HarvardX Capstone Project - Recommendation System

Arnaud RAULET

21/06/2020

Code provided for this Capstone Project

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

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

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

ratings <- fread(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))

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

# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding")
# if using R 3.5 or earlier, use `set.seed(1)` instead
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]

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

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

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

Before proceeding, please install and load the packages below :

```
if(!require(dendextend)) install.packages("dendextend", repos = "http://cran.us.r-project.org")
if(!require(polycor)) install.packages("polycor", repos = "http://cran.us.r-project.org")
if(!require(psych)) install.packages("psych", repos = "http://cran.us.r-project.org")
if(!require(ggplot2)) install.packages("ggplot2", repos = "http://cran.us.r-project.org")
if(!require(ggcorrplot)) install.packages("ggcorrplot", repos = "http://cran.us.r-project.org")
if(!require(missMDA)) install.packages("missMDA", repos = "http://cran.us.r-project.org")
if(!require(GPArotation)) install.packages("GPArotation", repos = "http://cran.us.r-project.org")
if(!require(FactoMineR)) install.packages("FactoMineR", repos = "http://cran.us.r-project.org")
if(!require(factoextra)) install.packages("factoextra", repos = "http://cran.us.r-project.org")
if(!require(recommenderlab)) install.packages("recommenderlab", repos = "http://cran.us.r-project.org")
if(!require(recosystem)) install.packages("recosystem", repos = "http://cran.us.r-project.org")
if(!require(microbenchmark)) install.packages("microbenchmark", repos = "http://cran.us.r-project.org")
```

Introduction

The goal of this capstone project is to build a movie recommendation system with a RMSE below 0.85, based on the MovieLens dataset. Such a big dataset presents different challenges. One of these is a physical problem, where memory allocation and the limitations of a computer can prevent us from trying different theories. I tried a lot and failed a lot. More precisely, my computer crashed a lot. We know that we cannot perform regression with the `lm` function because it will be too slow or, more probably, R will crash. So is it possible to build a better model than the one's proposed by Professor Irizarry with a fast computation time ? Is it possible to train a model without waiting hours and still get good results ?

After reading the different reports of the Netflix winners, I found that matrix factorization and single value decomposition were a game changer and one of the most important techniques to build predictions. We'll discover later that it is indeed a very fast tool. Moreover, one of the authors of the report, Edwin Chen, says that it is possible to get a low RMSE with "just a few well-selected models". This last comment encouraged me to follow Pareto's rule in order to get 80% of the outcomes with 20% of the efforts.

Fist of all, we will explore the data and see what kind of insights we can get. We will then build a model and discuss its performance.

First, let's look at the data.

Data exploration

Data overview

```
str(edx)
```

```
## '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|Ac
```

```
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                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
```

The dataset is in a tidy format, and we have 6 variables : the rating column, which is the outcome we'll try to predict, and 5 features (`movieId` and `title` are the same, but one is of class integer while the second is of class character). We also have more than 9 millions rows, and more precisely 9 millions and fifty-five ratings. By using the is.na() function, we know that we have no NAs.

But this doesn't mean there is no sparsity in the data. Actually there is a lot of sparsity. Not every user votes for the same movies, nor do every movie is watched by every user.

Let's summarize the data with the following code :

```
edx %>% summarize(users = n_distinct(userId),
                titles = n_distinct(title),
                movid = n_distinct(movieId),
                genres = n_distinct(genres),
                mu = mean(edx$rating))
```

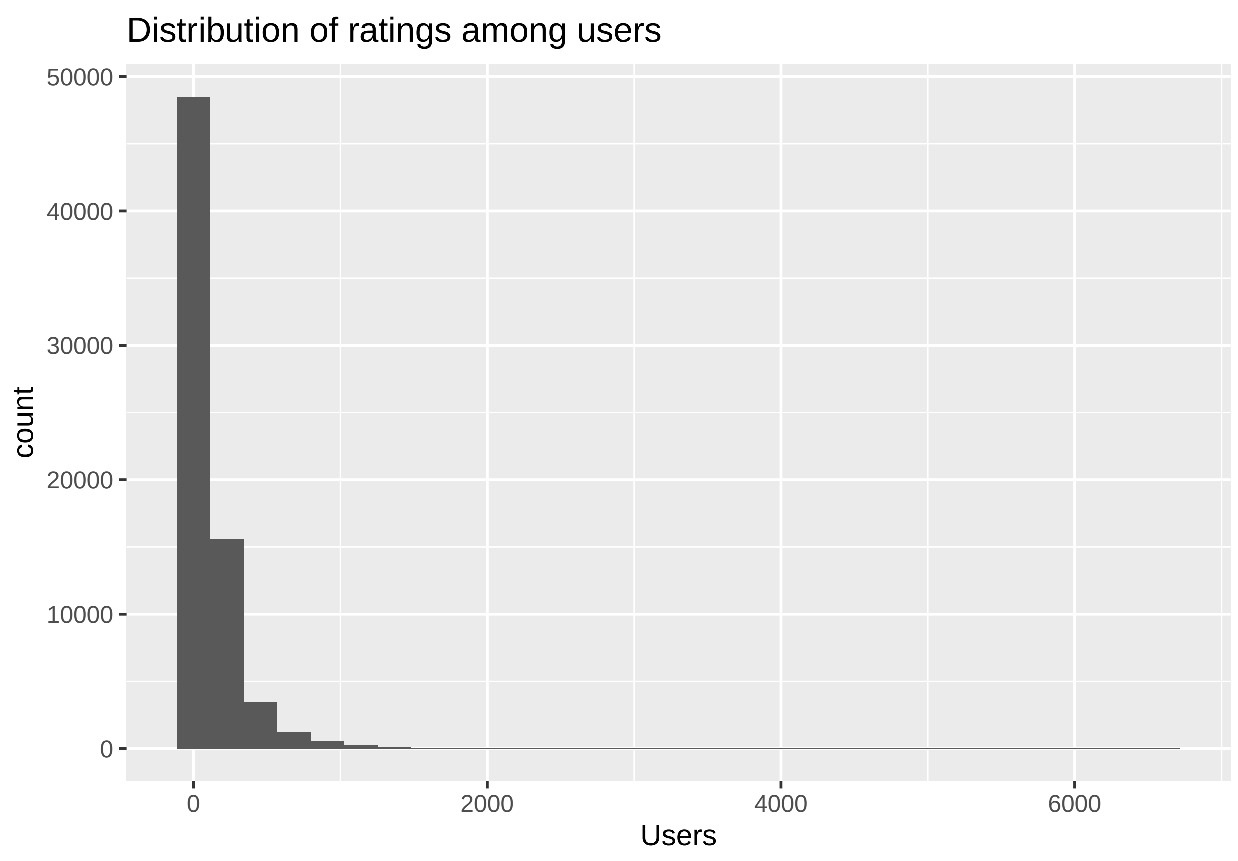
```
##   users titles movid genres       mu
## 1 69878  10676 10677    797 3.512465
```

There are 69,878 users and more than ten thousands titles, so we should have more than 700 millions of ratings if every user was watching and rating every movie. But the density of the data is actually 1.2%. This is not a surprise. After all, have you ever rated every product you bought on Amazon ? As the plots below suggest, a few users rate a lot of movies, and a few movies are watched and rated by a lot of users. Who said Pareto ?

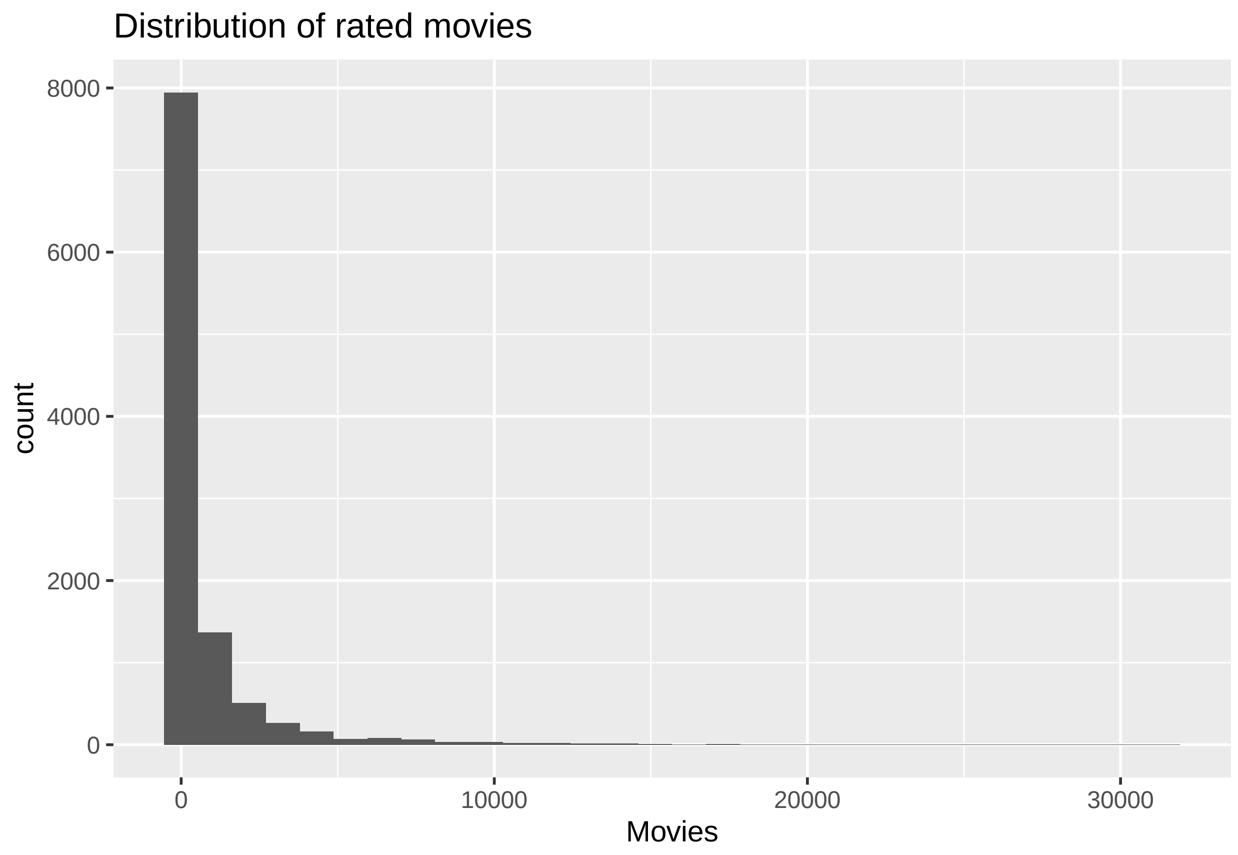
```
# Density of data is 1.2% : Nb of observations / (Nb of users * nb of movies).
9000055 / (69878 * 10676)
```

```
## [1] 0.01206413
```

```
edx %>% group_by(userId) %>% summarize(n = n()) %>%
  ggplot(aes(n))+
  geom_histogram(bins = 30)+
  ggtitle("Distribution of ratings among users")+
  xlab("Users")
```

