MovieLens Rating Prediction Using SVD and Funk SVD Ensemble Models

Rebecca L.E. Miller

28 October, 2020

Summary

This analysis presents a rating-based recommender system for the MovieLens 10M dataset. Per the capstone project instructions, the 10M dataset was subdivided into an `edx` set and a `validation` set with 90% of the data in the `edx` set. We further subdivided the `edx` set into `train` and `test` sets, again with the $9:1$ ratio, for our initial training, analysis, and model development. The system we developed was an ensemble blend of effects calculated from grouped averages and regularized to penalize small sample size, with two effects modeled with matrix factorization. The Funk SVD method was used for the user/genre interaction and standard SVD was used to model the user/movie interaction. As a final step, the prediction was coerced to fit within the known range of the `validation` set ratings, with the clipping function referred to as $\kappa()$. All of the regularized effects were trained on the `edx` set with internal cross-validation of 5x, including the input to the Funk SVD user/genre model. The two matrix factorization models were trained with 1x cross-validation on the `edx` set. The final model took the form of the following equation.

$$\hat{y}_{u,i,t} = \kappa\Big|_{0.5}^{5.0}(\hat{\mu} + \hat{b}_i + \hat{b}_u + \hat{b}_{i,t} + \hat{b}_{u,t} + \hat{b}_{u,g} + \hat{b}_{u,i})$$

And the final $RMSE$ was

$$RMSE_{final} = 0.8228107$$

We conclude that matrix factorization is an excellent tool for the machine learning toolbox as it allows us to fill empty cells in a response ~ predictor matrix while simultaneously allowing us to reduce the dimensionality of our data. Our prediction used just $K = 12$ and $K = 80$ components, respectively for the user/genre and user/movie interactions, reducing the user/movie data from a single matrix of about $70,000x10,000$ to two matrices of $70,000x80$ and $10,000x80$ with a third $80x80$ element or $7e+8$ elements down to $6.4e6$ elements, a reduction of two orders of magnitude. With ever-increasing access to data, tools that provide reduction in complexity will be increasingly valuable in the machine learning world, and with this analysis we can add SVD and FSVD to our toolbox.

Introduction

The task set forth in the MovieLens Capstone Project is to create a recommendation system that can predict how a user will rate a movie based on that user's past ratings and the ratings of other users. This is a task similar to the algorithm once used by Netflix for recommending movies or by Amazon for recommending new products and has become its own class of machine learning task, with the term of art being a recommender system. The motivation for examining these systems is both in the ubiquitous nature of the task, to predict what movies or products someone will like, and in the potential monetization of the concept. Netflix has a business model that is strongly based in its recommendation algorithm and has a current market capitalization in the range of $218B, while Amazon has a market capitalization of roughly $1.48T,[1] more than the Gross Domestic Products of Greece and Spain, respectively.[2]

In the course material for the EdX Data Science Capstone, we examined the grand prize winning solution to the Netflix Prize. The Netflix Prize was a competition started in 2006 with a prize of $1M and the goal of improving on Netflix's own Cinematch algorithm for predicting movie ratings. The Netflix Prize dataset consisted of just four predictors, user numeric identifier, movie numeric identifier, date of grade, and grade. Competitors were to construct an algorithm that would predict unknown grades of the same format in a portion of the dataset that was held back, a simulation of Netflix's day-to-day task of recommending movies to users based on their past preferences. Within two weeks of the competition start, three teams had improved on Cinematch's performance. Two teams were eventually able to improve upon the performance of the Cinematch algorithm by 10% or more and the grand prize was awarded in 2009 to the "BellKor's Pragmatic Chaos Team". The competition was discontinued, however, due to privacy concerns and the resulting lawsuits after individual users were identified in the dataset by matching their movie ratings in the Internet Movie Database (IMDb). Ultimately, Netflix never incorporated the grand prize winning algorithm into its recommender system, in part because Netflix moved to online streaming and therefore changed its recommendation scheme.[3] With online streaming, hits were used to identify user preference. A user was "interested" in an item if streaming was started and "liked" an item if they watched it to completion. Netflix ultimately found these hits to be better predictors of preference than a user's overt ratings.[4] As such, the coursework that essentially worked to duplicate the efforts of the prize-winning team is not particularly relevant to Netflix's business model today, but it is still a good exercise for the demonstration of how to construct a recommendation system, in general. In fact, some of the same types of analyses can be applied to user "hit" data as those applied to ratings data.

The data provided to us for our version of the Netflix Prize task is from the GroupLens research lab at the Department of Computer Science and Engineering at the University of Minnesota, Twin Cities and is derived from the online movie recommender service, MovieLens. [5][6] There are several datasets available. Some are stable benchmark sets that do not change over time, while others include new ratings and are therefore constantly changing. Our data is from the 10M dataset, a benchmark set that contains approximately 10 million ratings from 72 thousand users for 10 thousand movies with 95 thousand associated tags.[7] While the 10M dataset is large, the larger 25M dataset is the one recommended by GroupLens for new research. Even at 10 million ratings, the 10M dataset becomes unwieldy for certain modeling tasks, as we will see in later sections of this analysis.

Details of the 10M Dataset

The MovieLens 10M Dataset is considered to be, as previously mentioned, a stable benchmark set and was released January 2009. According to the README file, [8] the data are contained in three UTF-8 encoded files of the formats in the table below.

Each row of the `ratings.dat` file represents one rating by one user, ordered by `UserID` then `MovieID`. Each rating is on a 5-star scale, with half-star increments, and the `Timestamp` is measured in seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970, or seconds since the epoch. While the `tags.dat` file contains information that could potentially be very useful in predicting user ratings, the instructions for this capstone project explicitly say to develop the algorithm using the `edx` dataset, which we will generate from code provided to us in the course materials. That dataset does not include any of the information encoded in the `tags.dat` file, so we will not consider it to be part of our analysis here. Finally, the individual movies

[1] https://finance.yahoo.com/amzn or /nflx, Market Cap

[2] https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)

[3] https://en.wikipedia.org/wiki/Netflix_Prize

[4] https://www.wired.com/2013/08/qq-netflix-algorithm/

[5] http://www.movielens.org/

[6] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

[7] http://files.grouplens.org/datasets/movielens/ml-10m-README.html

[8] http://files.grouplens.org/datasets/movielens/ml-10m-README.html

Table 1: MovieLens 10M Database File Names and Formats

File	Format
ratings.dat	UserID::MovieID::Rating::Timestamp
tags.dat	UserID::MovieID::Tag::Timestamp
movies.dat	MovieID::Title::Genres

Table 2: MovieLens 10M Genre Options from README file

Action	Crime	Horror	Thriller
Adventure	Documentary	Musical	War
Animation	Drama	Mystery	Western
Children's	Fantasy	Romance	
Comedy	Film-Noir	Sci-Fi	

are given a distinct `MovieID` that is linked to a `Title` and a `Genres` list. The title should be entered in the Internet Movie Database (IMDb) style, including year of release in parentheses. Genres are provided in a pipe-separated list selected from the options in the table below.

However, the README file states that these movie titles are entered manually, so we can't presume that any of the dataset variables will be error-free or uniformly encoded. So, let's dive in and explore the data.

Methods

Generating Data Sets from the 10M Files and Data Exploration

Script Chunk Provided in Course Material

In order to explore the data, we need to download it and wrangle it into a format that we can manipulate with `R`. An R script chunk was provided to us in the course materials for the purpose of downloading and extracting the dataset for this Capstone Project and that code is included in the raw markdown file as well as the script file but it will not be displayed here. We assume that students who have completed the course will be familiar with this preliminary code, but encourage any reader who is not familiar to read through it in the `*.R` and `*.Rmd` files.

The instructions for the Capstone Project tell us to use the `validation` set as the final hold-back set for calculating our model loss function, which was defined to be the root mean squared error (RMSE). This means that intermediate development analysis and decision-making must be done with another test set that we hold back from the `edx` set. We therefore create a `test` and `train` set from the `edx` set using the same methods as the course-provided parsing code.

```r
#---------------- Create a Train and Test Set -------------#
# Create a test set from the edx train set for model development
test_index <- createDataPartition( y = edx$rating, times = 1,
                                   p = 0.1, list = FALSE)
train <- edx[-test_index,]
temp <- edx[test_index,]

# make sure all userId and movieId in test set are also in train set
test <- temp %>%
  semi_join(train, by = "movieId") %>%
  semi_join(train, by = "userId")

# Add rows removed from test set back into train set
removed <- anti_join(temp, test)
train <- rbind(train, removed)

# Remove the variables we no longer need
rm(temp, test_index, edx, removed)
```

Data Exploration

We examine the data structure and the values in the first few rows.

```
head(train,3)
```

```
##    userId movieId rating timestamp            title
## 1:      1     122      5 838985046 Boomerang (1992)
## 2:      1     185      5 838983525  Net, The (1995)
## 3:      1     292      5 838983421  Outbreak (1995)
##                           genres
## 1:                Comedy|Romance
## 2:         Action|Crime|Thriller
## 3: Action|Drama|Sci-Fi|Thriller
```

We find that our `train` set ended up containing just over 8.1M individual ratings, or rows. Our `userId` and `movieId` are both numeric, so we can use those values without having to convert them, although there may be instances when we need to treat them as factors. We will need to extract the release year from the title variable and we will need to convert the `timestamp` to one of the POSIX formats. Additionally, if we would like to consider each genre in our model as a separate predictor, we'll need to split out the genres list into binary predictor columns. We have the information from the `README` file concerning our data formats, but we should check those assumptions before we continue.

`userId` We examine the variable type, range, and number of distinct values for the `userId` variable with the code below. We verify that `userId` is already numeric, therefore requires no additional transformation, though there may be some instances where we would like to treat it as a factor. We see that there are unused user IDs between the minimum value and maximum value as indicated by differing values for the range and number of distinct `userId`s. This means we can't just use a sequence from `1:nrow(train)` as a replacement for `userId` and we'll need to be careful to keep track of indexes. The data type can be found with,

```
str(train$userId)
```

```
##  int [1:8100067] 1 1 1 1 1 1 1 1 1 1 ...
```

and the range of values in userId with

```
range(train$userId)
```

```
## [1]     1 71567
```

and number of distinct users.

```
n_distinct(train$userId)
```

```
## [1] 69878
```

A histogram of `userId`, shown below, shows us that frequency of ratings is fairly level across the range of `userId`, with the possible exception of the very last bin, so we have relative confidence that there is no underlying structure to the variable that will interfere with our machine learning tasks. Closer inspection of the last bin reveals that the range doesn't extend to the end of the range of that bin, so we expect that bin to contain fewer ratings.

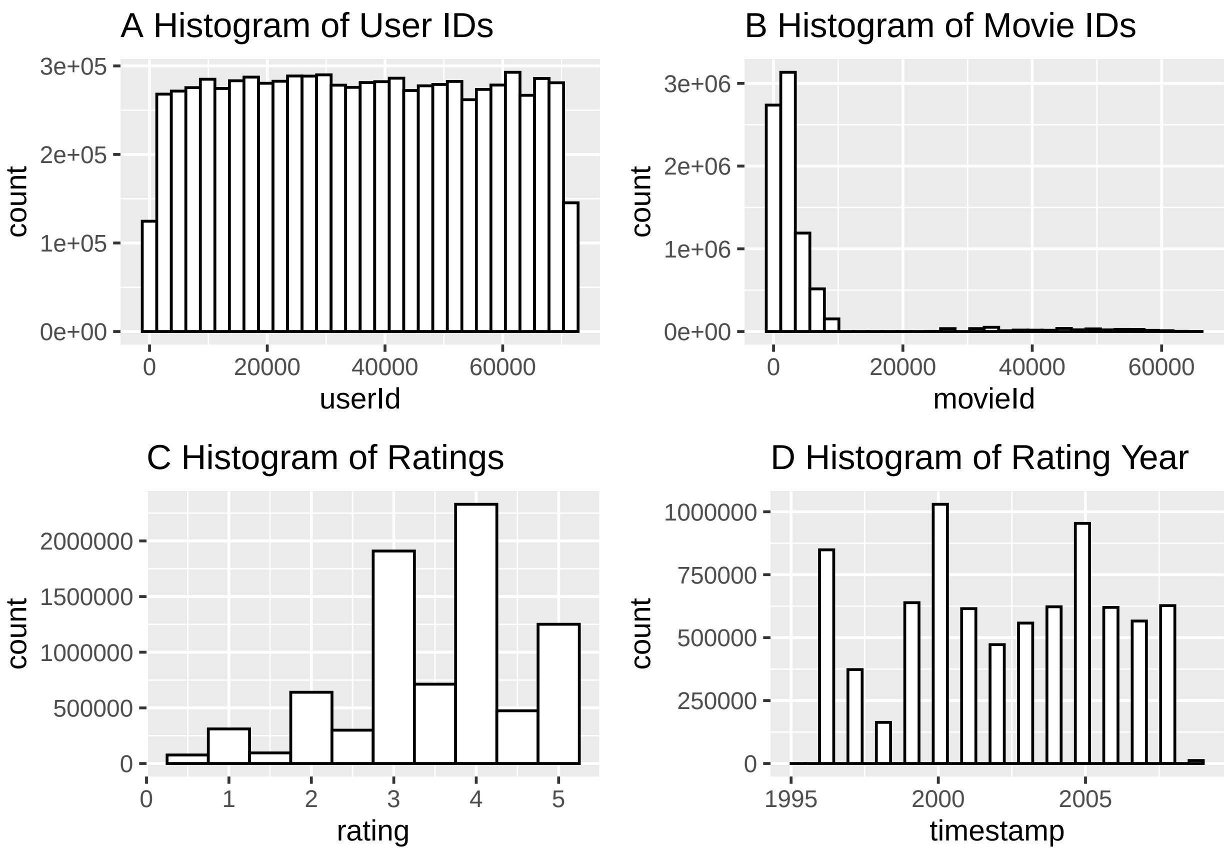


Figure 1: (A) Histogram of userId. Level frequency across the range of userId suggests no underlying structure to the variable. (B) Histogram of movieId. The majority of ratings are almost entirely from the first 10,000 movie IDs. (C) Histogram of rating. Shows a higher frequency for natural numbers, or whole-valued ratings, with the majority of ratings in the "3" or "4" levels. (D) Histogram of timestamp. Shows some potential underlying structure with rating activity varying over time.

`movieId` We also examine the `movieId` variable type, data range, and the number of distinct movie IDs. We find that, like `userId`, `movieId` is already numeric and has missing values because the number of distinct users is not the same as the range of the data.

We see that the basic structure of `movieId` with a structure

```
str(train$movieId)
```

```
##  num [1:8100067] 122 185 292 329 356 362 364 370 377 420 ...
```

And we can see the range of values

```r
range(train$movieId)
```

```
## [1]     1 65133
```

with the number of distinct users found with

```r
n_distinct(train$movieId)
```

```
## [1] 10677
```

A histogram of `movieId` shows that the majority of ratings are from the first ten thousand movie ID numbers. This will warrant further investigation into the relationship between release year or `timestamp` and `movieId` under the assumption that `movieId` is assigned with respect to release year or perhaps by number of ratings.

rating We now turn to the `rating` variable and find that it is not a continuous variable, but rather can only take on ten values, the half-star values between 0.5 and 5.0, and does not include zero. This suggests that we should consider coercing our predictions to match the range of `rating`, or predicting a "0.5" whenever our rating is less than that value and predicting a "5.0" whenever our prediction is higher than five. We also see that the mean value of `rating` is around 3.5 with a standard deviation of about 1.0, showing that `rating` skews toward more positive values of the available range. Note that 0 is not included in possible rating values

```r
  levels( factor(train$rating) )
```

```
##  [1] "0.5" "1"   "1.5" "2"   "2.5" "3"   "3.5" "4"   "4.5" "5"
```

The average rating in the `train` set is found with

```r
  mu = mean(train$rating)
  mu
```

```
## [1] 3.512509
```

And the standard deviation with

```r
  sd(train$rating)
```

```
## [1] 1.060242
```

A histogram of the `rating` variable reveals that the majority of ratings are either 3's or 4's. The single most probable rating is a 4, but we'll see that the selection of our loss function will drive our choice of the first term in a linear model toward the average rating rather than the most likely rating. Having noted that the actual data is discrete, we will therefore consider coercing our predicted ratings to fall within the range of `rating` and possibly to also coerce it to take on half-rating values, as well.

`timestamp` Examination of the `timestamp` variable shows that it is, indeed, of the format described in the README file for the 10M database, with a range consistent with the information in that file, as well. We can choose to use the `timestamp` in its provided format or we can convert it to POSIX format with the `lubridate` package. The POSIX format has the advantage of allowing us to use other `lubridate` tools for wrangling date and time objects and is more amenable to human readability.
We see the range of timestamps and the human-readable version with

```
range(train$timestamp)
```

```
## [1]  789652009 1231131736
```

```
as_datetime(range(train$timestamp))
```

```
## [1] "1995-01-09 11:46:49 UTC" "2009-01-05 05:02:16 UTC"
```

When we look at a histogram of `timestamp` we see that there are time periods of high rating activity and periods of low activity. This suggests there may be underlying structure to the `timestamp` variable.

`genres` When we look at the `genres` variable, we expect to find a pipe-separated character string with the possible values listed in the README file. The results of the following code show that our assumption does not hold. There are values in the dataset that are not in the `README` file, so we will need to extract the genres from the file rather than use the values provided by the dataset's README file.

