MovieLens Recommender Project

Collin Leonard

6/20/2020

Overview

The data set used for this machine learning recommender project is the MovieLens set. According to GroupLens:

> This data set contains 10000054 ratings and 95580 tags applied to 10681 movies by 71567 users of the online movie recommender service MovieLens.
> Users were selected at random for inclusion. All users selected had rated at least 20 movies. Unlike previous MovieLens data sets, no demographic information is included. Each user is represented by an id, and no other information is provided.

The set can be downloaded here.

Since the Netflix Prize was awarded on September 21, 2009, many teams have worked on expanding the methods of modern recommendation systems. This effort has broad applications in many user based streaming services such as Netflix, Hulu, Spotify, Pandora, and beyond.
These methods in their simplest forms provide a great, well documented launchpad for beginning students of machine learning to develop and compare their algorithms to a wide swath of skilled data scientists and amateurs. Additionally, there are numerous packages developed for R that make more advanced statistical techniques more accessible to those lacking the statistical/programming background, but desire to explore the theories at a big picture level.

This project is primarily designed to introduce students to two very basic methods of developing rating predictions based on a limited number of factors. In the 10M MovieLens set, the included tags are distilled to

```
## [1] "userId"    "movieId"   "rating"    "timestamp" "title"     "genres"
```

These allow a student to explore the relationships between these factors without being overwhelmed with other considerations requiring more complex methods.

The goal of this project is to minimize the root squared mean error between predicted and actual movie ratings. The general approach requires breaking the data up randomly into a test set (`validation`) and a training set (`edx`). Two methods compared in this paper are single value decomposition and a bias training with regularization. They are each trained by breaking the `edx` data into its own training and test sets, and after optimization, predicting ratings for the `validation` set.

Methods/Analysis

section that explains the process and techniques used, including data cleaning, data exploration and visualization

To begin, the "test-validation.R", provided by the course instructors and slightly edited, generated a reproducible `edx` and `validation` set from the MovieLens data. The code also guarantees the users and movies in `validation` are contained within a user-item matrix of `edx` data. This data was saved in the data folder in the working directory.

The timestamp column is hard to work with, so the time data was stratified by week using the code:

```
edx <- edx %>%
       mutate(date = as_datetime(timestamp)) %>%
       mutate(date = round_date(date, unit = "week"))
```

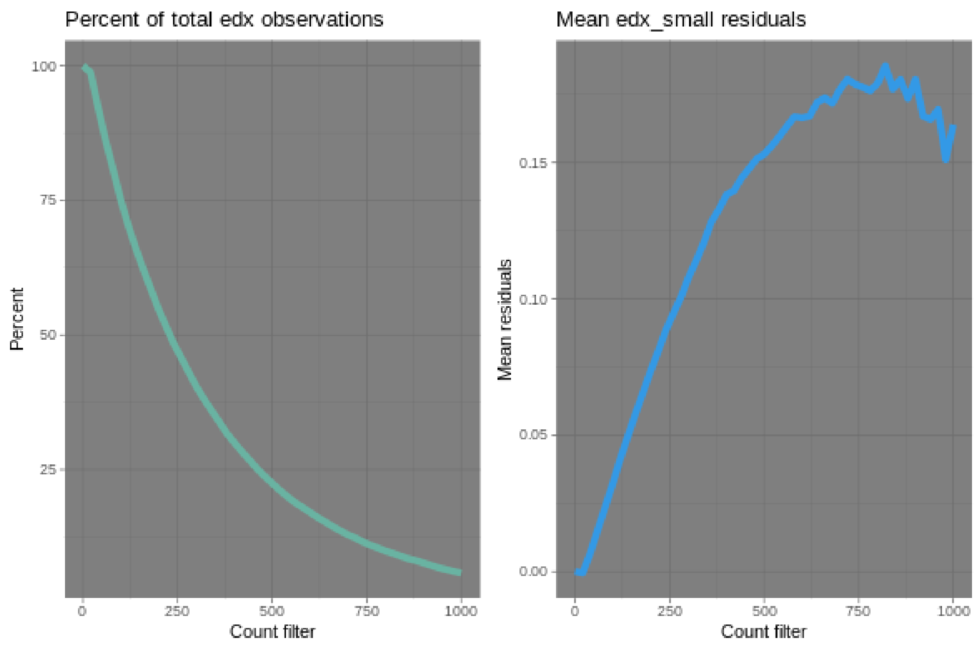
The validation time data was also manipulated in a similar manner.

