MovieLens Project Report

Bjoernar Tuftin

October 25, 2019

Introduction

This MovieLens project was a part of the capstone course for the Data Science Professional Certificate Program from HarvardX. The goal was to use a subset of the MovieLens set, a public dataset of movie ratings, to create a model to predict future ratings. A set of 9 million ratings, `edx`, was used for training, and a set of 1 million, `validation`, used for validating the final model.

The measure of success was to be the root mean square error (RMSE) of the predictions on the validation set, with an RMSE above .9 a poor result and below 0.8649 an excellent one.

Because of the size of the data I started by creating a much smaller dataset for experimentation by selecting at random 30% of users and then 30% of the movies. I confirmed that this data was similar to the original dataset and then applied a basic model using the overall mean and regularized movie and user effects getting a decent result.

When trying to improve on this, I eventually settled on adapting a very good solution to the same problem available in the `recommenderlab` package, which uses matrix decomposition by a stochastic gradient descent optimization, implemented in the function `funkSVD()`, to find patterns in how some movies are rated similarly, and how some users have similar preferences.

I looked at the available predictions in our set to see if there were any obvious alternatives that could do a similar job, for instance using the genre data, but when I couldn't find any, I turned to working on decreasing the time and memory use of the `funkSVD()`-approach and also allow tuning of at least some of the parameters with the resources available.

My solution was to write a version of `funkSVD()` which allows you to run it on parts of the data, predicting patterns for movies based on ratings for all movies, but a subset of users, and patterns for users based on ratings for all users but a subset of movies. This still produces predictions of movie-type/user-preference interactions for all users and movies, but with a smaller memory footprint and, potentially, depending on how much data is left out, shorter run time.

After tuning the portion of the data used on a small test set this performed better than the original, but I do not know if this is true for the full set as my hardware cannot cope with running the original on the full `edx` set. I suspect it is though, as the RMSE of 0.8116 was well below the goal.

Methods and analysis

Creating an exploration set

My first step was to examine the data and look at how ratings were distributed for the set as a whole and across movies and users, to get ideas on how to proceed and to check any assumption based on previous experience with a smaller subset of the data in a previous course in the program.

Because 9 million ratings is a lot of data to work with, and because I needed to split the training set into further training and validation sets anyway during experimentation with different models, my first step was to select a random subset of ratings for a `train_set` and `test_set`.

This approach led to a lot of ratings in the test set that were for movies and/or users not in the train set, which is not desirable. I next sampled the desired proportion of users randomly, and picked the ratings for all of those users, which worked better, but the dimensions of the dataset changed dramatically, with 10 % of the users, but almost 90 % of the movies, so instead I sampled 30 % of users and 30 % of movies reviewed by those, for approximately 10 % of the total ratings and then divided those ratings into `train_set` and `test_set` with 90 % in the first.

After moving any ratings in the test set that didn't have any representation in the train set, I was left with the distribution of ratings shown in table 1.

Table 1: Distribution of ratings in the exploration sets

set	# ratings
train_set	736142
test_set	147208

As a superficial check that we have a representative selection I had a look at the proportions of various ratings in the `edx` and `train_set` data. As figure 1 shows, they are almost identical.

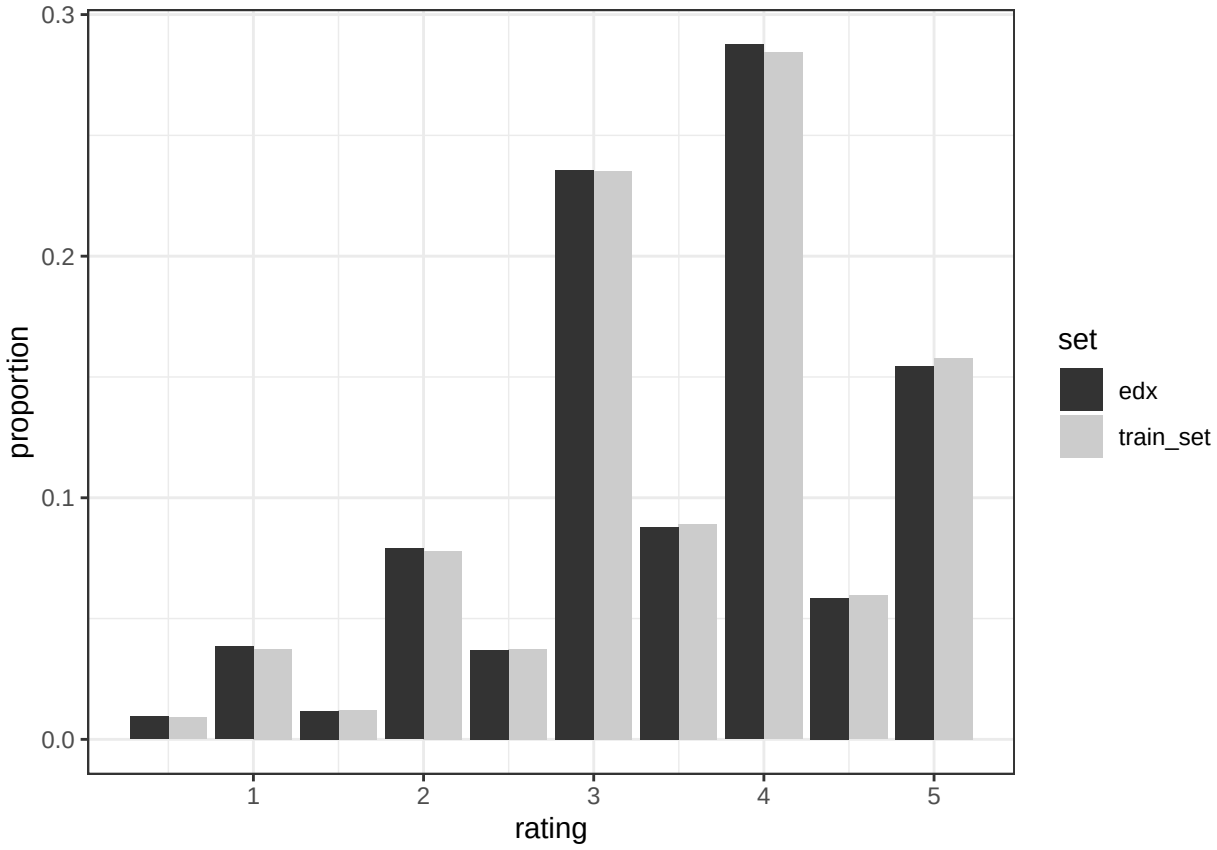

Figure 1: Proportion of ratings

Movie and User effects

From working with a similar subset of this data in the course I knew that a good initial approach would be to use a model where the rating $Y_{i,j}$ for movie i and user j is the sum of the mean rating overall, μ, a movie effect, b_i, based on the average residuals for each movie when the overall mean was removed, and a user effect, b_u based on the same for each user after removing the previous two means.

