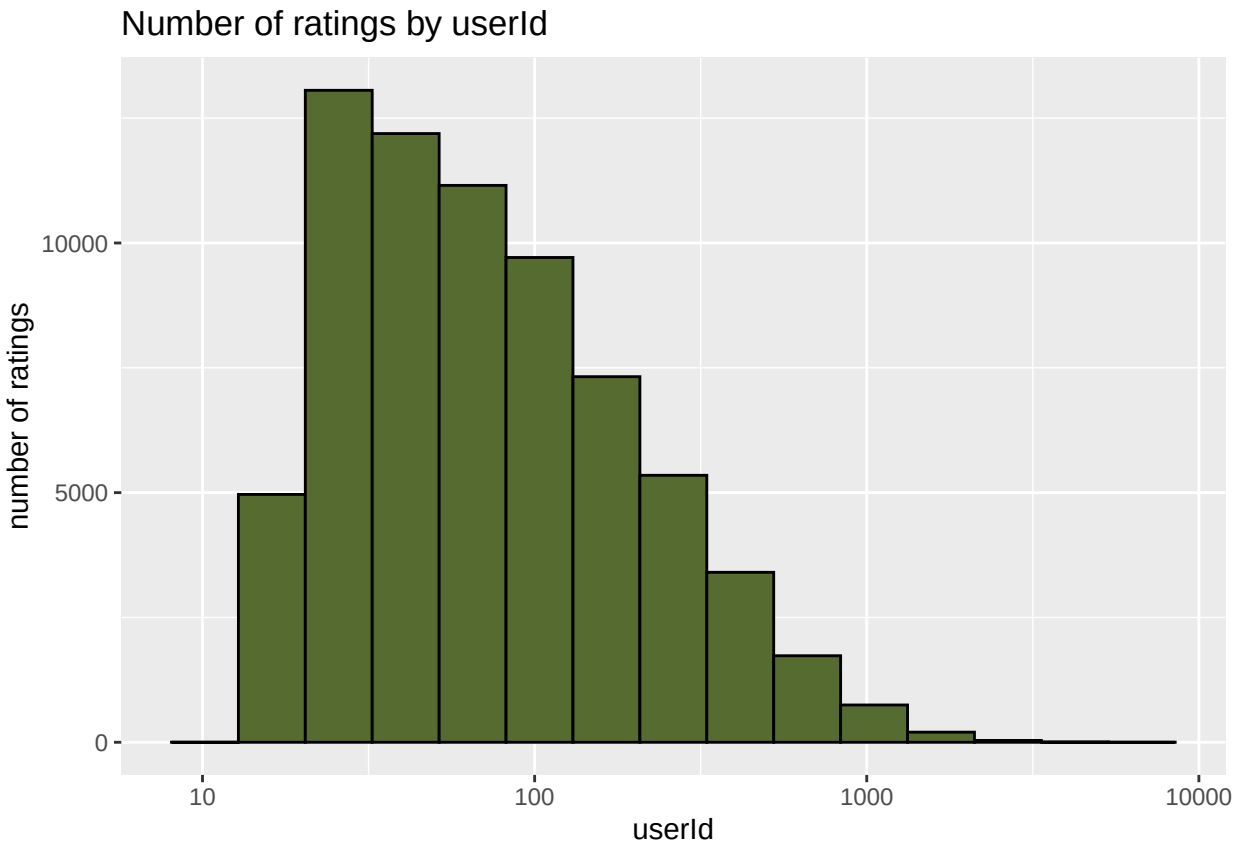


We see that most of the users have given a rating of four.

Movies and users variables Now, we will look some other properties of the data. The number of ratings by movieId and the number of ratings by userId will be plotted.

