MovieLens Rating Prediction

Mike Gessner

5/14/2019

Introduction and background

The objective of this study is to predict the rating of a movie by a user in the MovieLens data set, with the goal of having a root mean square error (RMSE) of less than 0.87750. RMSE is calculated as the average difference between the actual rating and predicted rating squared, and then take the square root, or in r:

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

The following steps are performed to meet the stated goal. First, we load the necessary libraries and data. Next, we explore the data structure and the data itself to look for any patterns. The next step is to perform data transformation, most of the work in this section is done for models that rely on matrix factorization. The final step is to work through some models to see which meet or exceed our stated goal. We start with some simple models and then progress there to regularization, then to recommender models, and finally matrix factorization. While this is not an exhaustive examination of potential models, most will meet the stated goal. In the end, we provide a summary of the results of the modeling.

The first step is to load the necessary libraries.

```
#knitr::opts_chunk$set()
library(tidyverse)
library(caret)
library(lubridate)
library(irlba)
library(recosystem)
library(recommenderlab)
library(ggplot2)
library(kableExtra)
```

Load the data

In this section, we load the data for the project. This is the same code as given in the edx course and is not shown below. The code reads in the data and ultimately splits the data into a training set and validation set (90% and 10%, respectively).

Exploring the data

```
head(edx)
```

```
##   userId movieId rating timestamp                         title
## 1      1     122      5 838985046              Boomerang (1992)
## 2      1     185      5 838983525               Net, The (1995)
## 4      1     292      5 838983421               Outbreak (1995)
```

```
## 5      1     316      5 838983392              Stargate (1994)
## 6      1     329      5 838983392 Star Trek: Generations (1994)
## 7      1     355      5 838984474       Flintstones, The (1994)
##                          genres
## 1                Comedy|Romance
## 2         Action|Crime|Thriller
## 4  Action|Drama|Sci-Fi|Thriller
## 5       Action|Adventure|Sci-Fi
## 6 Action|Adventure|Drama|Sci-Fi
## 7       Children|Comedy|Fantasy
```

The dataset we start with has 6 variables and just over 9,000,000 observations. Each observation is for a user who rates a movie (1-5 stars). Additional information include the timestamp of the rating by a user, the title and year of the movie, as well as genre(s) of the movie. Our variable of interest for the project is the rating variable. We want to predict the rating variable given some information (e.g., userId and movie).

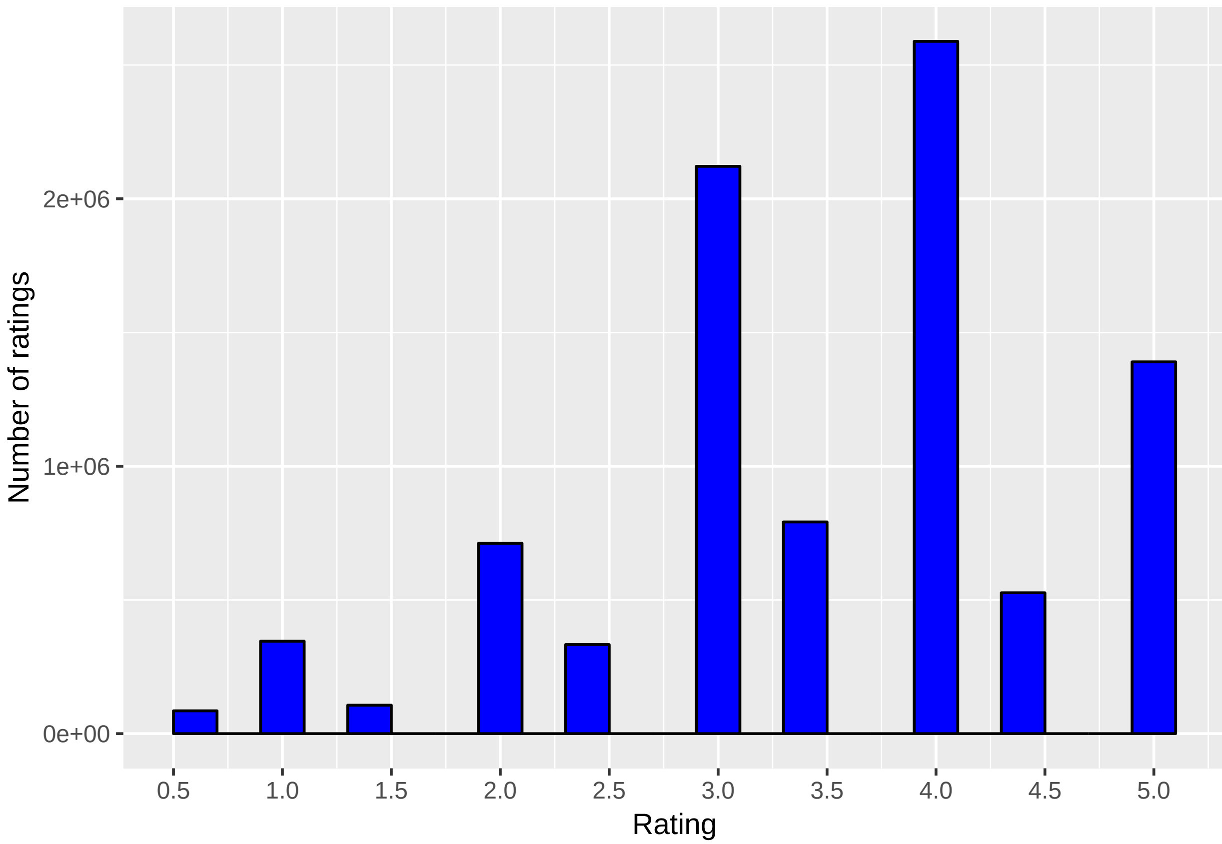
In particular, the userId variable is a unique identifier for a user–an integer value. The movieId variable is also a unique identifier, but for a movie. The timestamp variable displays the time and date a user logged a rating for a particular movie. The title of the movie is a variable that is not unique, but also contains information on the release year of the movie. The genres variable contains information about the genre of a particular movie (e.g., comedy). This variable can contain more than one genre, seperated by a '|'. Finally, the outcome variable is the rating. The value of the rating can be bvetween 0 and 5.