$$Y_{i,j} = \mu + b_i + b_u$$

This model is easy to calculate, so I ran it on both my original subset and on a 90/10 train/test sample of the full `edx` training set for comparison, giving us the RMSEs in table 2. The code below only shows the calculation for the `train_set` to avoid unnecessary repetition.

```
# Calculating the means of our train_set
train_mean <- mean(train_set$rating)

# We calculate the movie effect per movie
movie_effect <- train_set %>% group_by(movieId) %>%
  summarize(b_i = mean(rating - train_mean))

# We calculate the user effect per user
user_effect <- train_set %>% left_join(movie_effect, by='movieId') %>%
  group_by(userId) %>% summarize(b_u = mean(rating - b_i - train_mean))

# We create predictions for the test_set
predicted_ratings <- test_set %>%
     left_join(movie_effect, by='movieId') %>%
     left_join(user_effect, by='userId') %>%
     mutate(pred = train_mean + b_i + b_u) %>%
     .$pred

# We calculate the RMSE and store it for comparison
model_m_u_rmse <- RMSE(predicted_ratings, test_set$rating)
rmse_results <- tibble(set = "test_set", method="Movie+User Effect Model",
                                    RMSE = model_m_u_rmse)
```

Table 2: RMSE comparison

set	method	RMSE
test_set	Movie+User Effect Model	0.8764
large_test_set	Movie+User Effect Model	0.8647

I could see that this basic model performs quite well and a little better on the larger set than on our smaller sample. In fact, it performed well enough that it would meet our RMSE goal all by itself, if I could assume it would do as well with the full `edx` and `validation` sets, which of course I could not as it will vary with the sample.

To improve it further I tried three different approaches, which I'll cover in the next sections.

Regularization

The model I used treats all users and all movies the same, but statistical theory tells us that users or movies with few ratings should be poorer predictors of future ratings than those with many ratings. I attempted to confirm that this is true for this particular data by looking at the biggest misses.

Table 3: Largest errors

short_title	rating	miss	b_i
Distant Voices, Stil	4.5	4.000000	-3.0213838
Last Mistress, The (	0.5	-4.000000	0.9786162
Godfather, The (1972	0.5	-3.902719	0.8813352
Godfather, The (1972	0.5	-3.902719	0.8813352
Godfather, The (1972	0.5	-3.902719	0.8813352
Godfather, The (1972	0.5	-3.902719	0.8813352
Godfather, The (1972	0.5	-3.902719	0.8813352

Based on table three it did not look like the worst mistakes are for the least rated movies, but rather for the movies with the highest average rating, which shouldn't have surprised me. My mistake of course was to look at the movie effect for individual ratings, instead of the average for each movie, so I did the exercise again.

Table 4: Largest movie average error

short_title	avg_miss	no_ratings	b_i
Last Mistress, The (	-4.00	1	0.9786162
Distant Voices, Stil	3.75	2	-3.0213838
Charm's Incidents (C	-3.50	1	0.9786162
Blind Chance (Przypa	-3.45	1	0.4286162
My Sister Eileen (19	-3.25	1	0.2286162

Table 4 was more in line with the hypothesis. The largest average errors was for movies with a small number of ratings, and a lot of them have quite large movie effects. Note that the column for number of ratings refer to the number of ratings in `test_set`, but that does indicate it is a movie with only a few ratings in `train_set`.

Large effects predicted for some of the groups with few observations is a common statistical problem and is compensated for by regularization. To regularize the result I introduced a penalty for movies with few ratings. Instead of setting b_i to be the average for the movie I set it to $\frac{1}{n+\lambda}\sum ratings$. For large n this approaches the mean, but for smaller values of n it shrinks the effect.

I did this for both the movie effects and user effects and tried a range of lambda values to see which improved the results the most.

```
lambdas <- seq(0, 10, 1)

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

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

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

  predicted_ratings <-
    test_set %>%
    left_join(b_i, by = "movieId") %>%
```

```
    left_join(b_u, by = "userId") %>%
    mutate(pred = train_mean + b_i + b_u) %>%
    pull(pred)

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

qplot(lambdas, rmses)
```

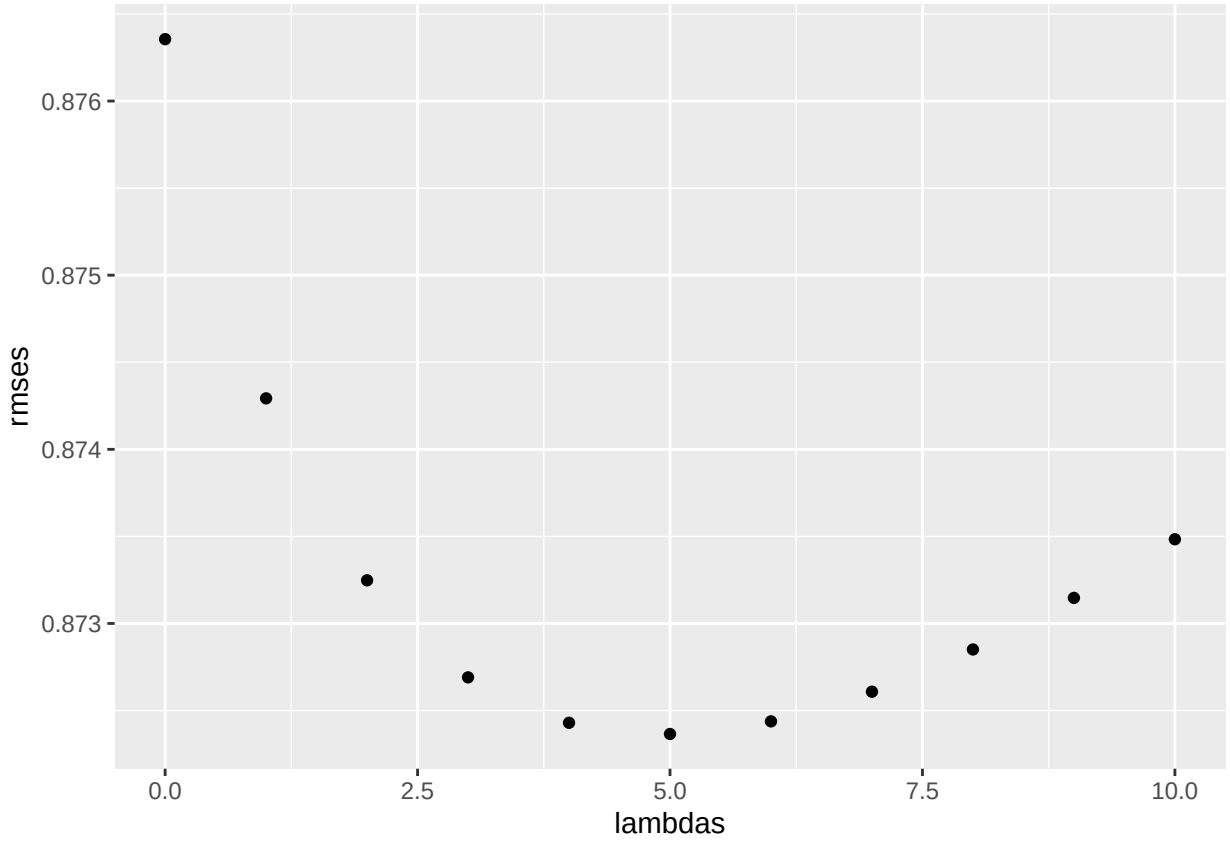

Figure 2: Regularization of user and movie effects

As figure 2 shows, this had clear effect, even if was not a dramatic one. At the optimal lambda, 5, the RMSE improved from just over 0.876 to just under .8725.

This was however on the small training and test set, and as it is related to the number of ratings per movie and user I had a suspicion effects on the larger set would be smaller and the optimal lambda a different one. Since these are still algorithms with a reasonable run time I ran it on the larger set as well.

