HarvardX Movielens Capstone Project

Nick Marum

30/09/2020

INTRODUCTION

This is a Movielens project report for the Harvardx Data Science professional program Capstone course. Inspired by the Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize), this machine learning project will use the 10 million entries from the "Movielens" data set, a publicly available data set user movie ratings, in order to predict movie user ratings based upon previous ratings the user made of other movies. Such an algorithm is the basis for movie recommendation systems used in popular movie streaming services.

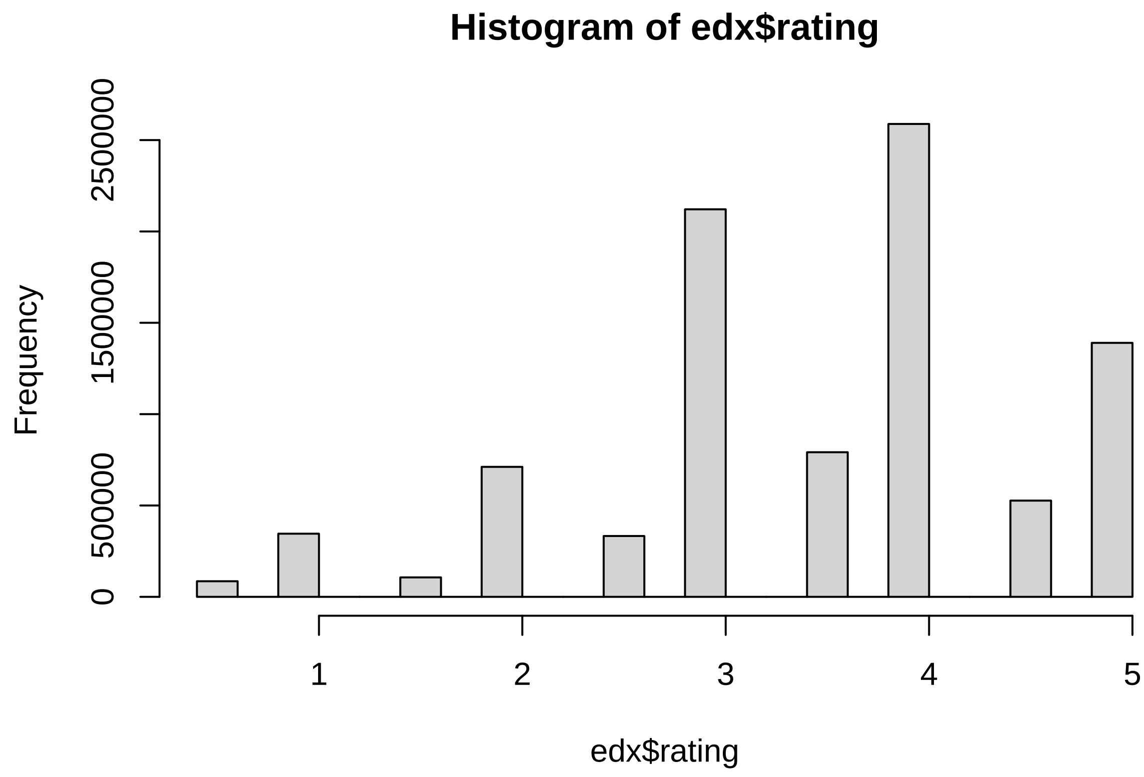
Specifically, this report will demonstrate the steps taken to identify and optimize a matrix factorization-based machine learning algorithm to predict in a randomly selected hold out validation set made up of 10% of the 10 million entries. The root mean squared error (RMSE) of the algorithm against the true values in the validation set will be used as part of the overall scoring of this project. (The validation set is not to be used for the training, evaluation or cross-validation of the algorithm.) This project focused on two packages that use matrix factorization algorithms for the prediction of user ratings: the *recommenderlab* package and the *recosystem* package.

Overview of the dataset

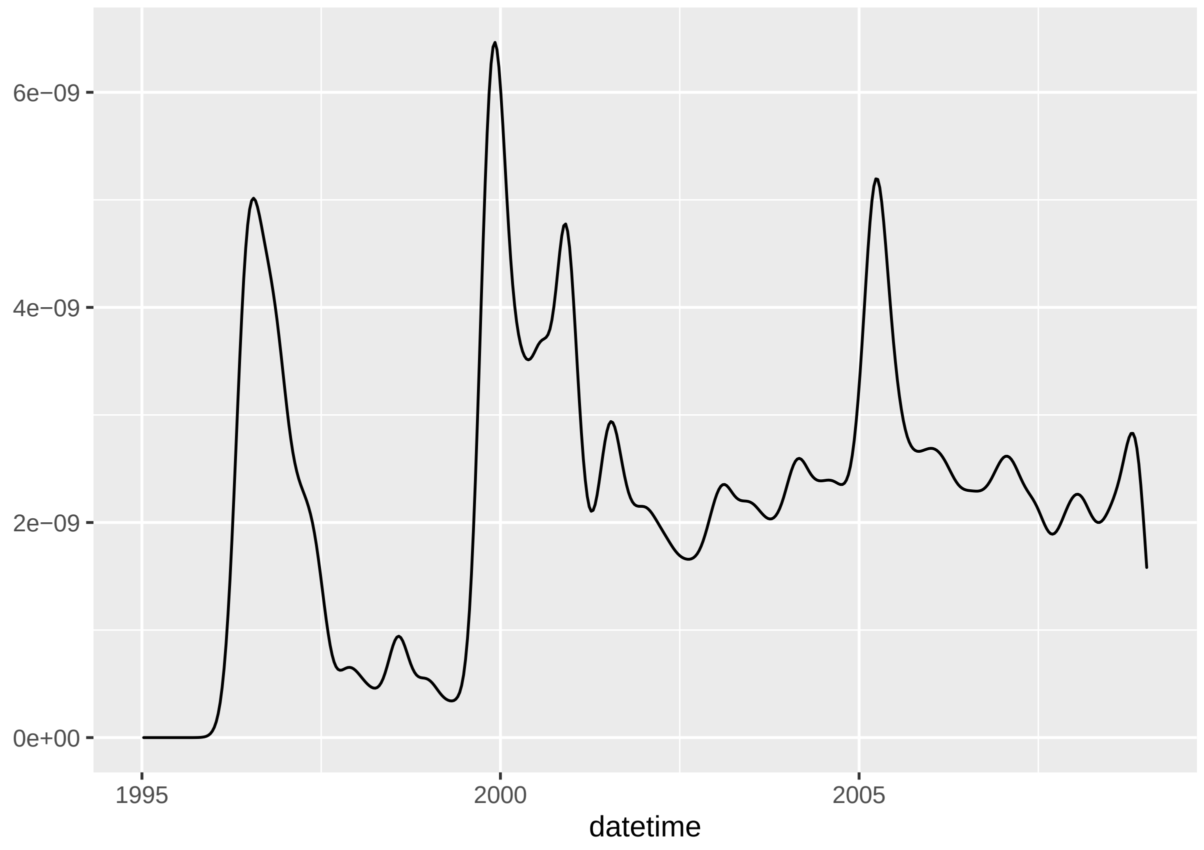
The project instructions require the use of the "edx" object as the training set portion of the project. The original edx set included six columns: "userID", which provide a unique user identification number; "movieID", which provided a unique identification number for each move; "rating", the rating assigned by the user for the movie; and "timestamp", which was the time the user rating was recorded in POSIXct format. Since "timestamp" is not easily interpretable, I added a seventh column, "datetime" which translated the timestamp into an accessible date and time format for each movie rating entry. In all there are little more than 9 million movie rating entries from more than 70,000 users rating about 10,000 movies in the dataset The earliest movie rating entry is from January 1995, and the latest is from January 2009. Ratings were given in .5 increments between .5 to 5 stars (there is no 0 entries in the data set). Ratings half star ratings were generally less common than full star ratings. Four and three stars were the most common ratings, followed by 5 stars and 3.5 stars.

```
##      userId         movieId          rating        timestamp
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08
##  Median :35738   Median : 1834   Median :4.000   Median :1.035e+09
##  Mean   :35870   Mean   : 4122   Mean   :3.512   Mean   :1.033e+09
##  3rd Qu.:53607   3rd Qu.: 3626   3rd Qu.:4.000   3rd Qu.:1.127e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
##     title              genres             datetime
##  Length:9000055     Length:9000055     Min.   :1995-01-09 11:46:49
##  Class :character   Class :character   1st Qu.:2000-01-01 23:11:23
##  Mode  :character   Mode  :character   Median :2002-10-24 21:11:58
##                                        Mean   :2002-09-21 13:45:07
##                                        3rd Qu.:2005-09-15 02:21:21
##                                        Max.   :2009-01-05 05:02:16
```

Ratings were given in .5 increments between .5 to 5 stars (there is no 0 entries in the data set). Ratings half star ratings were generally less common than full star ratings. Four and three stars were the most common ratings, followed by 5 stars and 3.5 stars.



As we can see from the historgram above, the most common rating was 4 stars followed by 3 stars. The least common rating was .5 stars



A density plot of ratings over time shows that while the earliest ratings were registered in 1995, the number of ratings spiked in 1997 and then spiked again in 2000 and maintained a steady stream of ratings until 2009 when data collection stopped.