Now, let's take a look at the data.

```
summary(edx)
```

```
##      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         
##  Length:9000055     Length:9000055    
##  Class :character   Class :character  
##  Mode  :character   Mode  :character  
##                                       
##                                       
##                                       
```

```
edx %>% select(rating) %>% ggplot(aes(x=rating)) +
        geom_histogram(binwidth=.2, color='black', fill='blue') +
        scale_x_continuous(breaks=seq(0,5, by=.5)) +
        labs(x='Rating', y='Number of ratings')
```



We can see from the histogram of ratings in the edx dataset that there are no users that give a 0 for a rating. The most likely ratings are 4, 3, 5, and 3.5. Furthermore, users tend to rate movies as whole numbers, with half ratings (e.g., 3.5) less common.

We can also create a table of the top genres as well for the films in the dataset.

```
genres = edx %>% separate_rows(genres, sep = "\\|") %>%
        group_by(genres) %>%
        summarize(count = n()) %>%
        arrange(desc(count))
genres
```

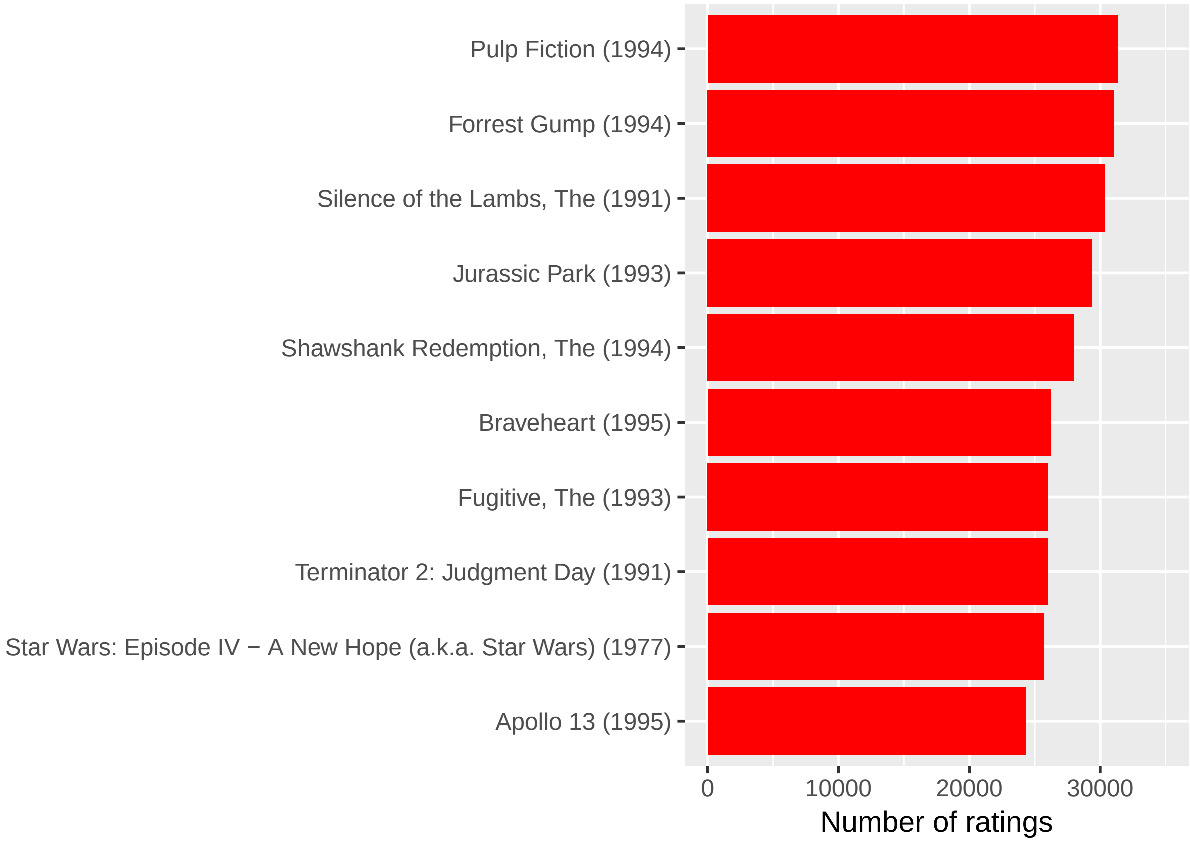
```
## # A tibble: 20 x 2
##    genres               count
##    <chr>                <int>
##  1 Drama              3910127
##  2 Comedy             3540930
##  3 Action             2560545
##  4 Thriller           2325899
##  5 Adventure          1908892
##  6 Romance            1712100
##  7 Sci-Fi             1341183
##  8 Crime              1327715
##  9 Fantasy             925637
## 10 Children            737994
## 11 Horror              691485
## 12 Mystery             568332
## 13 War                 511147
```

```
## 14 Animation           467168
## 15 Musical             433080
## 16 Western             189394
## 17 Film-Noir           118541
## 18 Documentary          93066
## 19 IMAX                  8181
## 20 (no genres listed)       7
```

