Capstone_movielens

KateBWilliams

April 22, 2020

Introduction:

The MovieLens data-set contains information on user's ratings of various films over time. This information can be used in order to predict the rating a user might give to a film, which would facilitate movie recommendations on various platforms(i.e. Netflix, Hulu, etc.)

The objective of this analysis is to establish a predictive algorithm that minimizes the root-mean-squared-error (RMSE). Minimizing the RMSE is a quantitative measure of the accuracy of the predictions produced by the algorithm, i.e. the smaller the RMSE, the better the algorithm.

Methods and Analysis:

The movieLens data set was downloaded from its source and cleaned according to the following code:

```
################################
# Create edx set, validation set
################################

# Note: this process could take a couple of minutes

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)
```

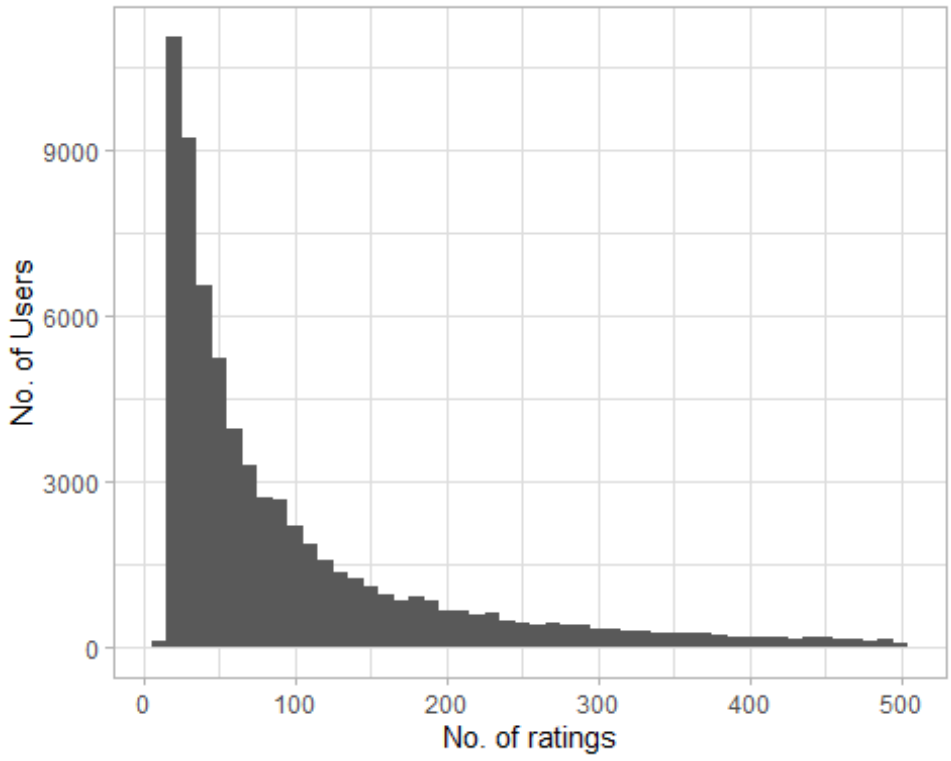
Data exploration:

Users and ratings:

The “average” user has submitted ~129 ratings, although there is wide variability in user rating activity.

```
##        avg median  st.dev.
## 1 128.7967     62 195.0602
```

Although there are some “power users” with reviews in the thousands, the majority of users have reviews in the 10s-100s range.