Methodology and Analysis

The text book materials for the HarvardX machine learning course included a detailed discussion of the Movielens data and went son to examine a number of prediction methodologies, including taking into account user and movie effects as well as regularizing movie effects by minimizing the impact of outlier ratings on less commonly reviewed movies. My intent is to start by recreating the most successful of the methodologies discussed in the text book (Regularized Movie + User Effect model: .881 RMSE) and attempt to improve upon it.

The text book did provide a clue as to how to improve on the model: using matrix factorization to identify latent patterns within the residuals of the Regularized Movie + User effect model (Chapter 33.11 - Matrix Factorization). Using single value decomposition and principle component analysis, the course materials showed there were patterns left in the data in terms of groupings of movies that generally are rated similarly, which could be a basis for further improving on the model. While the textbook did not discuss techniques that would use these estimates to fit a model, it recommended using the recommenderlab package for those students who were interested.

recommenderlab: Lab for Developing and Testing Recommender Algoritms in R

The *recommenderlab* R package version 0.2-6 (https://github.com/mhahsler/recommenderlab) is a framework for the testing and evaluation of recommendation algorithms. The vignette for *recommenderlab* describes that there are various types of recommendation algorithms that are used in commercial applications, the type of algorithms that the package used are "collaborative filtering" algorithms which uses given ratings data from a variety of users on a set of items in order to make predictions of missing ratings - which seems to be an ideal algorithm for the Movielens project. The package includes a number of helpful functions for recommendation tasks, including functions to normalize ratings data, transform ratings data into binary data, and a built-in RMSE function for estimating algorithm performance. *recommenderlab* appears to be one of the most popular purpose built packages for recommendation systems using matrix factorization, however there are other packages as well. For example, *recosystem* uses a different computational approach for recommender system matrix factorization but does not have the same number of features and algorithms as the *recommenderlab* package. We will discuss *recosystem* in more detail later.

User-based and item-based collaborative filtering models are prominently discussed in the *recommenderlab* literature. Both involve making comparisons based on the relationships between users and items (movies) in order to draw inferences and make predictions about what a user would like. There is also other methods based upon single value decomposition (SVD) and matrix factorization that was described in the textbook as a potential way to improve upon model results.

This project will examine collaborative filtering, single value decomposition and other matrix factorization algorithms within the *recommenderlab* and *recosystem* packages to try and improve on the baseline results of the text book Regularized Movie + User effect model.

Analysis

As a first step we will create a small subset of the 100,000 entries from the edx dataset and create a training and a test set.

```
set.seed(200907, sample.kind = "Rounding")#setting seed for replication later
mini <- edx[1:100000] #first 100K entries

ind <- createDataPartition(y= mini$rating, times = 1, p = .2, list = FALSE)
train_mini <- mini[-ind,]
test_mini <- mini[ind,]

test_mini <- test_mini %>%
  semi_join(train_mini, by = "movieId") %>%
  semi_join(train_mini, by = "userId")

summary(train_mini)
```

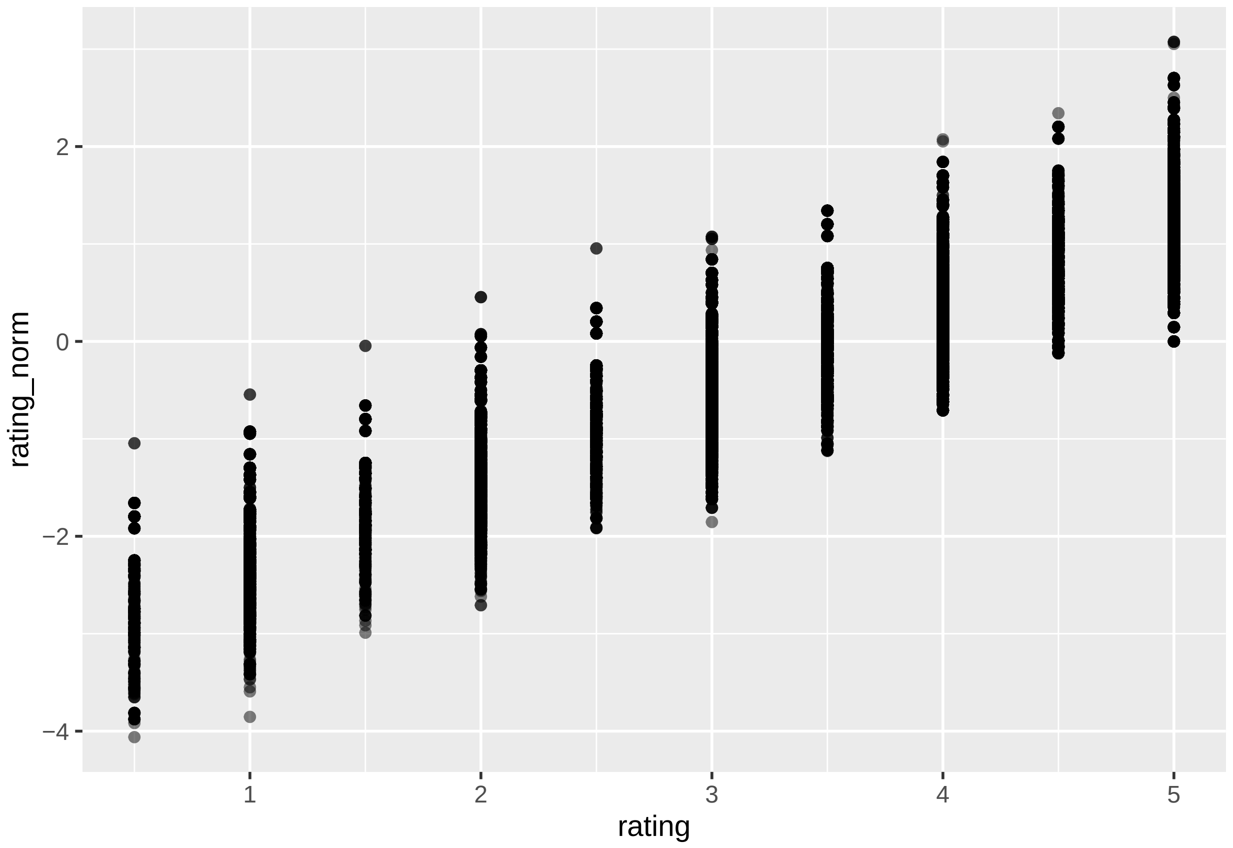
```
##      userId         movieId          rating        timestamp
##  Min.   :  1.0   Min.   :    1   Min.   :0.500   Min.   :8.281e+08
##  1st Qu.:214.0   1st Qu.:  597   1st Qu.:3.000   1st Qu.:9.459e+08
##  Median :426.0   Median : 1639   Median :4.000   Median :1.022e+09
##  Mean   :423.4   Mean   : 3690   Mean   :3.573   Mean   :1.019e+09
##  3rd Qu.:621.0   3rd Qu.: 3264   3rd Qu.:4.000   3rd Qu.:1.112e+09
##  Max.   :843.0   Max.   :64839   Max.   :5.000   Max.   :1.231e+09
##     title              genres             datetime
##  Length:80000       Length:80000       Min.   :1996-03-29 06:16:57
##  Class :character   Class :character   1st Qu.:1999-12-22 17:06:17
##  Mode  :character   Mode  :character   Median :2002-05-23 16:51:44
##                                        Mean   :2002-04-20 17:12:46
##                                        3rd Qu.:2005-03-23 04:10:10
##                                        Max.   :2009-01-04 00:38:19
```

With this training and test set identified, we will start by recreating the baseline model from the textbook. First, We will then normalize the user ratings to adjust for user rating bias by centering each rating by subtracting mean rating score (mu) for each user.

```r
mu <- train_mini %>% group_by(userId) %>%
  summarise(mu = mean(rating))

train_usernorm <- train_mini %>% left_join(mu, by="userId") %>%
  mutate(rating_norm = rating-mu)

train_usernorm %>% group_by(rating) %>% ggplot(aes(rating, rating_norm)) + geom_point(alpha = .5)
```



We will then regularize the movie data to adjust for movie effects. First we will demonstrate the regularization of movie ratings with tuning parameter (lambda). We will then show how we picked the ideal tuning parameter for the regularization.