Did the genres list provided in the README file match the dataset?

```
identical(genres_list, README_genres_list)
```

```
## [1] FALSE
```

The actual values found in the genres variable were

```
genres_list
```

```
##  [1] "(no genres listed)" "Action"             "Adventure"
##  [4] "Animation"          "Children"           "Comedy"
##  [7] "Crime"              "Documentary"        "Drama"
## [10] "Fantasy"            "Film-Noir"          "Horror"
## [13] "IMAX"               "Musical"            "Mystery"
## [16] "Romance"            "Sci-Fi"             "Thriller"
## [19] "War"                "Western"
```

Which values are present in our data that weren't in the README?

```
setdiff(genres_list, README_genres_list)
```

```
## [1] "(no genres listed)" "Children"           "IMAX"
```

We can further examine the information contained in the `genres` variable by creating binarized variables for each possible genre, essentially answering the question, "is this movie a `genre` movie?" for each possible genre, with a logical result, or TRUE/FALSE. We can include this binarizing code [9], along with our other wrangling tasks, in a function that can also be applied to the `test` and `validation` datasets.

```r
WrangleDF <- function(df){
  # extract title & year from the 'title (year)' format, place in separate columns
  df <- df %>% extract(data = . ,
                        col = title,
                        into = c("title", "year"),
                        regex = "(.+)[(](\\d+)[)]" )
  # convert timestamp to POSIX format and year to numeric
  df <- df %>% mutate(timestamp = lubridate::as_datetime(timestamp),
                       year = as.numeric(year))

  # create a binary column for each genre in our genres_list
  movie_genres <- setNames(
    data.frame( movie_genres$movieId,
                lapply(genres_list, function(i)
                  as.integer( grepl(i, movie_genres$genres))
                 )
    ),
    c("movieId",genres_list)
  )
  # Join our binarized genres list to the data frame
  df <- df %>% left_join(movie_genres, by = 'movieId')
  # Clean up
  rm(movie_genres)
  return(df)
}

# Convert the test and train sets
train <- WrangleDF(train)
head(train,3)
```

```
##    userId movieId rating           timestamp      title year
## 1:      1     122      5 1996-08-02 11:24:06 Boomerang  1992
## 2:      1     185      5 1996-08-02 10:58:45  Net, The  1995
## 3:      1     292      5 1996-08-02 10:57:01  Outbreak  1995
##                       genres (no genres listed) Action Adventure Animation
## 1:              Comedy|Romance                  0      0         0         0
## 2:        Action|Crime|Thriller                  0      1         0         0
## 3: Action|Drama|Sci-Fi|Thriller                  0      1         0         0
##    Children Comedy Crime Documentary Drama Fantasy Film-Noir Horror IMAX
## 1:        0      1     0           0     0       0         0      0    0
## 2:        0      0     1           0     0       0         0      0    0
## 3:        0      0     0           0     1       0         0      0    0
##    Musical Mystery Romance Sci-Fi Thriller War Western
## 1:       0       0       1      0        0   0       0
## 2:       0       0       0      0        1   0       0
## 3:       0       0       0      1        1   0       0
```

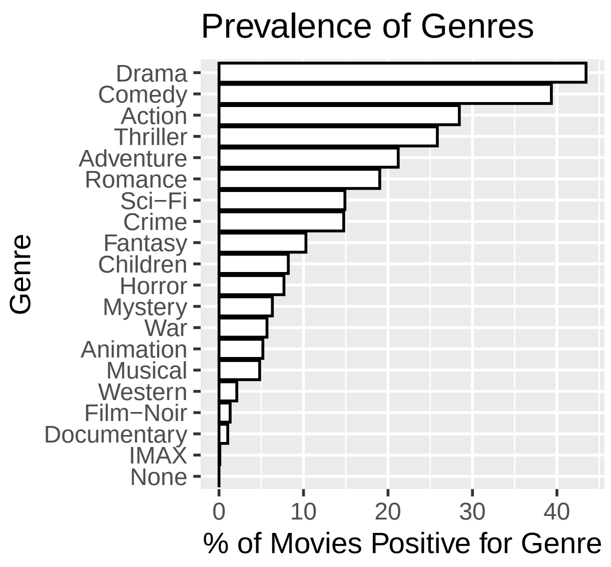
[9] Binarization algorithm informed by StackOverflow answer https://stackoverflow.com/questions/39461539/convert-column-with-pipe-delimited-data-into-dummy-variables

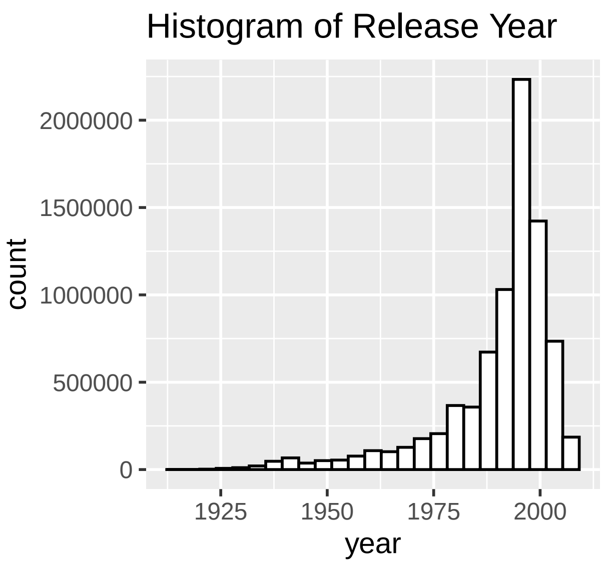
```r
test <- WrangleDF(test)

# We need to get rid of the "(no genres listed)" title. Those "()" will be
# problematic when we want to use the name as a column name in a dataframe.
train <- train %>% rename( None = "(no genres listed)")
test <- test %>% rename( None = "(no genres listed)")

# And let's keep the genres_list consistent
genres_list[genres_list == "(no genres listed)"] = "None"
```

Now we can examine each genre and we'll begin by just looking at the prevalence of each genre within the dataset, as shown below.





We see that `genres` is not uniformly distributed in the number of `ratings` for each genre, and that means we'll need to consider some form of regularization where the sample size is considered.

`year` Now that we've created a predictor from the release year, we can also examine this data.

```r
range(train$year)
```

```
## [1] 1915 2008
```

The figure above shows a majority of movies in this dataset were released in the 1990s. We will need to be aware of this underlying structure in the `year` variable if we choose to use it as a predictor.

Data Exploration: Sparsity

Now that we are familiar with the basic structure of our data, we can start to explore how the various available predictors are correlated with `rating`. We intuitively expect our `rating` to be an interaction of `movieId` and `userId` because we assume that the rating a user gives a movie is an indication of how much that user likes that movie. But examination of the interaction between `userId`, `movieId` and `rating` reveals that there is a great deal of sparsity, meaning that not all users rate all movies. In the figure below, we examine one hundred movies and one hundred users chosen at random. We introduce the `recommenderlab` package referenced by the course material and use its `image` function as an elegant way to display this type of sparse matrix. We note that even though we created a matrix with a random sample of dimension 100x100, our resultant matrix is smaller. This is because we happened to select user/movie combinations for which there were no ratings. The missing rows and columns in this randomly selected image highlight the sparsity of the data.

```r
# To use many of the recommenderlab functions, we need to convert our data to a
# realRatingMatrix format, ie a matrix with items as columns and users as rows

# Construct a matrix for these users and movies
# start by setting the seed since we'll use a random sample
suppressWarnings(set.seed(1234, sample.kind = "Rounding"))
y <- train %>% filter( movieId %in% sample( unique(movieId), 100) &
                         userId %in% sample( unique(userId), 100)) %>%
  select(userId, movieId, rating) %>%
  spread(movieId, rating) %>%
  as.matrix()
rownames(y) <- y[,1]  #colnames are already movieId
y <- y[,-1]   #trim off the userIds column

# Use the recommenderlab sparse matrix type with its image function
recommenderlab::image(as(y,'realRatingMatrix'))
```

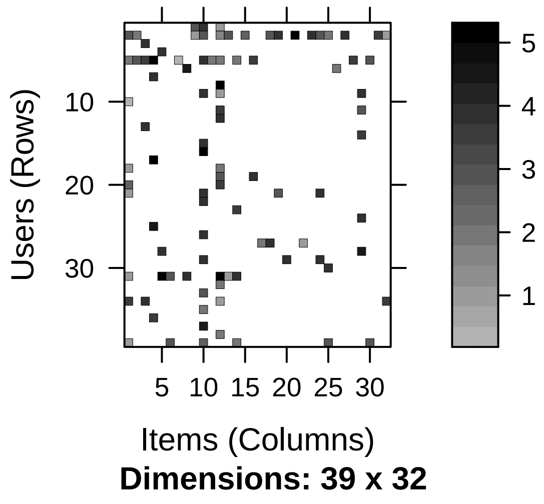


Figure 2: Image constructed from one hundred users and movies randomly sampled from the training dataset. There were entire rows and columns with no data because we didn't explicitly select movie and user combinations that were known to have ratings. It is possible to select a subset of users and movies from this datset such that the entire matrix is comprised of NAs.

```r
# Clean up
rm( y )
```

We might then ask how many of our users rated just one movie or how many movies received just one rating. We find that there were no users rating just a single movie, but there were several who rated just ten, and there were many movies with just a single rating. Further, we see from the figure below that there are many users with just a few ratings and many movies with just a few ratings.

Users with few ratings

```
train %>% group_by(userId) %>%
  summarize( n = n()) %>%
  filter(n<=10)
```

```
## # A tibble: 4 x 2
##   userId     n
##    <int> <int>
## 1     71    10
## 2  22170    10
## 3  29147    10
## 4  62516    10
```

Movies with just one rating

```
train %>% group_by(movieId) %>%
  summarize( n = n()) %>%
  filter(n==1) %>%
  nrow()
```

```
## [1] 152
```

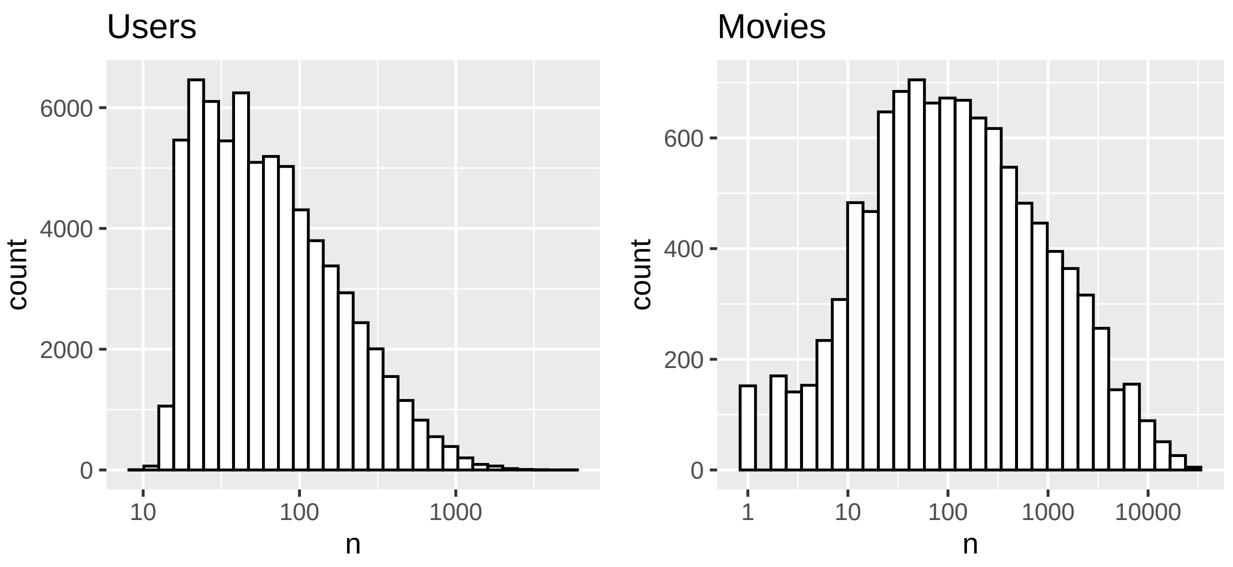


Figure 3: Histograms of the number of ratings for each movie and user, respectively.

First Model

Our loss function was defined as part of the problem statement and is the same one used by the Netflix Challenge. We will use the residual mean squared error (RMSE) and the function defined in the course materials to evaluate the RMSE. That definition of RMSE from the course material was for $y_{u,i}$ as the rating for movie i and user u and our predicted rating as $\hat{y}_{u,i}$ and N as the number of user/movie combinations within the ratings set. RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{u,i}\left(\hat{y}_{u,i} - y_{u,i}\right)^2}$$

The function in R was therefore defined in the following way, consistent with the course material:

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

Having defined this RMSE as the loss function, the natural choice for the simplest possible model, which would just be to predict a single value for all ratings in the dataset, is the mean of all those ratings. It is the natural choice because it is the single value that minimizes the loss. If we want to use a linear model that is the sum of the effects acting on rating, the mean is a good baseline. With just the mean as our prediction for all ratings and $Y_{u,i}$ is a random variable with error $\varepsilon_{u,i}$, which is also a random variable but is centered about 0:

$$Y_{u,i} = \mu + \varepsilon_{u,i}$$

Then our prediction $\hat{y}_{u,i}$ is just

$$\hat{y}_{u,i} = \hat{\mu}$$

and we estimate $\hat{\mu}$ as the global mean of ratings in our training set, $y_{u,i}$.

```r
  # The first model, made obvious by the choice of RMSE as loss function, just mu
  mu <- mean(train$rating)
  results <- tibble(method = "Just the average",
                    RMSE = RMSE(test$rating, mu))
  kable(results)
```

method	RMSE
Just the average	1.061135

Adding User and Movie Effects

The RMSE of 1.06 obtained with our simplest model was not a bad result for a single-valued prediction. If we follow the course material and the methods discussed by the “BellKor’s Pragmatic Chaos” team for the Netflix Prize, [10] [11] [12] we can try to improve our model by adding a movie effect and a user effect, b_i and b_u, respectively.

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

We can find a first estimate of b_i and b_u by approximating with the movie-wise or user-wise means of the residuals.

[10]https://netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf
[11]https://netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf
[12]https://netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf

$$\hat{b}_i = \frac{1}{N_i} \sum_u (y_{u,i} - \hat{\mu})$$

$$\hat{b}_u = \frac{1}{N_u} \sum_i (y_{u,i} - \hat{\mu} - \hat{b}_i)$$

We can also test whether the order of estimation of b_i and b_u affects the resultant RMSE.

```r
# Calculate the mean rating for each user based on the residual of rating
# less the overall mean rating
user_mus <- train %>%
  group_by(userId) %>%
  summarize( b_u = mean( rating - mu))

predicted_ratings <- mu + test %>%
  left_join(user_mus, by = 'userId') %>%
  pull(b_u)

  results <- results %>%
  add_row(method = "User Effect Only",
          RMSE = RMSE(test$rating, predicted_ratings))

#---------------------- Add Movie Effect ------------------------#
movie_mus <- train %>%
  left_join(user_mus, by = 'userId') %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu - b_u))

predicted_ratings <- test %>%
  left_join(user_mus, by = 'userId') %>%
  left_join(movie_mus, by = 'movieId') %>%
  mutate(prediction = mu + b_i + b_u) %>%
  pull(prediction)

results <- results %>%
  add_row( method = "User Then Movie Effect",
           RMSE = RMSE(test$rating, predicted_ratings))

#------- Does it matter if we reverse the order of Movie/User? ----------#

movie_mus <- train %>%
  group_by(movieId) %>%
  summarize( b_i = mean( rating - mu))

# Create a prediction based on the average rating for each movie and the
# overall average rating for all movies
movie_effect_predicted_ratings <- mu + test %>%
  left_join(movie_mus, by = 'movieId') %>%
  pull(b_i)

RMSE(test$rating, movie_effect_predicted_ratings)
```

```
## [1] 0.9441568
```

```r
results <- results %>% add_row(method = "Movie Effect Only",
                               RMSE = RMSE(test$rating, movie_effect_predicted_ratings))
# Clean up
rm( movie_effect_predicted_ratings)

#---------------------- Add in User Effect --------------------------#
user_mus <- train %>%
  left_join(movie_mus, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))

predicted_ratings <- test %>%
  left_join(movie_mus, by = 'movieId') %>%
  left_join(user_mus, by = 'userId') %>%
  mutate(prediction = mu + b_i + b_u) %>%
  pull(prediction)

results <- results %>% add_row( method = "Movie then User Effect",
                                RMSE = RMSE(test$rating, predicted_ratings))
kable(results)
```

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736

We see that the order of calculation of user and movie baseline effects does matter, with an improved result with calculating the movie effect first, which was the same order as the coursework.

