MovieLens Project

Beatriz Estrella

10/11/2020

R Markdown

1. Introduction

1.1. Project goal

The goal of this project is to use different Machine Learning algorithms to try to predict the rating that an user will give to a movie. To achieve this, we will use the Machine Learning models and statistics that we have learnt during the Data Science course and we will finally choose the one that gets the minimum RMSE number. RMSE is the Root Mean Square Error and it is the standard deviation of the residuals. To calculate the RMSE we create a function as:

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

1.2. Inspecting the dataset

First of all we are going to explain the dataset that is going to be used for this project. The dataset is allocated in http://files.grouplens.org/datasets/movielens/ml-10m.zip and has been pre-processed to achieve the movielens dataset. From the movielens dataset, 10% of the data will be used for the final validation and it is stored in the validation dataframe variable. This validation set will not be used in the calculations or algorithm in the project as it is reserved only for the final grading of the exercise.

The dataset we will work with will be stored in the edx dataframe variable and consists of the following characteristics:
· 9000055 rows
· 6 columns
The predictors are (5 predictors):
· $userId: integer variable with data from 69878 different users
· $movieId: numeric variable with data from 10677 different movies
· $timestamp: integer variable that will be converted to date variable with a name column “date”
· $title: character containing the title and year of the movie
· $genres: character variable containing the different genres of the movie, separated by |

The variable we want to predict is:
· $rating: numeric variable that goes from 0.5 (there are no zeros in the dataset) to 5 (maximum possible rating for a movie). Also, we have checked is there is NA is the rating variable that we should manage but all of the data contains a numeric value.

The validation set consits of 999999 and 6 columns.

As well, we make sure there are no N/As in the ratings column:

```r
nas <- sum(is.na(edx$rating)) #no NAs
nas
```

```
## [1] 0
```

1.3. Libraries

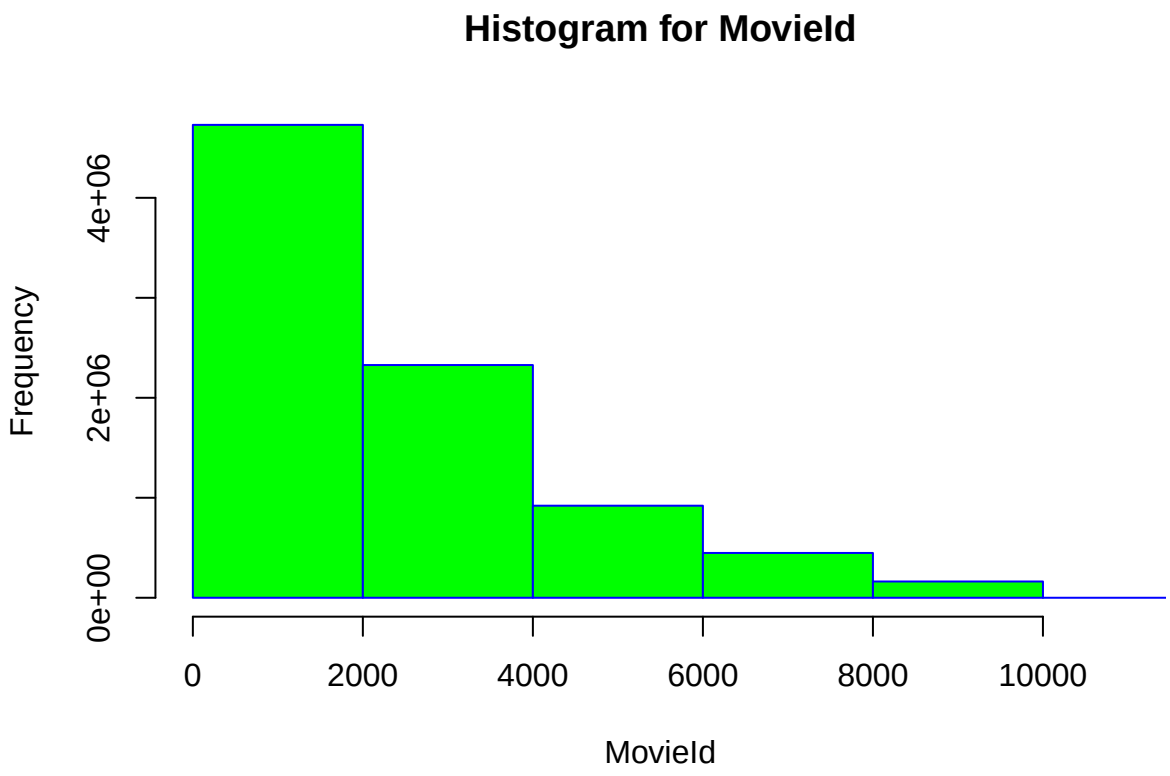
Besides the tidyverse and caret package, we will also load:
· lubridate <- it allows us to work with dates
· ggplot2 <- it allows us to create different plots
· stringr <- it allows to work with strings and regex synthax