As figure 3 shows I saw a much smaller effect from regularization on the larger set, but the optimal value was the same, so it made sense for my final model to include that.

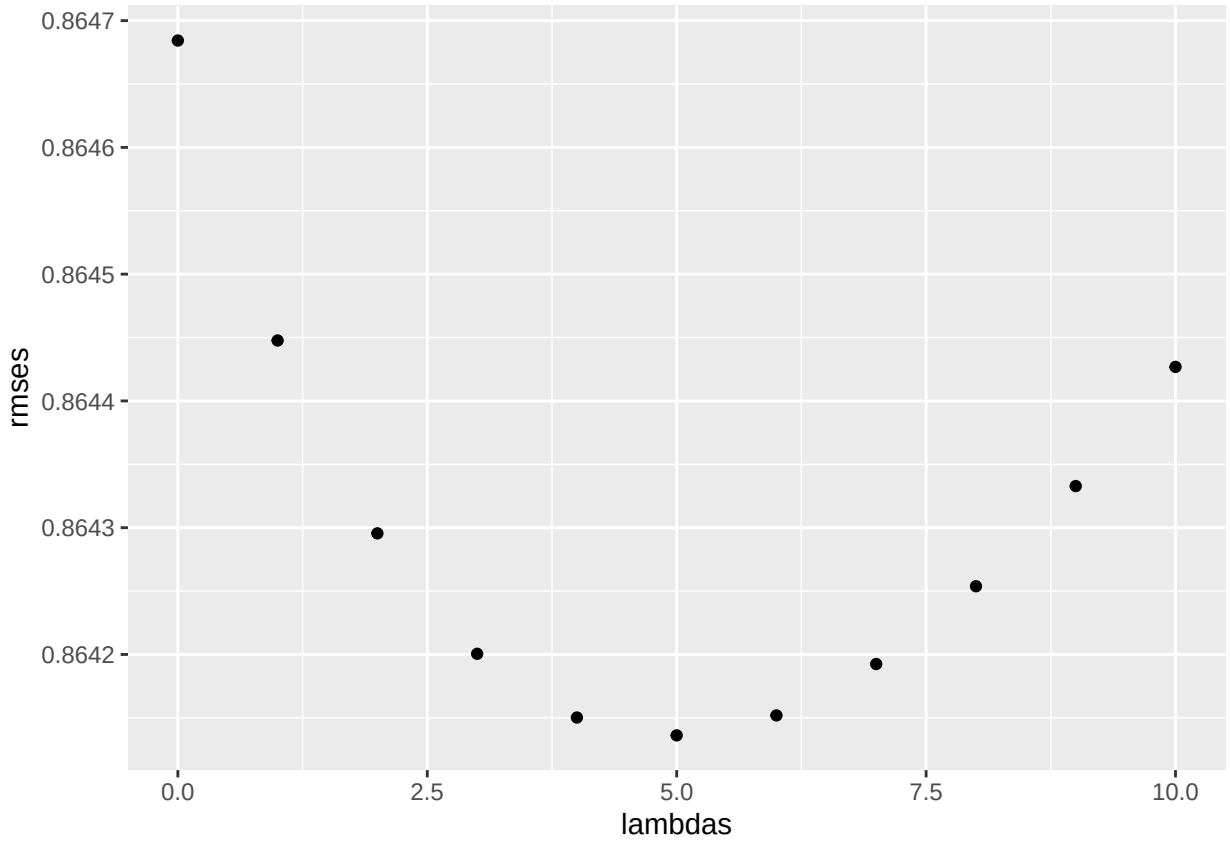

Figure 3: Regularization of user and movie effects on the larger set

Genre effects

My next analyses looked at additional data this set supplies. Each entry is of the form:

Table 5: Example rating

userId	movieId	rating	timestamp	title	genres
16	1196	5	912601645	Star Wars: Episode V ...	Action|Adventure|Sci-Fi

I could plausibly extract the release year and analyse ratings over time, but there was little reason to expect those to be different in a way that matters. On the other hand I considered it common sense that people like different genres and a plot of the distribution of ratings for each genre clearly showed a difference.

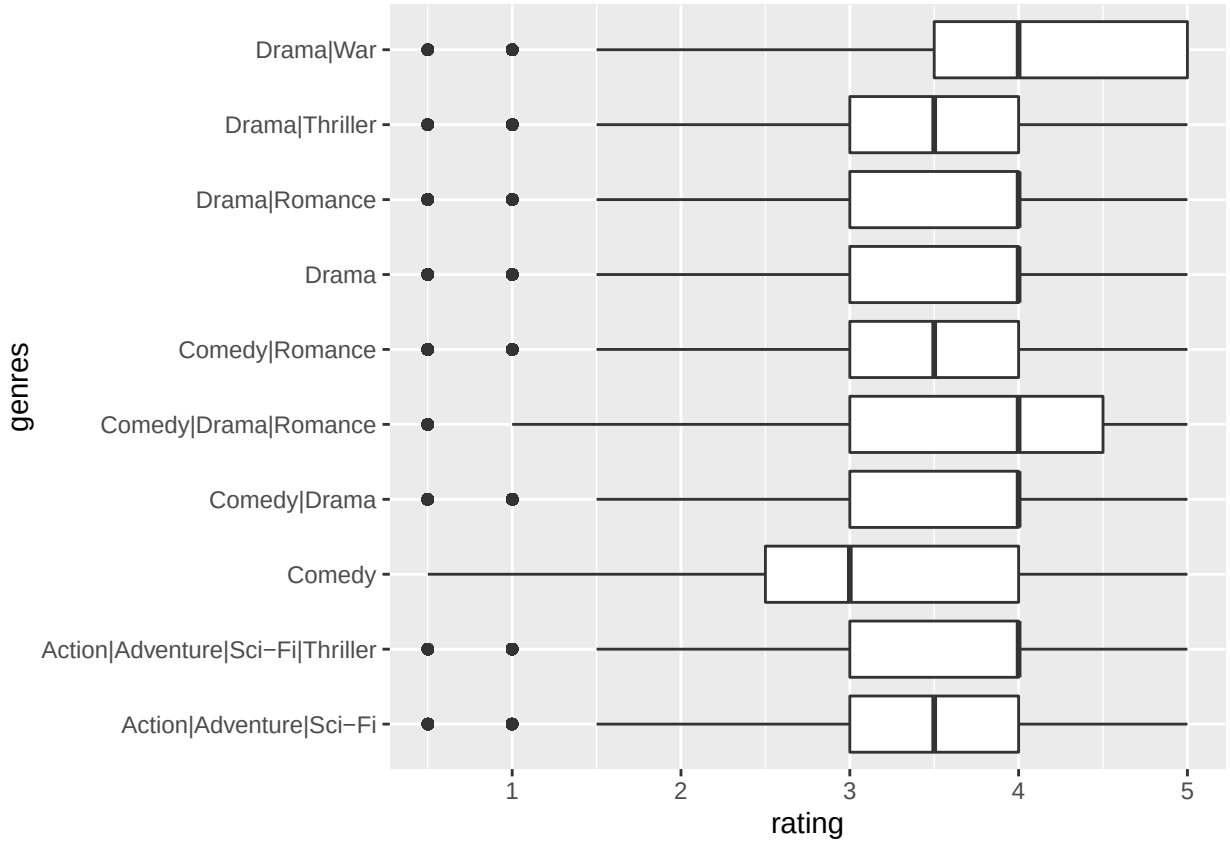

Figure 4: Boxplot of the most frequently rated genres

I tried to use this directly, adding a genre effect, but discovered I was replicating what is already built into the movie effect. I considered creating a matrix of genre effects per user, either for the genre as they are given in the dataset, with multiple tags in one field, e.g. "*action/comedy*", or splitting that data up and have a column for each of *action*, *comedy*, *drama* and so on, but I struggled to find a good way to model in a way that was less complicated than looking at the whole user/movie matrix and looking for patterns as done in `funkSVD()`, so I turned to finding a way to make that approach work with my hardware.