SVD Method

The goal of the SVD Method is to calculate a series of matrix factorizations capable of explain the variability in the data in order to make predictions. The equation used to make the final prediction, using all of the SVD is:

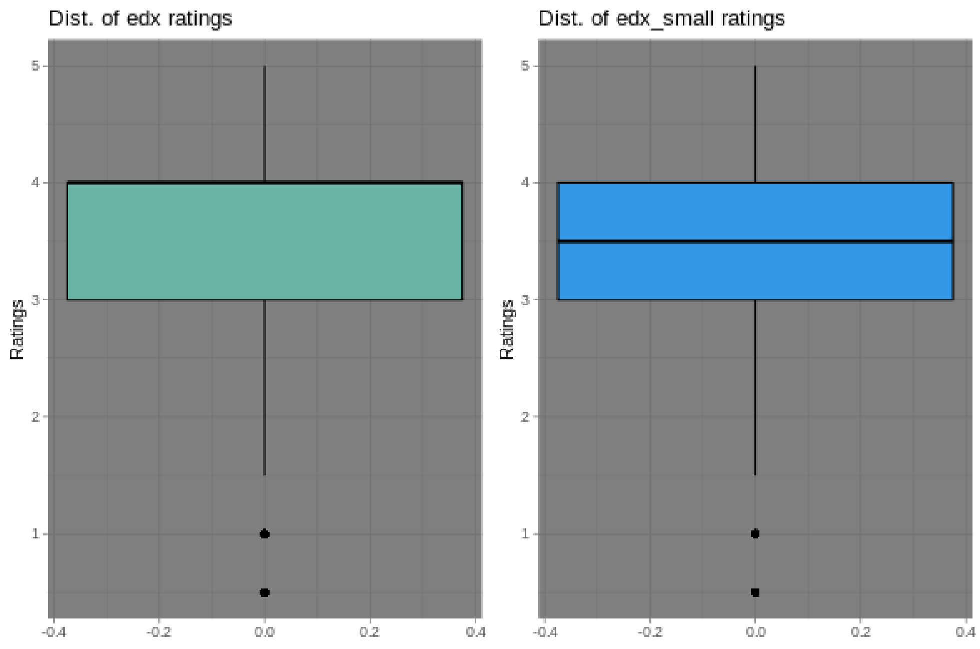
$$Y = d_{1,1}U_1V_1^T + d_{2,2}U_2V_2^T + \cdots + d_{p,p}U_pV_p^T$$

Before performing any major operations on the raw data, it is important to understand it. Because the computer used in this project is not robust enough to perform calculations on the full 900055 observations of 8 variables in `edx`, there must be a decision to filter out movies and users below a certain threshold of ratings. To do this, we must try to minimize the change in the original rating distribution, and understand what the most efficient cutoff should be. First, we generate counts of each movie and user, then use the size_change function to filter value "n" and return the percent of observations of edx after filtering. The mean_comp function is then used with the same filtering criteria as size_change, to return the residual difference between the mean rating of edx and the mean rating of the new filtered data set. The following plots allow us to understand relationships between the count cutoff, the resulting distribution, and the percent observations remaining.



`edx_small` is the filtered result of `edx`. This figure shows us that an ideal count filter is on the downslope of the first figure, and the upward slope of the second. The goal is a balance of retaining the original distribution (quantified by the distance of the set from the mean) while cutting down on the large data size. n > 250 was the chosen filter because it satisfied these conditions reasonably well.

