R Data Science Course 9: MovieLens Project

James Lo

2022-05-26

Contents

Introduction

This project is part of the EdX Harvard R Data Science program. It is based on the real-life Netflix Prize challenge announced in 2006, where the streaming service company offered $1M in USD to anyone who could create a movie rating prediction algorithm that outperformed their then-current system by 10 %.

The data used is a version of MovieLens dataset from grouplens.org with 10 million entries, each one representing a rating of a movie made by a user. The coding portion of the project is divided into three parts: exploratory analysis of the data, preparation and cleaning, and construction of the optimal machine learning model.

First, the required R packages and MovieLens dataset were loaded with the code below:

```
# Create edx set, validation set (final hold-out test set)

if(!require(tidyverse)) install.packages("tidyverse",
                                         repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret",
                                     repos = "http://cran.us.r-project.org")
```

```r
if(!require(data.table)) install.packages("data.table",
                                          repos = "http://cran.us.r-project.org")
if(!require(dplyr)) install.packages("dplyr",
                                     repos = "http://cran.us.r-project.org")
if(!require(dslabs)) install.packages("dslabs",
                                      repos = "http://cran.us.r-project.org")
if(!require(lubridate)) install.packages("lubridate",
                                         repos = "http://cran.us.r-project.org")
if(!require(ggplot2)) install.packages("ggplot2",
                                       repos = "http://cran.us.r-project.org")
library(tidyverse)
library(caret)
library(data.table)
library(dslabs)
library(dplyr)
library(lubridate)
library(ggplot2)

# MovieLens 10M dataset:
# https://grouplens.org/datasets/movielens/10m/
# http://files.grouplens.org/datasets/movielens/ml-10m.zip

# Downloading the MovieLens data
dl <- tempfile()
download.file("https://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

ratings <- fread(text = gsub("::",
                             "\t",
                             readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

movies <- str_split_fixed(readLines(unzip(dl,
                                          "ml-10M100K/movies.dat")),
                          "\\::",
                          3)

colnames(movies) <- c("movieId", "title", "genres")

# if using R 4.0 or later:
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres))

movielens <- left_join(ratings, movies, by = "movieId")

# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
test_index <-
  createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]

# Make sure userId and movieId in validation set are also in edx set
```

```
validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

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

rm(dl, ratings, movies, test_index, temp, movielens, removed)
```

To validate the accuracy of the finished algorithm, roughly 10 % of the dataset was set aside as the object `validation`, while the remaining data was named `edx`.

Methods

In order to determine the optimal model for predicting MovieLens ratings, the structure of the dataset and the distribution of variable values within it was analyzed. Some exploratory analysis was used to determine distributions of potential predictors as well as the ratings themselves.

Exploratory Analysis

Below are the code chunks used for exploratory analysis, accompanied by the insights gained from them, where applicable.

Format of the Dataset

```
str(edx)
```

```
## Classes 'data.table' and 'data.frame':   9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : num  122 185 292 316 329 355 356 362 364 370 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 838984474 838983653 838984885 83
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|A
##  - attr(*, ".internal.selfref")=<externalptr>
```

Printed above is a summary of various variables (columns) in `edx`, and the data types they are in. Knowledge of data format is crucial for proper analysis. The outcome variable `rating` is in numeric format. Meanwhile, the date-time variable `timestamp` (which represents the times at which ratings were submitted) is given in integer timestamps, requiring conversion to date-time formats easier to interpret by humans. User and movie IDs are represented by the variables `userId` and `movieId` respectively. Each movie's combination of genres is stored in the character vector `genres`, sharing the same data format as `title`.

Dimensions of the Dataset

```
# dimensions of the dataset
nrow(edx) # 9000055 rows
```

```
## [1] 9000055
```

```
ncol(edx) # 6 cols
```

```
## [1] 6
```

The `edx` dataset has 9000055 rows (entries) and 6 columns (variables).

Number of Unique Movies and Users

```
# number of unique movies
length(unique(edx$movieId))
```

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

```
# number of unique users
n_distinct(edx$userId)
```

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

`edx` contains 10677 unique movies and 68978 unique users. This translates into 736478106 possible combinations of movies and users, far larger than the 9000055 observations actually present. Converting `edx` to a very large and sparse user x movie matrix would not be a feasible analytical approach.

Number of Movies Rated by Users

```
# distribution of movies rated among users
edx %>%
  group_by(userId) %>%
  summarize(movies_rated = n()) %>%
  summarize(max = max(movies_rated),
            min = min(movies_rated),
            mean = mean(movies_rated),
            median = median(movies_rated))
```

max	min	mean	median
6616	10	128.7967	62

Table 1: Summary statistics of movies rated by each user

```r
# distribution of movies rated among users, but plotted
edx %>%
  group_by(userId) %>%
  summarize(movies_rated = n()) %>%
  ggplot(aes(movies_rated)) +
  geom_histogram(binwidth = 10) +
  xlim(c(0, 2000)) +
  ggtitle('Distribution of Movies Rated by Each User')
```

```
## Warning: Removed 58 rows containing non-finite values (stat_bin).
```

```
## Warning: Removed 2 rows containing missing values (geom_bar).
```

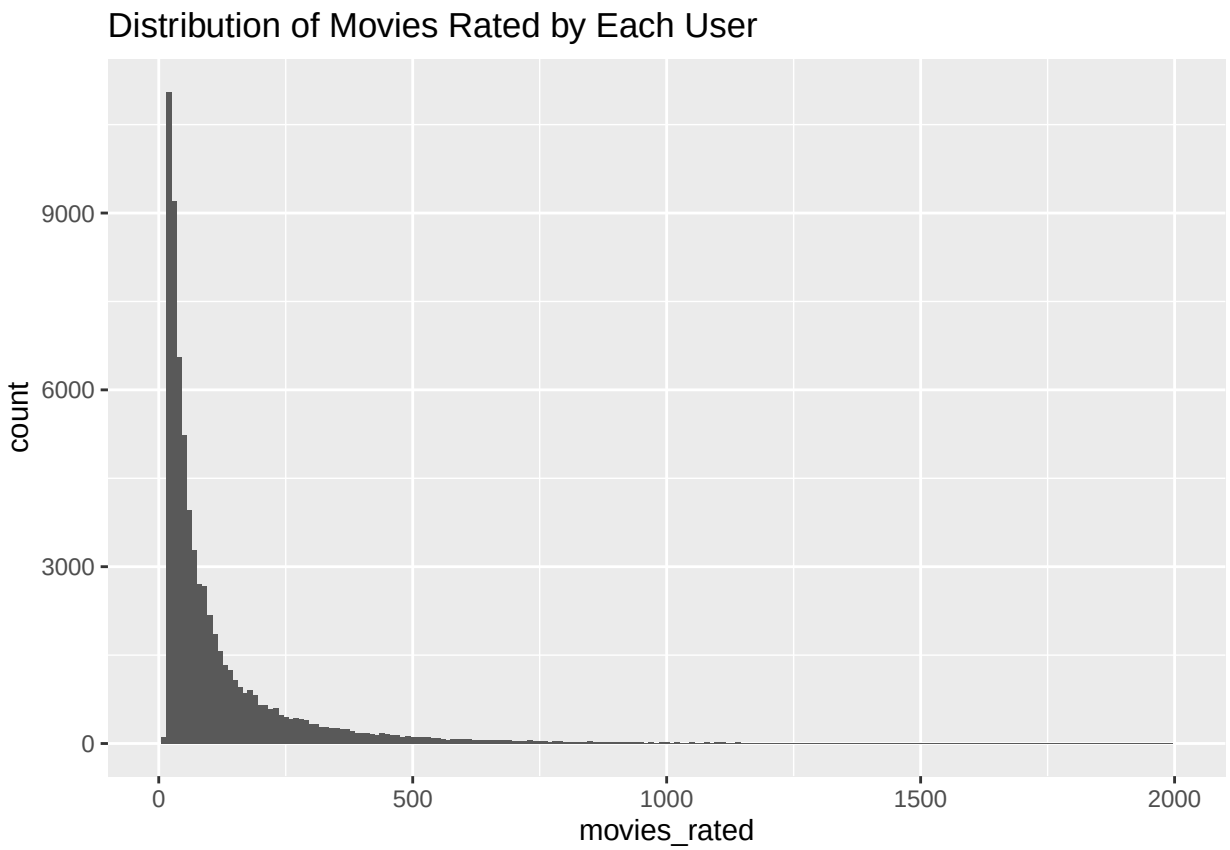


Figure 1: A Histogram of the amount of movies rated by each user

Both the summary table and histogram indicated a heavily right-skewed distribution of movies rated by each user, with the majority of users having rated a low amount of movies, while the mean is inflated by a small proportion of prolific reviewers.

Number of Ratings Received by Movies

```r
# quick glance at the 20 most frequently rated movies
edx %>% group_by(title) %>% summarize(ratings = n()) %>%
```

```
  arrange(desc(ratings)) %>%
  head(.,20)
```

title	ratings
Pulp Fiction (1994)	31362
Forrest Gump (1994)	31079
Silence of the Lambs, The (1991)	30382
Jurassic Park (1993)	29360
Shawshank Redemption, The (1994)	28015
Braveheart (1995)	26212
Fugitive, The (1993)	25998
Terminator 2: Judgment Day (1991)	25984
Star Wars: Episode IV - A New Hope (a.k.a. Star Wars) (1977)	25672
Apollo 13 (1995)	24284
Batman (1989)	24277
Toy Story (1995)	23790
Independence Day (a.k.a. ID4) (1996)	23449
Dances with Wolves (1990)	23367
Schindler's List (1993)	23193
True Lies (1994)	22823
Star Wars: Episode VI - Return of the Jedi (1983)	22584
12 Monkeys (Twelve Monkeys) (1995)	21891
Usual Suspects, The (1995)	21648
Fargo (1996)	21395

Table 2: 20 of the most frequently rated movies in the edx dataset

As expected, many of the most frequently rated movies in `edx` are popular releases that have left a lasting impact on popular culture, such as *Pulp Fiction* and *Jurassic Park*.

```
# distribution of how frequently each movie is rated
edx %>%
  group_by(movieId) %>%
  summarize(ratings_received = n()) %>%
  summarize(max = max(ratings_received),
            min = min(ratings_received),
            mean = mean(ratings_received),
            median = median(ratings_received))
```

max	min	mean	median
31362	1	842.9386	122

Table 3: Summary statistics of ratings received for each movie

```
# same thing but plot
edx %>%
  group_by(movieId) %>%
  summarize(ratings_received = n()) %>%
  ggplot(aes(ratings_received)) +
```

```
  geom_histogram(binwidth = 10) +
  xlim(c(0, 10000)) +
  ggtitle('Distribution of Ratings Received by Each Film')
```

```
## Warning: Removed 143 rows containing non-finite values (stat_bin).
```

```
## Warning: Removed 2 rows containing missing values (geom_bar).
```