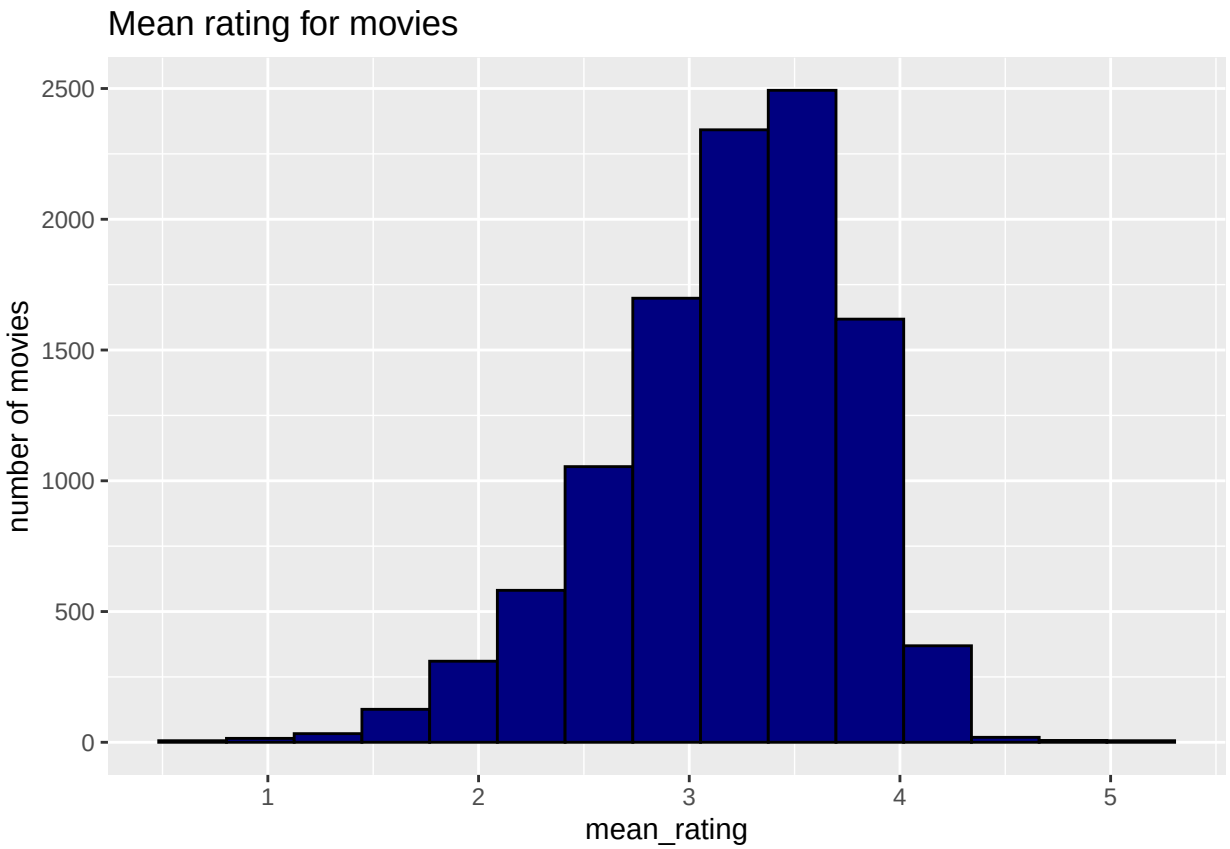


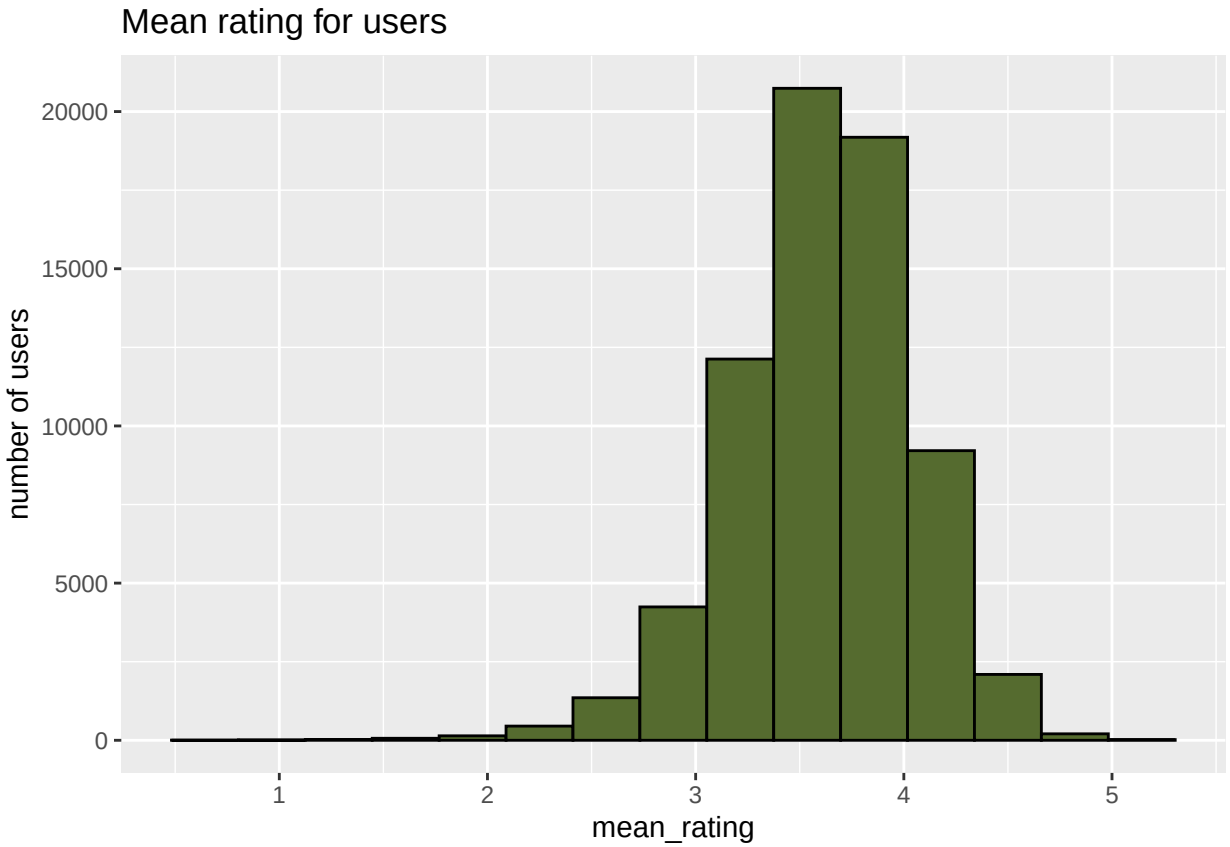
From the plot above, we observe that some movies get rated more than others. This could happen because some movies are well known and are watched by many people, whereas others are not and not many people watch them.



From this plot, we see that some users rate more movies than others, are more active in rating.