The table shows the top genres. Drama, comedy, and action are the top 3.

We can also look at the most popular movies by the number of ratings.

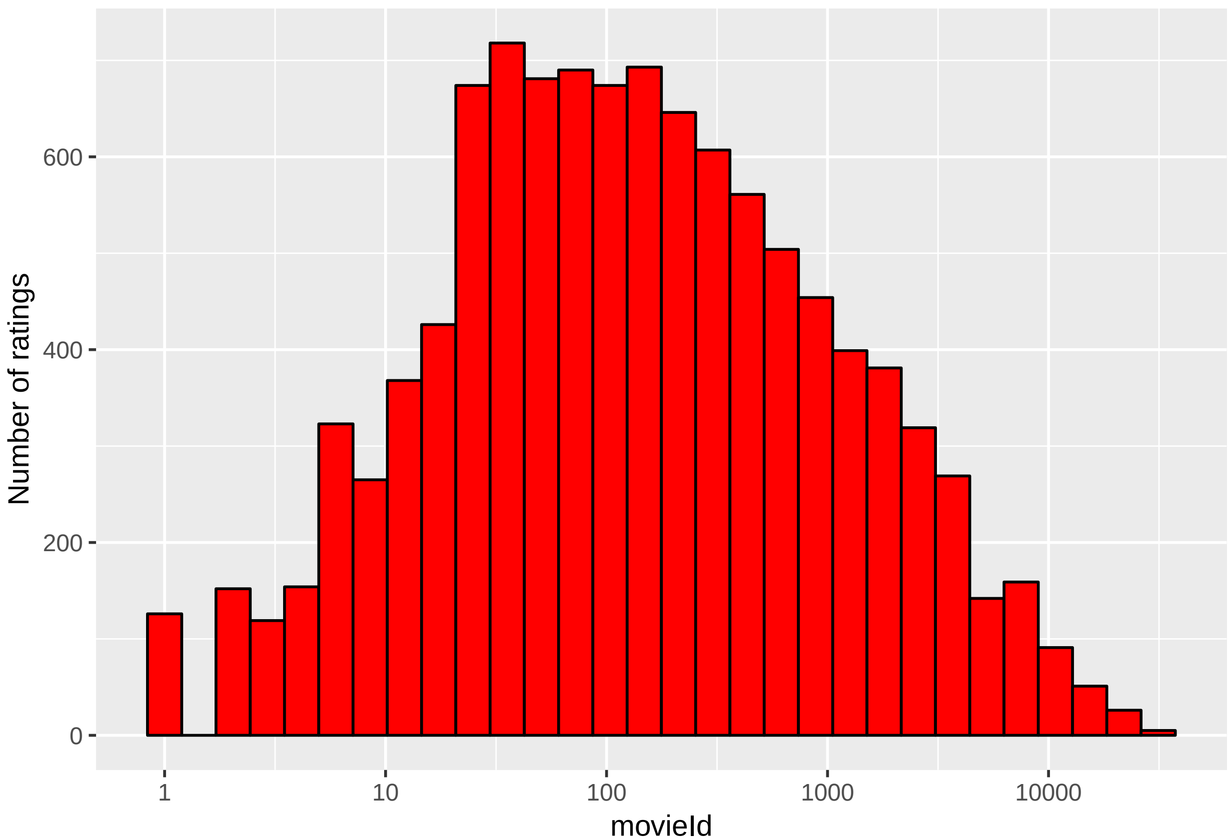
```
top_movies = edx %>% group_by(title) %>%
        summarize(count = n()) %>%
        top_n(10, count) %>%
        arrange(desc(count))

top_movies %>% ggplot(aes(x = reorder(title, count), y=count)) +
        geom_bar(stat='identity', fill='red') + coord_flip(y=c(0,35000)) +
        labs(x = "", y='Number of ratings')
```



We see that Pulp Fiction is the most rated movie, followed by Forrest Gump and Silence of the Lambs.