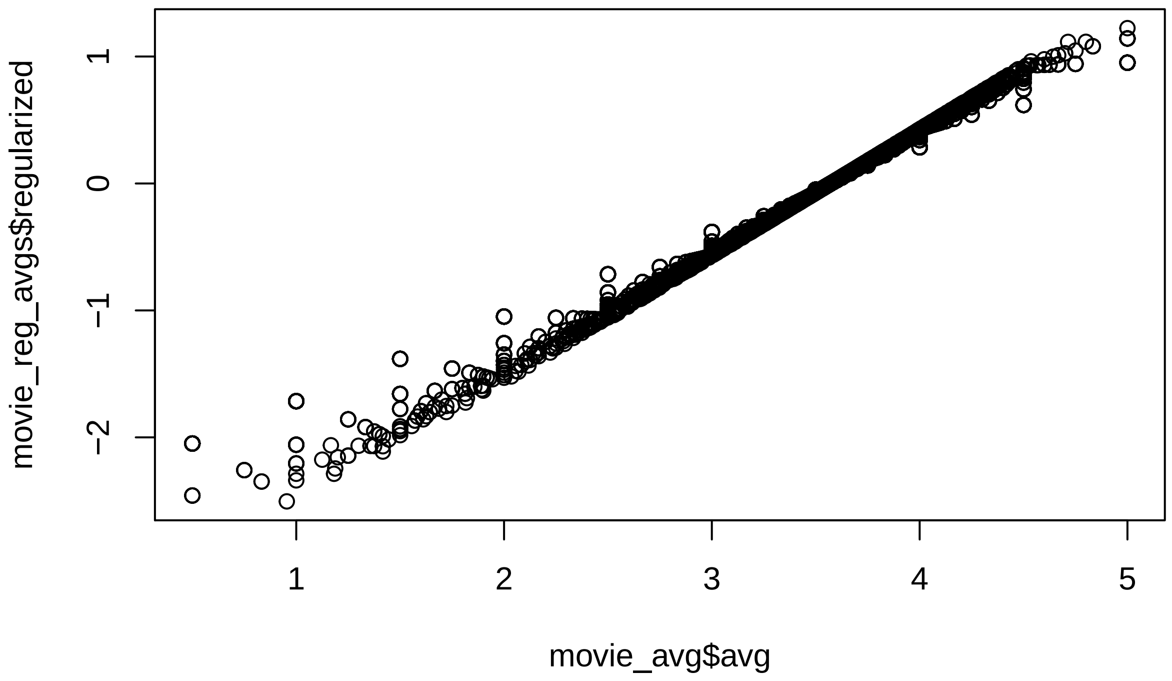
```r
lambda <- .5
mu <- mean(train_mini$rating)
movie_reg_avgs <- train_mini %>%
  group_by(movieId) %>%
  summarize(regularized = sum(rating - mu)/(n()+lambda), n_i = n())
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```r
movie_avg <- train_mini %>%
  group_by(movieId) %>%
  summarise(avg = mean(rating))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

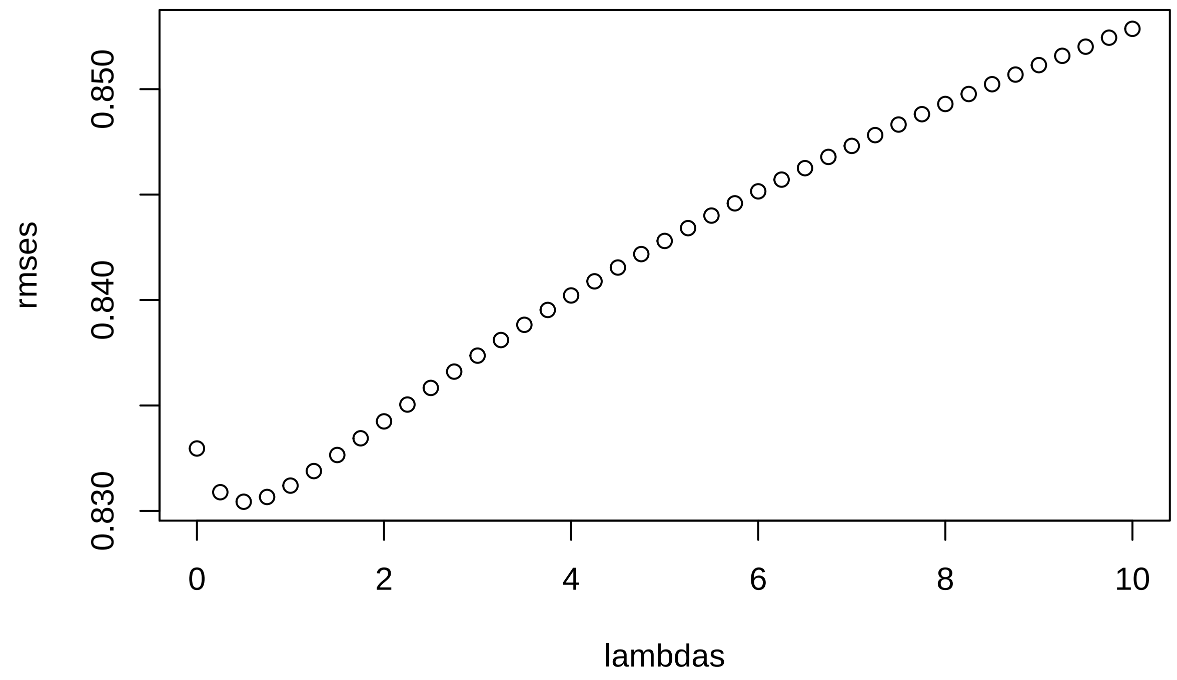
```r
plot(movie_avg$avg, movie_reg_avgs$regularized)
```



As we can see from the above plot, the regularization not only normalizes the data but also moderates some of the extreme ratings (1s and 5s) for films where there is relatively few ratings. I should note that unlike the textbook which regularized both the user and movie rating data, I did not regularize the user data. There was a good case made in the text book for regularizing the data for movie effects for obscure movies where there were relatively few ratings, normalizing the user effect (i.e, the tendency to rate movies in general more positively or negatively due to the personality of the reviewer) seemed to be sufficient.

The lambda parameter can be tuned to achieve the best results in terms of root mean squared error. Below is the process for selecting the optimal lambda. Note that we a selecting the optimized lambda based purely on the training data, but will use the test data to evaluate the overall performance.

```
## [1] 0.5
```



Using the training set data, we can see the optimized lambda is .5. We will use this parameter in order to make predictions using the textbook based model on the test set.

```
mu <- mean(train_mini$rating) #overall training wet average movie rating
l_opt <- .5 #lambda selected from training set optimization

  b_i <- train_mini %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l_opt)) #regularized movie effect
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
  b_u <- train_mini %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = mean(rating - mu)) # normalized user effect
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
  predicted_ratings <-
    test_mini %>% #using test set for prediction
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
#prediction based on overall set average, user and movie effects
```

```r
    pull(pred)

RMSE(test_mini$rating, predicted_ratings)
```

```
## [1] 0.8923301
```

Using the training set for optimizing the model and only using the test set for evaluation, can see that the overall performance is a little poorer than what was identified in the text book (.892 rather than .881). However if the test data is used to optimize the parameter (which was what was done in the textbook), the results is quite similar to textbook regularized effects model as we can see below.

```r
lambdas <- seq(0, 10, 0.25)

rmses <- sapply(lambdas, function(l){

  mu <- mean(train_mini$rating)

  b_i <- train_mini %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))

  b_u <- train_mini %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = mean(rating - mu))

  predicted_ratings <-
    test_mini %>% # note we are now using the test set for the predictions
    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, test_mini$rating))#RMSE based on test set
})

mu <- mean(train_mini$rating)
l <- lambdas[which.min(rmses)] #selecting the lambda with the best RMSE score

  b_i <- train_mini %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))

  b_u <- train_mini %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = mean(rating - mu))

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

    RMSE(predicted_ratings, test_mini$rating)
```

```
## [1] 0.8819273
```

Using the test data, we find that the optimal lambda parameter is 4.25. This gives us a similar RMSE result as the textbook.

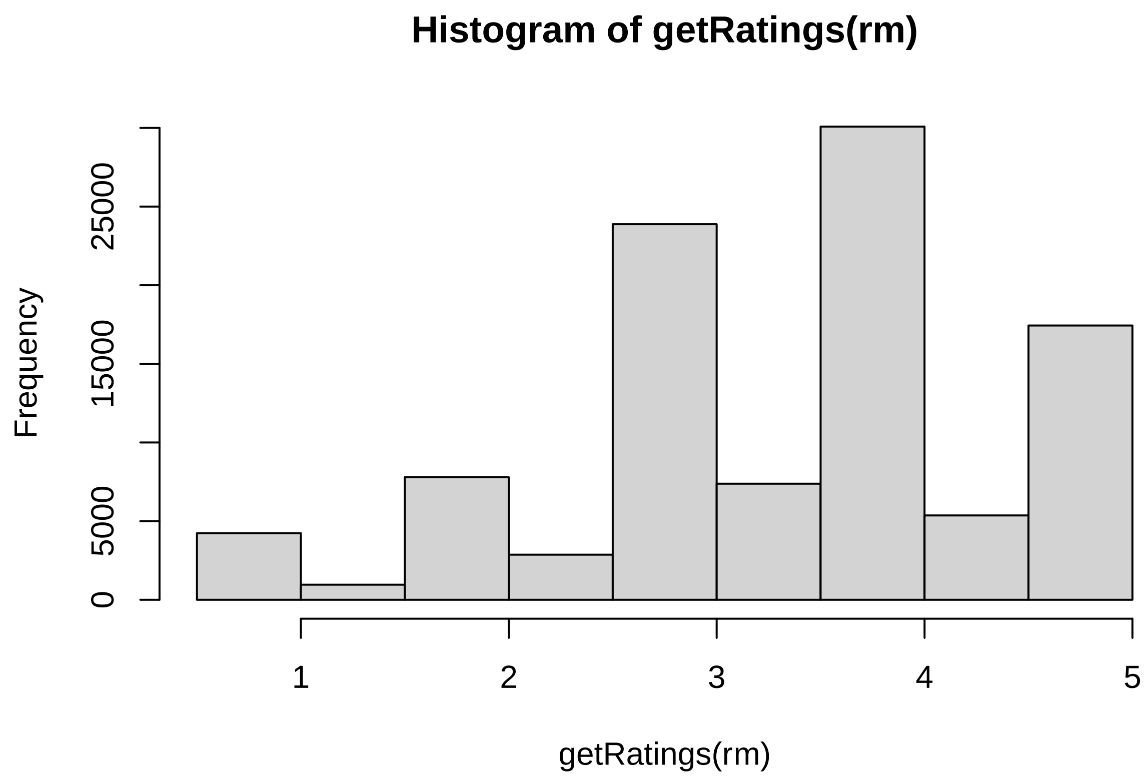
Now that we have our baseline model, we will explore the algorithms and tools available in the *recommenderlab* package.

recommenderlab Collaborative Filtering and SVD-based models

The first step is to import a data frame into the *recommenderlab* "realRatingMatrix" format. In order to use *recommenderlab* tools, a rating matrix needs to be created so that each user is represented by a single row and that each item (or film in this case) is a column. Given most users will only rate a handful of potential movies, this creates a "sparse" matrix comprised primarily of NAs that *recommenderlab* stores is a less memory intensive way for computations.

