Collaborative Filtering Recommender System

Veerasak Kritsanapraphan

February 15, 2020

Table of Contents

## Introduction

This is an introduction to building Recommender Systems using R. The major CRAN approved package available in R with developed algorithms is called recommenderlab by Michael Hahsler. Latest [documentation](https://cran.r-project.org/web/packages/recommenderlab/recommenderlab.pdf) and a [vignette](https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf) are both available for exploration. The code examples provided in this exploratory analysis came primarily through the material on Collaborative Filtering algorithms from this package, explored in the book [*Building a Recommendation System with R*](https://smile.amazon.com/Building-Recommendation-System-Suresh-Gorakala/dp/1783554495/ref=sr_1_1?ie=UTF8&qid=1507314554&sr=8-1&keywords=building+a+recommendation+system+R), by Suresh K. Gorakala and Michele Usuelli.

## Collaborative Filtering

Under *user-based collaborative filtering*, this memory-based method works under the assumption that users with similar item tastes will rate items similarly. Therefore, the missing ratings for a user can be predicted by finding other similar users (a neighborhood). Within the neighborhood, we can aggregate the ratings of these neighbors on items unknown to the user, as basis for a prediction. We’ll explore this one in detail in sections below.

An inverted approach to nearest neighbor based recommendations is *item-based collaborative filtering*. Instead of finding the most similar users to each individual, an algorithm assesses the similarities between the items that are correlated in their ratings or purchase profile amongst all users.

## Load recommenderlab

library(dplyr)  
library(ggplot2)  
library(recommenderlab)

Some of the preloaded datasets that come with recommenderlab for learning and exploring.

help(package = "recommenderlab")  
datasets\_available <- data(package = "recommenderlab")  
datasets\_available$results[,4] # titles

[1] "Jester dataset (5k sample)"   
[2] "Jester dataset (5k sample)"   
[3] "Anonymous web data from www.microsoft.com"  
[4] "MovieLense Dataset (100k)"   
[5] "MovieLense Dataset (100k)"

We’ll work with the already available *Movielense* dataset.

data(MovieLense) # loads dataset  
class(MovieLense)

[1] "realRatingMatrix"  
attr(,"package")  
[1] "recommenderlab"

movie\_r <- MovieLense   
remove(MovieLense)

It is formatted as a realRatingMatrix class already, an object class created within recommenderlab for efficient storage of user-item ratings matrices. It’s been optimized for storing sparse matrices, where almost all of the elements are empty. As an example, compare the object size of *Movielense* as a realRatingMatrix vs. a matrix.

library(pryr)  
object\_size(movie\_r)

1.41 MB

object\_size(as(movie\_r, "matrix"))

12.8 MB

The realRatingMatrix for this particular dataset is about 9 times more efficient in conserving memory than a traditional matrix object.

## Exploratory Analysis of the Movielense data

Below is a preview of the ratings matrix of users and their ratings. Rows represent the user indexes.

getRatingMatrix(movie\_r[1:10, 1:4])

10 x 4 sparse Matrix of class "dgCMatrix"  
 Toy Story (1995) GoldenEye (1995) Four Rooms (1995) Get Shorty (1995)  
1 5 3 4 3  
2 4 . . .  
3 . . . .  
4 . . . .  
5 4 3 . .  
6 4 . . .  
7 . . . 5  
8 . . . .  
9 . . . .  
10 4 . . 4

For a particular user such as User 1, they gave an average rating of 3.61. 10 of the movies rated by them are shown below.

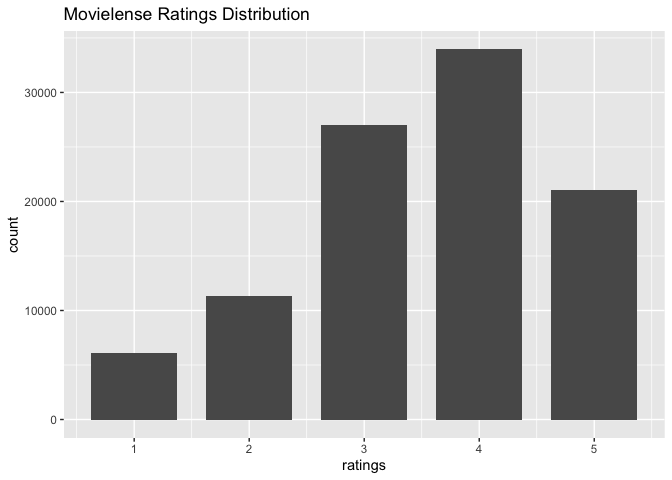
as(movie\_r[1, ], "list")[[1]][1:10]

