# MovieLens Project

Mark Khusidman

2023-05-25

## Introduction

A recommendation system is a model which predicts user content ratings based on the content's relevant features and prior user choices. The goal of this project is to create and evaluate a movie recommendation system based on the "10M" version of the "MovieLens" dataset. This dataset contains six columns and 10,000,054 rows in total. Each row represents a single rating of a specific film made by a particular user. The six columns which constitute this dataset are described as follows: *userId* is an integer column which contains a unique identifier for the user making a rating; *movieId*, also an integer column, contains a unique identifier for the movie being rated; *rating* is a numeric column which contains the user's rating for the movie in question; *timestamp* is an integer column containing the number of seconds between the Unix epoch (00:00:00 January 1st, 1970) and the time when the rating was made; *title* is a character column which contains both the *title* and the release year of the movie being rated; *genres*, also a character column, contains a pipe-separated list of all genres associated with the movie being rated. Prior to model training, the data is split into a training set (referred to as the *edx* set) containing 9,000,055 rows and a holdout set containing 999,999 rows. In order to create a movie recommendation system, movie ratings must be modeled based on relevant columns. In the case of this project, some columns used in the model preexist in the original *edx* set, while others are derived from preexisting columns. The model evaluated for this project is conceived of as the sum of average rating biases found along these columns. In other words, grouping is used to calculate an average bias for each unique value of each relevant column in the *edx* set. The predicted rating corresponding to any combination of these values can then be calculated by taking the sum of their associated average biases. Certain hyperparameters, which will be described later, are used during model training. The specific values that these hyperparameters take in the final model are determined via Monte Carlo cross-validation. Before this training can take place, however, the data must be explored and preprocessed.

## Methods

Note that the data exploration, data cleaning, and new column creation described in the following section must be applied to both the *edx* and holdout sets. Both datasets must be explored to confirm that the testing data is actually representative of the training data, while a failure to perform either data cleaning or new column creation on the holdout set will likely lead to failure during model testing. Unless otherwise specified, any data exploration done to the *edx* set yielded similar results when applied to the holdout set. Likewise, all column creation steps performed on the *edx* step were, by necessity, also performed on the holdout set. The holdout data remained completely untouched while implementing the model-tuning loop to prevent data leakage. After all data have been checked for empty values, examination of individual columns can begin.

```
# Detect any missing values in data
sprintf("Any missing values in edx?: %s", anyNA(edx))
```

```
## [1] "Any missing values in edx?: FALSE"
```

Because character columns are generally the most prone to error, these are the first columns checked for issues. Specifically, it is important to ensure that all string formats present in the *genres* and *title* columns are known and that none of the values contain formatting errors. This way, future attempts to parse the character data are much less likely to fail. Character matching via regular expressions is used on both the *genres* and *title* columns to check whether the formats seen in the first rows are consistent throughout the data.

```
# Detect any items in the genres column which do not fit the apparent format
edx$genres[str_detect(edx$genres,
                      "^[A-Z][a-z]+|^[A-Z][a-z]+\\|[A-Z][a-z]+", negate = TRUE)]
```

```
##  [1] "IMAX"                "IMAX"                "(no genres listed)"
##  [4] "(no genres listed)" "IMAX"                "IMAX"
##  [7] "IMAX"                "IMAX"                "(no genres listed)"
## [10] "IMAX"                "IMAX"                "(no genres listed)"
## [13] "IMAX"                "IMAX"                "IMAX"
## [16] "(no genres listed)" "IMAX"                "IMAX"
## [19] "(no genres listed)" "(no genres listed)" "IMAX"
```

In the case of the *genres* column, there do appear to be values present that do not match the initial format. It does not appear that these values will interfere with future parsing, however, and are left as they are.