```
rm <- as(mini, "realRatingMatrix")

hist(getRatings(rm), breaks = 15)
```

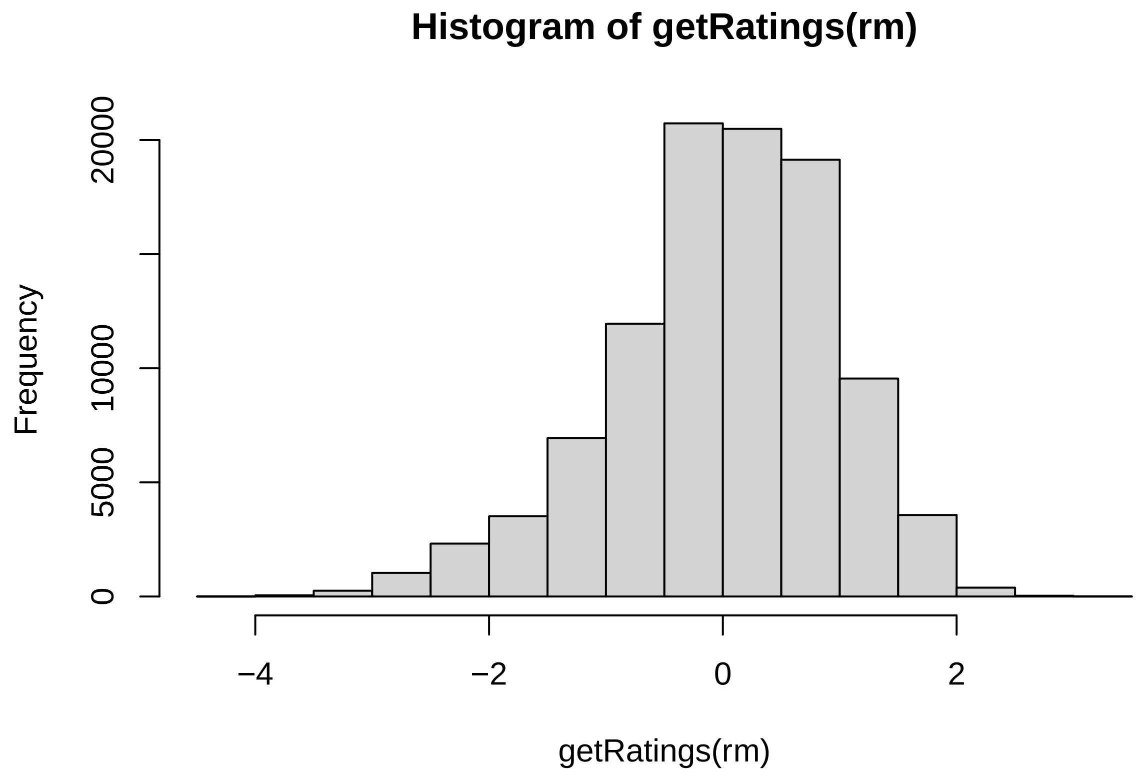


We have imported our mini data set into the *recommenderlab* environment. As you can see from the code for the histogram above, the environment uses a number of its own functions and unique class of objects.

We will also take a look at using some of the built in functionality in *recommenderlab* to normalize the data, which is a recommended step before applying the recommender models built into the package.

```r
rm <- normalize(rm) #normalize function which normalizes data by row (i.e., user)

hist(getRatings(rm))
```



There are a number of different recommender algorithms in the *recommenderlab* that can be accessed using the command: recommenderRegistry$get_entries(dataType = "realRatingMatrix"). It includes single value decomposition (SVD) based methods in addition to the Item-based Collaborative Filtering (IBCF) and User-based Collaborative Filtering (UBCF) methods.

We will try using an evaluation methodology (evaluationScheme) built into the package to train and evaluate a "recommender" model.

```r
e <- evaluationScheme(rm, method="split", train=.9, given=10, goodRating=5)
#building an evalutation scheme object from "realRatingMatrix".  90% of matrix will be used for trainin
#The given parameter is how many entries will be given to the model per user.
#It is an adjustable parameter with a default of 10.
#The "goodrating" parameter is simply to identify what a top rating would be under the model.

IBCF <- Recommender(getData(e, "train"), method="IBCF")
#training recommender model (IBCF) using the training set from the evaluation scheme "e".
IBCF_preds <- predict(IBCF, getData(e, "known"), type="ratings")
#using the "known" user and item values from the test set in the evaluation scheme to create prediction
IBCF_acc <- calcPredictionAccuracy(IBCF_preds, getData(e, "unknown"))
```

```
#accuracy is measured using the "calcPredictionAccuracy" function against the "unknown" part of the tes

UBCF <- Recommender(getData(e,"train"), method="UBCF")
UBCF_preds <- predict(UBCF, getData(e, "known"), type="ratings")
UBCF_acc <- calcPredictionAccuracy(UBCF_preds, getData(e, "unknown"))

SVDF <- Recommender(getData(e, "train"), method="SVDF")
SVDF_preds <- predict(SVDF, getData(e, "known"), type="ratings")
SVDF_acc <- calcPredictionAccuracy(SVDF_preds, getData(e, "unknown"))

SVD <- Recommender(getData(e, "train"), method="SVD")
SVD_preds <- predict(SVD, getData(e, "known"), type="ratings")
SVD_acc <- calcPredictionAccuracy(SVD_preds, getData(e, "unknown"))

data.frame(IBCF_acc, UBCF_acc, SVDF_acc, SVD_acc)
```

```
##       IBCF_acc  UBCF_acc  SVDF_acc  SVD_acc
## RMSE 1.3318539 1.1156708 0.9833164 1.031212
## MSE  1.7738347 1.2447213 0.9669112 1.063398
## MAE  0.9300847 0.8670055 0.7679187 0.812314
```

```
#building and printing a data frame with results of each method.
```

Much to my surprise, the collaborative filtering methods I assumed would be so powerful based on the literature did not do nearly as well as I had hoped. The Funk Single Value Decomposition (SVDF) appears to be the model that provides the best accuracy, however the accuracy is much inferior to the much simpler baseline model we developed. The SVD approximation method also did well and was much less computationally intensive and generated results much faster. It is possible that a larger dataset could improve the performance of the collaborative filtering and single value decomposition models, however tuning of the models could also improve results substantially.

Lets see if we can evaluate it in a different way using training and test sets and if it is possible to tune the performance.

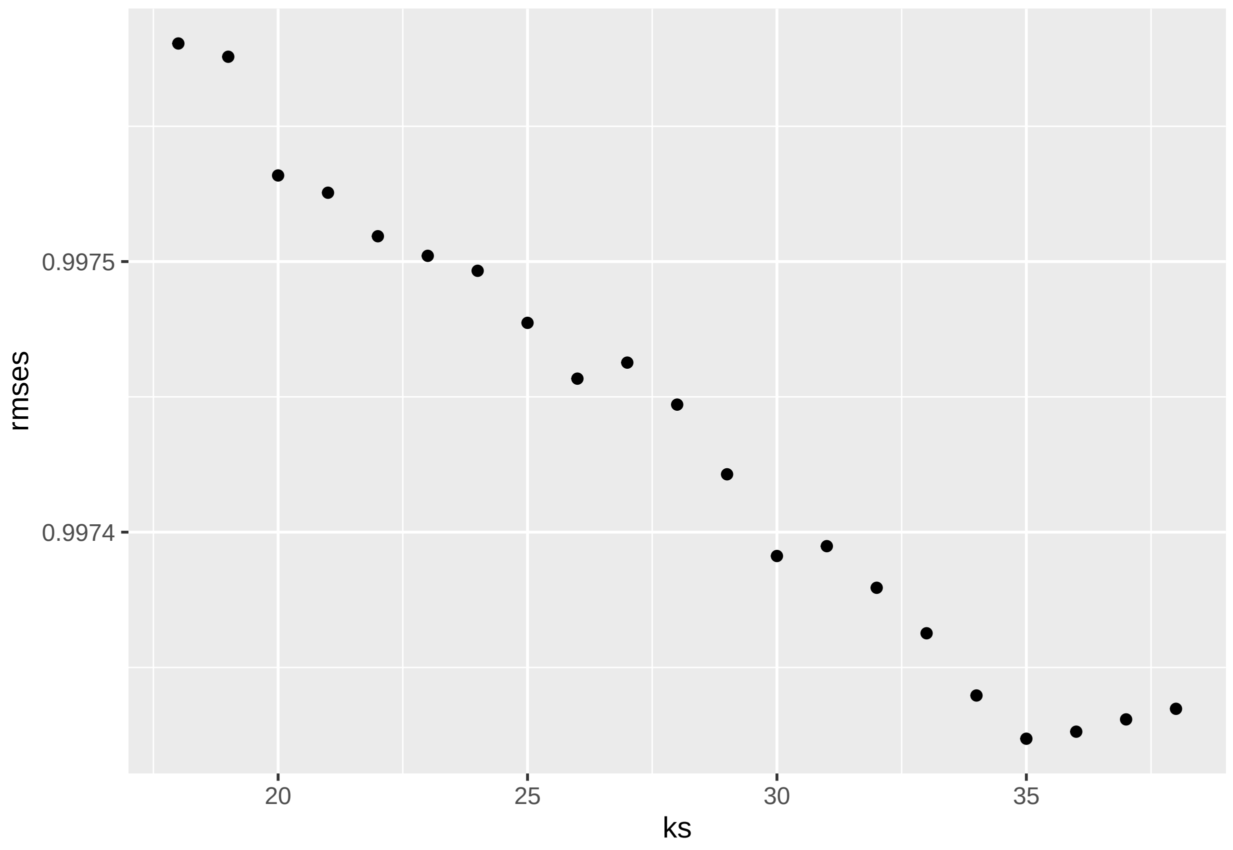
```
##   SVD.k SVD.maxiter SVD.normalize
## 1    10         100        center
```

Opening up and inspecting the registry entries for the SVD algorithm shows that there are essentially three parameters that are open to tuning. “k” as in nearest neighbors in terms of features, which has a default value of 10, “maxiter” which has a default value of 100, and “normalize” which has a default value of “center”. Presumably the normalize can also be z-score normalized (or “scaled”) which is another normalization technique described in the *recommenderlab* documentation.

Lets try by tuning the SVD model by tuning for “k”.