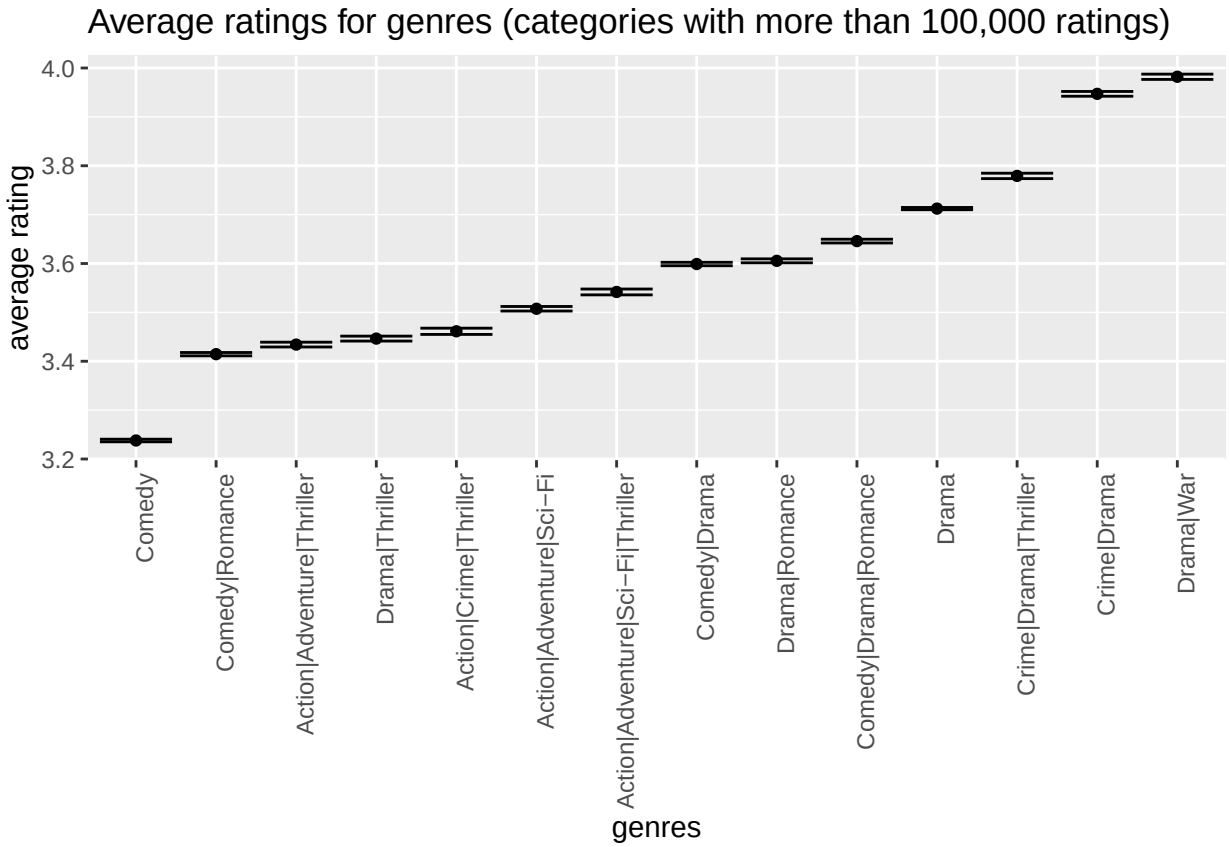
We will continue with plotting the mean rating for movies and for users


Mean rating for movies
number of movies
2500
2000
1500
1000
500
0
1
2
3
4
5
mean_rating



From the above plot, we deduce that some movies are rated higher than others and also, some users tend to give higher ratings and others are stricter. From all the above, we can clearly see that there are movies (movieId) and users (userId) effects (bias) in our ratings. As a result, we will have to take them into account when training our algorithm.

Genres variable Let us now explore the genres variable. This column includes every genre that a movie may fall under and some movies fall under several genres. For making a simpler visualization, we will make the following plot which includes the categories with more than 100,000 ratings.