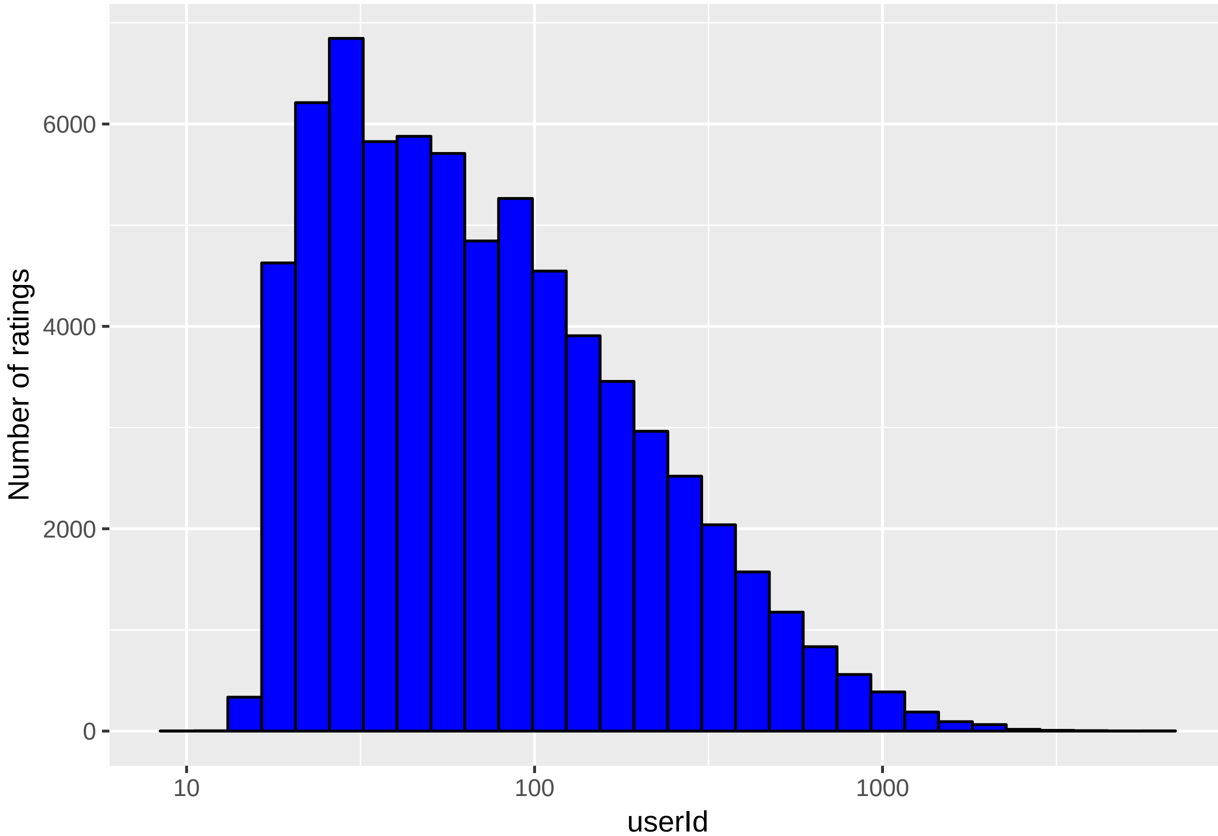
Now, let's see the distribution of the number of ratings by movie.

```
edx %>% count(movieId) %>%
        ggplot(aes(n)) +
        geom_histogram(bins=30, color='black', fill='red') +
        scale_x_log10()+
        labs(x = 'movieId', y = 'Number of ratings')
```



Let's also examine the number of ratings by userId

```
edx %>% count(userId) %>%
        ggplot(aes(n)) +
        geom_histogram(bins=30, color='black', fill='blue') +
        scale_x_log10() +
        labs(x = 'userId',
             y = 'Number of ratings')
```

