MovieLens Report

Xiaoshuo Liu

April 8, 2019

I. Introduction

The MovieLens data set was provided by the course and was examined for a functional recommendation system. The goal of this project is to construct a recommendation model that provide the right (or closest) movies to a user based on the potential that a high rating will be given by the person.

After testing through different models, I found that although k nearest neighbors and random forest methods would be used to create a model, their RMSEs were too high to be considered valuable (RMSE-knn >4, RMSE-rborist>1). Then, I used “recommenderlab” package to run recommendation-specific methods to construct models. The finding was that the “POPULAR” method generated the lowest RMSE. I also used principal component analysis to seek possibility of shrinking down the number of predictors or potentially improving the RMSE. I was able to reduce the number by five while keeping RMSE the same. I spent over two months working part time on this project and decided not to go further due to my time constraints. I thought it would be better to submit the project than to give up after numerous hours of work. To my fellow graders, I would love to hear your feedback so I could possibly improve the model further in the future when I have the time to work on it again.

I would like to note that I used train sets to train the model, test sets to validate, and validation sets to do the final test. This confusion was created due to my initial impression with the “edx” and “validation” sets. I thought that “edx” would be used for train and test while “validation” was for the final validation. The validation set was only used once to test the final model.

Another thing that I wanted to point out was that because I had to do this project on a laptop, the processing power was rather limited. The laptop was my only PC device to work on, and I had to use it for other things. Therefore, I had to shrink down the size of the sets so I could get results out for a reasonable amount of time. This could be a reason why my final model was not even good enough to reach the highest RMSE in the rubric.

II. Preparing the Data

The main data sets were generated by the script provided on the course page. Set edx was for training and testing while Set validation was for validating the model. The course has already required us to explore the data set, so I skipped the exploratory analysis. Knowing that the date and time in the set were labeled in the timestamp format, I first transformed the timestamp format to normal date and time format, then splitted them into separate colunms for easier analysis later.

First, I loaded the original data.

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

## Loading required package: tidyverse

## -- Attaching packages ---------------------------------------------
--------------------- tidyverse 1.2.1 --

## v ggplot2 3.1.1       v purrr   0.3.2
## v tibble  2.1.1       v dplyr   0.8.0.1
## v tidyr   0.8.3       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0

## -- 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

# 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 <- read.table(text = gsub("::", "\t", readLines(unzip(dl, "ml-1
0M100K/ratings.dat"))),
                      col.names = c("userId", "movieId", "rating", "tim
estamp"))

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)
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)
```

The original sets are quite large to handle. Therefore I decided to slice them into smallers sizes. The reason why a random draw was not used was because I wanted to give each user enough observations to better adapt the factorization method.

```
edx <- edx[c(1:300000, 3000000:3300000, 6000000:6300000),]
validation <- validation[c(1:33333, 300000:333333, 600000:633333),]
```

Then, I separated the dates and time.

```
#Load all the packages needed.
library(tidyverse)
library(randomForest)

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##
##     combine

## The following object is masked from 'package:ggplot2':
##
##     margin

library(rpart)
library(caret)
library(purrr)
library(chron)
library(stringr)


#Transform timestamp into normal date/time format
edx <- edx %>% mutate(timestamp = as.POSIXct(edx$timestamp, origin = "1
970-01-01", tz = "GMT"))

validation <- validation %>% mutate(timestamp = as.POSIXct(validation$t
imestamp, origin = "1970-01-01", tz = "GMT"))


#Splitting date and time
edx <- edx %>% separate(col = timestamp, into = c("date", "time"), sep
= " ")

validation <- validation %>% separate(col = timestamp, into = c("date",
"time"), sep = " ")

#Check the format of the two new columns
class(edx$date)

## [1] "character"

class(validation$date)

## [1] "character"

#Separate year, month, and date from the "date" column
datePrep <- edx$date
ratingYear <- str_sub(string = datePrep, start = 1, end = 4)
ratingMonth <- str_sub(string = datePrep, start = 6, end = 7)
ratingDay <- str_sub(string = datePrep, start = 9, end = 10)
edx <- cbind(edx, ratingYear)
edx <- cbind(edx, ratingMonth)
edx <- cbind(edx, ratingDay)
```

```r
datePrep <- validation$date
ratingYear <- str_sub(string = datePrep, start = 1, end = 4)
ratingMonth <- str_sub(string = datePrep, start = 6, end = 7)
ratingDay <- str_sub(string = datePrep, start = 9, end = 10)
validation <- cbind(validation, ratingYear)
validation <- cbind(validation, ratingMonth)
validation <- cbind(validation, ratingDay)

#Remove the original "date" column to slim down the set.
edx <- edx[, -4]
validation <- validation[, -4]

#Separate hour, minute, and second from the "time" column.
timePrep <- edx$time
ratingHour <- str_sub(string = timePrep, start = 1, end = 2)
ratingMin <- str_sub(string = timePrep, start = 4, end = 5)
ratingSec <- str_sub(string = timePrep, start = 7, end = 8)
edx <- cbind(edx, ratingHour)
edx <- cbind(edx, ratingMin)
edx <- cbind(edx, ratingSec)

timePrep <- validation$time
ratingHour <- str_sub(string = timePrep, start = 1, end = 2)
ratingMin <- str_sub(string = timePrep, start = 4, end = 5)
ratingSec <- str_sub(string = timePrep, start = 7, end = 8)
validation <- cbind(validation, ratingHour)
validation <- cbind(validation, ratingMin)
validation <- cbind(validation, ratingSec)

#Remove the original "time" column to slim down the set.
edx <- edx[, -4]
validation <- validation[, -4]

#Remove all the values created during this process.
rm(datePrep)
rm(ratingDay)
rm(ratingHour)
rm(ratingMin)
rm(ratingMonth)
rm(ratingSec)
rm(ratingYear)
rm(timePrep)
```

After getting the rating date and time prepared. I also extracted the release year out of the “title” column as a variable.