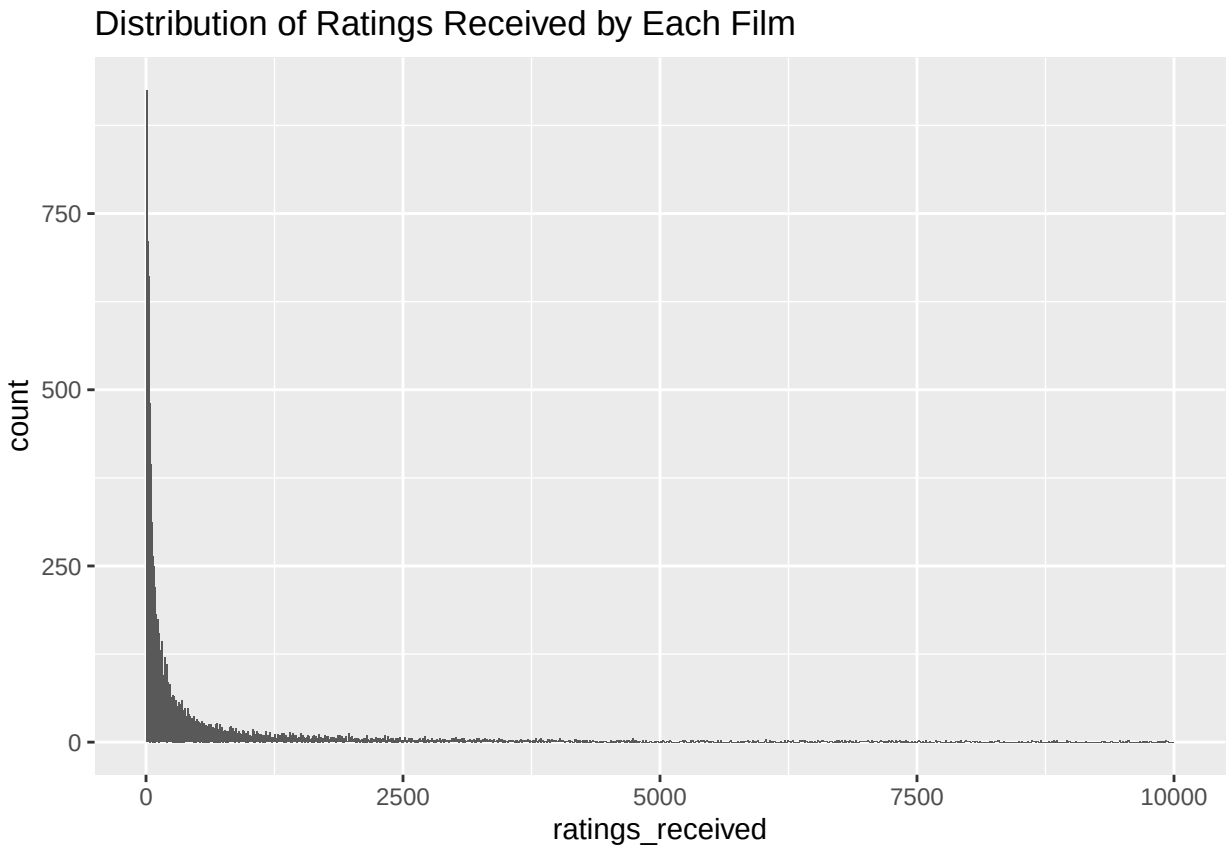


Figure 2: A histogram of the amount of ratings received for each movie

The distribution of ratings received by films is also heavily right-skewed, more extremely so than the distribution of user activities.

Time Span Covered by Dataset

```
# time span covered in data
range(as_datetime(edx$timestamp))
```

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

The `edx` dataset covers a time period from Jan 9th, 1995 to Jan 5th, 2009, with all date-times in Coordinated Universal Time. Given the level of detail in the date-time records (down to the second of each day), the variable is effectively

continuous. However, modeling date effects with such amounts of detail would lead to the inclusion of signal noise, so broader time scales would be modeled with smoothing techniques to exclude noise from trends of interest. In addition, given the amount of ratings in **edx**, plotting every entry to make a smooth model would not be feasible. Instead, the ratings would be grouped by date units of choice (e.g. months) and averaged, then plotted to produce a smooth function.

Genres and Their Distributions

```
# How many genres are there?
edx %>% separate_rows(genres, sep = "\\|") %>%
  group_by(genres) %>%
  summarize(count = n()) %>%
  arrange(desc(count))
```

genres	count
Drama	3910127
Comedy	3540930
Action	2560545
Thriller	2325899
Adventure	1908892
Romance	1712100
Sci-Fi	1341183
Crime	1327715
Fantasy	925637
Children	737994
Horror	691485
Mystery	568332
War	511147
Animation	467168
Musical	433080
Western	189394
Film-Noir	118541
Documentary	93066
IMAX	8181
(no genres listed)	7

Table 4: All the individual genres present in the dataset, arranged from most to least common

There are 19 separate genres (plus the 20th category, "no genres listed") within the **edx** dataset, some more popular than others. Only 7 movies have no genres listed. The most popular genre is Drama (featured in 3910127 ratings), while the least popular genre is IMAX (more of a movie display method than an actual genre) at 8181 ratings. Excluding IMAX and "no genres listed", the least popular genre is Documentary at 93066 ratings.

```
# genre groups
edx %>%
  group_by(genres) %>%
  summarize(count = n()) %>%
  arrange(desc(count)) %>%
  nrow()
```

```
## [1] 797
```

```
# examining data format
edx %>%
  group_by(genres) %>%
  summarize(count = n()) %>%
  arrange(desc(count)) %>%
  head(.,20)
```

genres	count
Drama	733296
Comedy	700889
Comedy|Romance	365468
Comedy|Drama	323637
Comedy|Drama|Romance	261425
Drama|Romance	259355
Action|Adventure|Sci-Fi	219938
Action|Adventure|Thriller	149091
Drama|Thriller	145373
Crime|Drama	137387
Drama|War	111029
Crime|Drama|Thriller	106101
Action|Adventure|Sci-Fi|Thriller	105144
Action|Crime|Thriller	102259
Action|Drama|War	99183
Action|Thriller	96535
Action|Sci-Fi|Thriller	95280
Thriller	94662
Horror|Thriller	75000
Comedy|Crime	73286

Table 5: The genre combinations as they are stored in the dataset, arranged from most to least popular

However, the `genres` column stores genre information as character strings, with each string being a combination of genres separated by the character `|`. There are 797 such combinations. Movies listed as Dramas alone is the most popular at 733296 ratings received, and Drama is included in 6 out of the 10 most common values in the column. Comedies and Action films are also highly popular, corresponding to the ranking of individual genres displayed in Table 4.

Distribution of Ratings

```
mean(edx$rating)
```

```
## [1] 3.512465
```

```
median(edx$rating)
```

```
## [1] 4
```

The grand mean of ratings in the `edx` dataset is 3.512465 stars, while the median is 4.

```
# quick glance at the most commonly given out ratings
edx %>%
  group_by(rating) %>%
  summarize(counts = n()) %>%
  arrange(desc(counts))
```

rating	counts
4.0	2588430
3.0	2121240
5.0	1390114
3.5	791624
2.0	711422
4.5	526736
1.0	345679
2.5	333010
1.5	106426
0.5	85374

Table 6: Ranking of the most commonly given rating scores

Movie ratings in `edx` are given as scores out of 5. No movie in the dataset got 0 points, while the lowest score present in the dataset (0.5) is also the least commonly given. Ratings of 4 and 3 are the most commonly given at 2588430 and 2121240 counts, respectively.

```
# plot distribution of rating scores
edx %>%
  group_by(rating) %>%
  summarize(frequency = n() / nrow(edx)) %>%
  ggplot(aes(x = factor(rating), y = frequency)) +
  geom_bar(stat = 'identity') +
  ggtitle('Frequencies of Ratings Given')
```

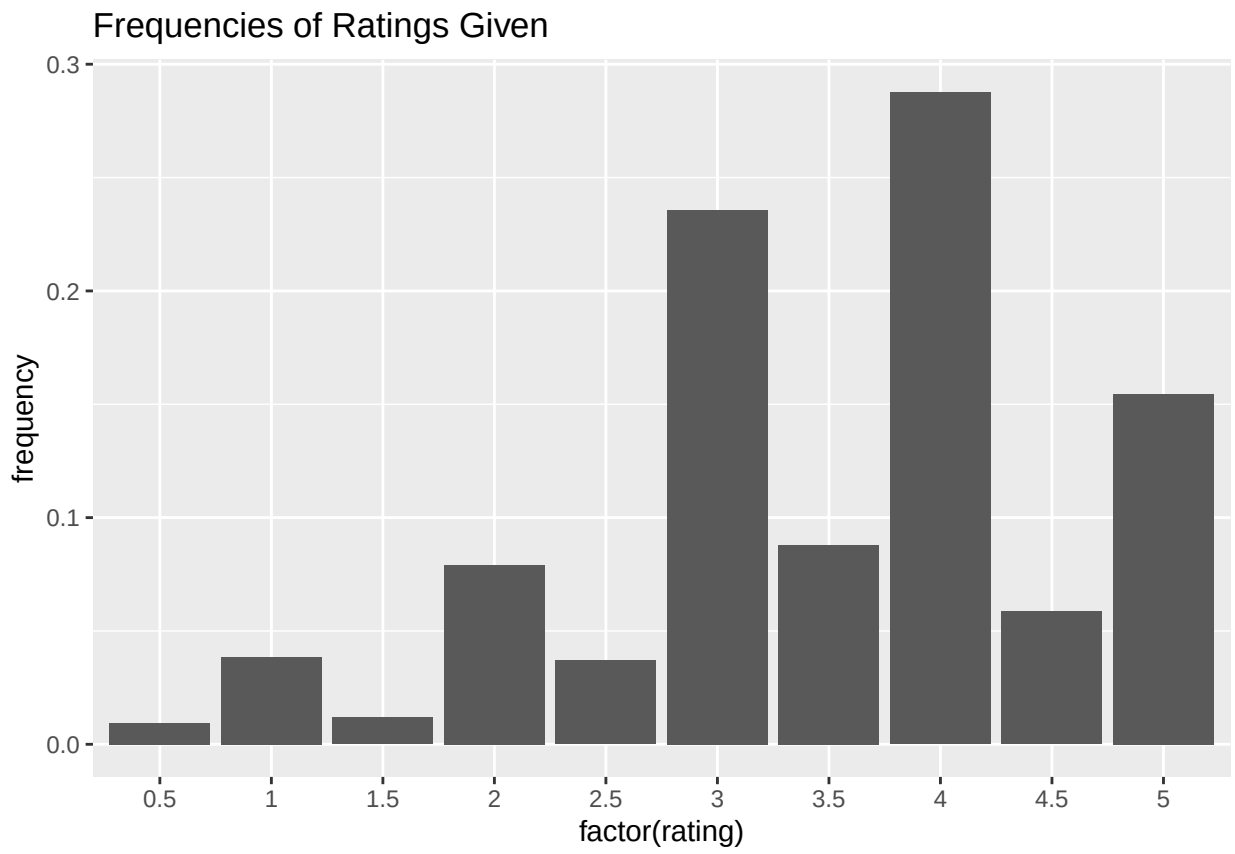


Figure 3: A bar plot of the various ratings in the dataset given out by reviewers

Unlike the distribution of ratings given per user and ratings received per movie, the distribution of rating scores in `edx` is left-skewed, with the mean score of ~3.51 being lower than the median of 4. Ratings of whole numbers are more common than their neighboring decimal scores.