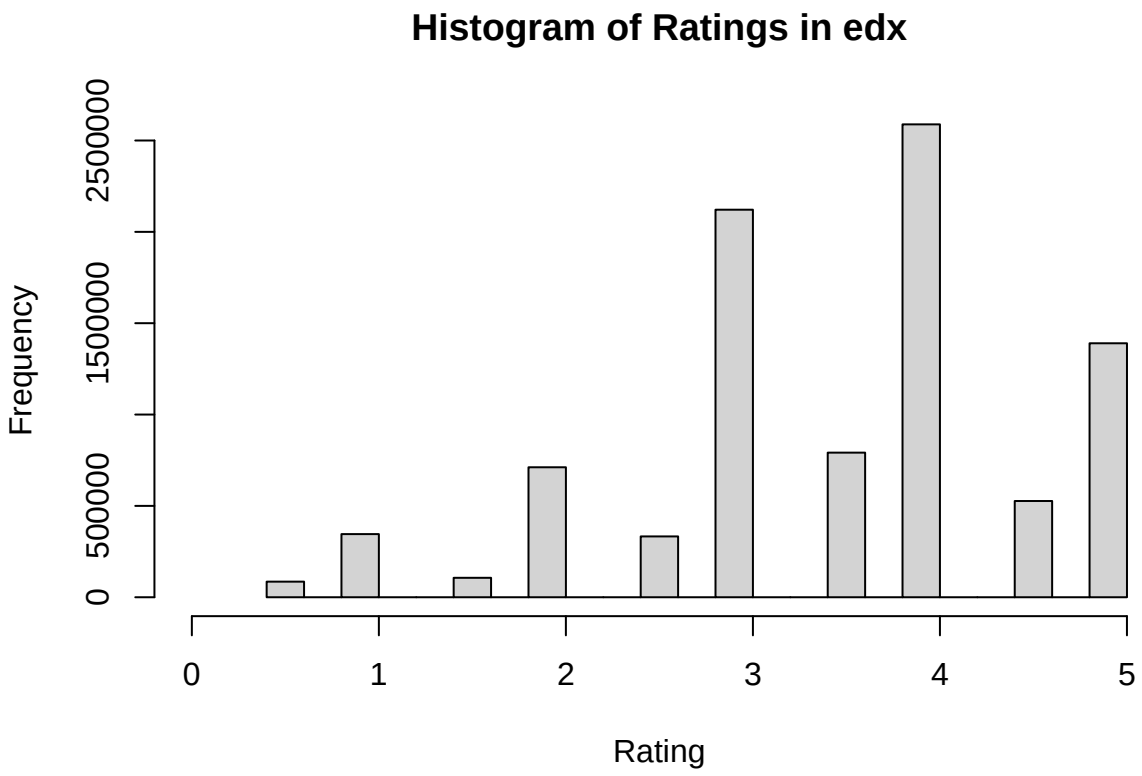
```
# Detect any items in the title column which do not fit the apparent format
edx$title[str_detect(edx$title, "\\s\\(\\d+\\)$", negate = TRUE)]
```

```
## character(0)
```

In the case of the *title* column, all values appear to be consistent with the initial format.

Next, it is useful to ensure that there are no outliers present in the *rating* column, as these values would likely be erroneous. An easy way to check for outliers while also getting an idea of set's distribution is to use a histogram. Shown below is a histogram of the *edx* dataset's *rating* column.

```
# Visualize rating distribution in edx
hist(edx$rating, breaks = 20, xlim = c(0, max(edx$rating)), xlab = "Rating",
     main = "Histogram of Ratings in edx")
```




Upon viewing the above chart, a few facts become apparent: First, values in the column are bound between 0.5 and 5. Second, all values appear to be multiples of 0.5. Third, it does not appear as though any outliers are present.

It is important to note that all further steps are taken after the "data.frame" objects representing the *edx* and holdout datasets are transformed into "data.table" objects. This is done because the data.table library was found to be much faster than the "tidyverse" library when large quantities of data are considered.

```
# Convert edx to data.table object
setDT(edx)
```

Because the values in the *userId* and *movieId* columns have no inherent meaning apart from their uniqueness, testing for outliers in these columns is not necessary. It is useful, however, to make sure that each value in *movieId* corresponds to only one movie *title*. The code below demonstrates that there is in fact a movie with two corresponding *movieId* values.

```
# See if any titles have more than one associated movie ID
edx[, .(id_count = length(unique(movieId))), by = title][id_count > 1,]
```

```
##                         title id_count
## 1: War of the Worlds (2005)        2
```

To correct this issue, the more prevalent ID associated with "War of the Worlds (2005)" is found and used to replace the less common value. Note that this replacement will be repeated on the holdout set at a later point. Also note that, because the holdout set does not contain any movies that are not present in the training set, checking for more duplications in the holdout set is unnecessary.

```
# Find the most common movie ID associated with this title and replace instances
# of the less common ID

primary_id <- edx[title == "War of the Worlds (2005)",
                  length(title), keyby=movieId][1, movieId]

edx$movieId[which(edx$title == "War of the Worlds (2005)")] <- primary_id
```

Before designing and testing a complex model, it is often desirable to first derive a baseline RMSE by testing a very simple model on the data. In this case, always predicting the mean of the *edx* dataset's *rating* column (*global_mean*) represents a suitable first model.