The new `edx_small` data set can be compared to the original `edx` data to determine if the two distributions are similar enough to proceed with manipulation. The figure below shows the comparison:



While the median of the `edx` ratings appears to be 4, the `edx_small` median rating is 3.5. Even though this difference is observed, the general shape of the distribution is acceptable to proceed cautiously with further calculations.

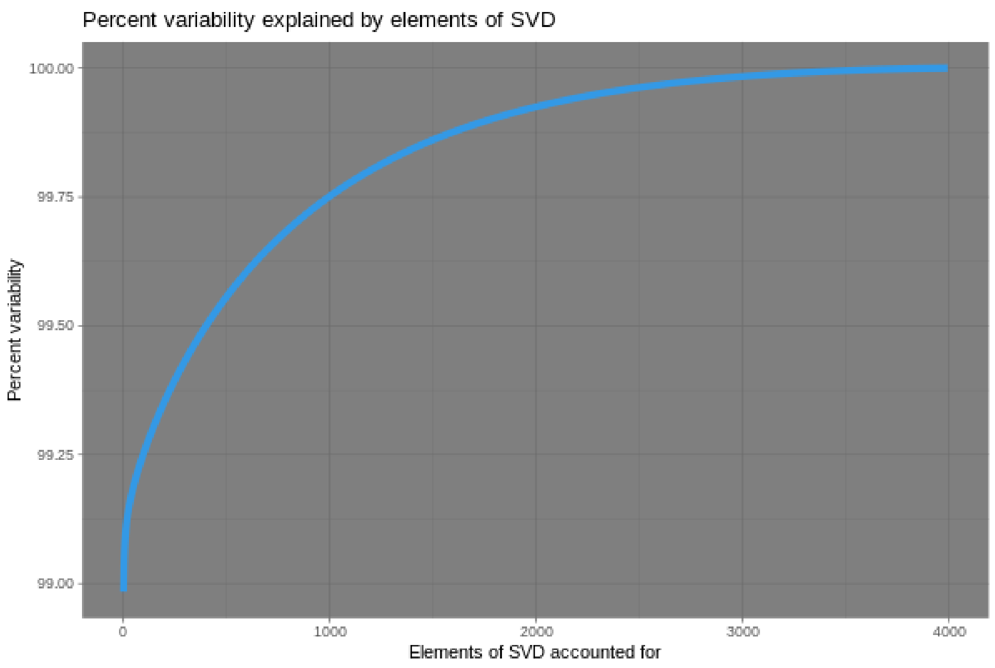
Using the **spread** function from the *tdyr* package, the new data is used to generate a user-item matrix, `y`:

```
y <- edx_small %>%
  select(userId, movieId, rating) %>%
  spread(movieId, rating) %>%
  as.matrix()
```