Single value decomposition

An important justification for not spending a lot of time figuring out how to use the genre tags is that they are not very specific. A comedy from the late 60s and one from the early 2000s might be the same genre, but

to a movie watcher one might be appealing and the other appalling.

I could extract the year from the title, decompose the genre column into separate columns for Comedy, Drama etc., but I'd still be missing patterns such as users liking specific actors, or directors, or big Hollywood productions rather than indie films. And since there already exists approaches to looking at the whole matrix of movies and users as our predictors, and identify all the strongest such patterns and classify movies through machine learning rather than relying on us adding data to each rating, it made more sense to use that approach.

The core of this approach it's to use a matrix with userIds along one dimension and movieIds along the other. I wanted to combine the use of this matrix with the movie and user effect model, so the matrix to analyze would be one with just the residuals after subtracting movie and user averages. I called this matrix m_res.

Table 6: Ratings matrix for train_set

	150	165	316	317	344
4	5	5	5	5	2.0
6	NA	NA	NA	NA	NA
7	NA	NA	NA	NA	NA
16	NA	NA	NA	NA	NA
18	4	NA	NA	NA	3.5

Table 7: Rating residuals matrix for train_set

	150	165	316	317	344
4	0.1690	0.582	0.6821	0.9186	-1.9228
6	NA	NA	NA	NA	NA
7	NA	NA	NA	NA	NA
16	NA	NA	NA	NA	NA
18	-0.0649	NA	NA	NA	0.3434

Table 6 and 7 show how these matrices differ by displaying a small subset. `m` shows how userId 4 rated movieId 150 slightly higher than userId 18 did, and vice versa for movieId 344. `m_res`, the matrix of residuals, shows that this difference is a lot smaller for movieId 150 when taking movie and user effects into consideration, and about the same for movieId 344.

It also shows us that this format contains a lot of NAs, which cause trouble for a lot of approaches to analyze matrices. It his however still possible look for patterns in movies and users, which is equivalent to looking for two smaller matrices U and V, that approximate this matrix, ignoring the NAs, when cross multiplied.

Matrix U will have one dimension equal to the number of users, and the other represents the number of user patterns we have identified. Matrix V has one dimension equal to the number of movies, and the other represents the number of movie patterns. The resulting matrix will have a predicted residual, $r_{u,i}$ for each user and movie.

$$r_{u,i} = p_{u,1}q_{1,i} + p_{u,2}q_{2,i} + ... + p_{u,m}q_{m,i}$$

p_u is the row of matrix U representing user u. $p_{u,1}$ is how strongly this user's ratings correlate with the first grouping of users.

q_i is the column of matrix V representing movie i. $q_{1,i}$ is how strongly ratings for this movie correlate with the first grouping of movies.

The product then represents how much this particular user is predicted to like this particular movie based based on these two groupings.

One term for these groupings is *features*, and with several of them, the predicted residuals for one particular movie and user is the dot product of the vector of features for that user, multiplied by the vector for features for that movie.

Creating U and a V with arbitrary values, cross multiplying the two, comparing them to `m_res`, adjusting all the values based on how well the result matches and iterating like that until a threshold for improvement is achieved, or a certain number of iterations has passed, give a U and V that can approximate `m_res`, but with values for every u and i, which gives us predictions also for movie/user-combination that were `NA` in the matrix put into the function.

This is implemented in the function `funkSVD` in the recommenderlab package. The report will detail later a model that incorporates this, and attempt a better justification of the approach, but I first ran it on `train_set` with parameters changed slightly to prioritize execution time rather than speed. Using this approach without the regularization of user and movie effects I got the improvement in RMSE shown in table 8.

Table 8: Using funkSVD() and validating on test_set.

method	RMSE
Movie+User Effect Model	0.8764
Movie+User+SVD Model	0.8461

From 0.876 to 0.846 is a huge improvement. If we got a similar improvement with the whole set that would be great. There is just two small problems.

`m` and `m_res` are very large matrices. For `train_set` they contain almost 66 million elements, the product of the number of movies and users. With the much larger `edx` set the matrices have 746 million elements, more than 10 times as much.

With increased size comes increased processing time and memory issues. `m` is 505 Mb, the full matrix is over 5 Gb. That ran close to the capacity of my computer, but I found I could prevent errors by deleting intermediate matrices and manually running garbage collection frequently. A bigger problem was that while the "quick" run of the function in my exploration took 22 minutes, with a matrix 10 times larger experimentation showed it didn't just take 10 times longer, i.e. 220 minutes or just shy of 4 hours, it increased by another magnitude. Not because it increases exponentially with size, but because at some point the computer started swapping memory to disk, with the time penalty that includes.

So I experimented with ways to use this approach, but with a subset of the data.

My idea was that since `funkSVD` takes a matrix with dimensions $m \times n$ and produces two matrices $m \times f$ and $n \times f$, representing patterns over two different dimensions of the analyzed matrix, I could do the fine adjustment of those matrices on the most dense data in either direction. I.e. I would run the analysis first on a smaller matrix $m \times \frac{n}{2}$, producing a decomposition matrices with dimensions $m \times f$ and $\frac{n}{2} \times f$, and then run it again, this time on $\frac{m}{2} \times n$, but keeping the $m \times f$ matrix constant, producing a corresponding $n \times f$.

To examine the viability of this I first did a simulation, but since it turned out to work just as well on the real data, I have moved the details of the simulation to appendix A.

Partial SVD

In addition to the idea described above of running the SVD analysis on parts of the data, my idea included strategically choosing what data to use. It's obviously true that you cannot establish reliable patterns of similarities for movies or users with very few ratings, so one idea is to sort the rows and columns by density and choose the densest parts.

Doing this risks eliminating some users or movies completely from one or both steps of the analysis, resulting in even larger inaccuracy. I briefly experimented with selection processes that would prioritize density, but also ensure movies or users with few ratings were represented, but ultimately I chose to use density alone. I later show how I tested one way to compensate for the movies and users potentially left out, and why I abandoned that approach to apply the same model to all ratings.