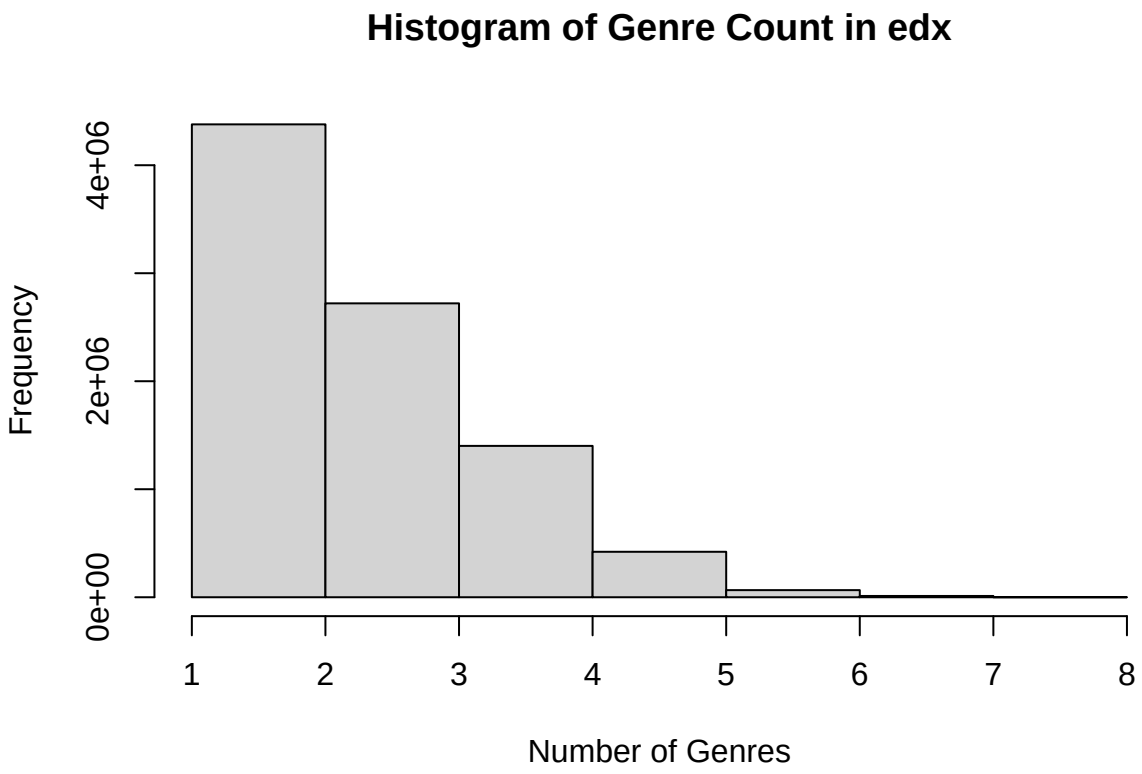
```
# Evaluate simple model for baseline RMSE
global_mean <- mean(edx$rating)
baseline_rmse <- sqrt(mean((global_mean - edx$rating)^2))
sprintf("Baseline RMSE: %f", baseline_rmse)
```

```
## [1] "Baseline RMSE: 1.060331"
```

As shown above, the baseline RMSE yielded by this simple model is slightly over 1.06. The simple model is tested on the *edx* set rather than the holdout set to prevent any kind of tuning based on the holdout set.

Now that preliminary data exploration and cleaning have been conducted and a baseline RMSE has been defined, the creation of new columns can begin. First the *genres* column is split into multiple individual genre columns. Before this can be done, however, the number of genre columns to use must be decided. Although values in the *genres* column can contain as many as 8 genres, the vast majority of values do not include so many items. This is demonstrated in the graph below.

```
# Visualize proportion of rows containing 6, 7, or 8 listed genres
genre_count_edx <- str_count(edx$genres, "\\|") + 1
hist(genre_count_edx, breaks = 8, xlab = "Number of Genres",
     main = "Histogram of Genre Count in edx")
```




In fact, genres 6, 7, and 8 are present in only 0.868% of rows in the *edx* set. As such, only columns corresponding to the first 5 listed genres are created. The total number of non-null genres in each row is also added as a column named *n_genres*, and it should be noted that this value only takes into account the aforementioned 5 individual genre columns. This was found to yield a slightly better result than taking into account all 8 genres potentially found in a *genres* column value. Note that the *genres* column itself was not found to improve RMSE when used in conjunction with its derived columns, and is therefore not used as an input during model training.

```
# Separate genres and calculate number of genres given. Convert genres to factors
separate_genres <- function(dt){

  dt[, c("genre_1", "genre_2", "genre_3", "genre_4", "genre_5") :=
        tstrsplit(dt$genres, "|", fixed=TRUE, fill = "None", keep = 1:5)]

  dt[, n_genres := factor(rowSums(.SD != "None")), .SDcols =
        c("genre_1", "genre_2", "genre_3", "genre_4", "genre_5")]

  dt[, c("genre_1", "genre_2", "genre_3", "genre_4", "genre_5") :=
        list(factor(genre_1), factor(genre_2), factor(genre_3), factor(genre_4),
             factor(genre_5))]
}

separate_genres(edx)
```

Next, the *timestamp* column was used to extract the year, month, day, and hour that each review was made. These columns are named *rev_year*, *rev_month*, *rev_day*, and *rev_hour*, respectively. The data.table

implementations of various "lubridate" functions are used for this purpose. Note that the *timestamp* column itself is not used as a model input because it contains too many unique values; most average biases associated with its values would be based on a single observation.

```
# Extract timestamp-related data
extract_ts <- function(dt){
  iso <- as_datetime(dt$timestamp)

  dt[, c("rev_year", "rev_month", "rev_day", "rev_hour") :=
        list(factor(year(iso)), factor(month(iso)),
             factor(day(iso)), factor(hour(iso)))]
}

extract_ts(edx)
```

A regular expression group is now used to extract movie release years from the *title* column. The resulting column is named *movie_year*. Note that the "mdy" function in the code below comes directly from the lubridate package. Note that the *title* column itself is not used as an input to the model because it is redundant with the *movieId* column.

```
# Extract movie year from title
m_year_edx <- str_match(edx$title, "\\s\\((\\d+)\\)$")[,2]
edx[, movie_year := factor(year(mdy(paste("1-1-", m_year_edx))))]
```

