Movielens Project

Yeti44

10 3 2020

Contents

0.1 Executive Summary

The task is to establish a movie recommendation algorithm based on the MovieLens dataset. This dataset consists of millions of movie ratings of thousands of movies from thousands of users. The goal is it to predict the movie ratings with an "root-mean-square-error" (RMSE) below **0.86490**. For this exercise we used the 10M MovieLens dataset version to make computation easier possible. The calculations are done as approximations and not with a predefined algorithm like e.g. glm() function as this would have crashed my computer. The approach was to build an algorithm following this logic: $Y_{i,u,c} = \mu + b_i + b_u + b_? + \epsilon_{i,u,c}$ where $\mathrm{Y}_{\mathrm{i,u,c}}$ are the predicted ratings based on μ, the movie bias (b_i), the user bias (b_u), the bias out of findings from Rating-Matrix decomposition ($\mathrm{b}_?$) and the non-further specified remaining error (ϵ). To get to the final version of this algorithm (incl. final validation) I followed the following five steps:

1. Generate the edx dataset: Download and wrangle "10M Movilens" data. Split it into a dataset to work with - the "edx" dataset - and a validation dataset. This step was done by HarvardX for us.
2. Understand, wrangle and analyse edx dataset
3. Build a train- and test dataset out of edx dataset and define the RMSE function
4. Building the algorithm by a step by step approach:
    i) Just use $\mu + \epsilon$
    ii) add movie bias (b_i) to the equation
    iii) add user bias (b_u) to the equation
    iv) do improvements thanks to regularization
    v) Adding a bias, we calculate out of the decomposition of the Rating-Matrix
5. Validate algorithm against validation dataset and discuss result

The knowledge/information used - beside different internet researches - is mainly from the material we learned in the module 8 of the HarvardX Data Science course series, recommenderlab documentation [1], and two further publications one about singular value decomposition [2] (SVD) and a tutorial about principal component analysis [3] (PCA).

The RMSE value the algorithm achieved running against validation dataset is **0.86446** which means the final algorithm is good enough to achieve the goal set.

0.2 Method/Analysis

0.2.1 Generate edx dataset

The code to generate the edx dataset was provided by HarvardX. As there is only one minor change (added three additional libraries) it will not be commented further. In the .Rmd version of this document the code will be included.

0.2.2 Understand, clean and analyse edx dataset

First we will have a look at edx dataset to get a "look and feel" for it:

```r
head(edx)
```

```
##   userId movieId rating timestamp                         title
## 1      1     122      5 838985046              Boomerang (1992)
## 2      1     185      5 838983525               Net, The (1995)
## 4      1     292      5 838983421               Outbreak (1995)
## 5      1     316      5 838983392               Stargate (1994)
## 6      1     329      5 838983392 Star Trek: Generations (1994)
## 7      1     355      5 838984474       Flintstones, The (1994)
##                           genres
```

[1] http://www2.uaem.mx/r-mirror/web/packages/recommenderlab/recommenderlab.pdf

[2] https://medium.com/@jonathan_hui/machine-learning-singular-value-decomposition-svd-principal-component-analysis-pca-1d45e885e491

[3] https://strata.uga.edu/software/pdf/pcaTutorial.pdf

```
## 1                 Comedy|Romance
## 2          Action|Crime|Thriller
## 4  Action|Drama|Sci-Fi|Thriller
## 5        Action|Adventure|Sci-Fi
## 6 Action|Adventure|Drama|Sci-Fi
## 7        Children|Comedy|Fantasy
```

```
dim(edx)
```

```
## [1] 9000055       6
```

```
class(edx)
```

```
## [1] "data.frame"
```

So we have a data.frame with 9000055 rows and 6 columns, which is by the way composed of 10677 different movies and 69878 different users.

Next we will check if there are any zeros or "NA" in the dataset:

```
edx %>% filter(rating == 0) %>% tally()
```

```
##   n
## 1 0
```

```
sum(is.na(edx))
```

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

As everything looks quite good for this exercise no further data cleaning would be necessary. Nevertheless, it could be handy for future work to separate the movie title from the release year - so that we get a data frame with 7 columns. Even so for better readability there is an adjustment of the column sequence. After these changes the new data.frame looks like this (new dataset name is "edx1"):

```
##   movieId                  title year                        genres userId
## 1     122              Boomerang 1992                Comedy|Romance      1
## 2     185               Net, The 1995         Action|Crime|Thriller      1
## 3     292               Outbreak 1995  Action|Drama|Sci-Fi|Thriller      1
## 4     316               Stargate 1994       Action|Adventure|Sci-Fi      1
## 5     329 Star Trek: Generations 1994 Action|Adventure|Drama|Sci-Fi      1
##   rating timestamp
## 1      5 838985046
## 2      5 838983525
## 3      5 838983421
## 4      5 838983392
## 5      5 838983392
```

Now let us do some visualization to get more insights about the data - first, we have a look at random sample of 100 users and 100 movies to see if there are Gaps in the ratings:

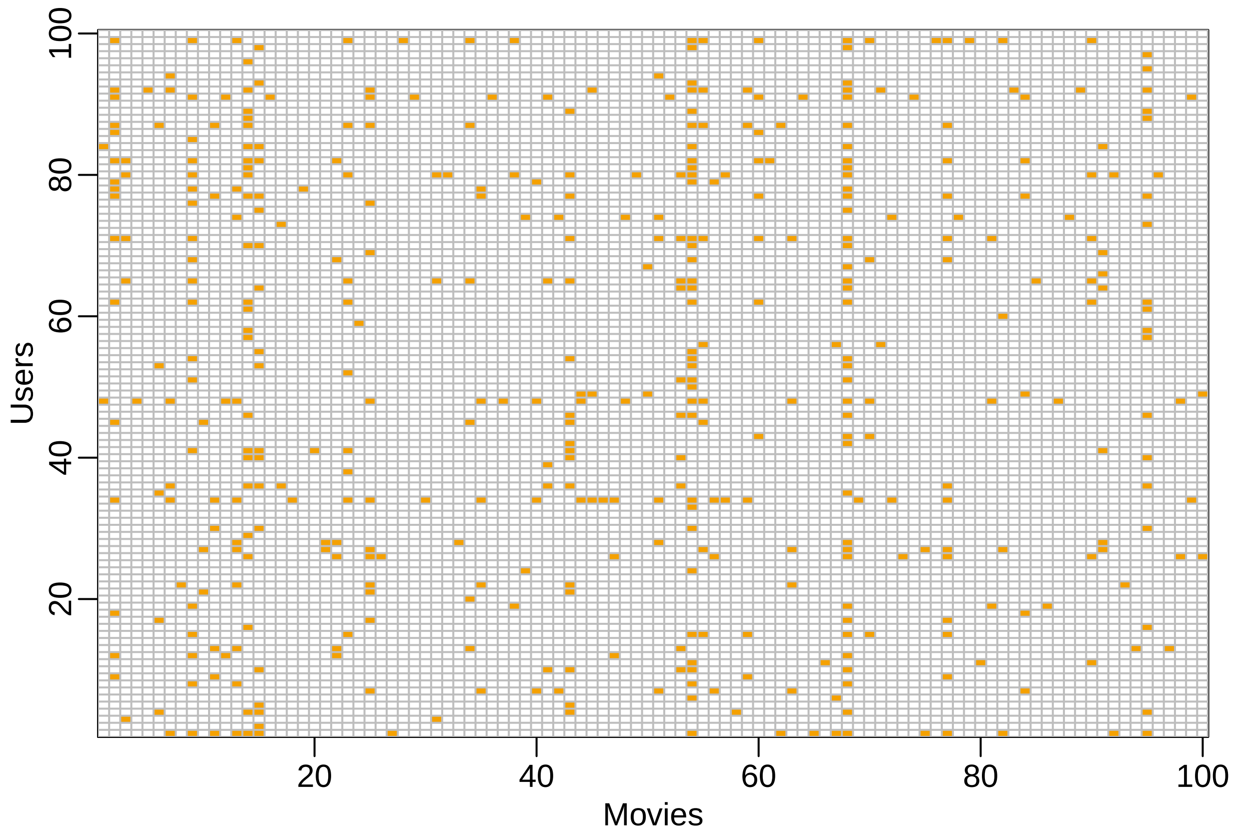

Figure 1: Visualization of the Gaps in movie ratings shown by a random sample of 100 users

Second, we look at the distribution of the movie ratings:

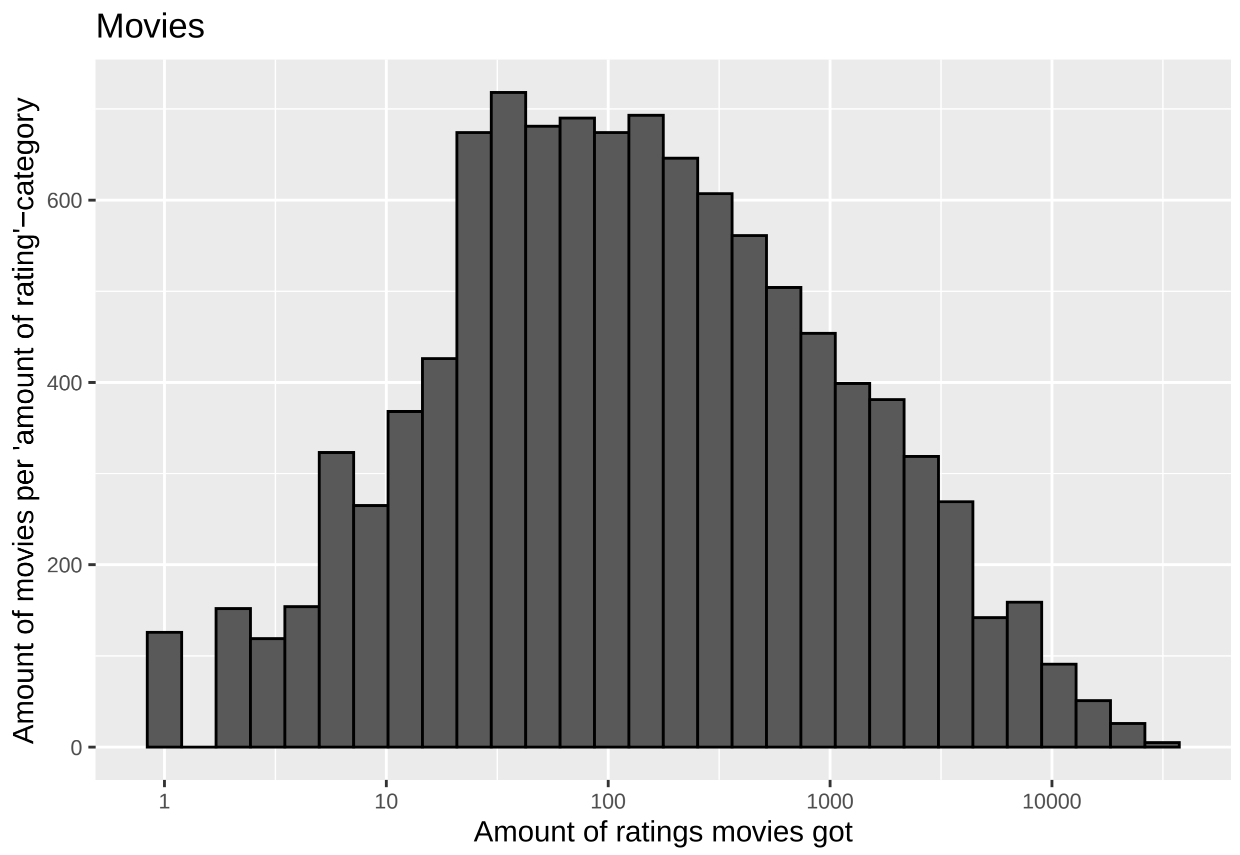

Figure 2: Distribution of amount of movie ratings

Figure 1 and 2 show in different ways the “gaps” we have in the ratings, meaning that by far not all movies got rated by all the users. This we must take into consideration when we build the algorithm as this could adulterate our results. For example, when ratings far away from the average are backed only from few users or when we want to work with a userId/movieId/rating-Matrix to do factorization.
Concerning the ratings, it seems that people do prefer the integer values. Even so there is a clear tendency towards ratings above the scale average of 2.5 towards a rating between “above average” to “good” (3-4):

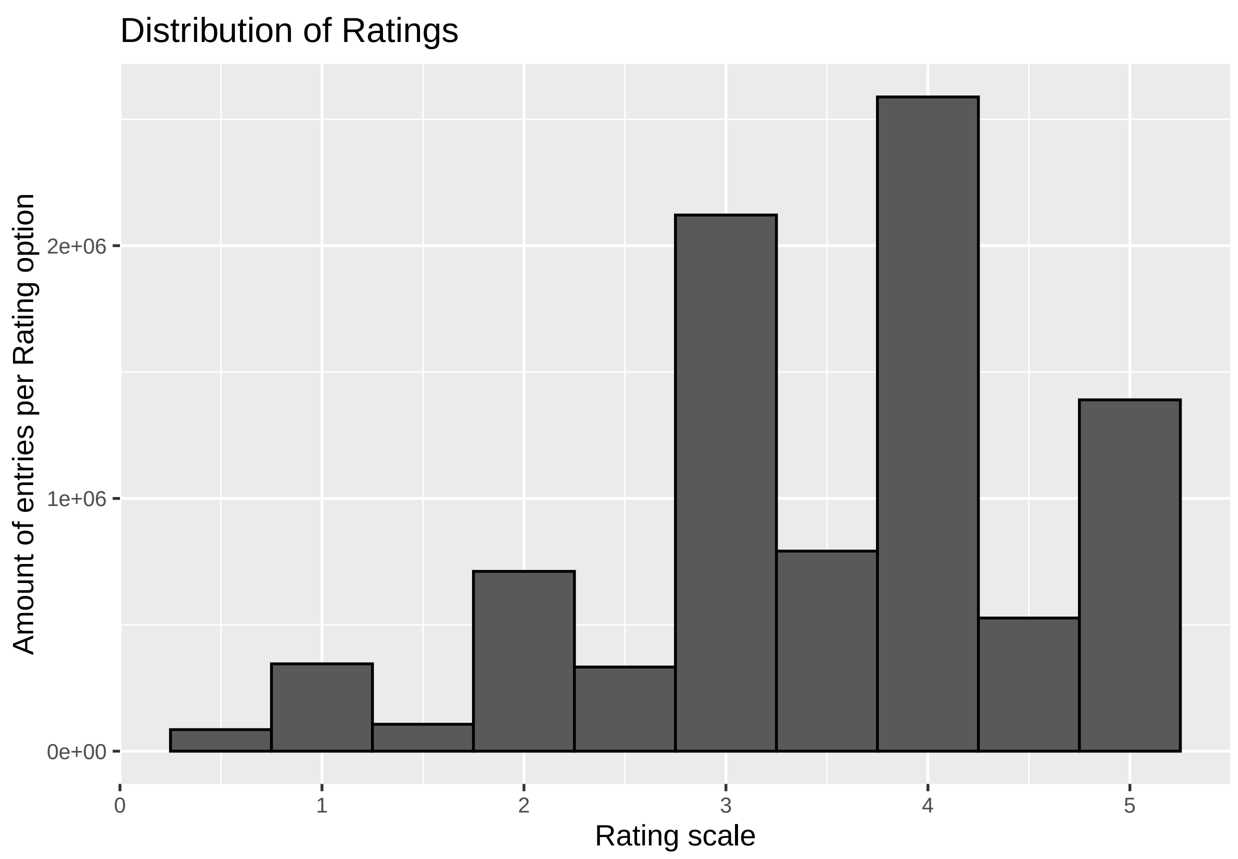

Figure 3: Distribution of ratings

0.2.3 Build a train- and test dataset + define RMSE function

With the `createDataPartition` function we build us two datasets out of the edx1 dataset. One is the edx_train set which we will use to train/build our algorithm. The other is edx_test against which we will test the prediction capabilities of our algorithm throughout the different steps.