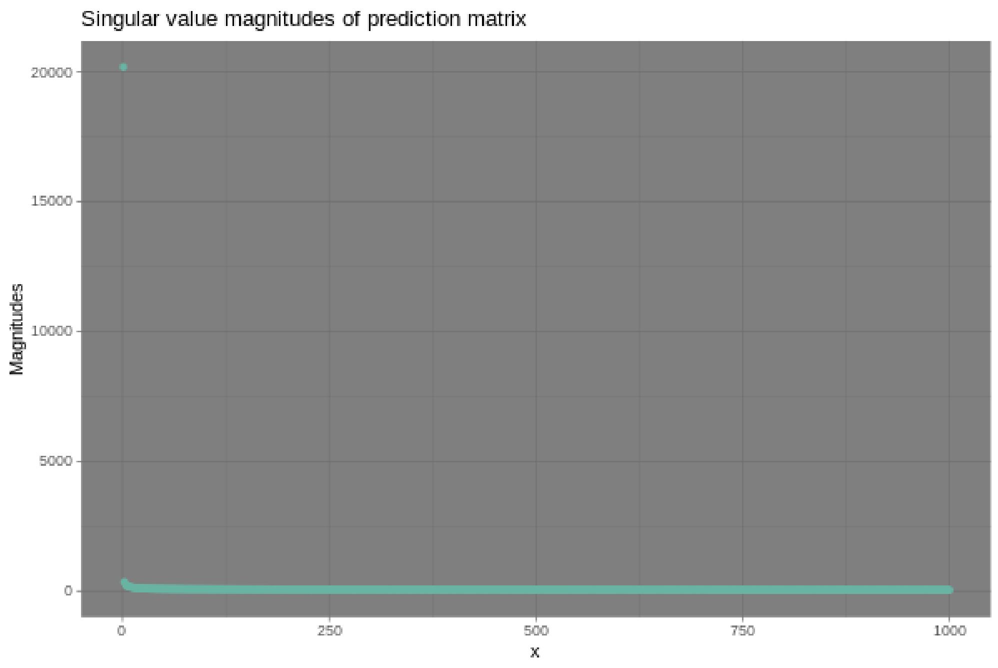
When generating a sparse matrix like this one, many of the entries are *NAs*, and something must be done with them. Two options are to replace the *NAs* with *0's* or replace them with estimates of the ratings. The rating estimates are obviously closer than the true values than *0*. A copy of the matrix `y` was generated named `ty`, and filled in with the average of the user-item combination. The for loop code that does this was based on code from the excellent report of Taras Hnot, called *Recommender Systems Comparison*. Taking the matrix that combined real ratings with estimates, we can implement the singular value decomposition of the matrix:

```
s <- svd(ty)
```

To find the k value to use in the final recommendation matrix, it is important to visualize the effects each factor of the decomposition has on the total variability explained. The R code `sum(s$d[1:i]^2)/sum(s$d^2) * 100` was used to calculate the percent variability of factors `1:i`. The plot of this result follows:



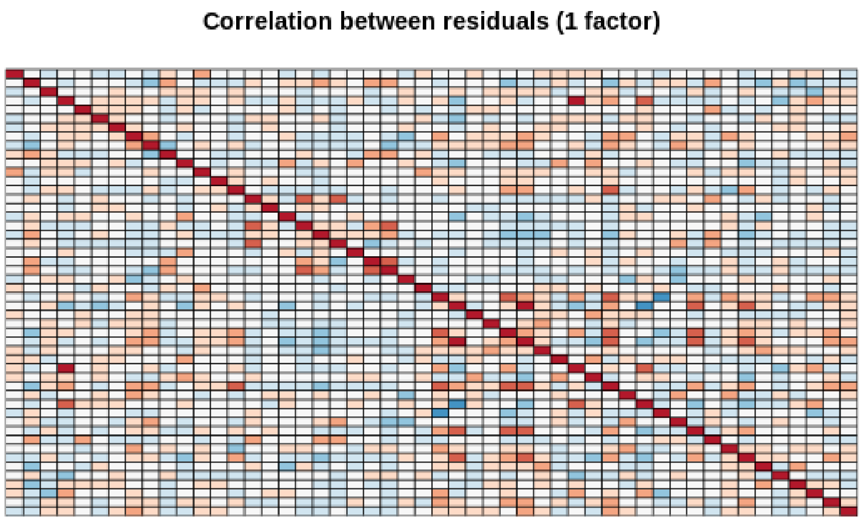
As shown, the percent variability starts very high, with the first factor explaining 99% of the variability. Plotting the magnitudes of each factor in the decomposition shows us similar information:



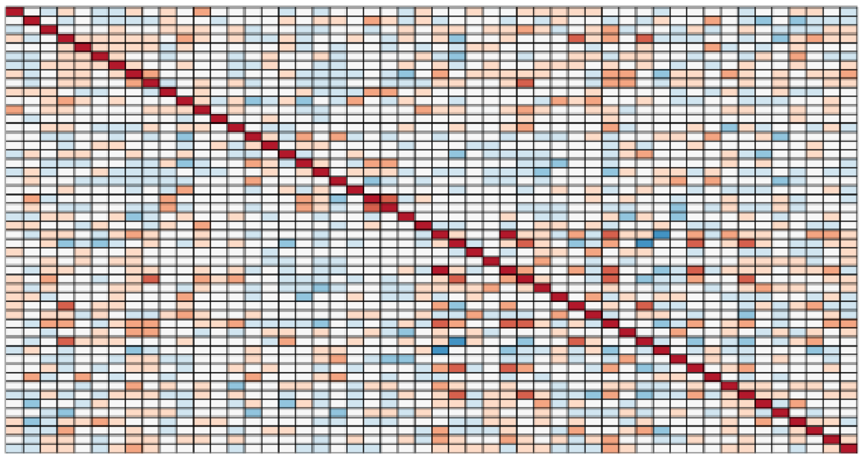
The first term's magnitude dwarfs the rest of the terms in the prediction matrix. To be sure that this is the case it is helpful to plot a matrix of correlations between the first chunk of the matrix. The sample code below shows the method used to plot the first factor, using the instructors function *my_image*.

```r
resid  <- ty - with(s,sweep(u[,1], 2, d[1], FUN="*") %*% t(v[,1]))
my_image(cor(resid[1:50,1:50]),  "Correlation between residuals (1 factor)")
```

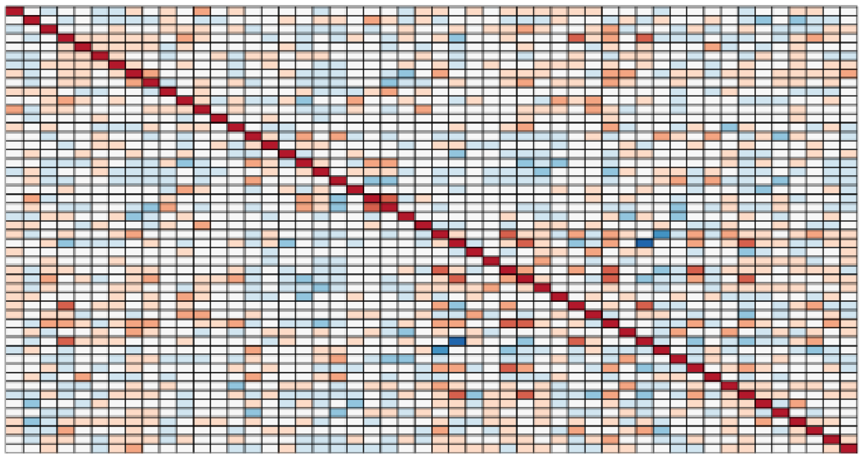
The results of the first three term plotted are:



Correlation between residuals (1:2 factors)



Correlation between residuals (1:3 factors)



We see that in the first *50 x 50* matrix of users and movies, the correlation in the images changes only slightly when compared to the first term. This is a good indication, along with the other evidence, that the first factorization is suitable in itself for generating a prediction matrix, with ϵ as random variation. The first matrix factorization will be used for the prediction matrix.

$$Y \approx d_{1,1} U_1 V_1^T + \epsilon$$

```r
k = 1

s_k <- Diagonal(x = s$d[1:k])

U_k <- s$u[, 1:k]
dim(U_k)

V_k <- t(s$v)[1:k, ]
dim(V_k)

predicted <- U_k %*% s_k %*% V_k## Results
```

The new prediction matrix is then used to generate a two column data frame of user-movie combinations, and added to `validation` data set. While there are many *NAs* left in the prediction set, the RMSE of the existing predictions was calculated to gauge how well the method performed.

```r
svd_pred <- pivot_longer(pred_df, -userId, names_to = "title", values_to = "svd_pred")
svd_pred$userId <- as.integer(svd_pred$userId)

edx <- readRDS("./data/edx.rds")
validation <- readRDS("./data/validation.rds")
edx <- left_join(edx,svd_pred,by = c("userId","title"))
validation <- left_join(validation,svd_pred,by = c("userId","title"))
```