Next, an *rbias* column is created in the *edx* set by subtracting the previously referenced *global_mean* from all values in the *rating* column. This column represents the movie ratings as biases centered around *global_mean*. These bias values will be used during model training to calculate the average biases associated with input values. It is important to note that because the *rbias* column merely facilitates average bias calculation, rather than serving as a model input itself, it is not added to the holdout set.

```
# Create the initial bias column which will be used to calculate average biases
edx[, rbias := rating - global_mean]
```

After the aforementioned columns are created, the model-tuning loop is run. Below is the full code for the model-tuning loop. It is not evaluated due to computational constraints, but will be explained in detail further below.

```
# Hyperparameters
era_len_pars <- seq(5, 25, 5) # Controls era length
lambda_pars <- 2:8 # Controls extent of regularization

# Array to hold hyperparameter values and results
params <- expand.grid(era_len_pars, lambda_pars)
names(params) <- c("era_len_pars", "lambda_pars")
params$rmse <- NaN
params$sd <- NaN

# Model-tuning loop

for(row in 1:nrow(params)){

  # Set era_len hyperparameter and extract movie era from movie year
  era_len <- params$era_len_pars[row]
```

```
edx[, movie_era := factor(floor(as.integer(m_year_edx) / era_len) * era_len)]

# Set lambda hyperparameter
lambda = params$lambda_pars[row]

for(i in 1:3){

  # Split edx dataset into train and validate sets

  validate_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1,
                                        list = FALSE)
  train <- edx[-validate_index,]
  temp <- edx[validate_index,]

  validate <- temp |>
    semi_join(train, by = movieId) |> semi_join(train, by = userId)

  removed <- anti_join(temp, validate, by = c(userId, movieId))
  train <- rbind(train, removed)

  # Model-training loop

  for(col in c("genre_1", "genre_2", "genre_3", "genre_4", "genre_5", "n_genres",
               "movie_era", "movie_year", "rev_hour", "rev_day", "rev_month",
               "rev_year", "movieId", "userId")){

    # Calculate regularized average biases based on training set
    train[, paste0(col, "_bias") := sum(rbias) / (length(rbias) + lambda),
        by = col]

    # Remove effect of average biases from training set rbias column
    train[, rbias := rbias - .SD, .SDcols = paste0(col, "_bias")]

    # Join average biases with appropriate values in validation set
    temp <- train[, .SD[1], .SDcols = paste0(col, "_bias"), by= col]
    validate <- temp[validate, on = col]
  }

  # Calculate predictions by summing all bias columns with global_mean
  col_names <- names(train)
  bias_cols <- col_names[str_detect(col_names, "_bias")]
  train[, pred := rowSums(.SD) + global_mean, .SDcols = bias_cols]
  validate[, pred := rowSums(.SD) + global_mean, .SDcols = bias_cols]

  # Clip predictions such that they are between 0.5 and 5
  train[, pred := fifelse(pred > 5, 5, pred)]
  train[, pred := fifelse(pred < 0.5, 0.5, pred)]
  validate[, pred := fifelse(pred > 5, 5, pred)]
  validate[, pred := fifelse(pred < 0.5, 0.5, pred)]

  # Calculate and report training and validation RMSEs
  train_rmse <- sqrt(mean((train$pred - train$rating)^2))
  rmses[i] <- sqrt(mean((validate$pred - validate$rating)^2))
```

```
    sprintf("Training RMSE: %f  Validation RMSE: %f", train_rmse, rmses[i])
  }

  # Record mean and sd of RMSEs for each hyperparameter combination
  params$rmse[row] <- mean(rmses)
  params$sd[row] <- sd(rmses)
}
```

Every iteration of the loop requires setting the values of two hyperparameters: *era_len* and *lambda*. The first of these determines the manner in which the *movie_era* column is calculated. This column is created by grouping values from the movie year column into eras of a length equal to *era_len*, and is calculated for the *edx* set during each full iteration of the model-tuning loop. The second hyperparameter is used to regularize the model by reducing the effect of average biases that were calculated with few observations. A larger value assigned to *lambda* will cause a larger penalty to be applied to low-observation average biases. The values tested for *era_len* are 5, 10, 15, 20, and 25. The values tested for *lambda* are 1 through 8.

```
# Hyperparameters
era_len_pars <- seq(5, 25, 5)  # era_len controls era length
lambda_pars <- 2:8  # lambda controls extent of regularization
```

The array of hyperparameter combinations used for model tuning is created by calling "expand.grid" on a pair of vectors containing all the hyperparameter values to be tested.

```
# Array to hold hyperparameter combinations and results
params <- expand.grid(era_len_pars, lambda_pars)
names(params) <- c("era_len_pars", "lambda_pars")
params$rmse <- NaN
params$sd <- NaN
```

Once the array of combinations is created, the model-tuning loop begins. First, the value of *era_len* is set and subsequently used to create the *movie_era* column. The value of *lambda* is also set at this time.

```
# Extract movie era from movie year. Performed once per hyperparameter combination.
era_len <- params$era_len_pars[row]
edx[, movie_era := factor(floor(as.integer(m_year_edx) / era_len) * era_len)]

# Set lambda hyperparameter
lambda = params$lambda_pars[row]
```