Following we look at the dimensions of the two datasets - we will see that edx_train set consists of about 85% of the original edx1 dataset and edx_test of 15% accordingly:

```
dim(edx_train)
```

```
## [1] 7650073       7
```

```
dim(edx_test)
```

```
## [1] 1349982       7
```

Last but not least we define the RMSE function before we can start building the algorithm:

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

0.2.4 Building the algorithm step by step

The algorithm will follow this logic: $Y_{i,u,c} = \mu + b_i + b_u + b_c + \epsilon_{i,u,c}$. We will build the algorithm in a step by step approach so we can compare results (RMSE against edx_test) after every step.
#### step 1: Starting with "mu" To start with we just predict the ratings by setting them on the rating average from edx_train $Y = \mu$:

```r
#Calculating mu
mu <- mean(edx_train$rating)
mu
```

```
## [1] 3.512528
```

This results in the following RMSE value:

method	RMSE	Improvement
Mu	1.060593	0

0.2.4.1 Step 2: Adding $\mathbf{b_i}$

As a next step we add the movie bias ($\mathrm{b_i}$) to the algorithm. For this we take the average of each movie "corrected" by mu:

```r
bi_avg <- edx_train %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu), n = n())
```

Including $\mathrm{b_i}$ RMSE gets the following value:

method	RMSE	Improvement
Mu	1.0605925	0.0000000
Mu, b_i	0.9442124	0.1163802

As we can see there is a substantial improvement of 11% in the RMSE so the bias per movie seems to be somehow big. It seems by some movies people were +/- in agreement to give them a rating clearly above or below the overall rating average - as if a movie got a controversial rating (equally good and bad ratings) the deduction of mu would have "corrected" the effect.

0.2.4.2 Step 3: Adding $\mathbf{b_u}$

Now we add the User bias ($\mathrm{b_u}$):

```r
bu_avg <- edx_train %>%
  left_join(bi_avg, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))
```

method	RMSE	Improvement
Mu	1.0605925	0.0000000
Mu, b_i	0.9442124	0.1163802
Mu, b_i, b_u	0.8660544	0.0781580

Also adding $\mathrm{b_u}$ brought us a step forward. We achieved a further improvement of RMSE value by 8% (against edx_test dataset).

0.2.4.3 Step 4: Adding regularization

As a next step we look at regularization. So, we want to penalize large deviations of the average caused from small sample sizes. But before doing the calculations we want to do a quick analysis of the data to check if such an effect could exist.
To do so we want to have a special look at the residuals still bigger than |1| after deducting mu, b_i and b_u:

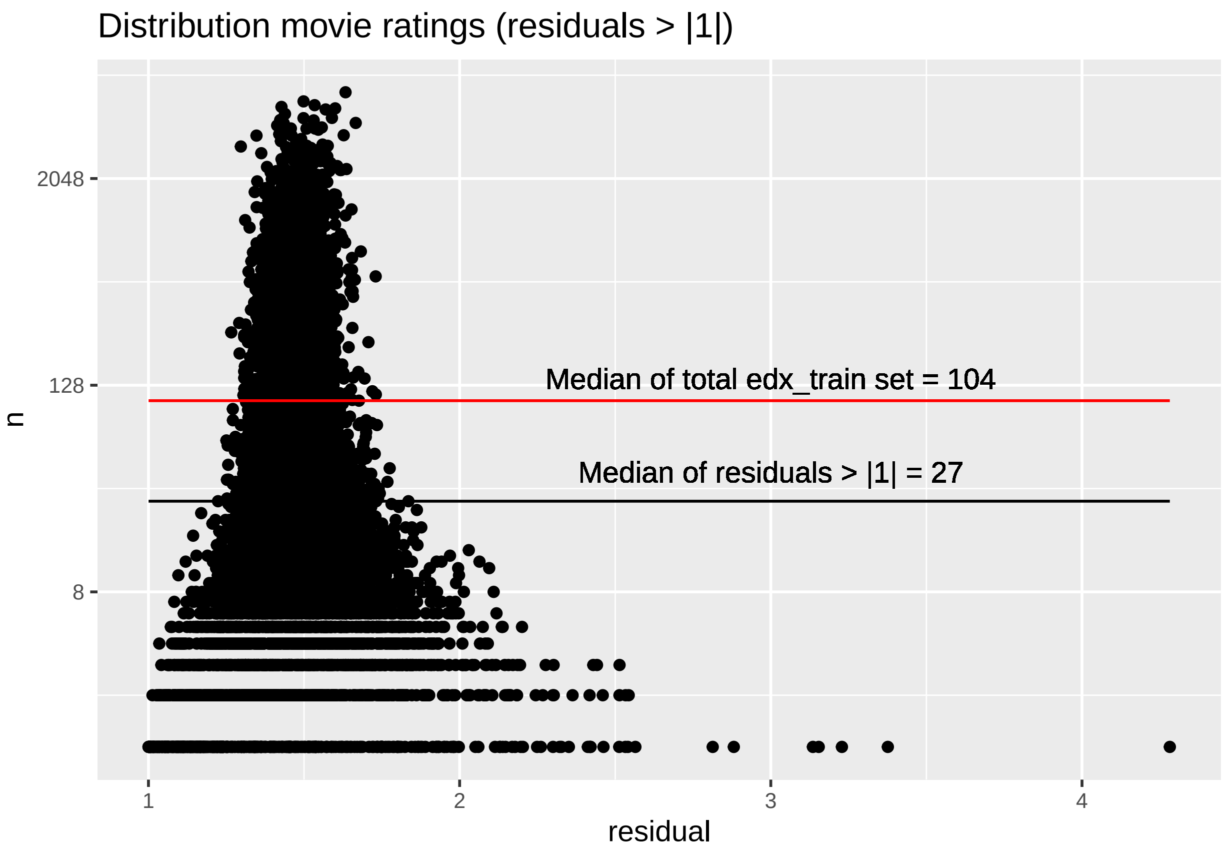

Figure 4: Amount of ratings by movies with residuals > |1|

We see that the amount of ratings a movie got with residual still bigger |1| are clearly lower compared to the median of the whole training dataset (red line). Similar we can observe when we sort it by userId instead of movieId. Therefore, it is probably reasonable to do a regularization.
First we calculate the optimal choice of lambda.

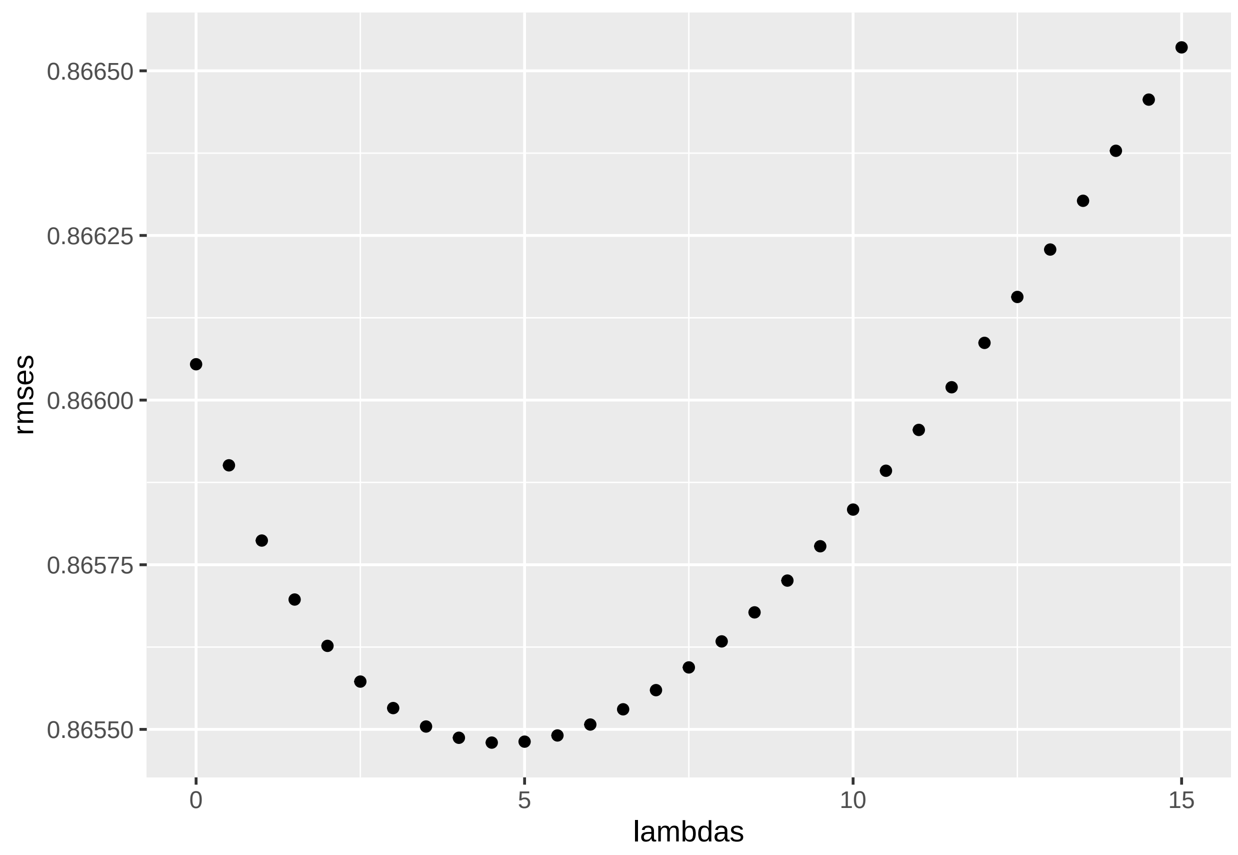

Figure 5: RMSE development with differnt lambdas

Now we recalculate the movie- and user bias “penalized” with optimal lambda (4.5) - the new variables we will call b_i_reg and b_u_reg.

```
#Calculating b_i_reg based on best lambda value:
bi_reg_avg <- edx_train %>%
  group_by(movieId) %>%
  summarize(b_i_reg = sum(rating - mu)/(n()+lambda))

#Calculating b_u_reg based on best lambda value:
bu_reg_avg <- edx_train %>%
  left_join(bi_reg_avg, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u_reg = sum(rating - b_i_reg - mu)/(n()+lambda))
```

If we now generate RMSE based on the regularized biases we get the following values:

method	RMSE	Improvement
Mu	1.0605925	0.0000000
Mu, b_i	0.9442124	0.1163802
Mu, b_i, b_u	0.8660544	0.0781580
‘Regularized’	0.8654799	0.0005745

With a further improvement of RMSE score of 0% this step did not bring us too much of an additional benefit. But it is also to say that we are at a stage with our algorithm where it gets harder and harder to find further improvements.