Following the same analytical path of the coursework, we can examine the largest deviations from our predicted ratings, $\hat{y}_{u,i} = \mu + \hat{b}_i + \hat{b}_u$.

```r
# Do the largest errors in prediction on the test set correlate with having
# few ratings in the train set?

# Number of ratings per user
n_us <- train %>%
  left_join(user_mus, by = 'userId') %>%
  group_by(userId) %>%
  summarize( n_u = n())

# Number of ratings per movie
n_is <- train %>%
  left_join(movie_mus, by = 'movieId') %>%
  group_by(movieId) %>%
  summarize( n_i = n())

# Highest errors with associated samples sizes
test %>% left_join(movie_mus, by = 'movieId') %>%
  left_join(user_mus, by = 'userId') %>%
  left_join(n_us, by = 'userId') %>%
  left_join(n_is, by = 'movieId') %>%
  mutate( prediction = mu + b_i + b_u) %>%
  mutate( error = abs(rating - prediction) ) %>%
  arrange(desc(error)) %>%
  slice(1:10) %>%
  select(userId,movieId,rating,n_u,n_i,prediction,error)
```

```
##      userId movieId rating  n_u   n_i prediction    error
##  1:   24193    5952    0.5   95 11748  5.5380064 5.038006
##  2:   37651     922    0.5  130  2603  5.3595973 4.859597
##  3:   37651    3435    0.5  130  1917  5.3577573 4.857757
##  4:   10803    4398    5.0  410   124  0.2521858 4.747814
##  5:   52348   31410    0.5  204  1072  5.2288918 4.728892
##  6:   49808    2571    0.5   54 18815  5.1744525 4.674452
##  7:   32463    6724    0.5 1119   519  5.0317484 4.531748
##  8:   19243     608    0.5   30 19263  5.0286138 4.528614
##  9:   14269     858    0.5  191 16055  4.9725545 4.472554
## 10:   63818    4235    0.5   25  2054  4.9699382 4.469938
```

Interestingly enough, none of the movies with just one rating appear in our highest error list. In fact, our results don't really seem to justify the regularization step that both the "BellKor's Pragmatic Chaos" team and our coursework took. We can look at our data a bit differently and see how large the error was among our ratings with the lowest number of ratings per user or movie.

```
  # Smallest sample sizes with associated error
  test %>% left_join(movie_mus, by = 'movieId') %>%
    left_join(user_mus, by = 'userId') %>%
    left_join(n_us, by = 'userId') %>%
    left_join(n_is, by = 'movieId') %>%
    mutate( prediction = mu + b_i + b_u) %>%
    mutate( error = abs(rating - prediction) ) %>%
    filter( n_i <= 10) %>%
    group_by( n_i ) %>%
    summarize( avg = mean(error))
```

```
## # A tibble: 10 x 2
##      n_i   avg
##    <int> <dbl>
##  1     1 0.863
##  2     2 0.716
##  3     3 0.715
##  4     4 0.812
##  5     5 0.890
##  6     6 0.805
##  7     7 0.893
##  8     8 0.736
##  9     9 0.698
## 10    10 0.779
```

```r
# Clean up
rm( n_is, n_us)
```

We find that our average error for very low numbers of ratings per movie is only about 0.8, so the same justification for regularization does not apply to our data as it did to the smaller dataset used in the course material. We can go through the process anyway, however, to determine if there is an improvement in the loss function.

Regularization

The regularization scheme used in the BellKor solution and in the coursework penalized large values of b_u and b_i when the corresponding number of ratings was small. The coursework minimized $RMSE$ for both $\hat{b}_u$ and $\hat{b}_i$ simultaneously. We will decouple the user and movie effects, as seen in the following figure, in part because we will want to perform some regularization on other estimates for additional effects and it will become very cumbersome to try to regularize all of the terms simultaneously.

```r
lambdas <- seq(0.5,4,0.25)
regularized_rmses <- sapply(lambdas, function(L){
  movie_mus <- train %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n() + L))

  predicted_ratings <- test %>%
    left_join(movie_mus, by = 'movieId') %>%
    mutate(prediction = mu + b_i) %>%
    pull(prediction)

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

p1 <- data.frame(lambdas = lambdas, RMSE = regularized_rmses) %>%
  ggplot(aes(lambdas, RMSE)) +
  geom_point() +
  labs(title="Movie Effect, b_i", y="RMSE", x="lambda")

min(regularized_rmses)
```

```
## [1] 0.9441152
```

```r
L_i <- lambdas[which.min(regularized_rmses)]
L_i
```

```
## [1] 1.75
```

And we can go through the same process to regularize the user effects.

```
## [1] 0.8654905
```

Note that by using two different λ values, we effectively decouple the b_u and b_i optimizations. With both user and movie effect regularized, our $RMSE$ becomes

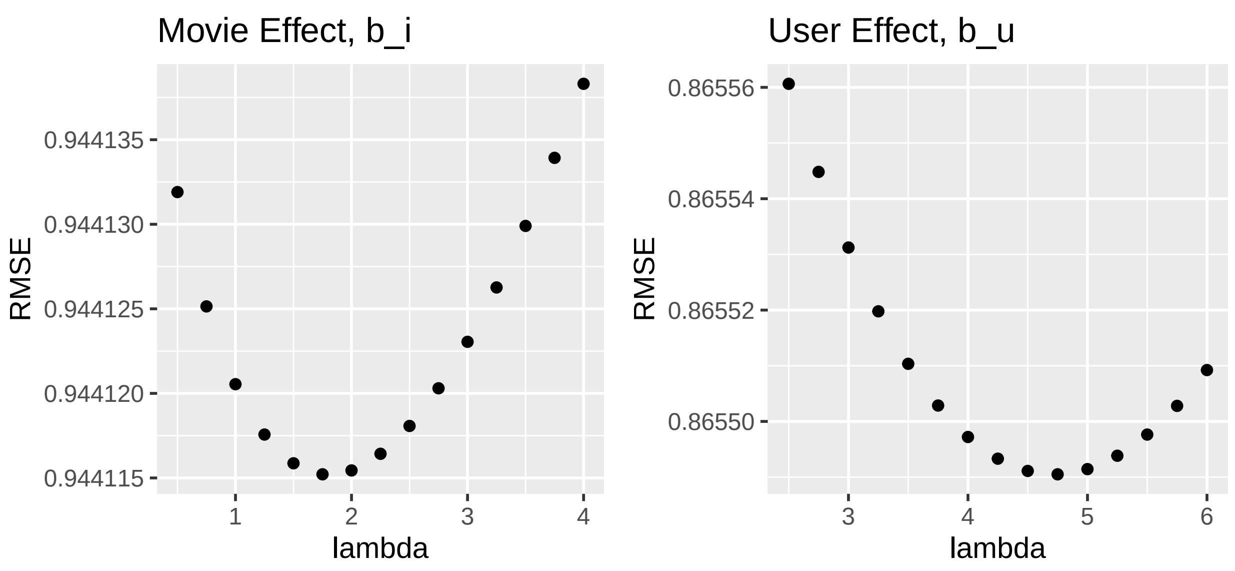


Figure 4: Regularization curves for movie effect and user effect.

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905

So, while regularization does not provide a massive improvement, it is still an improvement. At this point, rather than add our movie and user effects to our `train` and `test` sets, we can take this opportunity to examine methods to reduce accumulated error. Though the ranges of our individual effects are sensible

```
range(movie_mus$b_i)
```

```
## [1] -2.5919817  0.9614213
```

and

```
range(user_mus$b_u)
```

```
## [1] -2.649901  1.696972
```

when we add the two effects together, we start to get undesirable accumulation of error. Our estimated residual goes outside the bounds of the data we are trying to predict.

```
predicted_residual <- test %>%
  left_join(movie_mus, by = 'movieId') %>%
  left_join(user_mus, by = 'userId') %>%
  mutate(prediction = b_i + b_u) %>%
  pull(prediction)

range(predicted_residual)
```

```
## [1] -3.856062  2.334440
```

```
rm(predicted_residual)
```

Where the range of the residual should be between

Table 6: Variables in Sigmoid Transform

Effect	Description
A	Lower asymptote, default to residual
K	Upper asymptote, default to residual
B	Growth rate, or slope at $x = 0$
ν	Tunes slopes near asymptotes, radius of curvature
Q	Tunes origin. Found by setting $G(x = 0) = 0$ here

We will examine two methods for scaling our data to the desired range. The first method is to simply assign all values outside to the range to the nearest in-range value. The second method we'll explore is to apply a sigmoid function to our prediction, which both limits the bounds of the output and re-scales the predictions toward zero, essentially regressing our prediction to the mean. The second method is also trainable, as we can choose constants that minimize our loss function.

With the first clipping method, which we will call simple clipping and represent with a $\kappa()$, we simply find the values in our predicted effect, $\hat{b}_u$ or $\hat{b}_i$, such that values outside the range map to the nearest value within the allowable values for the data range. So, with $\hat{b}_u$, the clipping function would be

$$\kappa(\hat{b}_u)\Big|_{0.5-\mu}^{5-\mu} \mapsto \begin{cases} \hat{b}_u & 0.5-\mu < \hat{b}_u < 5-\mu \\ 5.0 & \hat{b}_u \geq 5.0-\mu \\ 0.5 & \hat{b}_u \leq 0.5-\mu \end{cases}$$

We can write a simple function in R to perform this task.

```
SimpleClip <- function(b,UB,LB){
  b[b > UB] <- UB
  b[b < LB] <- LB
  return(b)
}
```

The sigmoid clipping procedure takes the basic form of a generalized logistic function \footnote{Based on Generalised Logistic Function Article https://en.wikipedia.org/wiki/Generalised_logistic_function} and is similar to methods used by Simon Funk and by the "BellKor" team in the Netflix Challenge. The form is

$$G(x) = A + \frac{K - A}{(1 + Qe^{-Bx})^{1/v}}$$

With the variables explained in the table below and expressed in the following `R` function.

```
SigmaFunc <- function(x, B = 1, A = 0.5-mu, K = 5-mu, nu = 2){
   Q = (1 - (K/A))^nu - 1
   y <- A + (K - A)/(1 + Q *exp(-B*x))^(1/nu)
   return(y)
}
```

We take a look at the general performance of these two clipping functions below.

Then we can test our estimated error with each of the two clipping functions, beginning with the simpler transformation. Because we have chosen to decouple regularization of $\hat{b}_i$ and $\hat{b}_u$, we must re-calculate the regularization term λ_u after evaluating the clipping function for $\hat{b}_i$. The regularization code is given below, with results in the figure that follows.

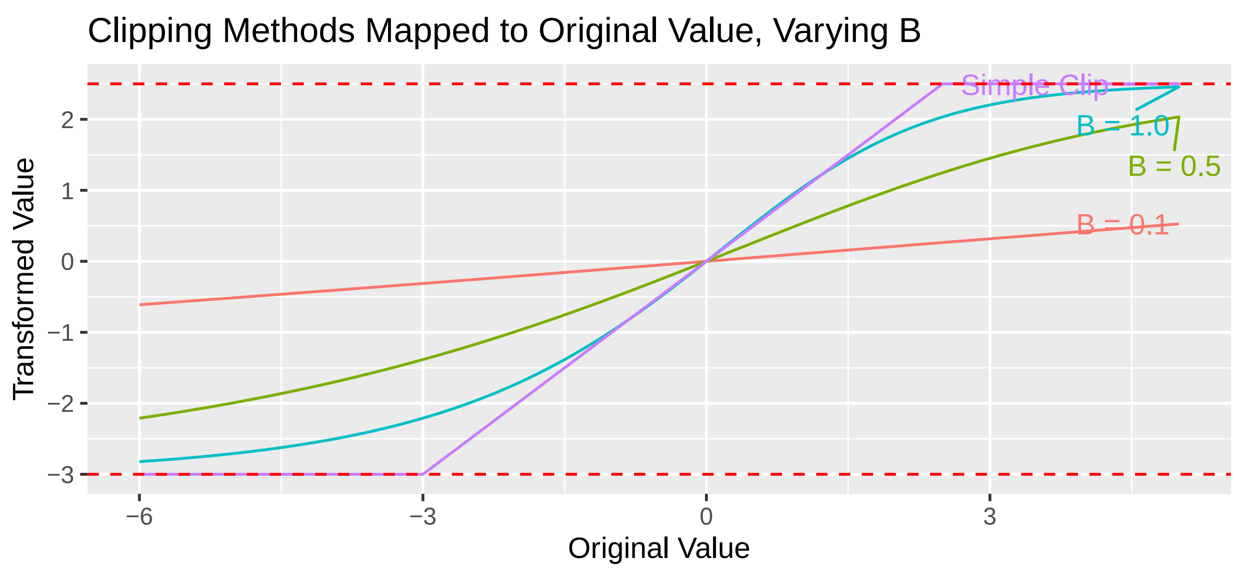


Figure 5: Comparison of clipping transformation functions. $B = 1$, $\nu = 1.75$ are good choices for our residual data and should perform in a way comparable to the simple clipping function.

```
lambdas <- seq(2.5,6,0.25)
regularized_rmses <- sapply(lambdas, function(L){
  UB <- 5.0 - mu
  LB <- 0.5 - mu
  movie_mus <- train %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n() + L_i))
  movie_mus <- movie_mus %>% mutate(b_i = SimpleClip(b_i, UB, LB))
  user_mus <- train %>%
    left_join(movie_mus, by = "movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n() + L))
  user_mus <- user_mus %>% mutate(b_u = SimpleClip(b_u, UB, LB))

  UB <- 5.0
  LB <- 0.5
  predicted_ratings <- test %>%
    left_join(movie_mus, by = 'movieId') %>%
    left_join(user_mus, by = 'userId') %>%
    mutate(prediction = SimpleClip(mu + b_i + b_u, UB, LB)) %>%
    pull(prediction)

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

p1 <- data.frame(lambdas = lambdas, RMSEs = regularized_rmses) %>%
        ggplot(aes(lambdas, RMSEs)) +
        geom_point() +
        labs(title="Simple Clipping")
```

The resulting loss $RMSE$ with the simple clipping procedure is then

```r
min(regularized_rmses)
```

```
## [1] 0.8653825
```

And the results of our analysis thus far

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825

With regularization terms

```r
L_u <- lambdas[which.min(regularized_rmses)]
L_u_simple <- L_u
```

When we compare the loss functions for unclipped versus clipped predictions, we see that our simple clipping procedure did provide an improvement in our estimate as measured by $RMSE$. Note that we not only clipped each estimated effect as it was calculated, we also clipped the final rating prediction.

We can train the output of the sigmoid function by changing the values of B and ν, but we have already found reasonable values for these parameters. The value of Q is determined by solving the boundary value problem where the input residual of 0 has output of 0 and then solving for Q in terms of K, A and ν.

We recalculate the $RMSE$ for user and movie effects with sigmoidal clipping:

```r
  lambdas <- seq(0,3,0.25)
  regularized_rmses <- sapply(lambdas, function(L){
    UB <- 5.0 - mu
    LB <- 0.5 - mu
    movie_mus <- train %>%
      group_by(movieId) %>%
      summarize(b_i = sum(rating - mu)/(n() + L_i))
    movie_mus <- movie_mus %>%
      mutate(b_i = SigmaFunc(b_i, A = LB, K = UB))
    user_mus <- train %>%
      left_join(movie_mus, by = "movieId") %>%
      group_by(userId) %>%
      summarize(b_u = sum(rating - b_i - mu)/(n() + L))
    user_mus <- user_mus %>%
      mutate(b_u = SigmaFunc(b_u, A = LB, K = UB))

    UB <- 5.0
    LB <- 0.5
    predicted_ratings <- test %>%
      left_join(movie_mus, by = 'movieId') %>%
      left_join(user_mus, by = 'userId') %>%
      mutate(prediction = SigmaFunc(mu + b_i + b_u, A = LB, K = UB)) %>%
      pull(prediction)

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

  p2 <- data.frame(lambdas=lambdas, RMSEs=regularized_rmses) %>%
          ggplot(aes(lambdas, RMSEs)) +
          geom_point() +
          labs(title="Sigmoid Clipping")
  gridExtra::grid.arrange(p1,p2,nrow=1)
```

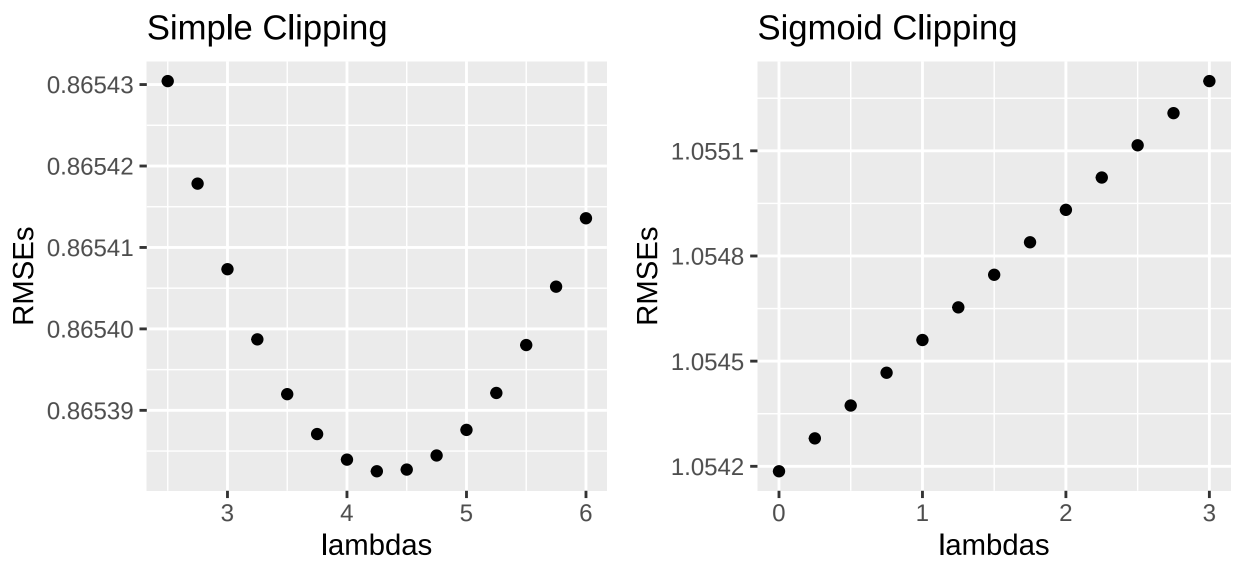


Figure 6: Regularization with simple clipping and sigmoid clipping on the user and movie effect model.

```
rm(p1,p2)
```

What we see is the attempted regularization of a non-linear function. Even so, we should have been able to obtain a better result with our initial values that were chosen from the basic form of the equation. We can consider whether it is appropriate to apply both regularization that punishes small sample sizes and simultaneously scale the output with our sigmoid function, but overall, we see that we simply were not able to achieve the results that were achieved with the simple clipping strategy. We could do more work on tuning the sigmoid clipping function, but we have achieved good results with the simple clipping method and we therefore relegate the sigmoid method to future work.