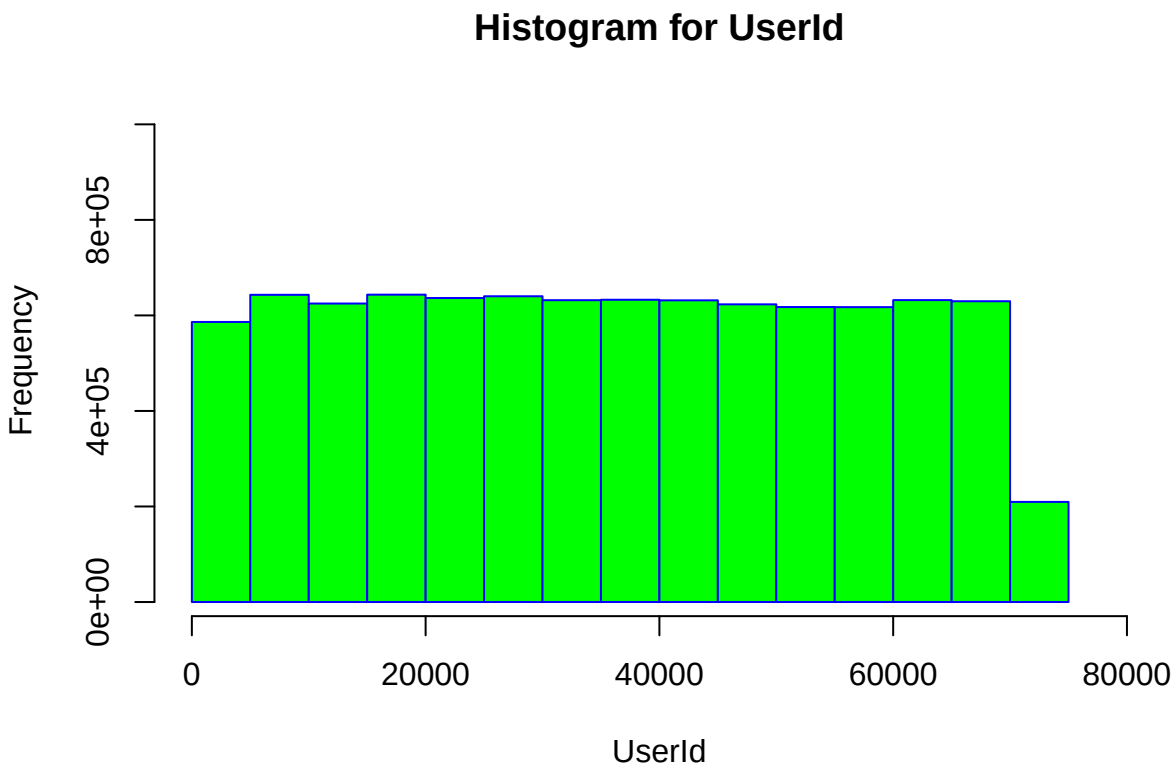
2. Method and Analysis

2.1. Exploration of the edx dataset

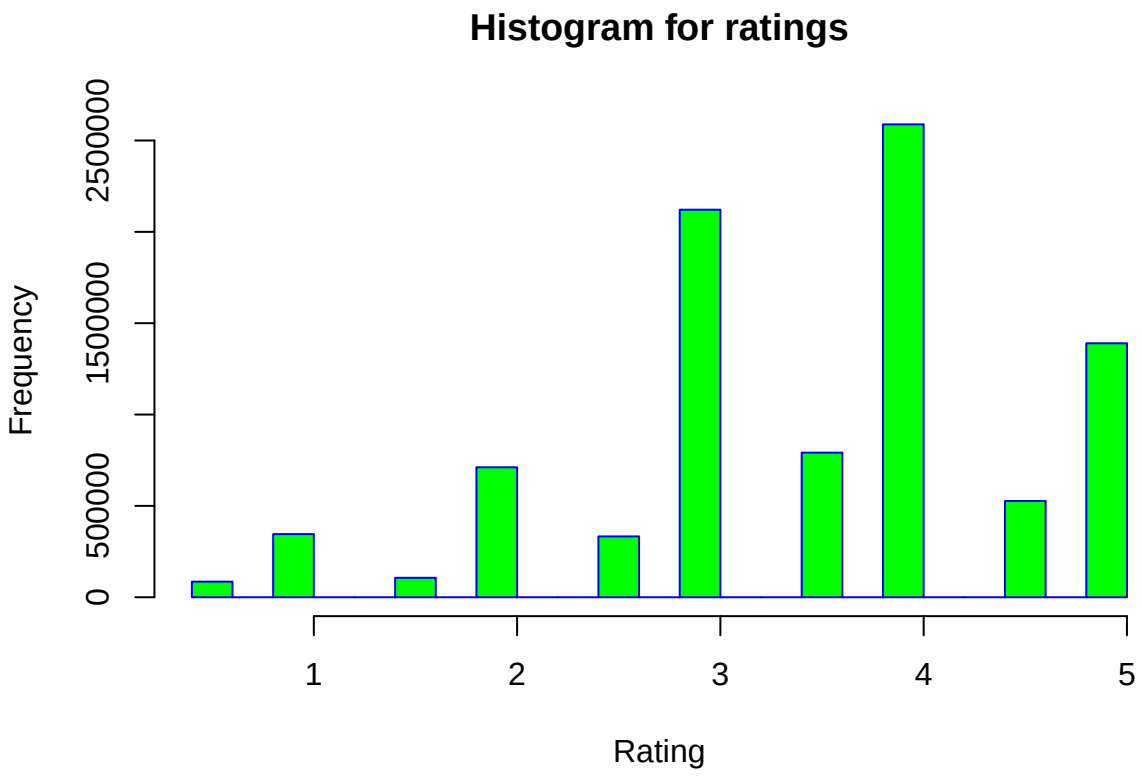
To first understand the data we are going to work with, we first plot two histograms of the movieId and userId.