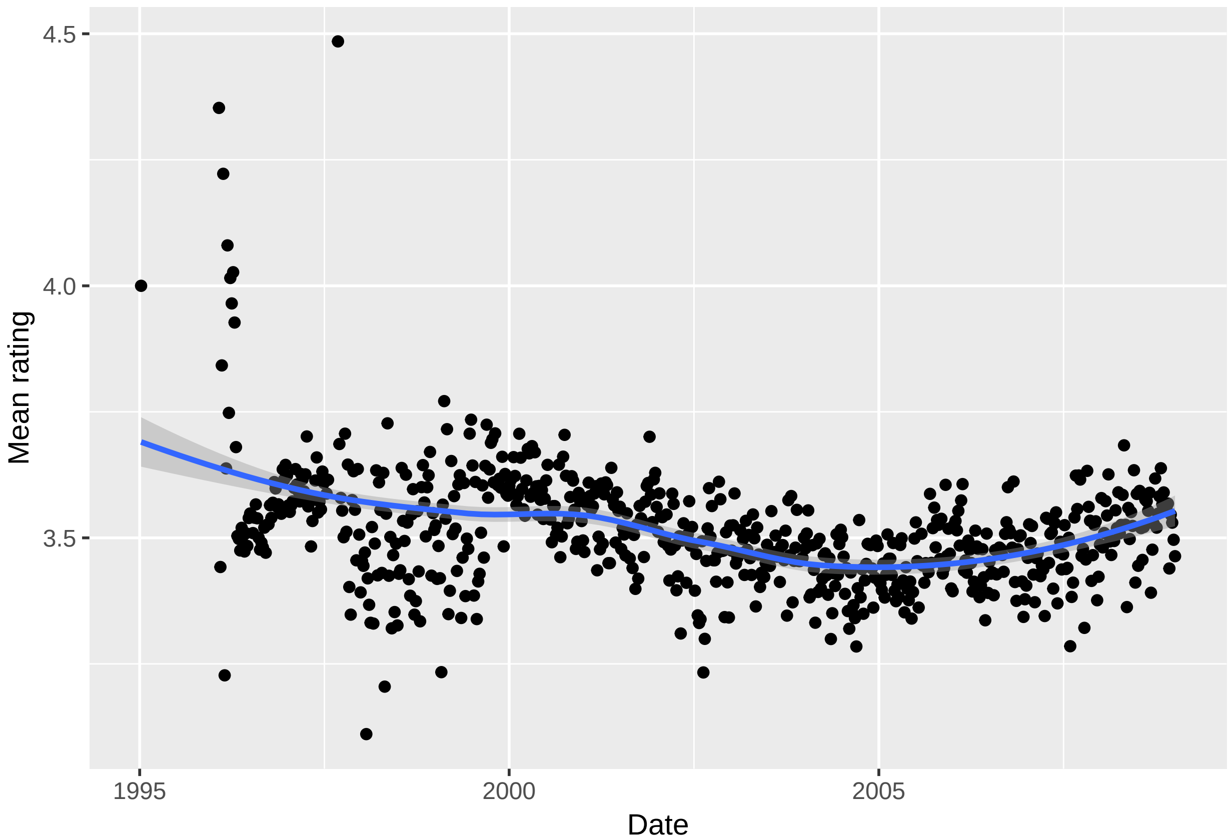


What can we tell from the two histograms? We can see that some movies get rated more than others (also from the bar plot above) and that some users are more active in rating movies than others. This indicates that there may be some movie and user effects for ratings.

We can also check for time effect. We observe the timestamp (time a user made a review) in the data and can convert that to a date and look at the average rating in the dataset over time. Converting the timestamp to a date format using as_datetime from the lubridate package is very easy using piping from the tidyr pakcage (or dplyr).

```
edx %>% mutate(date = round_date(as_datetime(timestamp), unit = 'week')) %>%
        group_by(date) %>%
        summarize(rating = mean(rating)) %>%
        ggplot(aes(date, rating)) +
        geom_point() +
        geom_smooth() +
        labs(x = 'Date', y = 'Mean rating')
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



For the most part, there does not to appear to be a strong time effect.

Matrix factorization/Data transformation

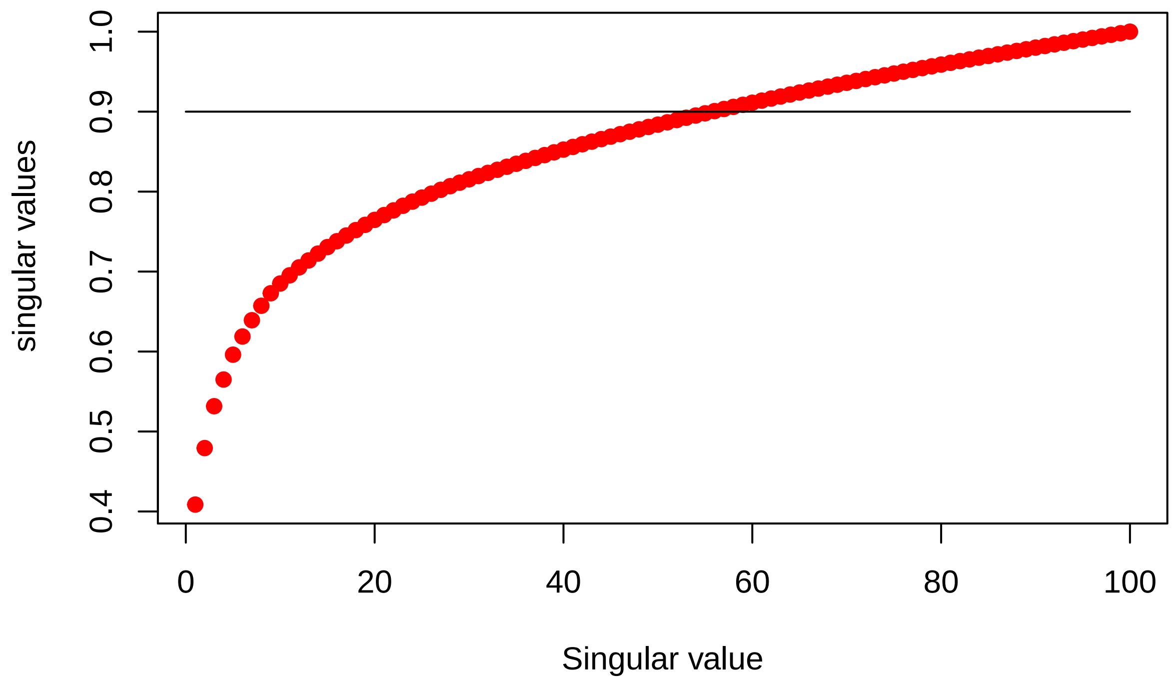
In this section, the data is transformed–mostly for use in the matrix factorization and recommender systems approaches shown later. Matrix factorizatin is similar to factor analysis and principal component analysis. We are going to make use of the fact that in the data that groups of movies have similar ratings patterns, as do user ratings.