The ratings are neither continuous nor perfectly categorical. Instead, they are an ordinal variable: the numbers are discrete and can be thought of as different categories, but there is a clear order between said categories. For instance, if an user actually rated a movie 3 out of 5, a prediction of 3.5 would be more accurate than 5 or 1.

Choice of Methodology

Model and Accuracy Metrics

The dataset is large, but unevenly distributed. Attempting to construct an users x movies matrix would not be feasible given the limitations of the laptop used in this project. In addition, many of the more computationally intensive models such as K-Nearest Neighbors and Decision Trees could not be run due to the heavy demands on computational power and memory (corroborated by this student's attempts to build such models). As such, a regression-based model as described in the program's online textbook was chosen for analyzing the data. This technique was used in the original Netflix Prize challenge, where it was known as Normalization of Global Effects

The linear model for can be written in the following format:

$$Y_i = \mu + b_1 + b_2 + ... + b_n + \epsilon$$

Where Y is the predicted rating, μ is the average rating across the entire dataset, and n is the amount of predictors. ϵ is the residual error, which is assumed independent and averaging at 0. For a given predictor variable x, b_x is the

modeled effect of x on movie ratings. b_x was computed by grouping each predictor variable x by its values, then taking the group means before subtracting μ and all predictor effects before it. Each rating prediction incorporated a value of b_x depending on the rating's corresponding value of predictor x. The `dplyr` function `left_join()` was used to match rating entries with correct b_x values, by the predictor x.

Each model would produce a vector of continuous numeric variables as the predicted ratings. As such, the metric Root Mean Squared Errors (RMSE) was used; this was the same metric used in the original challenge. For this project, the threshold of success was set at RMSE < 0.86490.

This project did not choose to compute the accuracy of rating predictions as categorical variables for multiple reasons. First, due to the ordinal nature of rating scores, treating ratings as purely categorical would not factor in margins of error. Second, even if the ratings are treated as ordinal categories, the heavily imbalanced distribution of ratings (see Table 6 and Figure 3) would lead to an accuracy paradox where during optimization, an algorithm is trained to simply predict the most common class at all times to raise accuracy.

Regularization

Note that groups means were taken to build the b_x terms, even though the `edx` dataset contains uneven distributions of values in each variable; many such group means would inevitably be calculated from very small sample sizes, greatly increasing data noise and uncertainty in the predictions. To account for the heavily skewed movie and user representations, a regularization technique was used.

The regularization formula was attached as a parameter to each b_x, and it can be written in the following format:

$$1/(\lambda + n_i)$$

Where λ is the term, and n_i is the sample size of the group i within a given predictor variable. For a given λ, as n_i increases $\lambda + n_i$ approximates n_i alone, and the penalty term becomes less significant.

As an example, a regularized first predictor term b_1 for value i would look like this:

$$\hat{b_{1,i}}(\lambda) = \frac{1}{\lambda + n_i} \sum (Y_{1,i} - \hat{\mu})$$

As seen in the formula, larger values of λ shrinks the estimate $\hat{b_{1,i}}$ further, which translates to a heavier penalization at a given n_i.

Parameter Tuning

The optimal value for λ depends on the group sample sizes involved. To tune for λ, a holdout method was used with a 90 % of `edx` entries being assigned to the training set `edx_train` and 10 % to the test set `edx_test`. The holdout approach was chosen over more accurate but computationally expensive techniques, such as k-fold validation, to minimize extra computation on the already-large `edx` dataset. The `sapply()` function was used to try out multiple λ values, from which the one producing the lowest RMSE was chosen.

Iterative Inclusion of Predictor Effects

To compare the changes in performance as more predictors are included into the linear regression model, 7 iterations were trained on `edx_train` and their RMSEs computed by predicting ratings in `edx_test`. Model 1, derived from the online textbook, included the movie and user effects. Its formula is displayed below:

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

Where b_i denotes an movie-specific effect, and b_u denotes a user-specific effect.

Date Effect The function `as.datetime()` from the `lubridate` package was used to convert all timestamps into dates before computing for date-related effects, which were included in the second to fifth models.

Models 2a1 and 2a2 incorporated regularized mean ratings, grouped by different time frames; 2a1 contained yearly means only, while 2a2 contained a term each for year- and month-specific effects. Their formulas are shown below:

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

$$Y_{i,u,yr,mo} = \mu + b_i + b_u + b_{yr} + b_{mo}$$

Where b_{yr} and b_{mo} are year- and month-specific effects, respectively.

For models 2b1 and 2b2, regularized average effects of dates (year-month-day, rather than years or months) on rating were taken and used to plot a smooth function using `loess()`. Both RMSEs and extent of smoothing were considered in choosing the final date-effect model. To check for under- or oversmoothing, smooth models were plotted at different spans. The formula for spans is given by the following expression:

$$d/(lastdate - firstdate)$$

Where d is the amount of days included in the span.

The formula is thus written as:

$$Y_{i,u,d} = \mu + b_i + b_u + f(b_d) + \epsilon$$

Where $f(b_d)$ is a smooth function of the date effect b_d.

Genre Effect The sixth and seventh models factored in genre effects, in addition to the optimized model chosen from previous tuning runs (which included user, movie, and date effects). Model 3a grouped the values in the `genres` column as they are to compute regularized means, and is represented by the following formula:

$$Y_{i,u,d,gg} = \mu + b_i + b_u + f(b_d) + b_{gg} + \epsilon$$

Where b_{gg} is the effect of genre combinations.

For model 3b, a vector `genre_list` (containing all the `genre` column values in the dataset) was used with the function `str_detect()` (in the `stringr` package) to generate a logical matrix with 20 columns for the values in `genres`, and rows corresponding to ratings in the dataset. To generate average genre effects, the `sapply()` function and genre matrix were used to repeat the following steps for each genre listed in `genre_list`:

- Filter the rating residuals (with the mean and previous effects subtracted) for values whose indexes contain TRUE in the genre matrix column corresponding to the given genre.
- Then take the regularized average residual associated with the given genre.

To include each genre effect b_g (stored in a vector produced by the `sapply()` function), the model is first fitted on `edx_test` with the other effects, then a `for()` loop is called on `genre_list` to add each genre's b_g onto predicted ratings whose indexes correspond to `TRUE` values under the relevant genre matrix column. The final model is represented by the formula:

$$Y_{u,i,d,gg} = \mu + b_u + b_i + f(b_d) + \sum_{g=1}^{g} x_{u,i} b_g + \epsilon$$

For each genre g that movie i belongs to, $x_{u,i} = 1$ and the genre effect b_g is added to the genre effect term $\sum_{k=1}^{k} x_{u,i} b_g$.

Genre matrices for both the train and test sets were created to avoid calling `str_search()`, which takes some time to run, repeatedly during the tuning process.

Data Cleaning and Wrangling

The following data preparation steps were performed before generating and comparing the planned models:

Creating a Vector of All Genre Names

```
genre_list <- c('Drama', 'Comedy', 'Action', 'Thriller',
                'Adventure', 'Romance', 'Sci-Fi', 'Crime',
                'Fantasy', 'Children', 'Horror', 'Mystery',
                'War', 'Animation', 'Musical', 'Western',
                'Film-Noir', 'Documentary', 'IMAX', '(no genres listed)')
```

Converting Timestamps to More Human-Readable Formats

```
# convert timestamp to year, month, and date
edx <- edx %>%
  mutate(year = year(as_datetime(timestamp)),
         month = month(as_datetime(timestamp)),
         date = date(as_datetime(timestamp))) %>%
  select(-c(timestamp, title))

# Repeat the same process with validation set
validation <- validation %>%
  mutate(year = year(as_datetime(timestamp)),
         month = month(as_datetime(timestamp)),
         date = date(as_datetime(timestamp))) %>%
  select(-c(timestamp, title))
```

After `year`, `month`, and `date` were created, the original `timestamp` column was no longer needed and therefore deleted.

Partition edx Between Training and Testing Sets

```
# split your edx data into train and test before modeling!
# Let's assign 90 % for train, 10 % for test
set.seed(1, sample.kind = 'Rounding') # arbitrary seed for reproducibility
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

```
test_index <- createDataPartition(y = edx$rating,
                                  times = 1,
                                  p = 0.1,
                                  list = F)
# Test-train split
edx_train <- edx[-test_index,]

edx_temp <- edx[test_index,]

# making sure that the movie and user IDs overlap using semi_join
edx_test <- edx_temp %>%
  semi_join(edx_train, by = "movieId") %>%
  semi_join(edx_train, by = "userId")

# Add rows removed from edx_test back into edx_train
removed <- anti_join(edx_temp, edx_test)
edx_train <- rbind(edx_train, removed)
rm(edx_temp)
rm(removed)
rm(test_index)
```

A 90-10 split was made using the `caret` package function `createDataPartition()`, while an arbitrary seed of 1 was set to ensure reproducibility in the pseudo-random split.

Computing the baseline average

```
base_avg <- mean(edx_train$rating)
```

The base average for training set ratings was calculated to be 3.5124556.

Creating the RMSE function

```
RMSE <- function(pred, real) {
  sqrt(mean((pred - real)^2))}
```

The RMSE function is written mathematically as follows:

$$RMSE = \sqrt{\frac{\sum(\hat{Y} - Y)^2}{n}}$$

Creating the Genre Matrices

The code chunks displayed below are deliberately kept from running with the R markdown option eval = FALSE, to save memory until the matrices are needed. Functional copies of the code are run in the Results section under model 3b, but hidden from the knitted .pdf file using include = FALSE to avoid redundancy.