We see here that some movies have been rated much more than other movies. However, the frequency of ratings for each user is very stable.





As well, we can see by ploting an histogram of the ratings, that integers are much more used that those with a decimal number.



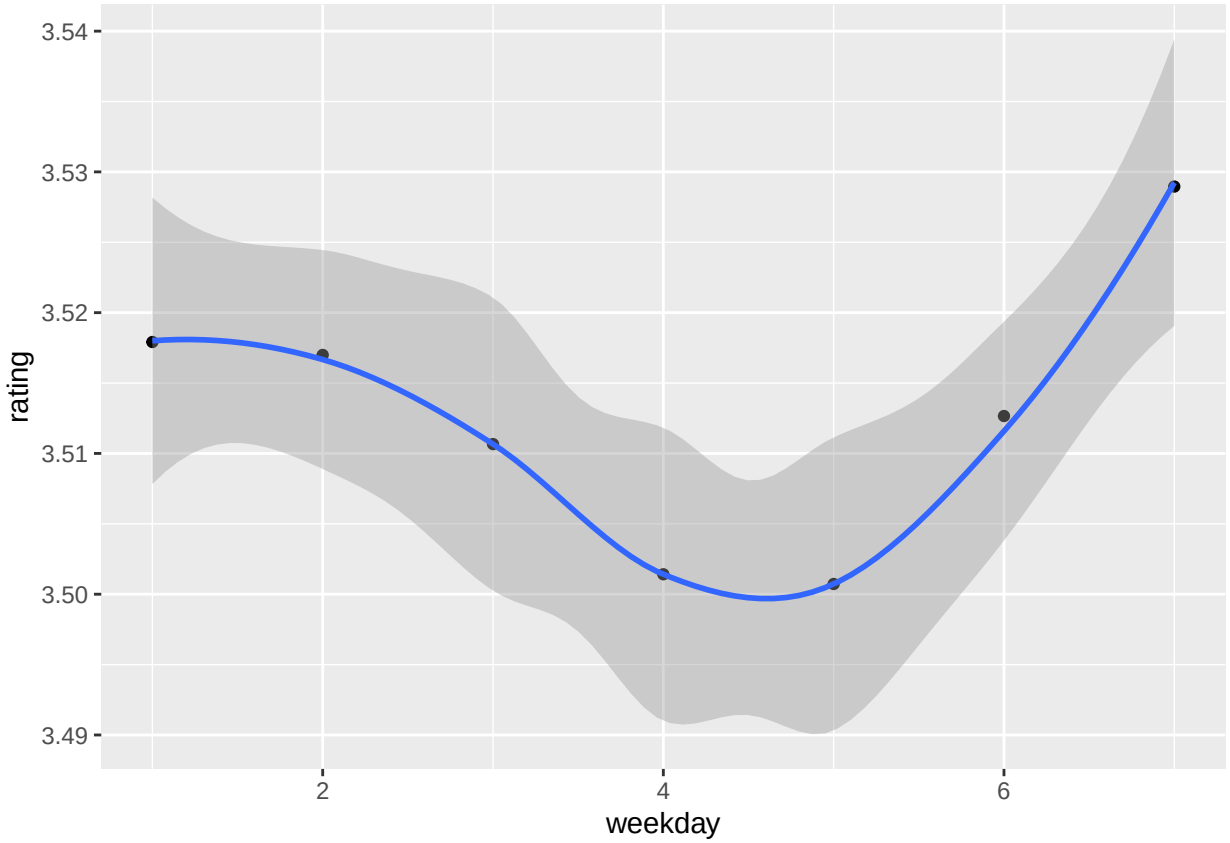
We use lubridate package to first, convert timestamp variable to a date variable and store in date column, and second, to check whether weekday, month or year influences rating, we create these new columns.