```
## SVD run fold/sample [model time/prediction time]
##   1  [2.8sec/0.66sec]
##   2  [2.77sec/0.48sec]
##   3  [3.05sec/0.78sec]
##   4  [2.6sec/0.58sec]
##   5  [2.7sec/0.51sec]
##   6  [2.74sec/0.56sec]
##   7  [2.72sec/0.66sec]
##   8  [2.6sec/0.59sec]
##   9  [2.66sec/0.54sec]
##   10  [2.55sec/0.54sec]
## SVD run fold/sample [model time/prediction time]
##   1  [2.86sec/0.6sec]
##   2  [2.75sec/0.53sec]
##   3  [2.89sec/0.95sec]
##   4  [2.78sec/0.6sec]
##   5  [2.82sec/0.56sec]
##   6  [2.61sec/0.56sec]
##   7  [2.83sec/0.56sec]
##   8  [2.64sec/0.54sec]
##   9  [2.64sec/0.52sec]
##   10  [3.4sec/0.51sec]
## SVD run fold/sample [model time/prediction time]
##   1  [2.78sec/0.55sec]
##   2  [2.95sec/0.52sec]
##   3  [2.87sec/0.78sec]
##   4  [2.88sec/0.51sec]
##   5  [2.75sec/0.54sec]
##   6  [2.72sec/0.54sec]
##   7  [2.63sec/0.64sec]
##   8  [2.68sec/0.53sec]
##   9  [2.88sec/0.57sec]
##   10  [3.02sec/0.66sec]
## SVD run fold/sample [model time/prediction time]
##   1  [3.02sec/0.53sec]
##   2  [2.89sec/0.59sec]
##   3  [2.99sec/0.75sec]
##   4  [3.18sec/0.53sec]
##   5  [3.02sec/0.52sec]
##   6  [2.74sec/0.52sec]
##   7  [2.69sec/0.65sec]
##   8  [3.1sec/0.57sec]
##   9  [3.02sec/0.53sec]
##   10  [3.01sec/0.54sec]
## SVD run fold/sample [model time/prediction time]
##   1  [3.01sec/0.58sec]
##   2  [3.12sec/0.5sec]
##   3  [3.19sec/0.8sec]
##   4  [3.1sec/0.54sec]
##   5  [2.9sec/0.57sec]
##   6  [3.14sec/0.52sec]
##   7  [3.05sec/0.73sec]
##   8  [3.22sec/0.5sec]
##   9  [3.06sec/0.53sec]
##   10  [3.09sec/0.55sec]
## SVD run fold/sample [model time/prediction time]
##   1  [2.84sec/0.54sec]
##   2  [3.11sec/0.58sec]
##   3  [3.56sec/0.91sec]
##   4  [3.33sec/0.53sec]
##   5  [3.01sec/0.58sec]
##   6  [3.6sec/0.53sec]
##   7  [3.17sec/0.74sec]
##   8  [3.61sec/0.54sec]
##   9  [3.37sec/0.55sec]
##   10  [3.43sec/0.53sec]
## SVD run fold/sample [model time/prediction time]
##   1  [3.23sec/0.59sec]
##   2  [3.28sec/0.54sec]
##   3  [3.62sec/0.76sec]
##   4  [3.35sec/0.63sec]
##   5  [3.2sec/0.54sec]
##   6  [3.45sec/0.57sec]
##   7  [3.23sec/0.62sec]
##   8  [3.43sec/0.53sec]
##   9  [3.48sec/0.6sec]
##   10  [3sec/0.56sec]
## SVD run fold/sample [model time/prediction time]
##   1  [3.14sec/0.55sec]
##   2  [4.01sec/0.54sec]
##   3  [3.46sec/1.46sec]
##   4  [3.68sec/0.51sec]
##   5  [3.65sec/0.65sec]
##   6  [3.78sec/0.53sec]
##   7  [4.02sec/0.69sec]
##   8  [3.62sec/0.53sec]
##   9  [3.66sec/0.55sec]
##   10  [3.56sec/0.74sec]
## SVD run fold/sample [model time/prediction time]
##   1  [3.87sec/0.63sec]
##   2  [3.87sec/0.53sec]
##   3  [3.45sec/0.78sec]
##   4  [3.55sec/0.59sec]
##   5  [3.64sec/0.65sec]
##   6  [3.71sec/0.55sec]
##   7  [3.56sec/0.72sec]
##   8  [3.7sec/0.59sec]
##   9  [4.09sec/0.53sec]
##   10  [3.81sec/0.59sec]
## SVD run fold/sample [model time/prediction time]
##   1  [3.79sec/0.57sec]
##   2  [4.02sec/0.62sec]
##   3  [4.08sec/0.81sec]
##   4  [3.79sec/0.59sec]
##   5  [3.79sec/0.53sec]
##   6  [3.69sec/0.58sec]
##   7  [3.83sec/0.62sec]
##   8  [3.78sec/0.58sec]
##   9  [3.53sec/0.57sec]
##   10  [3.84sec/0.57sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4sec/0.56sec]
##   2  [3.95sec/0.57sec]
##   3  [3.96sec/0.87sec]
##   4  [3.68sec/0.55sec]
##   5  [4.13sec/0.57sec]
##   6  [3.97sec/0.62sec]
##   7  [4.24sec/0.63sec]
##   8  [3.78sec/0.58sec]
##   9  [4.47sec/0.57sec]
##   10  [3.74sec/0.61sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4.25sec/0.57sec]
##   2  [4.02sec/0.63sec]
##   3  [4.03sec/0.8sec]
##   4  [3.99sec/0.58sec]
##   5  [4.31sec/0.63sec]
##   6  [3.95sec/0.61sec]
##   7  [4.33sec/0.73sec]
##   8  [4.01sec/0.62sec]
##   9  [4.18sec/0.56sec]
##   10  [3.91sec/0.61sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4.19sec/0.57sec]
##   2  [4.38sec/0.67sec]
##   3  [4.21sec/0.83sec]
##   4  [3.9sec/0.53sec]
##   5  [4.2sec/0.58sec]
##   6  [4.07sec/0.62sec]
##   7  [4.6sec/1.53sec]
##   8  [3.97sec/0.59sec]
##   9  [4.28sec/0.66sec]
##   10  [4.35sec/0.56sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4.46sec/0.67sec]
##   2  [4.28sec/0.59sec]
##   3  [4.64sec/0.89sec]
##   4  [4.3sec/0.59sec]
##   5  [4.69sec/0.56sec]
##   6  [4.15sec/0.65sec]
##   7  [4.53sec/0.69sec]
##   8  [5.12sec/0.58sec]
##   9  [4.21sec/0.59sec]
##   10  [4.16sec/0.59sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4.05sec/0.6sec]
##   2  [4.53sec/0.66sec]
##   3  [4.5sec/0.82sec]
##   4  [4.79sec/0.63sec]
##   5  [4.13sec/0.61sec]
##   6  [4.27sec/0.58sec]
##   7  [4.87sec/0.67sec]
##   8  [4.34sec/0.63sec]
##   9  [4.35sec/0.57sec]
##   10  [4.26sec/0.57sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4.75sec/0.68sec]
##   2  [4.49sec/0.61sec]
##   3  [5.26sec/0.82sec]
##   4  [4.52sec/0.62sec]
##   5  [4.77sec/0.62sec]
##   6  [4.56sec/0.58sec]
##   7  [4.49sec/0.77sec]
##   8  [4.67sec/0.59sec]
##   9  [4.49sec/0.57sec]
##   10  [4.91sec/0.61sec]
## SVD run fold/sample [model time/prediction time]
##   1  [4.7sec/0.65sec]
##   2  [4.48sec/1.28sec]
##   3  [5.16sec/0.82sec]
##   4  [4.86sec/0.62sec]
##   5  [4.43sec/0.61sec]
##   6  [5.08sec/0.63sec]
##   7  [5.11sec/0.71sec]
##   8  [4.95sec/0.62sec]
##   9  [4.55sec/0.65sec]
##   10  [4.67sec/0.61sec]
## SVD run fold/sample [model time/prediction time]
##   1  [5.58sec/0.62sec]
##   2  [5.22sec/0.6sec]
##   3  [5.14sec/0.89sec]
##   4  [5.73sec/0.62sec]
##   5  [5.21sec/0.68sec]
##   6  [4.78sec/0.61sec]
##   7  [4.85sec/0.8sec]
##   8  [5.04sec/0.6sec]
##   9  [5.08sec/0.61sec]
##   10  [5.63sec/0.67sec]
## SVD run fold/sample [model time/prediction time]
##   1  [5.2sec/0.63sec]
##   2  [5.13sec/0.66sec]
##   3  [5.14sec/0.89sec]
##   4  [5.59sec/0.61sec]
##   5  [4.73sec/0.64sec]
##   6  [4.99sec/0.62sec]
##   7  [4.7sec/0.71sec]
##   8  [4.85sec/0.61sec]
##   9  [5.36sec/0.68sec]
##   10  [5.43sec/0.61sec]
## SVD run fold/sample [model time/prediction time]
##   1  [5.48sec/0.6sec]
##   2  [5.43sec/0.64sec]
##   3  [5.43sec/1.55sec]
##   4  [5.5sec/0.6sec]
##   5  [5.28sec/0.6sec]
##   6  [5.82sec/0.63sec]
##   7  [5.02sec/0.71sec]
##   8  [5.18sec/0.73sec]
##   9  [5.02sec/0.63sec]
##   10  [5.26sec/0.71sec]
## SVD run fold/sample [model time/prediction time]
##   1  [5.24sec/0.61sec]
##   2  [5.23sec/0.7sec]
##   3  [5.49sec/0.84sec]
##   4  [5.62sec/0.59sec]
##   5  [5.27sec/0.71sec]
##   6  [5.79sec/0.64sec]
##   7  [5.4sec/0.69sec]
##   8  [5.57sec/0.65sec]
##   9  [5.57sec/0.59sec]
##   10  [5.69sec/0.63sec]
```