```
#create a copy of the edx dataset, since we are going to be messing around with it
#keeping only three variables: userId, movieId, and rating
edx_copy = subset(edx, select = userId:rating)

#thought this might be useful...but probably not
edx_copy$userId = as.factor(edx_copy$userId)
edx_copy$movieId = as.factor(edx_copy$movieId)

#sparsematrix, must convert userId and movieId to numeric
edx_copy$userId = as.numeric(edx_copy$userId)
edx_copy$movieId = as.numeric(edx_copy$movieId)

#create the sparse matrix
ratings = sparseMatrix(i = edx_copy$userId,
                       j = edx_copy$movieId,
                       x = edx_copy$rating,
                       dims = c(length(unique(edx_copy$userId)),
                                length(unique(edx_copy$movieId))),
```

```r
                              dimnames = list(paste('u', 1:length(unique(edx_copy$userId)), sep=''),
                                              paste('m', 1:length(unique(edx_copy$movieId)), sep='')))

rm(edx_copy)

#now convert rating matrix into the r package 'recommenderlab' sparse matrix (real rating)
ratings_matrix = new('realRatingMatrix', data=ratings)

library(irlba)
#begin the dimension reduction through the use of a partial SVD.
#Y (matrix of N x P) can be decomposed into UDV^T where
# U = ortogonal matrix of dimension N x m
# D = diagonal matrix containing singular values of original matrix Y
# V = orthogonal matrix of dimension m x P
set.seed(23)
y = irlba(ratings, tol=1e-4, verbose=TRUE, nv=100, maxit=1000)
```

```
## Working dimension size 107

## Initializing starting vector v with samples from standard normal distribution.
## Use `set.seed` first for reproducibility.

## irlba: using fast C implementation
```

```r
#sum squares of singular values
all_sq = sum(y$d^2)

#variability by singular values
first_six = sum(y$d[1:6]^2)
#print(first_six / all_sq)

percent_vec = NULL
for (i in 1:length(y$d)) {
        percent_vec[i] = sum(y$d[1:i]^2) / all_sq
}

#plot showing k for dimensionality reduction of the matrix and sum squared singular
#values
plot(percent_vec, pch=20, col='red', cex=1.5, xlab='Singular value', ylab='% SS of
     singular values')
lines(x = c(0,100), y=c(.9,.9))
```



From the plot, we see that roughly 70% of the variability can be explained by the first 12 singular values and the first 20 singular values explain about 75%. That said, we're really after the k values that explain at least 90% of the variability. Below, we calculate that value of k. Additionally, we want to make best use of the data by using relevant users and movies that explain 90% of the variability. In the end, we create a reduced ratings matrix that meets that criteria for use in the recommender and matrix factorization models below.

```
k = length(percent_vec[percent_vec <=.9])

U_k = y$u[, 1:k]

D_k = Diagonal(x=y$d[1:k])

V_k = t(y$v)[1:k,]

#find minimum number of movies per user
min_movies = quantile(rowCounts(ratings_matrix), .9)

#find minimum number of users per movie
min_users = quantile(colCounts(ratings_matrix), .9)

#select users with those criteria for # movies, # users
ratings_movies = ratings_matrix[rowCounts(ratings_matrix) > min_movies,
                                colCounts(ratings_matrix) > min_users]
```

Modeling and results

We start with the simple approach using regression models to see if we can improve on the Root Mean Square Error (RMSE) of the simple approach.

Movie effects

The first step is to create mu, the average of all ratings in the data. This simple modeling approach uses the average rating in the data and accounts for movie specific effects (b_i). In other words, the predicted rating is a function of the average rating in the data plus a movie-specific effect (and error).

```
mu = mean(edx$rating)
```

We find mu to be 3.5124652.

Then, calculate b_i of the training set as the average of the difference between the movie's rating and the average rating in the data:

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

Next, use the above to calculate the predicted ratings

```
#predicted ratings
predicted_ratings_bi = mu + validation %>%
        left_join(movie_avgs, by ='movieId') %>%
        .$b_i
```

Movie and user effects

Now onto the movie and user effects. In this case, we also account for user effects in the model (b_u). The histogram showing the ratings and users does show differences by users. So let's account for those effects.

Like above, this model adds an adjustment for the user, so that a predicted rating is a function of the average rating in the data, movie effects, and user effects. The code is very similar to the movie effect (b_i) above.