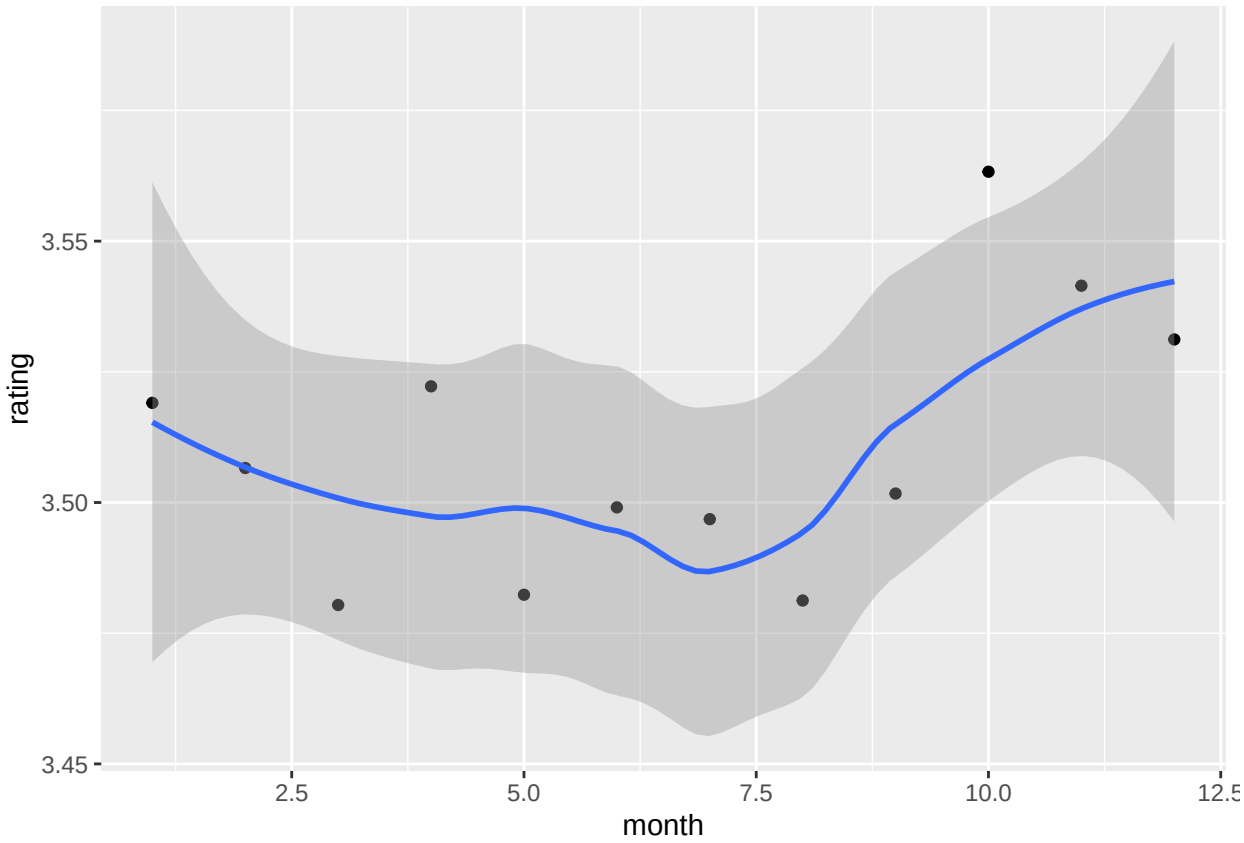
```
edx <- edx %>%
  mutate(date=as_datetime(timestamp), weekday = wday(date), month=month(date),   year=year(date)) %>%
  select(userId, movieId, rating, title, genres, weekday, month, year, date)
```

2.2. Checking predictors effect

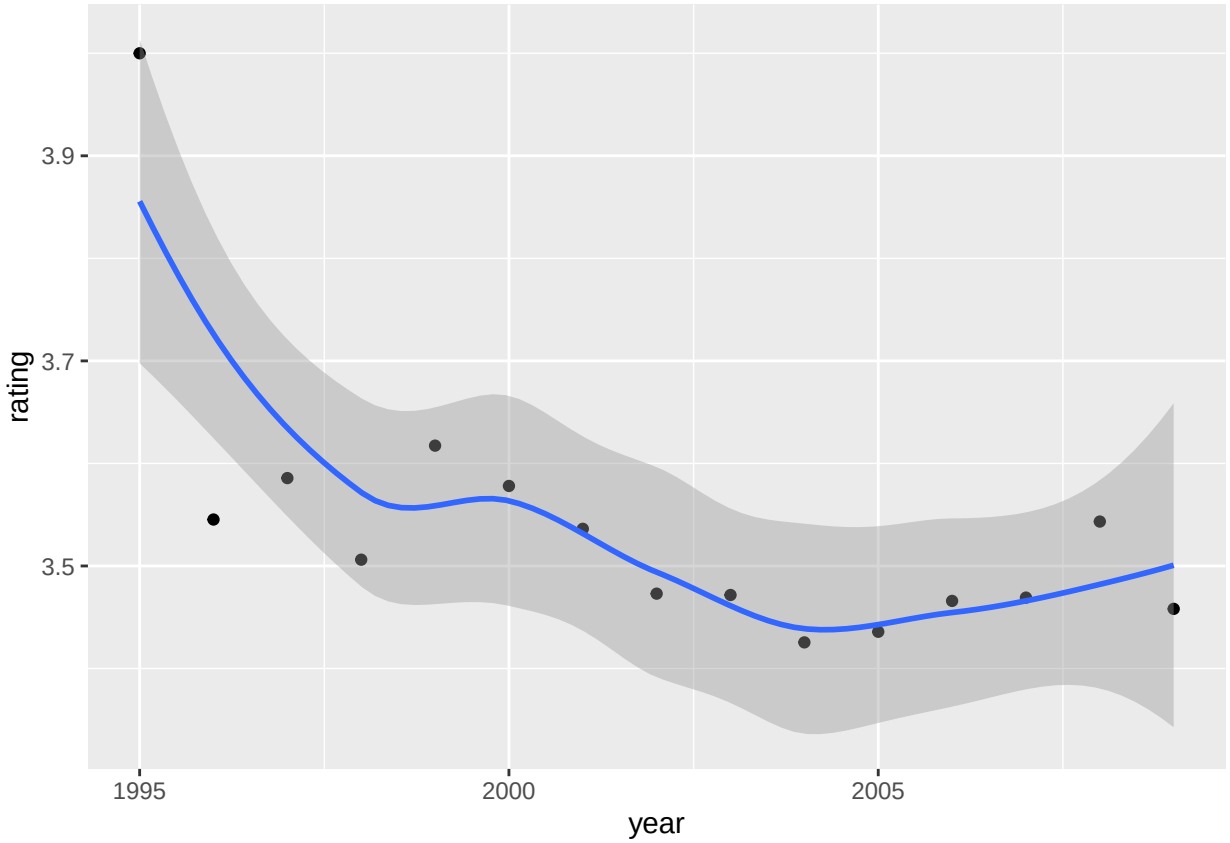
2.2.1. Checking timestamp effect

When plotting average ratings vs weekday, month, and year, we can see that the effect of weekday is minimum (slightly lower on Wednesdays and Thursdays), and that there is no relevant effect due to month.