```
## [1] 35
```



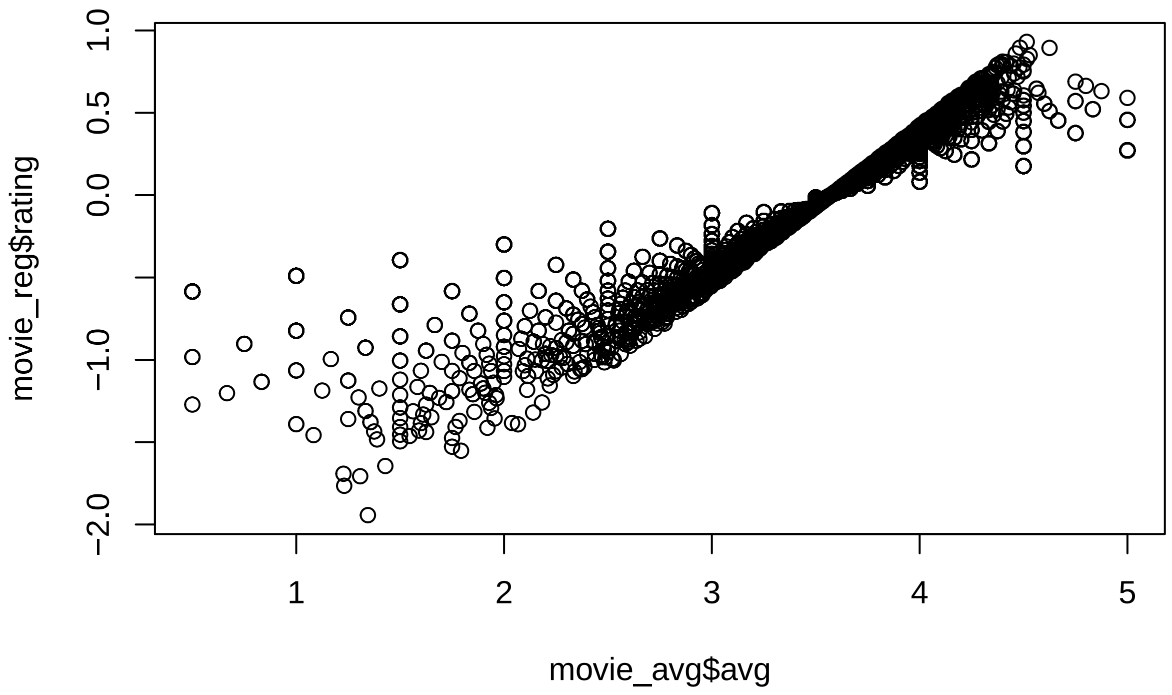
After a series of trial and error of different sequences of integer values for “k”, I was finding a somewhat linear trend of better performance due to higher level of k values, approximately 30 being the maximum value - much more than the default of 10. This raised raised concerns of overtraining for me. As such, using the evaluation scheme of *recommenderlab* I introduced k-fold cross-validation which suggested an optimal value of 35 which provided an RMSE of .9973. A slight improvement.

Using this knowledge gained from the computationally less intensive SVD model, I would like to see how much we can improve the SVDF model by simply inputing the optimal the optimal SVD k-value of 35.

After an extremely long processing time and using a k parameter of 35, the resulting RMSE is .8986. Still short of the .8819 of our baseline model but getting closer. Additionally tuning could refine the result, but it would take a significant amount of time with relatively little improvement as we saw with the SVD model tuning. This is still way short of the grading-scale full-points RMSE score of .8649.

However, given what we learned through the textbook of the importance of not only normalizing data, but regularizing the data so as not to allow it to be skewed by few entries, and how that significantly improved our baseline model, I tried doing the same thing here.

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

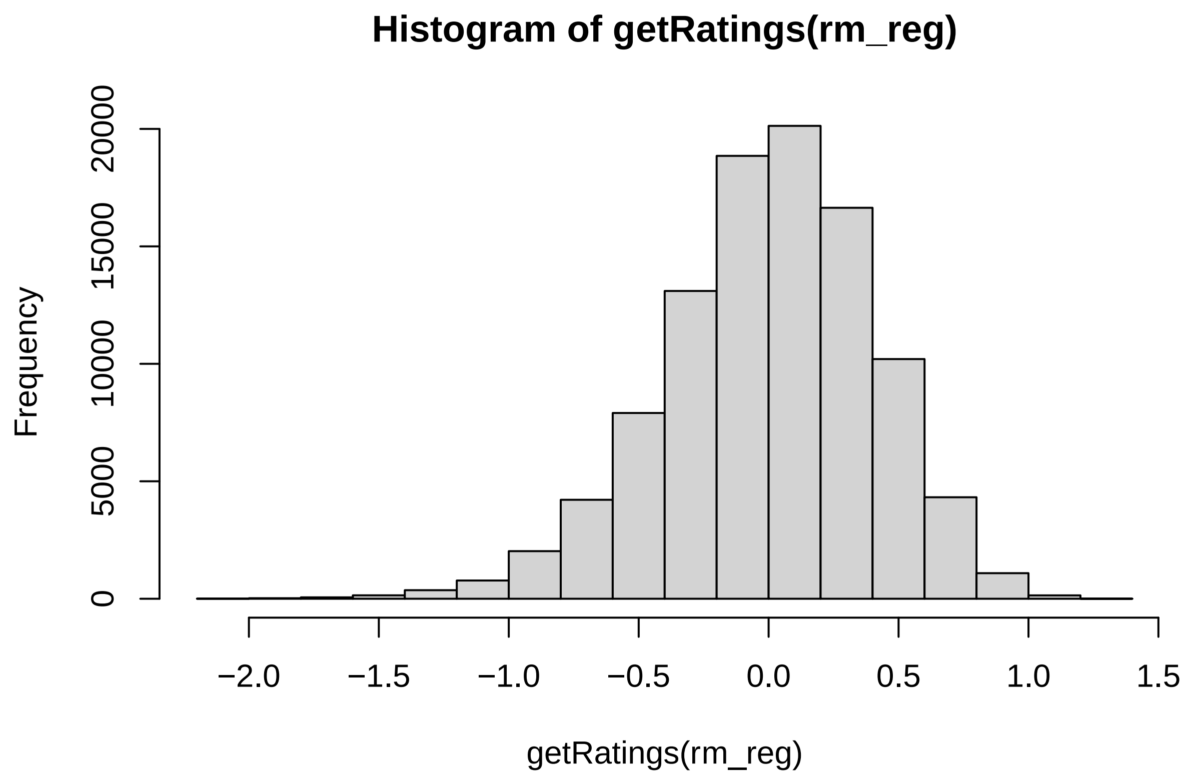


Above is showing the effect of the regularization using a larger lambda parameter (4.25) identified through tuning the baseline model against the test data set. You can see a much more significant impact on the scores.