Our analysis results up to this point are now:

```
## [1] 1.054186
```

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201

method	RMSE
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825
Movie + User Effect, Regularized, Sigmoid Clipping	1.0541859

The best performance for our movie and user effects estimates, $\hat{b}_i$ and $\hat{b}_u$, were with simple averages, followed by regularization to penalize low rating counts, simple clipping to keep the estimates within the range of the `rating` variable, and a final clipping step performed on the linear sum of the estimates. This completes our training of the movie and user effects and we can add `b_u` and `b_i` to the `train` set. We also add the values to our `test` set to expedite calculations for other effects, but note that `test` is not used to train these estimates of the movie and user effects. In addition, we create a `residual` column in both `test` and `train` and this will be the value that we try to estimate as we go forward. The residual is just the rating less all of the effects we have calculated thus far, but we additionally apply our clipping function, $\kappa()$, where the upper and lower bounds are the same ones we used for training.

$$r_{u,i} = \kappa(y_{u,i} - \hat{\mu} - \hat{b}_i - \hat{b}_u)\Big|_{0.5-\mu}^{5-\mu}$$

Temporal Effects

The "BellKor" portion of the prize-winning Netflix Challenge team noted that temporal effects were a significant predictor of the baseline rating. We have two raw temporal predictors, `timestamp` and `year`. These allow us to examine release year, the day of rating, and the age of the movie at the day of rating for possible temporal effects. We can also potentially examine long-term drift in ratings by individual users, as well as seasonal or circadian patterns for ratings, as shown below.

It does appear that the year of release has an effect on ratings, but it also is correlated with the number of ratings. We see there are many fewer ratings for very early movies, and those ratings tend to be higher. There is also a dip in ratings that correlates with an increase in the number of movies rated, somewhere in the mid- to late-nineties. We found that most timestamps were also from the mid- to late-nineties, so we can examine how this effect differs when we look at the age of the movie at the time it was rated.

Age of Movie at Rating Effect

We can also examine how ratings change with the age of the movie. If we find the average ratings of all movies of the same age in the dataset, we get the plot below.

We see that there is an age effect, though it appears complex. There is a spike in both the number of ratings and their values at three years, and that would be consistent with new releases being rated more often and also more positively. The number of ratings tapers off as age increases, but the value of the rating oscillates. Perhaps there are cultural effects here, where movies experience changes in popularity due to complex social factors. The movies that were rated at a point in time farthest from the release date have the most variability in average rating and standard deviation. It might make sense to group all ages over 75 into one group if we want to tweak the model further.

What if we group this age effect by movie as well as by age. Do we observe an improved prediction when we estimate the rating based on the specific movie, as shown in the figure.

We see that not all movie ratings are dependent on age at the time of rating in the same way. Some ratings show a trend that decreases steadily with the age of the movie, while others have a flat trend in ratings over time and still others have increases in ratings over time. It might be possible to group movies based on the

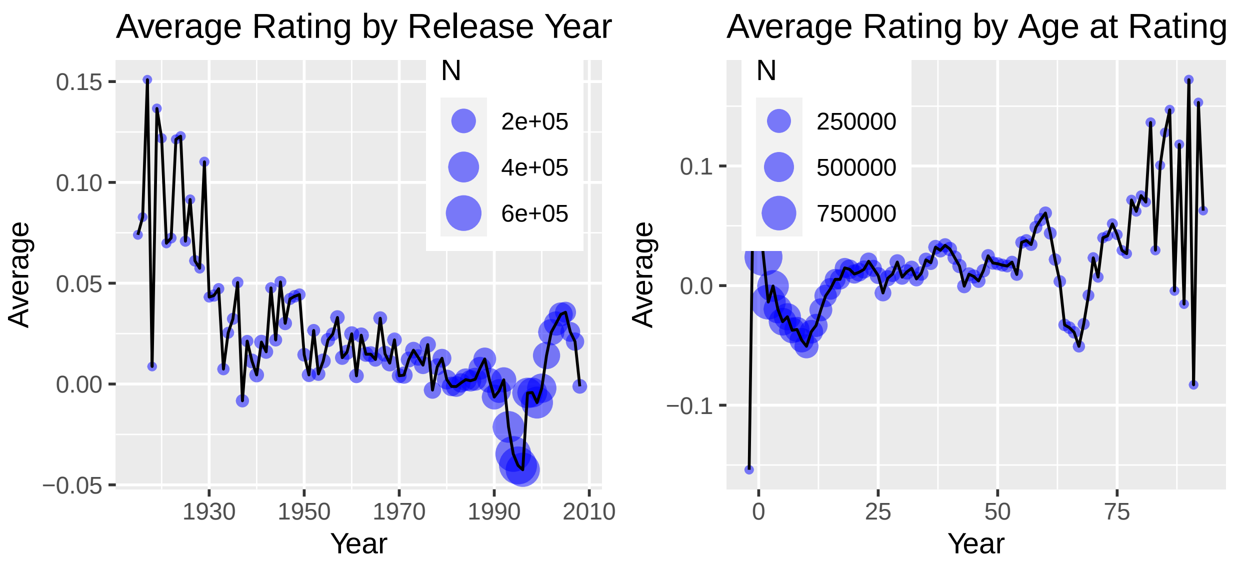


Figure 7: Average rating for movies by release year and by age of movie at the time of rating.

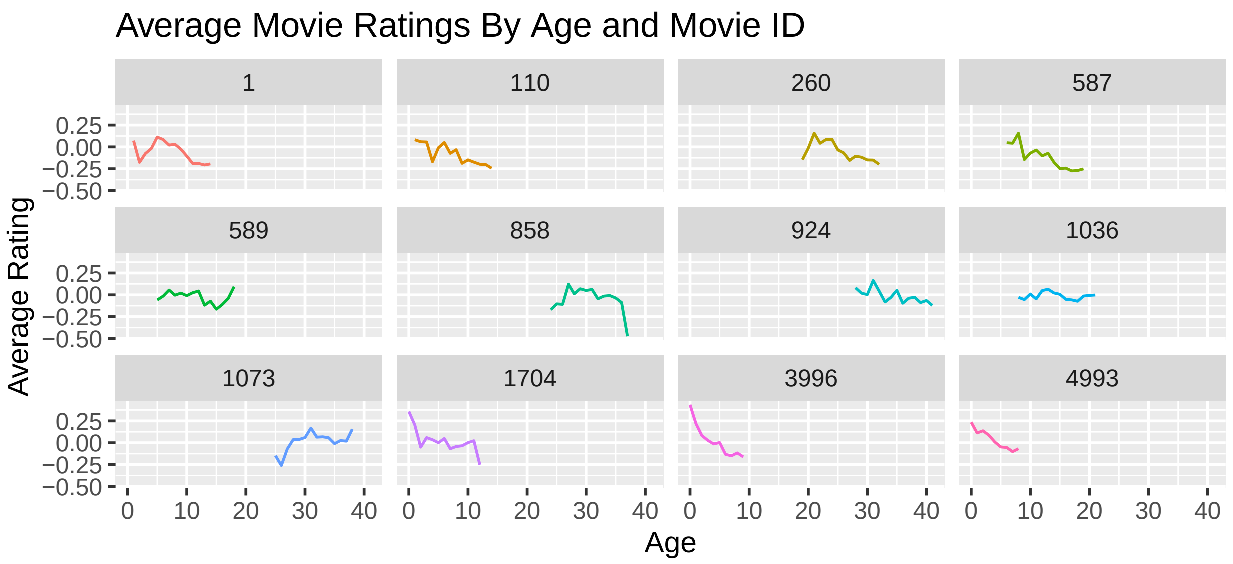


Figure 8: Rating averages grouped by both movie age and user.

shape of the age-rating curves, but that is beyond the scope of this analysis. Instead, we will simply calculate $\hat{b}_{i,t}$ based on both age and `movieId`, as follows. Note that we need to fill `NA` values with some value to be able to calculate $RMSE$. We use $NA \rightarrow 0$ with the understanding that a lack of ratings for a particular year might not be best estimated by this value. In order to fill `NA` values more intelligently, we would need to do some kind of regression to fill in missing values.

If we regularize the movie effect, as grouped by age and user, we get a regularization curve as seen below.

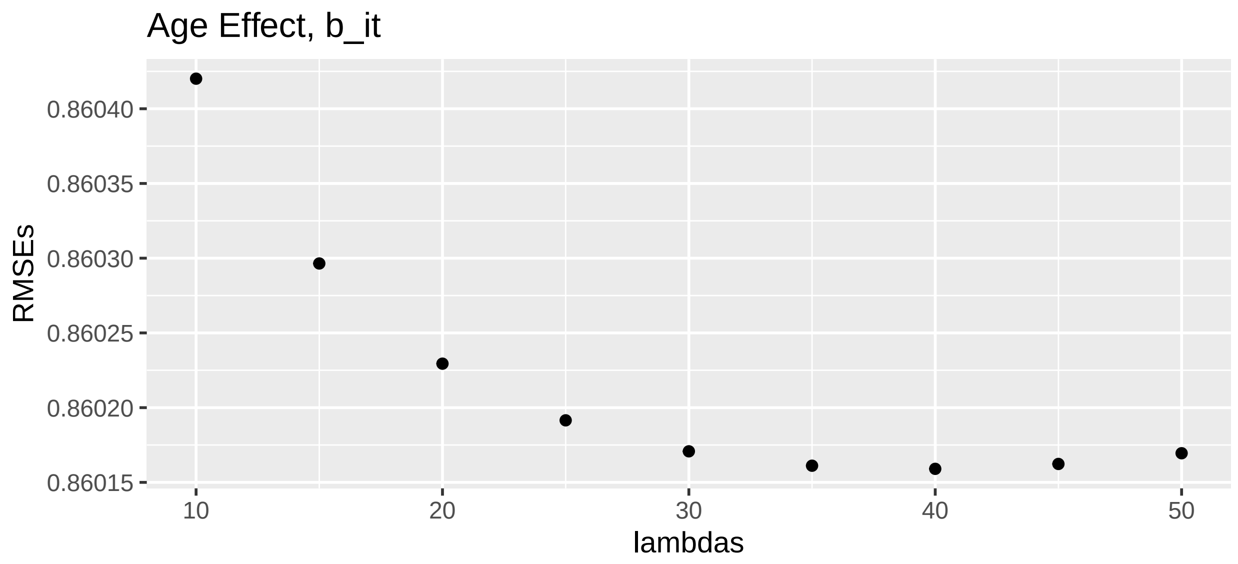


Figure 9: Regularization curve for movie age effect, grouped by user and age of movie at time of rating.

When we perform the same regularization task, but also include clipping, our `RMSE` is

```
## [1] 0.8601591
```

And our results up to this point in the analysis are

```r
# We add this temporal effect to the train set and re-calculate the residual
b_its <- train %>%
    mutate( age = year(timestamp) - year) %>%
    group_by(movieId,age) %>%  # movieId AND age, not just age
    summarize(b_it = sum(residual)/(n() + L_it))

train <- train %>%
  mutate( age = year(timestamp) - year) %>%
  left_join(b_its, by = c('movieId','age')) %>%
  mutate( residual = rating - mu - b_i - b_u - nafill(b_it, fill=0))

# And do the same with the test set
test <- test %>%
  mutate( age = year(timestamp) - year) %>%
  left_join(b_its, by = c('movieId','age')) %>%
  mutate( residual = rating - mu - b_i - b_u - nafill(b_it,fill=0))

kable(results)
```

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825
Movie + User Effect, Regularized, Sigmoid Clipping	1.0541859
Add Movie Age at Rating Effect	0.8650189
Add Movie Age at Rating Effect, Clipped	0.8650189
Age Effect by both Age and Movie	0.8621989
Age Effect by Age and Movie, Regularized	0.8601591
Age Effect by Age and Movie, Regularized, Clipped	0.8601591

We observe that predicting a rating based on the average age of the movie at the time of rating, with a different prediction for each movie at each one year age interval, has a large impact on our loss function. We also note that regularizing the effect requires a large lambda compared to our other regularization terms. This suggests that there must be more than forty ratings for a particular movie when it is a particular age for the average of those ratings to become predictive. We also note that clipping is either resulting in no change to our predictions or increasing our error. We will leave clipping out of the following estimates and re-evaluate its use on the final model.

Circadian Effect

Humans can be strongly influenced by their circadian rhythms. Time of day can affect many predictors of decision-making, from mood to impulse control. The average rating by time of day for all user is shown in the following figure.

The result suggests there is no predictive power in the hour at which a rating was made. It does tell us that few ratings were created in the mid-morning hours, but that isn't particularly surprising since that would correlate with school and work hours. The hour of rating does not allow ratings to be clustered when viewed as a communal effect. However, when viewed as a user effect, as in the following figure, where we first group our data by user and then by hour of the day, we see that there are large differences in individual user's rating behavior with respect to time of day. A similar effect was mentioned by the "BellKor" team in their explanation, though they looked more at spikes or rating frequency changes as a function of linear time. They postulated that these effects could be caused by multiple users generating ratings under a single `userId`. We could further postulate that it could also be a true circadian effect observed for a single user, in which some activity or biological state is correlated with negative ratings, such as lack of sleep or low blood sugar, while another state is correlated with positive ratings. Whatever the underlying cause, we see that putting all users into one bag would fail to leverage differences in rating patterns by time of day, so we explore how these differences can be added to our blend below.

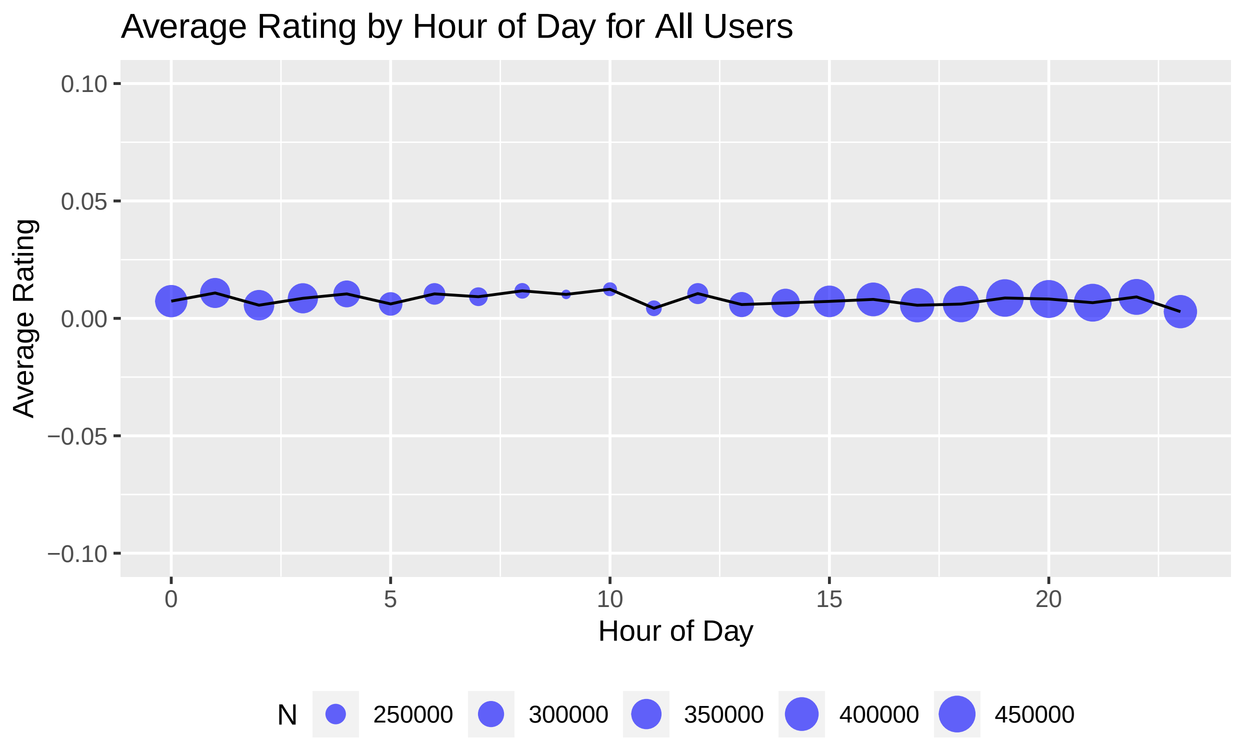
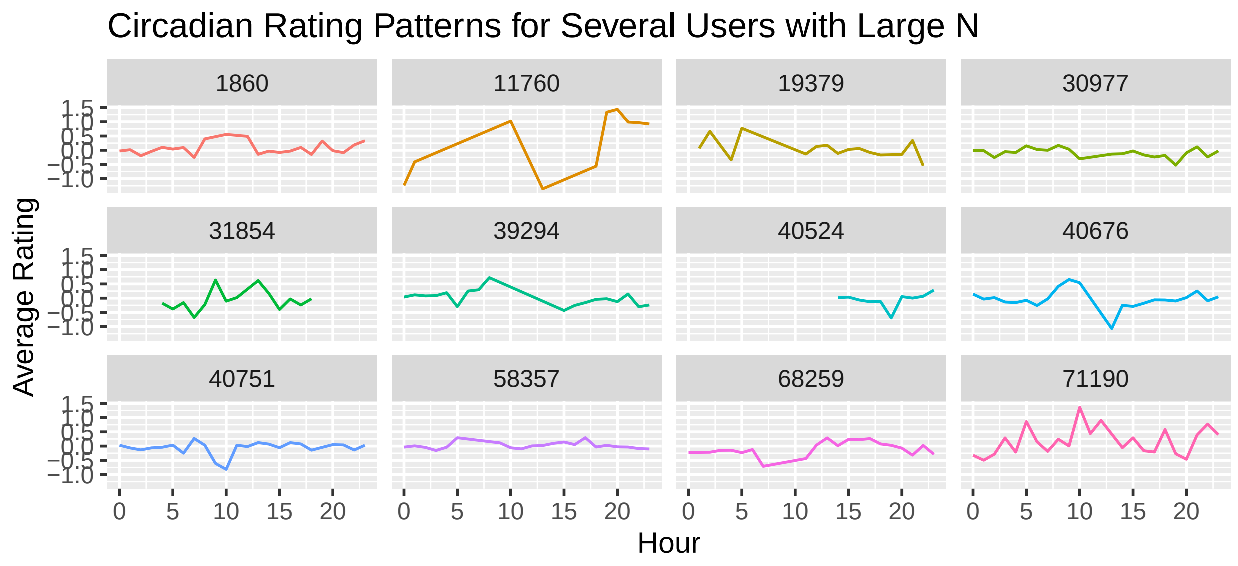


Figure 10: Average rating by time of day for all users. The total number of ratings at that hour of day is given by the point size. Note that the y axis only covers plus or minus 0.1 rating points. This suggests little predictive power in time of day.