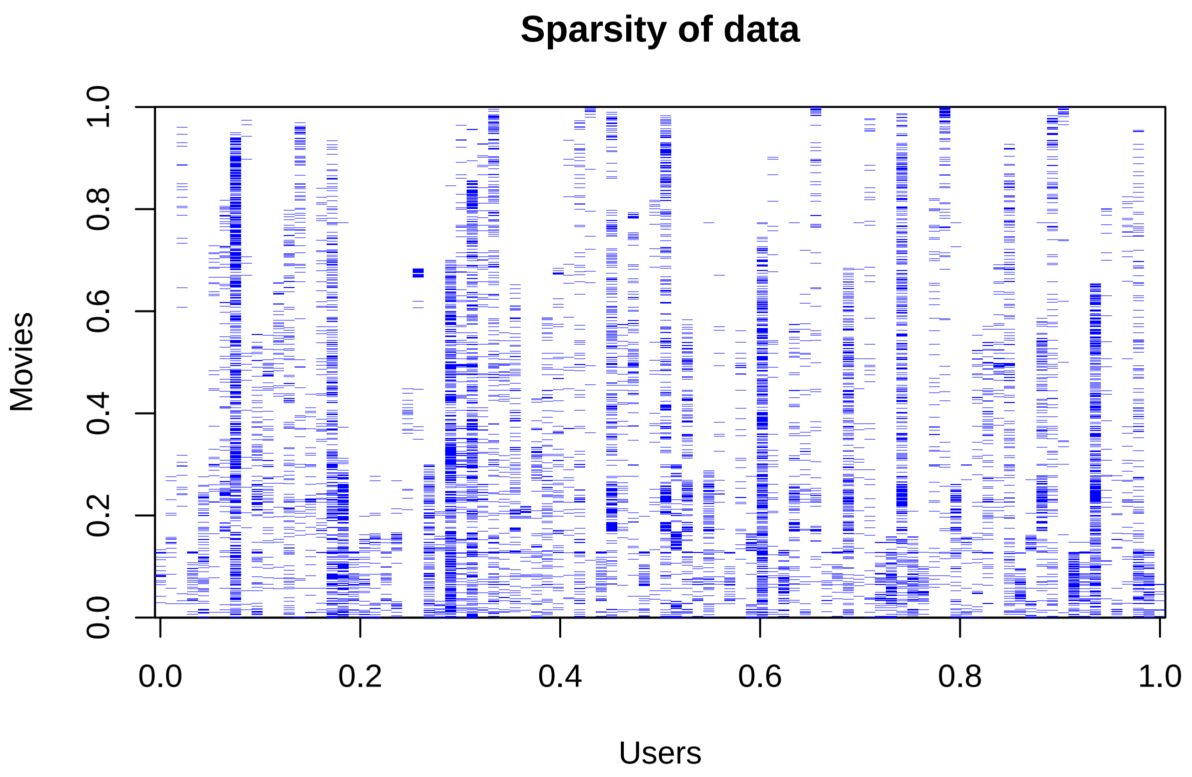


```
edx %>% group_by(title) %>% summarize(n = n()) %>%
  ggplot(aes(n))+
  geom_histogram(bins=30)+
  ggtitle("Distribution of rated movies")+
  xlab("Movies")
```



If we spread the data and create a matrix of the ratings among movies and users with the following code, we get a better point of view of the sparsity :

```
#Let's pick a subset of the first 10,000 observations :
edx_wide <- edx[1:10000,] %>%
  select(userId, movieId, rating) %>%
  spread(movieId, rating)


image(as.matrix(edx_wide[,-1]),
      main = "Sparsity of data",
      xlab = "Users",
      ylab = "Movies",
      col = "blue")
```



As we can see, some users rate more than others, and some movies are more rated than others. And the matrix is filled with a lot of NAs, although we could not see it in the previous tidy format.

The overall mean is 3.51, but as we know, we cannot perform regression from this mean as it will cause computation problems.

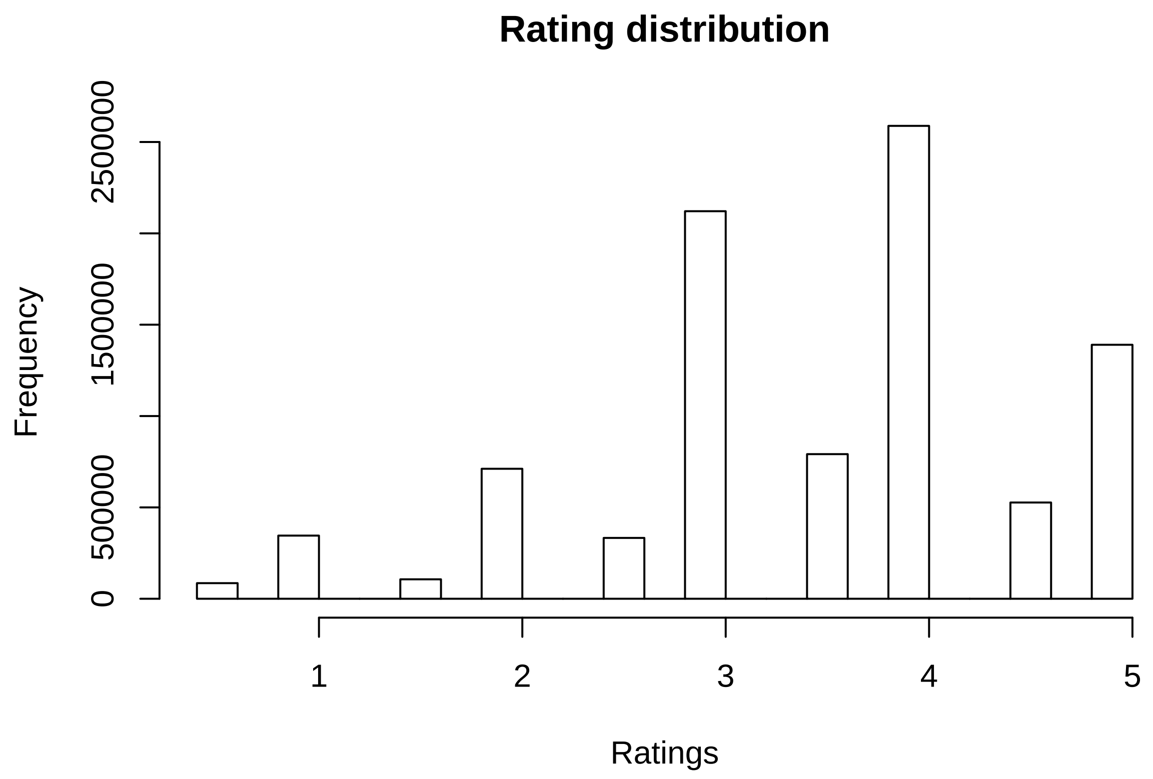
What about the median ?

```r
# Median of the dataset

median(edx$rating)
```

```
## [1] 4
```

So the mean is 3.51 and the median is 4. That means users tend to rate movies they liked or loved. The histogram below shows it better.

```r
# Rating distribution

hist(edx$rating, main = "Rating distribution", xlab = "Ratings")
```

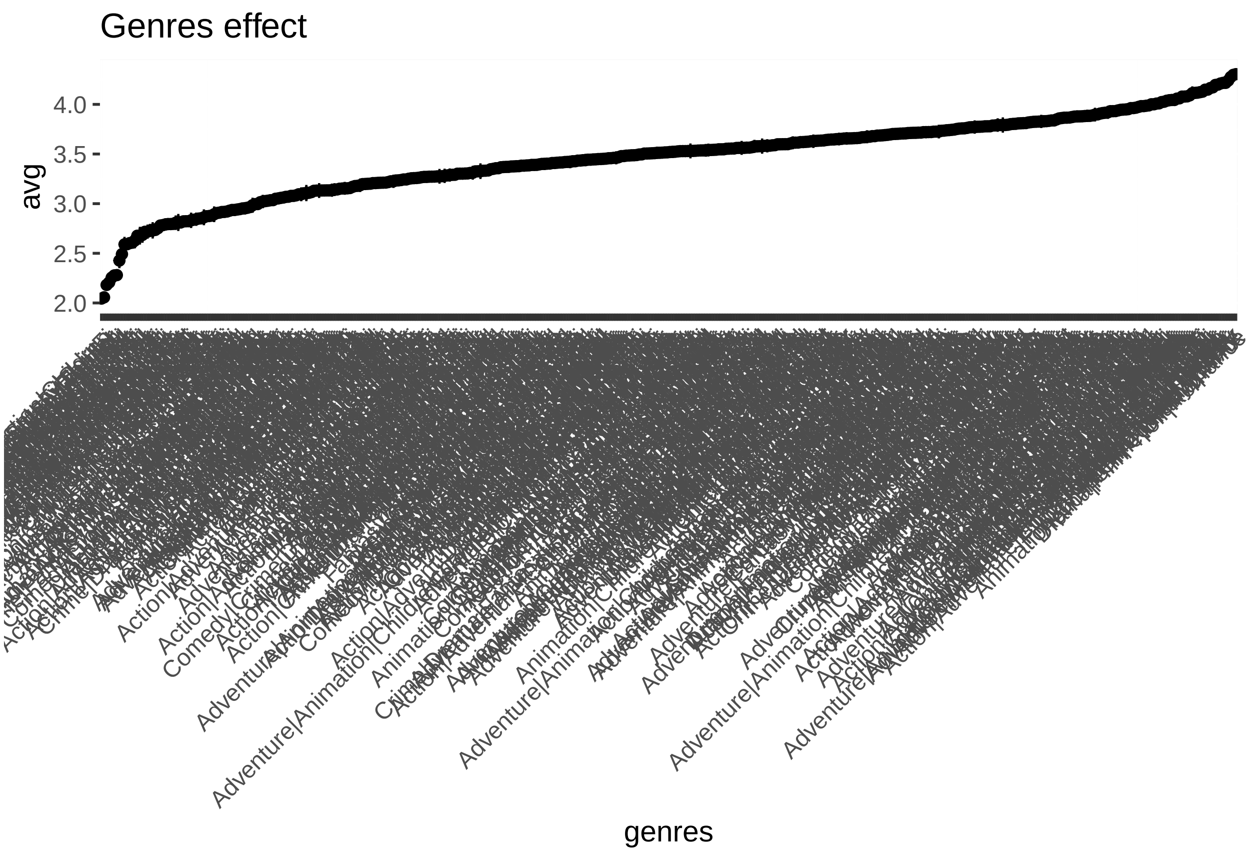


We can relate this to the `genres` column. We all have our own preferences : some prefer western movies, some prefer sci-fi, comedy or romance, and others like maybe all theses genres all at once.

So if we find groups of users that like the same kind of movies, it may be possible to predict what they will rate.

Moreover, we can see there is a `genres` effect on the ratings with the plot below :

```r
# Genres effect

edx %>% group_by(genres) %>%
  summarize(n = n(), avg = mean(rating), se = sd(rating)/sqrt(n())) %>%
  filter(n >= 1000) %>%
  mutate(genres = reorder(genres, avg)) %>%
  ggplot(aes(x = genres, y = avg, ymin = avg - 2*se, ymax = avg + 2*se)) +
  geom_point() +
  geom_errorbar() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  ggtitle("Genres effect")
```



Can we find groups that could help us predicting ratings ? Let's try hierarchical clustering.