```
edx_train_genres <-
  edx_train[,4]
# the column number with genre information, after the previous modifications
# only the needed column is chosen, to avoid overextention of memory during computation

edx_train_genres <-
  sapply(genre_list, function(g) {
    edx_train_genres$g <- str_detect(g, edx_train_genres$genres)})
```

```
edx_test_genres <-
  edx_test[,4]

edx_test_genres <-
  sapply(genre_list, function(g) {edx_test_genres$g <-
           str_detect(g, edx_test_genres$genres)})
```

For each genre in `genre_list`, `str_detect()` checks if it exists in each row of the `genres` column, assigning `TRUE` if it does and `FALSE` if it doesn't.

Results

Iteration 1: Movie and User IDs only

```
# Progressively narrower lambdas for tuning
lambdas <- seq(0, 20, 1)
lambdas <- seq(4, 6, 0.1)
lambdas <- seq(4.85, 5, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
  # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # fitting (for each given lambda l)
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(pred = base_avg + bi + bu) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, edx_test$rating))})
```

```
plot(lambdas, rmses) # for visualization during tuning
```

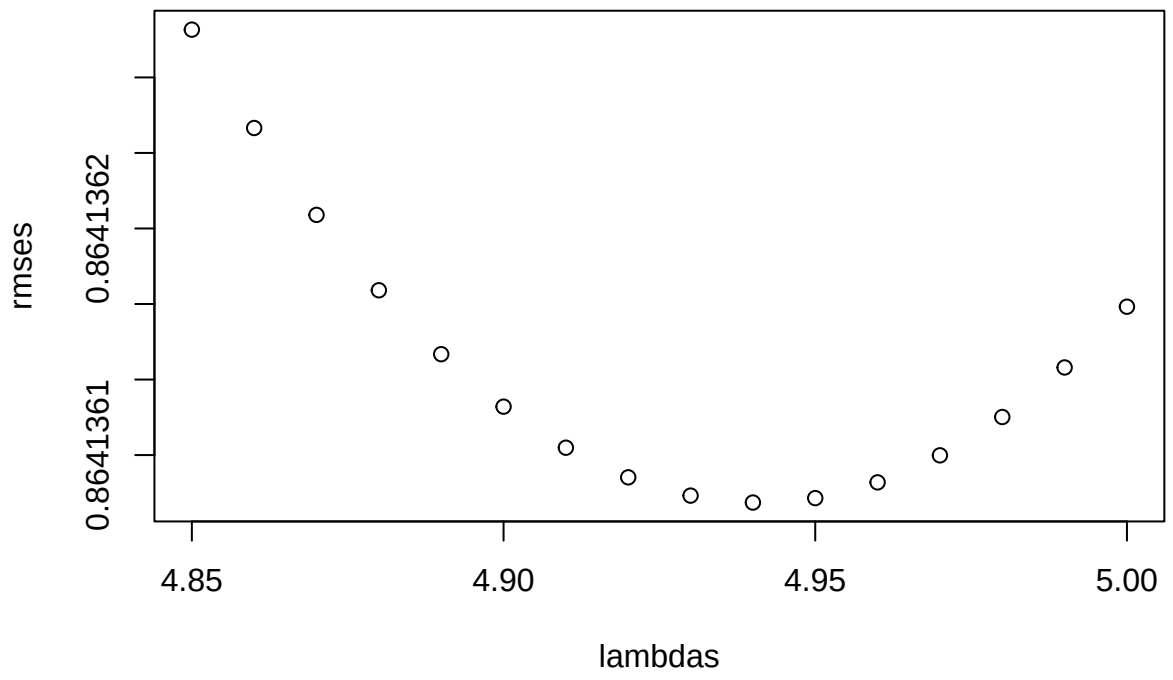


Figure 4: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 1

```
# saving the lambda and RMSE for comparison later
lambda_1 <- lambdas[which.min(rmses)]
train_rmse_1 <- min(rmses)
```

Model 1 returned an RMSE of 0.8641361, with a λ of 4.94.

Iteration 2a1: Grouped Year Means

```
# Iteration 2a1: year effect modeled by grouped averages
lambdas <- seq(0, 10, 1)
lambdas <- seq(4.5, 5.5, 0.1)
lambdas <- seq(4.9, 5.1, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
  # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
```

```
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # year effect
  year_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(year) %>%
    summarize(byr = sum(rating - base_avg - bi - bu)/(n() + l))
  # fitting (for each given lambda l)
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    left_join(year_avg, by = 'year') %>%
    mutate(pred = base_avg + bi + bu + byr) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, edx_test$rating))})

plot(lambdas, rmses) # for tuning purposes only
```

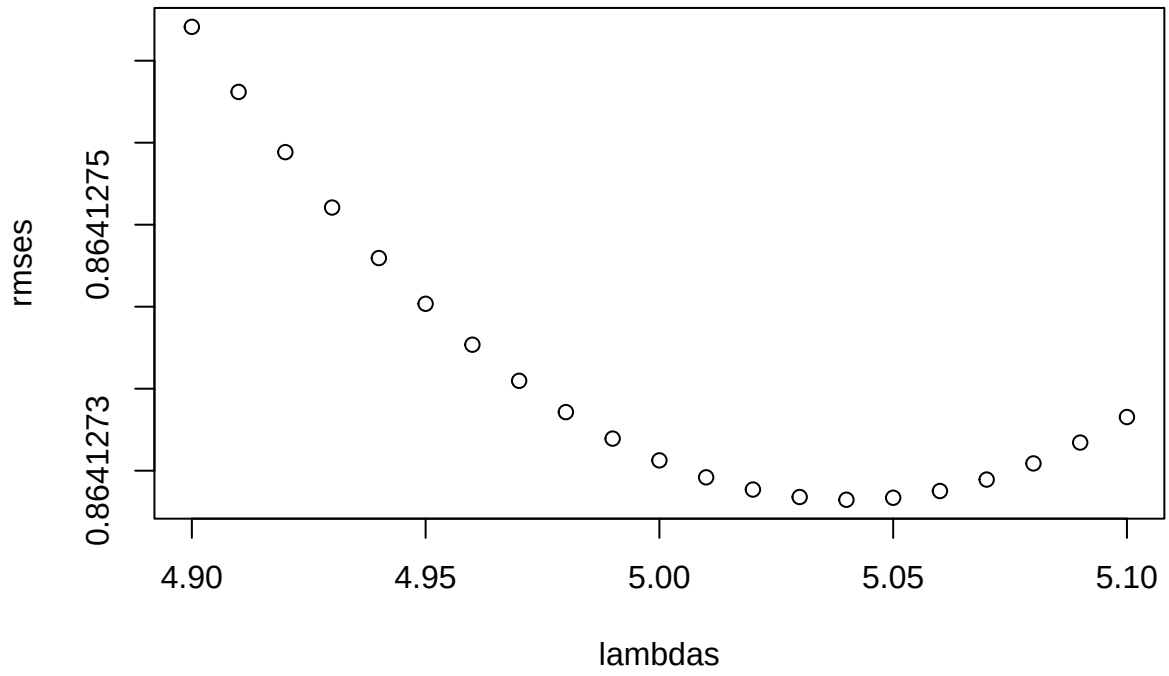


Figure 5: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 2a1

```
lambda_2a1 <- lambdas[which.min(rmses)]
train_rmse_2a1 <- min(rmses)
```

Model 2a1 returned an RMSE of 0.8641273, with a λ of 5.04.

Iteration 2a2: Grouped Year and Month Means

```
# iteration 2a2: year, month modeled by grouped avgs
lambdas <- seq(0, 10, 1)
lambdas <- seq(4, 6, 0.1)
lambdas <- seq(4.95, 5.1, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
 # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # year effect
  year_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(year) %>%
    summarize(byr = sum(rating - base_avg - bi - bu)/(n() + l))
  # month effect
  month_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    left_join(year_avg, by = 'year') %>%
    group_by(month) %>%
    summarize(bmo = sum(rating - base_avg - bi - bu - byr)/(n() + l))
  # fitting (for each given lambda l)
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    left_join(year_avg, by = 'year') %>%
    left_join(month_avg, by = 'month') %>%
    mutate(pred = base_avg + bi + bu + byr + bmo) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, edx_test$rating))})

plot(lambdas, rmses) # for tuning only
```

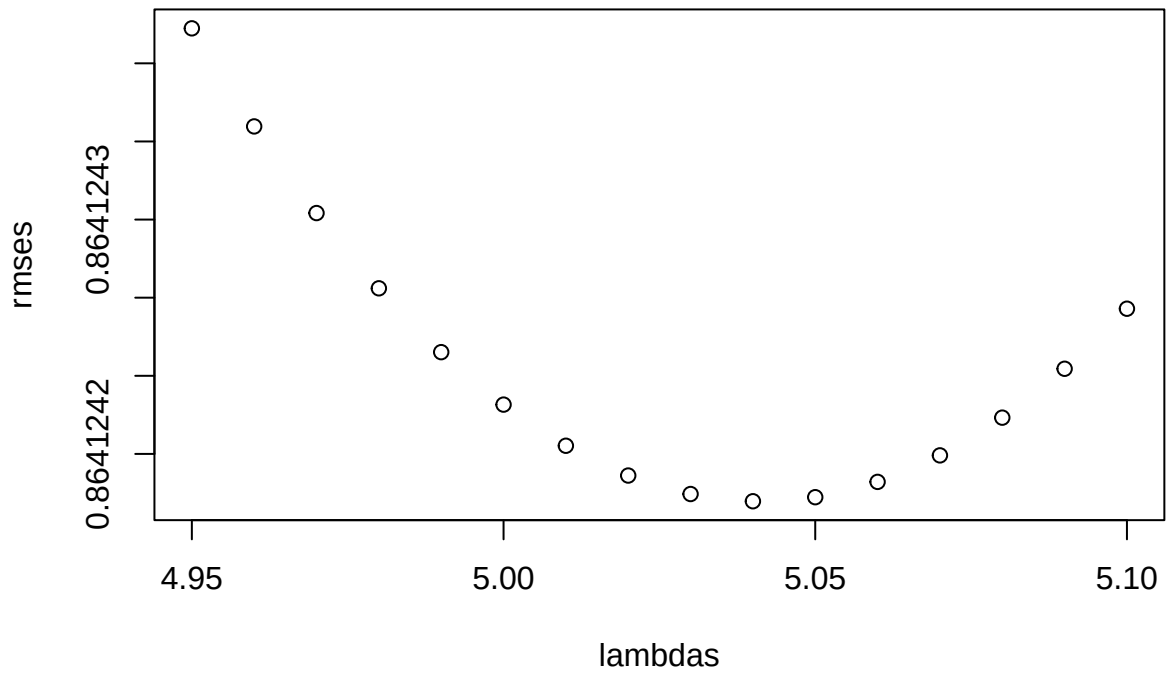


Figure 6: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 2a2

