Movie Lens Capstone Project

Kevin Nelson

April 13, 2021

INTRODUCTION

The objective of this project is to develop a movie recommendation system based on a data set that provides information on previously made ratings by users. For each rating in the data set, there are corresponding details on movie identification, release year, genre, and when the rating was made. Using concepts and machine learning techniques from the edx coursework, we can use the data provided to extract valuable relationships between the current ratings on file and the characteristics of the associated user, movie, and genre. This report discusses the methods and processes used in creating this recommendation system and includes a few iterations of the model, though only the final model is set up to run if the reviewer desires to do so.

The Movie Lens data set provided ratings information from 1995 to 2008 for films released from 1915 to 2008. Each rating defines a user who made the rating, which film the rating is for, what date and time the rating was made, and what genre the movie is considered. The movie title is also included with its release year attached within the same column.

```
# Capstone Project - MovieLens  ------------------------------------------------

# Kevin Nelson
# Submitted Friday April 13, 2021
# movieLens_KNelson.R

# For a more detailed description of this project, please refer to the .Rmd file
# titled moviesLens_KevinNelson.Rmd.
# That .Rmd file expands on the code presented in this .R file.

# The objective was to design a program that predicts movie ratings and calculates the
# resulting RMSE. With the dataset provided, there are several variables we could
# analyze to develop this model.

# Factors to potentially consider
#  - Average rating of the movie
#  - Average rating of genre
#  - User's preference towards a genre
#  - Average rating of all movies (for movies with low ratings totals)
#  - When the rating was made
#  - When the movie was released
#  - Rating time in relation to release time
#  - Genre and release year as variables are somewhat baked into the movie's rating
#    already, may be better to relate these things to the user's preference instead
```

```
# Not all factors will be considered for a variety of reasons.
# For example, rating time in relation to release time is not helpful since ratings
# only took place in the last 15 years of the range of movie release years.

# The ratings go from 0.5 to 5.0 in increments of 0.5. Our model should put a cap on
# predictions so they can not fall below/above this min/max.

##############################################################################################
```

Libraries

This section loads the libraries required to run this code and install the packages if they are not already installed. The necessary libraries are *caret*, *data.table*, *dplyr*, *ggplot2*, *ggrepel*, *gtools*, *lubridate*, *stringr*, and *tidyverse*.

```
### INTRODUCTION

# Load Libraries ----------------------------------------------------------

## Caret package
if(!require(caret)){
  install.packages("caret")
  library(caret)
}
## Data.table package
if(!require(data.table)){
  install.packages("data.table")
  library(data.table)
}
## Dplyr package
if(!require(dplyr)){
  install.packages("dplyr")
  library(dplyr)
}
## Ggplot2 package
if(!require(ggplot2)){
  install.packages("ggplot2")
  library(ggplot2)
}
## Ggrepel package
if(!require(ggrepel)){
  install.packages("ggrepel")
  library(ggrepel)
}
## Gtools package
if(!require(gtools)){
  install.packages("gtools")
  library(gtools)
}
## Lubridate package
if(!require(lubridate)){
  install.packages("lubridate")
  library(lubridate)
```

```
}
## Stringr package
if(!require(stringr)){
  install.packages("stringr")
  library(stringr)
}
## Tidyverse package
if(!require(tidyverse)){
  install.packages("tidyverse")
  library(tidyverse)
}
```

Data

This section loads in the edx and validation data set. The code seen here has been provided with the edx Capstone course and has only been slightly modified to remove unnecessary lines.

```
# Load Movies Dataset ----------------------------------------------------
# This section is the code provided from the class with the steps to download the edx and
# validation datasets that will be used throughout this project.

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

# Note: this process could take a couple of minutes

# 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")
```

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

# Make sure userId and movieId in validation set are also in edx set
validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

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

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

Basic Functions, Logicals, and Variables

The following chunk of code defines several logicals that do not need to be altered unless the reviewer is interesting in running processes from this report that are not included in the final model. There were three general methods that were tested throughout. One of them was found to overtrain the data and excluded, though the code and explanation from that method is still included to illustrated the thought process. If the reviewer would like to run the model in that form, turn the *original_tuning* variable to TRUE and the remaining four logicals in lines 187, 191, 197, and 198 to FALSE.

Another method was attempted where unique regularization parameters for each variable involved in the calculation but it was found to be less accurate than a single lambda. That model is also excluded in the current version, but again, the code is included here for review. If the reviewer would like to run the model in that form, turn the *simplified_tuning* variable to TRUE for complete model tuning or *noTuning_multipleLambdas* to TRUE for operation with the previously determined lambdas and whichever remaining four logicals to FALSE.

The provided version of the code sets the *noTuning_oneLambda* to TRUE, which runs the model in its finalized form. If the reviewer would like to run the tuning process that determined the lambda parameters defined in the final model, turn the *modified_tuning* variable in line 195 to TRUE and the *noTuning_oneLambda* to FALSE.

This section of code also defines the RMSE function, defines the rating average for all movies in the data set.

An extra column was also added to the data set by extracting the release year from the title column using regular expressions and the *mutate* function, since that variable will be considered for the recommendation system.

```r
# Define basic variables and functions ---------------------------------

## None of the following variables need to be set to TRUE to review this code and project.
## Optimized values from tuning have been included as alternatives to these logical
# variables.
```

```
# Set to true if tuning based on original plan (data set and user level lambdas)
original_tuning = FALSE

# Set to true if tuning simplified version (single lambda on data set level variables)
simplified_tuning = FALSE

# Set to true if tuning modified simplified version (different lambdas across data set
# level variables)
modified_tuning = FALSE
# Set variable that activates the parts of this code used for tuning.
# Warning: Turning tuning on when reviewing the code will activate some sections of the
# code that are time consuming.

# Logicals to set if not performing tuning
noTuning_oneLambda = TRUE ## FINAL VERSION OF CODE SET TO TRUE
noTuning_multipleLambdas = FALSE

options(dplyr.summarise.inform = FALSE) # Turn off the warnings created by the group_by
# function so to not congest the console during operation.


# Create a function that calculates the RMSE
RMSE <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}

# Create a variable that is the mean of all rating in the edx dataset
mu <- mean(edx$rating)

# Create new column that extracts release year from title column ----------

# The original dataset combined movie title and release year in a string within a
# single column. Release year may be a variable that informs this model and the following
# piece of code extracts that information from the title column and isolates it for
# additional analysis.

# Add column for release year
edx <- edx %>% mutate(release_year = str_extract(title, "(?<=\\()\\d{4}?(?=\\))"))

################################################################################
```

METHODS & ANALYSIS

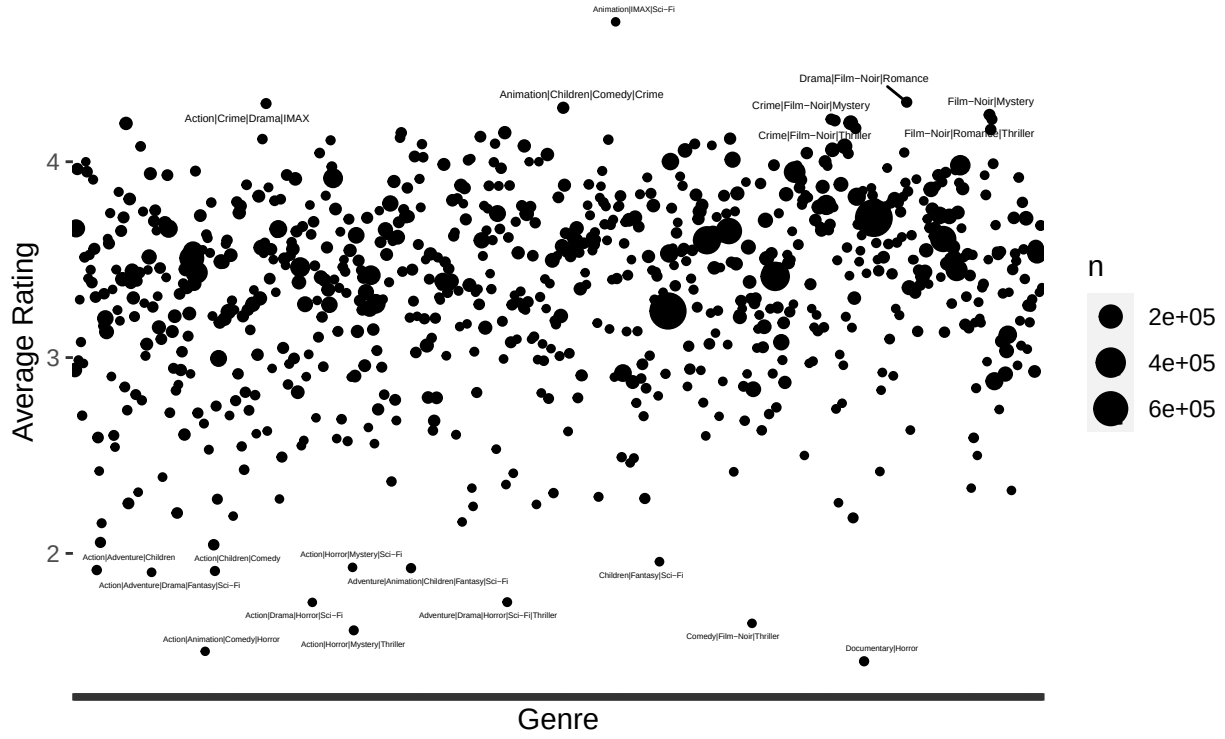
The procedure started by creating indices that will be used to partition the data into training and test sets throughout the code. Cross validation will be used on several occasions in this project and the indices are created to reflect that with 10 different partitions made. The training set is established here by slicing the data based on the first partition in the collection of indices created. This will be the default training set for the first stages of this project.

```
### METHODS & ANALYSIS

# Create training and test sets from edx dataset -------------------------
```

```r
# Create a separate test and training set from the edx dataset
#  The test set has been sliced as 10% of the original dataset
set.seed(25, sample.kind = "Rounding")
index <- createDataPartition(y = edx$rating, times = 10, p = 0.10, list = FALSE)

# Cross validation will ultimately will be used but the first partition from the previous
# step will be taken to establish a default training set. Testing sets will be
# established during cross validation.
edx_train <- edx[-index[ , 1], ]
```