Hierarchical clustering

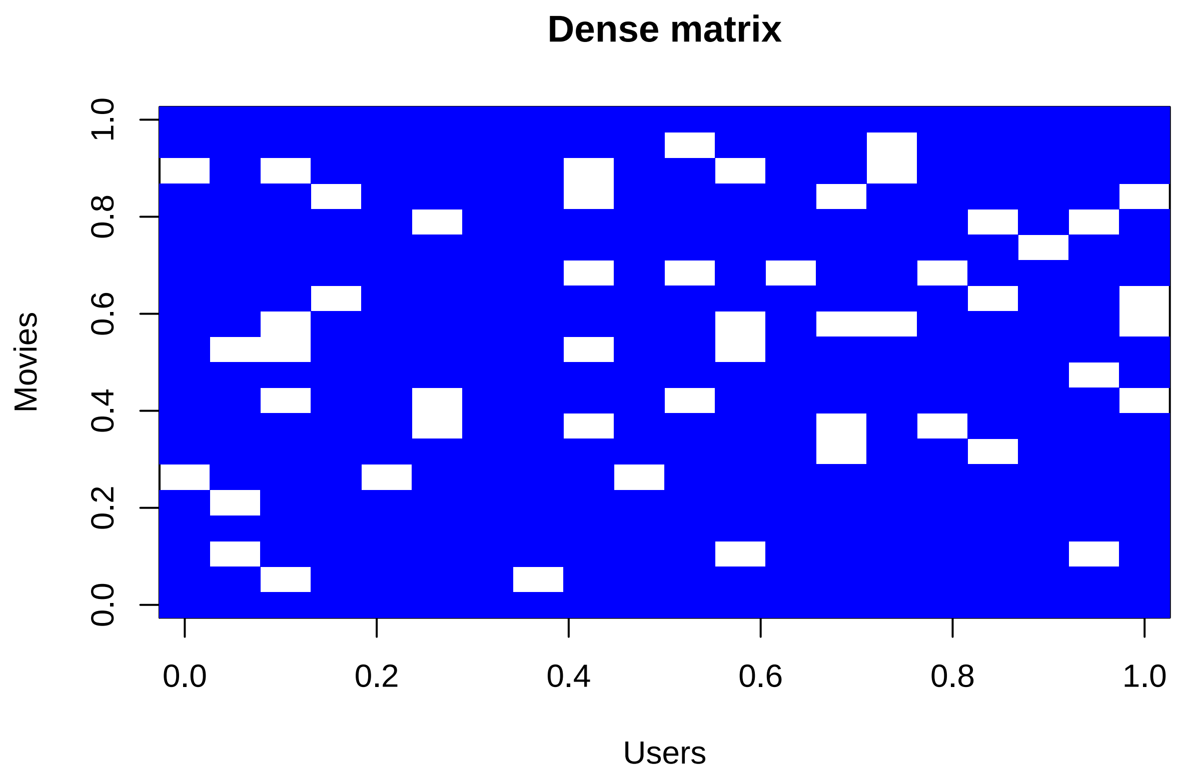
We will first take a very small subset so that results and plots can be readable and visualized. We need a more dense matrix to study correlations, so we will take a subset of most rated movies and most rating users.

```
# Creating a dense matrix

top_users <- edx %>%
  group_by(userId) %>%
  summarize(n=n()) %>%
  arrange(desc(n)) %>%
  top_n(20) %>%
  pull(userId)

top_movies <- edx %>%
  group_by(movieId) %>%
  summarize(n=n()) %>%
  arrange(desc(n)) %>%
  top_n(20) %>%
  pull(movieId)

top_df <- edx %>%
  filter(movieId %in% top_movies, userId %in% top_users)
```

```r
# Adjusting length of titles for better visualization

pattern_year <- c(" \\(19\\d{2}\\)")
pattern_sw <- ": Episode"
pattern_sw2 <- "\\(a.k.a. Star Wars\\)"

col_names <- str_remove(top_df$title, pattern_year)
col_names <-str_remove(col_names, pattern_sw)
col_names <- str_remove(col_names, pattern_sw2)

top_df$title <- col_names

# Creating matrix

top_matrix <- top_df %>%
  select(userId, title, rating) %>%
  spread(title, rating)
```

We now have a less sparse matrix.

```r
# Image of dense matrix

image(as.matrix(top_matrix[,-1]),
      main = "Dense matrix",
      xlab = "Users",
      ylab = "Movies",
      col = "blue")
```

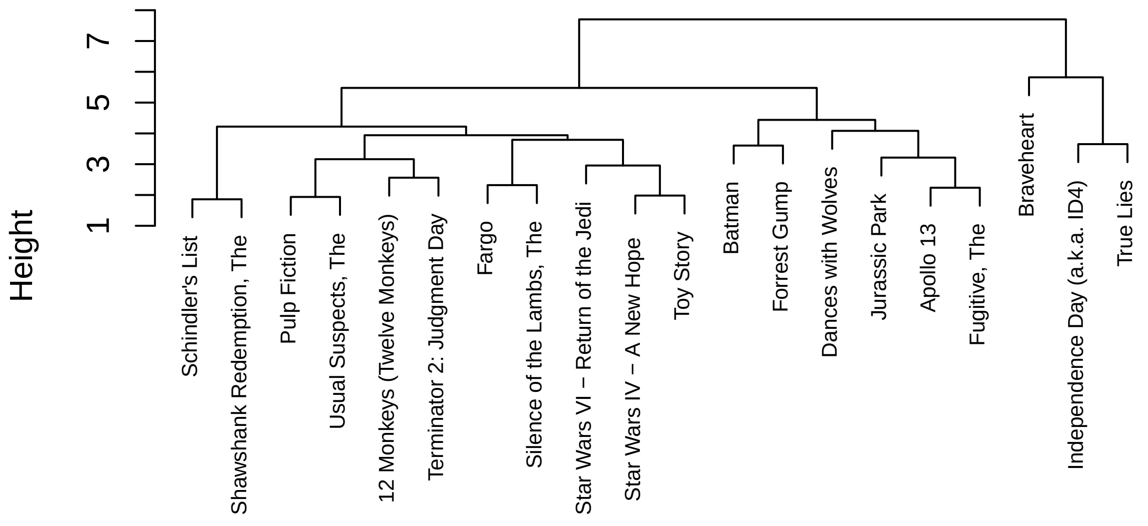


We need to calculate the distance between the different observations, then we identify the resulting clusters.

```
# Calculating distances

dist_set <- dist(t(top_matrix[,-1]), method = "euclidean")

# Clustering

hc_set <- hclust(dist_set, method = "complete")
```

We can now see how it looks on a dendrogram.

```
# Dendrogram

plot(hc_set, cex = 0.65, main = "", xlab = "")
```

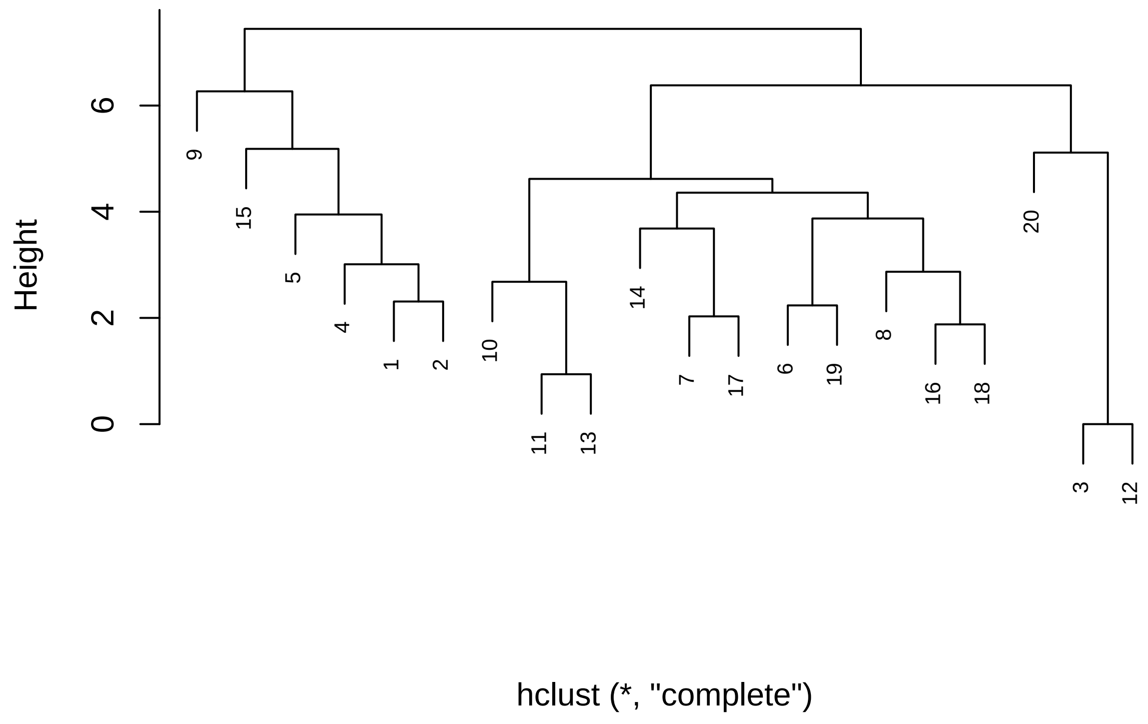


hclust (*, "complete")

All theses movies are very famous movies, but we can see that the only comedy *True Lies* is far from the other movies, while *Star Wars* movies are close as well as “Drama” movies like *Schindler’s list* and *The Shawshank Redemption*.