Next, the *edx* dataset is divided into a new training set and a validation set. The training set will be used to train a model with the chosen hyperparameters, while the validation set will be used to evaluate the resulting model. This iterative training and evaluation is performed 3 times for each combination of hyperparameters. Each of these sub-iterations utilizes different subsets of the *edx* set for training and validation, thereby implementing a form of cross-validation. It should be noted that this is not K-fold cross-validation because the data is split randomly. Rather, this process is best described as Monte Carlo cross-validation.

```
# Split edx dataset into train and validate sets. Performed 3 times for each
# hyperparameter combination.

validate_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1,
                                      list = FALSE)
train <- edx[-validate_index,]
```

```
temp <- edx[validate_index,]

validate <- temp |>
  semi_join(train, by = movieId) |> semi_join(train, by = userId)

removed <- anti_join(temp, validate, by = c(userId, movieId))
train <- rbind(train, removed)
```

As mentioned previously, the model in question can be viewed as the sum of the average rating biases associated with a set of input values. The calculation of these biases involves a third loop, this time over all of the *edx* columns which will be used as inputs in the final model design. This innermost loop is referred to as the model-training loop. During each iteration, the *edx* data is grouped by the column in question. The groups are then aggregated using a regularized mean where the extent of regularization is controlled by the *lambda* hyperparameter. The calculated average biases are then subtracted from the *rbias* column, thereby removing their influence on any subsequent iterations of the model-training loop. Finally, the average biases are joined to their associated input values in the validation set.

```
# Model-training loop. Run 3 times for each hyperparameter combination.

for(col in c("genre_1", "genre_2", "genre_3", "genre_4", "genre_5", "n_genres",
             "movie_era", "movie_year", "rev_hour", "rev_day", "rev_month",
             "rev_year", "movieId", "userId")){

  # Calculate regularized average biases based on training set
  train[, paste0(col, "_bias") := sum(rbias) / (length(rbias) + lambda),
      by = col]

  # Remove effect of average biases from training set rbias column
  train[, rbias := rbias - .SD, .SDcols = paste0(col, "_bias")]

  # Join average biases with appropriate values in validation set
  temp <- train[, .SD[1], .SDcols = paste0(col, "_bias"), by= col]
  validate <- temp[validate, on = col]
}
```

Once this is done for all input columns, the average biases in each row are all summed along with *global_mean* to yield predictions. Predictions are made for both the validation set and the training set. This allows training RMSEs, which are used to make sure that the validation RMSEs are reasonable, to be calculated. Because it is known that ratings in these data do not take values lower than 0.5 or higher than 5, predictions are clipped such that they are between those values prior to the calculation of RMSEs.

```
# Calculate predictions by summing all bias columns with global_mean
col_names <- names(train)
bias_cols <- col_names[str_detect(col_names, "_bias")]
train[, pred := rowSums(.SD) + global_mean, .SDcols = bias_cols]
validate[, pred := rowSums(.SD) + global_mean, .SDcols = bias_cols]

# Clip predictions such that they are between 0.5 and 5
train[, pred := fifelse(pred > 5, 5, pred)]
train[, pred := fifelse(pred < 0.5, 0.5, pred)]
validate[, pred := fifelse(pred > 5, 5, pred)]
validate[, pred := fifelse(pred < 0.5, 0.5, pred)]
```

```
# Calculate and report training and validation RMSEs
train_rmse <- sqrt(mean((train$pred - train$rating)^2))
rmses[i] <- sqrt(mean((validate$pred - validate$rating)^2))
sprintf("Training RMSE: %f  Validation RMSE: %f", train_rmse, rmses[i])
```

The 3 RMSEs yielded by each iteration of the second loop are averaged and recorded, allowing the best combination of hyperparameters to be found. The standard deviation of the 3 RMSEs is also recorded.

```
# Record mean and sd of RMSEs for each hyperparameter combination
params$rmse[row] <- mean(rmses)
params$sd[row] <- sd(rmses)
```

The relationship between hyperparameter value and resulting average RMSE (aRMSE) can be effectively visualized with a set of box plots for both *era_len* and *lambda*. However, due to the time it would take to run the model-tuning loop with all combinations of hyperparameters, the code necessary to create these box plots is not run in this report. Although it is tempting to determine the best combination by simply picking the pair of hyperparameters associated with the lowest aRMSE, the aforementioned box plots reveal that many of the smallest aRMSEs are outliers. As a result, these box plots are used to visualize the median aRMSE associated with each tested hyperparameter value. For both *era_len* and *lambda*, the value associated with the lowest such median is chosen for use in the final model. In this case, the best value for *era_len* is found to be 15 and the best value for *lambda* is found to be 6. Although it is not evaluated, the code used to create the aforementioned box plots is provided below for reference.

```
# Visualize relationship between hyperparameter values and average RMSE.

params |> ggplot(aes(lambda_pars, rmse, group = lambda_pars)) + geom_boxplot() +
  geom_point()

params |> ggplot(aes(era_len_pars, rmse, group = era_len_pars)) + geom_boxplot() +
  geom_point()
```

Some columns and hyperparameters explored during the design of this model were found to be counterproductive and discarded. The first of these were a pair of columns which contained the number of observations associated with each *userId* and *movieId* value, respectively. These columns used a separate regularization hyperparameter for average bias calculation; Its use was discontinued along with the columns. A column which grouped users by their *userId* value was also explored. Like the *movie_era* column, its calculation involved the use of its own hyperparameter to determine group size. Both this column and the associated hyperparameter were discarded in the final model design.

