MovieLens Analysis

Ananthajit Srikanth

Introduction

This project serves to build upon Rafael Irizarry's movie recommendation system. This is also the Capstone Project for the ninth course in the HarvardX Data Science series of courses.

Github link: https://github.com/Ant-28/movielens/

About this dataset

The dataset being used in this course is the Movielens 10M dataset, containing 10 million rows of movie ratings. The dataset contains the following information:

userId: A number used to classify ratings by a given user. movieId: A number used to classify ratings for a given movie. title: The title of the movie genres: The genres of the movie timestamp: The time (in seconds, from 1/1/1970 UTC) at which the rating was made. rating: The rating given by a user for a movie.

A warning about system requirements

This code was tested on a system with 8 GB of RAM. As R performs all of its operations and stores datasets within RAM, the code may not scale well to systems with less RAM. This code was run on R 4.1.0.

Background information

Before using the MovieLens 10M dataset, it has to be converted into a training and test set.

Creating the datasets

The dataset is converted into a training set, called *edx* and a test set, called *validation.*

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

# Note: this process could take a couple of minutes

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

```
## Loading required package: tidyverse

## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.2     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()

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

## Loading required package: caret

## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift

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

## Loading required package: data.table

##
## Attaching package: 'data.table'

## The following objects are masked from 'package:dplyr':
##
##     between, first, last

## The following object is masked from 'package:purrr':
##
##     transpose

library(tidyverse)
library(caret)
library(data.table)

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

dl <- tempfile()
download.file("http://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 3.6 or earlier:
# movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[
    movieId],
#                                            title = as.character(title),
#                                            genres = as.character(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)`

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

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)

## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")

edx <- rbind(edx, removed)

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

Creating The Datasets - Splitiing the *edx* set

The models will be trained on the *edx* set. However, to test the effectiveness of models and to debug any potential issues, the *validation* set will not be used until the final model is made and tested. Hence, the *edx* set will be split into 3 parts: 1. *edx_train* 2. *edx_validation* 3. *edx_debug*

edx_train serves as a training set, *edx_validation* is a validation set that shall be used when k-fold cross validation is not applicable (for example: when selecting a regularization parameter), and *edx_debug* is a debugging set. It is similar to a test set, for computing RMSE, however it is used in this case to 'debug' the model (and compare multiple models) as the *validation* set is used only for final testing.

Similar to how the *edx* and *validation* set were created, the three datasets will be split as follows:

First the required libraries will be loaded using a for loop (this allows required libraries to be stored as a character vector and easily loaded):

```
# Loading the required libraries, can add required libraries later
required_libraries <- c("caret", "matrixStats", "tidyverse", "knitr", "lubridate",
    "broom", "ranger", "knitr", "rmarkdown")
# The for loop installs and loads libraries one by one from required_libraries
for(i in 1:length(required_libraries)){
if(!require(required_libraries[i], character.only = TRUE)){
  install.packages(required_libraries[i], repos = "http://cran.us.r-project.org")
  library(required_libraries[i], character.only = TRUE)
}
  else{
require(required_libraries[i], character.only = TRUE)}}

## Loading required package: matrixStats

##
## Attaching package: 'matrixStats'

## The following object is masked from 'package:dplyr':
##
##     count

## Loading required package: knitr

## Loading required package: lubridate

##
## Attaching package: 'lubridate'

## The following objects are masked from 'package:data.table':
##
##     hour, isoweek, mday, minute, month, quarter, second, wday, week,
##     yday, year

## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union

## Loading required package: broom

## Loading required package: ranger

## Loading required package: rmarkdown

# Run this line only if necessary:
# update.packages()
```

Once the packages are loaded, the dataset is split into *edx_train* and a temporary dataset *edx_foo*, which will be used to create the validation and test sets. set.seed is used for replicability of results.

```
# Splits the edx set into training and validation+debug sets
# Set seed so that results are replicable
set.seed(2000, sample.kind = "Rounding")

## Warning in set.seed(2000, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used

#Dataset creation: This is similar to how the edx and validation sets are created

edx_index <- createDataPartition(y = edx$rating, times = 1, p = 0.2, list = FALSE)
#edx_train is the training set
edx_train <- edx[-edx_index,]
#edx_foo is a temporary set which needs to be modified
edx_foo <- edx[edx_index,]
```

The validation and test sets can only be used if they contain movie IDs from the training set. So edx_foo must be modified accordingly. The results are stored in *edx_temp*.

```
# Remove all rows from edx_foo that aren't in edx_train as
# a show cannot be recommended without prior knowledge of the user
# and movie biases
edx_temp <- edx_foo %>% semi_join(edx_train, by = "movieId") %>%
  semi_join(edx_train, by = "userId") %>% as.data.frame()
# edx_removed includes items in edx_foo that are not in edx_train
# if you have the latest version of tidyverse with dtplyr use the as.data.frame
    command
```

Columns that are in *edx_foo* but not in *edx_temp* are saved in *edx_removed*, which is then added to *edx_train*.

```
edx_removed <- anti_join(edx_foo, edx_temp) %>% as.data.frame()

## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")

#edx_removed is added to edx_train otherwise it would go to waste.
edx_train <- bind_rows(edx_train, edx_removed)
```

Then, *edx_temp* is split equally into two sets: *edx_validation* and *edx_debug*.

```
# Split edx_temp into two halves, the first half is used for crossvalidation,
# Such as choosing regularization parameters while the other half acts like a
# 'debug' set, which is a type of test set to see the efficacy of the algorithm
# before training on the entire edx set and testing on validation.

# Again, set.seed for consistency
set.seed(5000, sample.kind = "Rounding")

## Warning in set.seed(5000, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used

debug_ind <- createDataPartition(y = edx_temp$rating,
                                 time = 1, p = 0.5, list = FALSE)
#creating the validation and debug sets
edx_validation <- edx_temp[-debug_ind,]
edx_debug <- edx_temp[debug_ind,]
```

Verifying that the splitting process was successful:

To confirm that no rows are lost, misarranged, the following steps are taken: * First the three datasets are joined together into a data frame called *edx_extra*. It is then ordered by user and movie IDs since those are unique identifiers. The *edx* set is also sorted similarly. This is so that they can be compared using the identical function:

```
# check if the the split datasets and edx contain the exact same elements
#edx_extra contains the split datasets combines
edx_extra <- bind_rows(edx_train, edx_validation,edx_debug)
# order by user and movie ID as those are the only unique orderings
# since users wouldn't rate the same movie twice
order1 <- order(edx_extra$userId, edx_extra$movieId)
order2 <- order(edx$userId, edx$movieId)
# ordered versions of edx_extra and edx for identical to work
edx_extra_sorted <- edx_extra[order1,]
edx_sorted <- edx[order2,]

identical(edx_extra_sorted, edx_sorted)
```

```
## [1] TRUE
```

The result should be TRUE, suggesting that the join worked.

As R performs all tasks in RAM, it is a good idea to run the gc function while performing tasks with large datasets. The extra datasets which were used in intermediate steps will also be removed.

```
# Clear memory because R isn't good at it, you'll see this quite a few times
# across the code

gc()

##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    2381135  127.2    7764989  414.7  22283954 1190.1
## Vcells  283604658 2163.8  566513144 4322.2 563671916 4300.5

# Remove redundant tables
remove(edx_extra, edx_sorted, edx_index, edx_removed, edx_extra_sorted, edx_foo,
    edx_temp, order1, order2)
```

Now, I shall proceed with inital analysis techniques.

Inital Analysis

The first techniques used in Professor Irizarry's system is to model movie and user effects.

Mean Rating

The *edx_train* dataset can be used to predict ratings by finding the mean of all the movies in the dataset. The quality of this model is then tested using *caret*'s RMSE function.

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

# first prediction
only_the_mean <- edx_debug %>% mutate(pred = mu) %>% .$pred

rmse0 <- RMSE(only_the_mean, edx_debug$rating)
rmse0

## [1] 1.060126
```

This is not a good result, as there is an average error of one star.

Movie Effects

The model can now be expanded to:

$$R = \mu + b_i + \epsilon$$

Where R is the rating, μ is the average rating, b_i is the bias towards a certain movie and ϵ is variability due to randomness.

This is performed by grouping ratings by movie, and subtracting the mean from each rating. The average deviation is calculated by movie.

```
# Adding movie effects
movie_effects <- edx_train %>% group_by(movieId) %>%
  summarise(effect_bi = mean(rating - mu))

mean_and_movie <- mu + edx_debug %>%
  left_join(movie_effects, by = 'movieId') %>%
   pull(effect_bi)
```

```
rmse1 <- RMSE(mean_and_movie, edx_debug$rating)
rmse1

## [1] 0.9441217
```

User Effects

The initial model also models user effects as follows:

$$R = \mu + b_i + b_u + \epsilon$$

where b_u is the user bias. This can be calculated by finding the deducting mean and movie effects from the rating, grouping by user and finding the mean deviation.

```
# Adding user effects
user_effects <- edx_train %>% left_join(movie_effects, by = "movieId") %>%
  group_by(userId) %>%
  summarize(effect_bu = mean(rating-mu-effect_bi))

mean_movie_user <- mu + edx_debug %>% left_join(movie_effects, by = "movieId") %>%
  left_join(user_effects, by = "userId") %>% mutate(total_effect = effect_bi +
      effect_bu) %>%
  pull(total_effect)

rmse2 <- RMSE(edx_debug$rating, mean_movie_user)
rmse2

## [1] 0.8665775
```

It can be observed that the RMSE has considerably decreased. Using these models, I shall now present additional techniques for movie recommendation.

Analysis

The main focus of this analysis is genre effects. As there are multiple genres that one movie can have, it will be assumed that genre effects are independent of one another.

Genre Effects

The model can now be extended to consider the effect of genres. It can be assumed that each genre works independently of one another, so the formula is as follows:

$$R = \mu + b_i + b_u + \sum_{p=1}^{n} (b_g)_p + \epsilon$$

Where $\sum_{p=1}^{k} (b_g)_p$ is the sum of all genre effects, $(b_g)_1 + (b_g)_2 + (b_g)_3 + ... + (b_g)_k$, assuming the movie has k genres.

To understand how to approach this method, the genres column has to be analysed.

```
edx_train$genres %>% head(5)

## [1] "Comedy|Romance"                    "Action|Drama|Sci-Fi|Thriller"
## [3] "Action|Adventure|Sci-Fi"           "Action|Adventure|Drama|Sci-Fi"
## [5] "Children|Comedy|Fantasy"
```

As can be seen, the genres are separated by a pipe ("|"). To find the maximum number of genres in any one entry of the training set, the str_count from the stringr package. 1 is added to the number of pipes, as one pipe separates two genres, so n pipes separate n + 1 genres.

```
maxgenre <- 1 + str_count(edx_train$genres, "\\|") %>% max()
maxgenre
```

```
## [1] 8
```

Using this, a set of columns are made for each genre.

```
# paste0 concatenates strings without any space.
genrecols <- paste0("genre",as.character(1:maxgenre))
```

A new dataset is made, called, *edx_train_genres* where each genre is separated into a new column. As there are 8 columns, it will be separated as columns **genre1, genre2, . . . , genre8**. This is performed using the separate function. Note that for ease of retrieval the movie and user biases are added to the *edx_train_genres* column in advance:

```
# invisible function hides gc output
invisible(gc())
#this takes a while
edx_train_genres <- edx_train %>% separate(col = genres, into = genrecols, sep = "
    \\|")
```

```
## Warning: Expected 8 pieces. Missing pieces filled with `NA` in 7199884 rows [1,
    2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
invisible(gc())

edx_train_genres <- edx_train_genres %>% left_join(movie_effects, by = "movieId")

invisible(gc())

edx_train_genres <- edx_train_genres %>% left_join(user_effects, by = "userId")
```

gc() is used to lower ram usage.