```r
#Create a column of release year.
releasePrep <- edx$title
releaseYear <- str_sub(string = releasePrep, start = -5, end = -2)
edx <- cbind(edx, releaseYear)
```

```r
releasePrep <- validation$title
releaseYear <- str_sub(string = releasePrep, start = -5, end = -2)
validation <- cbind(validation, releaseYear)

#Remove the values no longer used.
rm(releasePrep)
rm(releaseYear)
```

I also converted the “genres” column into a series of columns that are dummy variables for later analysis.

```r
library(dplyr)
library(tidyr)

#Separate genres under the same title
edx <- separate_rows(data = edx, genres, sep = "\\|", convert = FALSE)

#Transform character values into dummy variables
edx <- edx %>% mutate(Drama = ifelse(genres == "Drama", 1, 0)) %>%
  mutate(Crime = ifelse(genres == "Crime", 1, 0)) %>%
  mutate(Action = ifelse(genres == "Action", 1, 0)) %>%
  mutate(Adventure = ifelse(genres == "Adventure", 1, 0)) %>%
  mutate(Sci_Fi = ifelse(genres == "Sci-Fi", 1, 0)) %>%
  mutate(Thriller = ifelse(genres == "Thriller", 1, 0)) %>%
  mutate(Comedy = ifelse(genres == "Comedy", 1, 0)) %>%
  mutate(Mystery = ifelse(genres == "Mystery", 1, 0)) %>%
  mutate(Romance = ifelse(genres == "Romance", 1, 0)) %>%
  mutate(Animation = ifelse(genres == "Animation", 1, 0)) %>%
  mutate(Children = ifelse(genres == "Children", 1, 0)) %>%
  mutate(Fantasy = ifelse(genres == "Fantasy", 1, 0)) %>%
  mutate(War = ifelse(genres == "War", 1, 0)) %>%
  mutate(Horror = ifelse(genres == "Horror", 1, 0)) %>%
  mutate(Musical = ifelse(genres == "Musical", 1, 0)) %>%
  mutate(Western = ifelse(genres == "Western", 1, 0)) %>%
  mutate(Film_Noir = ifelse(genres == "Film-Noir", 1, 0)) %>%
  mutate(Documentary = ifelse(genres == "Documentary", 1, 0)) %>%
  mutate(IMAX = ifelse(genres == "IMAX", 1, 0)) %>%
  mutate(no_genres_listed = ifelse(genres == "(no genres listed)", 1, 0
))

#Deleting the "genres" column after converting all genres into dummy va
riables and title column (assume movieId would just do the same).
edx <- within(edx, rm(genres))
edx <- within(edx, rm(title))
```

After all the foundation work, partitions were created for modeling later.

```r
#Creating Partitions
rating <- edx$rating
```

```
set.seed(1)
testIndex <- createDataPartition(rating, times = 1, p = 0.5, list = FAL
SE)
trainSet <- edx[-testIndex,]
testSet <- edx[testIndex,]

#Making sure that the two sets share users and movies.
testSet <- testSet %>%
          semi_join(trainSet, by = "movieId") %>%
          semi_join(trainSet, by = "userId")

#releasing memory
rm(edx)
rm(testIndex)
rm(rating)
```

Splitting genres for the validation set. It was done after removing edx because of RAM limit.

```
#Applying the same method to the validation set
validation <- separate_rows(data = validation, genres, sep = "\\|", con
vert = FALSE)

validation <- validation %>% mutate(Drama = ifelse(genres == "Drama", 1
, 0)) %>%
  mutate(Crime = ifelse(genres == "Crime", 1, 0)) %>%
  mutate(Action = ifelse(genres == "Action", 1, 0)) %>%
  mutate(Adventure = ifelse(genres == "Adventure", 1, 0)) %>%
  mutate(Sci_Fi = ifelse(genres == "Sci-Fi", 1, 0)) %>%
  mutate(Thriller = ifelse(genres == "Thriller", 1, 0)) %>%
  mutate(Comedy = ifelse(genres == "Comedy", 1, 0)) %>%
  mutate(Mystery = ifelse(genres == "Mystery", 1, 0)) %>%
  mutate(Romance = ifelse(genres == "Romance", 1, 0)) %>%
  mutate(Animation = ifelse(genres == "Animation", 1, 0)) %>%
  mutate(Children = ifelse(genres == "Children", 1, 0)) %>%
  mutate(Fantasy = ifelse(genres == "Fantasy", 1, 0)) %>%
  mutate(War = ifelse(genres == "War", 1, 0)) %>%
  mutate(Horror = ifelse(genres == "Horror", 1, 0)) %>%
  mutate(Musical = ifelse(genres == "Musical", 1, 0)) %>%
  mutate(Western = ifelse(genres == "Western", 1, 0)) %>%
  mutate(Film_Noir = ifelse(genres == "Film-Noir", 1, 0)) %>%
  mutate(Documentary = ifelse(genres == "Documentary", 1, 0)) %>%
  mutate(IMAX = ifelse(genres == "IMAX", 1, 0)) %>%
  mutate(no_genres_listed = ifelse(genres == "(no genres listed)", 1, 0
))


validation <- within(validation, rm(genres))
validation <- within(validation, rm(title))
```

III. Predictive Modeling

Although in the lectures, Dr. Irizarry used matrix factorization to construct his sample model, I want to try the two predictive models, K nearest neighbors and regression tree. I noticed that the train and test sets were still quite big for testing models, so I created a smaller set. The “movieID” column was also removed because it caused RAM error on my computer which made it unable to complete model training and testing.

