MovieLens Recommendation System (Rough Draft)

Kaylee Robert Tejeda

9/11/2021

Introduction and Overview

(an introduction/overview/executive summary section that describes the dataset and summarizes the goal of the project and key steps that were performed)

One common application of machine learning techniques is the creation of what are known as "recommendation systems". These are mathematical models that take as input the parameters associated with a particular observation and predicts the value of a specific unobserved variable generated by that observation. The more accurate this prediction is, the more useful and valuable it is to specific industries.

Goal Summary (objective, motivation, etc.)

Our goal is to create such a recommendation algorithm using a data set containing movie ratings provided by GroupLens known as the "10M version of the MovieLens dataset", and is available here: https://grouplens.org/datasets/movielens/10m/ . We seek to develop a model for movie rating given 5 base variables (user ID, movie title, movie year, movie genre, and review date).

Outline of Steps Taken

The performance of the algorithm was evaluated by RMSE, or root mean square error in stars between predicted rating and actual rating. The MovieLens data set contains 10000054 rows, 10677 movies, 797 genres and 69878 users.The edx dataset (train set) contains 9,000,055 ratings of 10,677 movies by 69,878 users and consisted of 90% of the original benchmark MovieLens 10M dataset while validation set contains 999,999 ratings.

Data

Each observation in the dataset has the following variables associated:

"userId", "movieId", "rating", "timestamp", "title", "genres"

- `userId`: Unique user ID number.
- `movieId`: Unique movielens movie ID number.
- `rating`: User-provided ratings on a 5-star scale with half-star increments starting from 0.5
- `timestamp`: Time of user-submitted review in epoch time,
- `title`: Movie titles including year of release as identified in IMDB
- `genres`: A pipe-separated list of film genres

Analysis

(a methods/analysis section that explains the process and techniques used, including data cleaning, data exploration and visualization, insights gained, and your modeling approach)

Preparation

```
############################################################
# Create edx set, validation set (final hold-out test set)
############################################################

# Note: this process could take a couple of minutes

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

```
## Loading required package: tidyverse
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

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

```
## Loading required package: caret
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

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

```
## Loading required package: data.table
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
```

```
## The following object is masked from 'package:purrr':
##
##     transpose
```

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

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

```
tic("setup")

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(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)`
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

```
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)
```

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```
edx <- rbind(edx, removed)

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

toc()
```

```
## setup: 136.277 sec elapsed
```

Exploration / Visualization

A first attempt:

```
# Assuming the data has already been prepared as per the instructions...

# Use the training set to build a model with several of the models available
# from the caret package. We will test out all of the following models in this
# exercise:
```

```
models <- c("glm", "lda",  "naive_bayes",  "svmLinear",
            "gamboost",  "gamLoess", "qda",
            "knn", "kknn", "loclda", "gam",
            "rf", "ranger",  "wsrf", "Rborist",
            "avNNet", "mlp", "monmlp",
            "adaboost", "gbm",
            "svmRadial", "svmRadialCost", "svmRadialSigma")


# Loss function that computes the RMSE for vectors of ratings and their corresponding predictors:

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

# Find the average of all ratings in the test set

mu_hat <- mean(edx$rating)
mu_hat
```

```
## [1] 3.512465
```

```
#  First attempt:  Naive RMSE

naive_rmse <- RMSE(validation$rating, mu_hat)
naive_rmse
```

```
## [1] 1.061202
```

```
#  [1] 1.061202   (not very good at all)

#  Keep track of RMSE results in a tibble ??

#rmse_results <- tibble(method = "Just the average", RMSE = naive_rmse)

# Compute least squares estimate the quick way

mu <- mean(edx$rating)
movie_avgs <- edx %>%
group_by(movieId) %>%
summarize(b_i = mean(rating - mu))

#  Calculate RMSE after accounting for movie bias

predicted_ratings <- mu + validation %>%
left_join(movie_avgs, by='movieId') %>%
pull(b_i)
RMSE(predicted_ratings, validation$rating)
```

```
## [1] 0.9439087
```

```
#  [1] 0.9439087   (at least this is under unity, better)

#  Now train based on user bias as well, quick method:

user_avgs <- edx %>%
```

```
left_join(movie_avgs, by='movieId') %>%
group_by(userId) %>%
summarize(b_u = mean(rating - mu - b_i))

#  Check the new RMSE

predicted_ratings <- validation %>%
left_join(movie_avgs, by='movieId') %>%
left_join(user_avgs, by='userId') %>%
mutate(pred = mu + b_i + b_u) %>%
pull(pred)
RMSE(predicted_ratings, validation$rating)
```