For ease of understanding in summaries, two additional vectors will be made for columns:

```
biascols <- paste0("effect_bg", as.character(1:(maxgenre)))
numcols <- paste0("n",as.character(1:(maxgenre)))
```

To find the bias for each genre column, the structure of *edx_train_genres* must be considered.

```
str(edx_train_genres)
```

```
## Classes 'data.table' and 'data.frame':   7200087 obs. of  15 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : num  122 292 316 329 355 362 364 370 377 420 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983421 838983392 838983392 838984474 838984885
    838983707 838984596 838983834 838983834 ...
##  $ title    : chr  "Boomerang (1992)" "Outbreak (1995)" "Stargate (1994)" "Star
    Trek: Generations (1994)" ...
##  $ genre1   : chr  "Comedy" "Action" "Action" "Action" ...
##  $ genre2   : chr  "Romance" "Drama" "Adventure" "Adventure" ...
##  $ genre3   : chr  NA "Sci-Fi" "Sci-Fi" "Drama" ...
##  $ genre4   : chr  NA "Thriller" NA "Sci-Fi" ...
## $ genre5    : chr  NA NA NA NA ...
## $ genre6    : chr  NA NA NA NA ...
## $ genre7    : chr  NA NA NA NA ...
## $ genre8    : chr  NA NA NA NA ...
## $ effect_bi: num  -0.6569 -0.0944 -0.1636 -0.1762 -1.0354 ...
## $ effect_bu: num  1.71 1.71 1.71 1.71 1.71 ...
```

As can be seen, the genres are split into 8 columns, from column 6 to column 13.

To provide a system that is less hard-coded to the number of columns, and given that the genre columns are right after one another (this holds true due to the nature of the separate function), the genre biases can now be calculates. As the same genre can be in different columns, a weighted average has to be taken. Before doing so however, the *edx_train_genres* dataframe can be grouped by each genre column. Then, the genre bias, for each genre column (genre1, genre2, . . . , genre8) can be stored in a list, using a for loop:

```
ind <- -1 + which(colnames(edx_train_genres) == 'genre1')
genre_effects <- list()
for(i in 1:maxgenre){

  genre_effects[[i]] <- setNames(edx_train_genres %>% group_by_at(ind+i) %>%
                                  summarize(effect_bg = mean(rating - mu - effect_
                                     bi - effect_bu), n = n()),
                                c(genrecols[i],biascols[i], numcols[i]))

}
```

A list called genre_effects is created.

The for loop creates 8 list items, each containing a summary table. For each number i, the ith item of the list is *edx_train_genres* grouped by the ith genre column and summarized to provide the genre biases.

The variable ind represents the column that precedes the genre columns. So, ind+i signifies the ith genre column. For example, if *ind* equals 5, ind+1 is 6, which is the index of the first genre column. group_by_at groups by this column.

Then, for each genre column, the genre bias is calculated as the average deviation from the rating, after considering mean, movie and user effects. i.e., it is the average of rating - mean - movie effect - user effects . The frequency of movies that contain this genre (in the genre column that is being used as a grouping variable) are also calculated and stored in n. Then they are named similar to the genre columns , using the vectors biascols and numcols respectively.

The setNames function sets the names of the columns in the order in which they are provided.

For example, the bias column for genre1 is effect_bg1 and the number of movies that contain a given genre in genre1 is n1. This columns are names using the setNames function. The column nomenclature will be used for joining the summary tables together.

Joing the genre tables together

As the maximum possible number of genres in the training set is 8 columns, there are 8 genre tables. However, the same genre can be in two different genre tables and have different effects. Hence, a weighted average must be taken. To do so, the eight tables have to be joined together by genre.

The tables have to be joined recursively (i.e., the first two tables are joined together. This joined table is joined to the third and so on. While a right join could be used, only the first table has all of the genres (some genres are not present in genre2. . . genre8 at all, since genres are stored in alphabetical order.))

As these are list elements and left_join does not support lists, a modified version of left_join that accepts list arguments must be used:

```
join_genres <- function(...){
  df1 <- list(...)[[1]]
  df2 <- list(...)[[2]]
  # Left join a la merge
  merge(df1, df2, by.x = 1, by.y = 1, all.x = TRUE)

}
```

merge is used instead of left_join as it supports column indexing. As the genre column is the first column in all tables, by.x and by.y are 1 (this represents the columns to join by; x and y are the first and second column respectively). Column indexes are used instead of column names since the column names are different. The argument all.x = TRUE means all rows/columns from x are kept (which is what left_join does), and y is joined to x using the joining column. This is then used with the Reduce function to join the tables recursively. The joined table is stored in *total_genre_effects*

```
total_genre_effects <- Reduce(join_genres, genre_effects)
```

The structure of *total_genre_effects* is as follows:

```
head(total_genre_effects)
```

```
##                genre1   effect_bg1      n1   effect_bg2     n2   effect_bg3
## 1 (no genres listed)  0.263161452       6           NA     NA           NA
## 2              Action -0.012571297 2048028           NA     NA           NA
## 3           Adventure -0.010201833  602585 -0.017942794 924141           NA
## 4           Animation -0.014565210  174679 -0.017848698 173198  0.009246025
## 5            Children -0.026968224  145395 -0.025170392 271530 -0.020986385
## 6              Comedy  0.001185796 1949926 -0.006173077 489582 -0.011054748
##       n3  effect_bg4     n4 effect_bg5   n5 effect_bg6 n6 effect_bg7 n7
## 1     NA          NA     NA         NA   NA         NA NA         NA NA
## 2     NA          NA     NA         NA   NA         NA NA         NA NA
## 3     NA          NA     NA         NA   NA         NA NA         NA NA
## 4  25989          NA     NA         NA   NA         NA NA         NA NA
## 5 166526  0.00599603   7236         NA   NA         NA NA         NA NA
## 6 260145 -0.02111642 126176 0.01319045 6611         NA NA         NA NA
##   effect_bg8 n8
## 1         NA NA
## 2         NA NA
## 3         NA NA
## 4         NA NA
## 5         NA NA
## 6         NA NA
```

From this table, multiple genres are not present in subsequent genre columns, so summaries result in an NA when computing values. Thus, To compute the weighted average NA columns have to be converted to zero.

```
total_genre_effects[is.na(total_genre_effects)] <- 0
```

To compute the weighted average, the following formula must be used:

$$\frac{\sum_{p=1}^{k}(b_g)_p \cdot n_p}{\sum_{p=1}^{k} n_p}$$

Where n is the number of columns containing the genre in a given genre column in *edx_train_genres*. The bias columns as well as the frequency columns' names are stored in biascols and numcols respectively. Using this, the column indices of the bias and frequency columns in *total_genre_effects* can be used to calculated the weighted average.

The ith bias and frequency column are multiplied together in a table called dat.

```
n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices<- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})
```

By looking at the structure of dat it can be seen that it contains 8 columns, one for each genre from *total_genre_effects*.

```
head(dat)

##               [,1]       [,2]       [,3]        [,4]     [,5] [,6] [,7] [,8]
## [1,]      1.578969      0.000     0.0000     0.00000  0.00000    0    0    0
## [2,] -25746.369244      0.000     0.0000     0.00000  0.00000    0    0    0
## [3,]  -6147.471832 -16581.672     0.0000     0.00000  0.00000    0    0    0
## [4,]  -2544.236357  -3091.359   240.2949     0.00000  0.00000    0    0    0
## [5,]  -3921.044986  -6834.516 -3494.7787    43.38727  0.00000    0    0    0
## [6,]   2312.213766  -3022.227 -2875.8375 -2664.38541 87.20205    0    0    0
```

This is then summarized in a data frame called *genre_bias*:

```
genre_bias <- data.frame(genre = total_genre_effects$genre1, effect_bg = rowSums(
    dat)/rowSums(total_genre_effects[, n_indices]))
```

The rowSums of *dat* is the weighted sum of the genre biases, while the rowSums of *total_genre_effects[, n_indices]* is the sum of the frequencies. This provides a weighted average for each genre.

The genre names are then added to the weighted average.

The unncessary tables are then removed:

```
remove(genre_effects, dat)
```

Joining the weighted genre effects to the respective genres in the debug set

To test the model on the debug set, the genres have to separated, similar to the training set. However, the number of genre columns may vary. First, the genre biases have to be joined to each genre column.

```
# Testing on the debug set

debug_genre_count <- 1 + str_count(edx_debug$genres, "\\|") %>% max()

genrecols_debug <- paste0('genre', 1:debug_genre_count)

edx_debug_genres <- edx_debug %>% separate(genres, genrecols_debug, sep = "\\|")

## Warning: Expected 8 pieces. Missing pieces filled with `NA` in 899962 rows [1,
    2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].

genrecols_debug

## [1] "genre1" "genre2" "genre3" "genre4" "genre5" "genre6" "genre7" "genre8"
```

debug_genre_count is equivalent to maxgenre, except for the debug set. *edx_debug_genres* contains 8 genre columns, same as *edx_train_genres*' 8.

Similar to creating the genre bias summaries on the training set, a matrix of

```
invisible(gc())
genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug))
```

genre_ind is the index for the column that precedes the genre columns. This will be used when joining genre biases to columns. A matrix *debug_genrebias* is initialized. Each column of the matrix represents one genre column from the debug set, and each row represents one testing example from *edx_debug*.

The for loop is used to find the genre bias for each genre column:

```
for(i in 1:debug_genre_count){

  foo <- merge(edx_debug_genres, genre_bias, by.x = (genre_ind + i), by.y = 1, all.x
       = TRUE)
 debug_genrebias[,i] <- foo$effect_bg
}
```

The above for loop groups by the genre_ind + i column in *edx_debug_genres*. As its structure is similar to *edx_train_genres*, the genre columns are placed in sequential order. So genre_ind + i gives the index of the ith genre column; the genre bias table is joined to this column (by genre) and the biases are pulled for each column. This is then stored in the ith column of *debug_genrebias*.

However, not every movie has 8 genres, as can be seen below:

```
invisible(gc())
# Contains NAs
noquote("NAs by genre column")

## [1] NAs by genre column

apply(debug_genrebias,2,function(x){sum(is.na(x))})

## [1]      0 174013 437641 709568 850016 892124 898772 899962

#Does not contain NAs
noquote(" ")

## [1]

noquote("Not NAs by genre column")

## [1] Not NAs by genre column

apply(debug_genrebias,2,function(x){sum(!is.na(x))})

## [1] 899985 725972 462344 190417  49969   7861   1213     23
```

As can be seen, multiple genre columns contain NAs (while some do not)

As an NA implies there is no genre in the respective columns, the genre bias should be zero.

```
debug_genrebias[is.na(debug_genrebias)] <- 0
```

Once that is completed, the sum of the rows can be taken to determine the overall genre effect for each movie in the debug set:

```
debug_genrebias <- rowSums(debug_genrebias)
```

Using the above model, the predictions can be made for the ratings and stored in a variable called *by_movie_user_and_genre*:

```
by_movie_user_and_genre <- mu + {edx_debug %>% left_join(movie_effects, by = "
    movieId") %>%
  left_join(user_effects, by = "userId") %>% mutate(total_effect = effect_bi +
      effect_bu) %>%
  pull(total_effect)} + debug_genrebias
```

The RMSE of this model can now be calculated:

```
rmse3 <- RMSE(edx_debug$rating, by_movie_user_and_genre)
rmse3
```

```
## [1] 0.8668083
```

The RMSE has increased! This suggests that the model has been overtrained.

The Problem

We can see what went wrong by looking at the genre biases with the frequencies:

```
# Looking at what went wrong
genre_bias %>% mutate(n = rowSums(total_genre_effects[,n_indices])) %>% arrange(
    desc(abs(effect_bg)))
```

```
##                  genre    effect_bg       n
## 1   (no genres listed)  0.263161452       6
## 2          Documentary  0.063413927   74454
## 3            Film-Noir  0.030544730   94992
## 4             Children -0.024051575  590687
## 5            Adventure -0.014887507 1526726
## 6            Animation -0.014431107  373866
## 7              Mystery  0.013584742  454630
## 8               Action -0.012571297 2048028
## 9               Sci-Fi -0.012317259 1073298
## 10               Drama  0.010726557 3128177
## 11             Musical -0.009786582  346871
## 12               Crime  0.008234989 1061786
## 13             Western -0.006999165  151240
## 14              Horror  0.006775621  553143
## 15             Fantasy -0.005615740  741090
## 16            Thriller -0.004846697 1860678
## 17                IMAX -0.004433193    6461
## 18             Romance -0.003668257 1369871
## 19              Comedy -0.002175875 2832440
## 20                 War  0.001946354  408587
```

The ‘no genres listed’ column has high effect_bg but low n, so a penalized average model should be considering. Now, regularization will be considered.