```
#movie and user effect
user_avgs = edx %>%
        left_join(movie_avgs, by='movieId') %>%
        group_by(userId) %>%
        summarize(b_u = mean(rating - mu - b_i))

predicted_ratings_bu = validation %>%
        left_join(movie_avgs, by='movieId') %>%
        left_join(user_avgs, by='userId') %>%
        mutate(pred = mu + b_i + b_u) %>%
        .$pred
```

And finally, calculate the RMSE for each model

```
rmse_model = RMSE(validation$rating, predicted_ratings_bi)
#rmse_model

rmse_model2 = RMSE(validation$rating, predicted_ratings_bu)
#rmse_model2
```

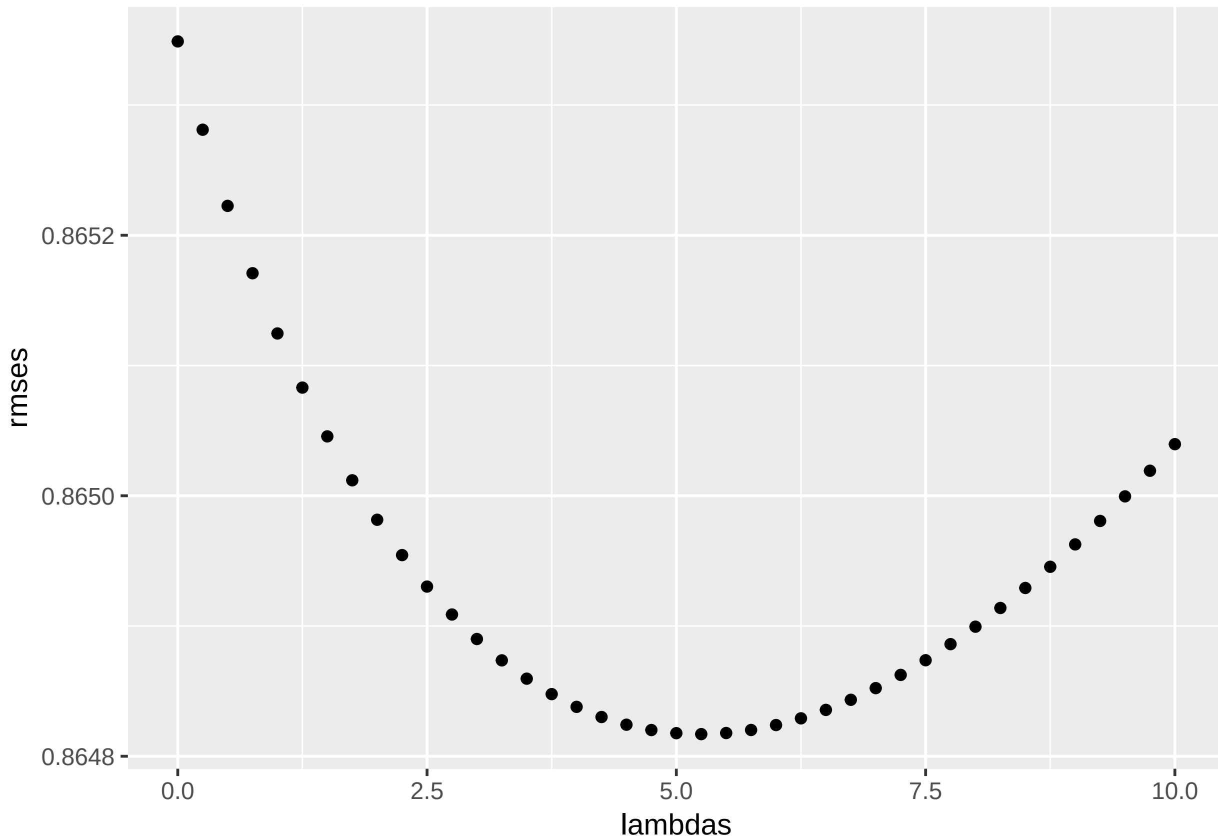
The RMSE for the movie effect (model 1) is 0.9439087. While adding in the user effects lowers the RMSE (model 2) to 0.8653488.

Regularization model

Regularization accounts for small sample sizes by penalizing large estimates that are the result of small samples. This may come in handy as there are some movies in the data that don't have a large number of ratings. Essentially, the regularization model is an extension of the above simpler models by adding a regularizing term that helps avoid overfitting a model by penalizing the magnitudes of the movie and user effects. If there are a large number of ratings, then lambda is essentially ignored in the model. However, if n is small, then the estimate of the effect (either user or movie effects) is reduced towards zero and the predicted rating is closer to the mean rating in the data.

We learned that lambda is a tuning parameter and we can choose the optimal tuning parameter viw cross-validation. That optimal value of lambda is the one that minimizes the RMSE. Let's try that now.

```
lambdas = seq(0, 10, .25) #tuning parameter

rmses = sapply(lambdas, function(l) {

        mu_reg = mean(edx$rating)

        bi_reg = edx %>%
                group_by(movieId) %>%
                summarise(bi_reg = sum(rating - mu_reg)/(n()+l))

        bu_reg = edx %>%
                left_join(bi_reg, by='movieId') %>%
                group_by(userId) %>%
                summarise(bu_reg = sum(rating - bi_reg - mu_reg)/(n()+l))

        predicted_ratings_biu = validation %>%
                left_join(bi_reg, by='movieId') %>%
                left_join(bu_reg, by='userId') %>%
                mutate(pred = mu_reg + bi_reg + bu_reg) %>%
                .$pred

        return(RMSE(validation$rating, predicted_ratings_biu))
})

qplot(lambdas, rmses)
```




```
#find optimal lambda
lambda = lambdas[which.min(rmses)]


rmse_model3 = min(rmses)
```

After choosing the optimal lambda, 5.25, we calculate the minimum RMSE. We see that the regularization model has an RMSE of 0.864817. The new RMSE is only slightly lower than that of model 2 (user and movie effects).

Recommender models

In this section, we rely on the recommenderlab package to develop two models using the popular and UBCF algorthims. A discussion of the particular methods of the recommenderlab package is found here: https://cran.r-project.org/web/packages/recommenderlab/recommenderlab.pdf