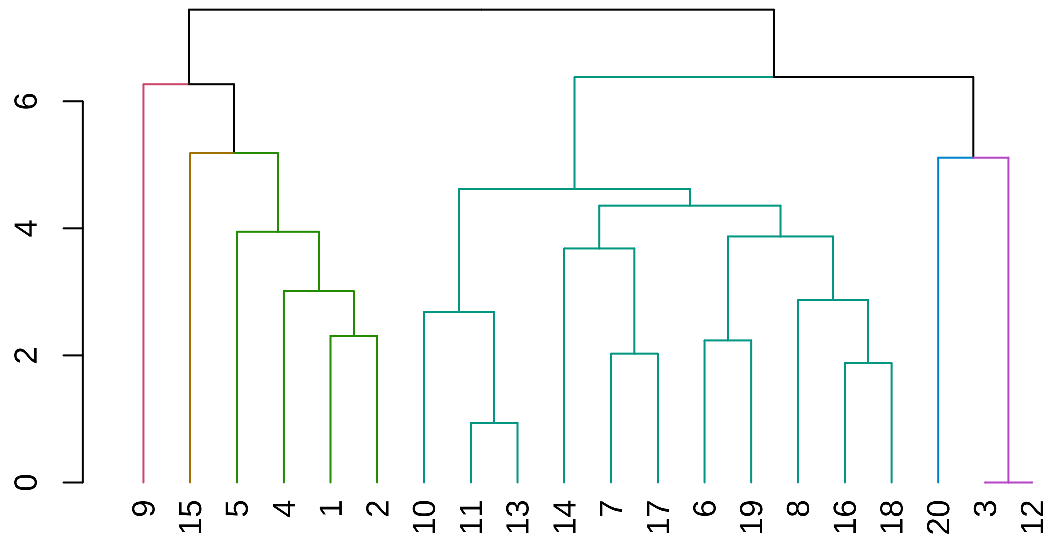
What about users ?

```
# Distance between users
dist_users <- dist(top_matrix[,-1], method = "euclidean")

# Clusters of users
hc_users <- hclust(dist_users, method = "complete")

# Dendrogram
plot(hc_users, cex = 0.65, main = "", xlab = "")
```



Depending on where we cut the tree, we can find different number of clusters. For example, if we cut the above dendrogram to height $h = 5$, we have 6 groups.

```r
# Coloured dendrogram
library(dendextend)

dend <- as.dendrogram(hc_users)
color_dend <- color_branches(dend, h = 5)
plot(color_dend, cex=0.65)
```



Let’s check how the lonely users 9 and 20 rated movies.

```
# 9th user’s ratings
top_matrix[9,]
```

```
##   userId 12 Monkeys (Twelve Monkeys) Apollo 13 Batman Braveheart
## 9  27584                         3.5       3.5      2          2
##   Dances with Wolves Fargo Forrest Gump Fugitive, The
## 9                3.5     3          3.5            NA
##   Independence Day (a.k.a. ID4) Jurassic Park Pulp Fiction Schindler’s List
## 9                             3             2           NA                4
##   Shawshank Redemption, The Silence of the Lambs, The
## 9                       3.5                        NA
##   Star Wars IV - A New Hope  Star Wars VI - Return of the Jedi
## 9                         5                                 3
##   Terminator 2: Judgment Day Toy Story True Lies Usual Suspects, The
## 9                         NA        NA         4                 3.5
```

The 9th user on the matrix seems to be a very picky one. His best rate is a 5 just for *Star Wars : Episode IV* while most movies are blockbusters.

On the other hand, the 20th user loved almost every movie except *True Lies*, an action-comedy film, and *Braveheart*.

```
# 20th user’s ratings
top_matrix[20,]
```

```
##    userId 12 Monkeys (Twelve Monkeys) Apollo 13 Batman Braveheart
## 20  68259                           5         4      5          1
##    Dances with Wolves Fargo Forrest Gump Fugitive, The
## 20                  5     5            4             4
##    Independence Day (a.k.a. ID4) Jurassic Park Pulp Fiction Schindler's List
## 20                            NA             4            5               NA
##    Shawshank Redemption, The Silence of the Lambs, The
## 20                        NA                         5
##    Star Wars IV - A New Hope  Star Wars VI - Return of the Jedi
## 20                          5                                 5
##    Terminator 2: Judgment Day Toy Story True Lies Usual Suspects, The
## 20                         NA         5         2                   5
```

As we can see with these two examples, every user has his own preferences, likes and dislikes, and even popular movies and blockbusters can be badly rated. We cannot see these likes and dislikes, all we can see is ratings from some users for some movies. That's why we 're going to proceed with an exploratory factor analysis to determine the latent factors behind movies, that is to say, why users rate for certain types of movies.

Exploratory Factor Analysis

To get a deeper understanding of what's behind users' preferences and genres effect, we will do an exploratory factor analysis (a.k.a. EFA) to find latent factors that could explain most of the dimensionality of the dataset. If we find these hidden factors behind ratings, we could find who rate which movies and why they rated for those movies, and thus could predict ratings.

To conduct an EFA, we first need to construct a correlation matrix. We can do this with the following piece of code.

```r
library(polycor)

# Getting a correlation matrix
hc_top <- hetcor(top_matrix[,-1], use = "pairwise.complete.obs")

# Getting the correlations
polyc_top <- hc_top$correlations
```

Before proceeding, we must check if our data is factorable. We do this with the Bartlett test and the Kaiser-Meyer-Olkin measure of sampling adequacy. Bartlett's test enables us to check the null hypothesis (p-value < 0.05), whereas KMO indicates the degree to which each variable in a set is predicted without error by the other variables. The closer to 1, the better.

```r
library(psych)

# Bartlett test : p-value should be < 0.05
cortest.bartlett(polyc_top, n=20)
```

```
## $chisq
## [1] 1385.158
##
## $p.value
## [1] 1.769402e-180
##
```

```
## $df
## [1] 190
```

```
# KMO : results should be close to 1
KMO(polyc_top)
```

```
## Kaiser-Meyer-Olkin factor adequacy
## Call: KMO(r = polyc_top)
## Overall MSA =  0.57
## MSA for each item =
##       12 Monkeys (Twelve Monkeys)                         Apollo 13
##                              0.53                              0.59
##                            Batman                        Braveheart
##                              0.53                              0.37
##                Dances with Wolves                             Fargo
##                              0.47                              0.55
##                      Forrest Gump                     Fugitive, The
##                              0.45                              0.73
##     Independence Day (a.k.a. ID4)                     Jurassic Park
##                              0.50                              0.64
##                      Pulp Fiction                  Schindler's List
##                              0.60                              0.63
##         Shawshank Redemption, The         Silence of the Lambs, The
##                              0.54                              0.62
##        Star Wars IV - A New Hope  Star Wars VI - Return of the Jedi
##                              0.67                              0.63
##        Terminator 2: Judgment Day                         Toy Story
##                              0.48                              0.73
##                        True Lies               Usual Suspects, The
##                              0.36                              0.53
```

The Bartlett test result seems good, but the results of KMO are not ideal and may be due to the size of our subset, which is maybe too small, or to the fact that we have mainly blockbusters in our subset. Or maybe because we had NAs when calculating correlations.

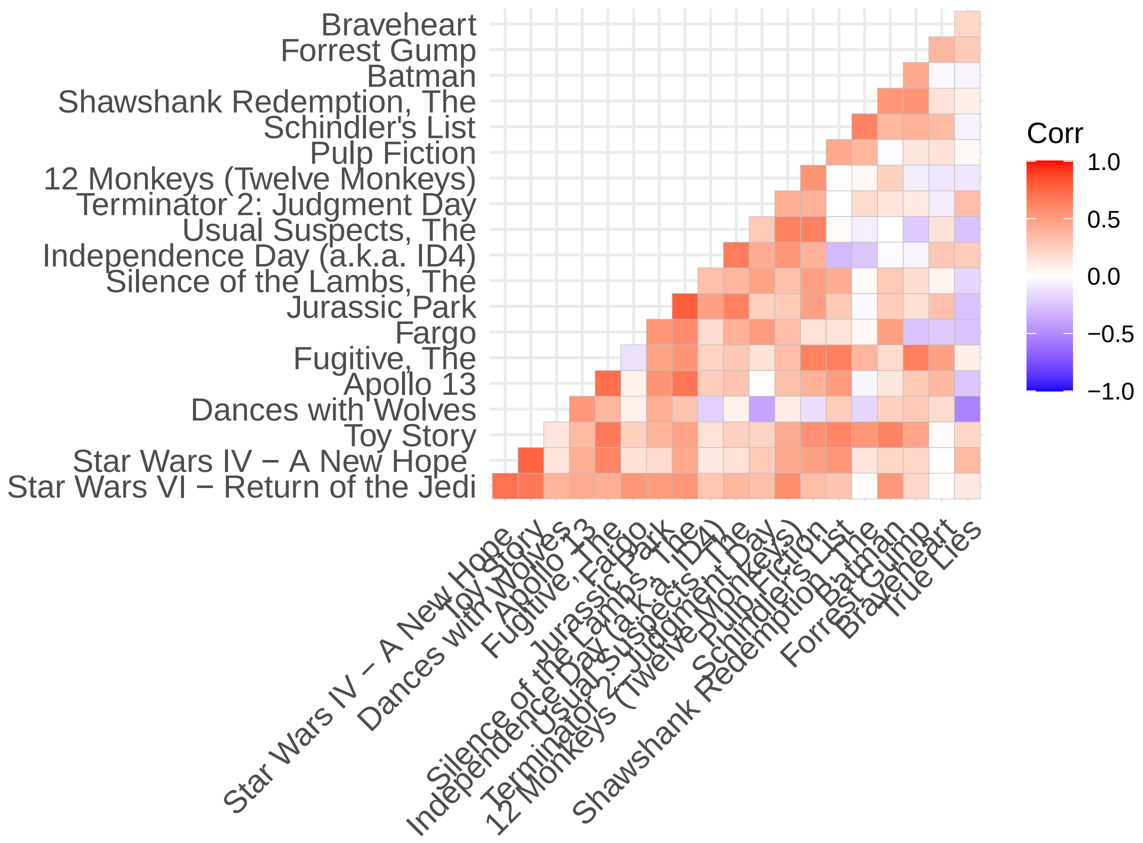
We could impute missing NAs in our first matrix with the following code.

```
library(missMDA)

# Checking the number of missing NAs
sum(is.na(top_matrix))

# Estimating the optimal number of dimensions for imputation
edx_ncp <- estim_ncpPCA(top_matrix[,-1])

# Data imputation
complete_edx <- imputePCA(top_matrix[,-1], ncp = edx_ncp$ncp)

new_matrix <- complete_edx$completeObs
```

We are still exploring, so we will keep the `top_matrix` in the following analysis.

Let's see if we can get some useful insights.

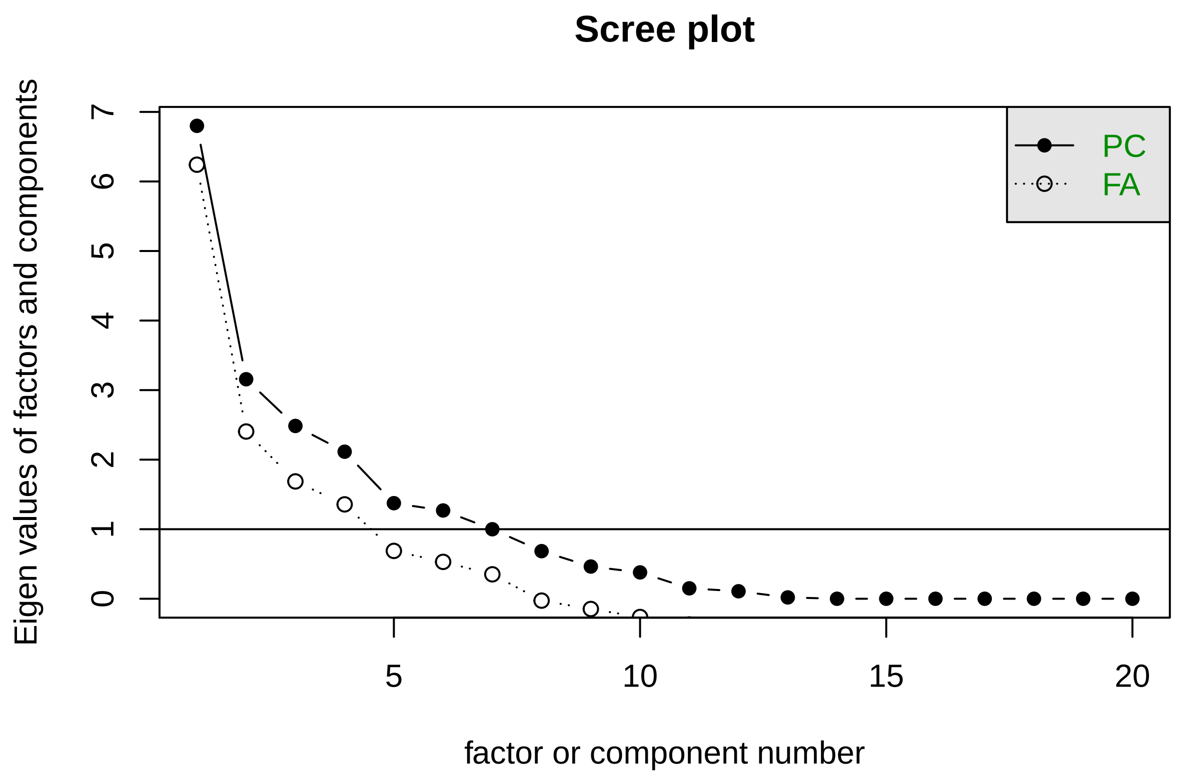
```
library(ggcorrplot)

#Visualizing correlations
ggcorrplot(polyc_top, hc.order = TRUE, type = "lower")
```



As we saw in hierarchical clustering, *True Lies*'s distance was far from the other movies. Here it seems negatively correlated to other movies. It is the only comedy film among those blockbusters, and users in our subset may not like comedy that much.