The getRatings function returns the non-missing ratings values from the matrix as a numeric vector. The following histogram shows the distribution of all movie ratings in the dataset. We can see that ratings typically skew higher, centered around a median rating of 4.

summary(getRatings(movie\_r))

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 1.00 3.00 4.00 3.53 4.00 5.00

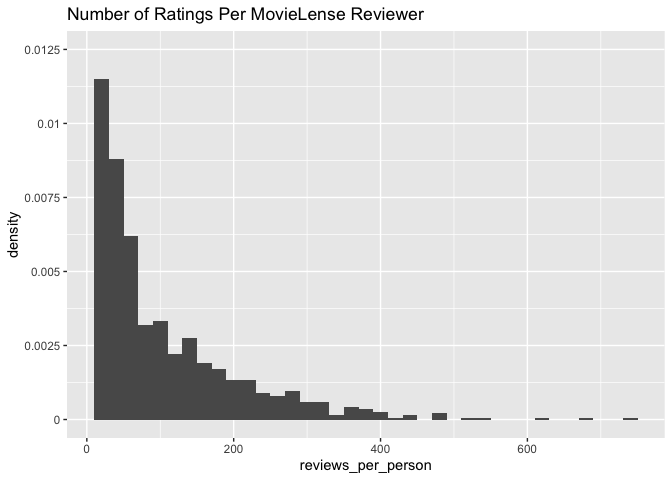
data.frame(ratings = getRatings(movie\_r)) %>%  
 ggplot(aes(ratings)) + geom\_bar(width = 0.75) +  
 labs(title = 'Movielense Ratings Distribution')



summary(rowCounts(movie\_r))

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 19.0 32.0 64.0 105.4 147.5 735.0

rowCounts(movie\_r) %>%  
 data.frame(reviews\_per\_person = .) %>%  
 ggplot(aes(x = reviews\_per\_person)) +   
 geom\_histogram(aes(y = ..density..), binwidth = 20) +  
 scale\_y\_continuous(limits = c(0,.0125),   
 breaks = seq(0, .0125, by = 0.0025),  
 labels = seq(0, .0125, by = 0.0025)) +  
 labs(title = 'Number of Ratings Per MovieLense Reviewer')

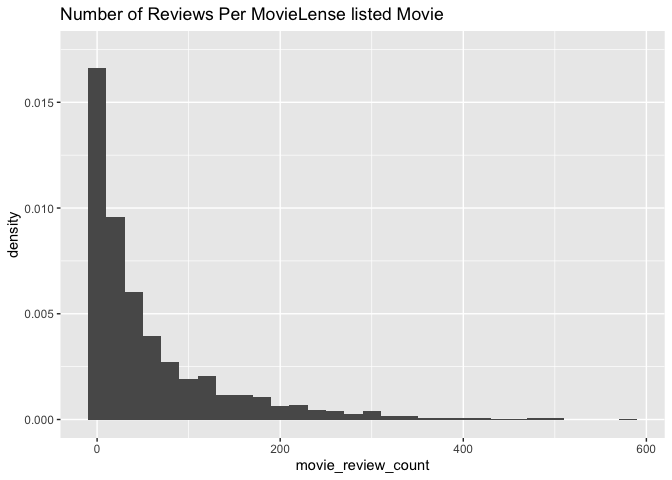


With a median number of reviews of 27 per user and 1664 different movies available to rate, we know that the data is sparse with a lot of users not having rated most of the movies available.

summary(colCounts(movie\_r))

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 1.00 7.00 27.00 59.73 80.00 583.00

colCounts(movie\_r) %>%  
 data.frame(movie\_review\_count = .) %>%  
 ggplot(aes(x = movie\_review\_count)) +   
 geom\_histogram(aes(y = ..density..), binwidth = 20) +  
 scale\_y\_continuous(limits = c(0,.0175)) +  
 labs(title = 'Number of Reviews Per MovieLense listed Movie')