We calculate a user effect based on hour of the day using just the average residual rating given by that user at that hour on any day.

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825
Movie + User Effect, Regularized, Sigmoid Clipping	1.0541859
Add Movie Age at Rating Effect	0.8650189
Add Movie Age at Rating Effect, Clipped	0.8650189
Age Effect by both Age and Movie	0.8621989
Age Effect by Age and Movie, Regularized	0.8601591
Age Effect by Age and Movie, Regularized, Clipped	0.8601591
Circadian Effect by Hour and User	0.8548503

This was a substantial improvement in RMSE. Does regularization make sense here? Are there hours of the day for certain users with so few ratings that the prediction should be penalized? Our data exploration suggests yes, so we should try regularization as seen below.

The regularized $RMSE$ for the circadian effect as grouped by user is

```
min(regularized_rmses)
```

```
## [1] 0.8502368
```

with training constant, $\lambda_{u,t}$

```
L_ut <- lambdas[which.min(regularized_rmses)]
L_ut
```

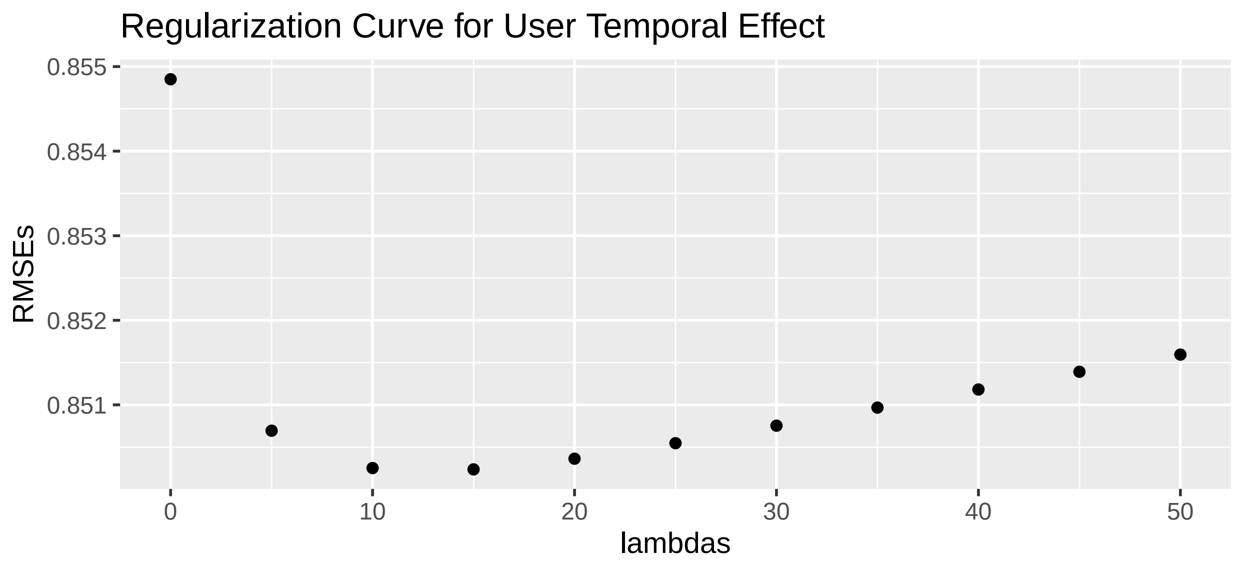


Figure 11: Regularization curve for circadian effect grouped by user.

```
## [1] 15
```

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825
Movie + User Effect, Regularized, Sigmoid Clipping	1.0541859
Add Movie Age at Rating Effect	0.8650189
Add Movie Age at Rating Effect, Clipped	0.8650189
Age Effect by both Age and Movie	0.8621989
Age Effect by Age and Movie, Regularized	0.8601591
Age Effect by Age and Movie, Regularized, Clipped	0.8601591
Circadian Effect by Hour and User	0.8548503
Circadian Effect by Hour and User, Regularized	0.8502368

We see that regularization has a large effect, so we re-calculate the circadian user effect based on the lambda we calculated and add this effect to the blend.

The "BellKor" team also found that rating behavior for each user tended to drift over time. If we look at a random sample of users, as shown below, we see that there are many users who were not very prolific and for whom a longer-term drift effect component is not especially useful. If we were to pursue this strategy, we would have to decide how to treat users without enough data to establish a trend. One potential strategy would be to combine all users to consider on overall trend.

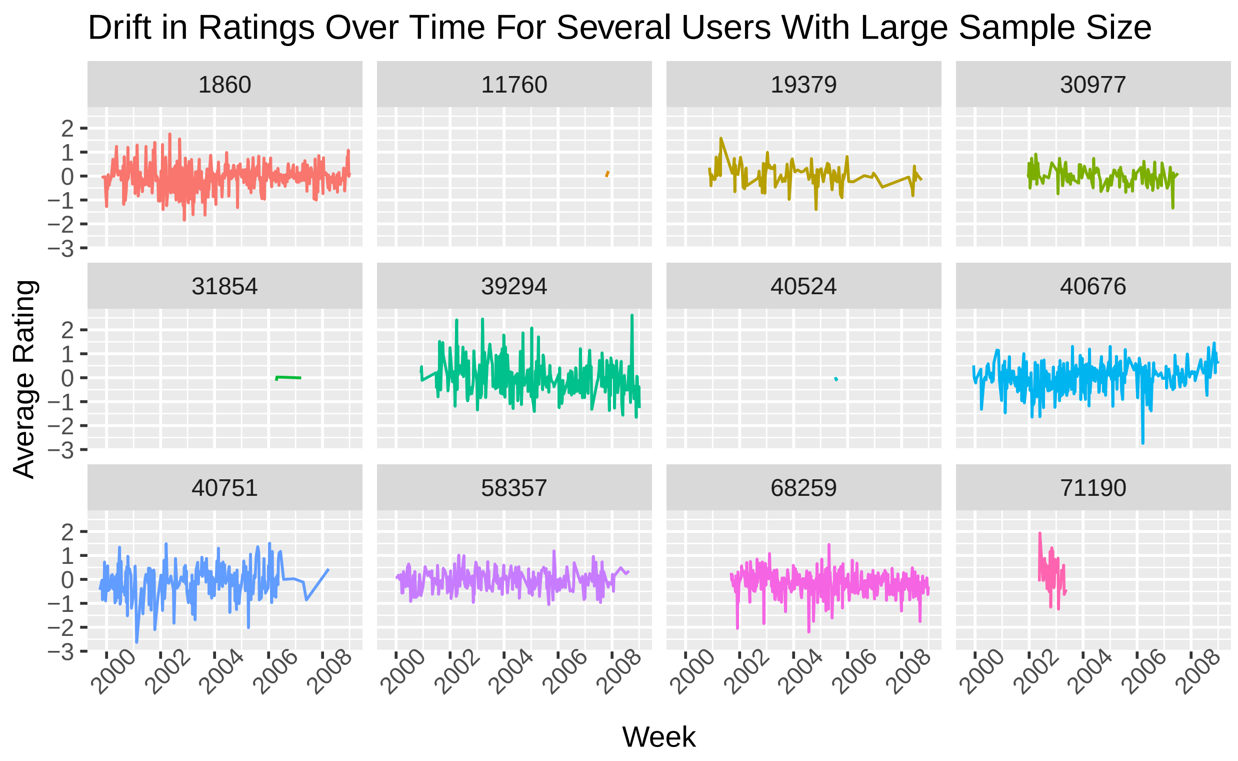


Figure 12: Drift in user rating over time, by week, for randomly sampled users.

We can observe a slight effect when we combine the data for five hundred randomly chosen users, as shown below. However, the effect is very slight, with a slope near zero. Most of the trend slopes for individual users in our plot are close to zero, as well. Additionally, we have already achieved approximately the same RMSE as the winning team for the Netflix Prize, and the computational cost of what we expect to be a very small incremental improvement begins to offer diminishing returns. We could continue to pursue these time-based effects on the user side as future refinements.

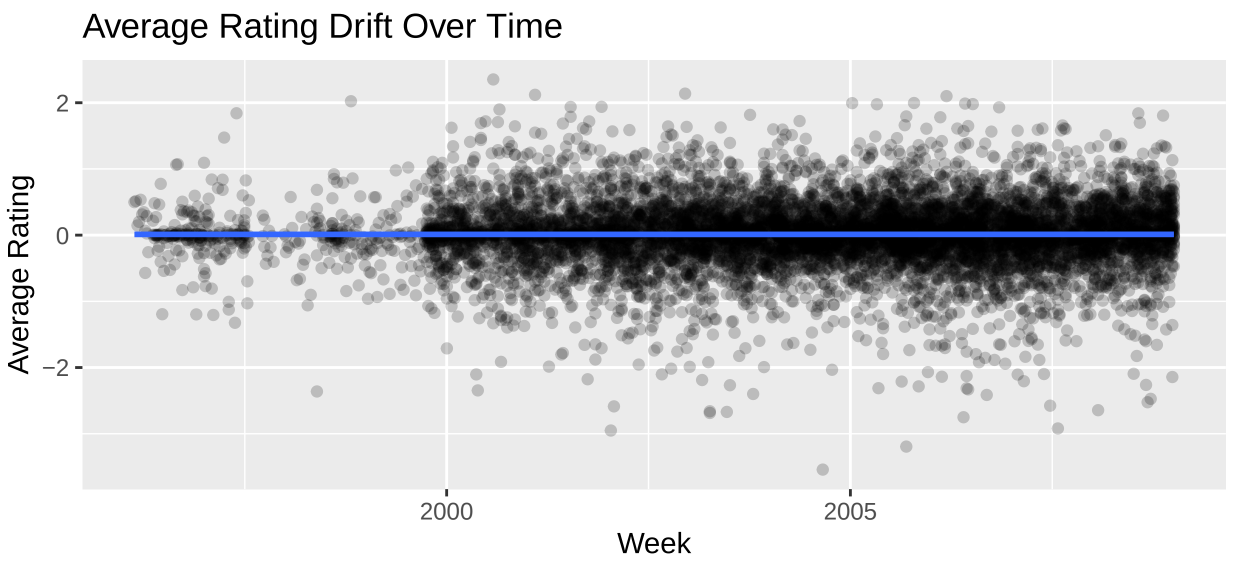


Figure 13: Cumulative drift for 500 randomly selected users.

Genre Effects

We improved the RMSE between our test and train sets to nearly the same overall value as the winning team for the Netflix Prize just by considering user effects that depended on user and a temporal variable and item effects that depended on item and a temporal variable. We can now begin to consider how users and items interact. Perhaps the most intuitive next step is to group movies into item groups based on information already provided to us, specifically, the `genres` field. We will also attempt to group users and individual items in a later section, but let's begin with the information already codified in the dataset.

Predictions with Raw `genres` Data

We can look at a random sample of very prolific users and examine how their average ratings change for the most popular genre groups, where each genre group is just the raw genres data treated as a factor and converted to a number. We can see that data represented in the figure below. We see that there is some similarity between user's rating patterns, but none have identical patterns. We are only looking at the first ten `genres` and representing more of the raw genre combinations becomes unreadable very quickly as there are more than seven hundred. We can reduce the number of dimensions just by splitting the genres into individual columns and binarizing the data, as we will do after we examine predictions using just the raw `genres` field.

In the following code chunks, we explore how we can improve our prediction using just the factorized raw genres field, along with regularization. We see that regularization provides a considerable improvement over the unregularized version, and this should not be surprising given that `genres` has more than seven hundred possible values and not all users will have rated even one movie for every genre, much less enough movies to provide real predictive power. For users with few ratings overall, there will be multiple `NA` values, and we can deal with those in a variety of ways. To begin, we predict those `genres` combinations with no ratings

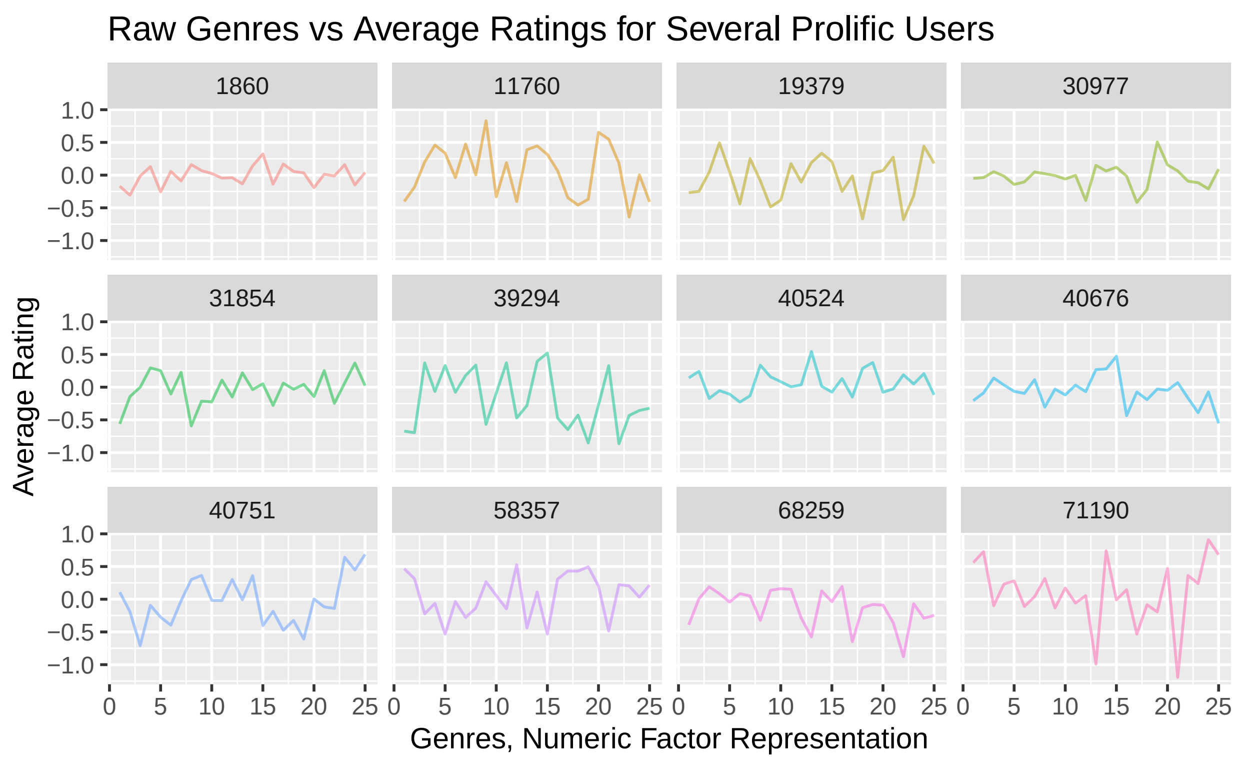


Figure 14: Raw genres as numeric factor vs. the average residual per user for a small random set of prolific users. Some clustering becomes apparent, but there are still far too many variables for this method to be human-readable.

in them as zeroes, essentially saying that no information contributes no change in the prediction. We might also consider replacing `NA` values for genres with a slightly negative value, say -0.1, to indicate that a user who has not rated any movies in a `genres` category is assumed not to like movies in that category. This could be a faulty assumption, however, as different users will have different strategies and reasoning when it comes to which movies they rate. Some users may choose to provide only negative ratings for movies they hate, for example, or may like movies in a given genre group reasonably well, but not well enough to make a concerted effort to rate movies in that genre. This could be an area for additional refinement.