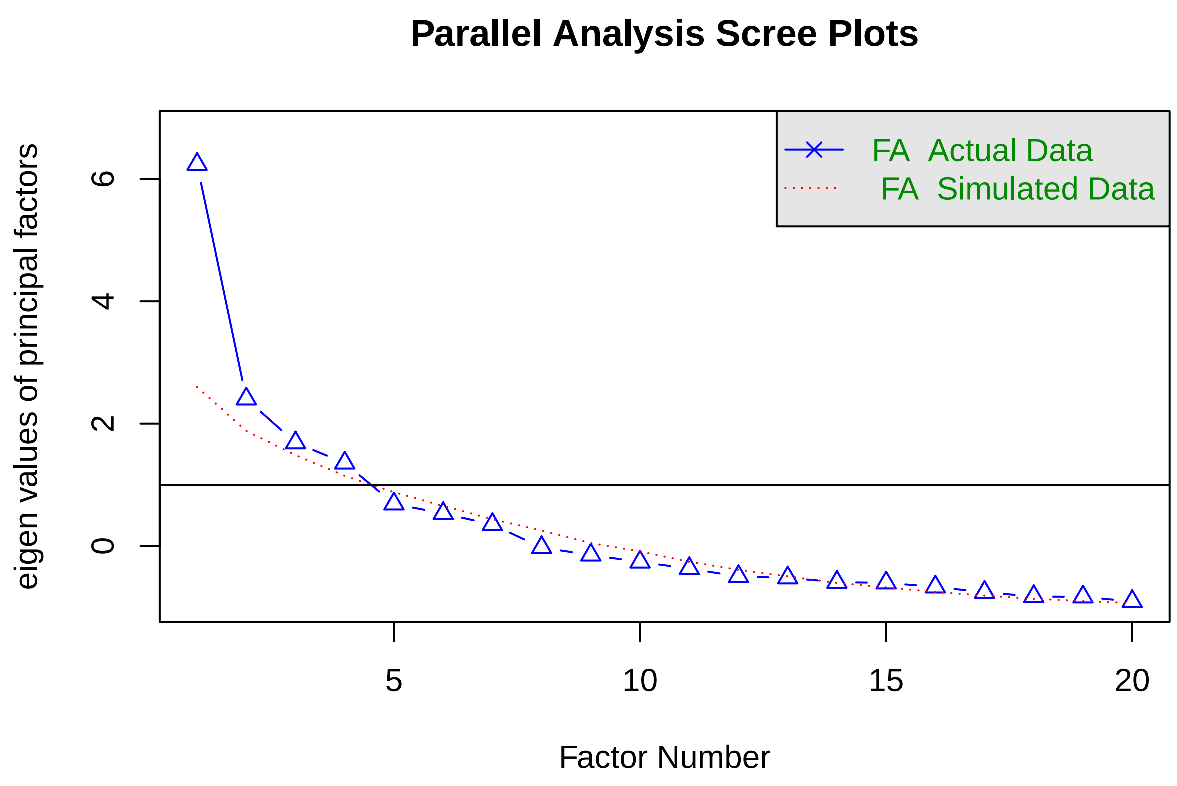
It is now time to extract the factors. Let's check how many factors we should retain with the following scree plot. Only eigen values above or equal to 1 should be retained.

```
# Scree plot
scree(polyc_top)
```



The plot suggests 7 factors in the subset. However if we do a parallel analysis, the result is different.

```
# Parallel analysis scree plot
fa.parallel(polyc_top, n.obs = 20, fm="minres" , fa = "fa")
```

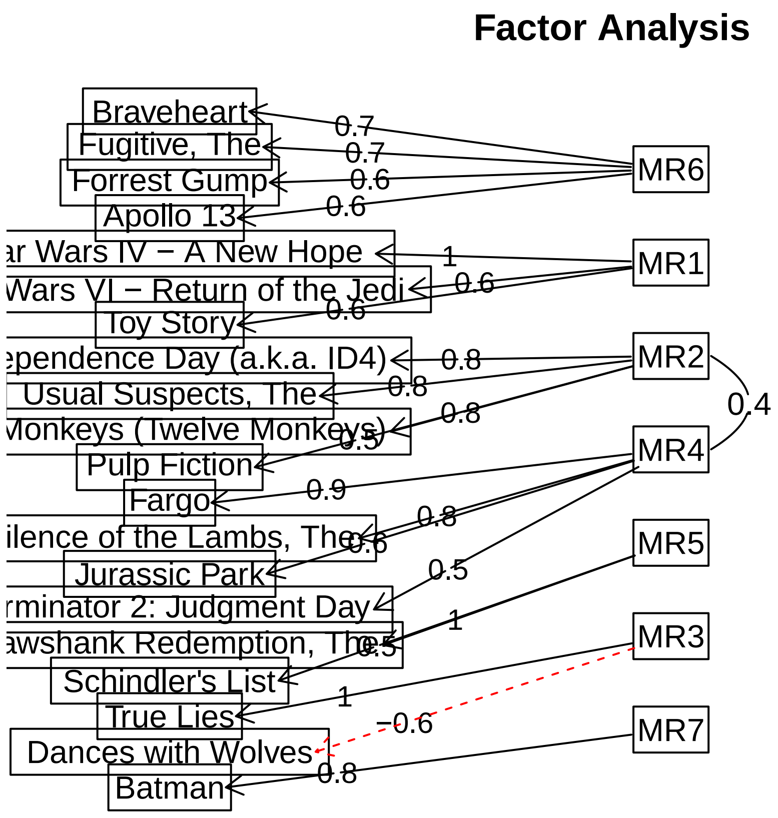


```
## Parallel analysis suggests that the number of factors =  4  and the number of components =  NA
```

This time, it suggests 4 factors. (And warning messages.)

We will extract 7 factors as suggested by the first plot and visualize these latent factors in a path diagram. Also, we will use an oblique rotation as users could like different kind of movies. An orthogonal rotation doesn't seem to be appropriate here.

```
library(GPArotation)

# Factor analysis with 7 factors and oblique rotation
f7_top_obli <- fa(polyc_top, nfactors = 7, rotate = "oblimin")

# Path diagram
fa.diagram(f7_top_obli)
```