```
#Create smaller dataset for model testing
set.seed(1)
tinyTrain <- trainSet[c(1:33333, 300000:333333, 600000:633334),]
tinyTest <- testSet[c(1:33333, 300000:333333, 600000:633334),]

#Remove the original train and test sets.
rm(trainSet)
rm(testSet)

#Eliminating "movieID" only for predictive modeling (causing RAM issue)
trainMovieId <- tinyTrain$movieId
testMovieId <- tinyTest$movieId
tinyTrain <- within(tinyTrain, rm(movieId))
tinyTest <- within(tinyTest, rm(movieId))
```

1. K Nearest Neighbors To reduce lengthy processing time, I adopted the control from the lectures.

```
#Create dataframe for predictors.
predictors <- within(tinyTrain, rm(rating))
#k nearest neighbors
control <- trainControl(method = "cv", number = 3, p = .8)
pred <- tinyTrain[, 2]
train_knn <- train(predictors, pred,
                   method = "knn",
                   tuneGrid = data.frame(k = seq(15,25,2)),
                   trControl = control)
train_knn$bestTune

##    k
## 1 15

pred <- as.factor(pred)
fit_knn <- knn3(predictors, pred, k = train_knn$bestTune)
testPredictors <- within(tinyTest, rm(rating))
y_hat_knn <- predict(fit_knn,
                     testPredictors,
                     type = "class")
testPred <- tinyTest[,2]
cm_knn <- confusionMatrix(as.factor(y_hat_knn), as.factor(testPred))
rmse_knn <- RMSE(as.numeric(testPred), as.numeric(y_hat_knn))
cm_knn
```

```
## Confusion Matrix and Statistics
##
##              Reference
## Prediction   0.5     1   1.5     2   2.5     3   3.5     4   4.5
5
##        0.5    80    13    32    30    24    22    40    27    29
20
##        1      29   675    33   293    57   445    87   296    40   2
24
##        1.5    15    26   109    44    68    42    43    59    16
11
##        2      32   289    70  1102   140   954   209   693    93   3
48
##        2.5    54    83   100   188   625   384   335   318   140
93
##        3     117  1020   190  2426   669 11284  1259  6656   593  31
06
##        3.5    93   157   142   307   532  1130  2276  1394   622   3
42
##        4     130   968   222  2220   778  7116  2369 15974  1906  54
55
##        4.5    42    52    66   141   162   386   481   828  1154   3
90
##        5      52   434    58   779   147  2357   477  3880   600  68
84
##
## Overall Statistics
##
##                Accuracy : 0.4016
##                  95% CI : (0.3986, 0.4047)
##     No Information Rate : 0.3012
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.2368
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: 0.5 Class: 1 Class: 1.5 Class: 2 Class:
2.5
## Sensitivity             0.12422  0.18160    0.10665  0.14635    0.19
519
## Specificity             0.99761  0.98438    0.99673  0.96942    0.98
249
## Pos Pred Value          0.25237  0.30978    0.25173  0.28041    0.26
940
## Neg Pred Value          0.99434  0.96890    0.99083  0.93309    0.97
362
## Prevalence              0.00644  0.03717    0.01022  0.07530    0.03
```

```
202
## Detection Rate         0.00080  0.00675    0.00109  0.01102    0.00
625
## Detection Prevalence   0.00317  0.02179    0.00433  0.03930    0.02
320
## Balanced Accuracy      0.56092  0.58299    0.55169  0.55788    0.58
884
##                       Class: 3 Class: 3.5 Class: 4 Class: 4.5 Class:
5
## Sensitivity           0.4678    0.30042   0.5303    0.22222  0.4079
9
## Specificity           0.7887    0.94894   0.6971    0.97312  0.8943
3
## Pos Pred Value        0.4130    0.32538   0.4301    0.31172  0.4393
7
## Neg Pred Value        0.8234    0.94302   0.7749    0.95806  0.8815
5
## Prevalence            0.2412    0.07576   0.3012    0.05193  0.1687
3
## Detection Rate        0.1128    0.02276   0.1597    0.01154  0.0688
4
## Detection Prevalence  0.2732    0.06995   0.3714    0.03702  0.1566
8
## Balanced Accuracy     0.6282    0.62468   0.6137    0.59767  0.6511
6
```

```r
rmse_knn
```

```
## [1] 4.167174
```

The result showed an accuracy of ~40% and a terrible RMSE of 4.17. Clearly, knn was not able to construct a high quality recommendation system.

2. Classification Tree (Rborist w/ five-fold cross validation) To reduce lengthy processing time, I adopted the control from the lectures.

```r
library(Rborist)
```

```
## Rborist 0.1-17
```

```
## Type RboristNews() to see new features/changes/bug fixes.
```

```r
control <- trainControl(method = "cv", number = 3, p = 0.8)
grid <- expand.grid(minNode = c(2,5), predFixed = c(15,20,25))
pred <- as.factor(tinyTrain[, 2])

train_rf <- train(predictors,
                  pred,
                  method = "Rborist",
                  nTree = 50,
                  trControl = control,
                  tuneGrid = grid,
```

```
                   nSample = 5000)
train_rf$bestTune

##   predFixed minNode
## 1        15       2

fit_rf <- Rborist(predictors, pred,
                  minNode = train_rf$bestTune$minNode,
                  predFixed = train_rf$bestTune$predFixed)

testPredictors <- within(tinyTest, rm(rating))
testPred <- tinyTest[,2]
pred_rf <- predict(fit_rf, testPredictors)
y_hat_rf <- as.factor(levels(pred)[predict(fit_rf, testPredictors)$yPre
d])
testPred <- as.factor(testPred)
cm_rf <- confusionMatrix(y_hat_rf, testPred)
rmse_rf <- RMSE(as.numeric(testPred), as.numeric(y_hat_rf))
cm_rf