We have seen from class that movie quality and user positivity are factors when considering a recommendation system, but there are additional factors we can consider from the provided dataset. Genre can also be included as certain genres can be rated generally higher or lower than average. The graph created in the next section of code explores those variations.

```r
# Test to see variation in ratings across genres --------------------------

# Based on the classwork, it is already known that the user and movie are heavily
# influential on developing an effective rating system. The remaining variables at
# our disposal which also play a role are genre and release year. This section
# explores the variation within those variables.

# Create variable that groups by genre
genreRating <- edx %>%
  group_by(genres) %>%
  summarize(rating = mean(rating), n = n())

# Graph to show ratings per genre
ggplot(data = genreRating, aes(x = genres, y = rating, size = n)) +
  geom_point() +
  geom_text_repel(data = filter(genreRating, rating > 4.2 | rating < 2),
                  aes(label = genres)) +
  theme(axis.text.x = element_blank()) +
  ggtitle(label = 'Average Rating as a Function of Genre',
          subtitle = 'Genres with Average Ratings < 2 & > 4.2 Are Labeled') +
  xlab('Genre') +
  ylab('Average Rating')
```



It is clear there are large variations betweens genres but also that certain genres have received many less or more ratings than others. Because of this, regularization may be required in order to adjust for genres with very small numbers of ratings being overly corrected for based on their sample size. The next few lines of code rank the genres by rating with the number of ratings included.

```
# Examine top and bottom 10 rated genres to see if regularization is required

# Bottom 10
slice_min(genreRating, order_by = rating, n = 10)
```

```
## # A tibble: 10 x 3
##    genres                                      rating     n
##    <chr>                                        <dbl> <int>
##  1 Documentary|Horror                            1.45   619
##  2 Action|Animation|Comedy|Horror                1.5      2
##  3 Action|Horror|Mystery|Thriller                1.61   327
##  4 Comedy|Film-Noir|Thriller                     1.64    21
##  5 Action|Drama|Horror|Sci-Fi                    1.75     4
##  6 Adventure|Drama|Horror|Sci-Fi|Thriller        1.75   217
##  7 Action|Adventure|Drama|Fantasy|Sci-Fi         1.90    57
##  8 Action|Children|Comedy                        1.91   518
##  9 Action|Adventure|Children                     1.92   824
## 10 Adventure|Animation|Children|Fantasy|Sci-Fi   1.92   691
```

```r
# Top 10
slice_max(genreRating, order_by = rating, n = 10)
```

```
## # A tibble: 10 x 3
##    genres                                    rating     n
##    <chr>                                      <dbl> <int>
##  1 Animation|IMAX|Sci-Fi                       4.71     7
##  2 Drama|Film-Noir|Romance                     4.30  2989
##  3 Action|Crime|Drama|IMAX                     4.30  2353
##  4 Animation|Children|Comedy|Crime             4.28  7167
##  5 Film-Noir|Mystery                           4.24  5988
##  6 Crime|Film-Noir|Mystery                     4.22  4029
##  7 Film-Noir|Romance|Thriller                  4.22  2453
##  8 Crime|Film-Noir|Thriller                    4.21  4844
##  9 Crime|Mystery|Thriller                      4.20 26892
## 10 Action|Adventure|Comedy|Fantasy|Romance     4.20 14809
```

```r
# It's clear there is variation throughout the genres variable and that regularization
# is required on the data level. But can we go one level deeper? It is intuitive to
# suspect that each user has their own unique preferences and may prefer some genres
# more than others. Let's take a look at a single user to see how their preferences
# may vary.
```

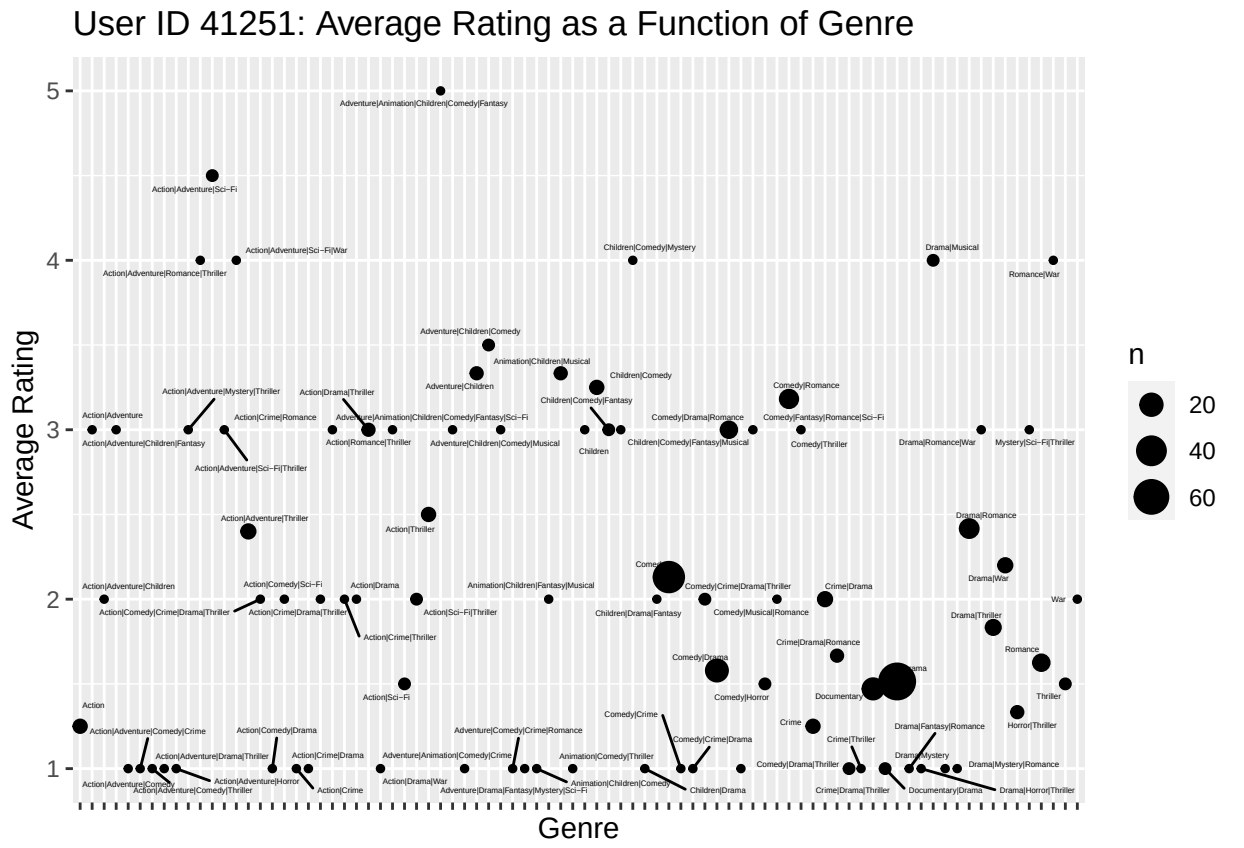
We can clearly see there is variation throughout the genres variable and that regularization is required on the data level. But can we go one level deeper? It is intuitive to suspect that each user has their own unique preferences and may like some genres more than others. Let's take a look at a single user to see how their preferences may vary.

```r
# Filter out users who have less than 200 ratings overall
edx_filter <- edx %>%
  group_by(userId) %>%
  summarize(total_user_ratings = n()) %>%
  filter(total_user_ratings > 200)

# Pull a sample from the data set
set.seed(23, sample.kind = "Rounding")
```

```
## Warning in set.seed(23, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```r
edx_sample <- sample(edx_filter$userId, 1)

# Plot that user's ratings by genre with this code
edx %>%
  filter(userId == edx_sample) %>%
  group_by(genres) %>%
  summarize(rating = mean(rating), n = n()) %>%
  ggplot(aes(x = genres, y = rating, size = n)) +
  geom_point() +
  geom_text_repel(aes(label = genres), size = 1) +
  theme(axis.text.x = element_blank()) +
  ggtitle(paste0('User ID ', edx_sample, ': Average Rating as a Function of Genre')) +
```

```
  xlab('Genre') +
  ylab('Average Rating')
```