```
#try popular method
model_pop = Recommender(ratings_movies, method='POPULAR',
                        param = list(normalize='center'))

predict_pop = predict(model_pop, ratings_movies, type='ratings')

set.seed(23)
e = evaluationScheme(ratings_movies, method='split', train=.7, given=-5)
model_pop = Recommender(getData(e, 'train'), 'POPULAR')

prediction_pop = predict(model_pop, getData(e, 'known'), type='ratings')
```

```
rmse_popular = calcPredictionAccuracy(prediction_pop, getData(e, 'unknown'))[1]
```

Using a recommender models lowers the RMSE to 0.8564721. This value is slightly improved over the regularization model above.

We can also examine the user-based collobaritve filtering (UBCF) method as well.

```
#try User Based Collaborative Filtering
set.seed(23)
model = Recommender(getData(e, 'train'), method='UBCF',
                    param = list(normalize='center', method='Cosine', nn=50))

prediction = predict(model, getData(e, 'known'), type='ratings')

rmse_ubcf = calcPredictionAccuracy(prediction, getData(e, 'unknown'))[1]
```

With this method, the RMSE is 0.8662051.

Matrix factorization

Finally, we can look at matrix factorization as a method to predict ratings.

```
invisible(gc())

#create a copy of the edx data set with only the userId, movieId, and rating
edx_copy = edx %>% select(c('userId', 'movieId', 'rating'))

#renaming
names(edx_copy) = c('user', 'item', 'rating')

#make a copy of the validation data set, keeping only userId, movieId, and rating
validation_copy = validation %>%
        select(c('userId', 'movieId', 'rating'))

names(validation_copy) = c('user', 'item', 'rating')

#convert to matrices
edx_copy = as.matrix(edx_copy)
validation_copy = as.matrix(validation_copy)

write.table(edx_copy, file='~/Documents/trainset.txt', sep=' ', row.names=FALSE, col.names = FALSE)
write.table(validation_copy, file='~/Documents/validationset.txt', sep= ' ', row.names = FALSE, col.name

set.seed(23)
#make sure the data sets are in the recosystem format
train_set <- data_file( "~/Documents/trainset.txt" , package = "recosystem")
validation_set = data_file('~/Documents/validationset.txt', package='recosystem')

r = Reco()

#tune the training set
opts = r$tune(train_set, opts=list(dim=c(10,20, 30), lrate=c(.1,.2),
                                   costp_l1 = 0, costql1=0,
                                   nthread = 1, niter =10))
```

```r
#now we train the recommender model
r$train(train_set, opts=c(opts$min, nthread = 4, niter=100, verbose=FALSE))

#create the prediction file
pred_file = tempfile()

#predict the ratings
r$predict(validation_set, out_file(pred_file))
```

```
## prediction output generated at /var/folders/cg/_n_fmpds0mg80ngyh0zp87yw0000gp/T//RtmpFov2ju/file89c0
```

```r
#print(scan(pred_file, n=10))

scores_real = read.table('~/Documents/validationset.txt', header=FALSE, sep = " ")$V3
scores_pred = scan(pred_file)

rm(edx_copy, validation_copy)

rmse_mf_opt = RMSE(scores_real, scores_pred)
```

Running the matrix factorization model, shows that the RMSE for this method is 0.7821759.

Summary

```r
rmse_summary = data.frame(Model = c('Movie effects', 'Movie & User effects',
                                    'Regularized', 'Recommender popular',
                                    'Recommender UBCF', 'Matrix factorization'),
                          RMSE = c(rmse_model, rmse_model2, rmse_model3,
                                   rmse_popular, rmse_ubcf, rmse_mf_opt))

kable(rmse_summary) %>%
        kable_styling(bootstrap_options = "striped" , full_width = F , position = "center") %>%
        kable_styling(bootstrap_options = 'bordered', full_width=F, position='center') %>%
        column_spec(1, bold=TRUE) %>%
        column_spec(2, bold=TRUE)
```

Model	RMSE
Movie effects	**0.9439087**
Movie & User effects	**0.8653488**
Regularized	**0.8648170**
Recommender popular	**0.8564721**
Recommender UBCF	**0.8662051**
Matrix factorization	**0.7821759**

The first model, considering only movie effects, has a RMSE of 0.9439087. This doesn't meet our stated goal of an RMSE below 0.8775. If we add in user effects to the basic model, our RMSE is lowered to 0.8653488, which is below our goal. We could have stopped there, but there is opportunity to improve the RMSE. For example, the regularized model improves slightly to 0.864817. Moving to the recommender models, the popular method also improves the RMSE slightly, but the UBCF method does not improve the RMSE over regularization. Finally, if we use the matrix factorization method, our RMSE improves greatly to 0.7821759.