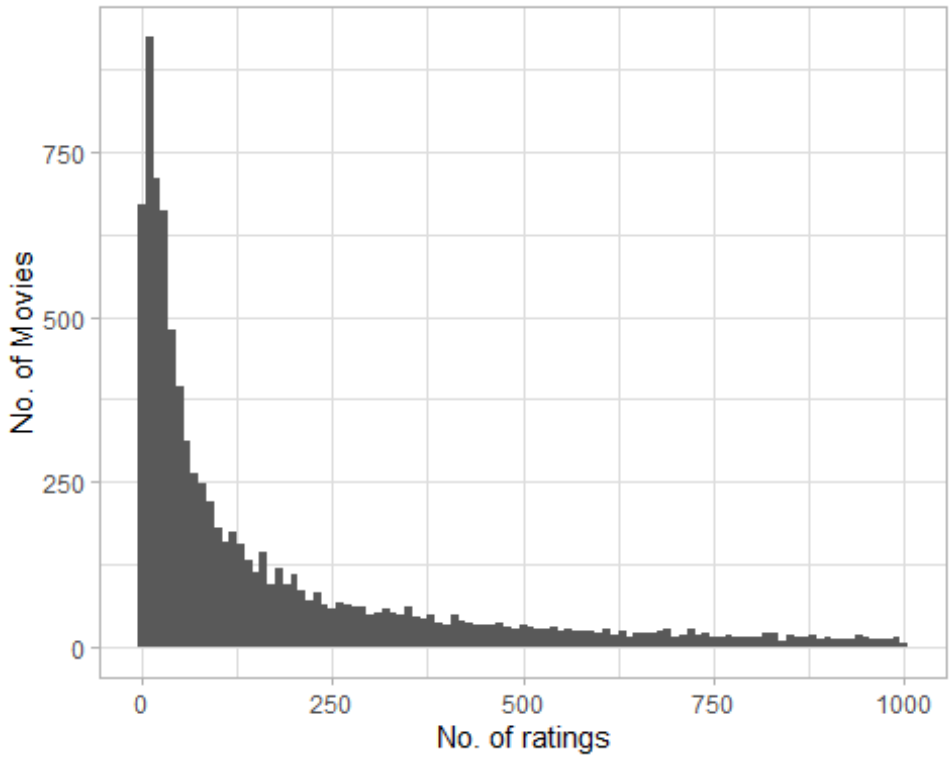
Movies and ratings:

The “average” movie has 843 ratings, but much like user ratings, there is a large distribution of rating numbers among movies.

```
##        avg median  st.dev.
## 1 842.9386    122 2238.481
```

The code below shows how a number of movies have under 100 ratings total.

```
edx %>% group_by(movieId) %>% summarize(count = n())%>% filter( count <=
1000) %>% ggplot(aes(count)) + geom_histogram(binwidth = 10) + labs(x= "No.
of ratings", y = "No. of Movies")+ theme_light()
```



Needless to say, the users and movies who have higher numbers of ratings will be easier to predict, with movies and/or users are the very low end of the number of ratings will be difficult to predict.

recommenderlab

The `recommenderlab` library provides a useful set of algorithms designed for predictive recommendations.

```
library(recommenderlab)
```

In order to assess the `edx` data, it must first be converted into a format that is recognized by the packages in the library, namely the `ratingMatrix` format. This is accomplished by first selecting the `userId`, `movieId`, and `ratings` data columns, then coercing them into a `realRatingMatrix`.

```
temp <- edx %>% select(userId, movieId, rating)

m<- as(temp, "realRatingMatrix")
```

Once the correct format has been achieved, and we need to create an `evaluationScheme` in order to run an `evaluate` function.

```
scheme <- evaluationScheme(m[1:200], method = "split", train = 0.9, k=1,
given = 10)
```

This method creates the correct object class, as well as a training set for evaluation (note that this is now a subset training set within the original `edx` training set). Now we