```
lambda_2a2 <- lambdas[which.min(rmses)]
train_rmse_2a2 <- min(rmses)
```

Model 2a2 returned an RMSE of 0.8641242, with a λ of 5.04.

Iteration 2b1: Smooth Function, Monthly Span

```
# Iteration 2b1: some kind of model on dates (chose loess smooth)
span <- 30 / as.numeric(diff(range(edx_train$date)))

lambdas <- seq(0, 10, 1)
lambdas <- seq(4.4, 5.6, 0.1)
lambdas <- seq(5.15, 5.25, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
  # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
```

```
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # date effect; will smooth
  date_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(date) %>%
    summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))
  #modeling a smooth function out of the date effect
  date_effect_model <- loess(bd ~ as.numeric(date),
                             data = date_avg,
                             span = span)
  #...then make predictions based on it
  date_effect_preds <- predict(date_effect_model, edx_test$date)
  # fitting (for each given lambda l)
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(pred = base_avg + bi + bu + date_effect_preds) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, edx_test$rating))})

plot(lambdas, rmses) # for tuning only
```

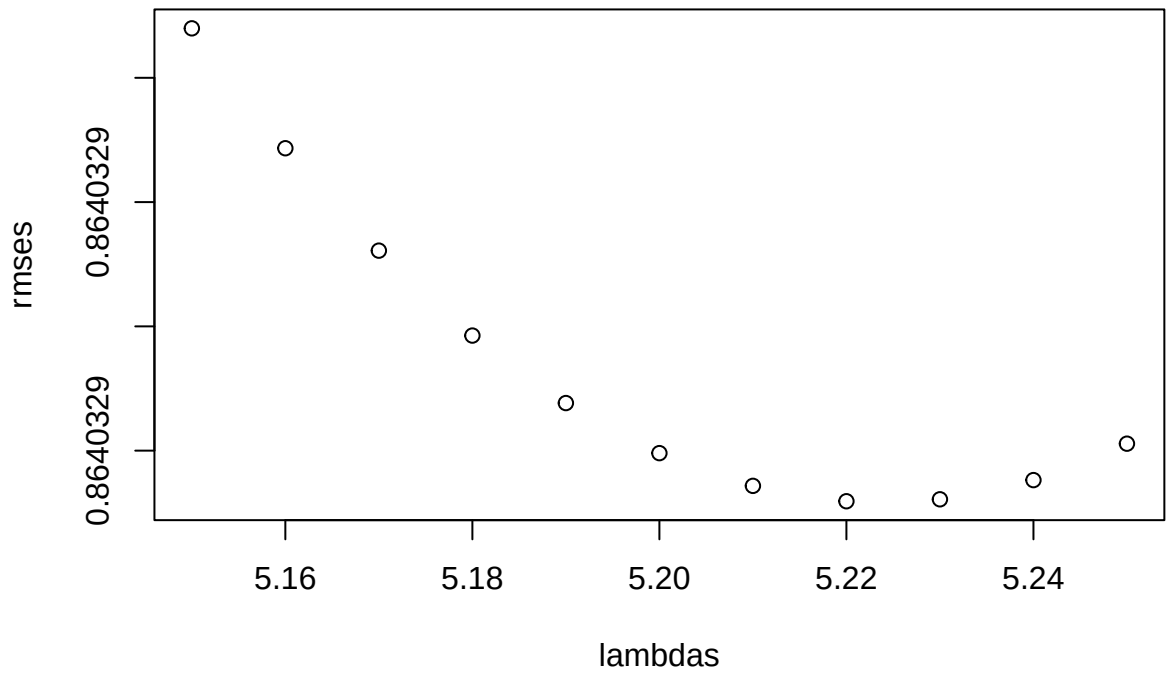


Figure 7: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 2b1

```
lambda_2b1 <- lambdas[which.min(rmses)]
train_rmse_2b1 <- min(rmses)
```

Model 2b1 returned an RMSE of 0.8640329, with a λ of 5.22.

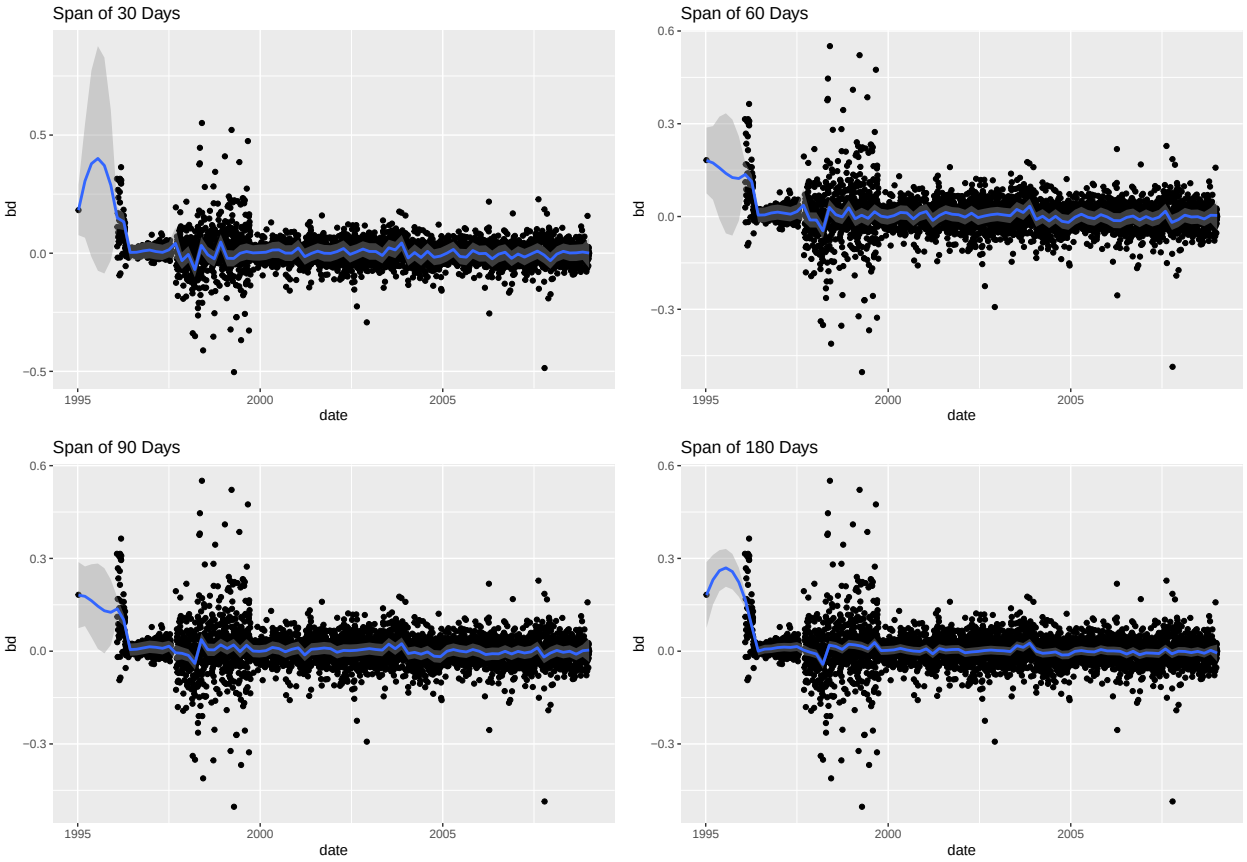
Did Model 2b1 Overfit?

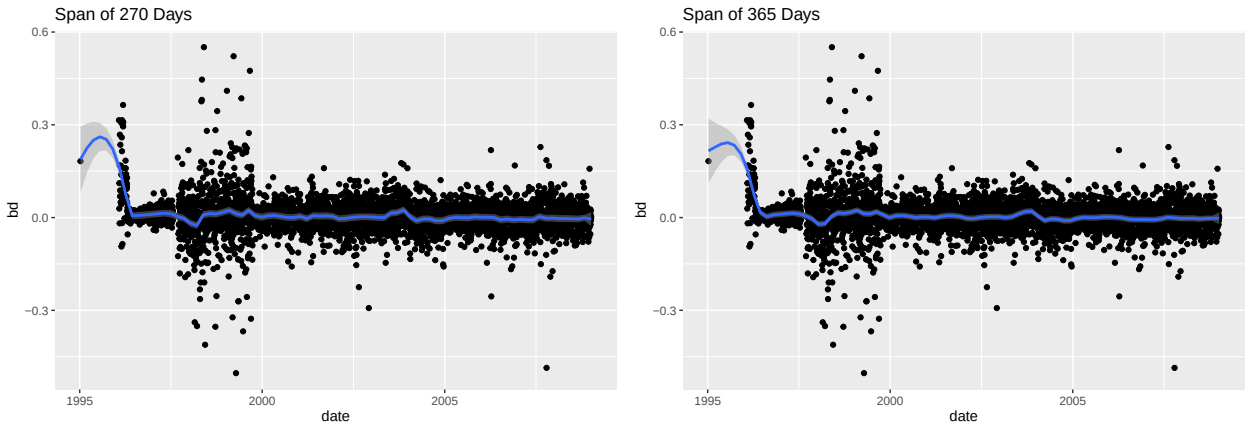
To check for signs of overfitting, the smooth date effect function was plotted with λ = 5.22 and various span Lengths:

```
l <- lambda_2b1
# the mu
base_avg <- mean(edx_train$rating)
# movie effect
movie_avg <- edx_train %>%
  group_by(movieId) %>%
  summarize(bi = sum(rating - base_avg)/(n() + l))
# user effect
user_avg <- edx_train %>%
  left_join(movie_avg, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(bu = sum(rating - base_avg - bi)/(n() + l))
# date effect; will smooth
date_avg <- edx_train %>%
```

```
  left_join(movie_avg, by = 'movieId') %>%
  left_join(user_avg, by = 'userId') %>%
  group_by(date) %>%
  summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))
# plotting the date effect, under different spans
# but controlling for lambda
spans <- c(30, 60, 90, 180, 270, 365)

plots <- for (s in spans) {
  plot <- date_avg %>%
    ggplot(aes(x = date, y = bd)) +
    geom_point() +
    geom_smooth(method = 'loess',
                span = s / as.numeric(diff(range(edx_train$date)))) +
    ggtitle(paste('Span of', as.character(s), 'Days', sep = ' '))
  print(plot)}
```





Figures 8a-e: loess functions of average rating residuals over date. Residuals were grouped by date, then averaged with regularization (lambda = 5.22). Spans of 30, 60, 90, 180, 270, and 365 days were used.

All but the 365-day span exhibited signs of undersmoothing, with jagged portions along their graphs' trend lines. Despite the improved RMSE of Model 2b1, it ran the risk of overfitting the data by including signal noise.

Iteration 2b2: Smooth Function, Yearly Span

```
# 2b2: optimized span of a year; re-tune lambda
span <- 365 / as.numeric(diff(range(edx_train$date)))