Regularization

The current model:

$$R = \mu + b_i + b_u + \sum_{p=1}^{n} (b_g)_p + \epsilon$$

Can be replaced with a penalized average

$$R = \mu + \frac{1}{\lambda + n_i} \sum B_i + \frac{1}{\lambda + n_u} \sum B_u + \sum_{p=1}^{n} (\frac{1}{\lambda + n_g} \sum B_g)_p + \epsilon$$

Where λ is the regularization parameter for the movie bias, user bias, and genre bias. n_i, n_u and n_g are the number of rating for the ith movie, uth user, and gth genre.

Where $\sum B_i$ is the sum of the movie biases for a given movie, $\sum B_u$ is the sum of the user biases for a given user, $\frac{1}{\lambda+n}$ is the penalized average for a given bias. If n is small, lambda compensates for the small sample size. Therefore, the term,

$$\sum_{p=1}^{n}(\frac{1}{\lambda+n_g}\sum B_g)_p$$

is the sum of the regularized genre biases, for n genres.

The regularization parameter is set in a logarithmic scale, with each additional term being three times the previous term.

```
params_lambda <- 1/30 * (3^seq(0,9,1))
```

The lowest RMSE will be determined by training on *edx_train* and validating on *edx_validation*. However, edx_validation has to be separated first, and the unregularized movie and user effects have to be removed:

```
invisible(gc())
#Remove unregularized movie and user effects in training set
edx_train_genres <- edx_train_genres %>% select(-effect_bi, -effect_bu)


#Creating edx_validation_genres
validation_genre_count <- 1 + str_count(edx_validation$genres, "\\|") %>% max()

genrecols_validation <- paste0('genre', 1:validation_genre_count)

edx_validation_genres <- edx_validation %>% separate(genres, genrecols_validation,
    sep = "\\|")
```

```
## Warning: Expected 8 pieces. Missing pieces filled with `NA` in 899953 rows [1,
   2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

A vector called rmses is generated to find the best regularization parameter, using the crossvalidation set. (Note: This code is very RAM intensive, if you encounter an error on the lines of `error: cannot allocate vector of x MiB`, save the R Session and Restart R.)

```
# Temporarily unload useless data frames
save(mu, rmse0, rmse1, rmse2, rmse3, edx, validation, edx_train, edx_validation,
    edx_debug, edx_train_genres, edx_validation_genres, edx_debug_genres, join_
    genres, maxgenre, genrecols, numcols, biascols, file = 'importants.Rdata')
remove(rmse0, rmse1, rmse2, rmse3, edx, validation, edx_train, edx_validation, edx_
    debug)

invisible(gc())
# Calculating best regularization parameter
# Same as debug set testing with genre effects, but this uses the validation set
# and regularization parameters
# this can take upwards of 30 minutes

rmses <- sapply(1:length(params_lambda), function(x){
  # remove(movie_bias, genre_effects, genre_bias, user_bias, edx_train_genres_movie
      , edx_train_genres_user)

movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)
```

```
edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )

user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

edx_train_genres_user <- edx_train_genres_movie %>% left_join(user_bias, by = "
    userId")
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_genres_user) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_genres_user %>% group_by_at(ind+i)
      %>%
                                          summarize(test = sum(rating - mu - effect_bi -
                                              effect_bu), n = n()) %>% mutate(effect_bg =
                                              test/(n+params_lambda[x])) %>% select(-test),
                                       c(genrecols[i], numcols[i], biascols[i]))
  # Bias column comes after frequency column in SetNames as mutate comes after
      summarize.
}


total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices<- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))

validation_genre_ind <- -1 + which(colnames(edx_validation_genres) == "genre1")

validation_genrebias <- matrix(ncol = validation_genre_count, nrow = nrow(edx_
    validation_genres))



for(i in 1:validation_genre_count){

 foo <- merge(edx_validation_genres, genre_bias_regularized, by.x = (validation_
     genre_ind + i), by.y = 1, all.x = TRUE)
validation_genrebias[,i] <- foo$effect_bg
}

validation_genrebias[is.na(validation_genrebias)] <- 0
```

```
validation_genrebias <- rowSums(validation_genrebias)

by_movie_user_and_genre_reg <- mu + {edx_validation_genres %>% left_join(movie_bias
    , by = "movieId") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + validation_genrebias
gc()
return(RMSE(edx_validation_genres$rating, by_movie_user_and_genre_reg))
})
```

This code will take 5 - 10 minutes to run.

The lambda which minimizes the RMSE in the validation set is:

```
params_lambda[which.min(rmses)]

## [1] 8.1
```

Using this, the model can be tested on the debug set:

```
invisible(gc())
# x is the index of the lambda parameter with the lowest rmse in the valid
x <- which.min(rmses)
movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)
edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )

user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

edx_train_genres_user <- edx_train_genres_movie %>% left_join(user_bias, by = "
    userId")
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_genres_user) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_genres_user %>% group_by_at(ind+i)
      %>%
                                        summarize(test = sum(rating - mu - effect_bi -
                                            effect_bu), n = n()) %>% mutate(effect_bg =
                                            test/(n+params_lambda[x])) %>% select(-test),
                                    c(genrecols[i],numcols[i], biascols[i]))

}


total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices<- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))

debug_genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias_reg <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug_
    genres))



for(i in 1:debug_genre_count){

 foo <- merge(edx_debug_genres, genre_bias_regularized, by.x = (debug_genre_ind + i
     ), by.y = 1, all.x = TRUE)
debug_genrebias_reg[,i] <- foo$effect_bg
}
debug_genrebias_reg[is.na(debug_genrebias_reg)] <- 0
debug_genrebias_reg <- rowSums(debug_genrebias_reg)

by_movie_user_and_genre_reg <- mu + {edx_debug_genres %>% left_join(movie_bias, by
    = "movieId") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + debug_genrebias_reg

rmse4 <- RMSE(edx_debug_genres$rating, by_movie_user_and_genre_reg)
rmse4
```

```
## [1] 0.8661324
```

As can be seen, the RMSE is lower than both the movie + user effects model, and the unregularized model with movie, user and genre effects.

Unneccesary tables can now be removed:

```
remove(genre_effects_reg, edx_train_genres_movie, edx_train_genres_user)

invisible(gc())
load(paste0(getwd(),"/importants.Rdata"))
file.remove("importants.Rdata")
```

```
## [1] TRUE
```

As can be seen, the RMSE is lower than both the movie + user effects model, and the unregularized model with movie, user and genre effects.

However, if we do not consider genre effects, we can see there is a problem:

```
by_movie_user_reg <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId
    ") %>%
          left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi +
              effect_bu) %>%
          pull(total_effect)}
rmsea <- RMSE(edx_debug_genres$rating, by_movie_user_reg)
rmsea
```

```
## [1] 0.8660142
```

When not considering genre effects, the RMSE is, in fact, lower!

Averaged Genre Effects

Instead of considering the genre effects as cumulative, they can be averaged. This changes the ratings formula to the following:

$$R = \mu + \frac{1}{\lambda + n_i}\sum B_i + \frac{1}{\lambda + n_u}\sum B_u + \frac{1}{n}\sum_{p=1}^{n}(\frac{1}{\lambda + n_g}\sum B_g)_p + \epsilon$$

This is done in the code as follows:

```
debug_genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias_reg_2 <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug_
    genres))


for(i in 1:debug_genre_count){

  foo <- merge(edx_debug_genres, genre_bias_regularized, by.x = (debug_genre_ind + i
      ), by.y = 1, all.x = TRUE)
debug_genrebias_reg_2[,i] <- foo$effect_bg
}

# na.rm = TRUE because many entries will have NAs. Switching the NAs to zeros will
    only result in the zeros being added in the mean, and skewing the bias towards
    zero.
debug_genrebias_reg_2 <- rowMeans(debug_genrebias_reg_2, na.rm = TRUE)

by_movie_user_and_genre_reg_2 <- mu + {edx_debug_genres %>% left_join(movie_bias,
    by = "movieId") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + debug_genrebias_reg_2
# identical(edx_debug_genres$rating, edx_debug$rating) is TRUE
# so it does not matter which vector one uses.
rmse5 <- RMSE(edx_debug_genres$rating, by_movie_user_and_genre_reg_2)
rmse5
```

```
## [1] 0.8660425
```

A sharp decrease, although the RMSE is still higher than regularized movie and user effects.

Since the genre bias involves taking the average of a few bias terms (ranging from one to eight terms) for each rating, this average genre bias can also be regularized. This changes the rating formula to the following:

$$R = \mu + \frac{1}{\lambda + n_i}\sum B_i + \frac{1}{\lambda + n_u}\sum B_u + \frac{1}{n + \lambda}\sum_{p=1}^{n}(\frac{1}{\lambda + n_g}\sum B_g)_p + \epsilon$$

Since lambda has to be used over one more term (the average genre biases), validation has to be redone to find the optimal lambda parameter. This is performed via *edx_validation*:

```
rmses_2 <- sapply(1:length(params_lambda), function(x){
  # optional line of code
  # remove(movie_bias, genre_effects, genre_bias, user_bias, edx_train_genres_movie
      , edx_train_genres_user)
```

```
movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)

edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )

user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

edx_train_genres_user <- edx_train_genres_movie %>% left_join(user_bias, by = "
    userId")
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_genres_user) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_genres_user %>% group_by_at(ind+i)
      %>%
                                        summarize(test = sum(rating - mu - effect_bi -
                                            effect_bu), n = n()) %>% mutate(effect_bg =
                                            test/(n+params_lambda[x])) %>% select(-test),
                                      c(genrecols[i], numcols[i], biascols[i]))
  # Bias column comes after frequency column in SetNames as mutate comes after
      summarize.
}


total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices<- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))

val_genre_ind <- -1 + which(colnames(edx_validation_genres) == "genre1")

validation_genrebias_2 <- matrix(ncol = validation_genre_count, nrow = nrow(edx_
    validation_genres))



for(i in 1:validation_genre_count){

 foo <- merge(edx_validation_genres, genre_bias_regularized, by.x = (val_genre_ind
     + i), by.y = 1, all.x = TRUE)
validation_genrebias_2[,i] <- foo$effect_bg
}

# sum(!is.na(u)) sums all non-NA terms in vector u and adds them to the lambda
    parameter
# 1 in apply means the operation is applied to each ROW, not COLUMN
validation_genrebias_2 <- apply(validation_genrebias_2,1,FUN = function(u){sum(u,
    na.rm = TRUE)/(sum(!is.na(u))+params_lambda[x])})

by_movie_user_and_genre_reg_2 <- mu + {edx_validation_genres %>% left_join(movie_
    bias, by = "movieId") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + validation_genrebias_2
gc()
return(RMSE(edx_validation_genres$rating, by_movie_user_and_genre_reg_2))
})
```

We can see that the regularization parameter remains the same:

```
params_lambda[which.min(rmses_2)]

## [1] 8.1
```

Once this is complete, the regularized mean genre bias can now be tested on the debug set:

```
invisible(gc())
x <- which.min(rmses_2)

movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)

edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )

user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

edx_train_genres_user <- edx_train_genres_movie %>% left_join(user_bias, by = "
    userId")
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_genres_user) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_genres_user %>% group_by_at(ind+i)
      %>%
                                       summarize(test = sum(rating - mu - effect_bi -
                                           effect_bu), n = n()) %>% mutate(effect_bg =
                                           test/(n+params_lambda[x])) %>% select(-test),
                                    c(genrecols[i],numcols[i], biascols[i]))