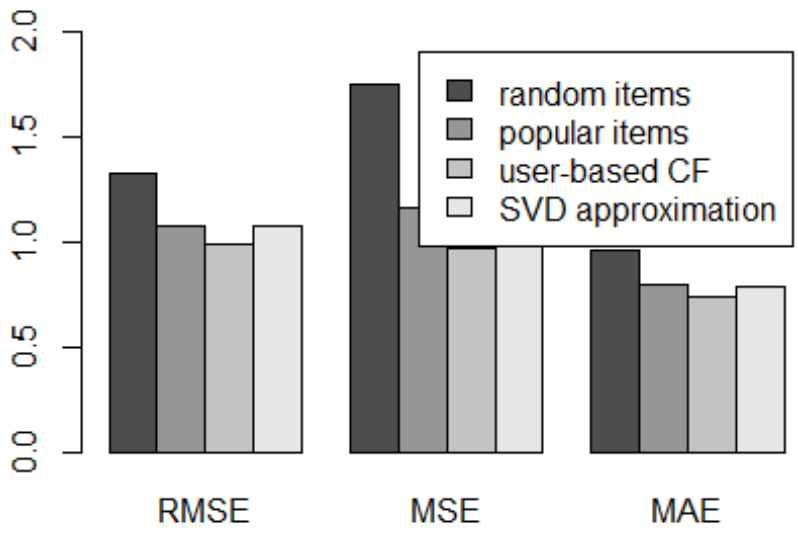
can evaluate different algorithms available for recommendation. Note that the default settings of the algorithms include their own normalization based on a ‘center’ technique. Otherwise, a normalization step would have been included on the data set before this evaluation step.

```
algorithms <- list(
  "random items" = list(name="RANDOM", param=NULL),
  "popular items" = list(name="POPULAR", param=NULL),
  "user-based CF" = list(name="UBCF", param=list(nn=50)),
  "SVD approximation" = list(name="SVD", param=list(k = 50))
)

results <- evaluate(scheme, algorithms, type = "ratings")

## RANDOM run fold/sample [model time/prediction time]
##   1  [0.01sec/0.14sec]
## POPULAR run fold/sample [model time/prediction time]
##   1  [0.04sec/0.02sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [0.02sec/0.42sec]
## SVD run fold/sample [model time/prediction time]
##   1  [1.33sec/0.11sec]

plot(results, ylim= c(0,2))
```



We can see from the plot which algorithm appears to have the lowest RMSE (the user-based CF). However, none of these have an RMSE of lower that 0.94 in the testing

environment, which suggests these algorithms are not very promising without considerable individual tuning.

Matrix factorization: `recosystem`

Another favored recommendation algorithm system is the matrix factorization system. This is available as a library that can be installed for R.

```
library(recosystem)
```

For this analysis, we must create our own train and test sets, and get them into the correct format class for the S4 based `recosystem` functions to recognize.

First, we create training and test sets out of the `edx` data set. This is done similarly to how the `edx` and `validation` sets were created.

The next step is to coerce the data into an object of class `DataSource`. We use the `data_memory()` function to achieve this. Note that much like the `recommenderlab`, we need to have a sparse matrix. We achieve this by inputting the three necessary elements, `useriD`, `movieId`, and `rating`.

```
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))
```

And then create the model object:

```
r <- recosystem::Reco()
```

Now we can assess options for tuning the algorithm we would like to train. Note that this a randomized algorithm, requiring a `set.seed()`. The value `123` is used as a default suggested by the `recosystem` documentation.

```
set.seed(123, sample.kind = "Rounding")

## Warning in set.seed(123, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used

opts <- r$tune(train_data, opts = list(nthread = 4))
```

Note that the tuning options here are using the the default options, except the `nthread` parameter set to 4 to increase processing speed. However, the **processing time may take more than an hour**.