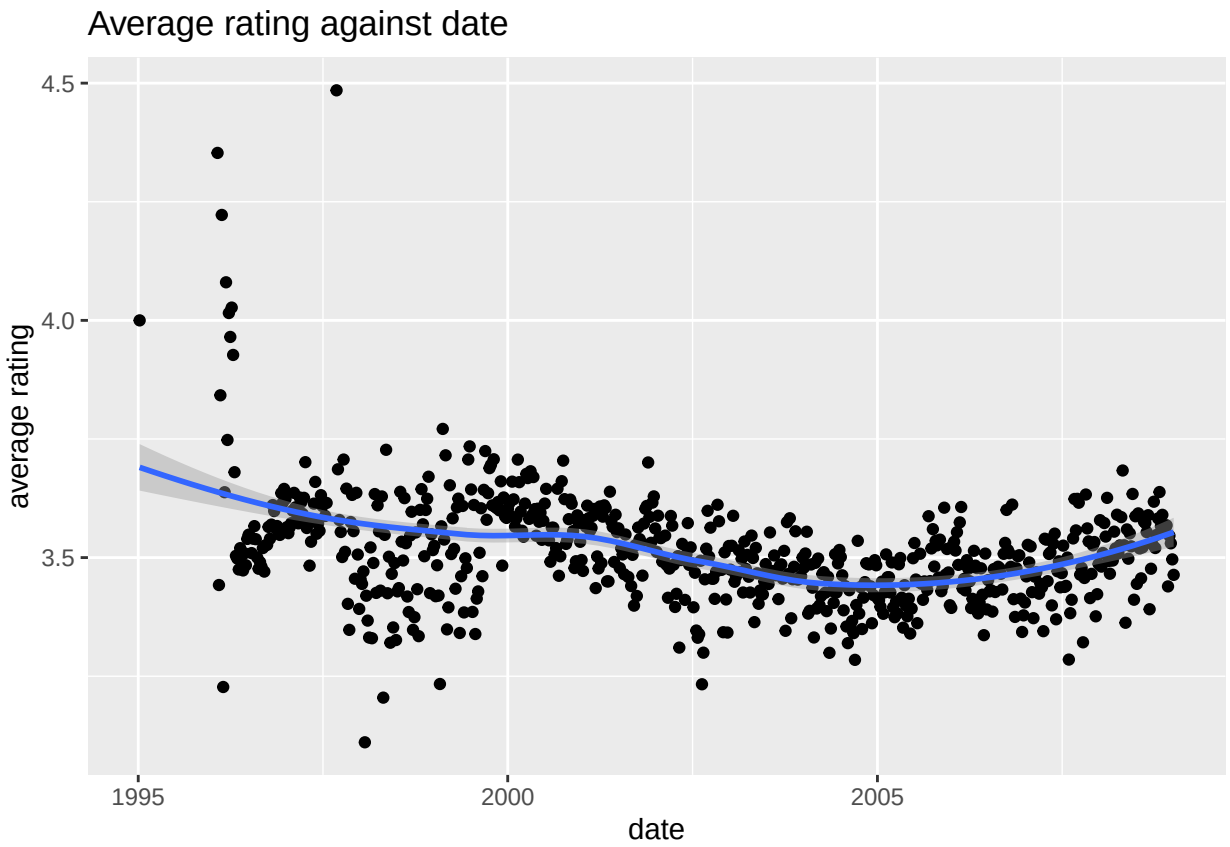


From the plot above, we obtain a strong effect of the genres variable over the ratings.Thus, we may have to consider it, when training our algorithm.

Timestamp variable The edx dataset also includes the variable timestamp. This variable represents the time and date in which the rating was provided. We will create a new column with the date rounding it to the nearest week and without taking into consideration the time.Then, we will compute the average rating for each week and plot this average against date.

```
edx<-edx %>% mutate(date = round_date(as_datetime(timestamp), unit = "week"))
```

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



From the plot, we can observe that we have a time effect on average rating however, not as strong as in the cases of movies, users and genres.

Title variable Finally, we have the title variable in which we see that there is the movie name and the year of the movie release

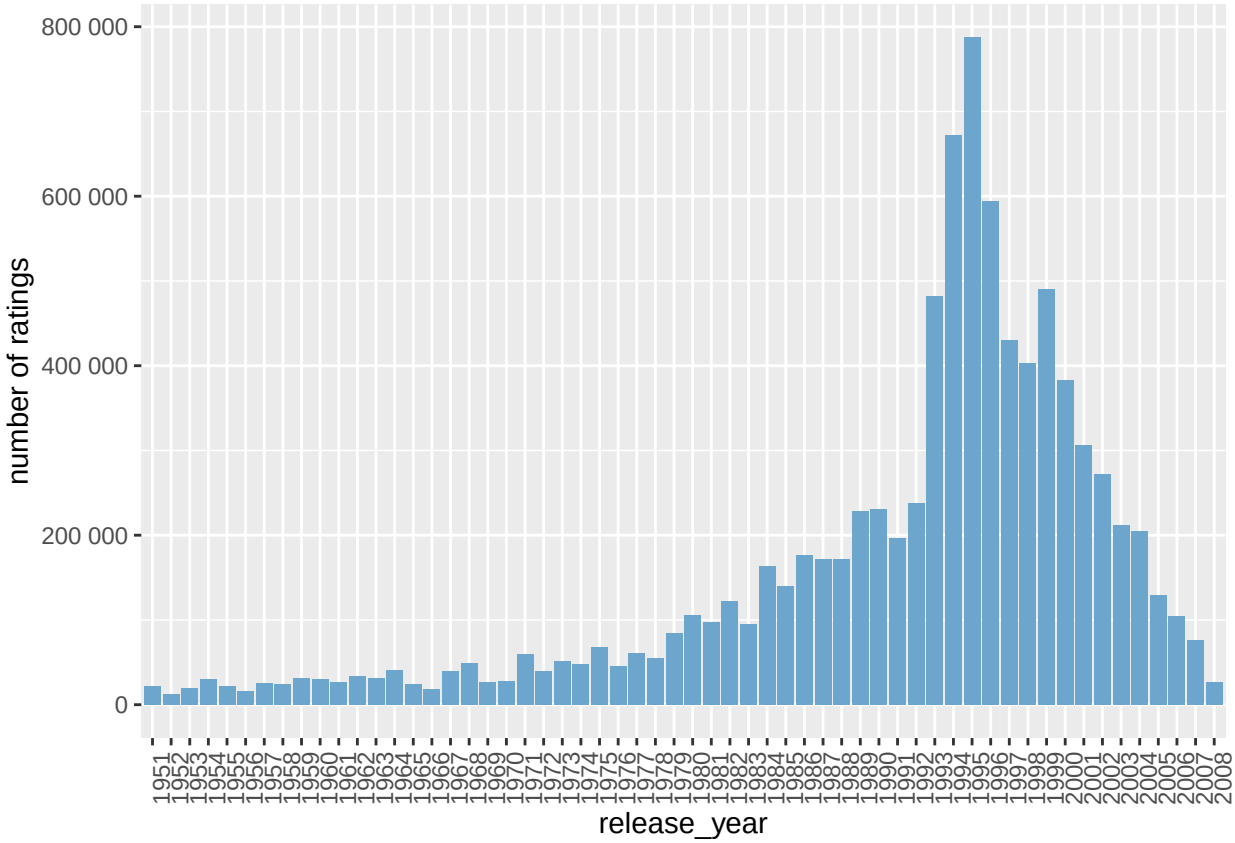
```
edx$title%>%head()
```

```
## [1] "Boomerang (1992)"              "Net, The (1995)"
## [3] "Outbreak (1995)"               "Stargate (1994)"
## [5] "Star Trek: Generations (1994)" "Flintstones, The (1994)"
```