We calculate the first user/genre effect estimate in the same way we have done for previous estimates, but we find that our $RMSE$ has not improved.

```
## [1] 0.9041875
```

We try regularizing as we have previously observed that there are many genre combinations for which users have given no ratings. We see that regularization does, indeed, help quite a bit in this case as seen in the regularization curve below.

```
lambdas <- seq(7,11,0.5)
regularized_rmses <- sapply(lambdas, function(L){
  b_ugs <- train %>% mutate( genres = as_factor(genres)) %>%
    group_by(userId, genres) %>%
    summarize(b_ug = sum(residual)/(n() + L))

  predicted_ratings <- test %>%
    mutate( age = year(timestamp) - year,
            hour = hour(timestamp),
            genres = as_factor(genres)) %>%
    left_join(b_ugs, by = c('userId', 'genres')) %>%
    mutate(prediction = mu + b_i + b_u +
             nafill(b_it, fill = 0) +   #fill NA with 0, ie no effect
             nafill(b_ut, fill = 0) +
             nafill(b_ug, fill = 0)) %>%
    pull(prediction)
  return(RMSE(test$rating, predicted_ratings))
})

data.frame(lambdas = lambdas, RMSEs = regularized_rmses) %>%
  ggplot(aes(lambdas, RMSEs)) +
  geom_point() +
  labs(title = "Regularization of Genre Effect, b_ug")
```

Our regularized minimum $RMSE$ was

```
min(regularized_rmses)
```

```
## [1] 0.843145
```

with regularization parameter

```
L_ug <- lambdas[which.min(regularized_rmses)]
```

We will explore another method for estimating the genre effect before we commit the predictions to our blend.

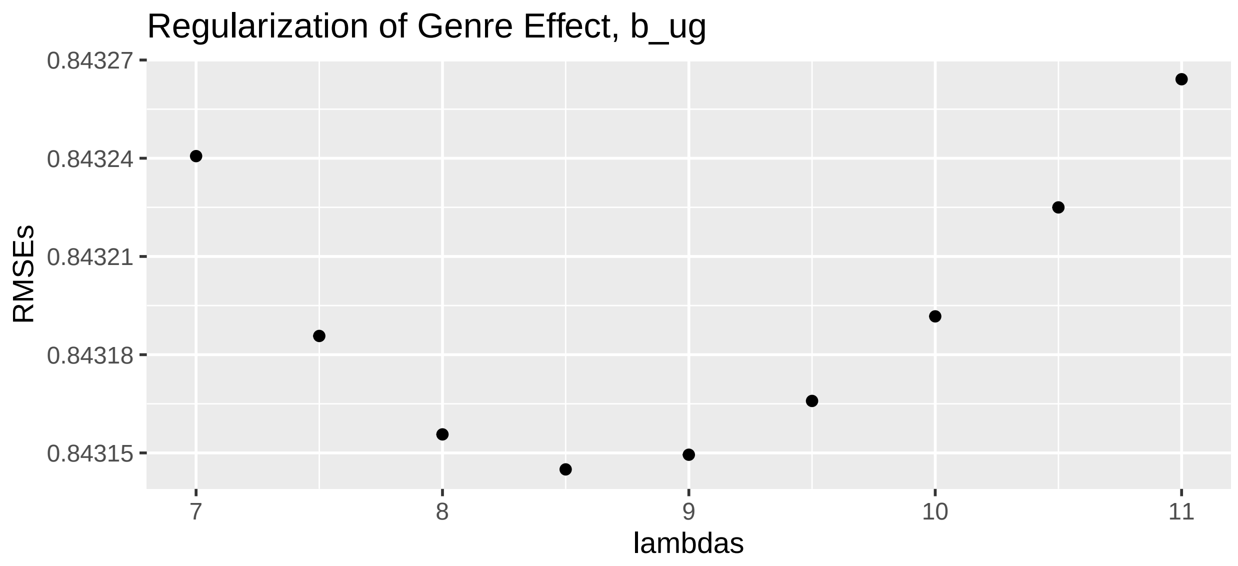


Figure 15: Regularization curve for genre effect based on factorized original genres string.

Predictions with Binarized `genres`

We created the binarized columns for `genres` in our Data Wrangling section. We can now begin to explore that data by looking at how the residual differs for each of the users in our randomly selected small group. We see in the following figure that for genres with many ratings by that user, the range within each genre is large, so even if we could predict the residual with just one genre, our error would still be large. The problem is that it doesn't really make sense to predict the residual with one genre when many of the items, or movies, in our dataset belong to more than one genre group, so the residual depends on multiple genre effects. That is why the model proposed in the coursework was a linear combination of the effects of the various genres. The big-data problem of trying to do a linear regression on millions of data points is still restrictive, just as it would be restrictive, in terms of time and computational resources, to do a direct linear regression for `rating` predicted by `userId` and `movieId`. One possible approach is to reduce the number of dimensions. We can see in our data that there is substantial correlation between genres for each user. For example, `Action` and `Adventure` have similar means and distributions for each of the `userIds` even though the distributions differ between users. So we see that we could both reduce the number of dimensions and then perhaps group users together, thereby reducing the size of a user/item matrix in both dimensions. A possible approach would be to do Principal Component Analysis to reduce the number of genres, followed by k-Means Clustering to reduce the number of users into user groups and finally to do a linear model predicting residual based on principal components of genre and user clusters.

```r
# Before doing anything complicated, how does an averaging approach improve RMSE? For each user/genre
dummy <- train[,None:Western] * train$residual
# Convert 0 to NA so we can easily remove those rows that don't apply
# to the genre in question
dummy[dummy==0] <- NA
dummy <- dummy %>% mutate(userId = train$userId)
# use dplyr::across to compute column means, removing NAs -
# this computes the user's mean residual rating for each genre
genre_avgs <- dummy %>% group_by(userId) %>%
  summarize(dplyr::across(.cols = None:Western,
                          .fns = ~ mean(.x, na.rm = TRUE)))
```

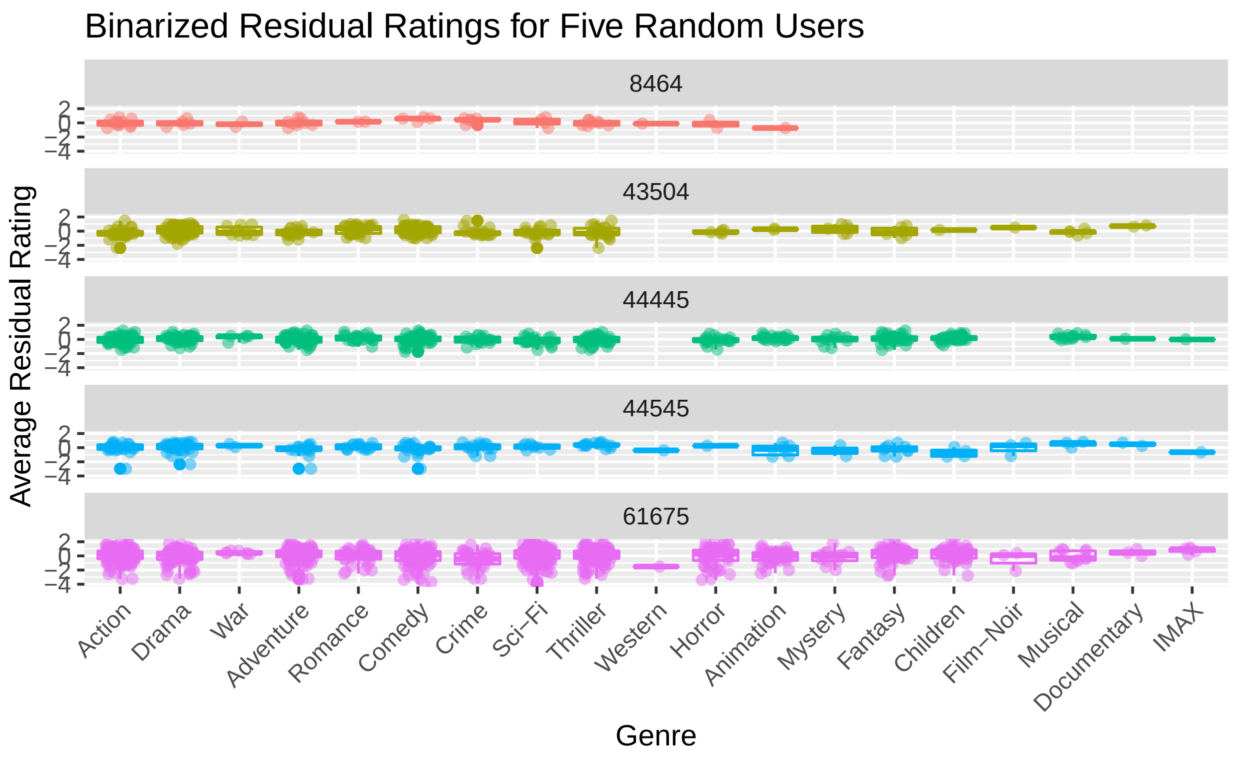


Figure 16: Average residual rating for each genre for five randomly selected users.

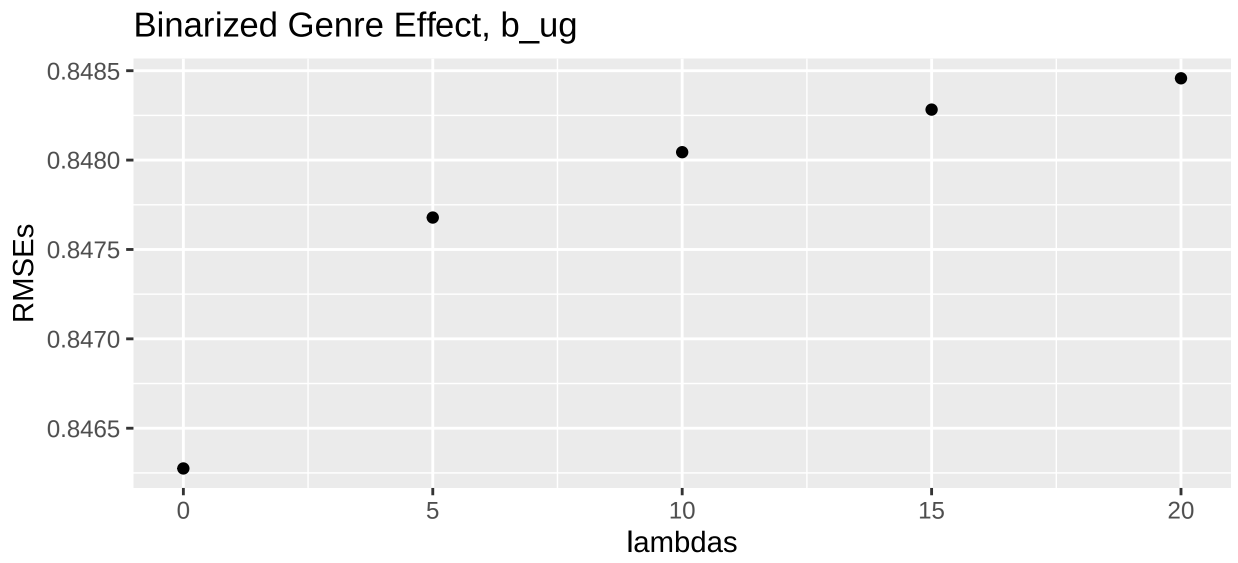
```
# Now estimate a residual for the test set based on userId and the genres
# of the movies rated
# Our best previous prediction for the ratings in test was:
predicted_ratings <- test %>%
  mutate(prediction = mu + b_i + b_u +
           nafill(b_it, fill = 0) +   #fill NA with 0, ie no effect
           nafill(b_ut, fill=0)) %>%
  pull(prediction)

#This gives unexpected result in terms of shape of regularized curve
prediction_genres <- test %>%
  left_join(., genre_avgs, by = 'userId', suffix = c("","_avg")) %>%
  mutate( pred = rowMeans(as.matrix(select(.,None:Western)) *
                  as.matrix(select(.,None_avg:Western_avg)),
                  na.rm = TRUE
                  )) %>%
  pull(pred)


# What is the RMSE for our best prediction plus the binarized genre
# effect?
results <- results %>% add_row(
  method = "Genre Effect, Binarized",
  RMSE = RMSE(test$rating, predicted_ratings + prediction_genres))
```

We see that binarizing the `genres` data does not improve our loss function. We should attempt to regularize this model, as well, because we saw that there were many `userId` and individual genre combinations with either few data points or no data.

We noted that some users have not rated any movies in a particular genre, and very few in others, therefore it might make sense to penalize estimates of these, or regularized on sample size as we have done previously as seen in the following figure. For future work, it might also make sense to try filling NAs with something other than 0, perhaps -0.5 or -1 on the assumption that if the user has not rated movies in that genre, they don't like that genre.



Our resultant $RMSE$ was

```
## [1] 0.846275
```

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825
Movie + User Effect, Regularized, Sigmoid Clipping	1.0541859
Add Movie Age at Rating Effect	0.8650189
Add Movie Age at Rating Effect, Clipped	0.8650189
Age Effect by both Age and Movie	0.8621989
Age Effect by Age and Movie, Regularized	0.8601591
Age Effect by Age and Movie, Regularized, Clipped	0.8601591
Circadian Effect by Hour and User	0.8548503
Circadian Effect by Hour and User, Regularized	0.8502368
Genre Effect, just Factors	0.9041875
Genre Effect, just Factors, Regularized	0.8431450
Genre Effect, Binarized	0.8462750
Genre Effect, Binarized, Regularized	0.8462750

We didn't improve our loss function with the binarization approach, but we might be able to use the format to create a decision tree or k nearest neighbor model of residual predicted by genres. We examine the heatmap of user vs. genre in the following figure.

```
# We can begin by asking which genres are not predictive, or have very little variance over the range
nzv <- nearZeroVar(
  sweep(train %>% select(None:Western), 1, train$residual, "*") )

# These correspond with the following genres
genres_list[nzv]
```

```
##  [1] "None"        "Animation"   "Children"    "Documentary" "Film-Noir"
##  [6] "Horror"      "IMAX"        "Musical"     "Mystery"     "War"
## [11] "Western"
```

```
# If we select a small, random group of users, we can see if filling in
# the missing values with one of the recommenderlab functions will
# help to improve our loss function
  # We start with a small version using a random set of users
suppressWarnings(set.seed(1234,sample.kind = "Rounding"))
# Randomly sample 100 users from the training set
users <- sample(train$userId, 100)

x <- genre_avgs %>%
  filter( userId %in% users) %>%
  as.matrix()
rownames(x) <- x[,1] #create rownames
x <- x[,-1]  #trim off userIds to leave just residuals
x[is.na(x)] <- 0  # heatmap has an na.rm = T option, but still errs

heatmap(x, col = RColorBrewer::brewer.pal(11,"RdBu"), na.rm = TRUE)
```

```
# Clean up
rm( nzv, genre_avgs )
```

We see that some relationships between groups of users and genres do exist. For example, at the lower right we see a group of users defined by a strong dislike of `Film-Noir`. Just above that group we have one that dislikes `Children` and `Animation` genres. User 71055 belongs to a group that likes `Documentary` and `Film-Noir` but mildly dislikes `Action`. It appears that the `Children` and `Animation` columns are positively correlated just based on a similarity of residual patterns.

Therefore perhaps a neighborhood or clustering approach would improve our model. If we try the following code to fit a k-Nearest Neighbor model, we find that we run into problems with too many ties in our data. This is due to the fact that we have integer, and worse, binary data, for all of our predictors. We can deal with this by either increasing the size of the neighborhood, or by adding very small random noise to our binary matrix data. We could also edit the source code for knn3, which throws the error for too many ties. We should consider this very carefully, however, to determine if it is appropriate or worthwhile. Another option is to fit the k-Nearest Neighbor model with the original `genres` data, but treat the list of genres as factors.

```
train_small <- train %>% filter(userId %in% users) %>%
  mutate(userId = as.factor(userId), genres = as.factor(genres)) %>%
```

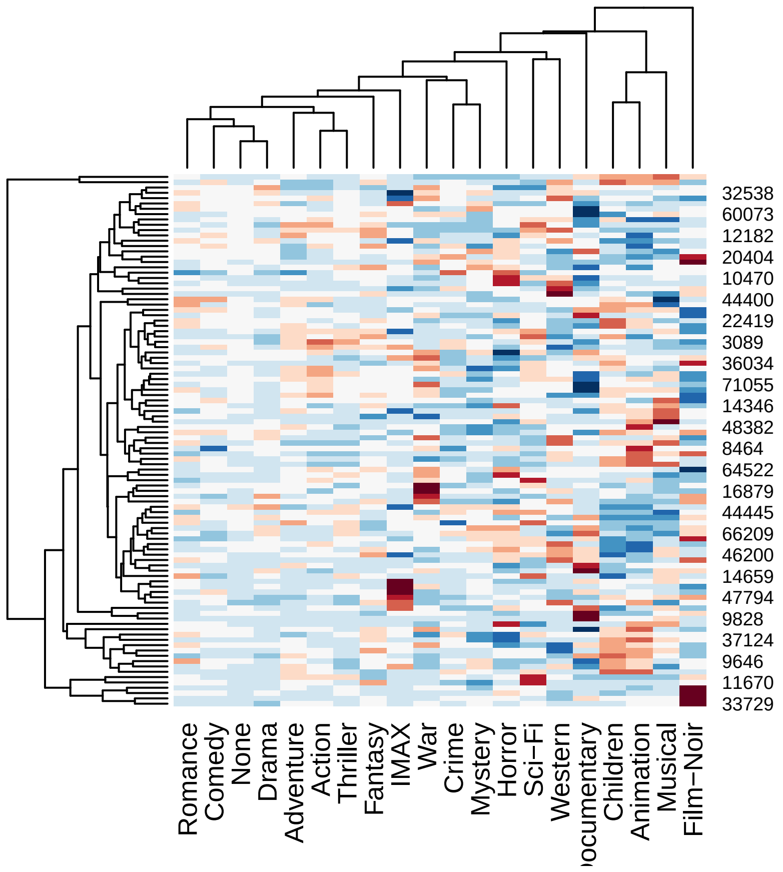


Figure 17: Heatmap of binarized genres average rating for several randomly chosen users. Binarized genres with near zero variance are removed and users and genres are clustered by distance. Red color indicates a negative residual and blue indicates a positive residual.

```
  select(userId, genres, residual)
test_small <- test %>% filter(userId %in% users) %>%
  mutate(userId = as.factor(userId), genres = as.factor(genres)) %>%
  select(userId, genres, residual)

# Note that just doing knn3(residual ~ ., data = train_small, K=100)
# results in an error for too many ties
knn_fit <- knn3(residual ~ ., data = train_small, k = 5)
y_hat <- predict(knn_fit, test_small, type = "class")
RMSE(test$residual, y_hat)
```

The above code is not evaluated because it results in an error. The new data in `test_small` contains levels that were not present in `train_small` set. We could go through the process of adding the missing levels as rows with `NA` or zero values for the `residual`, but this amounts to adding rows, and in fact data, to our original dataset. We must also consider that time spent wrangling data to allow us to fit a particular model is part of the overall return of the model. That is to say, we need to balance improved RMSE with compute time and development time. So rather than continuing to develop a k-Nearest Neighbor model, or other distance-based model such as Random Forest, we can explore the utility of matrix factorization methods for this application. Below, we introduce the RecommenderLab tools for this purpose.