```
## [1] 0.8653488
```

```
#  [1] 0.8653488  (getting close, full points for RMSE < 0.86490)


#  Now try a Peanlized Least Squares estimate with lambda = 3

lambda <- 3
mu <- mean(edx$rating)
movie_reg_avgs <- edx %>%
group_by(movieId) %>%
summarize(b_i = sum(rating - mu)/(n()+lambda), n_i = n())

#  Check RMSE

predicted_ratings <- validation %>%
left_join(movie_reg_avgs, by = "movieId") %>%
mutate(pred = mu + b_i) %>%
pull(pred)
RMSE(predicted_ratings, validation$rating)
```

```
## [1] 0.9438538
```

```
#  [1] 0.9438538   (much worse than before...)


#  Now optimize lambda

lambdas <- seq(0, 10, 0.25)

mu <- mean(edx$rating)
just_the_sum <- edx %>%
group_by(movieId) %>%
summarize(s = sum(rating - mu), n_i = n())
rmses <- sapply(lambdas, function(l){
predicted_ratings <- validation %>%
left_join(just_the_sum, by='movieId') %>%
mutate(b_i = s/(n_i+l)) %>%
mutate(pred = mu + b_i) %>%
pull(pred)
return(RMSE(predicted_ratings, validation$rating))
})
```

```
lambdas[which.min(rmses)]
```

```
## [1] 2.5
```

```
which.min(rmses)
```

```
## [1] 11
```

```
#  [1] 2.5
#  [1] 0.9438521  (still not very good)


#  Now regularize on user and movie effects using lambda

lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
mu <- mean(edx$rating)
b_i <- edx %>%
group_by(movieId) %>%
summarize(b_i = sum(rating - mu)/(n()+l))
b_u <- edx %>%
left_join(b_i, by="movieId") %>%
group_by(userId) %>%
summarize(b_u = sum(rating - b_i - mu)/(n()+l))
predicted_ratings <-
validation %>%
left_join(b_i, by = "movieId") %>%
left_join(b_u, by = "userId") %>%
mutate(pred = mu + b_i + b_u) %>%
pull(pred)
return(RMSE(predicted_ratings, validation$rating))
})

lambdas[which.min(rmses)]
```

```
## [1] 5.25
```

```
#  [1] 5.25  (optimized lambda for both user and movie effects)

min(rmses)
```

```
## [1] 0.864817
```

```
#  [1] 0.864817   (optimial RMSE, full points for RMSE < 0.86490, YAY!!!)
```

This used the entire edx set, which is not correct. This set needs to be split and used for both training AND testing of the algorithm, and then the validation set is only used to calculate RMSE at the very end.

Attempt #2:

```
# Split the edx data set into a training set and a test set

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

# Loss function that computes the RMSE for vectors of ratings and their corresponding predictors:
# Ignores NA values, which wind up in the predicted_ratings for some reason
```

```
RMSE <- function(true_ratings, predicted_ratings){
#sqrt(mean((true_ratings - predicted_ratings)^2, na.rm=TRUE))
sqrt(mean((true_ratings - predicted_ratings)^2))
}

# Find the average of all ratings in the training set

mu_hat <- mean(train_set$rating)

#  First attempt:  Naive RMSE

naive_rmse <- RMSE(test_set$rating, mu_hat)

#  Keep track of RMSE results in a tibble

rmse_results <- tibble(method = "Just the average", RMSE = naive_rmse)

#  Factor in movie bias the quick way

movie_avgs <- train_set %>%
group_by(movieId) %>%
summarize(b_i = mean(rating - mu_hat))

#  Calculate RMSE after accounting for movie bias

predicted_ratings <- mu_hat + test_set %>%
left_join(movie_avgs, by='movieId') %>%
pull(b_i)

bi_rmse <- RMSE(predicted_ratings, test_set$rating)

rmse_results <- bind_rows(rmse_results,
                          tibble(method="Movie Effect Model",
                                     RMSE = bi_rmse))

#  Now account for user bias as well, quick method:

user_avgs <- train_set %>%
left_join(movie_avgs, by='movieId') %>%
group_by(userId) %>%
summarize(b_u = mean(rating - mu_hat - b_i))