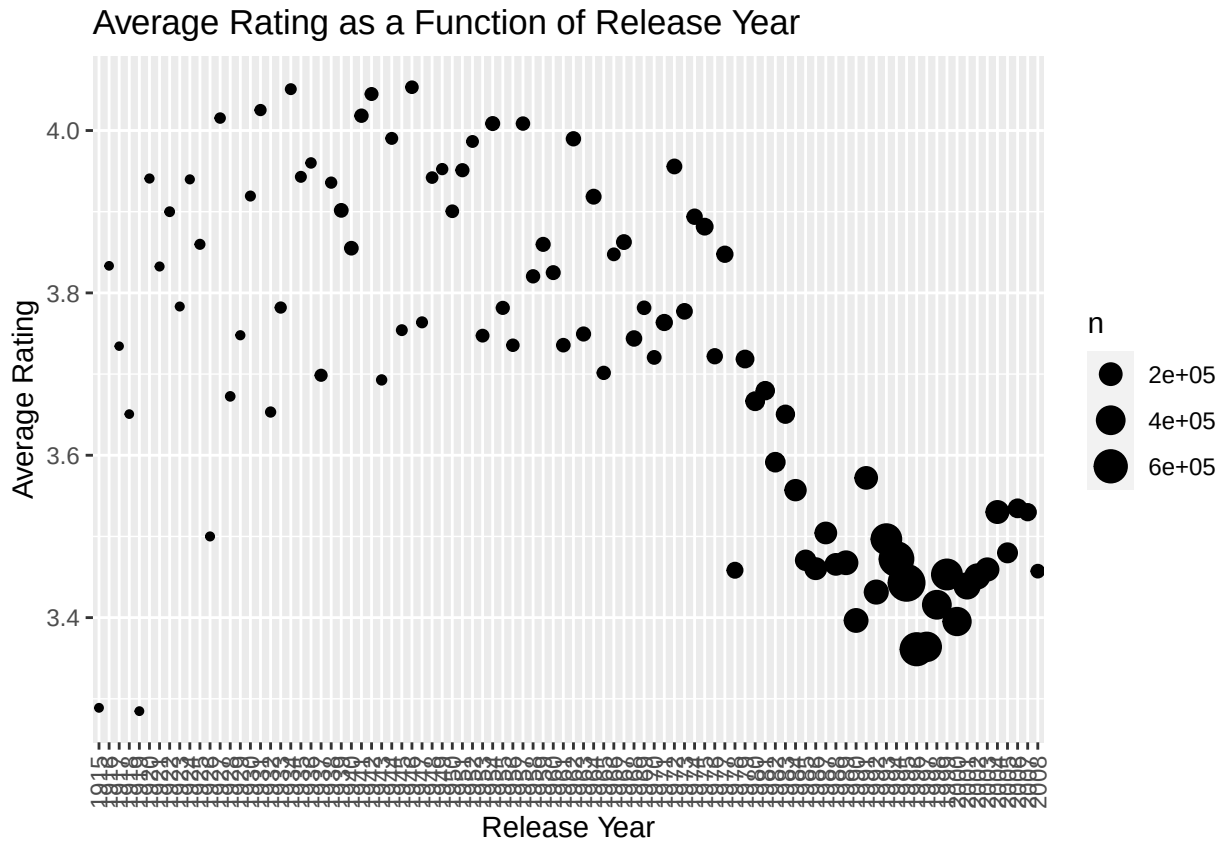


It's only a single user so broad conclusions can't be made, but the varition in ratings from this one user does support the claim that user preferences for genres is real and can be integrated into our model.

In the same way that user's may have preferences for certain genres, they also may have preferences for movies of a certain time. Some people like newer movies with their increased pacing and improved graphics while others may be nostalgic for the style from the 1970s and 1980s. It is worth exploring the variation those preferences create in our data set to see ifit can be exploited to improve our model.

```
# Test to see variation across release years ------------------------------

# In the same way that user's may have preferences for certain genres, they also may have
# preferences for movies of a certain time. Some people like newer movies with their
# increased pacing and improved graphics while others may be nostalgic for the style from
# the 1970s and 1980s. It is worth exploring the variation those preferences create in our
# data set to see if it can be exploited to improve our model.

# Create variable that groups ratings by release year
releaseRating <- edx %>%
  group_by(release_year) %>%
  summarize(release_rating = mean(rating), n = n())

# Plot average rating vs. release year with point size corresponding to number of ratings
ggplot(data = releaseRating, aes(x = release_year, y = release_rating)) +
```

```
  geom_point(aes(size = n)) +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1)) +
  ggtitle('Average Rating as a Function of Release Year') +
  xlab('Release Year') +
  ylab('Average Rating')
```



Perhaps unexpected, but there is definite variation in average rating by release year. This next section of codes ranks the best and worst rated release years while including the number of ratings that contribute to that.

```
# Examine top and bottom 10 rated genres to see if regularization is required

# Bottom 10
slice_min(releaseRating, order_by = release_rating, n = 10)
```

```
## # A tibble: 10 x 3
##    release_year release_rating      n
##    <chr>                 <dbl>  <int>
##  1 1919                   3.28    158
##  2 1915                   3.29    180
##  3 1996                   3.36 593518
##  4 1997                   3.36 429751
##  5 2000                   3.40 382763
##  6 1990                   3.40 230409
##  7 1998                   3.42 402187
```

```
##  8 1992                   3.43 236834
##  9 2001                   3.44 305705
## 10 1995                   3.44 786762
```