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0.5     1   1.5     2   2.5     3   3.5     4   4.5
5
##        0.5   433     6     7    12     7    13     7     3    15
8
##        1       7  2470    11   179    25   258    10   130     4
99
##        1.5     4    10   726    22     9     9    12    10     7
0
##        2      13   153    29  5027    36   529    41   380     3   1
59
##        2.5    23    46    54    97  2332   104    63    50    15
11
##        3      40   385    72   880   244 18552   299  2686   126  11
14
##        3.5    28    38    43    98   226   467  5973   528   248   1
13
##        4      62   421    64   885   263  3191   948 24463   764  24
58
##        4.5     8    16    12    26    29    76   146   285  3824   1
31
##        5      26   172     4   304    31   921    77  1590   187 127
80
##
## Overall Statistics
##
##                Accuracy : 0.7658
##                  95% CI : (0.7631, 0.7684)
```

```
##     No Information Rate : 0.3012
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7064
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: 0.5 Class: 1 Class: 1.5 Class: 2 Class:
2.5
## Sensitivity             0.67236  0.66451    0.71037  0.66760    0.72
829
## Specificity             0.99921  0.99249    0.99916  0.98548    0.99
522
## Pos Pred Value          0.84736  0.77357    0.89740  0.78917    0.83
435
## Neg Pred Value          0.99788  0.98712    0.99702  0.97327    0.99
105
## Prevalence              0.00644  0.03717    0.01022  0.07530    0.03
202
## Detection Rate          0.00433  0.02470    0.00726  0.05027    0.02
332
## Detection Prevalence    0.00511  0.03193    0.00809  0.06370    0.02
795
## Balanced Accuracy       0.83579  0.82850    0.85477  0.82654    0.86
176
##                      Class: 3 Class: 3.5 Class: 4 Class: 4.5 Class:
5
## Sensitivity            0.7692    0.78841   0.8120    0.73638   0.757
4
## Specificity            0.9230    0.98064   0.8704    0.99231   0.960
2
## Pos Pred Value         0.7604    0.76952   0.7298    0.83989   0.794
2
## Neg Pred Value         0.9264    0.98262   0.9148    0.98566   0.951
2
## Prevalence             0.2412    0.07576   0.3012    0.05193   0.168
7
## Detection Rate         0.1855    0.05973   0.2446    0.03824   0.127
8
## Detection Prevalence   0.2440    0.07762   0.3352    0.04553   0.160
9
## Balanced Accuracy      0.8461    0.88453   0.8412    0.86434   0.858
8
```

```
rmse_rf
```

```
## [1] 1.409727
```

The result showed an accuracy of ~77%, which was a big improvement compared to the knn result. Also, the RMSE improved from 4.16 to 1.41. Based on the project's rubric, the RMSE was still not good enough. It wasn't worth testing on the validation set.

In conclusion, both knn and regression tree methods failed to construct satisfying recommendation systems. Before moving into matrix factorization, I wanted to try principal component analysis and see if the procedure could help improve the results of knn and regression tree a bit.

IV. Principal Component Analysis

There are over twenty columns which represent different genres in the set. This has caused some serious lags on processing time for modeling. I wanted to find out whether a PCA can shrink down the number of variables needed for the prediction.

```
#Using Principal Component Analysis to shrink down the number of predic
tors

pca <- prcomp(tinyTrain[,c(10:29)], center = FALSE, scale. = FALSE)
pcaTest <- prcomp(tinyTest[,c(10:29)], center = FALSE, scale. = FALSE)

summary(pca)

## Importance of components:
##                           PC1    PC2    PC3    PC4     PC5    PC6
PC7
## Standard deviation     0.4191 0.3885 0.3259 0.3186 0.28190 0.2737 0.
24052
## Proportion of Variance 0.1756 0.1509 0.1062 0.1015 0.07947 0.0749 0.
05785
## Cumulative Proportion  0.1756 0.3266 0.4328 0.5343 0.61372 0.6886 0.
74647
##                            PC8     PC9   PC10    PC11    PC12    PC1
3
## Standard deviation     0.23668 0.19568 0.1709 0.16565 0.16263 0.1502
3
## Proportion of Variance 0.05602 0.03829 0.0292 0.02744 0.02645 0.0225
7
## Cumulative Proportion  0.80248 0.84077 0.8700 0.89741 0.92386 0.9464
3
##                           PC14    PC15    PC16    PC17    PC18    PC
19
## Standard deviation     0.13539 0.13483 0.08752 0.07450 0.05916 0.018
71
## Proportion of Variance 0.01833 0.01818 0.00766 0.00555 0.00350 0.000
35
## Cumulative Proportion  0.96476 0.98294 0.99060 0.99615 0.99965 1.000
00
##                        PC20
```

```
## Standard deviation       0
## Proportion of Variance   0
## Cumulative Proportion    1

#Here we can see that the first 15 components capture ~98% of the varia
bility. Let's use these 15 which represent genres and the rest of regul
ar predictors.

axes <- predict(pca, newdata = tinyTrain)
axesTest <- predict(pcaTest, newdata = tinyTest)

pcaData <- cbind(tinyTrain, axes)
pcaTestData <- cbind(tinyTest, axesTest)
```

After getting the PCA variables, I tested them with the random forest model. The knn model's results were too bad to test.