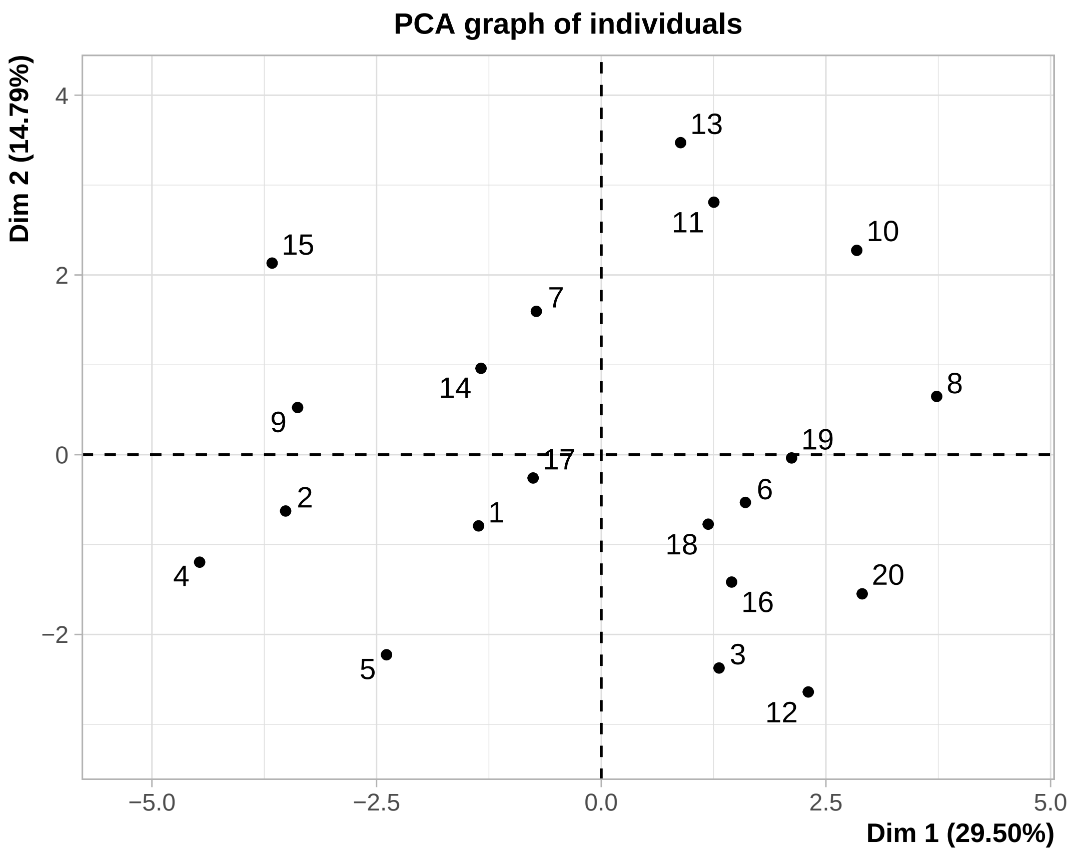


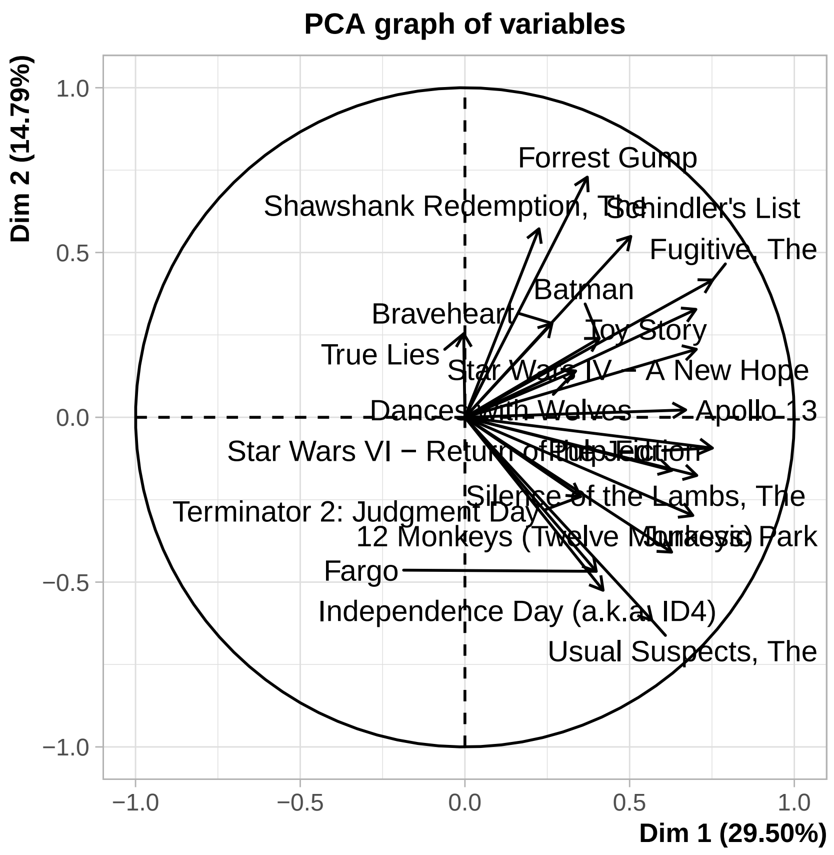
Star Wars movies are found in the first factor, drama movies like *The Shawshank Redemption* and *Schindler's list* are together. We can also see that the second and fourth factors are slightly correlated. Maybe people who liked movies in one group, liked the movies in the second group. The third factor may be related to humor and comedy as it is strongly correlated to *True Lies*, and negatively correlated to *Dances with Wolves*. The seventh factor may refer to people who like comics.

Principal Component Analysis

We will finish this data exploration with a PCA. We will use the `FactoMineR` library. This time, we will try with 4 dimensions.

```r
library(FactoMineR)

# PCA

pca_output <- PCA(top_matrix[,-1], ncp = 4)
```


PCA graph of individuals
Dim 1 (29.50%)
Dim 2 (14.79%)
−5.0
−2.5
0.0
2.5
5.0
4
2
0
−2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

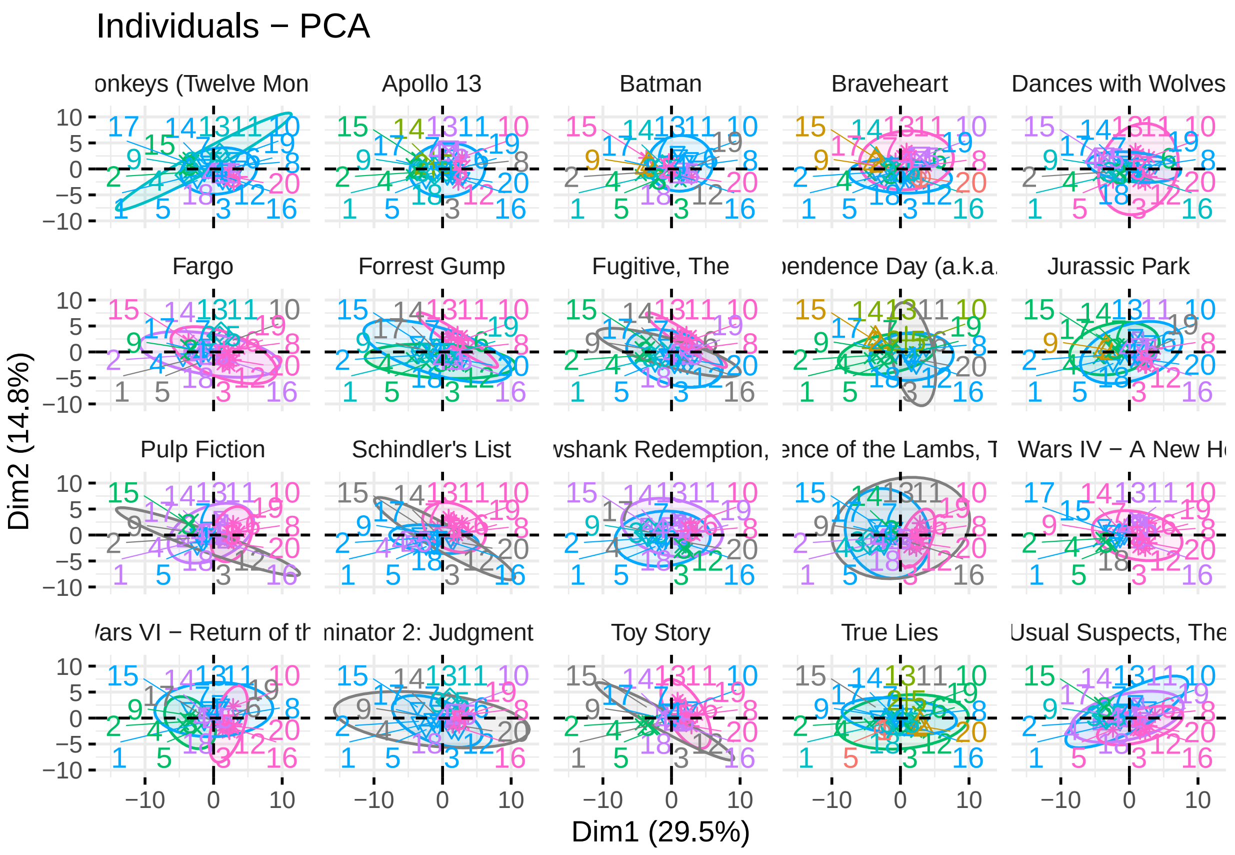


As we can see on the graph of variables, crime movies are below the x axis, drama movies are above it, and the only comedy movie, *True Lies* is completely on the y axis.

If we look at the graph of individuals and compare it to the previous dendrogram, users plotted on the graph have almost the same distance we found when building clusters.

We can get a different vision of the dimensions for each movie with the following code.

```r
library(factoextra)

# Dimensions plot by movie

fviz_pca_ind(pca_output, habillage = top_matrix[,-1], addEllipses = TRUE, repel = TRUE)
```



Each user is at the same place, but their colors vary depending on their ratings for each film. Most of the users loved *Star Wars : Episode IV*, but we can see at least three different dimensions for *Star Wars : Episode VI*. Ratings for *Independence day* and *True Lies* seem to be the lowest for this subset, and we can notice different dimensions for every movie. We have just analyzed 20 movies rated by 20 users, and we better understand now the complexity that we will face with more than 69,000 users and 10,000 movies. We have dealt with a very small subset and a few latent factors, but with such a large dataset, we can expect many more latent factors.

(NOTA BENE : the grey dimension represents NAs.)

Building a model with Matrix Factorization : the Recosystem package

In linear algebra, a $m * n$ matrix can be factorized as the product of two matrices as long as the first one has the same number of columns k as the number of rows of the second one.

$$R_{m*n} = P_{m*k} * Q_{k*n}$$

As we saw previously on the sparsity image, the dataset is filled with NAs when it is turned into a matrix containing ratings from users and for each movie. The 'Recosystem' package can help us predict those NAs based on observed values. Once we know the two above small matrices P and Q with their ratings, we can find the missing values for the R matrix. Moreover, for each rating the algorithm takes into account penalty parameters (ie, regularization) to optimize the rating and to avoid overfitting.

This package is an R wrapper of the LIBMF library, which is a parallelized library that enables to speed up computation thanks to multicore CPUs. The 'recosystem' is aimed for fast matrix factorization using parallel stochastic gradient descent. Stochastic means "random", and there's a part of randomness in the process when choosing the gradient. The gradient is a point of the optimization function the algorithm tries to solve for regularization. You can imagine this function as a parabola, it starts from a random point of it, and goes down along the slope to reach its lowest value. The process is iterative, so the gradient gets lower and lower (gradient descent) until convergence if there is one. For each predicted rating, the lowest value will be added as regularization.

Training the model

First, we randomly order the dataset. This ensures that the training set and test set are both random samples and that any biases in the ordering of the dataset are not retained in the samples. If we look carefully at the `edx` dataset, it is ordered by `userId`. So we will remove this bias.

```
# Randomly order the dataset

set.seed(69, sample.kind = "Rounding")

rows <- sample(nrow(edx))
random_edx <- edx[rows,]
```

We then split the data into train and test sets : 80% for the train set and 20% for the test set.

```
#Splitting in train and test sets

# Create Partition

test_index <- createDataPartition(y = random_edx$rating, times = 1, p = 0.8, list = FALSE)

train_set <- random_edx[test_index,]
test_set <- random_edx[-test_index,]
```

Before training the model, we must convert the train and test sets in new `Recosytem` objects with the same three columns we used previously to build the sparse matrix, that is to say `rating`, `userId` and `movieId`. We do this with the following code :

```
# Converting train and test sets in 'Recosystem' objects of class "DataSource"

library(recosystem)
library(recommenderlab) # We can use this package for the RMSE function.


train_reco <- with(train_set,data_memory(user_index = userId,
                                         item_index = movieId,
                                         rating = rating))


test_reco <- with(test_set,  data_memory(user_index = userId,
                                         item_index = movieId,
                                         rating = rating))

# Object class
class(train_reco)
```

We can then train the model and predict ratings with the following piece of code.