We are going to split this column into two, the one with the title and the other with the year of the release.

```
edx<-edx%>%
  extract(title, c("title", "release_year"), regex = "^(.*) \\(([0-9]*)\\)$")
edx$release_year<-as.numeric(edx$release_year)
```

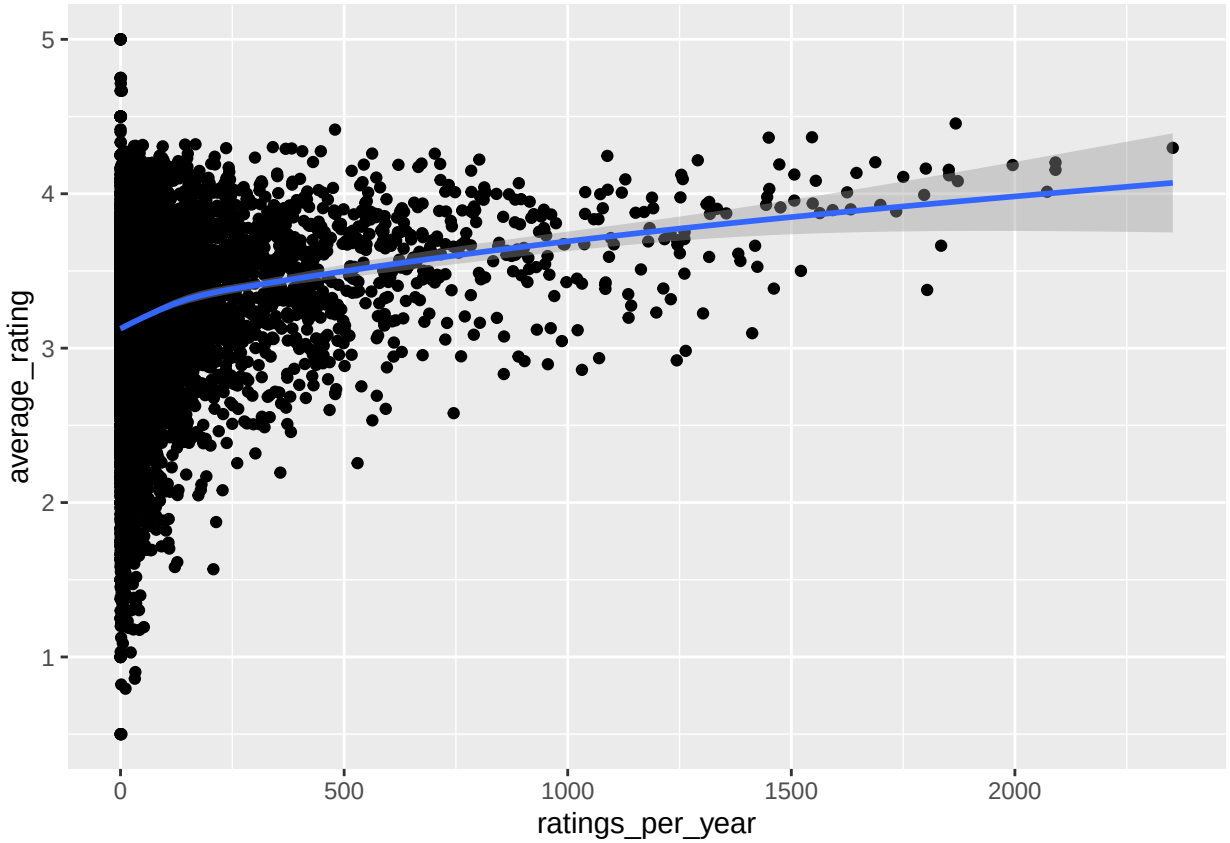
We will now check if the release year affects the ratings. First of all, we will plot the number of ratings for each movie against the release year of the movie. For a more clear visualization of the plot, we will plot the release_year after 1950 since until this year the number of ratings was quite small.



From the plot we observe that, on average, movies with release year after 1993 have been rated more. We also see that the more recent a movie is, the less time the users have to rate it.

Let us now plot the average rating against the ratings per year. We will use 2009 as the end year in order to compute the number of ratings per year.

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



From the above plot, we deduce that the movies that are rated more often tend to have above average ratings. This happens because popular movies are watched by more people.

Therefore, from the previous plots, we conclude that the release year also affects the rating.

Modeling approach

In this section, we will start to build our model. Basically, we will build different models and we will pick the one with the best predictive ability. One way to test how "good" is each model's predictive ability is to compare the differences between the values predicted by the model and the values observed. The root mean square error (RMSE) is a measure that helps us in doing the above. The less the RMSE value, the better our model will be. A general way to define it is:

$$RMSE = sqrt((\sum_{i=1}^{N}(Predicted_i - Actual_i)^2)/N)$$

or in R we can write the following function to compute it:

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

As we build our algorithm later on, we would like to check the RMSE values for each model that we will create till we get to the preferred one. In this way we can choose the best model for our recommendation system. However, because the validation set will be kept as the final hold-out test set, we will proceed with splitting the edx set to a test set and a training set. In this way, we will be able to experiment with different

parameters, test them to the test set and get to our final model. For this reason, we are going to split the edx dataset into the train set and the test set. Since we have a lot of observations, we will split our dataset to a 80:20 ratio, the training and the test set respectively.

```
set.seed(755, sample.kind="Rounding")
```

```
## Warning in set.seed(755, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
test_ind <- createDataPartition(y = edx$rating, times = 1, p = 0.2, list = FALSE)
train_set <- edx[-test_ind,]
test_temp <- edx[test_ind,]
```

We will again make sure that the users (userId) and the movies (movieId) in the test set are also in the train set

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

We add the rows removed from test_set back into the train_set

```
removed_1 <- anti_join(test_temp, test_set)
```

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

```
train_set <- rbind(train_set, removed_1)
```

and finally, we remove the temporary files

```
rm(test_ind, test_temp, removed_1)
```

Simplest model Let us now start constructing our model to predict ratings, taking into consideration the variables which have an effect on rating, as we showed in the data exploration.

The simplest model would be the one that assumes the same rating for all movies (i) and users (u) with all the differences explained by random variation. We could write the above as:

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

and we will compute it like this

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

```
## [1] 3.512523
```

Therefore, if we predict all unknown ratings with mu_hat we get the following RMSE:

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

```
## [1] 1.060561
```

The RMSE value that we get is quite high. Let us keeping finding a better model.

Movies effect In the data exploration we have seen that there is a movie effect. The rating is affected by the movieId variable. Hence, We can re-write the previous model by adding the term b_i to represent the movie i effect:

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

Because there are thousands of b_i as each movie gets one, the best way to compute them in terms of time efficiency is by taking into account that the least squares estimate $\hat{b}_i$ is just the average of

$$y_{u,i} - \hat{\mu}$$

for each movie i. Therefore,

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

Let us now compute our predicted ratings and the RMSE:

```
predicted_ratings_movie <- mu_hat + test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  pull(b_i)
```

```
rmse_movie<-RMSE(predicted_ratings_movie, test_set$rating)
rmse_movie
```

```
## [1] 0.9439868
```

We get a RMSE value already much better than before.

Users effect Let us now further improve our model by using the user effect which we have shown earlier that affects the ratings. As previously, we can re-write our model by adding the term b_u to represent the user u effect.

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

As previously, for reasons of time efficiency we compute the least squares estimate $\hat{b}_u$ as the average of

$$y_{u,i} - \hat{\mu} - \hat{b}_i$$

.

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

We can now calculate the predictors and see how much the RMSE improves:

```
  predicted_ratings_user <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  mutate(pred = mu_hat + b_i + b_u) %>%
  pull(pred)
```

```
rmse_user<-RMSE(predicted_ratings_user, test_set$rating)
rmse_user
```

```
## [1] 0.8666408
```

We get a further improvement of the RMSE value.

Release year effect We will continue taking into account the release year effect which we have seen that also affects the ratings. We can write the above model as

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

with b_y denoting the release year effect. We will again compute the least squares estimates of the year release $\hat{b}_y$ as the average that results from the above equation, that is as the average of

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

.

```
year_avgs <- train_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  group_by(release_year) %>%
  summarise(b_y = mean(rating - mu_hat - b_i - b_u))
```

Now we will calculate the predicted ratings taking into account the users, movies and year release effect.

```
predicted_ratings_year <- test_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  left_join(year_avgs, by = "release_year") %>%
  mutate(pred = mu_hat + b_i + b_u + b_y) %>%
  pull(pred)
```

Finally we will calculate the RMSE based on release year model

```
rmse_year <- RMSE(predicted_ratings_year, test_set$rating)
rmse_year
```

```
## [1] 0.8662943
```

We further improve the RMSE value. However, we do not observe big changes in the RMSE's values. Therefore, since we would like to minimize it more, we will use the regularization technique.

Regularization The regularization is a technique that we use in order to limit the variability of the effect sizes by penalizing large estimates that are formed from small sample sizes. The variability can largely increase due to noisy estimates and hence, with the regularization technique, we can prevent overfitting on the dataset. In order to control the total variability of the effects that we have, we use the penalized least squares. Basically, in our least squares equation, we add a penalty and then, we try to minimize the equation with the penalty. The penalty λ is a tuning parameter that we can use cross-validation to compute it.

To clarify all the above, in the case of the movie effects, instead of minimizing the least squares equation, we minimize this equation:

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

where the first term is the mean squared error and the second term is the penalty that gets larger when many b_i are large. The values of b_i that minimize the above equation are given by:

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

where n_i is the number of ratings for movie i. We observe that when our sample size n_i is large, then the penalty effect λ can be ignored since $n_i + \lambda \approx n_i$. On the other hand, when n_i is small, the estimate $\hat{b_i}(\lambda)$ is shrunken towards 0. Specifically, the larger the parameter λ, the more we shrink towards zero.