```
mu <- mean(train_mini$rating)
mini_reg <- mini %>%
  group_by(movieId) %>%
  mutate(rating = sum(rating-mu)/(n()+4.25)) %>%
  ungroup()

rm_reg <- as(as.data.frame(mini_reg), "realRatingMatrix")
rm_reg <- normalize(rm_reg) #normalize entries by row (e.g., user)

hist(getRatings(rm_reg))
```



As expected, ratings are centered around zero. A little more tight than just normalizing the data by user (row).

Examining a Sparse Rating Matrix

We will take a moment here to take a closer look at the nature of the sparse matrices we are working with. If we examine the rating matrix as a data frame we can explore a little more closely how the data is captured.

```
head(as(rm_reg, "data.frame"))
```

```
##       user item      rating
## 5724     1  122 -0.31984384
## 7913     1  185 -0.25738134
## 11644    1  292  0.07494152
## 12779    1  316 -0.05542891
## 13537    1  329 -0.07488610
## 14984    1  355 -0.68330431
```

We can see the top rating of the first 6 rows are all from the same user, user “1”. We see that item 292 is the top rating of the 6 films rated by this user while the lowest ratings are item 355 and item 122. Let’s inspect these items closer to see what we are dealing with.

```
c(mini_reg$title[292], mini_reg$title[122], mini_reg$title[355])
```

```
## [1] "Bridget Jones's Diary (2001)"
## [2] "Three Colors: White (Trois couleurs: Blanc) (1994)"
## [3] "Sum of Us, The (1994)"
```

The top rated film is the first entry in the data frame above, while the lowest rankings are the bottom two.

All three are romantic films, however Bridget Jones's Diary, the top rated film of the 6, was a romantic comedy blockbuster while the other two are more "art house" films. "Three Colors: White" is a french language romantic drama and "The Sum of Us" is a Australian LGBT romantic comedy. This all seems consistent with the film choices of a person who likes to watch romantic comedies however who prefers more popular Hollywood films over art house features.

Now We will take a look at a particular user to get a sense for the matrix. In this case I have have chosen user "102".

```
sub_set <- rm_reg[102]
as(sub_set, "matrix")[,1:50]
```

```
##          1          2          3          4          5          6          7
##         NA         NA         NA         NA         NA         NA         NA
##          8          9         10         11         12         13         14
##         NA         NA -0.1973067         NA         NA         NA         NA
##         15         16         17         18         19         20         21
##         NA         NA         NA         NA         NA         NA         NA
##         22         23         24         25         26         27         28
##         NA         NA         NA         NA         NA         NA         NA
##         29         30         31         32         33         34         35
##         NA         NA         NA         NA         NA         NA         NA
##         36         38         39         40         41         42         43
##         NA         NA         NA         NA         NA         NA         NA
##         44         45         46         47         48         49         50
##         NA         NA         NA         NA         NA         NA  0.8324942
##         52
##         NA
```

You can see that for user entry 102, the matrix is pretty sparse. Only two entries in the first 50 columns (films). This was actually better than most (and the reason why I picked 102) as generally speaking users did not rate any of the first 50 films.

The scores have been normalized and regularized for user and file effects, so rather than seeing a raw rating of .5 to 5, we are seeing the adjusted scores. Positive scores reflected higher than average rated movies and negative scores reflect lower than average scoring for that user.

```
max <- which.max(as(sub_set, "matrix"))
min <- which.min(as(sub_set, "matrix"))

c(as(sub_set, "matrix")[,max], as(sub_set, "matrix")[,min])
```

```
## [1]  0.8676348 -0.8427481
```

We can see the range of scores above for user "102". They are centered around 0 and it appears the user doesn't necessarily skew his scores one way or another.

Attempting to optimize the Baseline Model

Now that we have examined the baseline model data more closely, we will seek to apply the optimized k value using the SVD approximation model to see if we can get an improved score.

```r
e_reg <- evaluationScheme(rm_reg, method="cross", k=10, given=10, goodRating=5)
SVD_opt <- Recommender(getData(e_reg, "train"), method="SVD", para = list(k=35))
SVD_preds <- predict(SVD_opt, getData(e_reg, "known"), type="ratings")
calcPredictionAccuracy(SVD_preds, getData(e_reg, "unknown"))
```

```
##      RMSE       MSE       MAE
## 0.4059420 0.1647889 0.3197639
```

Wow! RMSE score of .43! This was initially exciting but it also raised a red flag. I realized in normalizing and regularizing the data I did so to the entire rating matrix, including to the values that make up the test set, before building my evaluation object. Naturally modifying these values would have an effect on a the calculated RMSE. The key to solving this appeared to be to try and apply this approach to objects other than the built in evaluation scheme that *recommenderlab* uses, however that is where I ran into significant challenges.

While this SVD matrix factorization approach appeared very promising and seems to produce significant improvements to the baseline model , the *recommenderlab* environment is somewhat self-contained, designed to use its own tools like the evaluation scheme for training and evaluating models. It does not easily apply to the edx and validation data sets separately. I tried to run the model developed in the *recommenderlab* environment on the code on the larger edx object and a holdout dataset and I ran into memory errors (for example, an error regarding R's inability to handle a multi-gigabyte sized vector). It simply would not run. Very unfortunate and discouraging after having invested so much time in the package.

Before completely abandoning the matrix factorization approach, I explored other packages that may be able to use a similar approach but are not in such a self contained environment that the parameters of this project would prevent me from using them.

recosystem: Recommender System Using Parallel Matrix Factorization

Through monitoring the Movielens course bulletin boards I saw references to the *recosystem* package. It is similar to the *recommenderlab* package in that it provides tools specifically for recommendation systems and leverages matrix factorization-based algorithms. Also, it has some features that are specifically designed to limit the computational and memory load. While it uses sparse matrices and matrix factorization in its computations, it is also to easily generate a vector of predictions in the same tidy format as the edx and validation data frames.

The *recosystem* package does not have the same variety of recommender algorithms and built-in functions for the examination of sparse rating matrix data as *recommenderlab*. The package is a user-friendly R wrapper of a LIBMF library of various high-performance C++ computational approaches to large scale matrix factorization. It includes matrix factorization algorithms for binary, real value and other types of rating matrices. More details on the package can be found at https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html.

Matrix factorization involves attempting to approximate an entire matrix through the product of lower dimension matrices. The method is closely related to principal component analysis and single value decomposition demonstrated in the text book as ways to break down latent effects in highly dimension datasets by identifying correlations within the sets that can explain the variability of the data. Model training is conducted through the process of solving the matrices and using a loss function based on observed ratings in order to optimize the model as well as penalty parameters to avoid overfitting.

Similar to *recommenderlab* the *recosystem* has built in functionality to transform ratings data into a sparse matrix. For training of models, the package requires the data in a triplet form with each row containing three numbers: the user index, the item index, and the rating. In the edx file this corresponds with the userId, movieId and Rating entries. Since ratings in testing data are often unknown, the rating entry can be ommitted in creating a testing data file in *recosystem*. Training and testing data is loaded either through the data_memory() function which creates an object in the local environment, or the data_file() function, which creates a file in the working directory with the data.

With the data in hand, the first step is to create a model object by calling the Reco() function from *recosystem*. This object and the various training, tuning and prediction methods operate a bit differently than the regular tidyverse environment. The object itself is used to call the training, tuning and prediction functions using the assessor "$" and then the function and relevant arguments built into the call.

Some initial testing and tuning of the *recosystem* recommender model is shown below.

```
train <- data_memory(train_mini$userId, train_mini$movieId, rating = train_mini$rating)
#Inputting training data into a recosystem sparse matrix object

r = Reco() #creating the recommender object: "r"

r$train(train_data = train)
```

```
## iter      tr_rmse          obj
##    0       1.3655   2.0840e+05
##    1       0.9010   1.2392e+05
##    2       0.8677   1.1903e+05
##    3       0.8516   1.1671e+05
##    4       0.8398   1.1546e+05
##    5       0.8269   1.1394e+05
##    6       0.8144   1.1284e+05
##    7       0.8021   1.1162e+05
##    8       0.7922   1.1058e+05
##    9       0.7849   1.1002e+05
##   10       0.7783   1.0931e+05
##   11       0.7730   1.0874e+05
##   12       0.7686   1.0837e+05
##   13       0.7652   1.0808e+05
##   14       0.7617   1.0768e+05
##   15       0.7590   1.0735e+05
##   16       0.7569   1.0732e+05
##   17       0.7541   1.0698e+05
##   18       0.7522   1.0674e+05
##   19       0.7508   1.0670e+05
```

```
#Training of LIBMF model - notice call is used from recommender object

test <- data_memory(test_mini$userId, test_mini$movieId)
#Test set userId and movieId data inputed into recosys environment for
#prediction.  Note that the actual ratings are not entered into object.
#The test set actual ratings will only be used for validating model prediction.

r$predict(test_data = test)
```

```
## prediction output generated at predict.txt
```

```
#prediction function for recosys based on validation data - it
#produces a file in the working directory that contains predictions. This can take a while.

pred <-read.csv("predict.txt", header = FALSE)
#reading the prediction file into the local environment.  Predictions are in CSV
#format without a header.
```

```
RMSE(test_mini$rating, pred$V1)
```

```
## [1] 0.8654353
```

Before any tuning was conducted, the model is already providing an improved RMSE score than the baseline model and the computation time was only a few moments rather than the many minutes spent on earlier matrix factorization based algorithms.

The *recosystem* does not have built in functionality to normalize ratings data the way that the previous package did, so the user and film effects are still present. If we adjust for those by leveraging our baseline model of user and film effects, we should see an improvement.