}


total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0
```

```r
n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices<- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))

debug_genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias_reg_3 <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug_
    genres))



for(i in 1:debug_genre_count){

 foo <- merge(edx_debug_genres, genre_bias_regularized, by.x = (debug_genre_ind + i
     ), by.y = 1, all.x = TRUE)
debug_genrebias_reg_3[,i] <- foo$effect_bg
}

debug_genrebias_reg_3 <- apply(debug_genrebias_reg_3,1,FUN = function(u){sum(u, na.
    rm = TRUE)/(sum(!is.na(u))+params_lambda[x])})

by_movie_user_and_genre_reg_3 <- mu + {edx_debug_genres %>% left_join(movie_bias,
    by = "movieId") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + debug_genrebias_reg_3

rmse6 <- RMSE(edx_debug_genres$rating, by_movie_user_and_genre_reg_3)
rmse6
```

```
## [1] 0.8660142
```

The RMSE looks very similar to the regularized movie and user effects model, but on closer inspection:

```r
rmse6 < rmsea
```

```
## [1] TRUE
```

The model is, in fact, better. Although this may be due to chance/oversmoothing, it suggests that the assumption of genre independence may not hold true, which is alleviated using the regularized mean of all the genre effects for a given movie, combined.

But can more be done? Looking at the training set:

```r
str(edx_train)
```

```
## Classes 'data.table' and 'data.frame':   7200087 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : num  122 292 316 329 355 362 364 370 377 420 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983421 838983392 838983392 838984474 838984885
    838983707 838984596 838983834 838983834 ...
##  $ title    : chr  "Boomerang (1992)" "Outbreak (1995)" "Stargate (1994)" "Star
    Trek: Generations (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Drama|Sci-Fi|Thriller" "Action|
    Adventure|Sci-Fi" "Action|Adventure|Drama|Sci-Fi" ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

We can see that there is one more variable, the timestamp of the rating. So, the effect of time on ratings (which is independent of user and movie, since those biases have already been calculated) can be considered.

Time effects

As movies are released in a seasonal cycle, where higher quality movies are released in the summer, or closer to awards ceremonies. So, ratings may fluctuate with time.

From the MovieLens documentation, it can be observed that all timestamps are in Unix Time at UTC. So, while the data can be converted from Unix Time (which is time, in seconds, from January 1st, 1970) to a date format, it is not advisable as certain models are not feasible with PosixCt variables.

The regularized movie and user effects can be added:

```
gc()

##           used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells 2284983 122.1    4065154 217.2  4065154 217.2
## Vcells 3805839  29.1    8388608  64.0  5963611  45.5

edx_train_date <- edx_train_genres %>% left_join(movie_bias, by = 'movieId')
gc()

##             used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells   2307364  123.3    4065154  217.2   4065154  217.2
## Vcells 183906842 1403.1  321133020 2450.1 316318403 2413.4

edx_train_date <- edx_train_date %>% left_join(user_bias, by = 'userId')

save(mu, movie_bias, user_bias, rmse0, rmse1, rmse2, rmse3, rmse4,rmse5, rmse6,
    rmsea, rmses,rmses_2, edx, validation, edx_train, edx_validation, edx_debug,
    edx_train_genres, edx_validation_genres, edx_debug_genres, join_genres, maxgenre
    , genrecols, numcols, biascols, file = 'importants.Rdata')
remove(rmse0, rmse1, rmse2, rmse3, rmse4, edx, validation, edx_train, edx_
    validation, edx_debug)
```

Depending on which approach is used to calculate the genre biases, the time effects may improve RMSE or worsen it (for example, the third model (with a regularized average of all the genre biases for a given movie) did not provide a substantial improvement in RMSE). Hence, the effectiveness of all three models, in tandem with a linear model to account for time effects shall be considered. The first approach is the cumulative genre biases model (genre biases for each rating are added):

```
invisible(gc())

x <- which.min(rmses)
movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)
```

```r
edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )

user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

edx_train_genres_user <- edx_train_genres_movie %>% left_join(user_bias, by = "
    userId")

genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_genres_user) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_genres_user %>% group_by_at(ind+i)
      %>%
                                        summarize(test = sum(rating - mu - effect_bi -
                                            effect_bu), n = n()) %>% mutate(effect_bg =
                                            test/(n+params_lambda[x])) %>% select(-test),
                                     c(genrecols[i],numcols[i], biascols[i]))

}

remove(edx_train_genres_movie, edx_train_genres_user)

total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices <- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))
```

```r
edx_train_date <- edx_train_genres %>% left_join(movie_bias, by = 'movieId')
edx_train_date <- edx_train_date %>%
  left_join(user_bias, by = 'userId')

invisible(gc())

train_genre_ind <- -1 + which(colnames(edx_train_date) == "genre1") %>% as.integer
    ()

train_genrebias_reg <- matrix(ncol = maxgenre, nrow = nrow(edx_train_date))
edx_train_date <- as_tibble(edx_train_date)
```

```
invisible(gc())

for(i in 1:maxgenre){

 foo <- merge.data.frame(edx_train_date, genre_bias_regularized, by.x = (train_
     genre_ind + i), by.y = 1, all.x = TRUE)
train_genrebias_reg[,i] <- foo$effect_bg
remove(foo)
invisible(gc())
}


train_genrebias_reg[is.na(train_genrebias_reg)] <- 0
train_genrebias_reg <- rowSums(train_genrebias_reg)



edx_train_date <- edx_train_date %>% mutate(effect_bg = train_genrebias_reg)
remove(train_genrebias_reg)
```

foo is removed from the for loop to overcome RAM limitations. The residuals can be saved in a new column:

```
edx_train_date <- edx_train_date %>% mutate(error = rating - mu -effect_bi - effect
    _bu - effect_bg)
```

Time effects on the error (aka, time bias can now be considered). The first model to be used is a linear model:

```
linear_date <- edx_train_date %>% lm(error ~ timestamp,data = .)
```

This can now be tested on the debug set:

```
debug_time <- predict(linear_date, newdata = data.frame(timestamp = edx_debug_
    genres$timestamp))
```

The mean, movie, user and genre effects can now be added:

```
debug_genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias_reg_temp <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug_
    genres))

for(i in 1:debug_genre_count){

 foo <- merge(edx_debug_genres, genre_bias_regularized, by.x = (debug_genre_ind + i
     ), by.y = 1, all.x = TRUE)
debug_genrebias_reg_temp[,i] <- foo$effect_bg
}

debug_genrebias_reg_temp[is.na(debug_genrebias_reg_temp)] <- 0
debug_genrebias_reg_temp <- rowSums(debug_genrebias_reg_temp)




time_effects <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId")
    %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + debug_genrebias_reg_temp + debug_time
```

```
rmse7 <- RMSE(edx_debug_genres$rating, time_effects)
rmse7

## [1] 0.8660986

remove(debug_time, debug_genrebias_reg_temp)
```

This is repeated with the averaged genre effects:

```
# Average of genre biases (copy-pasting the cumulative genre biases model with one
    small change)

remove(linear_date, movie_bias, user_bias)
invisible(gc())
# x is index for parameter of best lambda
x <- which.min(rmses)

# movie bias
movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)

edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )



# user bias
user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

edx_train_genres_user <- edx_train_genres_movie %>% left_join(user_bias, by = "
    userId")


# genre bias
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_genres_user) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_genres_user %>% group_by_at(ind+i)
      %>%
                                        summarize(test = sum(rating - mu - effect_bi -
                                            effect_bu), n = n()) %>% mutate(effect_bg =
                                            test/(n+params_lambda[x])) %>% select(-test),
                                     c(genrecols[i],numcols[i], biascols[i]))

}


total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

n_indices <- which(colnames(total_genre_effects) %in% numcols)
```

```
bias_indices <- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))


remove(edx_train_genres_movie, edx_train_genres_user)

# joining the movie and user biases to the table
edx_train_date <- edx_train_genres %>% left_join(movie_bias, by = 'movieId')
edx_train_date <- edx_train_date %>%
  left_join(user_bias, by = 'userId')

invisible(gc())

# adding genre biases to edx_train_date
train_genre_ind <- -1 + which(colnames(edx_train_date) == "genre1") %>% as.integer
    ()

train_genrebias_reg <- matrix(ncol = maxgenre, nrow = nrow(edx_train_date))
edx_train_date <- as_tibble(edx_train_date)
invisible(gc())

for(i in 1:maxgenre){

 foo <- merge.data.frame(edx_train_date, genre_bias_regularized, by.x = (train_
     genre_ind + i), by.y = 1, all.x = TRUE)
train_genrebias_reg[,i] <- foo$effect_bg
remove(foo)
invisible(gc())
}


train_genrebias_reg <- rowMeans(train_genrebias_reg, na.rm = TRUE)

edx_train_date <- edx_train_date %>% mutate(effect_bg = train_genrebias_reg)
remove(train_genrebias_reg)

# error
edx_train_date <- edx_train_date %>% mutate(error = rating - mu -effect_bi - effect
    _bu - effect_bg)

# linear model

linear_date <- edx_train_date %>% lm(error ~ timestamp,data = .)


# testing
debug_time <- predict(linear_date, newdata = data.frame(timestamp = edx_debug_
    genres$timestamp))
```

```
debug_genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias_reg_temp <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug_
    genres))

for(i in 1:debug_genre_count){

  foo <- merge(edx_debug_genres, genre_bias_regularized, by.x = (debug_genre_ind + i
      ), by.y = 1, all.x = TRUE)
debug_genrebias_reg_temp[,i] <- foo$effect_bg
}


debug_genrebias_reg_temp <- rowMeans(debug_genrebias_reg_temp, na.rm = TRUE)

time_effects <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId")
    %>%
   left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
       bu) %>%
   pull(total_effect)} + debug_genrebias_reg_temp + debug_time

rmse8 <- RMSE(edx_debug_genres$rating, time_effects)
rmse8

## [1] 0.8660108

remove(debug_time, debug_genrebias_reg_temp)
```