#  Check the new RMSE

predicted_ratings <- test_set %>%
left_join(movie_avgs, by='movieId') %>%
left_join(user_avgs, by='userId') %>%
mutate(pred = mu_hat + b_i + b_u) %>%
pull(pred)

bi_bu_rmse <- RMSE(predicted_ratings, test_set$rating)

rmse_results <- bind_rows(rmse_results,
```

```
                          tibble(method="Movie and User Effects Model",
                                     RMSE = bi_bu_rmse))

#  Now try a Peanlized Least Squares estimate with lambda = 5

lambda <- 5
movie_reg_avgs <- train_set %>%
group_by(movieId) %>%
summarize(b_i = sum(rating - mu_hat)/(n()+lambda), n_i = n())

#  Check RMSE

predicted_ratings <- test_set %>%
left_join(movie_reg_avgs, by = "movieId") %>%
mutate(pred = mu_hat + b_i) %>%
pull(pred)

lambda_rmse <- RMSE(predicted_ratings, test_set$rating)


rmse_results <- bind_rows(rmse_results,
                          tibble(method="Penalized LSE Model on movie bias, lambda=5",
                                     RMSE = lambda_rmse))

#  Now optimize lambda using cross-validation

# Set lambda sequence
# Increase third value for quicker run time
# Decrease third value for finer precision

lambdas <- seq(0, 10, 0.25)

# Calculate RMSEs for each lambda
# Use peanlized least squares on both user and movie bias
# (good method to try from experience/book lecture)

RMSEs <- sapply(lambdas, function(l){           # do the following for each lambda:
b_i <-  train_set %>%                         # calculate bias for each movie from training set
group_by(movieId) %>%
summarize(b_i = sum(rating - mu_hat)/(n()+l))
b_u <- train_set %>%                          # calculate bias for each user from training set
left_join(b_i, by="movieId") %>%
group_by(userId) %>%
summarize(b_u = sum(rating - b_i - mu_hat)/(n()+l))
predicted_ratings <-                      # calculate predictions on test set
test_set %>%
left_join(b_i, by = "movieId") %>%
left_join(b_u, by = "userId") %>%
mutate(pred = mu_hat + b_i + b_u) %>%
pull(pred)
return(RMSE(predicted_ratings, test_set$rating))    # calculate RMSE for a particular lambda
})
```

```
# Best RMSE is the smallest calculated, we want to minimize error

lambdas_rmse <- min(RMSEs)

# Best lambda is the one that generated the smallest error

lambda_best <- lambdas[which.min(RMSEs)]

# Best RMSE to more digits

# Make it look pretty in a data frame / tibble

#options(pillar.sigfig=8)  #increase default digits in tibble

#tibble(mu = mu, lambda=lambda_best, RMSE=rmse_best)

final_method <- paste("Optimized Penalized LSE Model with lambda =", as.character(lambda_best), sep=" ")

rmse_results <- bind_rows(rmse_results,
                          tibble(method=final_method,
                                     RMSE = lambdas_rmse))

####################################
## Done training!
## Run selected method against the validation set
############################################


#  Check the new RMSE
l <- lambda_best

b_i <- train_set %>%
group_by(movieId) %>%
summarize(b_i = sum(rating - mu_hat)/(n()+l))
```

```
## `summarise()` has grouped output by 'movieId'. You can override using the `.groups` argument.
```

```
b_u <- train_set %>%
left_join(b_i, by="movieId") %>%
group_by(userId) %>%
summarize(b_u = sum(rating - b_i - mu_hat)/(n()+l))
```

```
## `summarise()` has grouped output by 'userId'. You can override using the `.groups` argument.
```

```
final_predicted_ratings <-
validation %>%
left_join(b_i, by = "movieId") %>%
left_join(b_u, by = "userId") %>%
mutate(pred = mu_hat + b_i + b_u) %>%
pull(pred)

RMSE(final_predicted_ratings, validation$rating)
```

```
## [1] NA
```

Insights

- `NAs in predicted_ratings`: I was getting NAs in the predicted_ratings output. This lead to an investigation of outliers that helped reduce the overall RMSE.
- `Lambda can be refined step-wise`: Lambda can be optimized to two decimal places using 30 tests instead of 100. This ultimately proved to be worthless, but interesting.