```
#Create dataframe for predictors and predicted values.
PCApredictors <- pcaData[,c(1,3:9,30:44)]
PCApred <- pcaData[,2]

PCAtestPredictors <- pcaTestData[,c(1,3:9,30:44)]
PCAtestPred <- pcaTestData[,2]

#Random Forest
PCAcontrol <- trainControl(method = "cv", number = 3, p = 0.8)
PCAgrid <- expand.grid(minNode = c(2,5), predFixed = c(10,15,20))
PCApred <- as.factor(tinyTrain[, 2])

PCAtrain_rf <- train(PCApredictors,
                  PCApred,
                  method = "Rborist",
                  nTree = 50,
                  trControl = PCAcontrol,
                  tuneGrid = PCAgrid,
                  nSample = 5000)
PCAtrain_rf$bestTune

##   predFixed minNode
## 5        15       5

PCAfit_rf <- Rborist(PCApredictors, PCApred,
                  minNode = PCAtrain_rf$bestTune$minNode,
                  predFixed = PCAtrain_rf$bestTune$predFixed)

PCApred_rf <- predict(PCAfit_rf, PCAtestPredictors)
PCAy_hat_rf <- as.factor(levels(PCApred)[predict(PCAfit_rf, PCAtestPred
ictors)$yPred])
PCAtestPred <- as.factor(PCAtestPred)
PCAcm_rf <- confusionMatrix(PCAy_hat_rf, PCAtestPred)
```

```
PCArmse_rf <- RMSE(as.numeric(PCAtestPred), as.numeric(PCAy_hat_rf))
PCAcm_rf

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0.5     1   1.5     2   2.5     3   3.5     4   4.5
5
##        0.5    44    13     2    12     5     6     5    10    19
10
##        1      13   367     8   126    11   149     7   112     4
84
##        1.5     0     0     2     0     0     0     0     0     0
0
##        2       2   145     3   384    24   240     0   161     0
93
##        2.5    56   100   150   158   430   151   130    65    31
13
##        3     112  1124   258  2603   685 11874   776  5170   234  23
76
##        3.5    88   153   179   407   738  1322  3048  1138   637   2
80
##        4     241  1361   351  3217  1175  8521  3150 20494  2975  71
32
##        4.5    36     9    31    20    66   110   234   381   774   1
74
##        5      52   445    38   603    68  1747   226  2594   519  67
11
##
## Overall Statistics
##
##                Accuracy : 0.4413
##                  95% CI : (0.4382, 0.4444)
##     No Information Rate : 0.3012
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.267
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: 0.5 Class: 1 Class: 1.5 Class: 2 Class:
2.5
## Sensitivity             0.06832  0.09874   0.001957  0.05100    0.13
429
## Specificity             0.99917  0.99466   1.000000  0.99278    0.99
118
## Pos Pred Value          0.34921  0.41657   1.000000  0.36502    0.33
489
```

```
## Neg Pred Value          0.99399  0.96620   0.989800  0.92778    0.97
192
## Prevalence              0.00644  0.03717   0.010220  0.07530    0.03
202
## Detection Rate          0.00044  0.00367   0.000020  0.00384    0.00
430
## Detection Prevalence    0.00126  0.00881   0.000020  0.01052    0.01
284
## Balanced Accuracy       0.53375  0.54670   0.500978  0.52189    0.56
273
##                        Class: 3 Class: 3.5 Class: 4 Class: 4.5 Class:
5
## Sensitivity              0.4923    0.40232   0.6803    0.14905  0.3977
4
## Specificity              0.8242    0.94653   0.5975    0.98881  0.9243
1
## Pos Pred Value           0.4710    0.38148   0.4215    0.42180  0.5161
1
## Neg Pred Value           0.8363    0.95079   0.8126    0.95498  0.8831
9
## Prevalence               0.2412    0.07576   0.3012    0.05193  0.1687
3
## Detection Rate           0.1187    0.03048   0.2049    0.00774  0.0671
1
## Detection Prevalence     0.2521    0.07990   0.4862    0.01835  0.1300
3
## Balanced Accuracy        0.6583    0.67443   0.6389    0.56893  0.6610
2
```

```
PCArmse_rf
```

```
## [1] 2.132742
```

As the result entailed, the PCA variables didn't improve either the accuracy or the RMSE of the model. Instead, it even made the results worse. It was about time to use the matrix factorization method taught in the lectures.

V. "Recommenderlab" Package

Dr. Irizarry mentioned the "recommenderlab" package at the end of his lecture on a recommendation system. After some research, I learned that the package was specifically written for recommendation system research. The methods available are very similar to the ones introduced by Dr. Irizarry in his lectures. Therefore, instead of replicating and improving the process used in the lecture, I decided to utilize the package for further model construction. For more detailed description of functions that were used, please view the package's document.

The Recommenderlab package had a function called "evaluate", which evaluates the outcome of one or more methods with the same data set. The methods were user-based collaborative filtering, item-based collaborative filtering, singular value decomposition(SVD) with column-mean imputation, funk SVD, association rule-based recommender, popular items, randomly chosen items for comparison, and re-recommend liked items. Therefore, I decided to evaluate all the methods first before putting them into training.

First, I added the a crucial component, movie IDs, back to the set to where it used to be. I took it out earlier because it was causing RAM issue when running knn and Rborist models.

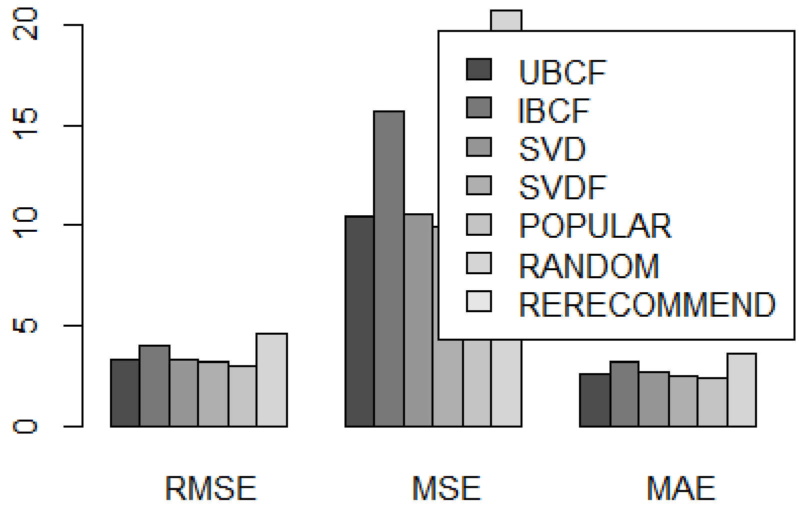
```
#Using the package "recommendarlab"
library(recommenderlab)

## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following object is masked from 'package:tidyr':
##
##     expand

## Loading required package: arules

##
## Attaching package: 'arules'

## The following object is masked from 'package:dplyr':
##
##     recode

## The following objects are masked from 'package:base':
##
##     abbreviate, write

## Loading required package: proxy

##
## Attaching package: 'proxy'

## The following object is masked from 'package:Matrix':
##
##     as.matrix

## The following objects are masked from 'package:stats':
##
##     as.dist, dist

## The following object is masked from 'package:base':
##
##     as.matrix

## Loading required package: registry
```

```
##
## Attaching package: 'recommenderlab'

## The following objects are masked from 'package:caret':
##
##     MAE, RMSE

library(dplyr)
library(tibble)

#Add "movieId" back into the sets.
tinyTrain <- add_column(tinyTrain, movieId = trainMovieId, .after = 1)
tinyTest <- add_column(tinyTest, movieId = testMovieId, .after = 1)
```

Once the sets were restored back to their original format, I started using the "recommenderlab" package for modeling.

```
#Getting a recommender training set.
recomTrain <- tinyTrain

#Convert the dataframe to a rating matrix.
recomTrain <- as(recomTrain, "realRatingMatrix")

#Create an evaluation scheme.
e <- evaluationScheme(recomTrain, method = "cross-validation", train =
0.5, k = 3, given = -1)
e

## Evaluation scheme using all-but-1 items
## Method: 'cross-validation' with 3 run(s).
## Good ratings: NA
## Data set: 647 x 5526 rating matrix of class 'realRatingMatrix' with
60518 ratings.

#Evaluate different methods
algos <- list(
  UBCF = list(name = "UBCF", param = NULL),
  IBCF = list(name = "IBCF", param = NULL),
  SVD = list(name = "SVD", param = NULL),
  SVDF = list(name = "SVDF", param = NULL),
  AR = list(name = "AR", param = NULL),
  POPULAR = list(name = "POPULAR", param = NULL),
  RANDOM = list(name = "RANDOM", param = NULL),
  RERECOMMEND = list(name = "RERECOMMEND", param = NULL)
)

evlist <- evaluate(e, algos, type = "ratings")

## UBCF run fold/sample [model time/prediction time]
##   1  [0.01sec/3.74sec]
##   2  [0sec/3.89sec]
```

```
##   3  [0sec/4sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [680.93sec/0.21sec]
##   2  [688.61sec/0.21sec]
##   3  [681.81sec/0.48sec]
## SVD run fold/sample [model time/prediction time]
##   1  [0.36sec/0.6sec]
##   2  [0.34sec/0.6sec]
##   3  [0.39sec/0.88sec]
## SVDF run fold/sample [model time/prediction time]
##   1  [135.95sec/68.93sec]
##   2  [134.93sec/69.95sec]
##   3  [134.57sec/67.72sec]
## AR run fold/sample [model time/prediction time]
##   1

## Timing stopped at: 0 0 0

## Error in .local(data, ...) :
##   Recommender method AR not implemented for data type realRatingMatr
ix .
## POPULAR run fold/sample [model time/prediction time]
##   1  [0sec/0.42sec]
##   2  [0.02sec/0.44sec]
##   3  [0.02sec/0.39sec]
## RANDOM run fold/sample [model time/prediction time]
##   1  [0sec/0.64sec]
##   2  [0sec/0.66sec]
##   3  [0sec/0.96sec]
## RERECOMMEND run fold/sample [model time/prediction time]
##   1  [0sec/0sec]
##   2  [0.01sec/0sec]
##   3  [0sec/0sec]

## Warning in .local(x, method, ...):
##   Recommender 'AR' has failed and has been removed from the results!

plot(evlist, legend = "topright")
```




```
evResults <- avg(evlist)
evResults

## $UBCF
##         RMSE      MSE      MAE
## res 3.232256 10.44853 2.614681
##
## $IBCF
##         RMSE      MSE      MAE
## res 3.947136 15.65417 3.227751
##
## $SVD
##         RMSE      MSE      MAE
## res 3.252254 10.57847 2.633225
##
## $SVDF
##         RMSE      MSE      MAE
## res 3.148501 9.924568 2.429291
##
## $POPULAR
##        RMSE      MSE      MAE
## res 3.01736 9.114525 2.355671
##
## $RANDOM
##         RMSE      MSE      MAE
## res 4.548748 20.70323 3.582987
##
```

```
## $RERECOMMEND
##     RMSE MSE MAE
## res  NaN NaN NaN
```

As we can see from the graph and the result sheet, the method "POPULAR" (based on popular items across users) had the lowest RMSE. I created a model with this method.

```
#Create POPULAR recommender
r <- Recommender(getData(e, "train"), "POPULAR")
r

## Recommender of type 'POPULAR' for 'realRatingMatrix'
## learned using 430 users.

#Create predictions
p <- recommenderlab::predict(r, getData(e, "known"), type = "ratings")
p

## 217 x 5526 rating matrix of class 'realRatingMatrix' with 1066956 ra
tings.

#Calculate accuray
Accu <- as.data.frame(calcPredictionAccuracy(p, getData(e, "unknown"),
byUser = TRUE)) %>% na.omit() %>%
        colMeans()
Accu

##     RMSE      MSE      MAE
## 2.313673 8.873735 2.313673
```

As we can see from the result, the RMSE improved from the result from the evaluation process. However, it was still far from the desired RMSE range provided by the rubric. Therefore, I wanted to ensemble several methods to improve the results.

The "recommenderlab" package offered a "Hybrid Recommender" function, which essnetially allowed combining several methods for one recommendation system. I chose the top three results, user-based collaborative filtering (UBCF), funk SVD (SVDF), and popular items (POPULAR) from the evaluation results and see if the RMSE would be improved.