```
b_u <- train_mini %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
#calculating the user bias

train_baseline <- train_mini %>%
  left_join(b_u, by="userId") %>%
  mutate(rating_norm = rating-b_u)
 #user effect is adjusted through normalization

train <- data_memory(train_baseline$userId, train_baseline$movieId, rating = train_baseline$rating_norm
#Inputting training data into a recosystem sparse matrix object

r = Reco() #creating the recommender object: "r"

r$train(train_data = train)
```

```
## iter      tr_rmse          obj
##    0       0.9525   7.5676e+04
##    1       0.9186   7.1436e+04
##    2       0.8837   6.7723e+04
##    3       0.8565   6.4974e+04
##    4       0.8345   6.2835e+04
##    5       0.8167   6.1466e+04
##    6       0.7996   6.0115e+04
##    7       0.7842   5.8952e+04
##    8       0.7705   5.7955e+04
##    9       0.7588   5.7160e+04
##   10       0.7484   5.6485e+04
##   11       0.7391   5.5842e+04
##   12       0.7312   5.5329e+04
##   13       0.7242   5.4950e+04
##   14       0.7177   5.4551e+04
##   15       0.7117   5.4186e+04
##   16       0.7065   5.3868e+04
##   17       0.7018   5.3619e+04
##   18       0.6976   5.3393e+04
##   19       0.6936   5.3171e+04
```

```
#Training of LIBMF model - notice call is used from recommender object

test_baseline <- test_mini %>%
  left_join(b_u, by="userId") %>%
  mutate(rating_norm = rating-b_u)

test <- data_memory(test_baseline$userId, test_baseline$movieId)
#Test set userId and movieId data inputed into recosys environment for
#prediction.  Note that the actual ratings are not entered into object.
#The test set actual ratings will only be used for validating model prediction.

r$predict(test_data = test)
```

```
## prediction output generated at predict.txt
```

```
#prediction function for recosys based on validation data - it
#produces a file in the working directory that contains predictions. This can take a while.

pred <-read.csv("predict.txt", header = FALSE)
#reading the prediction file into the local environment.  Predictions are in CSV
#format without a header.

RMSE(test_baseline$rating_norm, pred$V1)
```

```
## [1] 0.8644954
```

Adjusting for user affect bias makes a slight improvement in overall performance. However in order to normalize the scores by user to adjust for user bias, I also had to normalize the scores in the validation set by user to get the improved result as otherwise, the model would be reporting normalized scores and which would be measured against the actual ratings. (Doing so provided an RMSE score of about 3.60.) This approach seems to fall outside the parameters of the movielens capstone project which requires on the final hold out set ratings to be used for the sole purpose of generating the final RMSE score. Therefore we will see what we can do to tune the *recosystem* model to improve the score while respecting the parameters of the project.

There are a number of different parameters to tune for the LIBMF model, including lrate, which is the learning rate, dim, which is a reference to the number of dimensions or latent factors, as well as regularization parameters for user and item effects (costp and costq), By inserting vectors into the tuning function, the *recosystem* tuning function will return a list of all the results with a loss function for each tuning permutation that was evaluated using 5-fold cross-validation by default. The best tune parameter as defined by the loss function can be accessed by using the $min on the returned list.

```
train <- data_memory(train_mini$userId, train_mini$movieId, rating = train_mini$rating)
#Inputting training data into a recosystem sparse matrix object

r = Reco() #creating the recommender object: "r"


opts <- r$tune(train_data = train, opts = list(dim = c(10,20,30),
                                               lrate = .1,
                                               costp_l1 = 0, costp_l2=.1,
                                               costq_l1=0, costq_l2=.1,
                                                ntread =1, niter = 20, nfold=5))
#the "opts" argument works like the tune grid from the caret package.

opts$min
```

```
## $dim
## [1] 10
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0.1
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.1
##
## $lrate
## [1] 0.1
##
## $loss_fun
## [1] 0.9666354
```

```
#accessing the optimized values from the tuning
```

After a number of different attempts to adjust the user and item (movie) factors due to the effects identified in the baseline model, it seems, perhaps unsurprisingly, that many of the default tuning options provided the best results. Also, while a lower learning rate than the default seemed worked well to reduce the loss function, when applied against the test set, the RMSE score was not as good as the default results, which is likely due to overfitting. Additionally, there seemed to be significant variability in the tuning results as each time I would run the model using the same parameters, I would get a different result. Therefore I set aside the tuning function and a ran a replicate and sapply function on the training and evaluation of a LIBMF function with all of the parameters optimized, except for dimensionality. For "dim" I created a short vector sequence of 10, 20, and 30 which I ran through each 30 times each in order to leverage the central limit therom to approximate the true RMSE value of the model against the test set for each value. I plotted the mean of the 30 RMSE results against the "dim" values.

Increasing the number of dimensions (latent factors) seems to provide an increasing performance of the model, however there is additional computation time added the more dimensions added. As such I decided to target the minimum score that would provide a RMSE of .8649 or lower which is full marks for this project. A "dim" value of 20 seems to be sufficient but will use 30 in order to be on the safe side.

RESULTS

Now that we have our optimized parameters, we will test the model on the edx and validation sets.

```
load("validation.rda")

set.seed(1, sample.kind="Rounding")

train <- data_memory(edx$userId, edx$movieId, rating = edx$rating)
```

```
#sparse matrix of residuals are basis for training matrix factorization model

r = Reco() #creating the Recommender System Object of the recosys package

r$train(train_data = train, opts = list(dim=30, lrate = .1, costp_l1 = 0,
                                         costp_l2=.1, costq_l1=0, costq_l2=.1))
```

```
## iter      tr_rmse          obj
##    0       0.9725   1.4994e+07
##    1       0.8812   1.3401e+07
##    2       0.8552   1.3088e+07
##    3       0.8430   1.2953e+07
##    4       0.8347   1.2881e+07
##    5       0.8286   1.2828e+07
##    6       0.8239   1.2790e+07
##    7       0.8202   1.2764e+07
##    8       0.8173   1.2747e+07
##    9       0.8146   1.2731e+07
##   10       0.8125   1.2715e+07
##   11       0.8109   1.2705e+07
##   12       0.8093   1.2697e+07
##   13       0.8079   1.2689e+07
##   14       0.8066   1.2679e+07
##   15       0.8055   1.2674e+07
##   16       0.8044   1.2668e+07
##   17       0.8034   1.2660e+07
##   18       0.8026   1.2658e+07
##   19       0.8016   1.2651e+07
```

```
#LIMBF model is trained using residual sparse matrix

valid <- data_memory(validation$userId, validation$movieId)
#validation set userId and movieId data inputed into recosys environment for
#prediction.  Note that the actual ratings are not entered into object.
#The validation set actual ratings will only be used for validating model prediction.

r$predict(test_data = valid)
```

```
## prediction output generated at predict.txt
```

```
#prediction function for recosys based on validation data - it
#produces a file in the working directory that contains predictions. This can take a while.

pred <-read.csv("predict.txt", header = FALSE)
#reading the prediction file into the local environment.  Predictions are in CSV
#format without a header.

RMSE(validation$rating, pred$V1)
```

```
## [1] 0.8185807
```

```
#.819 RMSE
```

The final model has a RMSE score of .819. The performance of the model could be improved further by increasing the number of dimensions but would create an additional computational cost that is not necessary.

Conclusion

It is clear that a matrix factorization approach is a powerful machine learning solution for recommender systems. Its performance makes it obvious why it is used in some many recommender systems and so prevalent in literature and machine learning forums. The LIBMF matrix factorization algorithm was the most powerful and least costly of the matrix factorization-based algorithms I was able to evaluate. Further tuning of the algorithm by increasing the dimensions parameter could have even further improved the algorithm's performance in predicting movie ratings of users albeit at a higher computational cost. Additionally, the the *recosystem* package which housed the LIBMF algorithm is a relatively simple package to learn and use and provides for powerful results. While it is not as feature-filled as other packages it clearly gets the job done.

While I was happy to find an algorithm that worked well and could be optimized to provide the results that were the objective of the course, I have mixed feelings about the final outcome. The amount of time I spent trying to use the matrix factorization approach, including the time it took to run many computationally intensive algorithms, the technical challenges and the learning curve involved in exploring multiple new R packages, each very different from the Tidyverse, was a great source of frustration. In hindsight, developing a simpler model, based on the approached established for the baseline model would have been a more enjoyable experience. I could have spent more time hypothesizing and trying to model different kinds of effects in the data rather than banging my head against a wall trying to make matrix factorization work on my ancient laptop.

For example, I could have explored a hypothesis I had about leveraging the information encoded in the 'timestamp' variable of the data set to identify the effects that the day of the week or time of the year has on user ratings of films. Movie theaters in North America are much busier on Fridays and Saturdays which could have an impact on a film-goers experience of a film, and ratings provided to films during "Oscar season" versus the summer 'blockbuster' season could have an effect on user experiences. While these effects would likely not have been nearly as powerful in a predictive sense, they are much more easily interpretable and enjoyable to explore than the matrix factorization approach. Matrix factorization is easily enough to understand from a conceptual perspective, however the actual results feel like they come out of a black box.

All in all it was a valuable learning experience even though it is one I would try to avoid in my next machine learning project.