Bias training Method

The bias training method stratifies the data based on the given tags, and calculates the patterns from individual biases. The equation used before regularization is:

$$Y = \mu + b_i + b_u + f(d_{u,i}) + \sum_{k+1}^{K} x_{u,i}\beta_i + \xi_{u,i}$$

with f a smooth function of $d_{u,i}$, $x^k_{u,i} = 1$ if $g_{u,i}$ is genre k

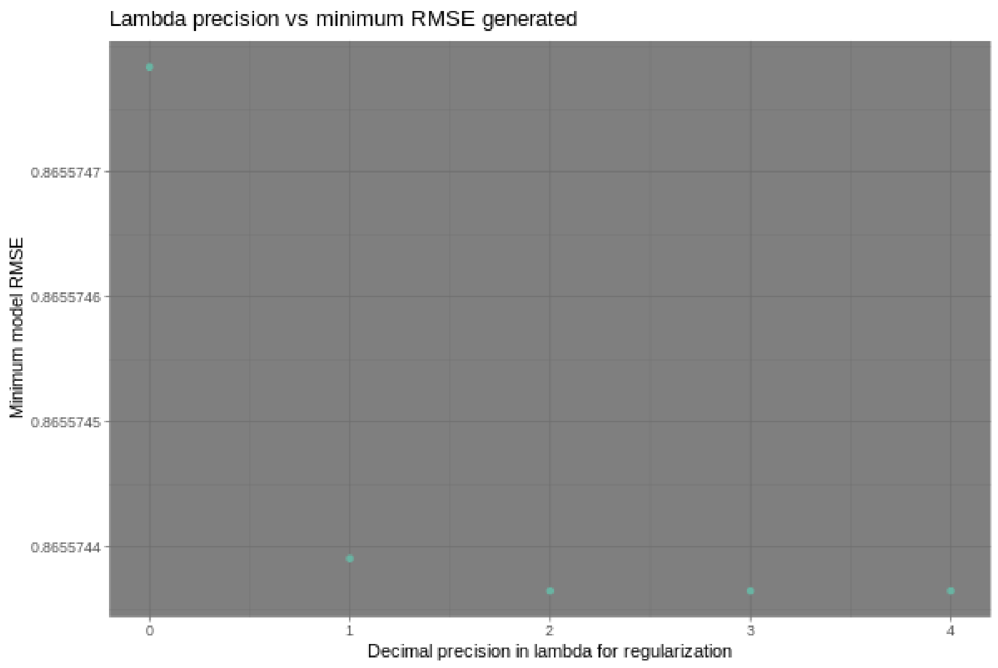
The bias training method was a more straight forward approach to generating predictions. Its starts by breaking `edx` data into a `train_set` and a `test_set`.

```r
set.seed(719)
test_index <- createDataPartition(edx$rating, times = 1, p = 0.2, list = FALSE)
test_set <- edx[test_index,]
train_set <- edx[-test_index,]

test_set <- test_set %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")
```