The function to run this partial SVD analysis, based on the function from the recommenderlab package, looks like this:

```
# If either U or V is supplied, that matrix will remain unchanged
# through the update process.
partialFunkSVD <- function(x, k = 10, U = NULL, V=NULL, gamma = 0.015,
                           lambda = 0.001, min_improvement = 1e-6,
                           min_epochs = 50, max_epochs = 200,
                           verbose = FALSE) {

  x <- as(x, "matrix")

  if (ncol(x) < k || nrow(x) < k)
    stop("k needs to be smaller than the number of users or items.")

  # initialize the user-feature and item-feature matrix if they're not parameters
  if(is.null(U)) {
    U <- matrix(0.1, nrow = nrow(x), ncol = k)
    updtU = TRUE
  } else {
    updtU = FALSE
    U <- as.matrix(U)[1:nrow(x),]
  }

  if(is.null(V)) {
    V <- matrix(0.1, nrow = ncol(x), ncol = k)
    updtV = TRUE
  } else {
    updtV = FALSE
    V <- as.matrix(V)[1:ncol(x),]
  }

  #list of indices pointing to ratings on each item
  itemIDX <- lapply(1:nrow(x), function(temp) which(!is.na(x[temp, ])))
  #list of indices pointing to ratings on each user
  userIDX <- lapply(1:ncol(x), function(temp) which(!is.na(x[, temp])))

  # go through all features
  for (f in 1:k) {
    if(verbose) cat("\nTraining feature:", f, "/", k, ": ")

    # convergence check
    last_error <- Inf
    delta_error <- Inf
    epoch <- 0L
    p <- tcrossprod(U, V)

    while (epoch < min_epochs || (epoch < max_epochs &&
        delta_error > min_improvement)) {
```

```
    # update user features
    error <- x - p
    temp_U <- as.matrix(U)

    if(updtU) {
      for (j in 1:ncol(x)) {
        delta_Uik <- lambda * (error[userIDX[[j]], j] * V[j, f] -
            gamma * U[userIDX[[j]], f])
        U[userIDX[[j]], f] <- U[userIDX[[j]], f] + delta_Uik
      }
    }

    # update item features
    if(updtV) {
      for (i in 1:nrow(x)) {
        delta_Vjk <- lambda * (error[i, itemIDX[[i]]] * temp_U[i, f] -
            gamma * V[itemIDX[[i]], f])
        V[itemIDX[[i]], f] <- V[itemIDX[[i]], f] + delta_Vjk
      }
    }

    ### update error
    p <- tcrossprod(U, V)
    new_error <- sqrt(sum(abs(x - p)^2, na.rm = TRUE)/length(x))
    delta_error <- abs(last_error - new_error)

    last_error <- new_error
    epoch <- epoch + 1L
    if(verbose) cat(".")
  }

    if(verbose) cat("\n-> ", epoch, "epochs - final improvement was",
      delta_error, "\n")
}

structure(list(U = U, V = V, parameters =
    list(k = k, gamma = gamma, lambda = lambda,
      min_epochs = min_epochs, max_epochs = max_epochs,
      min_improvement = min_improvement)),
  class = "funkSVD")
}
```

The various parameters, most of them from the original `funkSVD()` are

- x - the input matrix
- k - the number of features to create, the f in $m \times f$ and $n \times f$ previously
- U and V, my added variables, allowing one of the matrices to be from a previous analysis and remain unchanged. (You can actually supply both, and do a whole lot of nothing. A future version will include a warning about this.)
- *gamma* - the so called learning factor, which regulates the speed of change, to high a speed and the algorithm might oscillate instead of reaching an equilibrium
- *lambda* - a regularization factor reducing the impact of large and unusual elements

- *min_improvement* - a minimum improvement in RMSE to continue iterating over a feature
- *min_epochs* - a minimum number of iterations to perform, overrides *min_improvement* and *max_epochs*
- *max_epochs* - a maximum number of iterations to perform, overrides *min_improvement*
- *verbose* - whether or not to visually indicate progress, very useful when the algorithm moves slowly

The output is a structure containing the two decomposition matrices, U and V and a list of the parameters used.

Optimizing SVD approach on train_set

The analysis on the artificial data (see Appendix A) showed that good results could be had by using SVD on a subset on the data, but the real data is a lot sparser, and picking the densest parts of the set might leave some users and movies completely from the whole analysis.

There are at least two possible approaches to dealing with this.

- When choosing columns (or rows) ensure that no rows (or columns) end up being all NAs.
- Decide that predictions for those users and movies would improve little anyway and possibly use only the simple model for those users and movies left out.

The first option requires tuning and regularization, so due to time and hardware constraints I chose the second.

Baseline

I wanted to test how much data I could leave out, with the goal of running the algorithm on the full `edx` set with a reduced run time, so I ran the analysis on `train_set` and `test_set` again, but with the default parameters, to establish a baseline to compare with.

Running it with the default parameters gave this result.

Table 9: RMSEs

set	method	RMSE
test_set	Movie+User Effect Model	0.8764
test_set	Movie+User+SVD Model	0.8461
test_set	M+U+SVD def.param. Model	0.8431

For one thing this showed a good improvement from changing k from 5 to 10 and/or max_epochs from 100 to 200. Looking at the detailed output to console showed that it ran 200 iterations for each feature, so there would be likely be more improvement if max_epoch was changed beyond the default.

What mainly influences the run time, other than the size of the data and the strength of the patters are the five parameters k, *max_epochs*, *min_epochs*, *min_improvement* and *lambda*.

Ideally I'd wanted to tune all of these, but the execution time for each application of the function on anything other than a trivial subset of the data prohibited that. I opted to only examine the effects of varying the proportion of the data to run the partial analysis on, and the number of features k.

Reduced data

Because evaluating the benefits of various ways of selecting the subset of was going to be too time consuming I decided to sort the rows and columns by the density of ratings and select the densest x percent along each dimensions, testing for x in $\{20, 30, 40, 50\}$. I could have further made note of and removed vectors in the

other dimension that were made very sparse by that operation, but chose not to for the sake of reducing complexity.