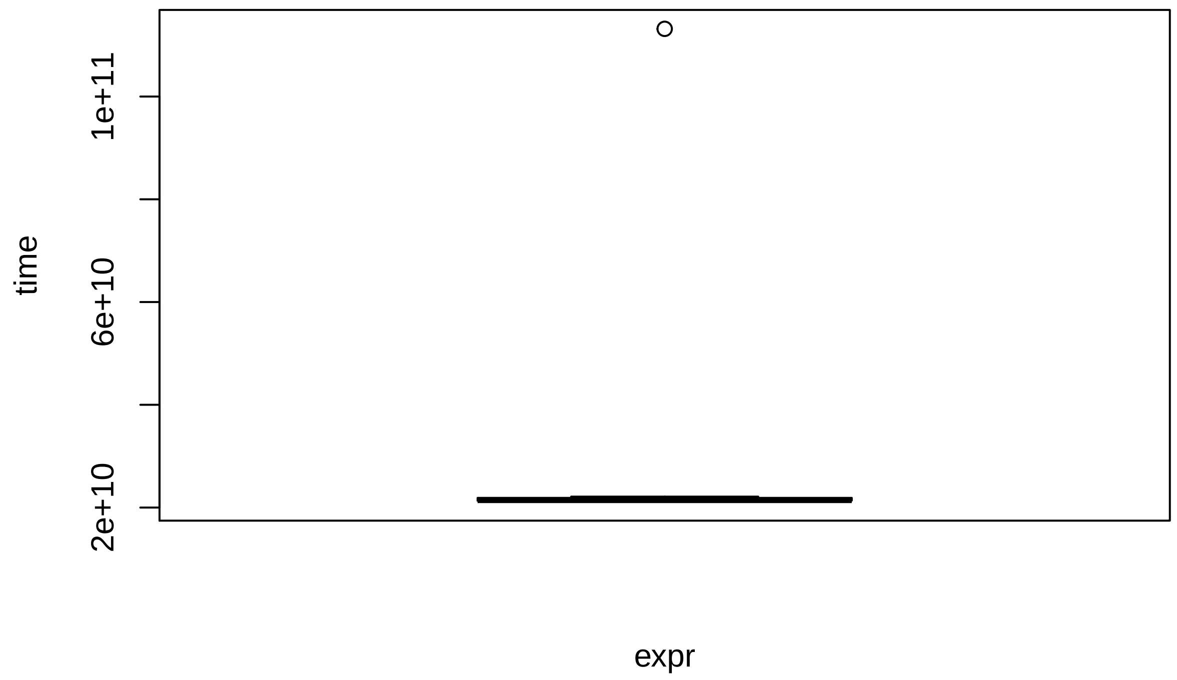
```r
# Simple training model
set.seed(69, sample.kind = "Rounding") # We must set seed because
# it's a randomized algorithm (ie, stochastic).

rec <- Reco()
rec$train(train_data = train_reco, opts = c(verbose = FALSE))

# Predicting ratings

test_pred <- rec$predict(test_reco,out_memory())

# RMSE

RMSE(test_set$rating,test_pred)
```

```
## [1] 0.833459
```

Our RMSE is 0.833 in just a few seconds. It is a pretty good result in a very short amount of time. We can calculate the processing time with the `microbenchmark` library and the following piece of code.

```r
library(microbenchmark)
set.seed(69, sample.kind = "Rounding")

computing_time <- microbenchmark(

  rec$train(train_data = train_reco),
  times = 10 # We calculate this 10 times and we'll get a data frame
  # of 10 different computing times.
)
```

```r
plot(computing_time)
```



In general, it took less than one minute to reach a RMSE around 0.82. Can we do better ? When training the model, we used the default values of different parameters. Here is the list of the main parameters :

- `dim` : it is the number of latent factors.
- `costp_l1` : this is L1 regularization parameter for user factors.
- `costp_l2` : this is L2 regularization parameter for user factors.
- `costq_l1` : this is L1 regularization parameter for item factors.
- `costq_l2` : this is L2 regularization parameter for item factors.
- `nfold` : this is the number of folds in cross validation. The default value is 5.
- `lrate` : it is the learning rate, which can be thought of as the step size in the gradient descent. If it's too high, we'll jump over the minima we try to reach.
- `niter` : it is the number of iterations. As we said, it is an iterative process in which the algorithm learns at each iteration while the gradient goes down to the minimum cost.
- `nthread` : it is the number of threads for parallel computing. We must be careful here because results may not be reproducible even if the seed is set when `nthread` > 1. In order to keep constant results, we'll keep `nthread` set to 1 for every test in this paper. It is indeed the default value.
- `nmf` : it performs non-negative matrix factorization if it's set to `TRUE`. Ratings on our dataset are only positive, so we'll tune this parameter to `TRUE` later. Default value is `FALSE`.
- `verbose` : show detailed information if it is `TRUE` (default value). We 'll use `FALSE` in the training process below.

(For more details about the `Recosystem` package or the optimization formula, please check the help file in RStudio or the links provided by the help page.)

The default values of each parameter are as below :

```
rec$train(train_data = train_reco, opts = c(dim = 10,
                                            costp_l1 = 0,
                                            costp_l2 = 0.1,
                                            costq_l1 = 0,
                                            costq_l2 = 0.1,
                                            lrate = 0.1,
                                            nfold = 5,
                                            niter = 20,
                                            nthread = 1,
                                            nmf = FALSE,
                                            verbose = TRUE))
```

Tuning parameters

The tuning of the parameters in the `opts` list of parameters can be done with the following `tune` option. However I did not succeed to run this code with the different values I wanted to try (R crashed), so we will tune the parameters with the default values of the 'tune' function. It then returns the best values with the lowest RMSE with `opt$min`. We will take the results of `opt$min` as a starting point to further tune each parameter. Tuning with the following code takes more time than the simple training we did above : around 2 hours with the default values. They are as follows :

```
set.seed(69, sample.kind = "Rounding")

# Tuning of parameters (WARNING ! LONG COMPUTING TIME : AROUND 2 HOURS)

opt = rec$tune(train_reco, opts = list(dim = c(10, 20),
                                       lrate = c(0.01, 0.1),
                                       costp_l1 = c(0, 0.1),
                                       costp_l2 = c(0.01, 0.1),
                                       costq_l1 = c(0, 0.1),
                                       costq_l2 = c(0.01, 0.1)
                                       ))
```

The best values are shown with the piece of code below :

```
# Best values :

opt$min
```

```
## $dim
## [1] 20
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0.01
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.1
##
## $lrate
## [1] 0.1
##
## $loss_fun
## [1] 0.8055157
```

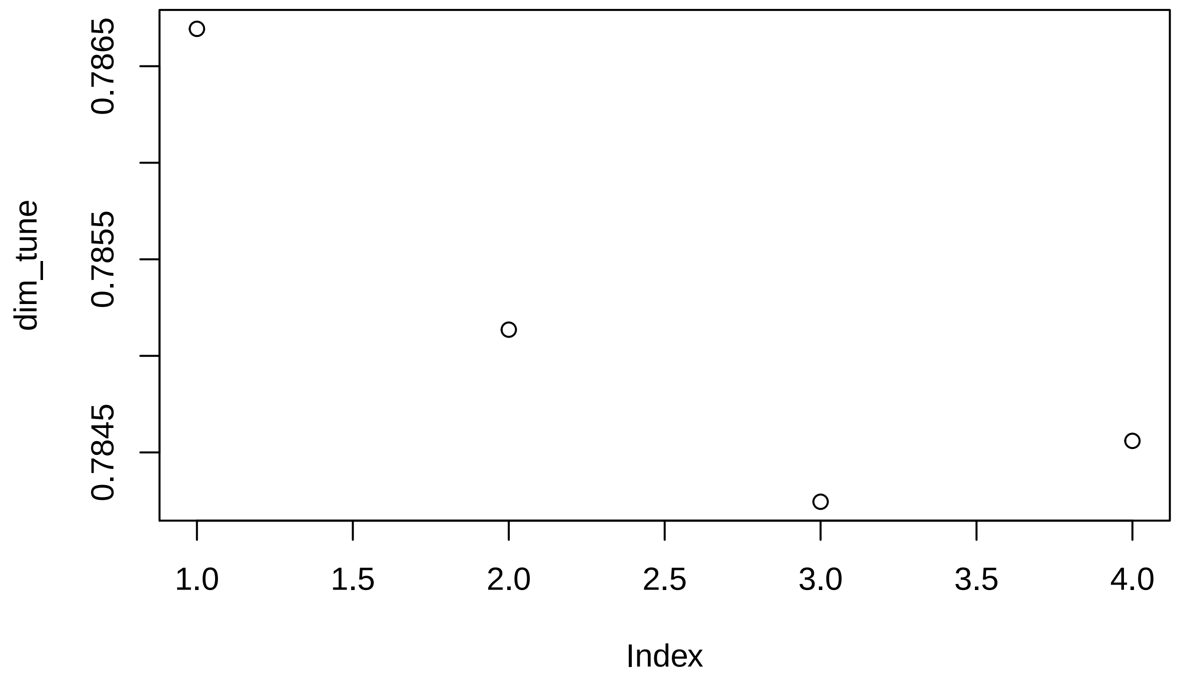
When tuning parameters with the best values above, our RMSE is close to 0.805. Can we still make it better ? As explained before, `rec$tune` may crash your computer if you input too many values. So we'll take the best values, and tune a bit more each parameter with different values. Of course, this is not perfect. The ideal method is to use the `tune` function that tries every possibility and every model with all the different values. That's why it takes so much time. Nevertheless, we'll try to do better bit by bit.

Let's start with the `dim` parameter. As we saw in the data exploration, the number of latent factors behind users and movies seems to be an important parameter. We found 4, 6 or 7 factors depending on the analysis. But it was for a very small subset of 20 movies and 20 users. Now we must train the whole dataset. According to the tuning results, 20 dimensions seem to give a better RMSE. We will try with more dimensions compared to the default values of the `tune` option.

```r
# Tuning 'dim' parameter : (It takes around 50 minutes)

d <- c(50, 100, 200, 300)

dim_tune <- sapply(d, function(dim){
  rec$train(train_data = train_reco,opts = c(dim = dim,
                                             costp_l2 = 0.01, # According to opt$min,
# this is the only best value that is not a default value
                                             verbose = FALSE))

  test_pred <- rec$predict(test_reco,out_memory())

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

We can plot the results like this.

```r
plot(dim_tune)
```

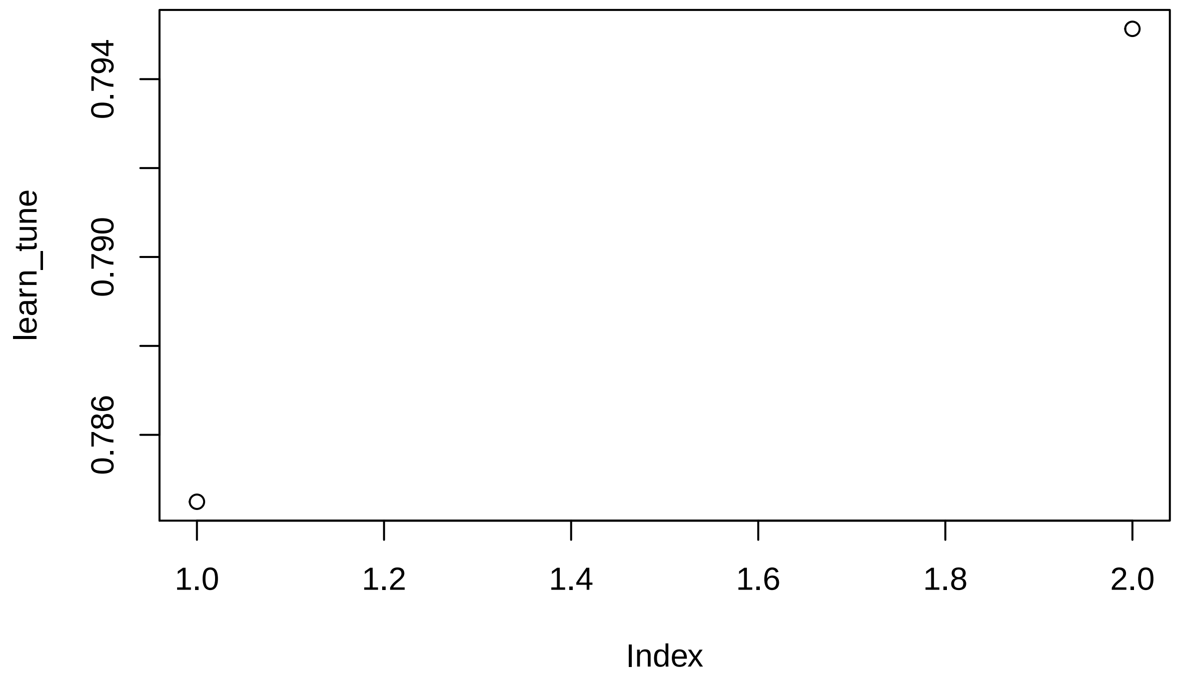


We get better results with 200 dimensions. We will use it for the next parameter tuning. Let's try to tune a bit more the learning rate.