## Recommender Algorithms Available

The recommender algorithms are stored in a registry object called recommenderRegistry. We can get a look at the different models based on the different matrix types.

names(recommenderRegistry$get\_entries())

[1] "ALS\_realRatingMatrix" "ALS\_implicit\_realRatingMatrix"   
 [3] "ALS\_implicit\_binaryRatingMatrix" "AR\_binaryRatingMatrix"   
 [5] "IBCF\_binaryRatingMatrix" "IBCF\_realRatingMatrix"   
 [7] "LIBMF\_realRatingMatrix" "POPULAR\_binaryRatingMatrix"   
 [9] "POPULAR\_realRatingMatrix" "RANDOM\_realRatingMatrix"   
[11] "RANDOM\_binaryRatingMatrix" "RERECOMMEND\_realRatingMatrix"   
[13] "RERECOMMEND\_binaryRatingMatrix" "SVD\_realRatingMatrix"   
[15] "SVDF\_realRatingMatrix" "UBCF\_binaryRatingMatrix"   
[17] "UBCF\_realRatingMatrix"

Since our matrix is a real ratings matrix, we’ll call the algorithms available for working on numeric ratings based review data as stored in the realRatingMatrix. Here, I’ve pulled the descriptions of each of the algorithms available for working with real user ratings data.

vapply(recommenderRegistry$get\_entries(dataType = "realRatingMatrix"),   
 '[[',   
 FUN.VALUE = "character",   
 "description")

ALS\_realRatingMatrix   
 "Recommender for explicit ratings based on latent factors, calculated by alternating least squares algorithm."   
 ALS\_implicit\_realRatingMatrix   
 "Recommender for implicit data based on latent factors, calculated by alternating least squares algorithm."   
 IBCF\_realRatingMatrix   
 "Recommender based on item-based collaborative filtering."   
 LIBMF\_realRatingMatrix   
"Matrix factorization with LIBMF via package recosystem (https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html)."   
 POPULAR\_realRatingMatrix   
 "Recommender based on item popularity."   
 RANDOM\_realRatingMatrix   
 "Produce random recommendations (real ratings)."   
 RERECOMMEND\_realRatingMatrix   
 "Re-recommends highly rated items (real ratings)."   
 SVD\_realRatingMatrix   
 "Recommender based on SVD approximation with column-mean imputation."   
 SVDF\_realRatingMatrix   
 "Recommender based on Funk SVD with gradient descend (https://sifter.org/~simon/journal/20061211.html)."   
 UBCF\_realRatingMatrix   
 "Recommender based on user-based collaborative filtering."

## Exploring User-based Collaborative Filtering

In the algorithms registry, the last algorithm provided in the listing is the one we’ll use to explore user-based collaborative filtering (UBCF) to fit the UBCF algorithm to the realRatingMatrix of MovieLense reviews data. Information about this algorithm per the registry:

ubcf\_model\_description <- tail(recommenderRegistry$get\_entries(dataType = "realRatingMatrix"), 1)  
ubcf\_model\_description

$UBCF\_realRatingMatrix  
Recommender method: UBCF for realRatingMatrix  
Description: Recommender based on user-based collaborative filtering.  
Reference: NA  
Parameters:  
 method nn sample normalize  
1 "cosine" 25 FALSE "center"

There are 4 parameters to account for with this model as described above:

* **method**: this is the type of similarity metric to calculate similarity between users real ratings profile. Cosine similarity, Pearson correlation coefficient, and Jaccard similarity are available options. The first two are not good options if using unary ratings, but work well for this scenario.
* **nn**: this parameter sets the neighborhood of most similar users to consider for each user profile. the ratings profiles of the k nearest neighbors will be the basis for making predictions on a users unrated items profile.
* **sample**: a logical value to indicate whether the data should be sampled for train/test. Probably best to explicitely set a reproducible seed and sample the data before running the model.
* **normalize**: how to normalize real ratings provided by different users. This is crucially important b/c all users have a different bias in how they tend to rate items. This can be done by passing a value to this parameter inside the algorithm or applied to the matrix before any modeling too. See ?normalize for additional details.