The following bias model groups the data by movie, user, genre, and week, and uses regularization and the simple mean to predict a value.

```r
# mu is naive mean
mu <- mean(train_set$rating)

# movie bias
b_i <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n()+l))

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

# genre bias
b_g <- train_set %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  group_by(genres) %>%
  summarize(b_g = sum(rating - b_u - b_i - mu)/(n()+l))

# time bias
b_t <- train_set %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  left_join(b_g, by="genres") %>%
  group_by(date) %>%
  summarize(b_t = sum(rating - b_g - b_u - b_i - mu)/(n()+l))

# join baises with test set to generate new predictions
predicted_ratings <- test_set %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  left_join(b_g, by = "genres") %>%
  left_join(b_t, by = "date") %>%
  mutate(pred = mu + b_i + b_u + b_g + b_t) %>%
  select(pred)

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

Using this general model, a for loop can be used to optimize the lambda value to the test set. With the loop. a plot of the lambdas effect on the RMSE based on the number of decimal points shows the how much precision is warranted:



The ideal lambda found by the `for loop` with a seed of 719 when using `createDataPartition()` is:

$$\lambda = 5.16$$

The same model was then applied to the `validation` set, using the whole of `edx` as a training tool.

Results

The result from each of the three methods outline in the previous section are as follows:

Model	RMSES
Naive Mean Model	1.061202
SVD Model	1.032113
Biases Model	0.864305

Clearly, the biases model has the lowest RMSE. The cross validation of λ resulted in a very successful algorithm to predict the ratings in the validation set. It was surprising how high the SVD method's RMSE was, being almost as high as guessing the global mean rating everytime. There are many possible explanations for this. One could be, the first matrix factorization was not enough to explain a sufficient amount of variation in the `validation` set. Another possibility is a failure to properly convert the prediction matrix into a form that is useful for predictive analysis. Instead of converting the prediction matrix into long form and using *left join* on the validation set, it could have been more advantageous to cycle through the validation set to generate predictions. One of the most challenging aspects of this project was managing the working memory of the R session. This limited the predictive process heavily. I was forced to cut the training set down by half to allow to computer the memory to work with the matricies.
The *recommenderlab* package is a great package to use in a project like this, but when attempting to convert the base matrix into *realRatingMatrix* format, the session would be aborted. Problems like this prevented me from really using the prediction matrix generated in a meaningful way, and are likely the reason the SVD method did not work well for me. The biases model presented a simple opportunity to use averages

across each tag, with a simple regularization constant and cross validation, to take more effects into account than the movie-user matrix made in the SVD method. The shortcomings of the biases model are the lack of ability to detect more subtle correlative influences, like the relationship between genres and the effects of time on ratings.

Conclusion

In this project, a great deal was learned about the relationships between variables in a relatively simple rating data set, and was eye opening in terms of how complicated recommenders at Spotify or Netflix must be. It also made obvious the need for computing power when working with large datasets.
Ironically the only model well suited to extrapolate ratings was the naive mean model. An SVD model, or biases model are unable to operate well outside a validation set that contains new users or movies. The biases model might perform slightly better because it can receive genre information, but these methods are really designed to be interpolative, and therefore are not incredibly practical in the real world.
The biases model has a lot of potential for better bias training. One example, could be to include the fact that ratings are discrete on a 0.5 star increment, and people are more likely to rate with a rounded whole star than half a star. This could allow a bias to be trained that rounds in a non-traditional way. Another bias could be included that takes into account the change in perception about a movie as the ratings become farther away from a release date. Someone who rates a movie years after the release date might be inclined to rate it better for nostalgic reasons, or the simple fact someone is watching it years later means the movie has aged well (in a general case). Unfortunately, it would be difficult to include many biases because the process of calculation gets much more intensive the more biases one collects. An approach that uses corellations is probably more robust, and one doesn't need to understand why certain movies or users are linked to use the data to improve the recommendations.

References

1. https://rstudio-pubs-static.s3.amazonaws.com/100749_6b0f55153e71461fa382fd2a2db66507.html, R markdown tips
2. https://rstudio.github.io/distill/tables.html, help with tables
3. https://rstudio-pubs-static.s3.amazonaws.com/100749_6b0f55153e71461fa382fd2a2db66507.html, R markdown notation
4. https://www.netflixprize.com/index.html, netflix prize information
5. https://www.r-graph-gallery.com/line-chart-dual-Y-axis-ggplot2.html, data visualization help
6. https://rstudio-pubs-static.s3.amazonaws.com/287285_226b14b0d57345cd84219bad6e60c0cd.html, Matrix Factorization & SVD
7. https://rpubs.com/tarashnot/recommender_comparison, recommender project example
8. https://grouplens.org/datasets/movielens, MovieLens data information
9. https://data-flair.training/blogs/data-science-r-movie-recommendation, recommender tutorial