From this tuning step, we can now train the algorithm against the training data.

```
r$train(train_data, opts = opts$min)

## iter      tr_rmse          obj
##    0       0.9766  1.0914e+007
##    1       0.8776  8.9952e+006
```

```
##    2       0.8469  8.3860e+006
##    3       0.8263  8.0134e+006
##    4       0.8110  7.7568e+006
##    5       0.7993  7.5722e+006
##    6       0.7900  7.4326e+006
##    7       0.7823  7.3177e+006
##    8       0.7758  7.2284e+006
##    9       0.7702  7.1508e+006
##   10       0.7654  7.0910e+006
##   11       0.7613  7.0376e+006
##   12       0.7578  6.9926e+006
##   13       0.7546  6.9519e+006
##   14       0.7519  6.9189e+006
##   15       0.7495  6.8895e+006
##   16       0.7472  6.8605e+006
##   17       0.7453  6.8388e+006
##   18       0.7435  6.8184e+006
##   19       0.7419  6.8003e+006
```

You can see a series of tr_rmse values approaching 0.74.

We can now use the trained algorithm to predict values in the testing data. The `out_memory()` function allows the output of the results as R objects.

```
y_hat <- r$predict(test_data, out_memory())
```

The final step is to calculate the RMSE. We have to do that by creating our own function for RMSE first.

```
RMSE <- function(observed, predicted){
  sqrt(mean((observed - predicted)^2))
}
```

Then we can call the `RMSE` function for our prediction vs. the test values.

```
## [1] 0.7901819
```

This gives a very promising RMSE.

Results:

In order to verify that the Matrix Factorization method gives a satisfactorily low RMSE, we need to train on the full `edx` set, and validate against the `validation` set. Note that this is the only time that we use the `validation` set in our analysis.

First, both data sets must be converted to `DataSource` class objects.

```
edx_final <-  with(edx, data_memory(user_index = userId, item_index =
movieId, rating = rating))
valid_final <- with(validation, data_memory(user_index = userId, item_index =
movieId, rating = rating))
```

Then the algorithm is trained on the `edx_final` data set, preditions are made on the `valid_final` data set, and the RMSE calculated against the original `validation` set.

```
r$train(edx_final, opts = opts$min)

## iter      tr_rmse          obj
##    0       0.9663  1.1884e+007
##    1       0.8729  9.8880e+006
##    2       0.8418  9.1956e+006
##    3       0.8228  8.8157e+006
##    4       0.8080  8.5517e+006
##    5       0.7968  8.3509e+006
##    6       0.7880  8.2100e+006
##    7       0.7806  8.0912e+006
##    8       0.7743  7.9956e+006
##    9       0.7690  7.9217e+006
##   10       0.7645  7.8526e+006
##   11       0.7607  7.7976e+006
##   12       0.7575  7.7533e+006
##   13       0.7546  7.7136e+006
##   14       0.7521  7.6796e+006
##   15       0.7499  7.6499e+006
##   16       0.7479  7.6226e+006
##   17       0.7461  7.5985e+006
##   18       0.7445  7.5769e+006
##   19       0.7430  7.5617e+006

y_hat <- r$predict(valid_final, out_memory())

print("Final RMSE:")

## [1] "Final RMSE:"

RMSE(validation$rating, y_hat)

## [1] 0.7876489
```

Therefore, the final RMSE is `0.7876`.

Conclusion

As shown by the two different packages available for ratings-based recommendation systems, the Matrix Factorization underlying the `recosystem` package offers clearly superior predictions, as quantified by its low RMSE on the 'validation' data set. Note that the default options for the tuning parameters were used; it is possible with additional optimization of the tuning parameters, the RMSE could be reduced even further.

This processing time variable was also a major consideration for not pursuing additional tuning of the `recommenderlab` algorithms. Not surprisingly, the 'random' algorithm produced a poor RMSE well above a value of 1. Interestingly, the 'UBCF' algorithm gave the best RMSE amongst the four that were compared, suggesting that the predictive value of `recommenderlab` was influenced most by users rating movies (i.e. those with the highest number of ratings).

For both analysis methods, given the limitations of processing power on a typical PC laptop, iterative tuning optimization to further reduce the RMSE is outside the scope of this analysis.

References and Links:

https://cran.r-project.org/web/packages/recommenderlab/index.html

https://cran.r-project.org/package=recosystem