3.54
3.53
3.52
3.51
3.50
3.49
rating
2
4
6
weekday



On the other hand, we can see a small effect due to year, movies being rated previous to 2000 usually perform higher than those rated in the last years. Nevertheless, this effect will not be taken into consideration in our model.

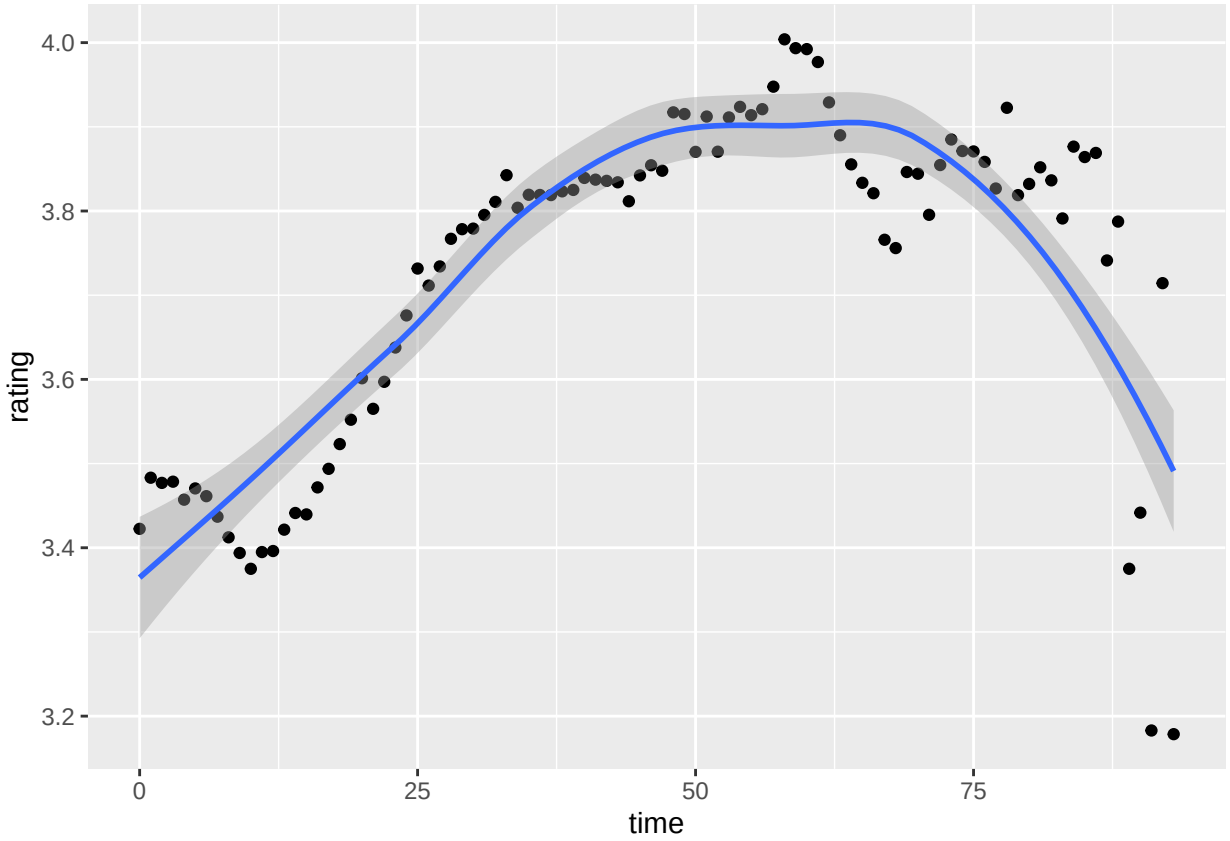


Besides this effect, we are going to add a new column named "time" that will measure years from release date to the user giving the rating. We add this new variable to include the effect of a user being a fan of a movie, so time will be low vs a user seeing a movie long time after release date, which will show probably that he/she is not a big fan. Also to include the effect of a movie being rated years later release date, which probably is due to being a better than average movie or a specific user preference.

