MovieLens Report

Henrique Casellato

22/07/2021

Introduction

Project goal:

The purpose of this project is to create a reliable and light movie recommendation system. Instead of the complete MovieLens dataset, this project will use its 10M version to facilitate the computation.

Dataset general information:

The dataset consists of approximately 10 million ratings of different users to different films (the exact numbers are displayed further in the data exploration section). Also, the dataset provides the films genres and time stamps, in case they are valuable to the analysis.

Project planning:

To achieve good performance in the analysis, this project will follow these steps:

Methods and Analysis - Investigate the data set to understand more about it and eventually find a preferred method.
Data preparation - Clean and prepare the data set for modelling.
Data modelling - Training and testing the chosen method to prove its effectiveness.
Method validation - Final testing and results report.

Project structure:

The project separation into three independent scripts has the purpose of only running the necessary and intended code. These divisions are:

Data exploration [data_exploration.R] - This script has the purpose of investigating the data for insights in further analysis.
Experiment [experiment.R] - This script has the purpose of showing the effectiveness of the LIBMF method.
Final validation [prediction_final.R] - This script has the intention of displaying the final script.

Methods and Analysis

Data Loading

The next code is supposed to load the required packages for data exploration:

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

```r
library(tidyverse)
library(caret)
library(data.table)
```

The next piece of code will load the train (edx) and validation sets for final testing:

```r
if(!exists("edx") & !exists("validation")){
  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")

  # if using R 4.0 or later:
  movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(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)`
  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)
}
```

Data Exploration

For this project, my research was able to identify some potential algorithms: collaborative filtering, popularity based, random based, and matrix factorization. I understand that there are more modern ways of recommending movies, such as deep learning and neural networks, but I must assume that those are far above my current knowledge, and their study would consume a time that I, unfortunately, don't have. To select one for testing, it is vital to explore our data and understand how it behaves.

Firstly, it is essential to find a favorite algorithm by eliminating others that might be less efficient. Having this in mind, a few interesting data points from our data exploration, such as the distribution of ratings per user, and movie, can provide helpful information. The methods UBCF (User-Based Collaborative Filtering) and IBCF (Item-Based Collaborative Filtering) seem prominent. They are part of a larger group of recommender techniques called *collaborative filtering*, meaning the process of filtering for patterns using the collaboration of other entities (users or items, for example).

To evaluate later data exploration, it is a must to compute the user and movie distributions (most relevant variables):

```r
# Distribution of votes per user:
d_user <- edx %>% group_by(userId) %>%
                  summarize(count = n())

# Distribution of votes per movie:
d_movie <- edx %>% group_by(movieId, title) %>%
                   summarize(count = n())
```

In the book *Introduction to Data Science* by professor Irizarry, the suggested package for PCA and SVD, fantastic methods sadly not examined in this report, is the **recommenderlab**. In the package documentation, UBCF, IBCF, and other valuable techniques are supported. Also, a version of the 100k MovieLens dataset is present. When analysing other sources of testing materials, it becomes clear the differences between the dataset from the package and the one we are supposed to work with: Ours is a hundred times bigger and possibly not as clean. The functions presented in the recommenderlab package work with a *realRatingMatrix*, but do not mention how it handles its likely enormous final dimensions. Finally, another obstacle related to the size of the matrix is its sparsity, more explicitly seen in the table:

	Users	Movies
Total:	69878	10677
Mean:	128.7966885	842.9385595
Median:	62	122

In detail, what can be interpreted from these numbers is that:
-Half of the users voted in less than 62 movies, 0.58% of the total number of movies;
-The average number of votes per user is ≈ 129, which means a share of users voted a lot;
-Half of the movies were voted less than 122 times.

With the next piece of code, we can see graphically the effects of the means and medians:

```r
if(!require(grid))      install.packages("grid",      repos = repo)
if(!require(gridExtra)) install.packages("gridExtra", repos = repo)

library(grid)
library(gridExtra)

# Users
h_user <- d_user %>% ggplot(aes(count)) +
          scale_x_log10() +
          geom_histogram(bins = 50, fill = "azure4") +
          ggtitle("Distribution of Ratings per User") +
          ylab("Number of Users") +
          xlab("Number of Ratings") +
          theme_bw(base_size = 12, base_family = "times")

# Movies
```

```
h_movie <- d_movie %>% ggplot(aes(count)) +
           scale_x_log10() +
           geom_histogram(bins = 50, fill = "azure4") +
           ggtitle("Distribution of Ratings per Movie") +
           ylab("Number of Movie") +
           xlab("Number of Ratings") +
           theme_bw(base_size = 12, base_family = "times")

# Grid
grid.arrange(h_user,h_movie, nrow = 2)
```