This can now be repeated using the penalized average of the genre effects:

```
remove(linear_date, movie_bias, user_bias)

invisible(gc())
# x is index for parameter of best lambda, rmses_2 used
# as lambda parameter also applies when averaging genre biases.
x <- which.min(rmses_2)

# movie bias
movie_bias <- edx_train_genres %>% group_by(movieId) %>% summarize(test = sum(
    rating - mu), n = n()) %>% mutate(effect_bi = test/(n+params_lambda[x])) %>%
    select(-test, -n)

invisible(gc())

edx_train_genres_movie <- edx_train_genres %>% left_join(movie_bias, by = 'movieId'
    )

# user bias
user_bias <- edx_train_genres_movie %>% group_by(userId) %>% summarize(test = sum(
    rating - mu - effect_bi), n = n()) %>% mutate(effect_bu = test/(n+params_lambda[
    x])) %>% select(-test, -n)

remove(edx_train_genres_movie)

# joining movie and user effects to the table
edx_train_date <- edx_train_genres %>% left_join(movie_bias, by = 'movieId')
edx_train_date <- edx_train_date %>%
  left_join(user_bias, by = 'userId')


# genre bias
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_train_date) == 'genre1')
for(i in 1:maxgenre){

  genre_effects_reg[[i]] <- setNames(edx_train_date %>% group_by_at(ind+i) %>%
                                        summarize(test = sum(rating - mu - effect_bi -
                                            effect_bu), n = n()) %>% mutate(effect_bg =
                                            test/(n+params_lambda[x])) %>% select(-test),
                                     c(genrecols[i],numcols[i], biascols[i]))

}

remove(edx_train_genres_movie, edx_train_genres_user)

## Warning in remove(edx_train_genres_movie, edx_train_genres_user): object
## 'edx_train_genres_movie' not found

## Warning in remove(edx_train_genres_movie, edx_train_genres_user): object
## 'edx_train_genres_user' not found

total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

n_indices <- which(colnames(total_genre_effects) %in% numcols)

bias_indices <- which(colnames(total_genre_effects) %in% biascols)

dat <- sapply(1:maxgenre, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
    = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))



invisible(gc())

# adding genre biases to edx_train_date
train_genre_ind <- -1 + which(colnames(edx_train_date) == "genre1") %>% as.integer
    ()

train_genrebias_reg <- matrix(ncol = maxgenre, nrow = nrow(edx_train_date))
edx_train_date <- as_tibble(edx_train_date)
invisible(gc())

for(i in 1:maxgenre){

 foo <- merge.data.frame(edx_train_date, genre_bias_regularized, by.x = (train_
     genre_ind + i), by.y = 1, all.x = TRUE)
```

```r
train_genrebias_reg[,i] <- foo$effect_bg
remove(foo)
invisible(gc())
}

train_genrebias_reg <- apply(train_genrebias_reg,1,FUN = function(u){sum(u, na.rm =
    TRUE)/(sum(!is.na(u)) + params_lambda[x])})

edx_train_date <- edx_train_date %>% mutate(effect_bg = train_genrebias_reg)
remove(train_genrebias_reg)

# error
edx_train_date <- edx_train_date %>% mutate(error = rating - mu -effect_bi - effect
    _bu - effect_bg)

invisible(gc())

# linear model
linear_date <- edx_train_date %>% lm(error ~ timestamp,data = .)

# testing
debug_time <- predict(linear_date, newdata = data.frame(timestamp = edx_debug_
    genres$timestamp))

debug_genre_ind <- -1 + which(colnames(edx_debug_genres) == "genre1")

debug_genrebias_reg_temp <- matrix(ncol = debug_genre_count, nrow = nrow(edx_debug_
    genres))

for(i in 1:debug_genre_count){

  foo <- merge(edx_debug_genres, genre_bias_regularized, by.x = (debug_genre_ind + i
      ), by.y = 1, all.x = TRUE)
debug_genrebias_reg_temp[,i] <- foo$effect_bg
}


debug_genrebias_reg_temp <- apply(debug_genrebias_reg_temp,1,FUN = function(u){sum(
    u, na.rm = TRUE)/(sum(!is.na(u)) + params_lambda[x])})

time_effects <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId")
    %>%
   left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
       bu) %>%
   pull(total_effect)}  + debug_genrebias_reg_temp + debug_time

rmse9 <- RMSE(edx_debug_genres$rating, time_effects)
rmse9
```

```
## [1] 0.8659792
```

As can be seen, the linear model shows that the penalized mean genre biases is the most effective model as it has the lowest RMSE.

Considering the penalized averaged biases model, for example, we can see that there is one issue with the linear model:

```r
noquote("R␣Squared")
```

```
## [1] R Squared
```

```
glance(linear_date)$r.squared
```

```
## [1] 6.139306e-05
```

```
noquote("P␣Value")
```

```
## [1] P Value
```

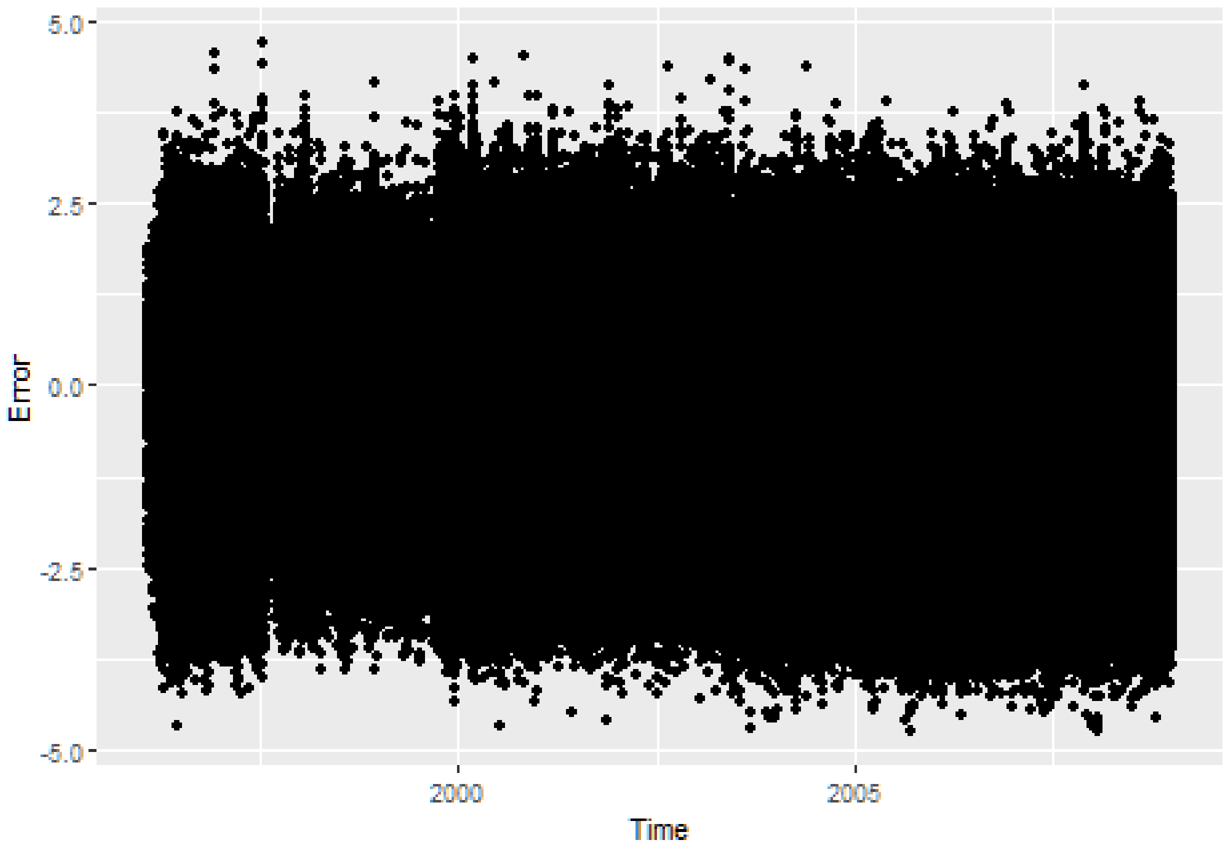
```
glance(linear_date)$p.value
```

```
##         value
## 3.877498e-98
```

Although the p-value is low, implying statistical significance, the r-squared is also low, implying that the fit is poor.

Looking at the errors over time, we can see the issue:

```
edx_train_date %>% ggplot(aes(as_datetime(timestamp), error)) + geom_point() + xlab
    ("Time") + ylab("Error")
```

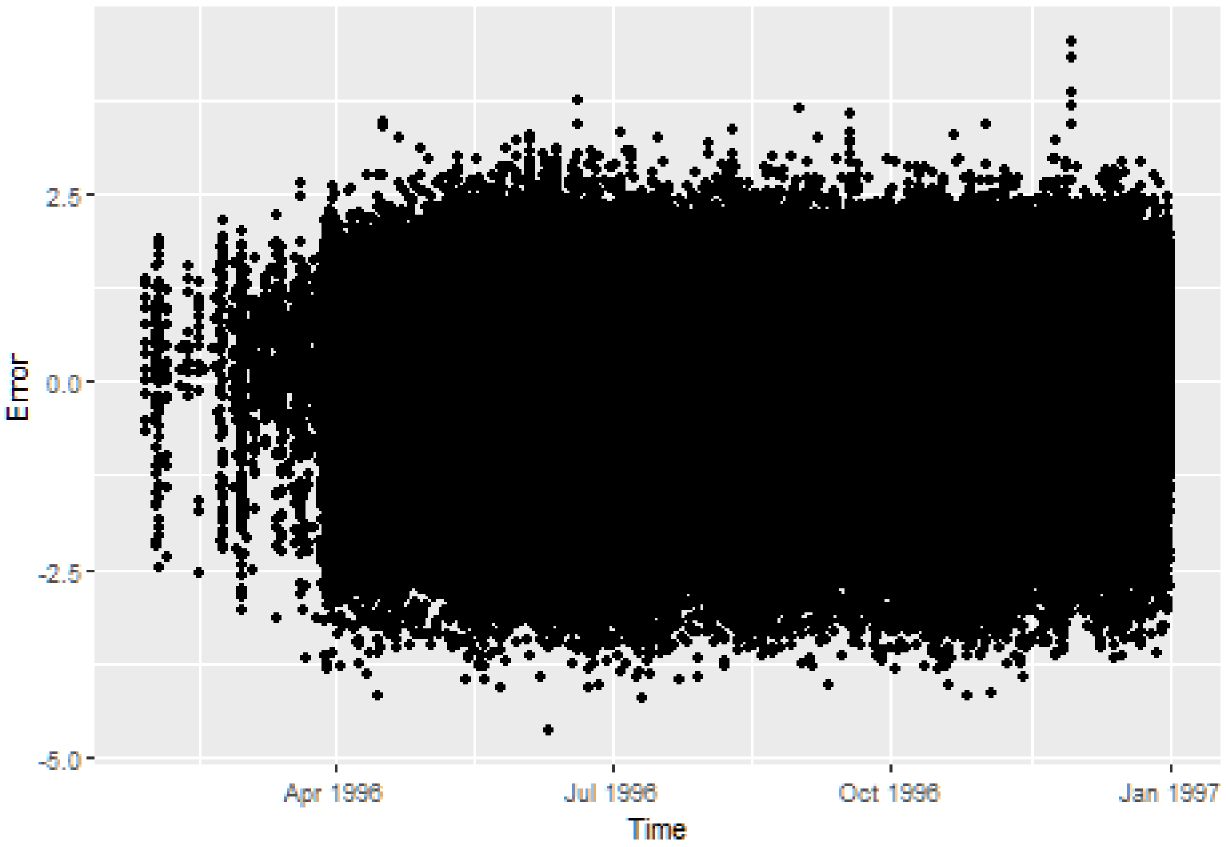


The errors have a large spread (and do not seem bivariate normal), so other methods can be considered. The graph also suggests that despite a poor fit, the linear model provides a good baseline (as seen by the high p-value).

Other fitting methods

To see if the yearly trend (based on movie releases) holds, errors for the year 1996 will be considered:

```
# Date format is YYYY-MM-DD
startdate <- as.integer(as_datetime("1996-01-01 00:00:00 UTC"))
enddate <- as.integer(as_datetime("1996-12-31 23:59:59 UTC"))
edx_train_date %>% filter(timestamp >= startdate & timestamp <= enddate) %>% ggplot
    (aes(as_datetime(timestamp), error)) + geom_point() + xlab("Time") + ylab("Error
    ")
```



There is still a large degree of variation, probably due to variations in user mood over time. This model will be tested on averaged genre biases (as the cumulative genre biases can lead to a greater overstatement/understatement of the true genre effect depending on the number of genres in the genres column).

```
# RAM Optimization, linear date is a large lm which, if saved, cannot be reloaded
    due to its large size
save(linear_date, file = "linear_date.RData")
remove(linear_date)

# group_split splits the train set into multiple data frames, each with one movieId
    .

movie_split <- group_split(edx_train_date, movieId)
# map_dbl and map are used to store the moveId and the linear model, map stores
    lists so this is why it can be used to store lms, as lm class is a type of list.
movie_lm <- tibble(movieId = map_dbl(movie_split, function(x) return(x$movieId[1]))
    ,
                    lm = map(movie_split, ~ lm(error ~ timestamp, data = .x)))
remove(movie_split)

invisible(gc())

# map2_dbl can take two inputs and produce one output
# Suppresswarnings used to prevent the following warning:
# Warning: Problem with `mutate()` column `foo`.
# i `foo = map2_dbl(.x = lm, .y = timestamp, ~predict(.x, newdata = tibble(
    timestamp = .y)))`.
# i prediction from a rank-deficient fit may be misleading
suppressWarnings(debug_movie_time <- edx_debug_genres %>% left_join(movie_lm, by =
    "movieId") %>% mutate(foo = map2_dbl(.x = lm, .y = timestamp, ~ predict(.x,
    newdata = tibble(timestamp = .y)))) %>% .$foo)

# testing
invisible(gc())
time_effects_movie <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "
    movieId") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
     bu) %>%
  pull(total_effect)}  + debug_genrebias_reg + debug_movie_time

rmse10 <- RMSE(edx_debug_genres$rating, time_effects_movie)
rmse10
```

```
## [1] 0.9315572
```