Firstly, we need to separate the release year from the title column to a new separate column, we show the code used for this below:

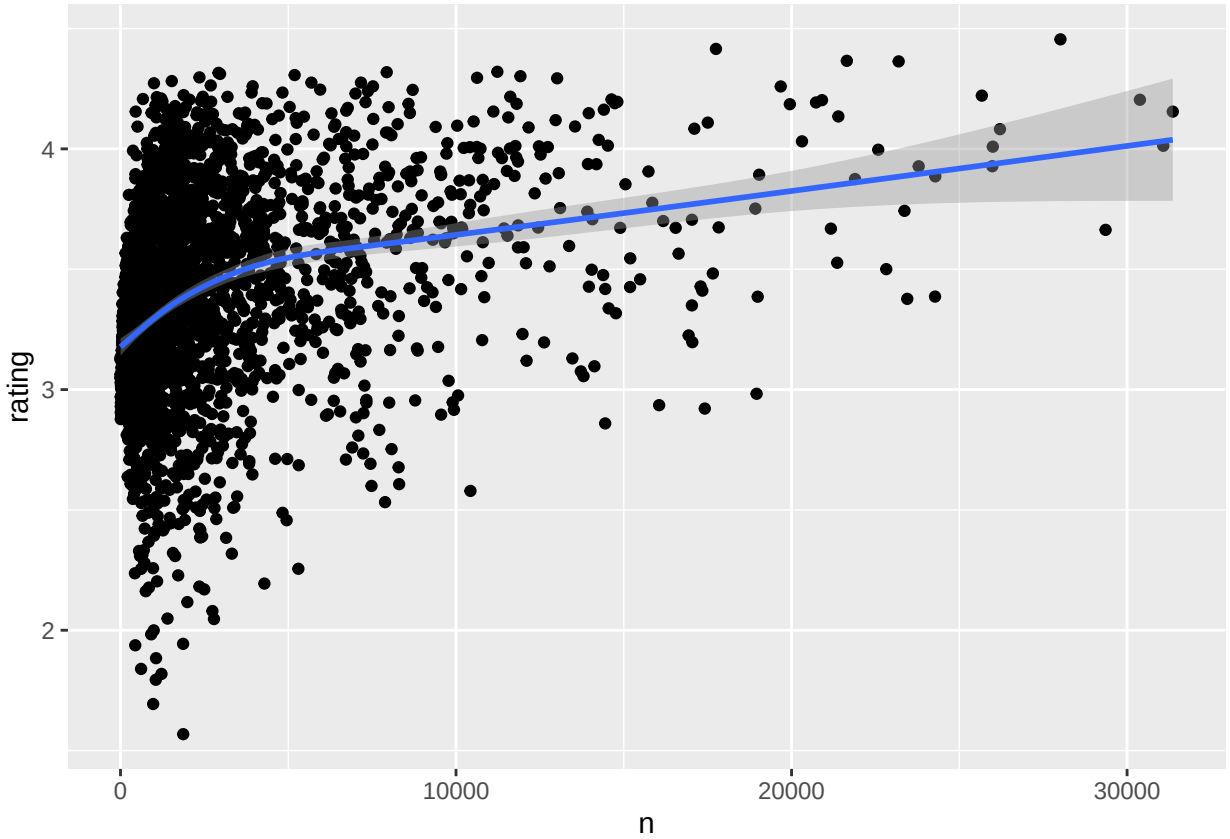
```
pattern <- "\\(\\d{4}\\)"
edx <- edx %>%
  mutate(temp_title=str_extract(title, pattern)) %>%
  #extract string starting with a ( and followed by 4 digits, keep in temp_title
  mutate(year_release=as.numeric(str_extract(temp_title, "\\d{4}"))) %>%
  #from temp_title we extract the 4 digits (year_release) and convert to numeric
  mutate(time=year-year_release) %>%
  #time between user rated the movie and the release year
  mutate(time=ifelse(time>=0, time, 0)) %>%
  #in a few cases year of rating was prior to year_release which is impossible,
  #that is why we replace those by 0.
  select(-temp_title) #remove temp_title column
```

Once we have the new column of year_release and the column that calculates time, we create the chart below to see the effect of time in ratings:



2.2.2. Check number of ratings per movie effect

Additionally, we will add a new column named n that will show the number of ratings given to a movieId. To see why this may have an effect to consider, see plot below:



As we can see, the more ratings a movie gets, the higher the average rating. This may be due to popular and awarded movies getting more ratings than unpopular ones.

2.3. Creation of train set and test set

In order to test the results we get with the different methods, we are going to partition the edx dataset into two different dataframes. Test set will consist of 20% of edx data. We then create:
· train_set <- to be used to train different models
· test_set <- to be used to test the results of each model

2.4. Method

Due to the large amount of data contained in the edx dataset, Machine Learning algorithms such as Linear Regression, k-nearest neighbors or others included in the *caret* package cannot be used as error is shown due to lack of computer memory.

For that reason, the approach will be to include the different effects that seem relevant in a model based approach as:

$$Y\ u,i\ = \mu + b\ i\ + b\ u\ + b\ n\ + b\ t\ + \epsilon\ u,i$$

Being
· b_i: effect on the rating of the different movies
· b_u: effect on the rating of the different users
· b_n: effect on the rating of the number of ratings per movie
· b_t: effect on the rating of the time between release year and provided rating

3. Results

3.1. Baseline results

First of all we create our baseline model, which simply predicts the average rating of all movies. That variable is called *mu_hat* in the code and in the formula of the model it refers to mu.

```
mu_hat <- mean(train_set$rating)
baseline_rmse <- RMSE(test_set$rating, mu_hat)
```

We as well create a tibble to store the different RMSE results gotten from the different methods.

```
options(pillar.sigfig = 5) #tibble to show 5 significant digits
rmse_results <- tibble(method = "Just the average", RMSE = baseline_rmse)
```

```
## # A tibble: 1 x 2
##   method             RMSE
##   <chr>             <dbl>
## 1 Just the average 1.0599
```

This model provides a RMSE of 1.0599. We can for sure do better!