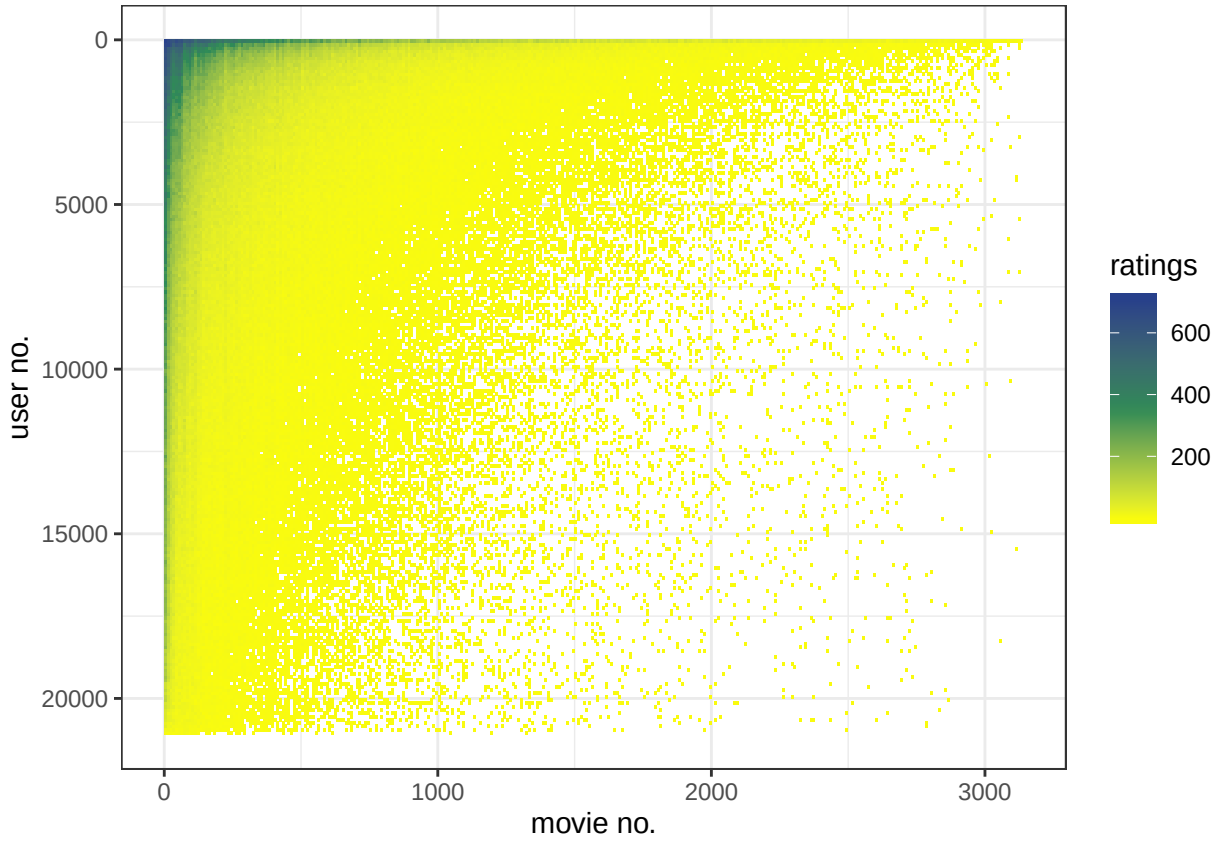

Figure 5: Matrix of residuals sorted by density

Sorting the matrix also gave me an opportunity to visualize the density. Based on the mathematics of SVD and the density of ratings in figure 5 it seemed plausible I might find good patterns all the way down to 20 % of each dimension. When calculating the error the algorithm creates values for the whole matrix, but the lower right of it is almost all NA, so even though this approach goes through the process of decomposing the matrix twice, I still hoped gain some speed, especially if I could get below the point where the computer starts swapping memory to disk.

The plot is slightly misleading though. Each bin has 1000 user/movie pairs so even the densest of them is in reality only 60% filled and most, as can be seen are less than that. So it didn't rule out the chance of a row or column being completely empty when analysing just a subset. To make a choice between applying the prediciton from the decomposition to all the data, or just those users or movies that had been involved in both analyses, I calculated the RMSE in two ways, giving this result.

Table 10: RMSEs - partialSVD analysis

method	percentage	RMSE_partial	RMSE_full
partialSVD model	100	NA	0.8431
partialSVD model	10	0.8473	0.8450
partialSVD model	20	0.8447	0.8442
partialSVD model	30	0.8430	0.8428
partialSVD model	40	0.8434	0.8433
partialSVD model	50	0.8444	0.8443

Table 10 clearly show this idea works. All the RMSEs are a large improvement on the simpler model, and working on 30% even outperforms, by a hair, running the function on the whole matrix. Applying this new SVD-term to all of the data also appears to be, if not a lot better, at least not worse than only running it on the dense area that was fully represented in the analysis. This could be because the frequent raters and the most rated movies are actually representative, or it could mean that the the ratings just outside the boundary get more precise and that increased errors for the sparsest part of the matrix aren't big enough to bring the result down.

This could be analyzed by doing a heatmap of errors on the matrix, but I chose to instead spend the time tuning the number of features. Ideally I would have done this tuning on a larger sample of the training set, but even reduced down to two runs on 30 % of this scaled down set, the process took 8-10 hours, so I chose to only tune k on the smaller `train_set`. I would only have needed to run the analysis once, since the first, say 5 features of a run with k = 10 will be the same as if we ran it with k = 5, but I chose to prioritize completing the project.

Figure 5 showed that the best RMSE for this training set was at k=5, so I decided to have my final model would be based on running the decomposition with 30%, k=5 and otherwise default variables and that I would apply the resulting predicted residuals to all ratings.

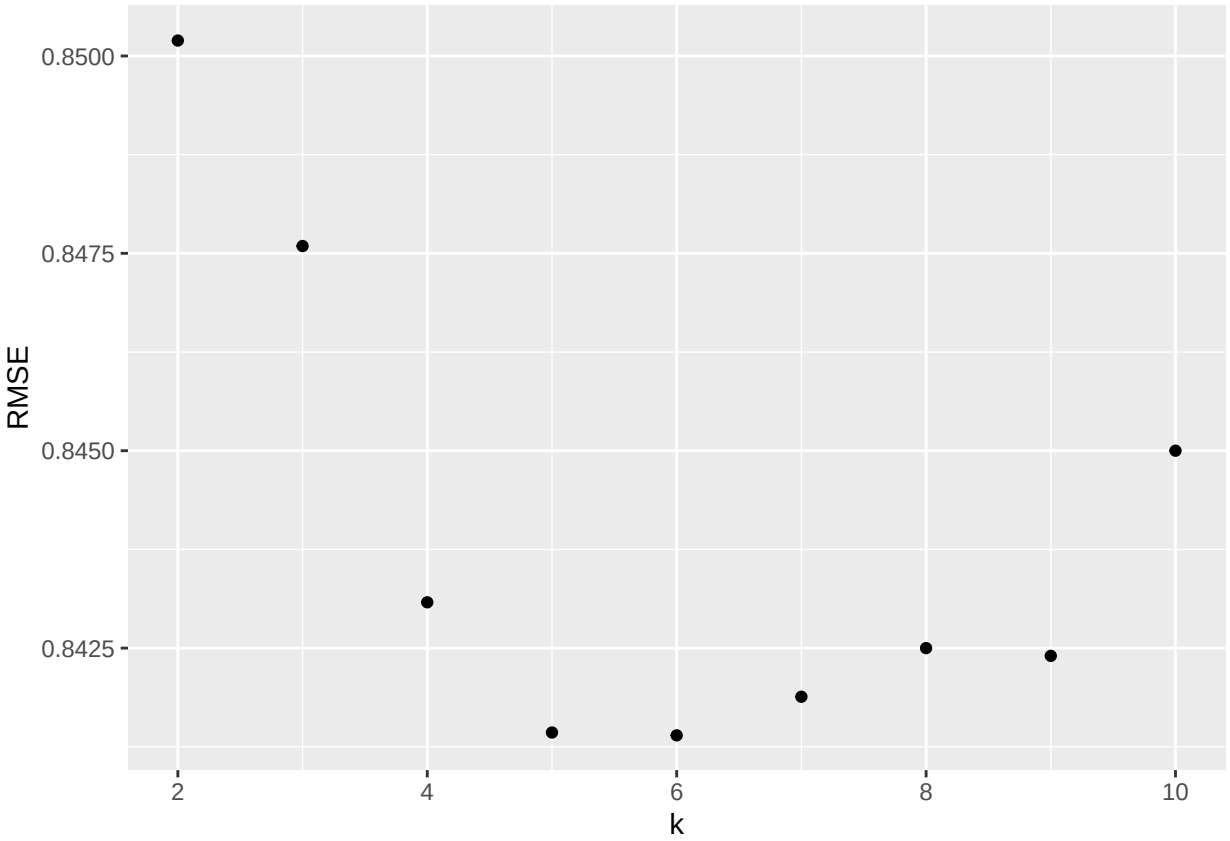

Figure 6: RMSEs for different values of k

Results

The final model then can be described thus:

$$Y_{u,i} = \mu + b_i + b_u + \sum_{k=1}^{5} U_{u,k} V_{i,k}$$

$Y_{u,i}$ is our prediction for user u and movie i. μ is the average rating for the whole data set. b_i and b_u are movie and user effects, respectively, regularized with a λ of 5. And finally $\sum_{k=1}^{5} U_{u,k} V_{i,k}$ is the residual predicted for this particular user/movie pair by the decomposition analysis.