There is a substantial increase in RMSE. Looking at the r-squared for each linear model, we can see what the problem is:

```
rsq = map_dbl(.x = movie_lm$lm, function(u) {glance(u)$r.squared})
rsq %>% head(10)
```

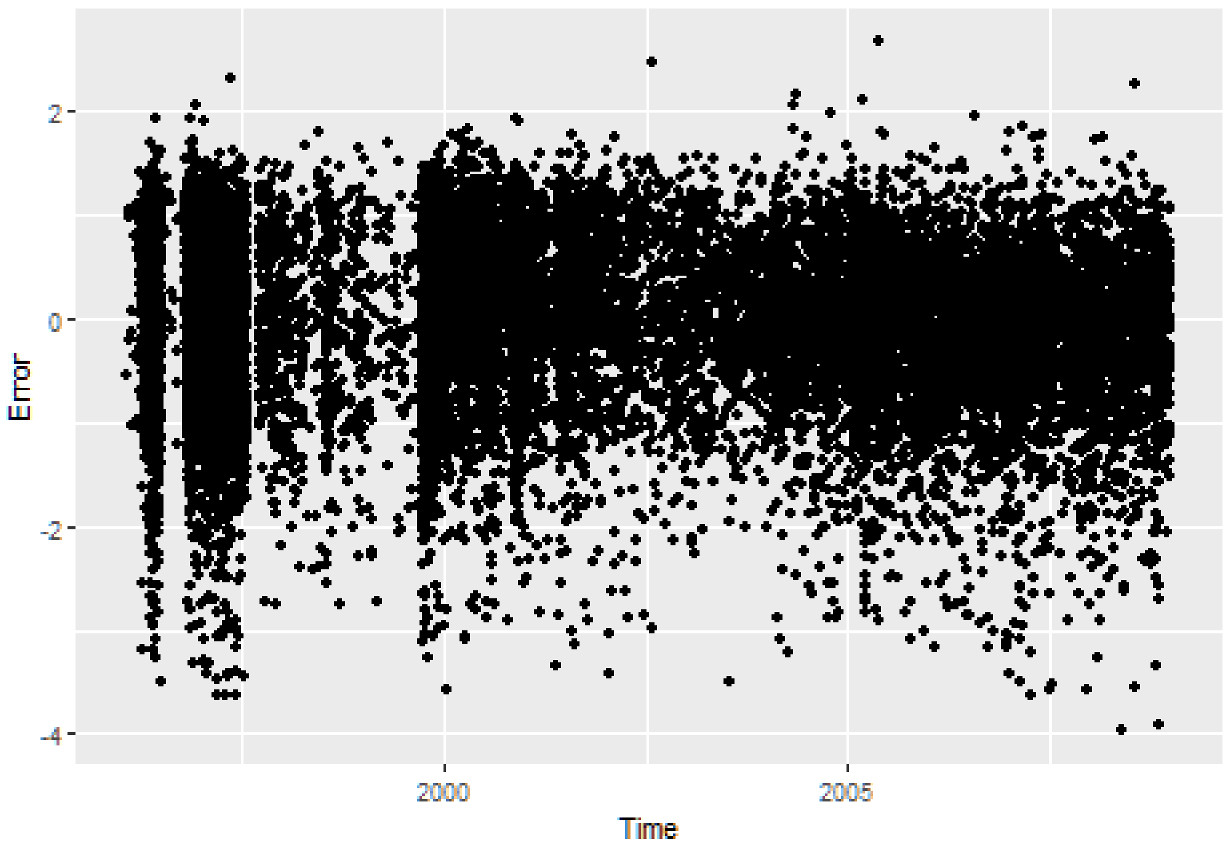
```
##  [1] 0.005328923 0.050803767 0.014594952 0.007378045 0.021288643 0.014685615
##  [7] 0.013704623 0.020920290 0.047908325 0.001808096
```

```
remove(rsq, movie_lm)
```

The r-squared is low! This suggests the movie disposition is not correlated with time.

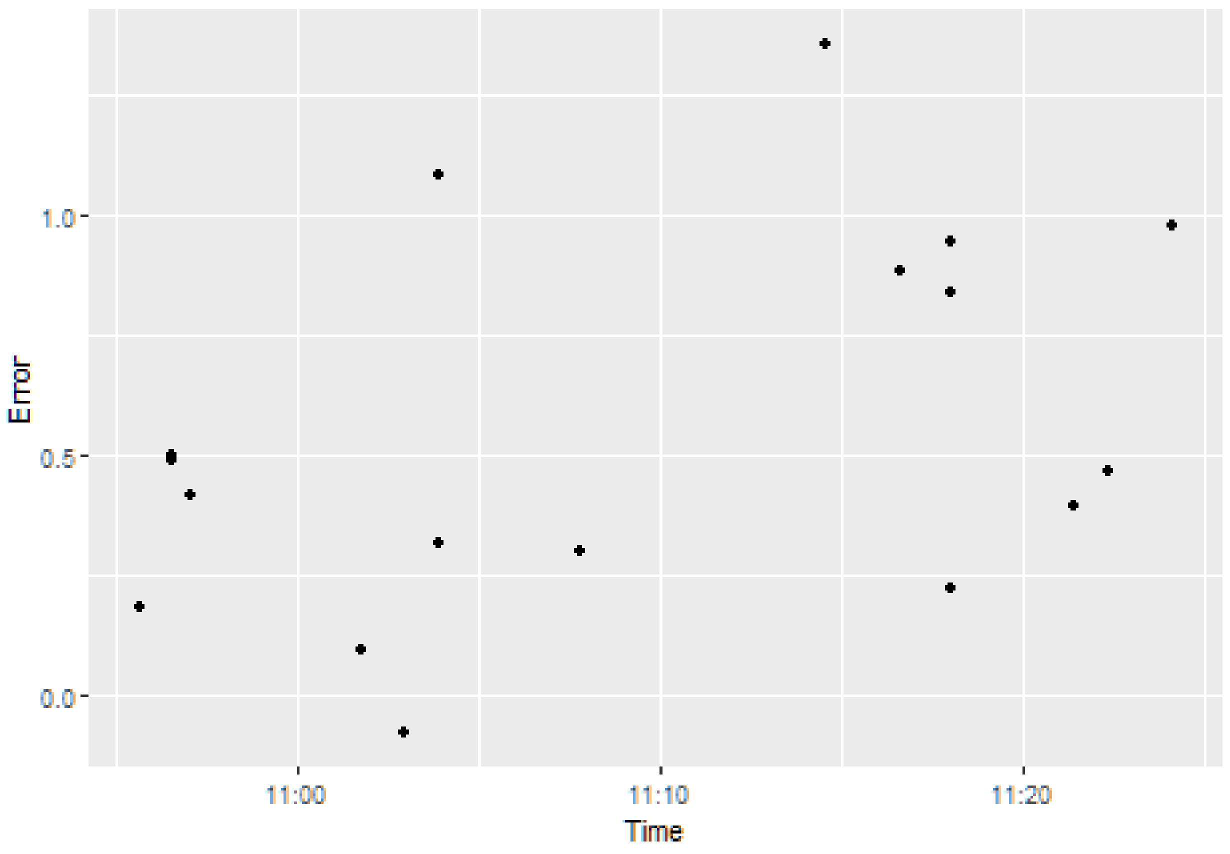
This can be seen in any one movie:

```
edx_train_date %>% filter(movieId == 1) %>% ggplot(aes(as_datetime(timestamp),
    error)) + geom_point() + xlab("Time") + ylab("Error")
```



However, if we look at a particular user:

```
edx_train_date %>% filter(userId == 1) %>% ggplot(aes(as_datetime(timestamp), error
    )) + geom_point() + xlab("Time") + ylab("Error")
```



There seems to be a trend over time!

User effects over time can also be considered.

```
# same as movie_lm, but with users
invisible(gc())
user_split <- group_split(edx_train_date, userId)
user_lm <- tibble(userId = map_dbl(user_split, function(x) return(x$userId[1])),
                  lm = map(user_split, ~ lm(error ~ timestamp, data = .x)))
remove(user_split)

# testing, but with user time effects
invisible(gc())
suppressWarnings(debug_user_time <- edx_debug_genres %>% left_join(user_lm, by = "
    userId") %>% mutate(foo = map2_dbl(.x = lm, .y = timestamp, ~ predict(.x,
    newdata = tibble(timestamp = .y)))) %>% .$foo)


time_effects_user <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId
    ") %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + debug_genrebias_reg + debug_user_time

rmse11 <- RMSE(edx_debug_genres$rating, time_effects_user)
rmse11

## [1] 291.8811
```

The RMSE is far too high. Looking at the r-squared values, the same issue is observed:

```
rsq = map_dbl(.x = user_lm$lm, function(u) {glance(u)$r.squared})
rsq %>% head(10)

##  [1] 0.1810017244 0.2352154176 0.0271733854 0.0075593051 0.0079469555
##  [6] 0.0034699645 0.0001427566 0.0445583148 0.0000000000 0.0142289672

remove(rsq, user_lm, time_effects, time_effects_movie, time_effects_user)
```

The r-squared is still poor. This is most likely due to a common error in both models (as this shows up every time lm is run in map_dbl, suppressWarnings was used)

Warning: Problem with mutate() column foo. i foo = map2_dbl(.x = lm, .y = timestamp, ~predict(.x, newdata = tibble(timestamp = .y))). i prediction from a rank-deficient fit may be misleading

This shows the ineffectiveness of user/movie effects changing over time.

Instead of considering a linear model, a quadratic/cubic model can also be considered for overall time effects:

Quadratic time effects model:

```
invisible(gc())
edx_train_date <- edx_train_date %>% mutate(t2 = timestamp^2)

quadratic_date <- edx_train_date %>% lm(error ~ timestamp + t2,data = .)

debug_time_2 <- predict(quadratic_date, newdata = data.frame(timestamp = edx_debug_
    genres$timestamp, t2 = (edx_debug_genres$timestamp)^2))

time_effects_2 <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId")
    %>%
   left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
   pull(total_effect)}  + debug_genrebias_reg + debug_time_2

rmse12 <- RMSE(time_effects_2, edx_debug_genres$rating)
rmse12

## [1] 0.8659758
```

Looking at the model, we can see that the r-squared has improved (albeit slightly).

```
noquote("R␣Squared")

## [1] R Squared

glance(quadratic_date)$r.squared

## [1] 6.270972e-05

noquote("P␣Value")

## [1] P Value

glance(quadratic_date)$p.value

##         value
## 8.946016e-99
```

Cubic time effects model

Using a third-degree polynomial model, we can try to find a better fit:

```
remove(quadratic_date)

edx_train_date <- edx_train_date %>% mutate(t3 = timestamp^3)

cubic_date <- edx_train_date %>% lm(error ~ timestamp + t2 + t3,data = .)

debug_time_3 <- predict(cubic_date, newdata = data.frame(timestamp = edx_debug_
    genres$timestamp, t2 = (edx_debug_genres$timestamp)^2, t3 = (edx_debug_genres$
    timestamp)^3))

time_effects_3 <- mu + {edx_debug_genres %>% left_join(movie_bias, by = "movieId")
    %>%
  left_join(user_bias, by = "userId") %>% mutate(total_effect = effect_bi + effect_
      bu) %>%
  pull(total_effect)}  + debug_genrebias_reg + debug_time_3

rmse13 <- RMSE(time_effects_3, edx_debug_genres$rating)
rmse13
```

```
## [1] 0.8659759
```

It seems that adding more terms only worsens the RMSE.

```
noquote("R␣Squared")
```

```
## [1] R Squared
```

```
glance(cubic_date)$r.squared
```

```
## [1] 6.302577e-05
```

```
noquote("P␣Value")
```

```
## [1] P Value
```

```
glance(cubic_date)$p.value
```

```
##        value
## 4.884341e-98
```

```
remove(cubic_date)
```

These results suggest that the quadratic model is the best-performing, and higher degree models only worsen the RMSE.