3.2. Include movie bias

Now we want to include the movie effect, as different movies have different average ratings. This is clearly due to some movies being simply better than others. This effect in our formula was called b_i.

To calculate b_i:

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

Once we have the movie effect b_i we apply it to our predictions, calculate the new RMSE and store it in our tibble.

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

movie_rmse <- RMSE(test_set$rating, predicted_ratings)
rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Movie Effect Model",
                                     RMSE = movie_rmse))
```

The new RMSE including the movie effect is 0.9437.

```
## # A tibble: 2 x 2
##   method                RMSE
##   <chr>                <dbl>
## 1 Just the average   1.0599
## 2 Movie Effect Model 0.94374
```

As we can see, we have already improved greatly our RMSE results from considering just the average. Let's see if we can do better.

3.3. Include user bias

Now we want to include the user bias. There is an effect of the user as some users may be more critic than others, and in average, provide lower or higher ratings to the movies. This effect in our formula was called b_u.

To calculate b_u:

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

Once we have the user effect b_u we apply it to our predictions, calculate the new RMSE and store it in our tibble.

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

user_rmse <- RMSE(test_set$rating, predicted_ratings)

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="User+Movie Effect Model",
                                     RMSE = user_rmse))
```

The new RMSE including the user effect is 0.8659.

```
## # A tibble: 3 x 2
##   method                     RMSE
##   <chr>                     <dbl>
## 1 Just the average         1.0599
## 2 Movie Effect Model       0.94374
## 3 User+Movie Effect Model 0.86593
```

Much better! Let's keep it going.

3.4. Include number of ratings effect

As we saw before, movies that have a greater number of ratings usually get a higher than average rating. With this piece of code we want to include that effect into our predictions. This effect in our formula was called b_n.

To calculate b_n:

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

Once we have the number of ratings effect b_n we apply it to our predictions, calculate the new RMSE and store it in our tibble.

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

ratings_rmse <- RMSE(test_set$rating, predicted_ratings)

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="User+Movie+#Ratings Effect Model",
                                     RMSE = ratings_rmse))
```

The new RMSE including the user effect is 0.8651.

```
## # A tibble: 4 x 2
##   method                              RMSE
##   <chr>                              <dbl>
## 1 Just the average                  1.0599
## 2 Movie Effect Model                0.94374
## 3 User+Movie Effect Model           0.86593
## 4 User+Movie+#Ratings Effect Model 0.86510
```

Great! We can see a sligh improvement.

3.5. Include time from release date to rating

Now we will include the effect that we saw that time has in the average rating given to a movie. As we discussed, this may be to a fan phenomenon, or also due to old movies only being newly rated if they already have a higher than average rating. This effect in our formula was called b_t.

To calculate b_t:

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

Once we have the time effect b_t we apply it to our predictions, calculate the new RMSE and store it in our tibble.

```
predicted_ratings <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(n_rating_avgs, by='n') %>%
  left_join(time_avgs, by='time') %>%
  mutate(pred = mu_hat + b_i + b_u + b_n + b_t) %>% pull(pred)

time_rmse <- RMSE(test_set$rating, predicted_ratings)
```

```
rmse_results <- bind_rows(rmse_results,
                          data_frame(method="User+Movie+#Ratings+Time Effect Model",
                                     RMSE = time_rmse))
```

The new RMSE including the time effect is 0.8647.

```
## # A tibble: 5 x 2
##   method                                  RMSE
##   <chr>                                  <dbl>
## 1 Just the average                      1.0599
## 2 Movie Effect Model                    0.94374
## 3 User+Movie Effect Model               0.86593
## 4 User+Movie+#Ratings Effect Model      0.86510
## 5 User+Movie+#Ratings+Time Effect Model 0.86477
```

Great! We got to being below 0.86500.

3.6. Regularization

Now we want to apply to our model regularization. Regularization penalizes large estimates that are formed using small sample sizes.

To apply regularization, the first thing we need to do is to choose the best *lambda* that minimizes RMSE.

3.6.1. Cross-validation

To do this, we will apply cross-validation with 5 different train and test set, obtained as a part of the original train set. We will check which is the *lambda* that minimizes RMSE for each of the 5 samples, and then our chosen *lambda* will be calculated as the average of the 5 best ones.

This part of the code creates 5 different data partitions from the train_set dataframe.

```
require(caret)
k <- 5
set.seed(5, sample.kind="Rounding")
flds <- createDataPartition(y=train_set$rating, times = k, p=0.2, list = FALSE)

#best_lambda vector will store the lambda that minimizes RMSE in each loop
best_lambda <- rep(NA, k)
```

Now, we create a for loop that predicts, calculates RMSE and optimizes *lambda* for each of the different train and test set created for cross-validation. We store the best lambda for each loop into the *best_lambda* vector.