# increasing narrow lambdas for tuning
lambdas <- seq(0, 10, 1)
lambdas <- seq(4, 6, 0.1)
lambdas <- seq(5.05, 5.15, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
  # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # date effect; will smooth
  date_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(date) %>%
    summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))
  #modeling a smooth function out of the date effect
  date_effect_model <- loess(bd ~ as.numeric(date),
                             data = date_avg,
                             span = span)
```

```
  #...then make predictions based on it
  date_effect_preds <- predict(date_effect_model, edx_test$date)
  # fitting (for each given lambda l)
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(pred = base_avg + bi + bu + date_effect_preds) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, edx_test$rating))})

plot(lambdas, rmses) # tuning only
```

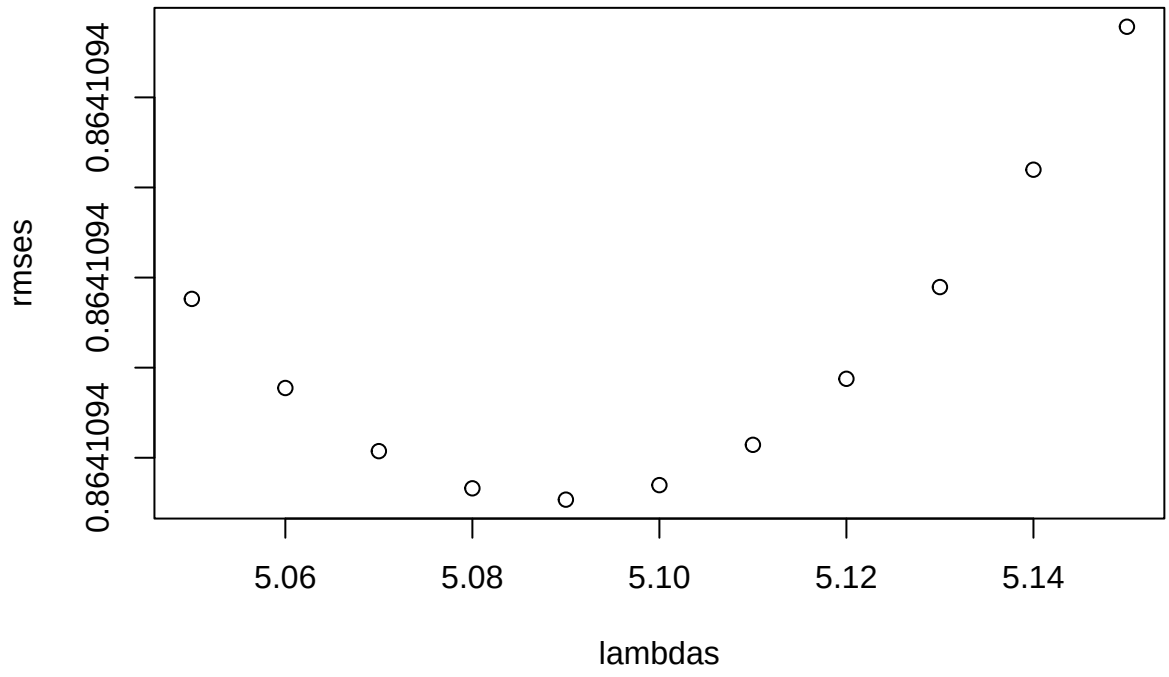


Figure 9: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 2b2

```
lambda_2b2 <- lambdas[which.min(rmses)]
train_rmse_2b2 <- min(rmses)
```

Model 2b2 returned an RMSE of 0.8641094, with a λ of 5.09.

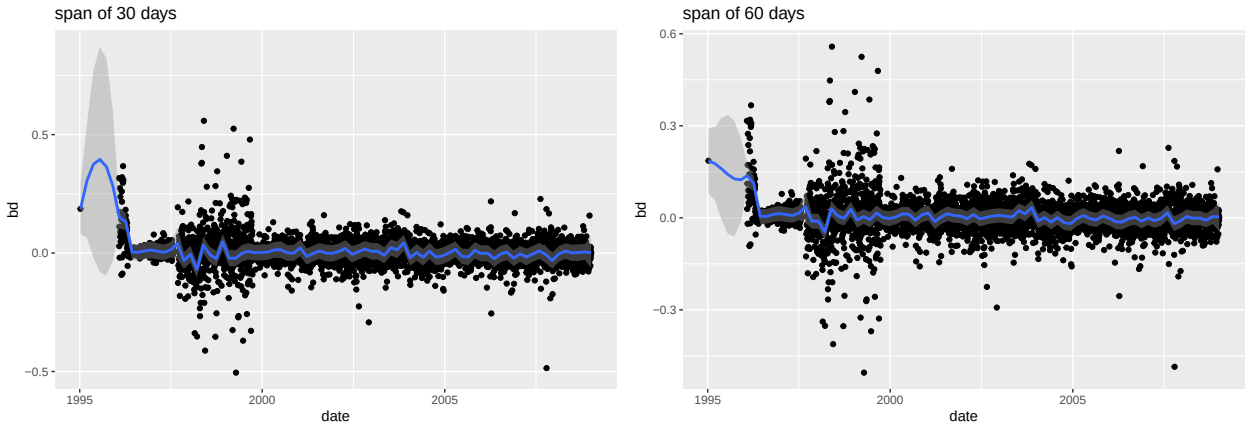
Did Changing the Lambda Affect the Smooth Model?

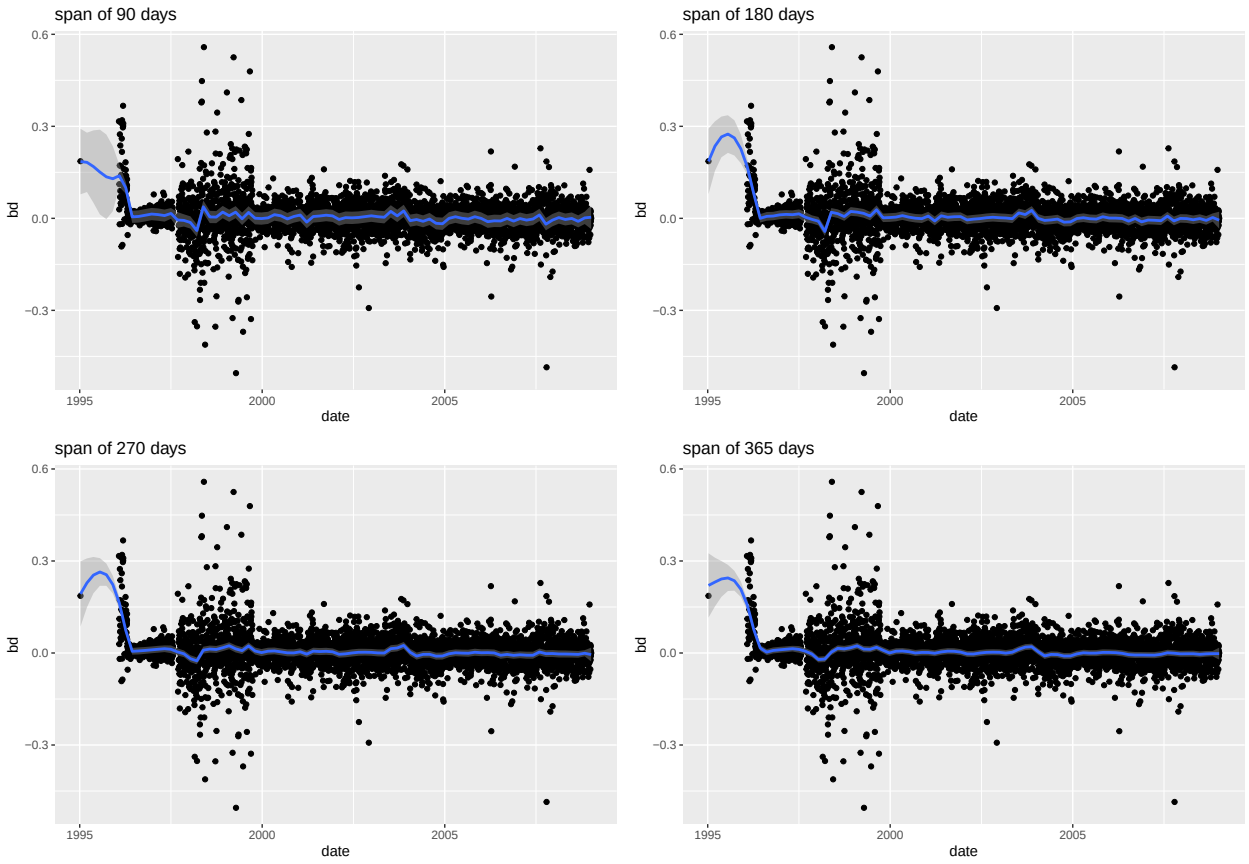
To see if slightly lowering λ changed the smooth models the same plotting process used to check Model 2b1 was repeated with the λ = 5.09.

```r
l <- lambda_2b2
# the mu
base_avg <- mean(edx_train$rating)
# movie effect
movie_avg <- edx_train %>%
  group_by(movieId) %>%
  summarize(bi = sum(rating - base_avg)/(n() + l))
# user effect
user_avg <- edx_train %>%
  left_join(movie_avg, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(bu = sum(rating - base_avg - bi)/(n() + l))
# date effect; will smooth
date_avg <- edx_train %>%
  left_join(movie_avg, by = 'movieId') %>%
  left_join(user_avg, by = 'userId') %>%
  group_by(date) %>%
  summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))

spans <- c(30, 60, 90, 180, 270, 365)

plots <- for (s in spans) {
  plot <- date_avg %>%
    ggplot(aes(x = date, y = bd)) +
    geom_point() +
    geom_smooth(method = 'loess',
                span = s / as.numeric(diff(range(edx_train$date)))) +
    ggtitle(paste('span of', as.character(s), 'days', sep = ' '))
  print(plot)}
```





Figures 10a-e: loess functions of average rating residuals over date. Residuals were grouped by date, then averaged with regularization (lambda = 5.09). Spans of 30, 60, 90, 180, 270, and 365 days were used.

Not much had changed, likely due to the small decrease in λ. However, if λ was significantly decreased, one would expect more severe undersmoothing at the lower span lengths since weaker regularization leads to more inclusion of signal noise in the date-grouped average ratings. Also, the smooth function mostly remained around 0 on the y-axis (save for a small region around 1995, which has a gap in data points), suggesting that submission dates were a weak predictor of the scores given in ratings.

A comparison of Two Different Date Models