```
#Create hybrid recommender
rHy <- HybridRecommender(
  Recommender(getData(e, "train"), "UBCF"),
  Recommender(getData(e, "train"),"SVDF"),
  Recommender(getData(e, "train"),"POPULAR"),
  weights = NULL
)
rHy
```

```
## Recommender of type 'HYBRID' for 'ratingMatrix'
## learned using NA users.

#Create predictions
pHy <- recommenderlab::predict(rHy, getData(e, "known"), type = "rating
s")
pHy

## 217 x 5526 rating matrix of class 'realRatingMatrix' with 1179457 ra
tings.

#Calculate accuray
AccuHy <- as.data.frame(calcPredictionAccuracy(pHy, getData(e, "unknown
"), byUser = TRUE)) %>% na.omit() %>%colMeans()
AccuHy

##     RMSE      MSE      MAE
## 2.274925 8.501002 2.274925
```

As we could see from the results, the hybrid model improved the RMSE by a tiny margin. This was still far from our desired results, so I decided to engage in further method research.

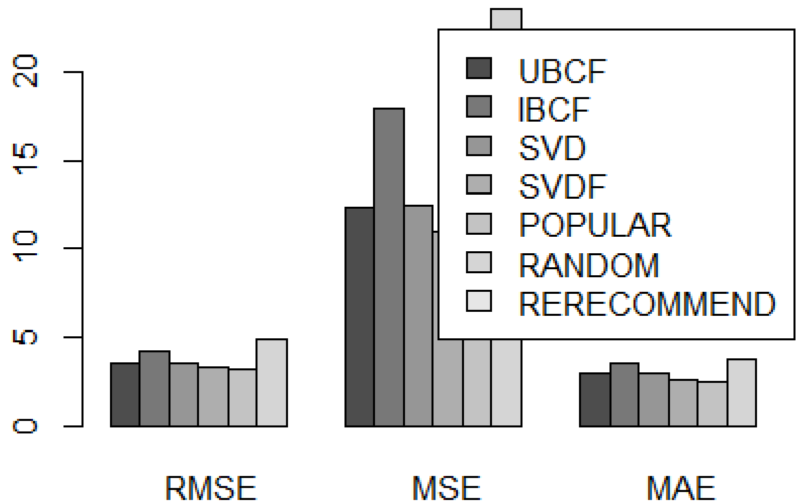
VI. "Recommenderlab" with Principal Component Analysis

Similar to the knn and regression tree model tries, I wanted to see whether PCA could reduce the number of predictors needed or help improve the results of models.

```
#PCA with Recommenderlab
pca <- prcomp(tinyTrain[,c(11:30)], center = FALSE, scale. = FALSE)
axes <- predict(pca, newdata = tinyTrain)
pcaData <- cbind(tinyTrain, axes)
pcaSet <- pcaData[,c(1:10,31:45)]

#Convert the dataframe to a rating matrix.
pcaSet <- as(pcaSet, "realRatingMatrix")

#Create an evaluation scheme.
e <- evaluationScheme(pcaSet, method = "cross-validation", train = 0.5,
k = 3, given = -1)
e

## Evaluation scheme using all-but-1 items
## Method: 'cross-validation' with 3 run(s).
## Good ratings: NA
## Data set: 647 x 5526 rating matrix of class 'realRatingMatrix' with
60518 ratings.

#Evaluate all methods.
algos <- list(
```

```
  UBCF = list(name = "UBCF", param = NULL),
  IBCF = list(name = "IBCF", param = NULL),
  SVD = list(name = "SVD", param = NULL),
  SVDF = list(name = "SVDF", param = NULL),
  AR = list(name = "AR", param = NULL),
  POPULAR = list(name = "POPULAR", param = NULL),
  RANDOM = list(name = "RANDOM", param = NULL),
  RERECOMMEND = list(name = "RERECOMMEND", param = NULL)
)

PCAevlist <- evaluate(e, algos, type = "ratings")

## UBCF run fold/sample [model time/prediction time]
##   1  [0.02sec/3.62sec]
##   2  [0sec/3.7sec]
##   3  [0.02sec/4.01sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [687.94sec/0.27sec]
##   2  [687.47sec/0.17sec]
##   3  [686.37sec/0.16sec]
## SVD run fold/sample [model time/prediction time]
##   1  [0.34sec/0.6sec]
##   2  [0.33sec/0.91sec]
##   3  [0.34sec/0.61sec]
## SVDF run fold/sample [model time/prediction time]
##   1  [133.22sec/68.8sec]
##   2  [134.4sec/69.4sec]
##   3  [134.91sec/66.42sec]
## AR run fold/sample [model time/prediction time]
##   1

## Timing stopped at: 0 0 0

## Error in .local(data, ...) :
##   Recommender method AR not implemented for data type realRatingMatr
ix .
## POPULAR run fold/sample [model time/prediction time]
##   1  [0sec/0.37sec]
##   2  [0.02sec/0.39sec]
##   3  [0sec/0.46sec]
## RANDOM run fold/sample [model time/prediction time]
##   1  [0sec/0.92sec]
##   2  [0sec/0.65sec]
##   3  [0sec/0.64sec]
## RERECOMMEND run fold/sample [model time/prediction time]
##   1  [0sec/0sec]
##   2  [0sec/0sec]
##   3  [0sec/0sec]