The files can now be reloaded.

```
dir <- getwd()
load(paste0(dir,"/importants.Rdata"))
```

While the effectiveness of timestamp-based regression models was not significant, there is one more observation that can be made of the data:

```
# Show that ratings can be considered a classification problem
levels(as.factor(edx_train$rating))
```

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

We can see that there are only 10 fixed values that the ratings can take. So, classification algorithms can be considered.

Classification algorithms and the Ranger library

By considering the ratings as fixed values, we can use classification techniques such as random forests (using the ranger library as it is built for large datasets) to classify the ratings.

```
# as.factor to ensure ranger uses classification
edx_train_2 <- edx_train %>% mutate(rating = as.factor(rating))


fit_ranger <- ranger(formula = rating ~ ., data = edx_train_2, num.trees = 100, max
    .depth = 10)
```

```
## Growing trees.. Progress: 1%. Estimated remaining time: 2 hours, 54 minutes, 54
    seconds.
## Growing trees.. Progress: 9%. Estimated remaining time: 35 minutes, 13 seconds.
## Growing trees.. Progress: 17%. Estimated remaining time: 25 minutes, 37 seconds.
## Growing trees.. Progress: 24%. Estimated remaining time: 18 minutes, 34 seconds.
## Growing trees.. Progress: 25%. Estimated remaining time: 21 minutes, 6 seconds.
## Growing trees.. Progress: 31%. Estimated remaining time: 16 minutes, 54 seconds.
## Growing trees.. Progress: 33%. Estimated remaining time: 17 minutes, 39 seconds.
## Growing trees.. Progress: 37%. Estimated remaining time: 15 minutes, 58 seconds.
## Growing trees.. Progress: 41%. Estimated remaining time: 15 minutes, 5 seconds.
## Growing trees.. Progress: 45%. Estimated remaining time: 13 minutes, 43 seconds.
## Growing trees.. Progress: 49%. Estimated remaining time: 12 minutes, 47 seconds.
## Growing trees.. Progress: 53%. Estimated remaining time: 11 minutes, 35 seconds.
## Growing trees.. Progress: 57%. Estimated remaining time: 10 minutes, 40 seconds.
## Growing trees.. Progress: 61%. Estimated remaining time: 9 minutes, 29 seconds.
## Growing trees.. Progress: 65%. Estimated remaining time: 8 minutes, 32 seconds.
## Growing trees.. Progress: 68%. Estimated remaining time: 7 minutes, 42 seconds.
## Growing trees.. Progress: 72%. Estimated remaining time: 6 minutes, 42 seconds.
## Growing trees.. Progress: 76%. Estimated remaining time: 5 minutes, 44 seconds.
## Growing trees.. Progress: 80%. Estimated remaining time: 4 minutes, 46 seconds.
## Growing trees.. Progress: 84%. Estimated remaining time: 3 minutes, 48 seconds.
## Growing trees.. Progress: 86%. Estimated remaining time: 3 minutes, 20 seconds.
## Growing trees.. Progress: 89%. Estimated remaining time: 2 minutes, 37 seconds.
## Growing trees.. Progress: 93%. Estimated remaining time: 1 minute, 39 seconds.
## Growing trees.. Progress: 97%. Estimated remaining time: 41 seconds.
```

```
edx_debug_2 <- edx_debug %>% mutate(rating = as.factor(rating))
test <- predict(fit_ranger, data = edx_debug_2)

rmse14 <- RMSE(as.numeric(test$predictions), as.numeric(edx_debug_2$rating))
rmse14
```

```
## [1] 2.306885
```

As can be seen, treating the rating as a categorical variable resulted in a high RMSE.

```
table(test$predictions)
```

```
##
##    0.5      1    1.5      2    2.5      3    3.5      4    4.5      5
##      0      0      0      0      0 165086   1220 715033      0  18646
```

```
table(edx_debug_2$rating)
```

```
##
##    0.5      1    1.5      2    2.5      3    3.5      4    4.5      5
##   8600  34363  10787  71108  33328 212116  79359 258641  52811 138872
```

It seems that the decision tree did not predict lower ratings at all!

However, ranger also supports regression algorithms:

```
# rating is a numeric in edx_train and edx_debug so ranger will default to
    regression
remove(fit_ranger)
fit_ranger <- ranger(formula = rating ~ ., data = edx_train, num.trees = 100, max.
    depth = 10)

## Growing trees.. Progress: 1%. Estimated remaining time: 2 hours, 10 minutes, 21
    seconds.
## Growing trees.. Progress: 9%. Estimated remaining time: 26 minutes, 37 seconds.
## Growing trees.. Progress: 17%. Estimated remaining time: 19 minutes, 26 seconds.
## Growing trees.. Progress: 25%. Estimated remaining time: 15 minutes, 57 seconds.
## Growing trees.. Progress: 33%. Estimated remaining time: 13 minutes, 30 seconds.
## Growing trees.. Progress: 41%. Estimated remaining time: 11 minutes, 30 seconds.
## Growing trees.. Progress: 49%. Estimated remaining time: 9 minutes, 42 seconds.
## Growing trees.. Progress: 57%. Estimated remaining time: 8 minutes, 3 seconds.
## Growing trees.. Progress: 65%. Estimated remaining time: 6 minutes, 27 seconds.
## Growing trees.. Progress: 73%. Estimated remaining time: 4 minutes, 56 seconds.
## Growing trees.. Progress: 81%. Estimated remaining time: 3 minutes, 26 seconds.
## Growing trees.. Progress: 89%. Estimated remaining time: 1 minute, 58 seconds.
## Growing trees.. Progress: 97%. Estimated remaining time: 31 seconds.

test <- predict(fit_ranger, data = edx_debug)
rmse15 <- RMSE(test$predictions, edx_debug$rating)
rmse15

## [1] 1.02602
```

The RMSE has improved! However, it is nowhere near the generative model, using genre and time effects. Improvements may be observed by adding more trees.

```
# try again because why not?

# rating is a numeric in edx_train and edx_debug so ranger will default to
    regression
remove(fit_ranger)
invisible(gc())
fit_ranger <- ranger(formula = rating ~ ., data = edx_train, num.trees = 200, max.
    depth = 10)

## Growing trees.. Progress: 1%. Estimated remaining time: 4 hours, 31 minutes, 58
    seconds.
## Growing trees.. Progress: 5%. Estimated remaining time: 58 minutes, 21 seconds.
## Growing trees.. Progress: 9%. Estimated remaining time: 44 minutes, 8 seconds.
## Growing trees.. Progress: 13%. Estimated remaining time: 38 minutes, 16 seconds.
## Growing trees.. Progress: 17%. Estimated remaining time: 34 minutes, 29 seconds.
## Growing trees.. Progress: 21%. Estimated remaining time: 31 minutes, 32 seconds.
## Growing trees.. Progress: 25%. Estimated remaining time: 29 minutes, 16 seconds.
## Growing trees.. Progress: 28%. Estimated remaining time: 27 minutes, 10 seconds.
## Growing trees.. Progress: 33%. Estimated remaining time: 25 minutes, 20 seconds.
## Growing trees.. Progress: 37%. Estimated remaining time: 23 minutes, 29 seconds.
## Growing trees.. Progress: 40%. Estimated remaining time: 21 minutes, 6 seconds.
## Growing trees.. Progress: 41%. Estimated remaining time: 21 minutes, 51 seconds.
## Growing trees.. Progress: 44%. Estimated remaining time: 19 minutes, 41 seconds.
## Growing trees.. Progress: 45%. Estimated remaining time: 20 minutes, 17 seconds.
## Growing trees.. Progress: 48%. Estimated remaining time: 18 minutes, 32 seconds.
## Growing trees.. Progress: 49%. Estimated remaining time: 18 minutes, 44 seconds.
## Growing trees.. Progress: 51%. Estimated remaining time: 17 minutes, 29 seconds.
## Growing trees.. Progress: 53%. Estimated remaining time: 17 minutes, 12 seconds.
## Growing trees.. Progress: 55%. Estimated remaining time: 16 minutes, 0 seconds.
## Growing trees.. Progress: 56%. Estimated remaining time: 15 minutes, 42 seconds.
## Growing trees.. Progress: 59%. Estimated remaining time: 14 minutes, 33 seconds.
## Growing trees.. Progress: 61%. Estimated remaining time: 14 minutes, 12 seconds.
## Growing trees.. Progress: 63%. Estimated remaining time: 13 minutes, 22 seconds.
## Growing trees.. Progress: 65%. Estimated remaining time: 12 minutes, 41 seconds.
## Growing trees.. Progress: 67%. Estimated remaining time: 11 minutes, 55 seconds.
## Growing trees.. Progress: 69%. Estimated remaining time: 11 minutes, 12 seconds.
## Growing trees.. Progress: 71%. Estimated remaining time: 10 minutes, 28 seconds.
## Growing trees.. Progress: 73%. Estimated remaining time: 9 minutes, 45 seconds.
## Growing trees.. Progress: 75%. Estimated remaining time: 9 minutes, 1 seconds.
## Growing trees.. Progress: 77%. Estimated remaining time: 8 minutes, 19 seconds.
## Growing trees.. Progress: 79%. Estimated remaining time: 7 minutes, 35 seconds.
## Growing trees.. Progress: 81%. Estimated remaining time: 6 minutes, 53 seconds.
## Growing trees.. Progress: 83%. Estimated remaining time: 6 minutes, 10 seconds.
## Growing trees.. Progress: 85%. Estimated remaining time: 5 minutes, 27 seconds.
## Growing trees.. Progress: 87%. Estimated remaining time: 4 minutes, 45 seconds.
## Growing trees.. Progress: 89%. Estimated remaining time: 4 minutes, 2 seconds.
## Growing trees.. Progress: 91%. Estimated remaining time: 3 minutes, 20 seconds.
## Growing trees.. Progress: 93%. Estimated remaining time: 2 minutes, 37 seconds.
## Growing trees.. Progress: 95%. Estimated remaining time: 1 minute, 56 seconds.
## Growing trees.. Progress: 97%. Estimated remaining time: 1 minute, 13 seconds.
## Growing trees.. Progress: 99%. Estimated remaining time: 31 seconds.
## Computing prediction error.. Progress: 46%. Estimated remaining time: 39 seconds
   .
## Computing prediction error.. Progress: 86%. Estimated remaining time: 10 seconds
   .
```

```
test <- predict(fit_ranger, data = edx_debug)
rmse16 <- RMSE(test$predictions, edx_debug$rating)
rmse16
```

```
## [1] 1.025602
```

```
remove(fit_ranger)
```

The RMSE has not changed significantly from the previous model. It seems that increasing the number of trees has little effect.