```
# a quick comparison of different time effect models
data.table(time_effects = c('year only', 'year and month'),
           grouped_means = c(train_rmse_2a1, train_rmse_2a2),
           smoothed = c(train_rmse_2b2, train_rmse_2b1))
```

time_effects	grouped_means	smoothed
year only	0.8641273	0.8641094
year and month	0.8641242	0.8640329

Table 7: RMSE comparisons of date effects modeled through regularized grouped means versus smoothed loess functions. For the smoothed models, 'year only' and 'year and month' correspond to spans covering 365 and 30 days, respectively.

Given the same level of detail (year only versus year and month), the smooth model provided a lower RMSE than its grouped-means counterpart. Also, the smooth date model with a yearly span produced a lower RMSE than both versions of the grouped-means model. While the smooth model with monthly span had the lowest RMSE of the four, it was likely undersmoothed and therefore overfitted. As such, Model 2b2 (smooth date function, yearly span) was chosen to be further built on with genre effects.

Iteration 3a: Grouped Means of Genre Combinations

```
# Iteration 3a: Genre groups avg
span <- 365 / as.numeric(diff(range(edx_train$date)))

lambdas <- seq(0, 10, 1)
lambdas <- seq(4, 6, 0.1)
lambdas <- seq(4.95, 5.15, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
  # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # date effect; will smooth
  date_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(date) %>%
    summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))
  #modeling a smooth function out of the date effect
  date_effect_model <- loess(bd ~ as.numeric(date),
                             data = date_avg,
                             span = span)
  # predict using model on train set...
  train_date_preds <- predict(date_effect_model, edx_train$date)
  # then subtract predictions to get train set residuals not explained by model
  # genre group effect
  genres_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(bd = train_date_preds) %>% # rows already correspond to edx_train
    group_by(genres) %>%
    summarize(bgg =
                sum(rating - base_avg - bi - bu - bd)/(n() + l))
  # fitting (for each given lambda l)
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
```

```
    left_join(user_avg, by = 'userId') %>%
    # fit model on test set for results, same as in 2b1/2b2
    mutate(bd_test = predict(date_effect_model, date)) %>%
    left_join(genres_avg, by = 'genres') %>%
    mutate(pred = base_avg + bi + bu + bd_test + bgg) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, edx_test$rating))
})

plot(lambdas, rmses) # tuning only
```

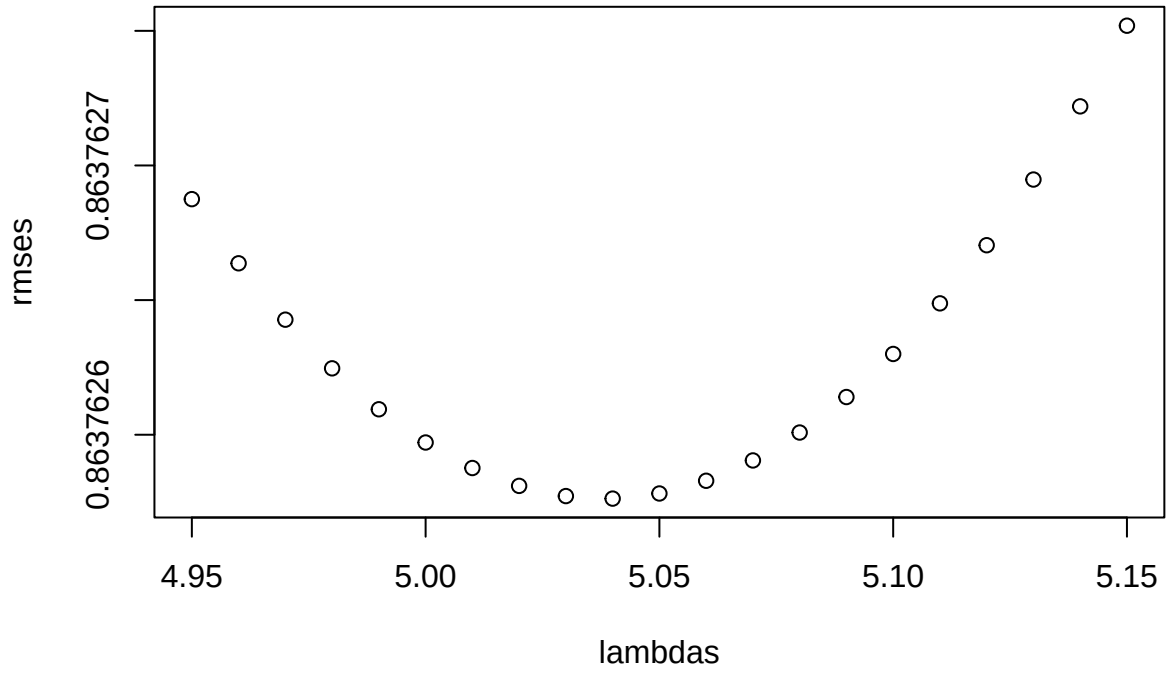


Figure 11: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 3a

```
lambda_3a <- lambdas[which.min(rmses)]
train_rmse_3a <- min(rmses)
```

Model 3a returned an RMSE of 0.8637625, with a λ of 5.04.

Iteration 3b: Grouped Means of Separate Genres

Creating the Genre Matrices

The functional code chunk for generating the matrices has been hidden using the R markdown option include = FALSE. Refer to the Methods section for a display of, and explanation on, the code used.

```
head(edx_train_genres, 3)
```

```
##      Drama Comedy Action Thriller Adventure Romance Sci-Fi Crime Fantasy
## [1,] FALSE   TRUE  FALSE    FALSE     FALSE    TRUE  FALSE FALSE   FALSE
## [2,]  TRUE  FALSE   TRUE     TRUE     FALSE   FALSE   TRUE FALSE   FALSE
## [3,] FALSE  FALSE   TRUE    FALSE      TRUE   FALSE   TRUE FALSE   FALSE
##      Children Horror Mystery   War Animation Musical Western Film-Noir
## [1,]    FALSE  FALSE   FALSE FALSE     FALSE   FALSE   FALSE     FALSE
## [2,]    FALSE  FALSE   FALSE FALSE     FALSE   FALSE   FALSE     FALSE
## [3,]    FALSE  FALSE   FALSE FALSE     FALSE   FALSE   FALSE     FALSE
##      Documentary  IMAX (no genres listed)
## [1,]       FALSE FALSE              FALSE
## [2,]       FALSE FALSE              FALSE
## [3,]       FALSE FALSE              FALSE
```

As expected, the matrix contained logicals and 20 columns.

Tuning the Model

```
# Iteration 3b: Genres examined separately
span <- 365 / as.numeric(diff(range(edx_train$date)))

lambdas <- seq(0, 6, 1)
lambdas <- seq(4.5, 5.5, 0.1)
lambdas <- seq(5.05, 5.15, 0.01)

rmses <- sapply(lambdas, function(l) { # do everything below for each lambda
  # the mu
  base_avg <- mean(edx_train$rating)
  # movie effect
  movie_avg <- edx_train %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # date effect; will smooth
  date_avg <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(date) %>%
    summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))
  #modeling a smooth function out of the date effect
  date_effect_model <- loess(bd ~ as.numeric(date),
                             data = date_avg,
                             span = span)
  # predict using model on train set...
  train_date_preds <- predict(date_effect_model, edx_train$date)
  # then subtract predictions to get train set residuals not explained by model
```

```
  # residuals for genre avg computations
  resids <- edx_train %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(bd = train_date_preds) %>% # rows already correspond to edx_train
    mutate(resids = rating - base_avg - bi - bu - bd)

  # find genre avgs of residual variability
  genre_avgs <- sapply(genre_list, function(genre) {
    resids %>%
      # filter for rows in train set AND genre = T
      filter(edx_train_genres[,genre]) %>%
      # regularized average resid by genre
      summarize(avg = sum(resids) / (n() + l))})
  # fitting without the genre effects/resids
  pred <-
    edx_test %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(bd_test = predict(date_effect_model, date)) %>% # same as in 2b1/2b2
    mutate(pred = base_avg + bi + bu + bd_test) %>%
    pull(pred)
  # adding average genre effects
  for (g in genre_list) {
    # genre effect avg, searched from corresponding genre position
    # in the vector genre_list
    bg <- as.numeric(genre_avgs[which(genre_list == g)])
    # add that avg to positions of predicted values vector (pred)
    # corresponding to edx_genres$g == T,
    pred[edx_test_genres[,g]] <-
      pred[edx_test_genres[,g]] + bg}
  # grab RMSE
  return(RMSE(pred, edx_test$rating))})

plot(lambdas, rmses) # tuning only
```

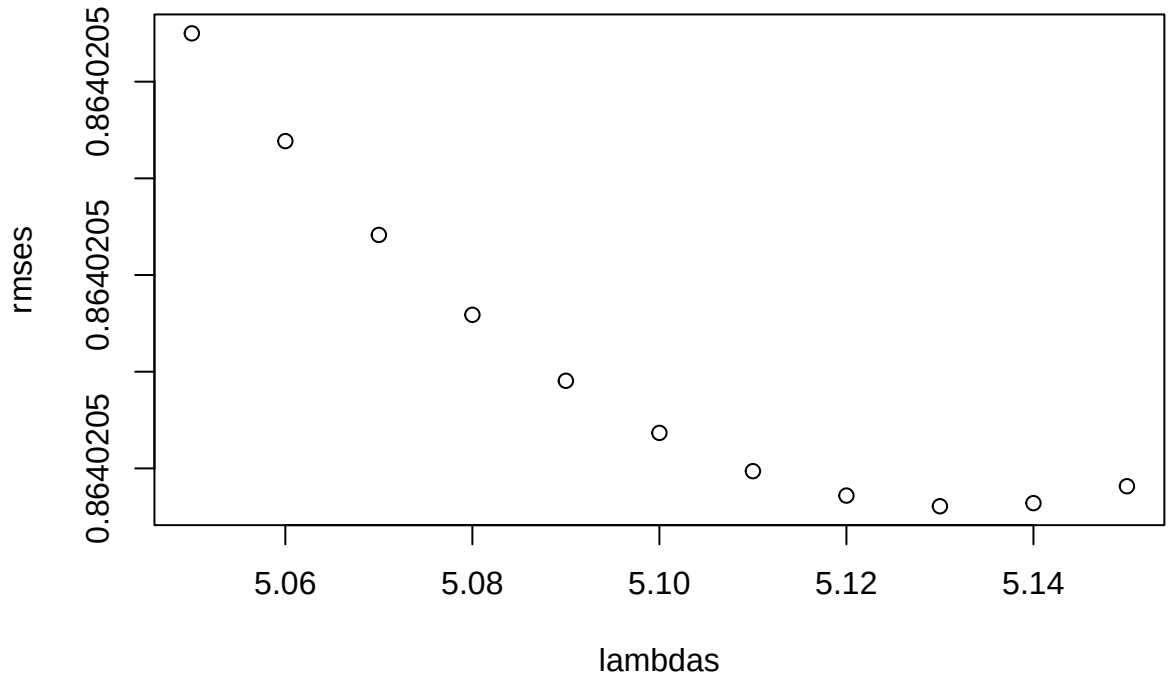


Figure 12: A plot of RMSEs and associated lambda regularization terms during the final stage of tuning model 3b