## Results

The holdout set is now preprocessed in the same manner as the *edx* set. As mentioned in the previous section, the best value for each hyperparameter is chosen by selecting the tested value with the lowest associated median aRMSE. Keep in mind that the best value for *era_len* was found to be 15 and the best value for *lambda* was found to be 6. Once the holdout set is preprocessed in the same manner as the *edx* set, the model-training loop (the innermost cycle of the model-tuning loop) is run one more time on the entire *edx* set using the aforementioned hyperparameter values. The resulting final model is then evaluated on the holdout set to yield a final RMSE.

```
# Preprocess holdout dataset
setDT(final_holdout_test)
```

```
final_holdout_test$movieId[
  which(final_holdout_test$title == "War of the Worlds (2005)")] <- primary_id
separate_genres(final_holdout_test)
extract_ts(final_holdout_test)
m_year_fht <- str_match(final_holdout_test$title, "\\s\\((\\d+)\\)$")[,2]
final_holdout_test[, movie_year := factor(year(mdy(paste("1-1-", m_year_fht))))]

# Extract movie era from title using the best era_len value
era_len <- 15
edx[, movie_era := factor(floor(as.integer(m_year_edx) / era_len) * era_len)]
final_holdout_test[, movie_era :=
                     factor(floor(as.integer(m_year_fht) / era_len) * era_len)]

# Set lambda equal to best value
lambda = 6

# Model-training loop for final model

for(col in c("genre_1", "genre_2", "genre_3", "genre_4", "genre_5", "n_genres",
             "movie_era", "movie_year", "rev_hour", "rev_day", "rev_month",
             "rev_year", "movieId", "userId")){

  # Calculate regularized average biases based on edx set
  edx[, paste0(col, "_bias") := sum(rbias) / (length(rbias) + lambda),
      by = col]

  # Remove effect of average biases from edx rbias column
  edx[, rbias := rbias - .SD, .SDcols = paste0(col, "_bias")]

  # Join average biases with appropriate values in holdout set
  temp <- edx[, .SD[1], .SDcols = paste0(col, "_bias"), by= col]
  final_holdout_test <- temp[final_holdout_test, on = col]

}

# Calculate final predictions by summing all bias columns with global_mean
col_names <- names(edx)
bias_cols <- col_names[str_detect(col_names, "_bias")]
final_holdout_test[, pred := rowSums(.SD) + global_mean, .SDcols = bias_cols]

# Clip final predictions such that they are between 0.5 and 5
final_holdout_test[, pred := fifelse(pred > 5, 5, pred)]
final_holdout_test[, pred := fifelse(pred < 0.5, 0.5, pred)]

# Calculate RMSE of final model
final_holdout_test[, error := pred - rating]
final_rmse <- sqrt(mean((final_holdout_test$error)^2))
sprintf("Final RMSE: %f", final_rmse)
```

```
## [1] "Final RMSE: 0.864215"
```

As can be seen above, a final RMSE value of slightly over 0.8642 is yielded when applying the resulting model to the holdout set. This represents a nearly 18.5% decrease from the baseline RMSE of approximately

1.06 calculated earlier.

One way to delve more deeply into the final model's performance is to plot the prediction errors against input columns. This will show the extent to which model performance varies by input. Ideally, model performance would be consistent across inputs. If the model suffers a significant increase in error when confronted with a certain input, this may be a good area of inquiry if one seeks to improve the model's overall performance. In this case, box plots are made comparing the values of certain columns to their respective error distributions. Three input columns are used for this purpose: *movie_era*, *n_genres*, and *rev_year*.