Results

The above models can now be compared:

```
results <- data.frame(method = c("Only␣the␣Mean","Movie␣Effects", "Movie␣and␣User␣
    Effects", "Movie,␣User␣and␣Genre␣Effects",
                          "Regularized␣Movie␣and␣User␣Effects",
                          "Regularized␣Movie,␣User␣and␣Cumulative␣Genre␣Effects", "
                              Regularized␣Movie,␣User␣and␣Averaged␣Genre␣Effects", "
                              Regularized␣Movie,␣User␣and␣Penalized␣Average␣Genre␣Effects",
                              "Time␣Effects␣+␣Regularized␣Movie,␣User␣and␣Cumulative␣Genre␣
                              Effects", "Time␣Effects␣+␣Regularized␣Movie,␣User␣and␣Averaged
                              ␣Genre␣Effects", "Time␣Effects␣+␣Regularized␣Movie,␣User␣and␣
                              Penalized␣Average␣Genre␣Effects␣(Linear)","Time␣Effects␣+␣
```

```
                        Regularized␣Movie,␣User␣and␣Penalized␣Average␣Genre␣Effects␣(
                        Quadratic)","Time␣Effects␣+␣Regularized␣Movie,␣User␣and␣
                        Penalized␣Average␣Genre␣Effects␣(Cubic)" ,"Movie␣Effects␣over␣
                        time␣+␣Regularized␣Movie,␣User␣and␣Penalized␣Average␣Genre␣
                        Effects", "User␣Effects␣over␣time␣+␣Regularized␣Movie,␣User␣
                        and␣Penalized␣Average␣Genre␣Effects", "Ranger␣Random␣Forest␣(
                        Classification,␣100␣Trees)", "Ranger␣Random␣Forest␣(Regression
                        ,␣100␣Trees)", "Ranger␣Random␣Forest␣(Regression,␣200␣Trees)")
                        , RMSE = c(rmse0,rmse1, rmse2,  rmse3, rmsea,rmse4, rmse5,
                        rmse6, rmse7, rmse8, rmse9, rmse12,rmse13,rmse10, rmse11,
                        rmse14, rmse15, rmse16))
print(results, right = F)
```

```
##    method
## 1  Only the Mean
## 2  Movie Effects
## 3  Movie and User Effects
## 4  Movie, User and Genre Effects
## 5  Regularized Movie and User Effects
## 6  Regularized Movie, User and Cumulative Genre Effects
## 7  Regularized Movie, User and Averaged Genre Effects
## 8  Regularized Movie, User and Penalized Average Genre Effects
## 9  Time Effects + Regularized Movie, User and Cumulative Genre Effects
## 10 Time Effects + Regularized Movie, User and Averaged Genre Effects
## 11 Time Effects + Regularized Movie, User and Penalized Average Genre Effects (
   Linear)
## 12 Time Effects + Regularized Movie, User and Penalized Average Genre Effects (
   Quadratic)
## 13 Time Effects + Regularized Movie, User and Penalized Average Genre Effects (
   Cubic)
## 14 Movie Effects over time + Regularized Movie, User and Penalized Average Genre
    Effects
## 15 User Effects over time + Regularized Movie, User and Penalized Average Genre
   Effects
## 16 Ranger Random Forest (Classification, 100 Trees)
## 17 Ranger Random Forest (Regression, 100 Trees)
## 18 Ranger Random Forest (Regression, 200 Trees)
##    RMSE
## 1     1.0601256
## 2     0.9441217
## 3     0.8665775
## 4     0.8668083
## 5     0.8660142
## 6     0.8661324
## 7     0.8660425
## 8     0.8660142
## 9     0.8660986
## 10    0.8660108
## 11    0.8659792
## 12    0.8659758
## 13    0.8659759
## 14    0.9315572
## 15  291.8810510
## 16    2.3068846
## 17    1.0260201
## 18    1.0256018
```

```
results$method[which.min(results$RMSE)]

## [1] "Time Effects + Regularized Movie, User and Penalized Average Genre Effects
    (Quadratic)"
```

As can be seen, the Quadratic Time effects model provides the lowest RMSE. This model can now be trained on the edx set and tested on the validation set. The regularization term can be assumed to be the same for the whole training set as *edx_train* and *edx_validation* account for 90% of the data. Using the previous regularization term of 8.1, the model can now be trained on the entire set:

```
# Saving all data frames to free up RAM
stuff <- ls()

ind <- which(!stuff %in% c("edx", "validation", "rmses_2", "params_lambda", "stuff"
    , "join_genres"))

stuff <- stuff[ind]

# save(list = stuff,file = "all_non_main.RData")
remove(list = stuff)

#Storing the lambda parameter
lambda <- params_lambda[which.min(rmses_2)]

invisible(gc())
# training on all of edx
edx_2 <- edx


save(edx, file = 'edx.Rdata')
remove(edx)

#maximum number of genres in edx set


edx_2_genrecount <- 1+ str_count(edx_2$genres, "\\|") %>% max()


edx_2_genrecols <- paste0("genre", 1:edx_2_genrecount)
edx_2_numcols <- paste0("n", 1:edx_2_genrecount)
edx_2_biascols <- paste0("bias", 1:edx_2_genrecount)

invisible(gc())
# edx_2 can't be split directly due to ram constraints so it will be divided first
rows_edx <- nrow(edx_2)
half_rows <- round(rows_edx/2,0)
edx_2_a <- edx_2[(1:half_rows),]
edx_2_b <- edx_2[((half_rows+1):rows_edx),]
remove(edx_2)
edx_2_a <- edx_2_a %>% separate(data = ., col = genres, into = edx_2_genrecols, sep
    = "\\|")

## Warning: Expected 8 pieces. Missing pieces filled with `NA` in 4499910 rows [1,
    2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].

edx_2_b <- edx_2_b %>% separate(data = ., col = genres, into = edx_2_genrecols, sep
    = "\\|")
```

```
## Warning: Expected 8 pieces. Missing pieces filled with `NA` in 4499889 rows [1,
   2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].

edx_2 <- bind_rows(edx_2_a, edx_2_b)
# Mean

mu <- mean(edx_2$rating)
# Recreating Movie Bias
movie_bias <- edx_2 %>% group_by(movieId) %>% summarize(test = sum(rating - mu), n
   = n()) %>% mutate(effect_bi = test/(n+lambda)) %>% select(-test, -n)


edx_2 <- edx_2 %>% left_join(movie_bias, by = "movieId")
# Recreating user bias

user_bias <- edx_2 %>% group_by(userId) %>% summarize(test = sum(rating - mu -
   effect_bi), n = n()) %>% mutate(effect_bu = test/(n+lambda)) %>% select(-test, -
   n)

edx_2 <- edx_2 %>% left_join(user_bias, by = "userId")

#Recreating genre bias
genre_effects_reg <- list()
ind <- -1 + which(colnames(edx_2) == 'genre1')
for(i in 1:edx_2_genrecount){

  genre_effects_reg[[i]] <- setNames(edx_2 %>% group_by_at(ind+i) %>%
                                       summarize(test = sum(rating - mu - effect_bi -
                                           effect_bu), n = n()) %>% mutate(effect_bg =
                                           test/(n+lambda)) %>% select(-test),
                                     c(edx_2_genrecols[i],edx_2_numcols[i], edx_2_
                                         biascols[i]))

}



total_genre_effects <- Reduce(join_genres, genre_effects_reg)
total_genre_effects[is.na(total_genre_effects)] <- 0

remove(genre_effects_reg)
n_indices <- which(colnames(total_genre_effects) %in% edx_2_numcols)

bias_indices <- which(colnames(total_genre_effects) %in% edx_2_biascols)

dat <- sapply(1:edx_2_genrecount, function(i){
  biascol <- total_genre_effects[, bias_indices[i]]
  numcol <-  total_genre_effects[, n_indices[i]]
  return(biascol*numcol)
})

genre_bias_regularized <- data.frame(genre = total_genre_effects$genre1, effect_bg
   = rowSums(dat)/rowSums(total_genre_effects[, n_indices]))
```

```
edx2_genre_bias <- matrix(ncol = edx_2_genrecount, nrow = nrow(edx_2))

invisible(gc())

for(i in 1:edx_2_genrecount){

 foo <- merge.data.frame(edx_2, genre_bias_regularized, by.x = (ind + i), by.y = 1,
     all.x = TRUE)
edx2_genre_bias[,i] <- foo$effect_bg
remove(foo)
invisible(gc())
}

edx2_genre_bias <- apply(edx2_genre_bias,1,FUN = function(u){sum(u, na.rm = TRUE)/(
   sum(!is.na(u)) + lambda)})

edx_2 <- edx_2 %>% mutate(effect_bg = edx2_genre_bias)
remove(edx2_genre_bias)

edx_2 <- edx_2 %>% mutate(error = rating - mu - effect_bi - effect_bu - effect_bg)

edx_2 <- edx_2 %>% mutate(t2 = timestamp^2)

quadratic_date <- lm(error ~ timestamp + t2, data = edx_2)


# testing on validation

validation_2 <- validation

validation_2_genrecount <- 1 + str_count(validation_2$genres, "\\|") %>% max()

validation_2_genrecols <- paste0("genre", 1:validation_2_genrecount)


validation_2 <- separate(data = validation_2, col = genres, into = validation_2_
   genrecols, sep = "\\|")

## Warning: Expected 8 pieces. Missing pieces filled with `NA` in 999972 rows [1,
   2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].

validation_2_time <- predict(quadratic_date, newdata = data.frame(timestamp =
   validation_2$timestamp, t2 = (validation_2$timestamp)^2))

remove(quadratic_date)

validation_index <- -1 + which(colnames(validation_2) == "genre1")

validation_genre_bias <- matrix(ncol = validation_2_genrecount, nrow = nrow(
   validation_2))

for(i in 1:validation_2_genrecount){

 foo <- merge.data.frame(validation_2, genre_bias_regularized, by.x = (validation_
    index + i), by.y = 1, all.x = TRUE)
validation_genre_bias[,i] <- foo$effect_bg
}


validation_final_genre_bias <- apply(validation_genre_bias,1,FUN = function(u){sum(
    u, na.rm = TRUE)/(sum(!is.na(u)) + lambda)})

yhat <- mu + {validation %>% left_join(movie_bias, by = 'movieId') %>% left_join(
    user_bias, by = 'userId') %>% mutate(total = effect_bi + effect_bu) %>% .$total}
     + validation_2_time + validation_final_genre_bias

# identical(validation_2$rating, validation$rating)
# the two vectors are the same so it doesn't make a difference
# as to which one is used to calculate the RMSE.

rmse_final <- RMSE(yhat, validation_2$rating)
rmse_final
```

```
## [1] 0.864883
```

This is a very good result.

Conclusion

This report uses the Movielens 10-M dataset to create a movie recommendation system. Methods considered include regularized and unregularized generative models, which consider the effects of movies, users and genres. Time effects are also considered using timestamps (and other grouping variables), and multiple degree polynomial models are implemented. Even though they display a poor correlation coefficient, they provide a reliable baseline for modeling errors over time, as well as a low p-value.

Decision trees using the ranger library, (albeit to not the same effectiveness as the generative model) are also an effective solution. However, multiple trees have to be trained (which can take hours, if not days), and saving the model can crash R (this happened on my R instance if the number of trees was, for example, 400).

The main limitation of this model is hardware limitations, since R stores all objects (function, data frames and variables) through RAM.

Potential solutions include a cloud-based implementation, such as Google Colaboratory, or a Hadoop implementation.

Another limitation of the model is the implementation of genre effects. However, as the biases have been averaged and penalized, this is not an issue.

Time effects could also be modeled with smoothing functions such as knn3 or loess, however, these are demanding computations that are too slow on a standard laptop/desktop. Integer overflows are also a risk, especially with knn3.

In all, the final model uses a recommendation system with movie, user and genre effects and a quadratic time effects model to give a final RMSE of 0.864883.

```
# remove all Rdata files

file.remove('importants.Rdata')
```

```
## [1] TRUE
```

```
file.remove('linear_date.Rdata')
```

```
## [1] TRUE
```

```
# file.remove('all_non_main.Rdata')
file.remove('edx.Rdata')
```

```
## [1] TRUE
```

Bibliography

https://stackoverflow.com/questions/34344214/how-to-join-multiple-data-frames-using-dplyr

https://rafalab.github.io/dsbook https://rafalab.github.io/dsbook/large-datasets.html#recommendation-systems

https://stackoverflow.com/questions/68259910/creating-a-grouped-summary-of-linear-or-nonlinear-models-to-join-to-a-table-an

https://bookdown.org/yihui/rmarkdown-cookbook/text-width.html

https://stackoverflow.com/questions/2470248/write-lines-of-text-to-a-file-in-r

https://adv-r.hadley.nz/functionals.html

https://bookdown.org/yihui/rmarkdown-cookbook/text-width.html

https://learning.edx.org/course/course-v1:HarvardX+PH125.9x+1T2021/block-v1:HarvardX+PH125.9x+1T2021+type@sequential+block@e8800e37aa444297a3a2f35bf84ce452/block-v1:HarvardX+PH125.9x+1T2021+type@vertical+block@e9abcdd945b1416098a15fc95807b5db