The full analysis from data download to validation was implemented in the script *edx-ML-analysis.R*. The main steps in it are:

1. Downloading data
2. Creating training set and validation set
3. Calculating the training set mean
4. Calculating the regularized movie effects
5. Calculating the regularized user effects
6. Widening the data to a matrix of ratings with all users in rows and movies in columns
7. Sorting the matrix by row and column density
8. Running `partialFunkSVD()` with all the movies and the 30 % most prolific raters to create the movie feature matrix V.
9. Running `partialFunkSVD()` with all users, the 30 % most frequently rated movies and with V as an input variable to create the user feature matrix U
10. Using the data from this analysis, applying the model previously described to calculate predictions for the test set
11. Calculating the RMSE for the difference between predictions and actual ratings in the test set.

Running this analysis on the full `edx` set produced an RMSE of 0.8115521.

Based on the evaluation criteria given for the project and the improvement over simpler approaches this seems to be a very good result. I would be more confident if I could have run a cross validation of a large portion of the training set without it taking several days of computing time, but as I never used the validation set during analysis or training, it appears to be a valid result, and not the result of overfitting, although I cannot rule out a fluke.

Conclusion

In this project I have shown that we can predict what rating a particular user will give a particular movie with an RMSE of less than .9 using a simple approach with the mean rating, a movie effect and a user effect. I have shown that this can be improved a little by regularizing the effects, penalizing movies and users with few ratings. I've also confirmed it can be improved a lot by using a form of single value decomposition. And I appear to have shown that it can be further improved from the `funkSVD()` approach by applying the decomposition to the densest part of the data in two steps.

Due to time and hardware limitations I have done less tuning and cross validation than I would have liked, and there are also multiple open questions about potential further improvements.

- I do not know if changing the tuning parameters in `funkSVD()` from the default have significant effects on the model. I did some small experiments left out from the final report that seemed to show very minor differences and did not trust changing them without cross validating a larger training sets.
- Prolific raters might be different from less frequent raters. In this challenge they are also more important, since they have a bigger influence on the RMSE, but in a real world recommendation system you want to give good suggestions to everyone, even if they're not good at giving you feedback.
- A different approach to subsetting could be tried. I experimented with selecting a certain percentage of ratings for each user and movie, or x, whichever was larger, to reduce the dominance of the most prolific raters and possibly increase accuracy for infrequent raters, but I did not run the analysis on such a subset as I assumed the RMSE would be worse even if it might be more accurate for those infrequent raters.

Appendix A

Exploring the viability of the partial SVD approach through simulation

Before I applied the partial SVD approach to the real ratings data I tried it out on some simulated data to be sure that the method made sense. Since the results on the `train_set` were so good, and since this section is a bit long, I chose to move it all to this appendix.

Simulated data

I started by defining three different relationships between 1000 different "movies".

```
# set.seed(1) # use if older version of R
set.seed(1, sample.kind = "Rounding")

# We're going to have 1000 movies, distinguished by 3 different classes.
N = 1000

# Our final residuals will be produced by u1*v1 + u2*v2 + u3*v3
# To get the range to be, mostly, -2, 2, we need the actual max/min to be +/- sqrt(2/3)

f = sqrt(2/3)

# The first class distinguishes the movies at each end from each other
v1 <- numeric(length = N)
for (i in 1:N){
  # We want the feature adherence to range from -2 to 2 with some randomness
  v1[i] <- 2*f/1000 * i - f + rnorm(1, sd = 0.1)
}

# The movies in the middle belong strongly to this class, the movies at the end randomly
# do or do not
v2 <- numeric(length = N)
for (i in 1:N){
  if(i > 300 & i < 700) {
    v2[i] <- rnorm(1, mean = f - 0.5, sd = 0.1)
  } else {
    v2[i] <- rnorm(1, sd = 0.1)
  }
}

# Belonging to the third class varies cyclicly
v3 <- numeric(length = N)
for (i in 1:N){
  # We want the feature adherence to range from -2 to 2 with some randomness
  v3[i] <- rnorm(1, mean = 0.8*f*sin(i), sd = 0.1)
}

V <- cbind(v1, v2, v3)
rm(v1, v2, v3)
```

This defines three ways the different movies relate to each other. Some movies strongly belong to several groups, some belong only to one, some to none. I then did the same for 1000 "users".

Next I took the cross product of the two and added a little more randomness, to get the "real" residuals.

```r
# Creating matrix and randomizing order
gen_res <- tcrossprod(U, V)
u_index <- sample(1:1000, 1000)
v_index <- sample(1:1000, 1000)
gen_res <- gen_res[u_index, v_index]

# adding randomness
noise <- rnorm(N*N, sd = 0.2)
noise <- matrix(noise, N, N)

gen_res <- gen_res + noise

rm(u_index, v_index, noise)
```

Then I deleted ratings, with higher and higher probabilty the higher the row or column number, creating a somewhat realistic data set with some users having rated lots of movies, and some few, and some movies being rated often, and some rarely.

```r
gen_res_sparse <- gen_res
for (i in 1:N){
  for (j in 1:N) {
    # formula created through trial and error to get a reasonable
    # final distribution
    p = 0.5*exp(-0.000015*i*j) + 0.0005
    del <- sample(c(TRUE, FALSE), 1, prob = c(1-p,p))
    if (del) {
      gen_res_sparse[i,j] <- NA
    }
  }
}
```

This is a lot less sparse a matrix than the one for the real dataset, with just over 10 % of the possible user/movie pairings, but it has a nice distribution as shown here.

Next I needed a train set and a test set. I parsed the matrix, picking a random ten percent as a test set stored in three columns (u, m, v), and at the same time set those to NA in the original matrix.

Analysis of simulated data

This data was generated by cross multiplying two N by 3 matrices U and V, transposing the second one, so every value $r_{u,i} = U_{u,1}V_{i,1} + U_{u,2}V_{i,2} + U_{u,3}V_{i,3}$, with a little randomness introduced. This is precisely the kind of patterns SVD is good at revealing. To get comparison values for my modified version of SVD I ran funkSVD on the full training set trying to predict 1 to 5 features, with otherwise default parameters.

Table 11: RMSEs on generated data with funkSVD

k	RMSEs
1	0.3475
2	0.3191
3	0.2806
4	0.2457
5	0.2438

There was an improvement in RMSE up to four features, one more than what we used to generate the set,