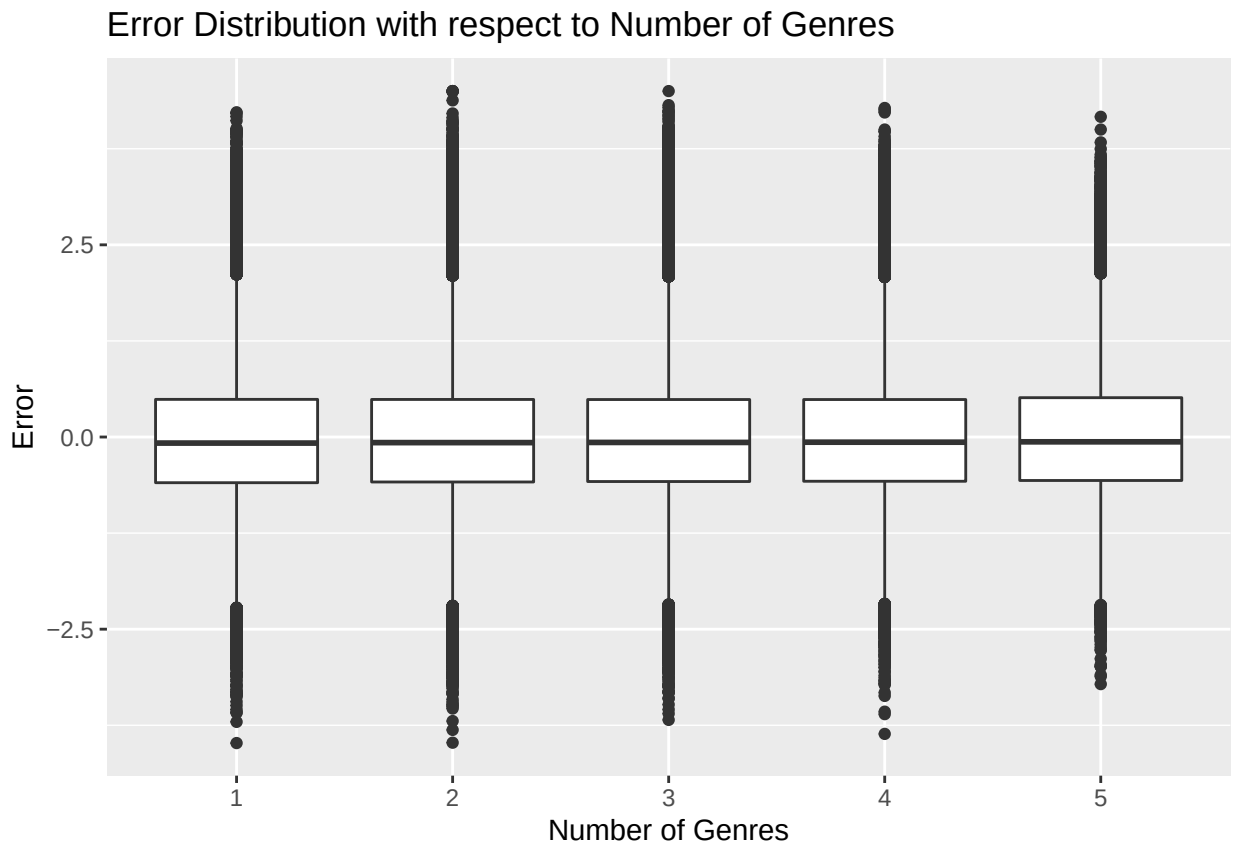
The following chart relates errors to their associated values in *movie_era,* a column created by grouping movie years into eras.

```
# Visualize relationship between movie era and model errors
final_holdout_test |> ggplot(aes(movie_era, error, group = movie_era)) +
  geom_boxplot() +
  ggtitle("Error Distribution with respect to Movie Era") +
  xlab("Movie Era") + ylab("Error")
```


The first quartile, median, third quartile, and range (excluding outliers) of each box plot stay fairly consistent across movie eras, with the notable exception of the first box. This first box is likely skewed because few observations are found corresponding to the era of 1905 through 1919. The total range of values in each plot (including outliers) appears to increase somewhat as the eras become more recent, likely due the increased number of observations associated with those eras.

The next chart compares errors to their associated values in *n_genres*, a column corresponding the number of non-null genres in each row.

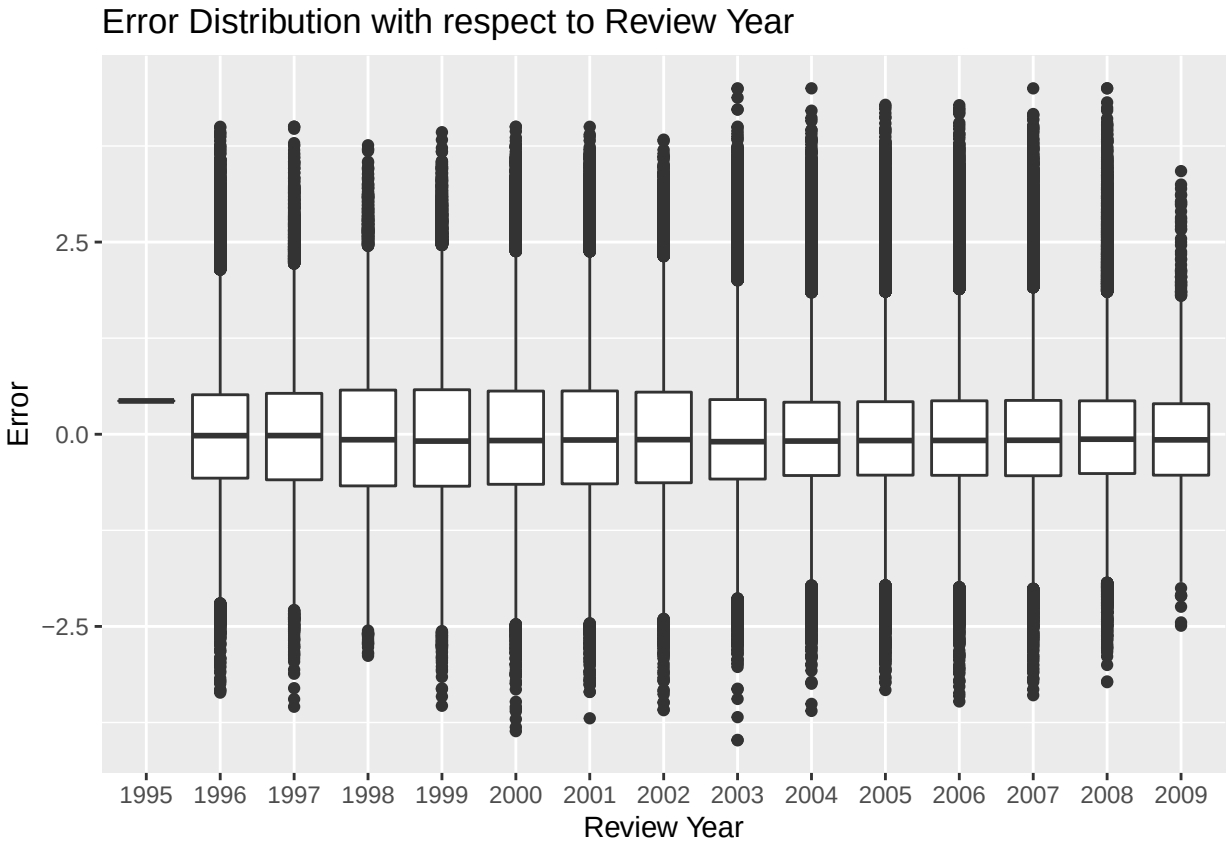
```
# Visualize relationship between first genre and model errors
final_holdout_test |> ggplot(aes(n_genres, error, group = n_genres)) +
  geom_boxplot() +
  ggtitle("Error Distribution with respect to Number of Genres") +
  xlab("Number of Genres") + ylab("Error")
```