```
lambda_3b <- lambdas[which.min(rmses)]
train_rmse_3b <- min(rmses)
```

Model 3b returned an RMSE of 0.8640205, with a λ of 5.13.

Despite its simplistic approach to classifying genres, Model 3a produced a lower RMSE than model 3b. This was likely due to some interaction effects existing between genres, which model 3b could not capture by its design of separately computing each averaged genre effect.

Comparison of Model Iterations

```
data.table(effects = c('model 1 (movie and user effects)',
                       'model 2b2 (movie, user, and date effects)',
                       'model 3a (movie, user, date, and genre group effects)'),
           RMSE = c(train_rmse_1,
                    train_rmse_2b2,
                    train_rmse_3a),
           lambda = c(lambda_1,
                      lambda_2b2,
                      lambda_3a))
```

effects	RMSE	lambda
model 1 (movie and user effects)	0.8641361	4.94
model 2b2 (movie, user, and date effects)	0.8641094	5.09
model 3a (movie, user, date, and genre group effects)	0.8637625	5.04

Table 8: A comparison of RMSEs and lambdas with models of increasing complexity.

As expected, including more predictors decreased the RMSE. The inclusion of date effects as a smooth function only decreased RMSE by about 2.67×10^{-5} from model 1 (movie and user effects only), while the inclusion of genre groups decreased RMSE by about 3.47×10^{-4} from model 2b2 (movie, user, and smoothed time function with yearly span) and 3.74×10^{-4} from model 1. This discrepancy in RMSE improvements revealed that when building upon a model that already accounted for movies and users, genres were a more effective predictor of movie ratings than the dates on which those ratings were uploaded.

Meanwhile, the optimal λ value increased by 0.15 from model 1 to 2b2, but decreased by 0.05 from 2b2 to 3a. Since the purpose of λ is to penalize extreme values coming from small sample sizes, the most likely explanation was that the date effect smooth model used for 2b2 had more such extreme values than the genre combinations under the `genres` column in `edx`. While this project did not examine the distributions of ratings under each value in the `genres` column, graph series 8 and 10 in the Results section did reveal a section of relatively extreme fluctuation in average rating effects -and sparse data points- a few years following 1995.

As the iteration with the lowest RMSE, model 3a was chosen for the validation run.

Validation Run

The chosen model, 3a, was re-trained on the entire `edx` dataset using the same λ, and used to predict ratings in the `validation` set for a final RMSE evaluation.

```
# validation run:
# Iteration 3a; now using edx as train set and validation as "test" set
span <- 365 / as.numeric(diff(range(edx_train$date)))

rmse_final <- function(l = lambda_3a) {
  # the mu
  base_avg <- mean(edx$rating)
  # movie effect
  movie_avg <- edx %>%
    group_by(movieId) %>%
    summarize(bi = sum(rating - base_avg)/(n() + l))
  # user effect
  user_avg <- edx %>%
    left_join(movie_avg, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(bu = sum(rating - base_avg - bi)/(n() + l))
  # date effect; will smooth
  date_avg <- edx %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    group_by(date) %>%
    summarize(bd = sum(rating - base_avg - bi - bu)/(n() + l))
  #modeling a smooth function out of the date effect
  date_effect_model <- loess(bd ~ as.numeric(date),
                             data = date_avg,
```

```
                              span = span)
  # predict using model on train set (edx)...
  train_date_preds <- predict(date_effect_model, edx$date)
  # then subtract predictions to get edx residuals not explained by model
  # genre group effect
  genres_avg <- edx %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    mutate(bd = train_date_preds) %>% # rows already correspond to edx
    group_by(genres) %>%
    summarize(bgg =
                sum(rating - base_avg - bi - bu - bd)/(n() + l))
  # fitting (for each given lambda l)
  pred <-
    validation %>%
    left_join(movie_avg, by = 'movieId') %>%
    left_join(user_avg, by = 'userId') %>%
    # fit model on dates in test set (validation) for results
    mutate(bd_test = predict(date_effect_model, date)) %>%
    left_join(genres_avg, by = 'genres') %>%
    mutate(pred = base_avg + bi + bu + bd_test + bgg) %>%
    pull(pred)
  # grab RMSE
  return(RMSE(pred, validation$rating))}

rmse_final(lambda_3a)
```

```
## [1] 0.8644012
```

```
final_rmse <- rmse_final(lambda_3a)
```

The final RMSE was 0.8644012, higher than the 0.8637625 derived from the training run, but still beneath the threshold of 0.86490 for this project. This was not surprising, as the model was optimized on `edx` rather than `validation`.

Conclusion

Limitations and Future Inquiries

Limitations of the Model Used

Despite a general trend of improved accuracy with more predictor variables included, the regression-based Normalization of Global Effects model built upon in this project produced relatively small improvements (in terms of RMSE decreases) with increasing complexity. Each subsequent effect term b_x modeled the *residuals* left over from the previous terms, and the first term (the movie effect term b_i) itself modeled rating deviations from the base average μ. A consequence of this model structure is that predictors added in later in the linear combination would yield reduced predictive power (versus if the same predictor was placed earlier) due to having less variation left to "explain", regardless of its actual correlation with the response variable. Placing more potent predictors in front of the model (for instance, genre combinations before dates) could mitigate the problem, but finding the optimal arrangement of predictors would entail large amounts of additional run time for the already time-consuming code written in this project.

Another limitation of the model used in this project was an inability to account for interaction effects, both between different predictors and different values within the same predictor. Model 3a (which computed group means for genre

combinations) outperformed model 3b (which treated each genre as a dummy variable independent from all others). This implied an interaction effect between different genres, which was rather crudely represented in 3a but outright ignored in 3b. In addition, real-life movie watchers tend to have preferences for different types of movie based on genres, time period of release, or other factors. By simply modeling user effects as various values of b_u, the average "harshness" of each reviewer across all submitted ratings was computed without consideration for individual preferences.

The Tuning Process

For each iteration of the model tuned in this project, only one λ was selected. This made each tuning attempt a compromise between the various predictor effects b_x, as each effect had different optimal amounts of regularization, but only one regularization term value could be used across all of them. Assigning a different λ for each b_x is theoretically possible, but also time-consuming. In addition, since the split between training and testing datasets is a (pseudo-)random process, any training RMSEs produced in a tuning process are effectively sample statistics, subject to sampling error. K-fold cross-validation and other techniques that assess average model accuracy through multiple train-test splits are thus another possible way to improve tuning effectiveness; this project deliberately avoided using them (see Methods section), but they would have been preferable alternatives if more RAM was available.

Alternative Algorithms

Other machine learning algorithms could more effectively predict movie ratings using the variables given in the MovieLens dataset.Singular Value Decomposition (SVD), used by winners of the original challenge, is a matrix factorization approach that utilizes linear algebra to model the most important features within each predictor as a matrix of latent features, then predicts movie ratings though computing the dot product of two such matrices (from movie and user IDs, in the original challenge). Principal Component Analysis (PCA) is another matrix factorization technique, and reduces the dimensionality of data with many predictor variables down to a smaller collection of principal components (PCs); those PCs represent latent features in the data that explain most of the outcome variance. The regression tree is another algorithm that could be used, and could handle multiple predictor variables. While regression trees produce models that are intuitive for the human eye, their sensitivity to even small changes in the data structure often results in low accuracy. The random forest method mitigates the issue by aggregating predictions from multiple trees, raising accuracy at the expense of computational time and model readability.

As previously mentioned in the Methods section, all of the models above are more computationally intensive than the Normalization of of Global Effects approach used in this project. While many R packages (such as ranger for random forests) exist to run machine learning algorithms more efficiently, investing in more RAM and/or GPU would be worthwhile for further inquiries in machine learning.

Project Summary

Building on the regularized Normalization of Global Effects approach with movie and user effects (featured in course 8 of the EdX Harvard Data Science program), this project has built a predictive model that included the effects of movie ID, user ID, date of rating submission, and genre combinations on movie ratings. A movie's combination of genres was found to be a more effective predictor of rating scores than the date on which a reviewer rated a movie. Increasing the amount of predictors always lowered the RMSE, while the optimal regularization term λ was determined by the presence of extreme values from small groups during the calculation of predictor effects. The optimized model produced a final RMSE of ~0.8644. More complex tuning techniques and algorithms could yield better accuracy, but would also require more computational power and time.

Citations

All sources, both within and outside EdX, have been linked in the text referencing them. Below is a list of sources referenced in this project, in order of appearance:

Harvard's Professional Certificate in Data Science program on EdX, taught by Dr. Rafael Irizarry

Netflix Recommendations: Beyond the 5 stars (Part 1) - A Netflix TechBlog post by Xavier Amatriain and Justin Basilico

The MovieLens download page on grouplens.org

Smoothing Techniques - A guide by FrontlineSolvers. Inc

Definition of IMAX, on Collins Online Dictionary

Chapter 34.7 ("Recommendation systems") of the Data Science Program online textbook, written by Dr. Irizarry

Winning the Netflix Prize - A Summary - Blog post by Edwin Chen

What are RMSE and MAE? - posted by Shwetha Acharya on Towards Data Science

8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset - Posted by Jason Brownlee on Machine Learning Mastery

Chapter 34.9 ("Regularization") of the Data Science Program online textbook, written by Dr. Irizarry

Understanding 8 types of Cross-Validation - posted by Satyam Kumar on Towards Data Science

Chapter 34.7.4 ("A first model") of the Data Science Program online textbook, written by Dr. Irizarry

Set Seed in R - from R CODER, created by José Carlos Soage

Dummy Variables - part of the Research Methods Knowledge Base, written by Dr. William M.K. Trochim; hosted by Conjoint.ly

Matrix Factorization Techniques for Recommender Systems - Scholarly Article by Yehuda Coren, Robert Bell, and Chris Volinsky; published by the IEEE Computer Society

The Netflix Prize and Singular Value Decomposition - Online Course Material from the Ney Jersy Institute of Technology

Understanding matrix factorization for recommendation (part 1) - preliminary insights on PCA - blog post by Nicholas Hug

What is a Latent Variable? - blog post by Tim Bock on displayr.com

A Step-by-Step Explanation of Principal Component Analysis (PCA) - posted by Zakaria Jaadi on builtin.com

Classification and Regression Trees - article published by Martin Krzywinski and Naomi Altman on Nature Methods]

Random Forest - article on IBM Cloud Learn Hub

RDocumentation Page for the ranger function

What to consider when choosing a laptop for machine learning - posted by Eugene Oduma on Zindi