```
for (i in 1:k){
flds_index <- flds[,i]
train_i <- train_set[-flds_index,]
temp_test_i <- train_set[flds_index,]
test_i <- temp_test_i %>%
  semi_join(train_i, by = "movieId") %>%
  semi_join(train_i, by = "userId")
removed_i <- anti_join(temp_test_i, test_i)
```

```
train_i <- rbind(train_i, removed_i)
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
  mu <- mean(train_i$rating)
  b_i <- train_i %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))
  b_u <- train_i %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))
  b_n <- train_i %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    group_by(n) %>%
    summarize(b_n = sum(rating - b_u - b_i - mu)/(n()+l))
  b_t <- train_i %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(b_n, by="n") %>%
    group_by(time) %>%
    summarize(b_t = sum(rating - b_n - b_u - b_i - mu)/(n()+l))
  predicted_ratings_i <-
    test_i %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(b_n, by="n") %>%
    left_join(b_t, by="time") %>%
    mutate(pred = mu + b_i + b_u + b_n + b_t) %>% pull(pred)
  return(RMSE(predicted_ratings_i,test_i$rating)) })
lambda_i <- lambdas[which.min(rmses)]
best_lambda[i] <- lambda_i
}
```

Once we have our *best_lambda* vector with the five different optimized *lambdas*, we calculate the average to choose *lambda*.

```
best_lambda
lambda <- mean(best_lambda)
```

In this case, *lambda* is 4.7.

3.6.2. Applying lambda to regularization

Now that we have the best *lambda*, we apply it to our model:

```
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") %>%
    group_by(userId) %>%
```

```
    summarize(b_u = sum(rating - b_i - mu)/(n()+lambda))
b_n <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    group_by(n) %>%
    summarize(b_n = sum(rating - b_u - b_i - mu)/(n()+lambda))
b_t <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(b_n, by="n") %>%
    group_by(time) %>%
    summarize(b_t = sum(rating - b_n - b_u - b_i - mu)/(n()+lambda))
predicted_ratings <-
    test_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(b_n, by="n") %>%
    left_join(b_t, by="time") %>%
    mutate(pred = mu + b_i + b_u + b_n + b_t) %>% pull(pred)
reg_rmse <- RMSE(predicted_ratings, test_set$rating)

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Regularized Total Effect Model",
                                     RMSE = reg_rmse))
```

The new RMSE including regularization is 0.8641.

```
## # A tibble: 6 x 2
##   method                                  RMSE
##   <chr>                                  <dbl>
## 1 Just the average                       1.0599
## 2 Movie Effect Model                     0.94374
## 3 User+Movie Effect Model                0.86593
## 4 User+Movie+#Ratings Effect Model       0.86510
## 5 User+Movie+#Ratings+Time Effect Model 0.86477
## 6 Regularized Total Effect Model         0.86419
```

Awesome! We could improve our RMSE results.

3.7. Final Model

3.7.1. Capping maximum and minimum predicted ratings

We know from our database that the maximum rating that a user can give is 5, but we can see that we are predicting ratings higher than 5. We will substitute any predicted rating over 5 to 5.

We will do the same with the minimum, which in this case is 0.5. Any predicted ratings below 0.5 will be substituted to 0.5.

3.7.2. Final Model

With that slight adjustment in our model, and from the RMSE results table, we can choose the final model as the "Capped Regularized Total Effect Model", which provides us with the lowest RMSE. As expected, as this model is the one with more effects included on it.

```
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") %>%
  group_by(userId) %>%
  summarize(b_u = sum(rating - b_i - mu)/(n()+lambda))
b_n <- train_set %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  group_by(n) %>%
  summarize(b_n = sum(rating - b_u - b_i - mu)/(n()+lambda))
b_t <- train_set %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  left_join(b_n, by="n") %>%
  group_by(time) %>%
  summarize(b_t = sum(rating - b_n - b_u - b_i - mu)/(n()+lambda))
predicted_ratings <-
  test_set %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  left_join(b_n, by="n") %>%
  left_join(b_t, by="time") %>%
  mutate(pred = mu + b_i + b_u + b_n + b_t) %>%
  mutate(pred = ifelse(pred>5, 5, pred)) %>%
  mutate(pred = ifelse(pred<0.5, 0.5, pred)) %>%
  pull(pred)

final_rmse <- RMSE(predicted_ratings, test_set$rating)

rmse_results <- bind_rows(rmse_results,
                          data_frame(method="Capped Regularized Total Effect Model",
                                     RMSE = final_rmse))
```

The new RMSE including capped max and min is 0.8640.

```
## # A tibble: 7 x 2
##   method                                  RMSE
##   <chr>                                  <dbl>
## 1 Just the average                      1.0599
## 2 Movie Effect Model                    0.94374
## 3 User+Movie Effect Model               0.86593
## 4 User+Movie+#Ratings Effect Model      0.86510
## 5 User+Movie+#Ratings+Time Effect Model 0.86477
## 6 Regularized Total Effect Model        0.86419
## 7 Capped Regularized Total Effect Model 0.86408
```

4. Conclusion

To do this project a simple linear model has been chosen. Due to the large amount of data, no machine learning general model could be run neither the train function from *caret* package to train any model. Thus,

different effects from predictors were added linearly to the overall prediction equation.

Also, due to the large amount of data, cross-validation took long to run (45min aprox), for lambda to be chosen. As with only 5 k-fold sets lambda converged and resulted only in two different possible values (4.5 or 4.75), it did not seem necessary to add any additional k-fold set.

Also, no genre effect has been added into our model, as needed RMSE was already achieved. Nevertheless, genre has an effect in the ratings and in an improved model, it should be included.