As we used regularization for the movie effects, we can use it for the user effects. Hence, we can have the following equation:

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

Let us now put all these in practice. We will use the regularization technique for the movie, users and release year effect. First of all, We will have to pick the best λ using cross_validation in order to achieve regularization. The cross-validation should be done just on the train_set as mentioned in Rafael A. Irizarry's book Introduction to Data Science, Machine Learning part, Large Datasets, Regularization "in practice we should be using full cross-validation just on the train set, without using the test set until the final assessment. The test set should never be used for tuning." We can do that as follows:

```
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
  mu_hat <- mean(train_set$rating)

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

  b_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu_hat)/(n()+l))

  b_y <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    group_by(release_year) %>%
    summarise(b_y = sum(rating - b_i - b_u - mu_hat)/(n()+l))
```

```
  ratings <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(b_y, by="release_year") %>%
    mutate(pred = mu_hat + b_i + b_u + b_y) %>%
    pull(pred)

  return(RMSE(ratings, train_set$rating))
})
```

We get the values of the best λ below

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

```
## [1] 0.25
```

After tuning our parameter λ, we will use the best_lambda in order to calculate the predicted ratings

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

b_i <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu_hat)/(n()+best_lambda))

b_u <- train_set %>%
  left_join(b_i, by="movieId") %>%
  group_by(userId) %>%
  summarize(b_u = sum(rating - b_i - mu_hat)/(n()+best_lambda))

b_y <- train_set %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  group_by(release_year) %>%
  summarise(b_y = sum(rating - b_i - b_u - mu_hat)/(n()+best_lambda))


predicted_ratings_regularization <- test_set %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  left_join(b_y, by="release_year") %>%
  mutate(pred = mu_hat + b_i + b_u + b_y) %>%
  pull(pred)
```

Finally we will calculate the RMSE based on the regularization model

```
rmse_regularization <- RMSE(predicted_ratings_regularization, test_set$rating)
rmse_regularization
```

```
## [1] 0.8662047
```

We notice a further but slight improvement in the RMSE value.

Recosystem package We are now going to build a model using the recosystem package. With this package, we build a recommender system using matrix factorization. Matrix factorization is a concept that covers a wide range of applications in machine learning. The basic idea of matrix factorization is that if we convert the data into a matrix so that each user gets a row, each movie gets a column, and the rating is the entry in row u and column i, then we have to find two matrices which describe the original matrix. Basically, it decomposes the user-movie matrix into the product of two matrices of lower dimensions, as follows:

$$R_{mxn} \approx U_{mxk} \cdot V_{nxk}^T$$

where R is the rating matrix of m rows/users and n columns/movies, U has a row for each user,is a user-factor matrix and V^T has a column for each movie, is a movie-factor matrix.

More detailed information about matrix factorization and the technique of singular value decomposition (SVD) that is used, can be found in [Introduction to Data Science by Rafael A. Irizarry, Machine Learning Part, Large Datasets] (http://rafalab.dfci.harvard.edu/dsbook/large-datasets.html#matrix-factorization).

More detailed information about the recosystem pacakage can be found in recosystem: Recommender System Using Parallel Matrix Factorization and in recosystem: Recommender System using Matrix Factorization.

The great advantage of using this package is that, except for the number of user-friendly R functions able to simplify data processing and model building, is capable of reducing memory use. Most other R packages for statistical modeling, store the whole dataset and model object in memory. However, recosystem can store in the hard disk the constructed model, and moreover, the output result can also be directly written into a file rather than be kept in memory.

Let us now, start constructing the model. According to the recosystem package, the data format of the train_set should be in the form of sparse matrix triplet, i.e., each line in the file contains three numbers: user_index, movie_index, rating. Therefore, from our train_set, we will keep only the variables userId, movieId and rating and we will convert this to a matrix.

```
train_set_reco <-  train_set %>%
            select(c("userId","movieId","rating"))

names(train_set_reco) <- c("user", "movie", "rating")

train_set_reco <- as.matrix(train_set_reco)
```

The test_set has the same data format as the train_set. The only difference is that the rating variable can be omitted since the ratings in testing data are usually unknown. However, even if we keep it, the system will ignore it. Hence, as above, we create a matrix.

```
test_set_reco <-  test_set %>%
            select(c("userId","movieId","rating"))

names(test_set_reco) <- c("user", "movie", "rating")

test_set_reco <- as.matrix(test_set_reco)
```

We will continue using the data_memory() function in order to specify the source of the data in the recommender system, which in this case, are data in the memory as R objects. The first and second argument in data_memory() must be integer vectors giving the user and movie indices respectively of rating scores. We also put the argument index1 = TRUE because in our data the user and movie indices start with 1.

```
set.seed(123,sample.kind="Rounding")
```

```
## Warning in set.seed(123, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
train_set_reco1 <- data_memory(as.integer(train_set_reco[,1]),
                               as.integer(train_set_reco[,2]),
                               train_set_reco[,3],index1 = TRUE)
test_set_reco1 <- data_memory(as.integer(test_set_reco[,1]),
                              as.integer(test_set_reco[,2]),
                              rating=NULL, index1 = TRUE)
```

Subsequently, we create a model object by calling Reco() and we call the tune() method that uses cross validation in order to select the best tuning parameters. The tune() method needs some time to run (almost 15 minutes in my laptop with specs: i7 + 32GB RAM).

```
r = Reco()

opts = r$tune(train_set_reco1, opts = list(dim = c(10, 20, 30), lrate = c(0.1, 0.2),
                                           costp_l1 = 0, costq_l1 = 0,
                                           nthread = 1, niter = 10))
```

Following, we will use the train() method in order to train our recommender model. We will set inside the function a number of parameters, coming from the result of tune().