### Normalize the data

User rating *zero mean centering* will be used for modeling, where each user’s vector of ratings is subtracted by its own mean to center the mean at zero. Z-scoring is an alternative method available too that additionally divides each user’s rating by its standard deviation.

\*\* maybe visualize the distribution of user ratings here too after normalization vs. before normalization \*\*

movie\_r\_norm <- normalize(movie\_r, method = 'center') # for visual comparison purposes.   
remove(movie\_r\_norm)

### How the UBCF algorithm works

1. Using cosine similarity, figure out how similar each user is to each other.
2. for each user, identify the *k* most similar users. Here, *k* parameter was the 10 most similar users who rated common items most similarly.
3. Per item, average the ratings by each user’s *k* most similar users.
4. weight the average ratings based on similarity score of each user whose rated the item. Similarity score equals weight, or
5. use any of the pythagorean averages, as suits the business case (arithmetic, geometric, harmonic)
6. Select a Top-N recommendations threshold.

### Set Up a Model Training & Evaluation Scheme

train\_proportion <- .75  
# shouldn't keep n rec. items > min(rowCounts(movie\_r))  
min(rowCounts(movie\_r))

[1] 19

items\_per\_test\_user\_keep <- 10  
# What's a good rating for a binary split?  
good\_threshold <- 4

# Building a Recommender System with R by Gorakala and Usuelli. Ch.4 pp 77 - 83  
set.seed(123)  
model\_train\_scheme <- movie\_r %>%  
 evaluationScheme(method = 'split', # single train/test split  
 train = train\_proportion, # proportion of rows to train.  
 given = items\_per\_test\_user\_keep, # shouldn't keep n rec. items > min(rowCounts(movie\_r))  
 goodRating = good\_threshold, # for binary classifier analysis.  
 k = 1)

Having set our evaluationScheme and stored it in an object called *model\_train\_scheme*, we can fit a UBCF recommender system model.

# Building a Recommender System with R by Gorakala and Usuelli. Ch.4 pp 84  
model\_params <- list(method = "cosine",  
 nn = 10, # find each user's 10 most similar users.  
 sample = FALSE, # already did this.  
 normalize = "center")  
  
model1 <- getData(model\_train\_scheme, "train") %>% #only fit on the 75% training data.  
 Recommender(method = "UBCF", parameter = model\_params)

### Evaluate Predictive Performance

# 5.5 - 5.6. Evaluation of predicted ratings in recommenderLab vignette. can use n = for predicting TopN or type = for predicting ratings.  
model1\_pred <- predict(model1, getData(model\_train\_scheme, "known"), type = "ratings")  
model1\_pred

236 x 1664 rating matrix of class 'realRatingMatrix' with 390344 ratings.

Now we can test the predicion error of model 1 on the *unknown* test user ratings using the calcPredictionAccuracy method. Three metrics for ratings test error are available: root mean squared error, mean squared error, or mean absolute error. The results below focus on RMSE with the errors calculated per test user on their *unknown* data.

test\_error <- calcPredictionAccuracy(model1\_pred, getData(model\_train\_scheme, "unknown"), byUser = TRUE)  
head(test\_error)

RMSE MSE MAE  
2 0.9912398 0.9825563 0.8050128  
3 1.6050600 2.5762175 1.2392562  
4 0.9340536 0.8724561 0.7898605  
10 0.5833831 0.3403358 0.4679950  
12 0.8065585 0.6505365 0.6456743  
13 1.5821156 2.5030898 1.2967690

Let’s visualize the distribution of the average RMSE of new predicted ratings for each 236 test user.

test\_user\_error\_data <- data.frame(user\_id = as.numeric(row.names(test\_error)),  
 rmse = test\_error[, 1],  
 predicted\_items\_cnt = rowCounts(getData(model\_train\_scheme, "unknown"))) # coerce from matrix to df.  
  
test\_user\_error\_data %>%  
 ggplot(aes(rmse)) +   
 geom\_histogram(aes(y = ..density..)) +  
 labs(title = 'RMSE on Predicted Recommendations per Test User',  
 subtitle = "User-Based Collaborative Filtering using Cosine Similarity")