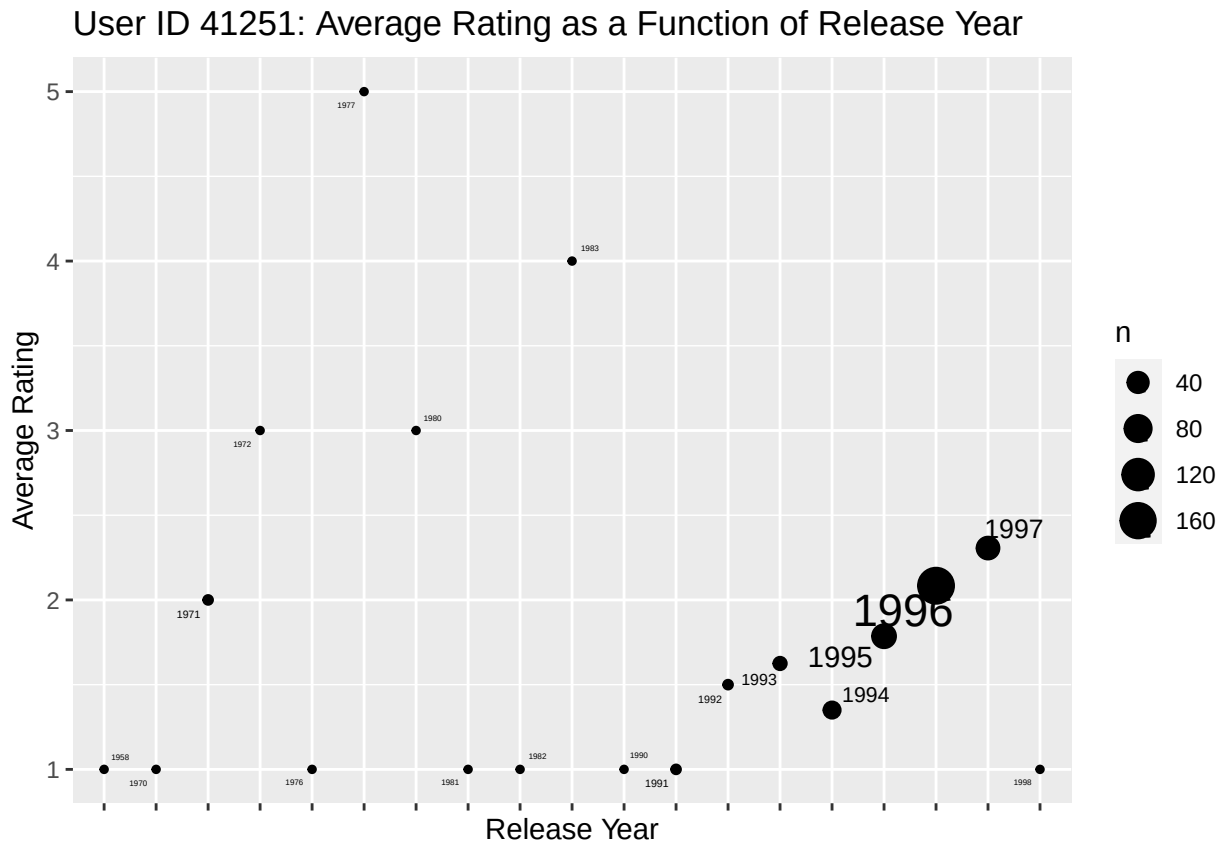
```
# Top 10
slice_max(releaseRating, order_by = release_rating, n = 10)
```

```
## # A tibble: 10 x 3
##    release_year release_rating     n
##    <chr>                 <dbl> <int>
##  1 1946                   4.05 16882
##  2 1934                   4.05  5954
##  3 1942                   4.04 20051
##  4 1931                   4.03  7669
##  5 1941                   4.02 23883
##  6 1927                   4.02  4133
##  7 1954                   4.01 30089
##  8 1957                   4.01 24557
##  9 1944                   3.99 11918
## 10 1962                   3.99 33622
```

```
# Regularization at the data set level doesn't appear to be necessary based on the number
# of ratings for each but let's explore if going an additional level doum, to the user
# preference, may be worth inclusion in our model using the same user from the previous
# section.
```

While a couple of the release years have lower numbers than the rest, the category as a whole probably does not require regularization. Though regularization at the data set level doesn't appear to be necessary based on the sample size for each release, let's explore if going an additional level down, to the user preference, may be worth inclusion in our model using the same user from the previous section. The following section plots the ratings of a single user by release year.

```
# Plot ratings by release year for a single user
edx %>%
  filter(userId == edx_sample) %>%
  group_by(release_year) %>%
  summarize(rating = mean(rating), n = n()) %>%
  ggplot(aes(x = release_year, y = rating, size = n, label = release_year)) +
  geom_point() +
  geom_text_repel() +
  theme(axis.text.x = element_blank()) +
  ggtitle(paste0('User ID ', edx_sample,
                 ': Average Rating as a Function of Release Year')) +
  xlab('Release Year') +
  ylab('Average Rating')
```




```
# Looking a single user is not a sufficient sample size to correctly say that all users
# have a preference for particular eras of movies, but it is reasonable to suspect that
# be the case. We can use regularization at the user level to correct for this.
```

There is clear variation with this user and while looking a single user is not a sufficient sample size to correctly say that all users have a preference for particular eras of movies, it is reasonable to suspect that be the case. We can use regularization at the user level to correct for this.

Regularization on the Data and User Level

NOT A PART OF FINAL MODEL

Based on the exploration from the previous setting, we will try to develop and optimize a recommendation system that considers a movie's average rating on a data set level and genre and release year ratings based on each individual user's preference. The information uncovered in the previous section will lead us to regularize for user release year preference based on non-regularized release year average ratings and user genre preference based on regularized genre average ratings. With four lambda parameters under consideration, it is intuitive to suspect that each will be unique in an optimized version of this model. The next section aims to perform that optimization.

The aspects of the regularization calcalution that do not require to be included in the loop are performed here and added to the data table. The added columns for release year are user specific values relative to the mean of the release year. The same will be done for genre, but due to genre requiring regularization at both the data set and user level, both calculations must take place in the tuning loop function.

```r
# Regularization Tuning ------------------------------------------------

# Based on the exploration from the previous setting, we will try to develop and optimize
# a recommendation system that considers a movie's average rating on a data set level and
# genre and release year ratings based on each individual user's preference. The
# information uncovered in the previous section will lead us to regularize for user
# preference based on non-regularized release year average ratings and user preference
# based on regularized genre average ratings. With four lambda parameters under
# consideration, it is intuitive to suspect that each will be unique in an optimized
# version of this model. The next section aims to perform that optimization.

# The following section will only be run if the original_tuning variable is set to TRUE
# at the beginning of the code.
if (original_tuning == TRUE){

  # This section creates the numerator and part of the
  # denominator for the regularization equation
  # in an effort to reduce the burden on the loop.

  # Movie
  reg_movie_sum <- edx_train %>%
    group_by(movieId) %>%
    summarize(movie_s = sum(rating - mu),  movie_m = mean(rating), movie_n = n())

  # Genre (on data set level)
  reg_genre_sum <- edx_train %>%
    group_by(genres) %>%
    summarize(genre_s = sum(rating - mu),  genre_m = mean(rating), genre_n = n())

  # Release year averages
  reg_relyr_mean <- edx_train %>% # Acquire the average ratings per release year to set up
    group_by(release_year) %>%    # the regularization calculations based on them
    summarize(release_mean = mean(rating)) %>%
    ungroup()

  # Release year averages by user
  reg_relyr_sum <- edx_train %>%
    left_join(reg_relyr_mean, by = 'release_year') %>%
    group_by(userId, release_year) %>%  # Regularization (on data set level)
    summarize(relyr_s = sum(rating - release_mean),
              relyr_m = mean(rating),
              relyr_n = n())

  edx_train <- edx_train %>%  # Add the new columns to the default training set
    left_join(reg_movie_sum, by = 'movieId') %>%
    left_join(reg_genre_sum, by = 'genres') %>%
    left_join(reg_relyr_sum, by = c('userId', 'release_year'))
}
```

Not included in final model, left in report to illustrate thought process.

It stands to reason that the optimal version of this model could possibly be one where there is a unique lambda parameter for each variable being regularized. Four variables are being regularized in this model - movie effect, overall genre effect, genre effect based on user preference, and release year effect based on

user preference. The original plan for this model was to tune all of these lambda parameters simultaneously considering the possibility that the optimal lambda parameter for each variable would be different. A lambda range was developed from 0 to 10 in increments of 0.5 and a tuning function created that would test all permutations of that range across the four variables. Due to the large number of permutations possible for that range, the computation time required to perform that optimal tuning was unreasonably time consuming. It was apparent during testing, however, that three lambda parameters optimized to the same number consistently. The parameter regularizing the user effect per genre was typically convergin lower on the lambda range, independent of the other three parameters. In an effort to reduce computation time, the tuning function was simplified to consider permutations of length two instead of four, with the lambda parameter staying the same across the three variables that trended together. The original tuning function can be found in the appendix of this report. The updated tuning function is defined in the next section and was used to optimize the lambda parameters.

If readers of this report want to run the model with this version of tuning included, return to the second section of this script and change the "original_tuning" variable to TRUE. It is not recommended to include tuning when reviewing this code as the optimization alone requires several hours of computation time.

Lambda Tuning Function for Regularization on Data and User Level

NOT INCLUDED IN FINAL MODEL

This section defines a function that will intake lambdas and a training set (for cross validation purposes) and output an RMSE. It was originally designed to process four different lambdas, going through each permutation within the desired range. Unfortunately, the computation for that was on the order of days, if not weeks, and the code was streamlined to intake two different lambdas, one for the parameters being tuned on the data set level and one for the parameters being tuned on the user level.