Like in the previous graph, the median errors stay fairly consistent across the X axis. Also like the previous graph, genre counts associated with fewer observations tend to have a smaller total range of errors. This is exemplified by the 5th plot, which represents the genre count with the lowest number of observations. The outliers of the 3rd plot appear to be shifted up slightly relative to the others, but it is hard to tell if this is a real effect.

The final chart compares errors to their associated values in *rev_year*, a column corresponding to the year that each rating was made.

```
# Visualize relationship between review year and model errors
final_holdout_test |> ggplot(aes(rev_year, error, group = rev_year)) +
  geom_boxplot() +
  ggtitle("Error Distribution with respect to Review Year") +
  xlab("Review Year") + ylab("Error")
```




As with the other charts, the median errors are fairly consistent apart from an initial single-observation value. The general correlation between total error range and number of observations is also evident to a small extent. One interesting effect seen in this graph is that box and whisker sizes (not including outliers) appear to shift over the X axis. This effect seems to be related to similar movements in rating variance over time. In other words, a *rev_year* value associated with more consistent ratings seems to yield more consistent errors.

Another way to investigate model performance is to calculate its mean error. This is a good indication of the extent to which a model is biased.

```
# Calculate final model's mean error
sprintf("Mean model error: %f",mean(final_holdout_test$error))
```

```
## [1] "Mean model error: -0.002505"
```

The yielded value of approximately -0.0025 is very close to zero, which suggests that the final model has low bias.

## Conclusion

Many steps were taken over the course of this project to design and implement a movie recommendation system. First, a simple model was defined to yield a baseline RMSE. The data were then explored in preparation for cleaning and the creation of new columns. The only data cleaning found to be necessary was the removal of a second movie ID associated with a particular film. An assortment of new columns were created based on the existing data: 5 individual genre columns and a column containing the quantity of

non-null genres were all derived from the existing *genres* column; movie year and movie era were both derived from the existing *title* column; and columns containing the hour, day, month, and year that each rating was created had been derived from the existing *timestamp* column. These new columns were used along with the existing *userId* and *movieId* columns to train a model whose function was based on the average rating biases associated with model inputs. The values of two hyperparameters, *era_len* and *lambda*, were determined via a model-tuning loop before the final model was trained. These hyperparameters affected the manner in which the movie era column was calculated and the extent to which average biases were regularized, respectively. Once the best hyperparameter values were found and used to train the final model, this model was evaluated on the holdout dataset. This evaluation yielded an RMSE of 0.8642, representing a significant improvement over the baseline RMSE calculated earlier.

A comparison of the values in 3 input columns to their respective error distributions revealed that errors stay fairly consistent across input values. Exceptions to this typically come in the form of values with few associated observations. It is interesting to note that the median errors associated with individual input values were almost always slightly below zero. This does not mean that the model is biased, however, as its mean error was found to be very close to zero. One limitation of this project, resulting from computational constraints, was the relatively low number of iterations devoted to each hyperparameter combination in the model-tuning loop. The use of 10 iterations rather than 3 would simplify hyperparameter selection by making the choice more obvious. Another limitation of this work was the use of a single regularization hyperparameter (*lambda*). The use of a unique regularization hyperparameter for each input column would likely result in more optimized regularization during model training. This would be very computationally expensive, however. A second alternative would be to group columns together and assign a regularization hyperparameter to each group. A third limitation of this project, also due to computational constraints, was that many reasonable hyperparameter values could not be tested during model tuning. Increasing the number of values tested, both by increasing their range and decreasing the interval between them, may cause a more optimal set of final hyperparameter values to be chosen. All of these limitations represent opportunities for improvement in future related work.