```
r$train(train_set_reco1, opts = c(opts$min, nthread = 1, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9943   1.0055e+07
##    1       0.8787   8.0797e+06
##    2       0.8471   7.5087e+06
##    3       0.8253   7.1593e+06
##    4       0.8078   6.9043e+06
##    5       0.7943   6.7194e+06
##    6       0.7834   6.5798e+06
##    7       0.7741   6.4681e+06
##    8       0.7661   6.3759e+06
##    9       0.7591   6.2977e+06
##   10       0.7529   6.2303e+06
##   11       0.7474   6.1735e+06
##   12       0.7424   6.1261e+06
##   13       0.7378   6.0813e+06
##   14       0.7338   6.0438e+06
##   15       0.7300   6.0095e+06
##   16       0.7266   5.9787e+06
##   17       0.7234   5.9507e+06
##   18       0.7205   5.9251e+06
##   19       0.7179   5.9033e+06
```

After training our model, we are going to use the predict() method to compute the predicted values in our test_set_reco1

```
ratings_pred <- r$predict(test_set_reco1, out_memory())
```

Finally, let us find the RMSE

```
rmse_recosystem <- RMSE(ratings_pred, test_set_reco[,3])
rmse_recosystem
```

```
## [1] 0.7910615
```

We observe that this method gave us the smallest RMSE value.

Results

In this section, the modeling results are going to be presented. We will create a results table

```
rmse_results <- tibble(method = c("Just the average","Movie effect", "Movie & User effects",
                                  "Movie, User $ Release year effects",
                                  "Movie, User $ Release year effects with Regularization",
                                  "Recosystem Package"),
                       RMSE = c(rmse_simple, rmse_movie, rmse_user, rmse_year,
                                rmse_regularization, rmse_recosystem))

knitr::kable(rmse_results,digits=5)
```

method	RMSE
Just the average	1.06056
Movie effect	0.94399
Movie & User effects	0.86664
Movie, User $ Release year effects	0.86629
Movie, User $ Release year effects with Regularization	0.86620
Recosystem Package	0.79106

As we notice, the best RMSE value was achieved with the recosystem package, hence with using Matrix Factorization, which minimized the most the root mean square error. Therefore, we will choose this method and we will check our model performance taking into consideration the edx set and our final hold-out test set that means the validation set. Therefore, let us run the recosystem package as previously mentioned but for the edx and validation set.

```
edx_reco <-  edx %>%
            select(c("userId","movieId","rating"))

names(edx_reco) <- c("user", "movie", "rating")

edx_reco <- as.matrix(edx_reco)
```

```
validation_reco <-  validation %>%
            select(c("userId","movieId","rating"))

names(validation_reco) <- c("user", "movie", "rating")

validation_reco <- as.matrix(validation_reco)
```

We will continue using the data_memory() function.

```
set.seed(123,sample.kind="Rounding")
```

```
## Warning in set.seed(123, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
edx_reco1 <- data_memory(as.integer(edx_reco[,1]),
                                as.integer(edx_reco[,2]),
                                edx_reco[,3],index1 = TRUE)
validation_reco1 <- data_memory(as.integer(validation_reco[,1]),
                               as.integer(validation_reco[,2]),
                                rating=NULL, index1 = TRUE)
```

Subsequently, we create a model object by calling Reco() and we call the tune() method. The tune() method needs some time to run (almost 20 minutes in my laptop with specs: i7 + 32GB RAM).

```
r = Reco()

opts = r$tune(edx_reco1, opts = list(dim = c(10, 20, 30), lrate = c(0.1, 0.2),
                                     costp_l1 = 0, costq_l1 = 0,
                                     nthread = 1, niter = 10))
```

Following, we will use the train() method in order to train our recommender model.

```
r$train(edx_reco1, opts = c(opts$min, nthread = 1, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9731   1.2038e+07
##    1       0.8725   9.8817e+06
##    2       0.8396   9.1838e+06
##    3       0.8173   8.7588e+06
##    4       0.8012   8.4727e+06
##    5       0.7890   8.2705e+06
##    6       0.7792   8.1190e+06
##    7       0.7712   8.0004e+06
##    8       0.7644   7.9040e+06
##    9       0.7585   7.8237e+06
##   10       0.7533   7.7551e+06
##   11       0.7487   7.6963e+06
##   12       0.7446   7.6485e+06
##   13       0.7409   7.6041e+06
##   14       0.7375   7.5659e+06
##   15       0.7344   7.5322e+06
##   16       0.7315   7.5016e+06
##   17       0.7290   7.4735e+06
##   18       0.7265   7.4478e+06
##   19       0.7242   7.4256e+06
```

After training our model, we are going to use the predict() method to compute the predicted values in our validation_reco1

```
ratings_pred_final <- r$predict(validation_reco1, out_memory())
```

Finally, let us find the RMSE

```
rmse_final <- RMSE(ratings_pred_final, validation_reco[,3])
rmse_final
```

```
## [1] 0.7827192
```

We get an RMSE < 0.86490. Our model performance is quite good since we have minimized our error, hence our predictions using this model can be more accurate and trustworthy.

Conclusion

In this project, we have explored our MovieLens 10M Dataset from grouplens, we got useful insights from this exploration and based on this, we have used different modeling approaches for creating a movie recommendation system. We concluded that the best modeling approach and therefore, the potential best movie recommendation system that could best predict the movie ratings is the one created by the recosystem package, that is with matrix factorization. The truth is that our modeling approaches were more simplistic, trying to give a general idea of how movies and users affect the final ratings. There are a lot more biases/effects that we could take into account and thus, further minimize our root mean square error. For example, one more effect could be that someone comes to like romance movies more and more over time or that someone can become a stricter critic over time. Furthermore, there are more techniques that we could use to create our movie recommendation system such as the recommenderlab package or the ensemble method in which different algorithms could be combined to provide a single rating taking into consideration the "best characteristics" of each model.

References

Introduction to Data Science, Rafael A. Irizarry,

recosystem: Recommender System using Matrix Factorization,

recosystem: Recommender System Using Parallel Matrix Factorization,

Package 'recosystem',

Winning the Netflix Prize: A Summary