This function is ultimately nested into the next function defined in this code, designed to use cross validation to train and test the model on different slices of the training set.

```
# Lambda Parameters Tuning Function ---------------------------------------

# This section defines a function that will intake lambdas and a training set and output
# an RMSE. It was originally designed to process four different lambdas, going through
# each permutation within the desired range. Unfortunately, the computation for that was
# on the order of days, if not weeks, and the code was streamlined to intake two
# different lambdas, one for the parameters being tuned on the data set level and one
# for the parameters being tuned on the user level.

# This function is ultimately nested into the next function defined in this code,
# designed to use cross validation to train and test the model on different slices of
# the training set.

# INPUTS
# - lambdas: a single regularization parameter value for tuning
# - setLevel: determines whether testing takes place on data or user level
#             TRUE for data level lambdas, FALSE for user level lambdas
# - locked_lambda: defined value for lambdas not specific for testing in setLevel
# - cross_val_train: training set from one of the cross validation slices

# OUTPUTS
# - RMSE for the given parameters

tuning_function <- function(lambdas, setLevel, locked_lambda, cross_val_train){
```

```
  # TRUE if training the lambda used for the
  # dataset level regularization (movies, genre #1)
  if(setLevel == TRUE){
    tune_lambda1 <- lambdas
    tune_lambda2 <- locked_lambda
  # FALSE if training the lambda used for the
  # user level regularization (genre #2, release year)
  } else {
    tune_lambda1 <- locked_lambda
    tune_lambda2 <- lambdas
  }

  # Create a temporary training set based on the inputted cross validation selection
  # and modify data frame to include regularization parameters based on previously
  # added columns.
  cross_val_temp <- cross_val_train %>%
    mutate(movie_b = movie_s / (movie_n + tune_lambda1),
           relyr_b = relyr_s / (relyr_n + tune_lambda2),
           genre_reg = genre_m + (genre_s / (genre_n + tune_lambda1)))

  # A second level of regularization is required for genres, accounting for user
  # preference based on the regularized average rating of each genre, calculated
  # in the previous step.
  user_reg_genre <- cross_val_temp %>%
    group_by(userId, genres) %>%
    summarize(genre_b = sum(rating - genre_reg)/(n() + tune_lambda2))

  # Add to overall table and create predictions
  cross_val_temp <- cross_val_temp %>%
    left_join(user_reg_genre, by = c('userId', 'genres')) %>%
    mutate(prediction = mu + movie_b + genre_b + relyr_b)

  # Keep track of progress with this line.
  print(paste0('Iteration complete with lambdas ',
               tune_lambda1, ' and ', tune_lambda2, '.'))

  # Calculate and return RMSE for an iteration
  RMSE(cross_val_temp$rating, cross_val_temp$prediction)
}
```

Not included in final model, left in report to illustrate thought process.

The edx dataset was previously split into a training set, with the model optimization operating exclusively on the training set. During optimization, cross validation was performed on the training set in an effort to compensate for overtraining due to the random error potentially introduced by limitations in the dataset and the segmentations required. To do this, 5 samples of 60% partitions were created from the training set. Optimization was performed on the 60% portion of the partition and testing was performed on the 40% portion. It is acknowledged that it is typically preferred to create an optimization portion of at least 80% of the training set, but that number was reduced due to concerns on computation time.

For each optimization partition, all permutations of the lambda parameters were processed in the tuning function and the optimal pair was selected based on which minimized the RMSE. This combination of lambda parameters was then put through the same process on the testing partition (the 40% portion of the training set). The five resulting RMSES were saved and averaged to gain an understanding of the average error

and the individual lambda parameters that create the lowest RMSE were selected as the final regularization parameters.

Cross Validation Function for Regularization on Data and User Level

NOT INCLUDED IN FINAL MODEL

The following chunk of code creates different regularization parameter values on both the user and data level and processes them through the tuning function defined previously. Using k = 10 for the number of cross validation sets, the optimal settings were determined for each instance of cross validation. Because of the design of the functions, only one of the two parameters (user level or data level) could be tested at time. For this reason, the user level lambda was locked at 3 for training the system with the data level regularization parameters and the optimal value from that was then plugged in as the locked lambda when the user level parameter was tuned.

```
# Run Optimization Functions ---------------------------------------------

k = 10 # Set the number of cross validation samples to 10

if (original_tuning == TRUE){
  userLevel_lambdas <- seq(0.5, 15, 0.5) # Set range of lambdas for user level testing
  dataLevel_lambdas <- seq(5, 150, 5)    # Set range of lambdas for data level testing
  # The ranges above may seem arbitrary but were deemed appropriate based on
  # experimentation with the functions and resulting RMSEs.

  # The current version of the functions can only test a single lambda parameter at a
  # time to save on computation time. Because of this, one has to be locked to start
  # the process. The data level lambdas were selected for tuning first and the user
  # level lambda was set at 3.

  # Tune the data level lambdas
  dataLevel_tune <- tuning_loop(tuning_lambdas = dataLevel_lambdas,
                                k = k,
                                lambda_dataset = TRUE,
                                locked_lambda = 3)

  # Save the average RMSE across cross validation sets
  dataLevel_RMSE <- mean(dataLevel_tune$RMSE)
  # Find the best lambda setting from the output for input into user level tuning
  finalData_lambda <- dataLevel_tune$Lambdas[which.min(dataLevel_tune$RMSE)]

  # Tune the user level lambdas with the tuned data level lambda from the previous step
  userLevel_tune <- tuning_loop(tuning_lambdas = userLevel_lambdas,
                                k = k,
                                lambda_dataset = FALSE,
                                locked_lambda = finalData_lambda)

  # Save the average RMSE across cross validation sets
  userLevel_RMSE <- mean(userLevel_tune$RMSE)
  # Find the best user level lambda
  finalUser_lambda <- userLevel_tune$Lambdas[which.min(userLevel_tune$RMSE)]

} else {
  finalData_lambda <- 60   # Data level lambda as determined by optimization
```

```
  finalUser_lambda <- 1.5  # User level lambda as determined by optimization
}
```

Not included in final model, left in report to illustrate thought process.

When optimization had been completed, the lambda parameters were determined to be 8 for movie effect, genre effect, and user preferred release year effect, and 1.5 for the user preferred genre effect. The complete model can now be applied to the test set. The original test set is used but once again, cross validation is applied to obtain the most accurate RMSE for the model. 5 different test sets (4 additional ones to the original test set) were processed and the RMSEs were averaged to provide the final estimate.

Cross Validation to Test Tuned Model on Different Slices of Training Set

NOT INCLUDED IN FINAL MODEL

Now that the model has been tuned and the regularization parameters determined, we can use cross validation on the test sets previously created to find a rough estimate of our RMSE. This next chunk of code performs that cross validation testing on the 10 different sets of data and creates the results.

```
# Cross Validation to Test Optimized Model on Different Sets --------------

if (original_tuning){
  # Update default training set by performing process with the final lambda
  # parameters as determined in last step
  edx_train_complete <- edx_train %>%
    mutate(movie_b = movie_s / (movie_n + finalData_lambda),
           relyr_b = relyr_s / (relyr_n + finalUser_lambda),
           genre_reg = genre_m + (genre_s / (genre_n + finalData_lambda)))

  user_reg_genre <- edx_train_complete %>%
    group_by(userId, genres) %>%
    summarize(genre_b = sum(rating - genre_reg)/(n() + finalUser_lambda))

  edx_train_complete <- edx_train_complete %>%
    left_join(user_reg_genre, by = c('userId', 'genres'))

  # Columns to add to data set
  # - movie_b: regularization parameter for each individual movie
  # - genre_b: regularization parameter for genre preference for each user
  # - relyr_b: regularization parameter for release year preference for each user

  # Isolate the columns desired and isolate the unique rows
  movie_fct <- edx_train_complete %>% select(movieId, movie_b) %>% unique()
  genre_fct <- edx_train_complete %>% select(userId, genres, genre_b) %>% unique()
  relyr_fct <- edx_train_complete %>% select(userId, release_year, relyr_b) %>% unique()

  # Create a vector to contain the RMSE results
  RMSE_testSets <- vector(mode = 'numeric', length = ncol(index))


  # Use cross validation to loop through the different test sets created in the
  # introduction of this project
  for (i in 1:ncol(index)){
```

```
    edx_test <- edx[index[ , i], ]  # Create a temporary test set

    # Add columns to the test set and create predictions
    edx_test <- edx_test %>%
      inner_join(movie_fct, by = 'movieId') %>%
      inner_join(genre_fct, by = c('userId', 'genres')) %>%
      inner_join(relyr_fct, by = c('userId', 'release_year')) %>%
      mutate(prediction = mu + movie_b + genre_b + relyr_b)

    # Output RMSE values from each set
    RMSE_testSets[i] <- RMSE(edx_test$rating, edx_test$prediction)

    # Keep track of progress as the loop runs.
    print(paste0('Iteration ', i, ' is complete.'))
  }
  # Determine final RMSE for optimized model.
  finalRMSE <- mean(RMSE_testSets)

}


## MODEL HAS BEEN OVERTRAINED, MUST CORRECT
# While the overall RMSE looks good, tt becomes abundantly clear in looking at the
# results that the model has been overtrained. The RMSE is quite high for the first
# test set where the values not included in training the model while the remaining
# 9, which probably are comprised signifantly of training material, are all
# considerably lower. All of this points to overtraining and the model likely became
# too granular in an attempt to adjust on the user level with the information
# available.
```

The exact results are unavailable but the average value of the RMSEs from this cross validation calculation came out to be approximately 0.75. But this number is quite misleading. While the overall RMSE looks good, it becomes abundantly clear in looking at the results that the model has been overtrained. The RMSE is quite high (~0.95) for the first test set where the content was not included in training the model while the remaining nine, which are comprised significantly of training material, are all considerably lower. All of this points to overtraining and the model likely became too granular in an attempt to adjust on the user level with the information available.