```
#####
#####

#lambdas <- seq(1, 10, 1)
#RMSEs <- sapply(lambdas, function(l){
#b_i <-  train_set %>% group_by(movieId) %>% summarize(b_i = sum(rating - mu_hat)/(n()+l))
#b_u <- train_set %>% left_join(b_i, by="movieId") %>% group_by(userId) %>% summarize(b_u #= sum(rating
#predicted_ratings <- test_set %>% left_join(b_i, by = "movieId") %>% left_join(b_u, by = #"userId") %>
#return(RMSE(predicted_ratings, test_set$rating))
#})

#lambda_best <- lambdas[which.min(RMSEs)]

#lambdas <- seq(lambda_best-1, lambda_best+1, 0.1)
#RMSEs <- sapply(lambdas, function(l){
#b_i <-  train_set %>% group_by(movieId) %>% summarize(b_i = sum(rating - mu_hat)/(n()+l))
#b_u <- train_set %>% left_join(b_i, by="movieId") %>% group_by(userId) %>% summarize(b_u #= sum(rating
#predicted_ratings <- test_set %>% left_join(b_i, by = "movieId") %>% left_join(b_u, by = #"userId") %>
#return(RMSE(predicted_ratings, test_set$rating))
#})

#lambda_best <- lambdas[which.min(RMSEs)]


#lambdas <- seq(lambda_best-0.1, lambda_best+0.1, 0.01)
#RMSEs <- sapply(lambdas, function(l){
#b_i <-  train_set %>% group_by(movieId) %>% summarize(b_i = sum(rating - mu_hat)/(n()+l))
#b_u <- train_set %>% left_join(b_i, by="movieId") %>% group_by(userId) %>% summarize(b_u = sum(rating
#predicted_ratings <- test_set %>% left_join(b_i, by = "movieId") %>% left_join(b_u, by = "userId") %>%
#return(RMSE(predicted_ratings, test_set$rating))
#})


#lambdas_rmse <- min(RMSEs)
#lambda_best <- lambdas[which.min(RMSEs)]

#final_method <- paste("Optimized Penalized LSE Model with lambda =", #as.character(lambda_best), sep="
#rmse_results <- bind_rows(rmse_results, tibble(method=final_method, RMSE = lambdas_rmse))

#rmse_results
```

Modeling Approach

Ultimately, what helped more than fine-tuning lambda was to throw away the outliers that were adding more noise than was worth in the overall weighted model. I experimented with a few hard-coded values:

```
edx2 <- edx %>% group_by(movieId) %>% filter(n()>15) %>% ungroup()
edx2 <- edx2 %>% group_by(userId) %>% filter(n()>15) %>% ungroup()
```

```
test_index <- createDataPartition(y = edx2$rating, times = 1, p = 0.2, list = FALSE)
train_set <- edx2[-test_index,]
test_set <- edx2[test_index,]
```

Then I moved to a statistical approach based on ignoring the lowest 10th percentile of both movies and users (in terms of numbers of ratings associated with each unique value).

```
movieId_cutoff <- edx %>% group_by(movieId) %>% summarize(n = n()) %>% .$n %>% quantile(.10)
userId_cutoff <- edx %>% group_by(userId) %>% summarize(n = n()) %>% .$n %>% quantile(.10)
edx2 <- edx %>% group_by(movieId) %>% filter(n()>movieId_cutoff) %>% ungroup()
edx2 <- edx2 %>% group_by(userId) %>% filter(n()>userId_cutoff) %>% ungroup()
test_index <- createDataPartition(y = edx2$rating, times = 1, p = 0.2, list = FALSE)
train_set <- edx2[-test_index,]
test_set <- edx2[test_index,]
```

10% seemed like a good number to work with, although this could be refined more in the future.

Results

(a results section that presents the modeling results and discusses the model performance)

We wind up with a model that looks like this:

```
# Calculate 10th percentile of movieId by rating
movieId_cutoff <-
edx %>%
group_by(movieId) %>%
summarize(n = n()) %>%
.$n %>%
quantile(.10)

paste("Movies below the 10th percentile with fewer than",
as.character(movieId_cutoff),
"ratings will be ignored.",
sep=" ")
```

```
## [1] "Movies below the 10th percentile with fewer than 10 ratings will be ignored."
```

```
#Calculate 10th percentile of userId by rating
userId_cutoff <-
edx %>%
group_by(userId) %>%
summarize(n = n()) %>%
.$n %>%
quantile(.10)

paste("Users below the 10th percentile with fewer than",
as.character(userId_cutoff),
"ratings will be ignored.",
sep=" ")
```

```
## [1] "Users below the 10th percentile with fewer than 22 ratings will be ignored."
```

```
# Remove any movie below the 10th percentile in terms of ratings
edx2 <-
edx %>%
group_by(movieId) %>%
```

```
filter(n()>=movieId_cutoff) %>%
ungroup()