Predicting Genre Interaction with Matrix Factorization

```
# Create an average ratings matrix for genres (original data) and users
x <- b_ugs %>% filter(userId %in% users) %>%
  spread(genres, b_ug) %>%
  as.matrix()
rownames(x) <- x[ ,1]
x <- x[ ,-1]

# We will want to try multiple methods from the recommenderlab, so
# create a function to handle the repetitive work
GenresCalcRecommenderRMSE <- function(x, Method){
  # Fit the model
  rec <- Recommender( as(x,'realRatingMatrix'), method = Method)

  # Make a prediction to fill in missing data
  predictions <- as( predict(rec, as(x,'realRatingMatrix'),
                             type = "ratingMatrix"),
                     "data.frame") %>%
    mutate(userId = as.numeric(.$user),  #format generic dataframe names
           genres = item ,
           residual_hat = rating) %>%
    select(userId, genres, residual_hat)

  # join the test data with the prediction data frame
  SVDrecs <- test %>% select(userId, genres, residual) %>%
    filter( userId %in% users) %>%
    left_join(., predictions, by = c('userId', 'genres')) %>%
    filter(!is.na(residual_hat))

  # And calculate the loss function, RMSE
  RMSE(SVDrecs$residual, SVDrecs$residual_hat)
}


# The available methods for real rating matrix types
methods = c('SVD','SVDF','UBCF','IBCF','ALS')

# See which method performs best on our small dataset
recommenderRMSEs <- sapply(methods, function(method)
  GenresCalcRecommenderRMSE(x, method))
recommenderRMSEs
```

```
##       SVD      SVDF      UBCF      IBCF       ALS
## 0.8282184 0.8016718 0.8418489 0.9908729 0.8072240
```

We can perform the same task as above, approximately, with `recommenderlab` evaluation tools. We create a larger dataset initially because the evaluation scheme controls test/train proportion and how many times we do a test.

```
  # Create an average ratings matrix for genres (original data) and users
suppressWarnings(set.seed(1234,sample.kind = "Rounding"))
# Randomly sample 100 users from the training set
users <- sample(train$userId, 1000)

# Build the realRatingMatrix of user vs genres filled with residual ratings
x <- b_ugs %>% filter(userId %in% users) %>%
  spread(genres, b_ug) %>%
  as.matrix()
rownames(x) <- x[ ,1]
x <- x[ ,-1]
x <- as(x, 'realRatingMatrix')

# Define a recommenderlab rating scheme
scheme <- evaluationScheme( x,
                            method = "split", # use cross-validation
                            train = 0.9, # 90 percent train
                            k = 2, # just 2 components to start
                            given = -5) # all but 5, use 5/user as test set

  # algorithms to train with parameters
algorithms <- list(
  "Random" = list(name = "RANDOM", param = NULL),
  "Popular" = list(name = "POPULAR", param = NULL),
  "ALS Latent Factors" = list(name = "ALS",
                              param = list(n_iterations = 5)),
  "User-Based CF" = list(name = "UBCF",param = list(nn=25)),
  "Item-Based CF" = list(name = "IBCF", param = list(k=30)),
  "SVD" = list(name = "SVD", param = list(k=10)),
  "Funk SVD" = list(name = "SVDF",
                    param = list(min_epochs = 10, max_epochs = 50))
)
```

Table 13: Training terms for SVDF in recommenderlab

Effect	Description
k	rank of the matrix/number of features/number of principal components
γ	regularization term that penalized large values
λ	learning rate
min epochs	min number of iterations per feature
max epochs	max number of iterations per feature

```r
# Evaluate the algorithms according to the scheme, use ratings
recommenderGenreResults <- evaluate(scheme, algorithms, type = "ratings")
```

```
## RANDOM run fold/sample [model time/prediction time]
##   1  [0.009sec/0.048sec]
##   2  [0.001sec/0.042sec]
## POPULAR run fold/sample [model time/prediction time]
##   1  [0.012sec/0.012sec]
##   2  [0.01sec/0.009sec]
## ALS run fold/sample [model time/prediction time]
##   1  [0.001sec/11.68sec]
##   2  [0.001sec/11.607sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [0.007sec/0.319sec]
##   2  [0.009sec/0.359sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [0.627sec/0.02sec]
##   2  [0.661sec/0.01sec]
## SVD run fold/sample [model time/prediction time]
##   1  [0.075sec/0.015sec]
##   2  [0.076sec/0.014sec]
## SVDF run fold/sample [model time/prediction time]
##   1  [11.663sec/1.393sec]
##   2  [11.509sec/1.731sec]
```

```r
# Do some cleanup
rm( algorithms, recommenderGenreResults, scheme )
```

We see that Funk SVD performed best, followed by ALS. Unfortunately, these were also the two most time-consuming methods. Part of the reason for this is that both SVDF and ALS allow for more control than just SVD, for example. For this reason, we will optimize the parameters with our smaller training set of randomly selected users.

In the code below, we tune the Funk SVD algorithm model of genre preference for each user. There are several parameters that we could tune with the `recommenderlab` SVDF method. These include

```r
# Tune the parameters for the genres Funk SVD matrix factorization
tuneGenreSVD <- sapply(c(2,7,12), function(K)
  sapply(c(0.01, 0.025, 0.05), function(Gamma){
    # Fit the model
    rec <- Recommender( x, method = 'SVDF',
                        list(k = K,
                             gamma = Gamma,
                                normalize = NULL))

      # Make a prediction to fill in missing data. User recommenderlab
      # predict function and output as data frame
      predictions <-
        as( predict(rec, as(x,'realRatingMatrix'), type = "ratingMatrix"),
                           "data.frame") %>%
        mutate(userId = as.numeric(.$user),  #format generic dataframe names
               genres = item ,
               residual_hat = rating) %>%
        select(userId, genres, residual_hat)

      # join the test data with the prediction data frame
      SVDrecs <- test %>% select(userId, genres, residual) %>%
        left_join(., predictions, by = c('userId', 'genres')) %>%
        filter(!is.na(residual_hat))

      # And calculate the loss function, RMSE
      return(RMSE(SVDrecs$residual, SVDrecs$residual_hat))
  }))

  # Make the results readable
  colnames(tuneGenreSVD) <- c("k = 2", "7", "12")
  rownames(tuneGenreSVD) <- c("G = 0.01", "0.025", "0.05")
  kable(tuneGenreSVD)
```

	k = 2	7	12
G = 0.01	0.8108463	0.8121000	0.8141530
0.025	0.8098001	0.8082782	0.8106512
0.05	0.8095262	0.8100664	0.8109699

```
  # Assign our trained k and gamma to use in final model
  kGenreSVDF <- 12
  GammaGenreSVDF <- 0.025

  #Clean up
  rm(tuneGenreSVD)
```

We see that our best results were achieved with the largest number of features trained, a number higher than the default of ten. Our best regularization was achieved with a value just slightly above the one mentioned in the Netflix Challenge Update by Simon Funk that is referenced in the `recommenderlab` manual [13] as well as just slightly above the default value. With this somewhat minimal training, we can calculate our final genre effect based on the entire set of user and genre pairs.

```
  # Re-calculate the user/genre interactions with the new k = 12, gamma = 0.025

  # Calculate new matrix for full set of user/genre pairs
  x <- b_ugs %>%
    spread(genres, b_ug) %>%
    as.matrix()
```

[13]https://sifter.org/ simon/journal/20061211.html

```
rownames(x) <- x[ ,1]
x <- x[ ,-1]
x <- as(x, 'realRatingMatrix')

# Fit the model
rec <- Recommender( x, method = 'SVDF',
                    list(k = kGenreSVDF,
                         gamma = GammaGenreSVDF,
                         verbose = FALSE,
                         normalize = NULL))

# Make a prediction to fill in missing user/genre interactions
b_ugs <- as( predict(rec, x, type = "ratingMatrix"), "data.frame") %>%
  mutate(userId = as.numeric(.$user),  #format generic dataframe names
         genres = item ,
         b_ug = rating) %>%
  select(userId, genres, b_ug)

# Add prediction elements to train set
train <- train %>%
    mutate( genres = as_factor(genres)) %>%
    left_join(b_ugs, by = c('userId', 'genres')) %>%
    mutate(residual = rating - b_i - b_u -
             nafill(b_it, fill = 0) -   #fill NA with 0, ie no effect
             nafill(b_ut, fill = 0) -
             nafill(b_ug, fill = 0))
# Add prediction elements to test set
test <- test %>%
  mutate( genres = as_factor(genres)) %>%
  left_join(b_ugs, by = c('userId', 'genres')) %>%
  mutate(residual = rating - b_i - b_u -
             nafill(b_it, fill = 0) -   #fill NA with 0, ie no effect
             nafill(b_ut, fill = 0) -
             nafill(b_ug, fill = 0))
# calculate prediction
predicted_ratings <- test %>%
  mutate(prediction = mu + b_i + b_u +
           nafill(b_it, fill = 0) +   #fill NA with 0, ie no effect
           nafill(b_ut, fill=0) +
           nafill(b_ug, fill = 0)) %>%
  pull(prediction)

RMSE(test$rating, predicted_ratings )
```

```
## [1] 0.8317858
```

```
## [1] 0.8312177
```

Movie User Interaction Effect

Having estimated many different baseline effects, and the user/genre effect, which is related to the user/movie effect but contains less information, we finally get to the core of the machine learning task presented to us in this capstone assignment. We will use `recommenderlab` to examine movie/user interactions. To start, we'll need to create a ratings matrix from the residuals. We start with a small version using a random set of users.

```r
suppressWarnings(set.seed(1234,sample.kind = "Rounding"))
# Randomly sample 1000 users from the training set
users <- sample(train$userId, 1000)

# Construct ratings matrix for training recommender models
x <- train %>% select(userId, movieId, residual) %>%
  filter( userId %in% users ) %>%
  spread(movieId, residual) %>%
  as.matrix()
rownames(x) <- x[,1] #create rownames
x <- x[,-1]  #trim off userIds to leave just residuals
x <- as(x, 'realRatingMatrix')

# We will want to try multiple methods from the recommenderlab, so
# create a function to handle the repetitive work
CalcRecommenderRMSE <- function(x, Method){
  # Fit the model
  rec <- Recommender( x, method = Method)

  # Make a prediction to fill in missing data
  predictions <- as( predict(rec, x, type = "ratingMatrix"), "data.frame") %>%
    mutate(userId = as.numeric(.$user),  #format generic dataframe names
           movieId = as.numeric(.$item),
           residual_hat = rating) %>%
    select(userId, movieId, residual_hat)

  # join the test data with the prediction data frame
  SVDrecs <- test %>% select(userId, movieId, residual) %>%
    filter( userId %in% users) %>%
    left_join(., predictions, by = c('userId', 'movieId')) %>%
    filter(!is.na(residual_hat))

  # And calculate the loss function, RMSE
  RMSE(SVDrecs$residual, SVDrecs$residual_hat)
}

# The available methods for real rating matrix types
methods = c('SVD','SVDF','UBCF','IBCF','ALS')

# See which method performs best on our small dataset
recommenderRMSEs <- sapply(methods, function(method)
  CalcRecommenderRMSE(x, method))
recommenderRMSEs
```

```
##       SVD      SVDF      UBCF      IBCF       ALS
## 0.7916637 0.7941569 1.0380360 0.9939977 0.8102157
```

```r
# Clean up
rm( methods, recommenderRMSEs )
```

We see that on the small subset of users, SVD had the best performance, and again SVDF and ALS perform similarly well. We will proceed with tuning the user and movie interaction SVD prediction in the same way that we tuned for user genre interaction, except we'll tune it on the entire `train` set. We will keep in mind that we might want to add in some of the features of the Funk SVD algorithm as it was the next best performer and we selected `SVD` based on its performance on a small subset of our `train` set.

```r
# Before doing the following calculation, we need to do some cleanup
# This calculation takes time and a lot of compute resources, so we need the
# minimum amount of extraneous data cluttering up the RAM
rm( b_its, b_ugs, b_uts, movie_mus, user_mus)

# Fill matrix with user vs movie residual rating and convert to recommenderlab
# type realRatingMatrix
x <- train %>% select(userId, movieId, residual) %>%
  spread(movieId, residual) %>%
  as.matrix()
rownames(x) <- x[,1] #create rownames
x <- x[,-1]  #trim off userIds to leave just residuals
x <- as( x, 'realRatingMatrix')

# Map the users and movies in test set to the train set index values
# the matrix factors are organized according to location in train set
uniq_userIds <- train %>% group_by(userId) %>%
  summarize(userId = first(userId)) %>%
  pull(userId)

uniq_movieIds <- train %>% group_by(movieId) %>%
  summarize(movieId = first(movieId)) %>%
  pull(movieId)

U_indexs <- data.frame( userId = sort(unique(test$userId)),
                       U_index = which(uniq_userIds %in% test$userId))

V_indexs <- data.frame( movieId = sort(unique(test$movieId)),
                       V_index = which(uniq_movieIds %in% test$movieId))

# Add the indexes to test set for easy access and organization
test <- test %>% left_join(U_indexs, by = 'userId') %>%
  left_join(V_indexs, by = 'movieId')

# Clean up
rm( uniq_userIds, uniq_movieIds, U_indexs, V_indexs )

# Train SVD model
rec <- Recommender( x,
                    method = 'SVD',
                    list(k = 85,
                         normalize = NULL,
                         verbose = FALSE))

#Do some cleanup, free up RAM before going on
rm(x)
```

We then calculate our prediction. Because of the size of the output matrix, we do this element-wise and only calculate a prediction for a user/movie pair found in the `test` set. The figure below shows the regularization curve used to minimize $RMSE$ by the number of matrix factor components, or k's.

```r
# Initialize a b_ui to build with a for loop, very un-R-like, but works
b_ui_SVD <- rep(0,10)

ks <- c(seq(10,80,10),85) # Values for number of components to use
rmses <- sapply( ks , function(k) {
  # Calculate b_ui element-wise
  for( i in 1:nrow(test)) b_ui_SVD[i] <-
      sum( rec@model$svd$u[test$U_index[i], 1:k ]  *
             rec@model$svd$d[1:k] *
             rec@model$svd$v[test$V_index[i], 1:k ])

  RMSE(test$residual,b_ui_SVD) # RMSE calculated for each k
})

# Plot the k that minimizes RMSE
qplot(ks,rmses, main="User Movie Interaction Components to Minimize RMSE")
```