Simplifying the Model

The overtraining discovered in the previous step is likely due to the corrections being applied on the user level. The focus of the model became too specific to the samples in the training set and any variation in new instances were predicted inaccurately as a result. The model obviously requires simplification to correct for the overtraining created from the previous method. To do so, accounting for variation at the user level has been eliminated from the model. Genre and release year will still be taken into effect, but only on the data set level and both variables will be regularized. It will also include a regularization parameter for each user to account for their positivity or negativity as a rater overall.

The following functions replicate the process performed in the previous tuning function but do not concern themselves with any corrections at the user level. Once again, this function has been set up to allow for cross validation and the second index partition was used to create a new default training set.

```r
# Simplified Model ----------------------------------------------------------

# The model obviously requires simplification to correct for the overtraining created
# from the previous method. To do so, accounting for variation at the user level has
# been eliminated from the model. Genre and release year will still be taken into
# effect, but only on the data set level. It will also include a regularization
# parameter for each user to account for their positivity or negativity as a rater
# overall.

# Create a new default training set to optimize the model on.
edx_train_simp <- edx[-index[ , 2], ]

# The following function replicates the process performed in the previous tuning
# function but does not concern itself with any corrections within the user level.
# Once again, this function has been set up to allow for cross validation.

# INPUTS
# - lambda_simp: lambda regularization value for testing
# - cross_val_train: training set from cross validation slice

# OUTPUTS
# - RMSE for the given lambda and training set
tuning_function_simp <- function(lambda_simp, cross_val_train){

  # Create movie specific regularization parameter
  reg_movie_simp <- cross_val_train %>%
    group_by(movieId) %>%
    summarize(movie_b = sum(rating - mu)/(lambda_simp + n()))
  # Replace with lambda_simp[1] if testing unique lambdas

  # Add regularized movie parameter to training set
  cross_val_temp <- cross_val_train %>%
    left_join(reg_movie_simp, by = 'movieId')

  # Create genre specific regularization parameter
  reg_genre_sum_simp <- cross_val_temp %>%
    group_by(genres) %>%
    summarize(genre_b = sum(rating - mu - movie_b)/(lambda_simp + n()))
  # Replace with lambda_simp[2] if testing unique lambdas

  # Create user specific regularization parameter
  reg_user_sum_simp <- cross_val_temp %>%
    group_by(userId) %>%
    summarize(user_b = sum(rating - mu - movie_b)/(lambda_simp + n()))
  # Replace with lambda_simp[3] if testing unique lambdas

  # Create release year specific regularization parameter
  reg_relyr_sum_simp <- cross_val_temp %>%
    group_by(release_year) %>%
    summarize(relyr_b = sum(rating - mu - movie_b)/(lambda_simp + n()))
  # Replace with lambda_simp[4] if testing unique lambdas

  # Add new correction terms to training set and create prediction
```

```
  cross_val_temp <- cross_val_temp %>%
    left_join(reg_genre_sum_simp, by = 'genres') %>%
    left_join(reg_user_sum_simp, by = 'userId') %>%
    left_join(reg_relyr_sum_simp, by = 'release_year') %>%
    mutate(prediction = mu + movie_b + genre_b + relyr_b + user_b)

  # Keep track of progress during operation
  # print(paste0('Iteration complete with lambda iteration ',
  #               lambda_simp[1], ', ', lambda_simp[2], ', ',
  #               lambda_simp[3], ', ', lambda_simp[4], '.'))
  # print(paste0('Iteration complete with lambda iteration ', lambda_simp, '.'))
  # Return RMSE for the current iteration
  RMSE(cross_val_temp$rating, cross_val_temp$prediction)

}
```

Updated Cross Validation for Simplified Model

The updated functions have been set up to run with the same lambda (single lambda form) used for all tuning parameters or with unique lambdas for each (multi lambda form), the latter being included due to the expectation that an optimized model with have independent lambdas at each regularization step. The way the report presents these functions is in the form that uses the same lambda for all regularization, which is the method ultimately used for the final model. If the reviewer would like to run the code with the multiple lambda method, the two functions require slight adjustment. The code required to make this change is included with both functions and commented out. Scan through the functions to find the annotated sections and change which line is commented (lines 801, 815, 828, 835, 846, 853, 860, 878). If altered to the multi lambda form, the input when the following function is called will be a four column data frame with the different permutations of lambdas while the single lambda form requires only a vector of lambda options. Again, no changes are required for running the code in its provided form.

```
# Simplified Looping Function ---------------------------------------------

# The following function replicates the previous looping function from the updated the
# testing process to be the modified and simplified version that accounts for movie,
# user, genre, and release year on the data set level.

# INPUTS
# - tuning_lambdas_simp: vector or dataframe of lambda(s) for testing
#                        function has been modified to allow for unique lambdas for
#                        each variable
# - k: number of cross validation slices to test on

# OUTPUTS (housed within data frame)
# - testing_RMSEs: vector with RMSEs from optimized settings for each cross validation
#                  sample
# - testing_lambdas: final lambda parameter used to create final RMSE for each cross
#                    validation sample

tuning_loop_simp <- function(tuning_lambdas_simp, k){

  # Create cross validation slices from training set
  cross_val_idx <- createDataPartition(y = edx_train_simp$rating,
                                       times = k,
```

```
                                    p = 0.90,
                                    list = FALSE)
# Create output data frame
testing_RMSEs <- vector(mode = 'numeric', length = k)
testing_lambdas <- vector(mode = 'numeric', length = k)

## If testing for multiple unique lambdas, replace the previous line with the following one
# testing_lambdas <- data.frame(Movie_Lambda = numeric(0),
#                               User_Lambda = numeric(0),
#                               Genre_Lambda = numeric(0),
#                               ReleaseYear = numeric(0))

for (i in 1:k){
  # print(paste0('Beginning slice  ', i, '.')) # Keep track of progress during operation

  # Define training and test set for each cross validation slice
  cross_val_train <- edx_train_simp[cross_val_idx[ , i], ]
  cross_val_test <- edx_train_simp[-cross_val_idx[ , i], ]

  # Run tuning function
  ## Use this line if testing different lambdas for each correction parameter
  # lambda_tune <- apply(tuning_lambdas_simp, 1,
  #                      tuning_function_simp,
  #                      cross_val_train = cross_val_train)

  ## Use this line if testing a single lambda across all correction parameters
  lambda_tune <- sapply(tuning_lambdas_simp,
                        tuning_function_simp,
                        cross_val_train = cross_val_train)

  # Extract the lambda(s) that created the smallest RMSE and apply to test set
  train_lambdas <- tuning_lambdas_simp[which.min(lambda_tune)]
  ## If training multiple unique lambdas, replace the previous line with the
  # following one
  # train_lambdas <- tuning_lambdas_simp[which.min(lambda_tune), ]

  # Repeat process to evaluate model on cross validation test set

  # Movie correction parameter
  reg_movie_simp <- cross_val_test %>%
    group_by(movieId) %>%
    summarize(movie_b = sum(rating - mu)/(train_lambdas + n()))
  # lambda from tuning, replace with train_lambdas[1] for
  # testing multiple unique lambdas

  # Add as a column to the test set
  cross_val_test <- cross_val_test %>%
    left_join(reg_movie_simp, by = 'movieId')

  # Genre correction parameter
  reg_genre_simp <- cross_val_test %>%
    group_by(genres) %>%
    summarize(genre_b = sum(rating - mu - movie_b)/(train_lambdas + n()))
```

```
    # lambda from tuning, replace with train_lambdas[2]
    # for testing multiple unique lambdas

    # User correction parameter
    reg_user_simp <- cross_val_test %>%
      group_by(userId) %>%
      summarize(user_b = sum(rating - mu - movie_b)/(train_lambdas + n()))
    # lambda from tuning, replace with train_lambdas[3]
    # for testing multiple unique lambdas

    # Release year correction parameter
    reg_relyr_simp <- cross_val_test %>%
      group_by(release_year) %>%
      summarize(relyr_b = sum(rating - mu - movie_b)/(train_lambdas + n()))
    # lambda from tuning, replace with train_lambdas[4]
    # for testing multiple unique lambdas

    # Add correction parameters to test set and create predictions
    cross_val_final <- cross_val_test %>%
      left_join(reg_genre_simp, by = 'genres') %>%
      left_join(reg_user_simp, by = 'userId') %>%
      left_join(reg_relyr_simp, by = 'release_year') %>%
      mutate(prediction = mu + movie_b + genre_b + relyr_b + user_b)

    # Keep track of progress throughout operation
    # print(paste0('Slice ', i, ' is complete.'))

    # Add results to the outputted data frame
    testing_RMSEs[i] <- RMSE(cross_val_final$rating, cross_val_final$prediction)
    testing_lambdas[i] <- train_lambdas
    ## If tuning multiple unique lambdas, replace the previous line with the following one
    # testing_lambdas[i, 1:4] <- train_lambdas

  }
  # Return final data frame with optimized results from each cross validation test set
  return(cbind(testing_RMSEs, testing_lambdas))
}
```