# Remove any user below the 10th percentile in terms of ratings
edx2 <-
edx2 %>%
group_by(userId) %>%
filter(n()>=userId_cutoff) %>%
ungroup()

# Create partition, reserving 20% of edx set for testing purposes
test_index <-
createDataPartition(y = edx2$rating,
times = 1, p = 0.2, list = FALSE)
train_set <- edx2[-test_index,]
test_set <- edx2[test_index,]

# Define loss function, and throw away any NA values that result from
RMSE <-
function(true_ratings, predicted_ratings){
    sqrt(mean((true_ratings - predicted_ratings)^2, na.rm=TRUE))}

# Calcualte the average rating for the training set.
mu_hat <- mean(train_set$rating)

paste("Penalized User and Movie Effect Least Squares approach with a sample mean rating of",
as.character(mu_hat),
", optimizing lambda to the nearest integer.",
sep=" ")
```

```
## [1] "Penalized User and Movie Effect Least Squares approach with a sample mean rating of 3.511812370
```

```
# Using Movie and User Effects with Penalization on both, optimize lambda to the nearest integer.

lambdas <- seq(1, 10, 1)

RMSEs <- sapply(lambdas, function(l){
b_i <-
train_set %>%
group_by(movieId) %>%
summarize(b_i = sum(rating - mu_hat)/(n()+l))

b_u <-
train_set %>%
left_join(b_i, by="movieId") %>%
group_by(userId) %>%
summarize(b_u = sum(rating - b_i - mu_hat)/(n()+l))

predicted_ratings <-
test_set %>%
left_join(b_i, by = "movieId") %>%
left_join(b_u, by = "userId") %>%
mutate(pred = mu_hat + b_i + b_u) %>%
pull(pred)
```

```
return(RMSE(predicted_ratings, test_set$rating))

})

lambda_best <- lambdas[which.min(RMSEs)]

paste("Penalized LSE Model with lambda =",
as.character(lambda_best),
"gives an RMSE of",
as.character(min(RMSEs)),
"on the test set.",
sep=" ")
```

```
## [1] "Penalized LSE Model with lambda = 5 gives an RMSE of 0.863743674823741 on the test set."
```

Performance

We can measure the performance of this final algorithm against the validation set (for the first time):

```
# Use this optimized lambda value to test against the validation set.
paste("Now testing this optimized lambda against the vaildation set.")
```

```
## [1] "Now testing this optimized lambda against the vaildation set."
```

```
l <- lambda_best
b_i <- train_set %>%
group_by(movieId) %>%
summarize(b_i = sum(rating - mu_hat)/(n()+l))
b_u <- train_set %>%
left_join(b_i, by="movieId") %>%
group_by(userId) %>%
summarize(b_u = sum(rating - b_i - mu_hat)/(n()+l))
final_predicted_ratings <-
validation %>%
left_join(b_i, by = "movieId") %>%
left_join(b_u, by = "userId") %>%
mutate(pred = mu_hat + b_i + b_u) %>%
pull(pred)
final_rmse <-
RMSE(final_predicted_ratings, validation$rating)

paste("Penalized LSE Model with optimized lambda =",
as.character(lambda_best),
"gives an RMSE of",
as.character(final_rmse),
"on the validation set.",
sep=" ")
```

```
## [1] "Penalized LSE Model with optimized lambda = 5 gives an RMSE of 0.863850672990124 on the validati
```

Conclusion

(a conclusion section that gives a brief summary of the report, its limitations and future work)

Summary

We basically used a linear estimate accounting for both movie and user effect as well as a penalized term for both effects. This got us close, but not as close as we wanted to be. The original mistake of not using semi_join lead to an exploration of low-value outliers.

*** insert histogram here of userId vs. frequency and movieId vs. frequency to highlight how many users have few ratings and how many movies have few ratings. ****

These add noise without contributing to predictability. Our model is based on the mean rating for each movie across userId, which is NOT resistant to outliers. Hence, the users and movies with low frequency add noise to the mean without adding much weight to the predictions. an attempt to remove these (to eliminate NAs in the final predicitons) inadvertently lead to a reduction in RMSE, which is what makes this method powerful.

Limitations

This seems to depend on large data sets. Elminiating below a certain percentile might work against you with a smaller data set. This needs to be explored.

Future Work

Optimizing the percentile removed from each predictor (movieId and userId) would be interesting. It would be nice to keep as much of the original data set as possible while still training a good solid model.

Many predictors were not considered, such as timestamps or genres. No matrix factorization was used to ??terminology goes here???.