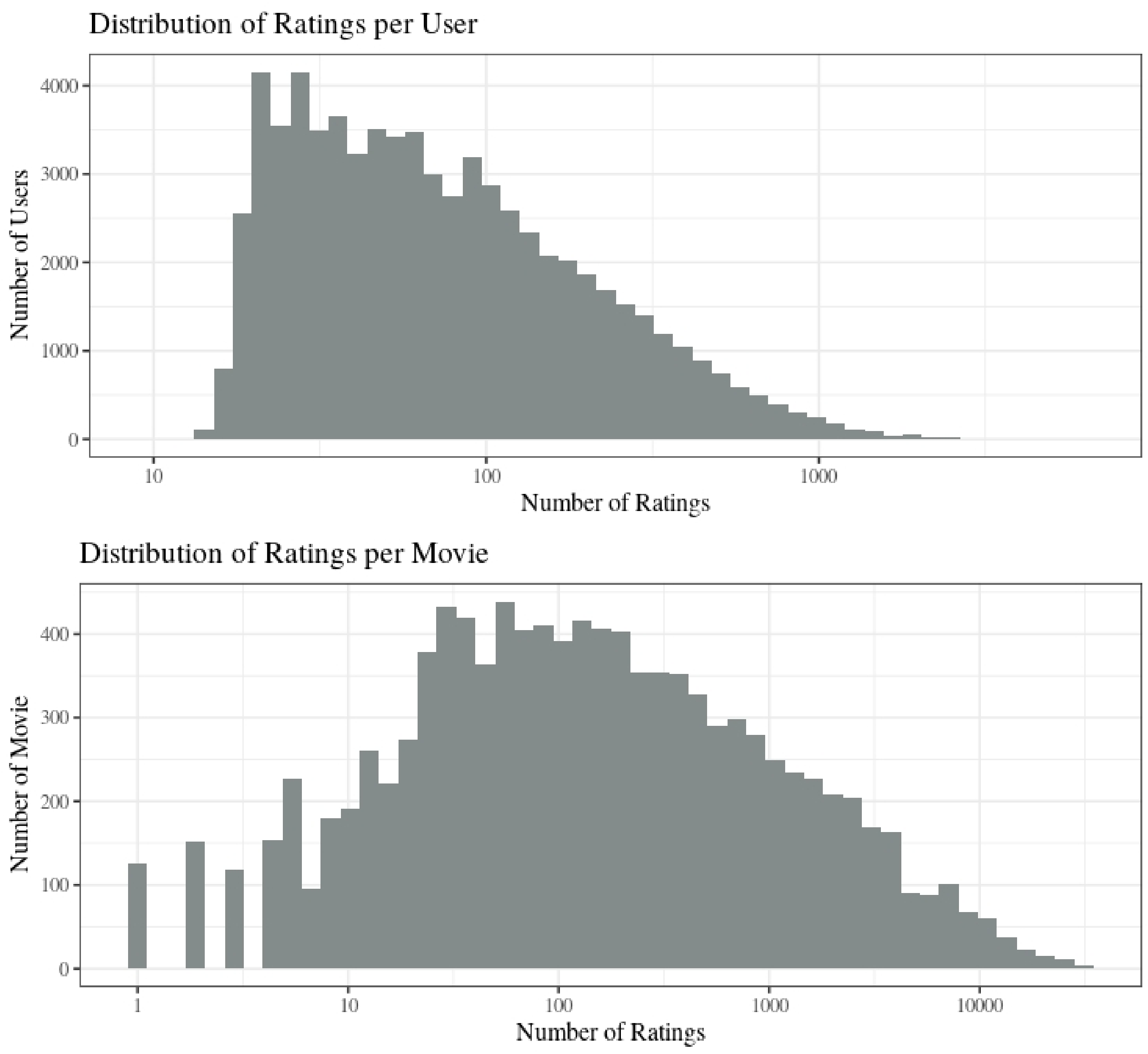

Figure 1: Distributions

Solution

Having all of this in mind, it becomes troublesome to work with the recommenderlab package. In detail, the future matrix will be composed of almost nothing, consuming the memory considerably. The solution to this problem is to select less data, but how much data could be sacrificed without compromising accuracy? Beyond this problem, in supplementary studies, the reported loss functions of the package's Movie Lens data set is not satisfactory. Therefore, recommenderlab is not an option.