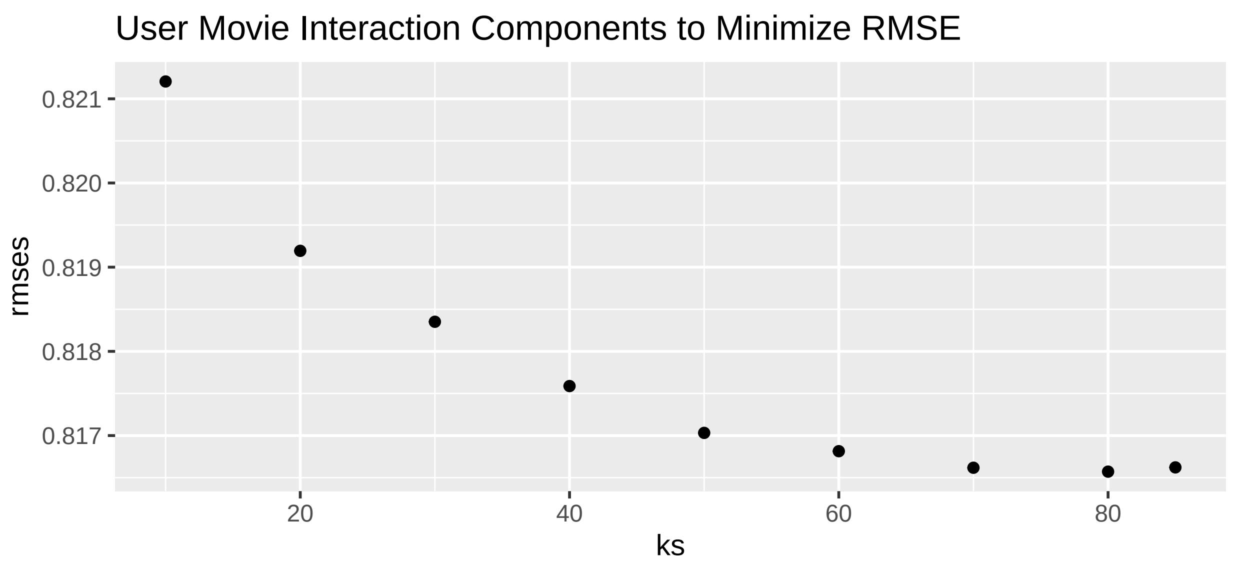


Figure 18: Regularization Curve for SVD number of matrix factor components to minimize RMSE.

```r
K_User_Movie_SVD <- ks[which.min(rmses)]  # store best k for final model
K_User_Movie_SVD
```

```
## [1] 80
```

```r
min(rmses)
```

```
## [1] 0.8165716
```

```r
rm(i, b_ui_SVD)

# Add it to the results
results <- results %>%
  add_row(method = "User/movie SVD",
        RMSE = min(rmses))

# Make a nice table
knitr::kable(results)
```

method	RMSE
Just the average	1.0611350
User Effect Only	0.9795916
User Then Movie Effect	0.8826201
Movie Effect Only	0.9441568
Movie then User Effect	0.8659736
Movie + User Effect, Regularized	0.8654905
Movie + User Effect, Regularized, Simple Clipping	0.8653825
Movie + User Effect, Regularized, Sigmoid Clipping	1.0541859
Add Movie Age at Rating Effect	0.8650189
Add Movie Age at Rating Effect, Clipped	0.8650189
Age Effect by both Age and Movie	0.8621989
Age Effect by Age and Movie, Regularized	0.8601591
Age Effect by Age and Movie, Regularized, Clipped	0.8601591
Circadian Effect by Hour and User	0.8548503
Circadian Effect by Hour and User, Regularized	0.8502368
Genre Effect, just Factors	0.9041875
Genre Effect, just Factors, Regularized	0.8431450
Genre Effect, Binarized	0.8462750
Genre Effect, Binarized, Regularized	0.8462750
Genre, Just Factors, Regularized, SVDF	0.8317858
Genre, Just Factors, Regularized, SVDF, clipped	0.8312177
User/movie SVD	0.8165716

Train the Final Model

What Is in the Final Model

The blend of effects we will use to train the final model includes the following effects:

So the final model is a linear combination of the effects as in the following equation:

$$\hat{y} = \hat{\mu} + \hat{b}_i + \hat{b}_u + \hat{b}_{i,t} + \hat{b}_{u,t} + \hat{b}_{u,g} + \hat{b}_{u,i}$$

where the hats indicate our estimate of the actual rating and effect components, and as we have done in previous sections, user is indicated with a u subscript and movie is indicated with an i subscript. The t subscript is used to indicate temporal effects, but the two time-dependent effects are not on the same time-scale. For $\hat{b}_{i,t}$ the scale of the time variable is in years since release, while for $\hat{b}_{u,t}$ the time variable is in hour of the day based on a twenty four hour clock.

Table 16: Effects Included in the Final Model

Effect	Description
μ	The overall average rating
b_i	Average rating for each movie
b_u	Average rating for each user
$b_{i,t}$	Average rating by movie and age in years
$b_{u,t}$	Average rating by user and hour of day
$b_{u,g}$	User-genre interaction by matrix factorization
$b_{i,u}$	User-movie interaction by matrix factorization

The final equations for each effect are given below.

$$\hat{\mu} = \frac{1}{N_y} \sum y$$

$$\hat{b}_i = \frac{1}{n_i + \lambda_i} \sum_{u=1}^{n_i} (y_{u,i} - \hat{\mu})$$

$$\hat{b}_u = \frac{1}{n_u + \lambda_u} \sum_{i=1}^{n_u} (y_{u,i} - \hat{\mu} - \hat{b}_i)$$

$$\hat{b}_{i,t} = \frac{1}{n_{i,t} + \lambda_{i,t}} \sum_{u=1}^{n_{i,t}} (y_{u,i,t} - \hat{\mu} - \hat{b}_i - \hat{b}_u)$$

$$\hat{b}_{u,t} = \frac{1}{n_{u,t} + \lambda_{u,t}} \sum_{i=1}^{n_{u,t}} (y_{u,i,t} - \hat{\mu} - \hat{b}_i - \hat{b}_u - \hat{b}_{i,t})$$

We then define a residual, $\hat{r}_{u,i}$ as

$$\hat{r}_{u,i,t} = y_{u,i,t} - \hat{\mu} - \hat{b}_i - \hat{b}_u - \hat{b}_{i,t} - \hat{b}_{u,t}$$

And we assign each `genre` in the list a number from one to the number of genre combinations in the training set. We know that we will include all possible `genre` combinations for the `test` set because we chose our test set such that all `userIds` and `movieIds` from the test set were represented in the training set, and the `genre` combination has a 1:1 correlation with `movieId`. In other words, all movies with the same `movieId` have the same `genres` combination. This residual was calculated from the residual in the same manner as our other baseline predictors, then regularized, and the result was decomposed into its matrix components with the Funk variant of Singular Value Decomposition (SVD).

$$\hat{b}_{u,g} \approx p_u q_i$$

where the values of the matrix factors are further approximated as

$$\hat{b}_{u,g} = \sum_{k=1}^{K} p_{u,k} q_{i,k}$$

We choose the number of singular vector pairs to use, or the value of K, through training, minimizing $RMSE$.

We then re-calculate the residual to be

$$\hat{r}_{u,i,t} = y_{u,i,t} - \hat{\mu} - \hat{b}_i - \hat{b}_u - \hat{b}_{i,t} - \hat{b}_{u,t} - \hat{b}_{u,g}$$

Table 17: Final Model Parameters for Regularized Elements

Parameter	Value
L_i	2.35
L_u	4.90
L_it	45.00
L_ut	14.00
L_ug	8.70

and approximate the interaction between users and movies, or $\hat{b}_{u,i}$ with a second set of decomposed vector pairs. This time our training indicated a better result with just the SVD method rather than the more complex Funk variant, so we get

$$\hat{b}_{u,i} = \sum_{j=1}^{J} p_{u,j} q_{i,j}$$

The final prediction was made by applying the clipping function, $\kappa()$.

$$\hat{y}_{u,i,t} = \kappa\Big|_{0.5}^{5.0}(\hat{\mu} + \hat{b}_i + \hat{b}_u + \hat{b}_{i,t} + \hat{b}_{u,t} + \hat{b}_{u,g} + \hat{b}_{u,i})$$

Training the Model

We deconstructed the `edx` set to do our exploration and initial training, so we now need to reconstitute it.

```
# Re-construct edx after our data exploration and model building
test <- test %>% select(userId, movieId, rating, timestamp, title, year,
                        genres, age, hour)
train <- train %>% select(userId, movieId, rating, timestamp, title, year,
                        genres, age, hour)
edx <- full_join(test, train,
                 by = c('userId', 'movieId', 'rating', 'timestamp', 'title',
                        'year', 'genres', 'age', 'hour'))

rm(test,train)
```

Then we can begin to calculate the various elements of the blend, starting with the estimated mean, $\hat{\mu}$. For all of the training parameters that are not based on matrix factorization, we will use cross-validation with the `edx` train:test ratio at 9:1.

```
# Re-set the seed before we begin the final training
set.seed(1234, sample.kind="Rounding")

# Cross-validate regularized elements B times
B <- 5
cv_results <- replicate(B, TrainFullModel() )

# transform the cross-validation results into usable data frame
cv_results <- as.data.frame(t(matrix(unlist(cv_results), ncol=B)))
```

We use the parameters trained on the full `edx` set to create the individual components of the final prediction and create a `residual` for the `edx` set.

We have now trained all parameters of the model up to the matrix factorization models. The matrix factorization models are too computationally intensive to perform multiple times, though we would likely improve our results with cross-validation. However, with limitations in our computing resources, we will use the previously trained parameters for the final, full `edx` model.

```r
# Calculate regularized average rating per user, per genre combination
# as residual of previously calculated effects
b_ugs <- edx %>%
    mutate( genres = as_factor((genres))) %>%
    group_by(userId, genres) %>%
    summarize(b_ug = sum(residual)/(n() + L_ug))

# Construct realRatingMatrix of user vs factorized genre combinations
# filled with residual average rating for that genre combo
x <- b_ugs %>%
  spread(genres, b_ug) %>%
  as.matrix()
rownames(x) <- x[ ,1]
x <- x[ ,-1]
x_pred <- x[ rownames(x) %in% validation$userId , ]
x <- as(x, 'realRatingMatrix')
x_pred <- as(x_pred, 'realRatingMatrix')

# Calculate the SVDF model for genre/user interaction
rec <- Recommender( x, method = 'SVDF',
                    list(k = kGenreSVDF,
                       gamma = GammaGenreSVDF,
                       verbose = FALSE,
                       normalize = NULL))



# Make a prediction to fill in missing data
b_ugs <- as( predict(rec, x_pred, type = "ratingMatrix"), "data.frame") %>%
  mutate(userId = as.numeric(.$user),  #format generic dataframe names
         genres = item ,
         b_ug = rating) %>%
  select(userId, genres, b_ug)

# Add the predicted b_ug to edx and validation sets and recalculate residual
# in the edx set - residual is input to movie/user SVD below
edx <- edx %>%
  mutate(genres = as_factor(genres)) %>%
  left_join(b_ugs, by = c('userId','genres')) %>%
  mutate(residual = residual - b_ug)

# Add b_ug to validation for later incorporation into the final prediction
validation <- validation %>%
  mutate(genres = as_factor(genres)) %>%
  left_join(b_ugs, by = c('userId','genres'))

# Clean up
rm(b_its, b_ugs, b_uts, cv_results, movie_mus, rec,
   user_mus, x, x_pred, predicted_ratings, i)
```

We can now calculate the SVD matrix factorization components.

```r
# Create realRatingMatrix of userId vs movieId, filled with rating residuals
x <- edx %>% select(userId, movieId, residual) %>%
  spread(movieId, residual) %>%
  as.matrix()
rownames(x) <- x[,1] # create rownames
x <- x[,-1]  # trim off userIds to leave just residuals
x <- as( x, 'realRatingMatrix') # convert to recommenderlab type

# Train SVD model. We can optimize with just one calculation if K is large
rec <- Recommender( x,
                    method = 'SVD',
                    list(k = 85,
                         normalize = NULL,
                         verbose = FALSE))

#Do some cleanup, free up RAM before going on
rm(x)
```

```r
# Create lists of unique users and movies in edx
uniq_userIds <- edx %>% group_by(userId) %>%
  summarize(userId = first(userId)) %>%
  pull(userId)

uniq_movieIds <- edx %>% group_by(movieId) %>%
  summarize(movieId = first(movieId)) %>%
  pull(movieId)

# Get index locations for each movie/user in validation set - map to location
# in edx set because output of SVD is in that order
U_indexs <- data.frame( userId = sort(unique(validation$userId)),
                         U_index = which(uniq_userIds %in% validation$userId))

V_indexs <- data.frame( movieId = sort(unique(validation$movieId)),
                         V_index = which(uniq_movieIds %in% validation$movieId))

# Add the location for each movie/user to the validatio set for easier access
validation <- validation %>% left_join(U_indexs, by = 'userId') %>%
    left_join(V_indexs, by = 'movieId')

# Clean up
rm( uniq_userIds, uniq_movieIds, U_indexs, V_indexs )

# Calculate b_ui, element-wise, only for movie/user pairs in validation
# we only use 1:K components of the factorization, per trained K value to
# minimize RMSE
b_ui_SVD <- rep(0,10)
for( i in 1:nrow(validation)) b_ui_SVD[i] <-
    sum( rec@model$svd$u[validation$U_index[i], 1:K_User_Movie_SVD ]  *
          rec@model$svd$d[1:K_User_Movie_SVD] *
          rec@model$svd$v[validation$V_index[i], 1:K_User_Movie_SVD ])

# Add completed b_ui to validation for later addition to final prediction
```

```r
validation <- validation %>%
  mutate(b_ui = b_ui_SVD)
```

Final RMSE

We can now calculate our final loss function value,

```r
# Add the final elements so we can calculate prediction
validation <- validation %>%
  mutate(prediction = mu + b_i + b_u +
           nafill(b_it, fill = 0) +
           nafill(b_ut, fill = 0) +
           nafill(b_ug, fill = 0) +
           b_ui) %>%
  mutate(prediction = SimpleClip(prediction, UB = 5.0, LB = 0.5))

RMSE(validation$rating, validation$prediction) #0.8228107
```

```
## [1] 0.8228107
```

Conclusion

The recommender system we constructed had a final $RMSE$ of 0.8228107. The final model was trained with $5x$ cross-validation on the entire `edx` set, with regularized baseline effects for average rating for each user, average rating for each movie, average rating for each movie in each year after its release date, average rating for each user for each hour of the day, and average rating for each user for each genre combination, each regularized to penalize low sample size. User/genre interaction was further predicted with the Simon Funk variant of Singular Value Decomposition (FSVD or SVDF) with the user/genre averages as the input values and `recommenderlab`'s default training parameters except for our trained values of $\gamma = 0.025, K = 12$. The final piece of the recommender was the user/movie interaction, which we estimated with straight Singular Value Decomposition and a trained value of $K = 80$. Both matrix factorization models were trained with $1x$ cross-validation on `edx`. The final prediction was coerced to the known data range for ratings of $\{0.5, 5.0\}$ by just assigning values that were out of range to the nearest in-bounds value. We refer to this clipping function as $\kappa()$ in the text.

While the final $RMSE$ of 0.8228107 was a very good result, there is still plenty of room for improvement. There were multiple baseline effects that we could have pursued further, such as user rating drift over time and sudden changes in rating frequency. Additionally, we could do more training on the two matrix factorization elements, and we chose not to do the extra training due to the computing resources required to calculate and evaluate multiple large matrices. The initial work on this analysis was done on a system with 8GB of RAM and a Funk SVD model that was trained early in the process took more than eight hours to run on that machine and crashed the `R` session if cleanup was not diligently performed before the calculation was attempted. The rest of the analysis was done on a platform with 62 Gigabytes of RAM. We note that just the `rec` object that is the output of the user/movie matrix factorization operation with `recommenderlab` is more than 55 Megabytes and multiple variables produced in this model are in excess of 1GB each. In short, with these large data objects, swap space and the time required to swap data becomes limiting. If we were going to do this analysis on a platform similar to a small laptop, we would need to explore other ways to handle large data, such as keeping our variables on disk rather than in RAM and working with just small chunks at a time. `R` does have multiple tools for handling these types of problems, but they weren't really covered in the course material and we didn't explore them in this analysis.

We also could have explored more of the neighborhood and clustering models presented in the course material. We noted that the `R` implementation of k-nearest neighbors will throw an error when we attempt to do an analysis with more than fifty five levels in a factorized response variable. A search through online forums suggests a solution to this problem is to edit the source code and recompile from source. We concluded that this was beyond the scope of the current analysis, even though neighborhood techniques seem to be an intuitively good fit for this type of recommender system. It seems that we should be able to cluster users and movies into groups with similar ratings and if users Billy and Bob have similar ratings for many movies, and Billy has rated a movie highly that Bob has not rated, then Bob is likely to also rate that movie highly. In essence, we are performing a similar task with matrix factorization, but it initially seemed that Random Forest or K-Nearest Neighbor techniques could also work well. We note that all of the Netflix Prize solutions we have referenced dealt with matrix factorization techniques and opted not to use decision tree techniques.

One interesting result was the lack of performance-boosting from our clipping techniques. The Funk SVD method uses a sigmoid style of clipping to shrink the predictions back from the outer bounds of the prediction space, and this seems to make sense from a logical perspective. We noted that our predicted ratings contained out-of-bounds values after just the calculation of the user and movie effects, and by feeding the out-of-bounds residuals into the calculation of the next effect, we should have been amplifying our error. However, we noted that both of the clipping methods we tried, even with training, did not improve our performance very much, if at all, and in some cases the clipping procedure contributed very negatively to performance. This was the justification for only applying clipping as the final step, when it would be mathematically guaranteed to improve performance because we do know the range of ratings in the `validation` set, though we know very little else about the ratings in that set. We did not test or train intermediary clipping on all elements of the model, however, so more work could be done to ascertain if intermediary clipping might have been beneficial in some cases.

Overall, this was a good exercise in the construction of a recommender system with matrix factorization. We have demonstrated that matrix factorization is a very powerful and elegant tool and is especially useful when we have missing data in a relational matrix, such as the movie-user matrix we constructed in the analysis. We further demonstrated that matrix factorization is very powerful when we have large datasets because it allows us to reduce dimensionality while simultaneously making a large number of predictions for the missing data. In all, matrix factorization is a wonderful tool to add to our machine learning toolbox.