## Warning in .local(x, method, ...):
##   Recommender 'AR' has failed and has been removed from the results!
```

```r
plot(PCAevlist, legend = "topright")
```




```r
PCAevResults <- avg(PCAevlist)
PCAevResults

## $UBCF
##         RMSE      MSE      MAE
## res 3.503486 12.33189 2.877482
##
## $IBCF
##         RMSE      MSE      MAE
## res 4.215331 17.96341 3.537317
##
## $SVD
##         RMSE     MSE      MAE
## res 3.524227 12.4755 2.882826
##
## $SVDF
##         RMSE      MSE      MAE
## res 3.306904 10.95721 2.592936
##
## $POPULAR
##         RMSE      MSE      MAE
## res 3.172004 10.09638 2.458764
##
## $RANDOM
##         RMSE      MSE     MAE
```

```
## res 4.843198 23.50578 3.75713
##
## $RERECOMMEND
##     RMSE MSE MAE
## res  NaN NaN NaN
```

As we can see from the result sheet, POPULAR method still had the lowest RMSE at 3.17. I created a model with this method. Since a single method model would most likely have similar or worse results from the non-PCA test, I decided to go straight into creating a hybrid recommender to see whether PCA would improve the strangely high RMSE.

```
#Use the top 3 models of different kinds to create a hybrid recommender
.
PCAHybrid <- HybridRecommender(
  Recommender(getData(e, "train"), "UBCF"),
  Recommender(getData(e, "train"),"SVDF"),
  Recommender(getData(e, "train"), "POPULAR"),
  weights = NULL
)

PCAHybrid

## Recommender of type 'HYBRID' for 'ratingMatrix'
## learned using NA users.

#Create predictions
PCAPred <- recommenderlab::predict(PCAHybrid, getData(e, "known"), type
= "ratings")
PCAPred

## 217 x 5526 rating matrix of class 'realRatingMatrix' with 1177063 ra
tings.

#Calculate accuray
PCAAccu <- colMeans(as.data.frame(calcPredictionAccuracy(PCAPred, getDa
ta(e, "unknown"), byUser = TRUE)))
PCAAccu

##     RMSE      MSE      MAE
## 2.460333 9.154674 2.460333
```

As we can see from the results, RMSE was not improved and still very high.

Finally, I tested the POPULAR model on the original validation set. Before doing so, I needed to remove some objects that’s no longer used to release enough RAM for the larger data set (although it was already slimed down earlier).

```
rm(pca)
rm(axes)
rm(axesTest)
```

```
rm(pcaData)
rm(pcaSet)
rm(tinyTrain)
rm(tinyTest)
rm(train_knn)
rm(train_rf)
rm(y_hat_rf)
rm(y_hat_knn)
rm(fit_rf)
rm(fit_knn)
```

Then, I did the final test on the validation set.

```
library(recommenderlab)
library(dplyr)
#PCA with Recommenderlab
pcaVali <- prcomp(validation[,c(11:30)], center = FALSE, scale. = FALSE
)
axesVali <- predict(pcaVali, newdata = validation)
pcaDataVali <- cbind(validation, axesVali)
pcaSetVali <- pcaDataVali[,c(1:10,31:45)]



#Convert the dataframe to a rating matrix.
pcaRecomSetVali <- as(pcaSetVali, "realRatingMatrix")

#Create an evaluation scheme.
eRecomSetVali <- evaluationScheme(pcaRecomSetVali, method = "cross-vali
dation", train = 0.5, k = 3, given = -1)
eRecomSetVali

## Evaluation scheme using all-but-1 items
## Method: 'cross-validation' with 3 run(s).
## Good ratings: NA
## Data set: 6986 x 6701 rating matrix of class 'realRatingMatrix' with
100001 ratings.

#Create the model
rRecomSetVali <- HybridRecommender(
  Recommender(getData(eRecomSetVali, "train"), "UBCF"),
  Recommender(getData(eRecomSetVali, "train"),"SVDF"),
  Recommender(getData(eRecomSetVali, "train"), "POPULAR"),
  weights = NULL
)
rRecomSetVali

## Recommender of type 'HYBRID' for 'ratingMatrix'
## learned using NA users.

#Create predictions
pRecomSetVali <- recommenderlab::predict(rRecomSetVali, getData(eRecomS
```

```
etVali, "known"), type = "ratings")
pRecomSetVali

## 2330 x 6701 rating matrix of class 'realRatingMatrix' with 15580904
ratings.

#Calculate accuray
AccuVali <- colMeans(as.data.frame(calcPredictionAccuracy(pRecomSetVali
, getData(eRecomSetVali, "unknown"), byUser = TRUE)))
AccuVali

##      RMSE       MSE       MAE
##  3.536752 22.533741  3.536752
```

As we can see, unfortunately, the model didn't work well on the validation set.

One thing I would like to note was that through my experience with the "recommenderlab" package, my RMSE outcome changed drastically. The best-result range was from 1.2 to 3.6. I would highly suspect that it was due to the structure of my data set, as I had to modify the training sets due to function errors from the package. It would be interesting to dig further into this finding in the future.

VII. Results

The results of this project were bad from the view of RMSE perspective but good from the learning perspective. As for RMSE, I really couldn't find a better method to improve the outcome within the time frame that I was able to work. I would love to see other people's method and improve my logic of problem solving, especially because I saw that several people in the discussion panel were able to crack the code and create a model with great RMSE outcomes. On the bright side, I was able to transfer the data sets into the form that I wanted with all the wrangling functions and apply various methods of machine learning. I did spend a lot of time on this and learned a lot, so I would think that this was a great experience. Thank you for reviewing my report and please provide constructive feedback!