```r
# Tuning learning rate 'lrate' : (Takes around 20 minutes)

l <- c(0.1,0.2)

learn_tune <- sapply(l, function(learn){
  rec$train(train_data = train_reco,opts = c(lrate= learn,
                                              dim = 200,# Value from the previous test
                                              costp_l2 = 0.01,
                                              verbose = FALSE))

  test_pred <- rec$predict(test_reco,out_memory())

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

Let's see the results.

```r
plot(learn_tune)
```



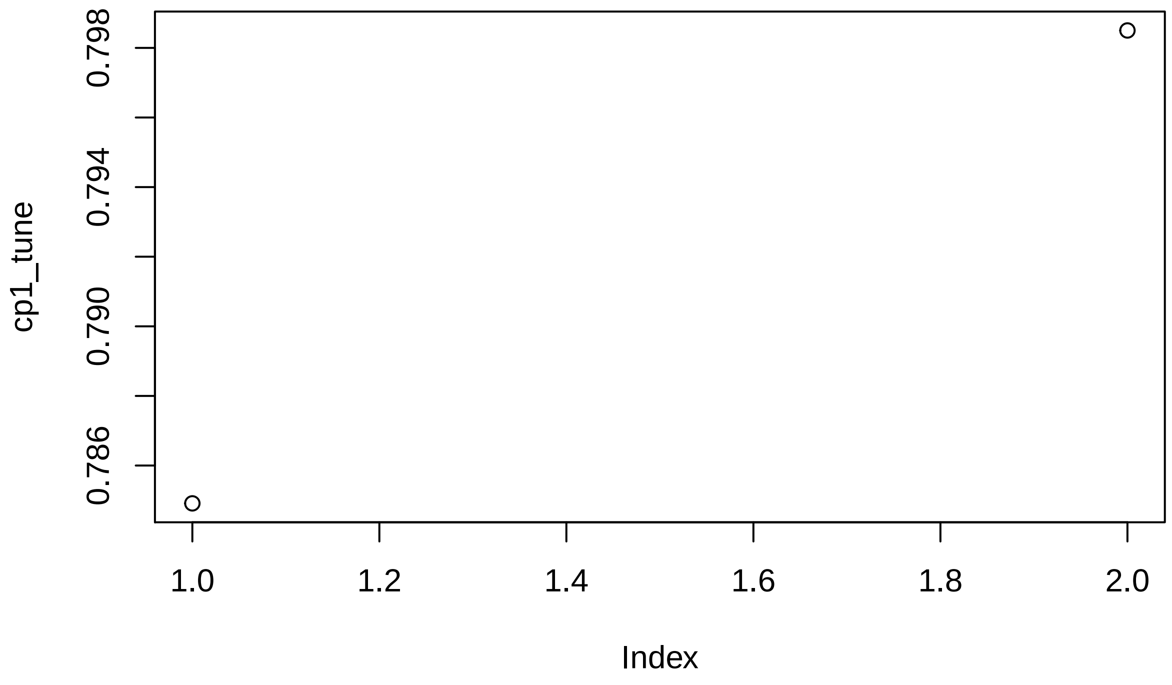
Here we can clearly see that our RMSE gets higher with a higher learning rate. If learning rate is too high, we jump over the minimum cost we want to reach during the gradient descent. So we'll keep using the best value we obtained with the tune option, which was 0.1.

What about the L1 P and Q costs ? The default values in the tune option were `c(0, 0.1)`. The best value selected for both costs was 0, but maybe 0.1 was a bit high as a regularization cost. Let's try with `c(0, 0.01)`.

```
# Tuning 'L1 p and q costs' :

# P1 cost :
c <- c(0,0.01)

cp1_tune <- sapply(c, function(cost){
  rec$train(train_data = train_reco,opts = c(dim = 200,
                                            costp_l1 = cost,
                                            costp_l2 = 0.01,
                                            verbose=F))

  test_pred <- rec$predict(test_reco,out_memory())

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

Here are the results :

```r
# Plot
plot(cp1_tune)
```



It seems that 0 is still the best value for `costp_l1`. We get the same results for `costq_l1`, so we won't show the code but the idea is the same. We keep 0 for both L1 costs.

What about iteration ? The number of iterations in the tune option was just 20. As we said, this algorithm is iterative and learns during gradient descent. For my first tries, bigger values for iteration resulted in better RMSE results in general, but the training model was not as tuned as now. Actually, I tried values between 5 and 500 for this model, and the best RMSE was reached with just 20 iterations, the default value. Again, more doesn't mean better.

To finish, we can add the 'non-negative matrix factorization' parameter in our model as our ratings are all positive values. We also set the folds for cross-validation to 10 instead of 5 (default value).

Our final training model looks like this :

```r
# Final training model

rec$train(train_data = train_reco, opts = c(dim = 200,
                                            costp_l1 = 0,
                                            costp_l2=0.01,
                                            costq_l1 = 0,
                                            costq_l2 = 0.1,
                                            lrate = 0.1,
                                            nfold = 10,
                                            niter = 20,
                                            nmf = TRUE,
                                            verbose = FALSE))
```

Predicting ratings

We can now predict results with the test set after training the model shown above.

```
# Predicting results on the test set :

test_pred <- rec$predict(test_reco, out_memory())### Takes a few minutes
```

The RMSE for the test set is 0.786.

```
# Calculating RMSE for the test set :

RMSE(test_set$rating, test_pred)
```

```
## [1] 0.7864296
```

Let's see how good are our estimates compared to the test set. Here are the best movies of the test set against the best movies according to our estimates.

```
# Best movies in test set
test_set %>%
  group_by(title) %>%
  summarize(n=n()) %>%
  arrange(desc(n)) %>%
  slice(1:10) %>%
  pull(title)
```

```
##  [1] "Forrest Gump (1994)"
##  [2] "Silence of the Lambs, The (1991)"
##  [3] "Pulp Fiction (1994)"
##  [4] "Jurassic Park (1993)"
##  [5] "Shawshank Redemption, The (1994)"
##  [6] "Fugitive, The (1993)"
##  [7] "Braveheart (1995)"
##  [8] "Terminator 2: Judgment Day (1991)"
##  [9] "Star Wars: Episode IV - A New Hope (a.k.a. Star Wars) (1977)"
## [10] "Apollo 13 (1995)"
```

Here are our predictions for the best movies.

```
# Best movies according to our predictions
test_set %>%
  mutate(predictions = test_pred) %>%
  arrange(desc(predictions)) %>%
  slice(1:10) %>%
  pull(title)
```

```
##  [1] "Rough Magic (1995)"
##  [2] "Gerry (2002)"
##  [3] "Passion of the Christ, The (2004)"
##  [4] "Cat from Outer Space, The (1978)"
##  [5] "Schindler's List (1993)"
##  [6] "Pulp Fiction (1994)"
##  [7] "Schindler's List (1993)"
##  [8] "Actor's Revenge, An (Yukinojo henge) (1963)"
##  [9] "Matrix, The (1999)"
## [10] "Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)"
```

We find common movies in both lists. We are not that far from the truth. Some predictions are different from the test set, but they still make sense and are popular movies.

Now let's take a look at the worst movies :

```r
# Worst movies in test set
test_set %>%
  arrange(rating) %>%
  slice(1:10) %>%
  pull(title)
```

```
##  [1] "Super Mario Bros. (1993)"
##  [2] "BASEketball (1998)"
##  [3] "First Wives Club, The (1996)"
##  [4] "Lost in Space (1998)"
##  [5] "Star Wars: Episode I - The Phantom Menace (1999)"
##  [6] "Village of the Damned (1995)"
##  [7] "Edward Scissorhands (1990)"
##  [8] "Little Miss Sunshine (2006)"
##  [9] "Raising Arizona (1987)"
## [10] "Anger Management (2003)"
```

And here are our predictions for the worst movies.

```r
# Worst movies according to our predictions
test_set %>%
  mutate(predictions = test_pred) %>%
  arrange(predictions) %>%
  slice(1:10) %>%
  pull(title)
```

```
##  [1] "From Justin to Kelly (2003)"
##  [2] "Simon Sez (1999)"
##  [3] "Eaten Alive (1976)"
##  [4] "Look Who's Talking Too (1990)"
##  [5] "Disaster Movie (2008)"
##  [6] "Disaster Movie (2008)"
##  [7] "From Justin to Kelly (2003)"
##  [8] "Ape, The (1940)"
##  [9] "Into the Arms of Strangers: Stories of the Kindertransport (2000)"
## [10] "PokÃ©mon Heroes (2003)"
```

Model performance

We tuned the model the best we could. The number of dimensions k or `dim` in our model was one of the most important parameters that reduced RMSE from 0.833 to 0.786. But as we tuned every parameter we could, it seems that we have reached a limit with this model. To improve our results and get an even lower RMSE, we should train different models with other methods and create an ensemble with the best models created.

However, we are close to our main objective, which is reaching a RMSE below 0.85. We will proceed to the final validation with this matrix factorization model.

Final Validation

It is now time to get the final predictions on the validation set and the final RMSE for this Capstone project.

```r
# Converting the 'validation' set in 'Recosystem' object

valid_reco <- with(validation, data_memory(user_index = userId,
                                           item_index = movieId,
                                           rating = rating))

# Getting predictions for the validation set :

valid_pred <- rec$predict(valid_reco, out_memory())
```

Here is the final RMSE for the validation set :

```r
# FINAL RMSE :

RMSE(validation$rating, valid_pred)
```

```
## [1] 0.7869614
```

Conclusion

Our final RMSE on the validation set is 0.7869, below the 0.85 target. Matrix factorization enabled us to do better, and faster. With just one model, we were able to get a low RMSE in just a few minutes. Edwin Chen's statement is true, and we can get good results with just a few models if they're well selected. Less is more.

During Data Exploration, we found latent factors, and thus the number of k for matrix factorization during tuning. Of course, there are still improvements to do. We did not construct a linear model with all the different variables, like `genres` or `timestamp`. There are also other calculation methods in matrix factorization like AdaGrad, which adapts the learning rate according to the sparsity of each parameter. It increases for sparser parameters, and decreases with less sparse ones.

During my research, I was very interested in parallel computing and packages like `H2O` and `SparklyR` to increase computation speed. It is possible to train algorithms with `H2O` very quickly, whereas `SparklyR` was low and setting a good configuration was a bit hard. But it has a lot of algorithms, more than `H2O`, and these packages can be really useful for large datasets like the Movielens dataset we used for this project.

The `recommenderlab` offers also a solution for matrix factorization and has many tools to build a recommendation system and to suggest movies to users.

We focused on one method in this paper that gives good results. Less is more, but there is still a lot of work to do in order to build a better model and a lower RMSE.