When researching recommender systems for sparse matrices, a method becomes applicable: **LIBMF**. This Matrix Factorization library is a *tool for approximating an incomplete matrix using the product of two matrices in a latent space*. With a variety of package options, a more user-friendly one can help substantially the predictions: the *recosystem* package.

Data Preparation

Data Loading

The next code is supposed to load the last required package:

```r
repo <- "http://cran.us.r-project.org"

if(!require(recosystem))     install.packages("recosystem",     repos = repo)
library(recosystem)
```

Data Arrangement

First, it is necessary to create the train and test sets based on the edx data set:

```r
set.seed(2021, sample.kind = "Rounding")

# Data sets
test_index <- createDataPartition(edx$rating, times = 1,
                                  p = 0.2, list = FALSE)
train_set <- edx[-test_index]
test_set  <- edx[test_index]

# Making sure all movies and users appear in both data sets
test_set <- test_set %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")
```

Then, it is necessary to "translate" the data table to something recosystem functions can interpret:

```r
set.seed(2021, sample.kind = "Rounding")

# Translating the train and test sets to a recosystem set
train_data <-  with(train_set, data_memory(user_index = userId,
                                           item_index = movieId,
                                           rating     = rating))

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

Also, before going to modelling, a loss function (RMSE) must be created:

```r
# RMSE function
rmse <- function(true, predicted){sqrt(mean((true - predicted)^2))}
```

The *Root Mean Square Error* is the standard deviation of the prediction errors, i.e. it is the distance between observed and predicted values.

Data Modelling

After data preparation, a recommender model is created. Even if not required, it is advantageous to tune parameters and select the finest.

```r
# Creating a recommender model:
r <- Reco()

# Tuning parameters:
# This process can take a while =(
tune <- r$tune(train_data, opts = list(dim      = c(10, 20, 30),
                                        lrate    = c(0.1, 0.2),
                                        costp_l2 = c(0.01, 0.1),
                                        costq_l2 = c(0.01, 0.1),
                                        nthread  = 4,
                                        niter    = 10))
```

The tune parameters used in later training can be found running the code:

```r
tune$min
```

```
## $dim
## [1] 30
##
## $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] 0.8062575
```

After done tuning, we are prepared for training and predicting:

```r
# Training model:
r_train <- r$train(train_data, opts = c(tune$min, nthread = 1, niter = 30))
```

```
## iter      tr_rmse          obj
##    0       0.9954   1.0068e+07
##    1       0.8780   8.0780e+06
##    2        0.8464   7.5038e+06
##    3        0.8252   7.1639e+06
##    4        0.8079   6.9091e+06
##    5        0.7946   6.7249e+06
##    6        0.7839   6.5880e+06
##    7        0.7748   6.4733e+06
##    8        0.7670   6.3806e+06
##    9        0.7602   6.3055e+06
##   10        0.7542   6.2407e+06
##   11        0.7488   6.1860e+06
##   12        0.7438   6.1349e+06
##   13        0.7393   6.0916e+06
##   14        0.7352   6.0548e+06
##   15        0.7314   6.0187e+06
##   16        0.7279   5.9889e+06
##   17        0.7246   5.9603e+06
##   18        0.7217   5.9342e+06
##   19        0.7189   5.9123e+06
##   20        0.7163   5.8907e+06
##   21        0.7140   5.8719e+06
##   22        0.7117   5.8527e+06
##   23        0.7097   5.8370e+06
##   24        0.7078   5.8220e+06
##   25        0.7060   5.8081e+06
##   26        0.7043   5.7956e+06
##   27        0.7027   5.7835e+06
##   28        0.7013   5.7720e+06
##   29        0.6998   5.7611e+06
```

```r
# Prediction model:
r_predict <- r$predict(test_data, out_memory())
```

With all set, we calculate the RMSE and find out that the LIBMF method is really efficient:

```r
# RMSE results:
result <- rmse(test_set$rating, r_predict)
result
```

```
## [1] 0.7897031
```

Method Validation

Given that the method worked really well on the partitioned data set, it should work on the validation set. Therefore, the steps will be repeated but with different training and testing sets:

```r
set.seed(2021, sample.kind = "Rounding")
```

```
## Warning in set.seed(2021, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```r
# Translating the train and test sets to a recosystem set
edx_data <- with(edx, data_memory(user_index = userId,
                                  item_index = movieId,
                                  rating     = rating))

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

# Creating a recommender model:
r_v <- Reco()

# Tuning parameters:
# This process can take a while =(
tune_v <- r$tune(edx_data, opts = list(dim      = c(10, 20, 30),
                                        lrate    = c(0.1, 0.2),
                                        costp_l2 = c(0.01, 0.1),
                                        costq_l2 = c(0.01, 0.1),
                                        nthread  = 4,
                                        niter    = 10))
```

Again, the tune parameters to be used are:

```
tune_v$min
```

```
## $dim
## [1] 30
##
## $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] 0.798001
```

Training and predicting:

```
# Training model:
r_train_v <- r$train(edx_data, opts = c(tune$min, nthread = 1, niter = 30))
```

```
## iter      tr_rmse          obj
##    0       0.9730   1.2033e+07
##    1       0.8727   9.8887e+06
##    2       0.8390   9.1706e+06
##    3       0.8178   8.7585e+06
##    4       0.8023   8.4785e+06
##    5       0.7908   8.2869e+06
##    6       0.7812   8.1358e+06
##    7       0.7732   8.0181e+06
##    8       0.7663   7.9197e+06
##    9       0.7603   7.8395e+06
##   10        0.7550   7.7673e+06
##   11        0.7503   7.7121e+06
##   12        0.7460   7.6611e+06
##   13        0.7422   7.6144e+06
##   14        0.7386   7.5760e+06
##   15        0.7354   7.5391e+06
##   16        0.7324   7.5079e+06
##   17        0.7297   7.4810e+06
##   18        0.7272   7.4534e+06
##   19        0.7248   7.4323e+06
##   20        0.7226   7.4088e+06
##   21        0.7206   7.3903e+06
##   22        0.7187   7.3713e+06
##   23        0.7170   7.3561e+06
##   24        0.7154   7.3396e+06
##   25        0.7138   7.3261e+06
##   26        0.7124   7.3129e+06
##   27        0.7110   7.3008e+06
##   28        0.7098   7.2896e+06
##   29        0.7086   7.2783e+06
```

```r
# Prediction model:
r_predict_v <- r$predict(validation_data, out_memory())
```

After that, we calculate the final RMSE:

```r
# RMSE results:
result_v <- rmse(validation$rating, r_predict_v)
result_v
```

```
## [1] 0.7810258
```

The results shown in the final validation are excellent! With that, we complete the final tests.

Conclusion

Results

The report's goal was to analyze, test, and evaluate preferred methods in the quest for light and reliable recommender systems. The chosen data set to work with was the 10M Movie Lens, and the best algorithm was the LIBMF. The expected (training and testing on partitioned edx data set) and observed(training on edx data and tested on validation set) final RMSEs were:

	RMSE:
Expected:	0.7897031
Observed:	0.7810258

Limitations and Future work

Machine learning algorithms demand a lot of processing velocity, time, and memory use, requiring plenty of study and investigation prior to code formulation. Nevertheless, this is a price to pay for good results. There are other algorithms not tested in this report that might work well in sparse matrices, for example, NOMAD, and GraphChi. Also, further studies in the applicability of neural networks and deep learning on recommender systems are necessary since they might produce more solid results.

References

Hahsler, Michael. “Recommenderlab - Lab for Developing and Testing Recommender Algorithms - R Package.” RDocumentation, Rcran, 26 Feb. 2021, www.rdocumentation.org/packages/recommenderlab/versions/0.2-7.

Hahsler, Michael. “Recommenderlab: A Framework for Developing AndTesting Recommendation Algorithms.” RDocumentation, 2016, cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf.

Irizarry, Rafael A. Introduction to Data Science. Leanpub, 2021.

Luo, Shuyu. “Intro to Recommender System: Collaborative Filtering.” Medium, Towards Data Science, 6 Feb. 2019, towardsdatascience.com/intro-to-recommender-system-collaborative-filtering-64a238194a26.

Qiu, Yixuan. “Recosystem Package.” RDocumentation, May 2021, www.rdocumentation.org/packages/recosystem/versions/0.4.5.