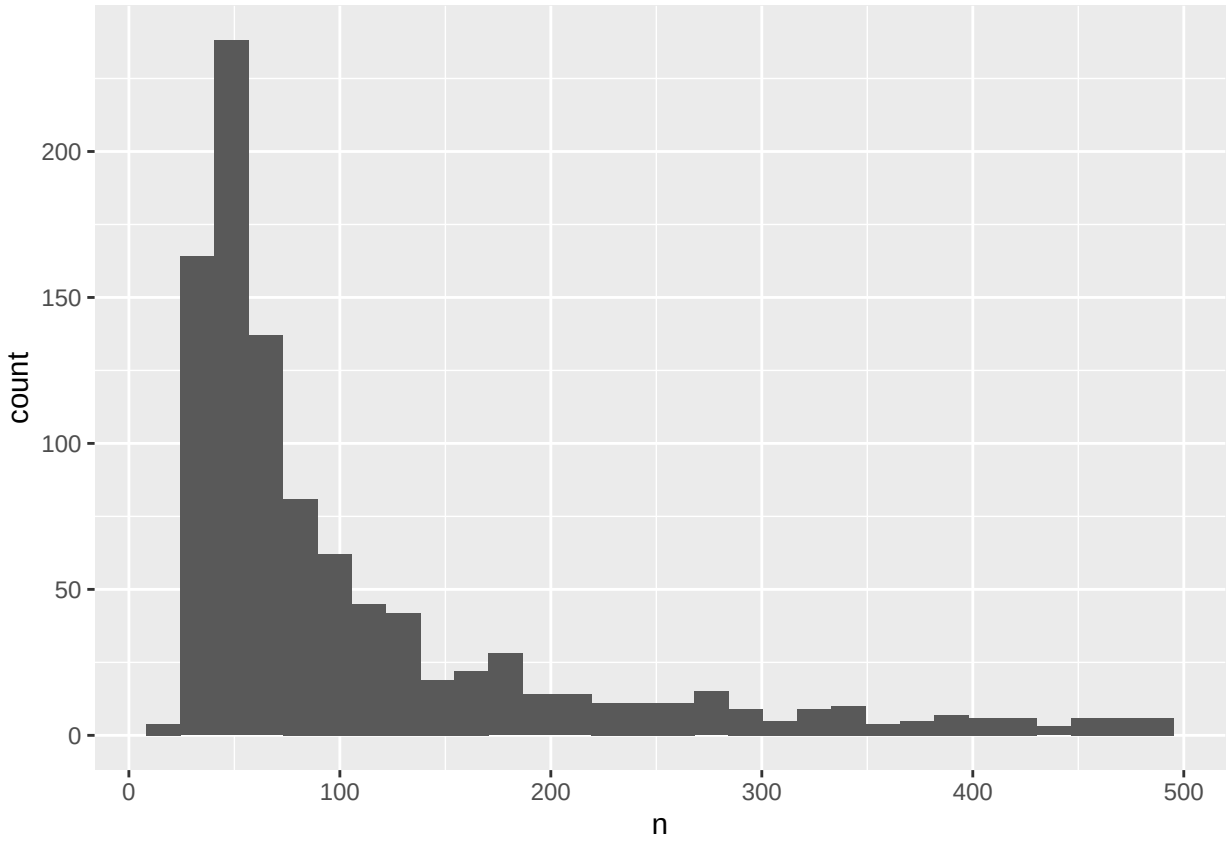

Figure 7: Ratings per user

possibly just random chance. Using a cross validation approach would likely have shown that.

The dataset was created with decreasing density for higher row and column numbers, which we can see in this visualization of the test set, where each point is a 10 by 10 bin of user/movie pairs.

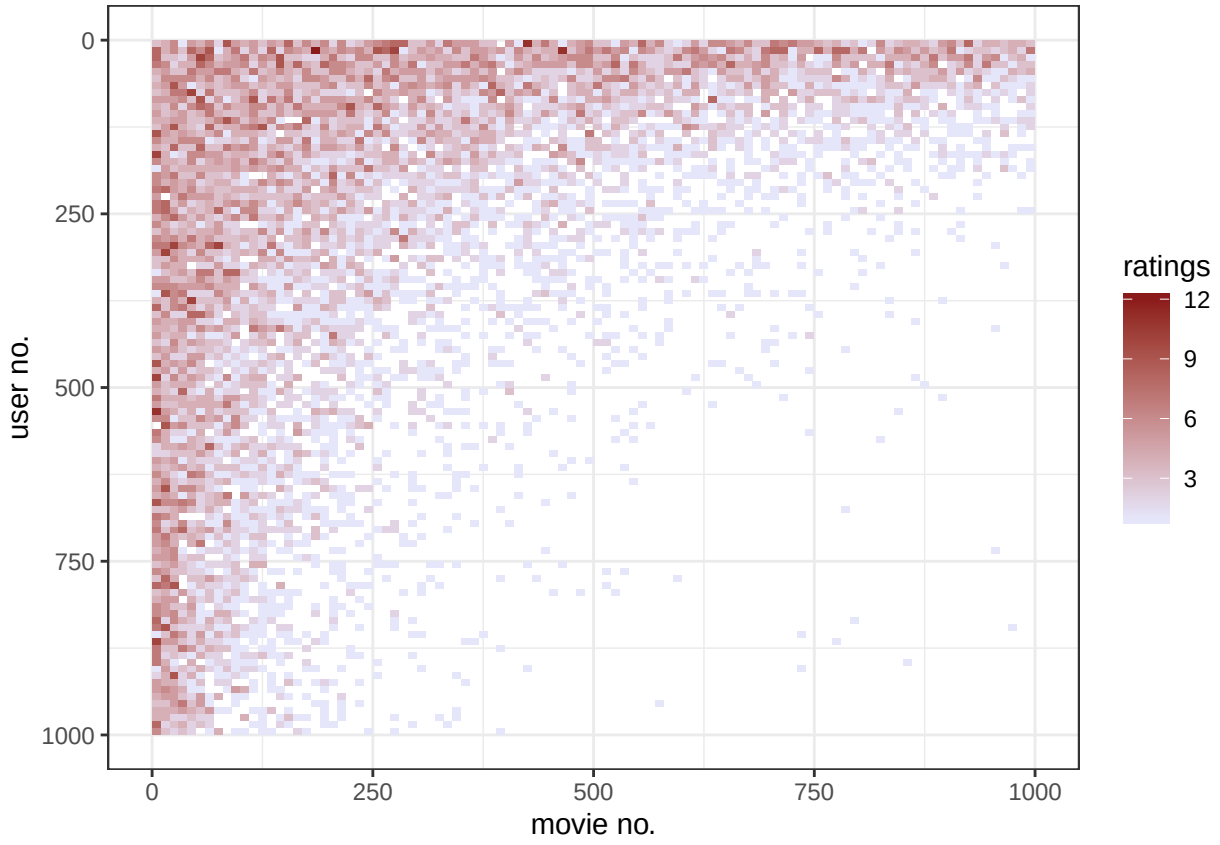

Figure 8: Visualization of the size and position of generated ratings

I repeated the analysis using the same parameters as before, once combining the decompositions produced by the 50 % of movies and the 50% of users, and once using 25%.

In pseudocode:

```
# Produces a user feature matrix U and movie feature matrix V
# based on the movies with the most ratings
decomp <- partialFunkSVD(matrix[, 1:half])

# Produces a move matrix V, based on the users with the most ratings
# and the user features for those users
decomp2 <- partialFunkSVD(matrix[1:half, ], U = decomp$U[1:half,])

# Creates predictions for all users and movies based on
# the output from the two runs
```

The results of the analysis is given in this table:

Table 12: rmses on generated data with partialFunkSVD

k	RMSE500	RMSE250
1	0.3214	0.3399
2	0.3080	0.3158
3	0.2315	0.2534
4	0.2322	0.2465
5	0.2341	0.2459

These RMSEs compared favorably to the ones produced by running the analysis on the whole dataset. Running on 50 % of the dataset actually showed better results, possibly because there is an in built regularization of using features trained on data that excludes the users with the fewest ratings when creating movie features and vice versa. Even at 25 % I get almost the same level of results, while reducing execution time significantly. But this artifical dataset is much denser than the actual database, so I was not very confident it would work there.