Cross Validation on Slices of Training Data with Simplified Model

After the optimal lambda settings for the simplified model have been determined, cross validation will be used on ten different test set partitions to determine the RMSE within the edx database. A function that performs this operation is defined in the next section. The function was created to reduce the amount of code required since this operation was performed twice - once for the single lambda form and once for the multiple lambda form.

```
# Cross Validation on Various Test Sets -----------------------------------

# Prior to running the optimization functions, a cross validation testing function is
# created that can take the optimal lambda parameters and apply them across each of
# the test sets for a final calculation on the accuracy of the model.

# INPUTS
```

```
# - movie_lambda: optimized movie specific regularization value from tuning
# - genre_lambda: optimized genre specific regularization value from tuning
# - user_lambda: optimized user specific regularization value from tuning
# - relyr_lambda: optimized release year specific regularization value from tuning

# OUTPUTS
# RMSE_testSets: vector with RMSE from each of the cross validation slices from
# overall edx set

cross_validation_test <- function(movie_lambda, genre_lambda, user_lambda, relyr_lambda){

  # Create output vector
  RMSE_testSets <- vector(mode = 'numeric', length = ncol(index))
  for (i in 1:ncol(index)){

    # print(paste0('Starting loop #', i)) # Keep track of progress during operation

    # Create the correction parameters based on the training portion of
    # cross validation slice

    # Movie specific correction
    reg_movie_simp <- edx[-index[ , i], ]  %>%
      group_by(movieId) %>%                    #
      summarize(movie_b = sum(rating - mu)/(movie_lambda + n()))
    # lambda optimized in tuning function


    edx_final <- edx[index[ , i], ] %>%
      inner_join(reg_movie_simp, by = 'movieId')

    # Genre specific correction
    reg_genre_simp <- edx_final %>%
      group_by(genres) %>%
      summarize(genre_b = sum(rating - mu - movie_b)/(genre_lambda + n()))
    # lambda optimized in tuning function

    # User specific correction
    reg_user_simp <- edx_final %>%
      group_by(userId) %>%
      summarize(user_b = sum(rating - mu - movie_b)/(user_lambda + n()))
    # lambda optimized in tuning function

    # Release year specific correction
    reg_relyr_simp <- edx_final %>%
      group_by(release_year) %>%
      summarize(relyr_b = sum(rating - mu - movie_b)/(relyr_lambda + n()))
    # lambda optimized in tuning function

    # Apply these parameters to the test set for each cross validation slice
    edx_test <- edx[index[ , i], ]

    # Keep track of progress during operation
    # print(paste0('Progress at iteration ', i, '.'))
```

```
    # Combine correction parameters into test set data frame
    edx_test <- edx_test %>%
      inner_join(reg_movie_simp, by = 'movieId') %>%
      inner_join(reg_genre_simp, by = 'genres') %>%
      inner_join(reg_relyr_simp, by = 'release_year') %>%
      inner_join(reg_user_simp, by = 'userId')

    # Make predictions
    edx_test <- edx_test %>%
      mutate(prediction = mu + movie_b + genre_b + relyr_b + user_b)

    # Correct for predictions under/over the possible min/max
    # allowed in the rating system
    edx_test$prediction[edx_test$prediction > 5] <- 5
    edx_test$prediction[edx_test$prediction < 0.5] <- 0.5

    # Add to results
    RMSE_testSets[i] <- RMSE(edx_test$rating, edx_test$prediction)

    # Keep track of progress throughout operation
    # print(paste0('Slice ', i, ' is complete.'))
  }
  # Return complete results
  return(RMSE_testSets)
}
```

Evaluate Simplified Model with Single Lambda

The following section of code performs the operation of lambda tuning and cross validation testing for the single lambda form in the simplified model. If the reviewer would like to run the code with tuning of this model active, the *simplified_tuning* from line 176 should be switched to TRUE. If the reviewer is running the code based simply on the final model, the lambdas parameters are created automatically based on prior testing if the *noTuning_oneLambda* variable is set to TRUE in line 182 (this is how the code is provided).

```
# Tuning Simplified Model with Single Lambda -------------------------------

# Run function with a single lambda
if (simplified_tuning){

  # Create vector of testing lambdas
  simplified_lambdas <- seq(0.1, 15, 0.1)
  # Run function with cross validation
  simplified_tune <- tuning_loop_simp(tuning_lambdas_simp = simplified_lambdas, k = k)
  # Maintain 10 cross validation slices

  # Extract best lambda parameter
  finalSimp_lambda <- simplified_tune[which.min(simplified_tune[ , 1]), 2]

  movie_lambda <- finalSimp_lambda # tuned movie lambda
  genre_lambda <- finalSimp_lambda # tuned genre lambda
  user_lambda <- finalSimp_lambda  # tuned user lambba
  relyr_lambda <- finalSimp_lambda # tuned release year lambda
```

```
  # Cross validation on test set to determine RMSE
  simplified_test <- cross_validation_test(movie_lambda,
                                           genre_lambda,
                                           user_lambda,
                                           relyr_lambda)

  # Find average RMSE
  simplified_RMSE <- mean(simplified_test)

  # Print RMSEs from cross validation test sets
  simplified_tune

} else if (noTuning_oneLambda){

  # 0.3 was determined to be the optimal lambda based on testing done prior to submission
  movie_lambda <- 0.3
  genre_lambda <- 0.3
  user_lambda <- 0.3
  relyr_lambda <- 0.3

  # Cross validation on test set to determine RMSE
  tuning_test <- cross_validation_test(movie_lambda,
                                       genre_lambda,
                                       user_lambda,
                                       relyr_lambda)
  tuning_RMSE <- mean(tuning_test)
}

# Print results
print(tuning_test)
```

```
##  [1] 0.8250863 0.8242892 0.8253422 0.8249957 0.8258348 0.8260395 0.8253686
##  [8] 0.8262492 0.8256467 0.8254174
```

```
print(tuning_RMSE)
```

```
## [1] 0.8254269
```

A lambda range from 0.1 to 15, in increments of 0.1, was tested and **0.3** was found to be the best value for the single lambda form of this model.

Evaluate Simplified Model with Multiple Unique Lambdas

The following section of code performs the operation of lambda tuning and cross validation testing for the multi lambda form in the simplified model. Though the results from this section do not contribute to the final model, this testing was performed alongside the single lambda version to explore whether the optimal version of this model would allow for a unique number at each correction paramater. If the reviewer would like to run the code with tuning of this form active, the *modified_tuning* from line 177 should be switched to TRUE. If the reviewer wants to run the code using the parameters optimized from this form (which is no the final form of the model), change the *noTuning_multipleLambdas* variable to TRUE in line 183.

```r
# Tuning Simplified Model with Multiple Lambdas ---------------------------


# Run function with two lambdas
if (modified_tuning){
  # It still stands to reason that each of the four variables should have a unique
  # regularization lambda in an optimized model.

  multi_lambdas <- seq(0.5, 6.5, 1.5)

  # Create lambdas permutations to test
  multi_lambdas_perm <- permutations(n = length(multi_lambdas),
                                     r = 4,
                                     v = multi_lambdas,
                                     repeats.allowed = TRUE)

  # Run updated model with unique lambdas for each regularization

  modified_tune <- tuning_loop_simp(tuning_lambdas_simp = multi_lambdas_perm, k = k)

  # tuned movie lambda
  movie_lambda <- modified_tune$Movie_Lambda[which.min(modified_tune$testing_RMSEs)]
  # tuned genre lambda
  genre_lambda <- modified_tune$Genre_Lambda[which.min(modified_tune$testing_RMSEs)]
  # tuned user lambda
  user_lambda <- modified_tune$User_Lambda [which.min(modified_tune$testing_RMSEs)]
  # tuned release year lambda
  relyr_lambda <- modified_tune$ReleaseYear[which.min(modified_tune$testing_RMSEs)]

  tuning_test <- cross_validation_test(movie_lambda,
                                       genre_lambda,
                                       user_lambda,
                                       relyr_lambda)
  tuning_RMSE <- mean(modified_test)

} else if (noTuning_multipleLambdas){

  # Optimal lambda settings for this particular unique multi lambda model
  movie_lambda <- 0.5
  genre_lambda <- 0.5
  user_lambda <- 6.5
  relyr_lambda <- 6.5

  # Cross validation on test set to get results
  tuning_test <- cross_validation_test(movie_lambda,
                                       genre_lambda,
                                       user_lambda,
                                       relyr_lambda)
  tuning_RMSE <- mean(tuning_test)
  # Results were consistently higher than the single lambda model
}
```

```
#########################################################################
```