0.2.4.4 Step 5: Matrix decomposition

In this next step we want to try out two different approaches by decomposing the Rating-Matrix.

1. Add a bias based on PCA. Thanks to this we should be able to consider dependency patterns in between movies and/or users in our algorithm. We call this bias “cluster bias” (b_c)
2. With support of SVD we will rebuild the Rating-Matrix in a way that the Rating-Matrix is fully filled out (no empty cells) and then we take the mean of every movie. We will call this bias b_{i2}

To anticipate one point already now: The initial idea was to do this in a sequence - so first calculate b_c, include it in the algorithm and then based on this rebuild the Rating-Matrix and do step 2. But it showed that this does only bring the second-best result. The best result was achieved by just including one of the two new biases (b_c, b_{i2}) - which one we will see at the end of this chapter.

0.2.4.4.1 PCA calculations (add b_c)

To start with this step, we have to generate the Rating-Matrix. We will do this directly with a capability of the `recommenderlab` package and use the function `as(<data.frame>,realRatingMatrix)`. This function is converting a data.frame directly in a so called “real-value rating matrix”, which is not only converting the data into a matrix but it will store it in a sparse format too (NA values will not be stored but zeros will). With this we can take into consideration too that not all movies got rated by all users and the matrix would therefore have a lot of “blanks” (as we found out in chapter “Understand, wrangle and analyse edx dataset”).

```r
#Calculate predictions for edx_train set:
Reg_rating_train <- edx_train %>%
  left_join(bi_reg_avg, by='movieId') %>%
  left_join(bu_reg_avg, by='userId') %>%
  mutate(pred = b_i_reg + b_u_reg) %>%
  .$pred

#Establish Rating-Matrix with Recommenderlab function - first deduct rating-predictions
#from real ratings so only the residuals remain (not standardized yet):
PreMatrix_edx_train <- edx_train %>%
  select(userId, movieId, rating) %>%
  mutate(rating = rating - Reg_rating_train)
rRM <- as(PreMatrix_edx_train, "realRatingMatrix")
getRatingMatrix(rRM[1:10,1:10])
```

```
## 10 x 10 sparse Matrix of class "dgCMatrix"
##
## 1  .         . . . .        .        .        . . .
## 2  .         . . . .        .        .        . . .
## 3  .         . . . .        .        .        . . .
## 4  .         . . . .        .        .        . . .
## 5  0.4365469 . . . .        .        2.998846 . . .
## 6  .         . . . .        .        .        . . .
## 7  .         . . . .        .        .        . . .
## 8  .         . . . 3.234356 3.480118 .        . . .
## 9  .         . . . .        .        .        . . .
## 10 .         . . . .        .        3.046651 . . .
```

Unfortunately it was not possible to directly do PCA with the `prcomp()` function - it took extremely lot of time (over 12hours - yes, tested it…) and it generated huge datasets. By the way same applied for `svd()`.

So we will do all next step based on svd methodology and svd calculation cone by `recommender()` function from `recommenderlab`. To do so we first check the default settings of the function parameters:

```
## $k
## [1] 10
##
## $maxiter
## [1] 100
##
## $normalize
## [1] "center"
```

These defaults settings should be good enough for us, so we generate singular value decomposition now:

```
r_rRM <- Recommender(rRM, method = "SVD")
str(r_rRM@model$svd)
```

```
## List of 5
##  $ d    : num [1:10] 5051 250 201 189 169 ...
##  $ u    : num [1:69878, 1:10] -0.00378 -0.00378 -0.00378 -0.00378 -0.00379 ...
##  $ v    : num [1:10677, 1:10] 0.00288 0.00271 0.00439 0.00323 0.00533 ...
##  $ iter : int 6
##  $ mprod: int 72
```

As a next step follows some math to reconstruct PCA elements. It is to say that the loadings-Matrix (in `prcomp()` function called "rotation") we do already have as it is the same as Matrix "V" from SVD:

```
# 1. Step: Calculate covariance matrix of rRM via SDV parameters V and D:
cov_M <- sweep(r_rRM@model$svd$v, 2, r_rRM@model$svd$d^2, FUN="*") %*%
  t(r_rRM@model$svd$v)/(ncol(rRM)-1)

# 2. step: Calculate Variation (sdev^2 value from prcomp() function):
eig_value <- sapply(i <- 1:ncol(r_rRM@model$svd$v),function(i) {
  (cov_M[i,]%*%r_rRM@model$svd$v[,i])/
    r_rRM@model$svd$v[i,i]})
sdev_calc <- sqrt(eig_value)

# 3. step: Calculate scores (x value from prcomp() function):
pcaScores <- getRatingMatrix(rRM) %*% r_rRM@model$svd$v
```

Now let's have a look at the variation parameters ($sdev\_calc^2$) to see what the influence on variation each principal component (PC) has:

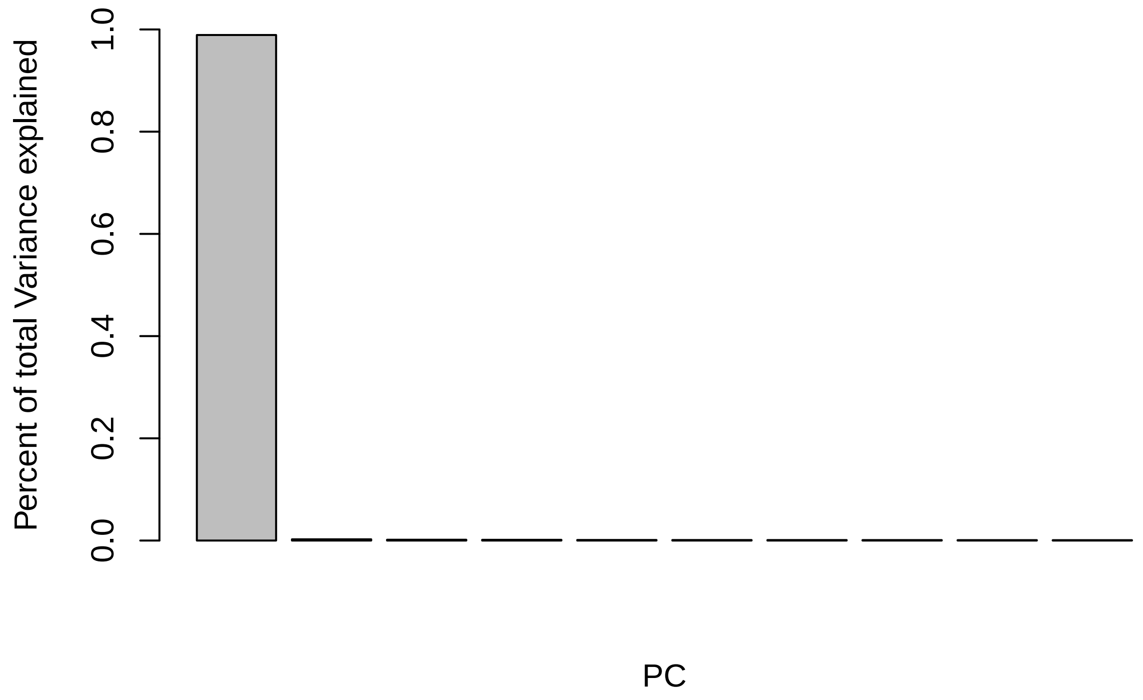

Figure 6: Variation per PC

```
##  [1] 2389.347950    5.855423    3.794144    3.353816    2.678737    2.438299
##  [7]    2.234577    2.128801    1.867526    1.820116
```

As we see first PC is "dominating" clearly and so with just one PC we are capable to explain almost the whole variation. Therefore we will just consider the first column each of our loading- and scores matrix and only the first value of our variation vector to calculate b_c ($b_c = p * q$, while p= user effect and q = principal component):

```
b_c <- (pcaScores[,1]) %*% t(r_rRM@model$svd$v[,1]*eig_value[1])
b_c <- colMeans(b_c)
```

Now we add b_c in the same approach to the algorithm as we have done it before and get the following RMSE table:

method	RMSE	Improvement
Mu	1.0605925	0.0000000
Mu, b_i	0.9442124	0.1163802
Mu, b_i, b_u	0.8660544	0.0781580
'Regularized'	0.8654799	0.0005745
'PCA'	0.8648765	0.0006034

0.2.4.4.2 SVD calculations (add b_{i2})

First, we will again have a look how much of variability is explained with each factor. In SVD we will use vector “D” for this. And again, the picture we gained from the PCA analysis that the first factor is explaining almost all the variation gets confirmed:

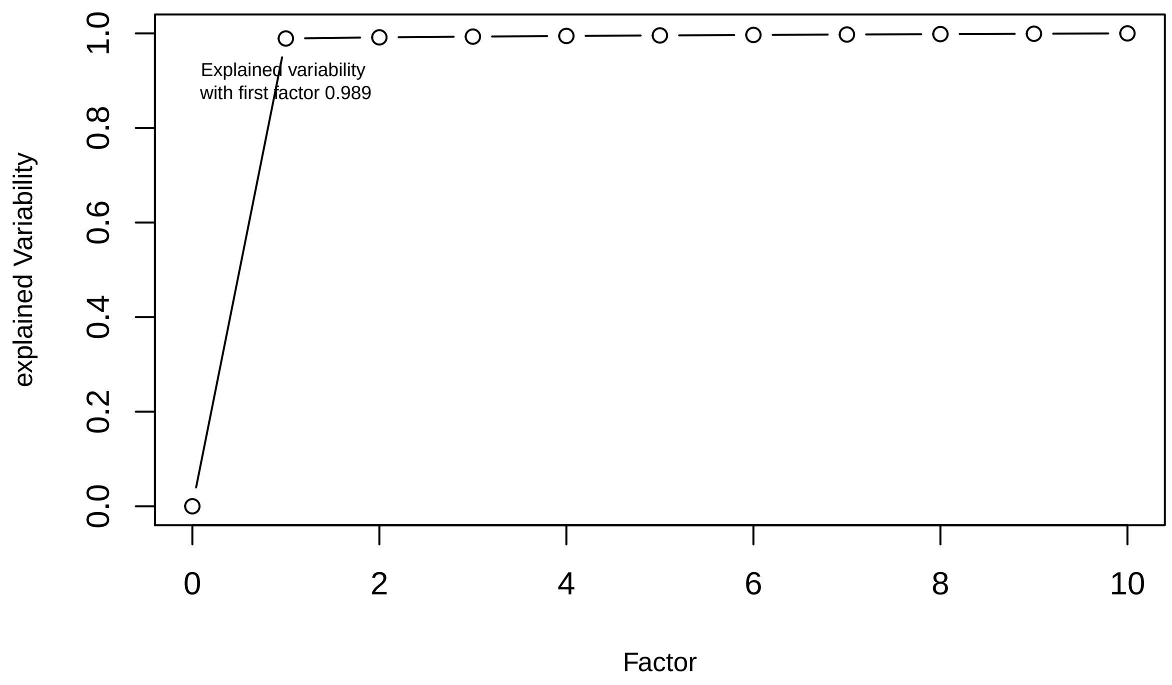

Figure 7: ‘Cumsum’ of factor’s variability

Now we will reconstruct the Rating-Matrix and generate our next bias (b_{i2}). To reconstruct the Rating-Matrix we will use the following formula:

$$Y = UDV^T$$

```r
#Calculate matrix Y with only with u,d,v of first factor:
resid <- with(r_rRM@model$svd,(u[, 1, drop=FALSE]*d[1]) %*% t(v[, 1, drop=FALSE]))
```

As we can see in the new Rating-Matrix there are no blanks left - in contrast to the initial Rating-Matrix shown above.

```
##             [,1]        [,2]        [,3]        [,4]       [,5]
##  [1,] -0.05506110 -0.05184594 -0.08394129 -0.06163106 -0.1018283
##  [2,] -0.05507244 -0.05185662 -0.08395858 -0.06164375 -0.1018492
##  [3,] -0.05506417 -0.05184882 -0.08394596 -0.06163448 -0.1018339
##  [4,] -0.05503732 -0.05182355 -0.08390504 -0.06160444 -0.1017843
##  [5,] -0.05514492 -0.05192486 -0.08406907 -0.06172487 -0.1019833
##  [6,] -0.05508965 -0.05187281 -0.08398481 -0.06166300 -0.1018811
##  [7,] -0.05511926 -0.05190070 -0.08402995 -0.06169615 -0.1019358
##  [8,] -0.05494538 -0.05173697 -0.08376487 -0.06150152 -0.1016143
##  [9,] -0.05508605 -0.05186943 -0.08397933 -0.06165898 -0.1018744
## [10,] -0.05500868 -0.05179657 -0.08386137 -0.06157237 -0.1017313
```

Next step is to generate b_{i2} by calculating the column means from our new Rating-Matrix:

```
b_i2 <- colMeans(resid)
```

So, we have everything to calculate RMSE with b_{i2} added to the algorithm INSTEAD b_c. Here we see the table with the final results (*Attention:* Improvement of "Full rating matrix" is measured against "Regularized" and not against "PCA"):

method	RMSE	Improvement
Mu	1.0605925	0.0000000
Mu, b_i	0.9442124	0.1163802
Mu, b_i, b_u	0.8660544	0.0781580
'Regularized'	0.8654799	0.0005745
'PCA'	0.8648765	0.0006034
'Full rating matrix'	0.8645982	0.0008818

As we can see the better result we get when we add b_{i2} into our formula and therefore our final algorithm looks like this:

$$Y_{i,u,c} = \mu + b_i + b_u + b_{i2} + \epsilon_{i,u,i2}$$

0.3 Results

It is time to apply the generated algorithm to the validation set and check the results we get from it:

```
#Final step: Test the Algorithm against validation set:
Overall_rating <- validation %>%
  left_join(bi_reg_avg, by='movieId') %>%
  left_join(bu_reg_avg, by='userId') %>%
  left_join(temp_factorization2, by='movieId') %>%
  mutate(pred = mu + b_i_reg + b_u_reg + b_i2) %>%
  .$pred

RMSE_Overall <- RMSE(Overall_rating, validation$rating)

#Visualize the RMSE result:
rmse_final <- tibble(RMSE_Validation = RMSE_Overall)
rmse_final %>% knitr::kable()
```

RMSE_Validation
0.8644633

With the final RMSE score of **0.8644633** we reached the goal to get a value below the expected 0.86490. Performance-wise it is certainly a result one can work (or at least start) with. But we also have to remain realistic then an average deviation of the effective rating of about 0.864 points is in our case almost as large as one "full rating category" (e.g. rating categories: very poor, poor, average, good, very good) - in today's time where people are already "spoiled" with quite exact recommendation predictions this would probably not be enough.

0.4 Personal conclusion

The chosen approach delivered the results we were looking for, which certainly is positive. Nevertheless the last step of the algorithm building process was not super satisfying as the PCA decomposition did not brought the results I had anticipated and therefore the efforts I put in it did not really pay off - at least I have now a slight idea about PCA and SVD.... But probably this is also part of a data scientist life: Sometimes you have to fail to get to the next level.
In general it is to say that with every additional bias we added to the algorithm it got tougher and tougher to generate additional benefits concerning the RMSE score. For me this is a hint that the chosen approach has its limitations somewhere around the RMSE score we achieved now. And if we want to bring it to a next level (let's say something like $\text{RMSE} < 0.6$) we would have to go for another approach.
As potential future work I would see three different things I would like to try out:

1. I would dig deeper into the last step of my algorithm building process - perhaps also by talking to a subject matter expert in SVD and PCA topics - to see if there is not more, I could get from it
2. It would be interesting to do this exercise once on a powerful "virtual machine" of a cloud solution (AWS, Azure...). With this extra compute- and memory power we could probably test out predefined algorithms like logistic regression or gradient boosting
3. Even though I do not have a clue about it now: It would be interesting to see the achievable results by applying deep learning algorithms - at least there should be enough data to give it a try with the huge MovieLens dataset