RESULTS

Based on extensive testing and tuning, it was determined that the best option was a simplified model that applied correction parameters on variables on the data set level and avoids involvement with user specific preferences aside from their overall inclination to rate more or less positively. The final model applies four regularized correction parameters to the average of all movie ratings to provide recommendations. The correction parameters consider past averaged ratings by movie, user, genre, and release year and were calculated using a single lambda value of 0.3. The assumption that a unique lambda value for each regularization would be optimal proved to be incorrect.

Data tables that create and collect those correction parameters are defined in the next section. Once these data tables are created, the appropriate values from each can be added to a prospective rating with matching criteria to determine an appropriate recommendation.

```
### RESULTS

# This section will create the optimized and regularized correction parameters based on
# the entire data set so those values can be applied to make predictions on the edx data
# set, validation data set, and future recommendations that arise.

# The inner join function has been used in combining the correction parameters with each
# data set in order to ensure that no NaNs enter the data set and nullify the RMSE
# calculations.


# Final Model ----------------------------------------------------------------

# The four variables created in this section will correspond to predictions made based
# on overall rating's average in addition to movie, genre, user, and release year
# adjusted terms. A prediction can be made on any rating given that those four criteria
# are available.

# Final movie correction term
movie_correction <- edx  %>%
  group_by(movieId) %>%
  summarize(movie_b = sum(rating - mu)/(movie_lambda + n()))
# lambda optimized in tuning function

# Add movie specific correction test set
edx <- edx %>%
  inner_join(movie_correction, by = 'movieId')

# Final genre specific correction
genre_correction <- edx %>%
  group_by(genres) %>%
  summarize(genre_b = sum(rating - mu - movie_b)/(genre_lambda + n()))
# lambda optimized in tuning function

# Final user specific correction
user_correction <- edx %>%
```

```
  group_by(userId) %>%
  summarize(user_b = sum(rating - mu - movie_b)/(user_lambda + n()))
# lambda optimized in tuning function

# Final release year specific correction
releaseYear_correction <- edx %>%
  group_by(release_year) %>%
  summarize(relyr_b = sum(rating - mu - movie_b)/(relyr_lambda + n()))
# lambda optimized in tuning function
```

RMSE on edx Data

The correction parameters can now be applied to the entire edx data set and predictions calculated with the resulting data frame. Once predictions for each rating have been made, we can calculate the RMSE to get an idea of the accuracy of this model.

```
# edx Data Set ------------------------------------------------------------

# Add correction parameters to entire edx data set
edx_final <- edx  %>%
  inner_join(genre_correction, by = 'genres') %>%
  inner_join(releaseYear_correction, by = 'release_year') %>%
  inner_join(user_correction, by = 'userId')

# Create predictions
edx_final <- edx_final %>% mutate(prediction = mu + movie_b + genre_b + relyr_b + user_b)

# Confine low/high predictions within the limits of the recommendation system
edx_final$prediction[edx_final$prediction > 5] = 5
edx_final$prediction[edx_final$prediction < 0.50] = 0.5

# Calculate RMSE of model on edx data set
edx_RMSE <- RMSE(edx_final$rating, edx_final$prediction)
print(paste0('RMSE on the edx data set is ', edx_RMSE, '.'))
```

```
## [1] "RMSE on the edx data set is 0.856534447446432."
```

The RMSE of the final model on the entire edx data set is 0.85653

RMSE on Validation Data

With an acceptable result on the edx data set, the final model must be tested on the validation data set, which was not used at all in training the model, to properly understand it's accuracy. The same correction parameters and prediction calculation has been applied to the validation set here and the RMSE calculated.

```
# Validation Data Set -----------------------------------------------------

# Create the necessary release year column in the validation data set
validation <- validation %>%
  mutate(release_year = str_extract(title, "(?<=\\()\\d{4}?(?=\\))"))
```

```r
# Add correction parameters to entire validation data set
validation <- validation %>%
  inner_join(movie_correction, by = 'movieId') %>%
  inner_join(genre_correction, by = 'genres') %>%
  inner_join(releaseYear_correction, by = 'release_year') %>%
  inner_join(user_correction, by = 'userId')

# Create predictions
validation_final <- validation %>%
  mutate(prediction = mu + movie_b + genre_b + relyr_b + user_b)

# Confine low/high predictions within the limits of the recommendation system
validation_final$prediction[validation_final$prediction > 5] = 5
validation_final$prediction[validation_final$prediction < 0.50] = 0.5

# Calculate RMSE of model on validation data set
validation_RMSE <- RMSE(validation_final$rating, validation_final$prediction)
print(paste0('RMSE on the validation data set is ', validation_RMSE, '.'))
```

```
## [1] "RMSE on the validation data set is 0.865091725808376."
```

```r
#################################################################

### DISCUSSION
```

The RMSE on the validation data set is 0.86509.

This RMSE is higher than the results from the edx data set by approximately 0.009. There is the possibility that this is due to slight overtraining but it more likely within the margin of error for this model.

DISCUSSION

Overview

While a final RMSE of **0.86509** is far from perfect, it does adequately represent that an effective movie recommendation system was designed and outlined in this report. Several parameters are used to develop these predictions with the average rating of all movies providing the foundation for building each recommendation with adjustments made for movie, user, genre, and release year averages. Regularization was applied for each correction parameter in order to not give too much weight to categories that do not have a significant number of ratings.

The methods and processes outlined in this report showcase and build on concepts covered throughout the coursework in the edx data science catalogue including data wrangling, visualization, machine learning, etc. As a student, this capstone project was a great learning experience in applying these concepts individually and the thought processes of this work has solidified an understanding in what has been taught over the courses within the certificate program. The final model creates predictions on a high level and there is room for improvement available, discussed in the next section.

Limitations & Future Improvement

This recommendation model has the potential to be further refined based on the information provided. The original intention for this model was to cater recommendations to user preferences at a deeper level than

simply their overall positivity, i.e. quantifying whether a user has inclinations towards certain genres and release eras and factoring that in. The attempts to create a model that does this were unsuccessful due clear overtraining that actual made the model less accurate. For the sake of this project, that aspect was excluded but it is intuitive to suspect that an idealized version that does include these user preference is not only possible, but would offer a great improvement.

There are also additional variables within the provided dataset that could also be exploited in the model but didn't warrant inclusion at first blush. The timestamp variable in the original dataset was not considered and the relationship between the rating and when it was given could offer an improvement in a couple ways. The overall average ratings given may change over time, with user positivity and willingness to give better ratings changing over time. There is also the possibility that the time of year is a factor for a movie's rating. For example, users may be more inclined to give better ratings during the summer months when moods are generally higher because there is more sunlight than in winter. The time of year could also be tied back to specific genres and movies as well, such as Christmas movies potentially receiving higher ratings during December and romantic movies having a similar response in the week's surrounding Valentine's Day.

We could also consider the relationship between when the rating was given and when the movie was came out as well, since users may view newly released movies differently than those that have been publicly available for years. This factor was not included for this recommendation system due to the challenges provided by the time period for which ratings were given is only a segment of the time frame from which the movies were released so it would only be possible to apply this part of our data set.

The model can be improved even further if additional information is provided within the data set, such as movie studio, star actor, or language. These types of aspects of a movie can offer another level of user preference that may not be captured within the current information provided. For example, a user may like movies with George Clooney disproportionately to the genre's of those movies or enjoys Disney movies more so than kid's movies made by other studios. Increasing the complexity of this model could minimize our error even further but the recommendation system as it currently stands is fairly capable of accurately predicting movie ratings based on a previous feedback on movie quality, user positivity, genre reputation, and release year audience preferences.