# Predicting Movie Ratings

Luke Fiorio

6/21/2020

## Introduction

**Project Summary and Goal:**

This project uses data from the GroupLens research lab on movie ratings to build a movie recommendation system.

Our goal is to make a movie recommendation system by predicting, as accurately as possible, how users will rate movies. Specifically, we aim to build a model that predicts users' movie ratings with minimal root-mean-square error (RMSE).

**Data Description:**

The dataset that we will use to build and test our model contains approximately 10 million movie ratings made by nearly 70,000 different users for more than 10,000 different movies. Our data has 1 target variable (`rating`) and 5 predictors (`userId`, `movieId`, `timestamp`, `title`, `genres`).

Movie ratings range from 0.5 up to 5, in half-point (0.5) increments. Each row in our data represents a movie rating (`rating`), given by a specific user (`userId`) for a specific movie (`movieId`). The movie's title (`title`) and genre(s) (`genres`) are also provided. The `timestamp` field indicates when the movie rating was given. [1]

**Key Steps:**

- Split the data into two datasets for:
  - **training** (90%, ~9 million ratings)
  - **validation** (10%, ~1 million ratings)
- Use k-fold crossvalidation to train a predictive model using regularization and GAM (Generalized additive models).
  - estimating the following effects:
    - user effect: how much a user tends to rate movies above or below average
    - movie effect: how much a movie tends to be rated above or below average
    - genre effect: how much a genre (or combination of genres) tends to be rated above or below average
    - time effect: how much ratings made in certain time periods tend to be above or below average
  - tuning the following parameters:
    - lambda: The penalty term to stabilize the estimates made on smaller sample sizes
    - span: The range of x-values used to make smoothed predictions
- Make predictions on the validation set using the optimally tuned model built on the training data.

[1] A movie may have multiple genres. The timestamp field is measured in Epoch (aka: Unix) time, so each `timestamp` is the number of seconds that have elapsed since January 1, 1970.

## Methods and Analysis

In this section, we explain the methods and analysis done to process, explore, and build our movie recommendation model.

**Data Cleaning:**

We start by loading the necessary packages.

- tidyverse
- caret
- data.table
- gam
- lubridate
- scale
- gridExtra

Then retrieving the movie ratings data from GroupLens and combining the two files into `movielens`.

```
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")
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))

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

Before we go any further, we need to split our `movielens` data into training and validation datasets. We'll set a seed for replicability, and then set aside 10% of our data to use **only** for validation of our final model.

```
set.seed(1, sample.kind="Rounding") # set seed

test_index <- createDataPartition(y = movielens$rating,
                                  times = 1,
                                  p = 0.1,
                                  list = FALSE)
edx <- movielens[-test_index,] # train data
temp <- movielens[test_index,] # validation data
```

Next, we'll make sure that our validation data doesn't include any ratings tied to a movie or user that aren't in the training data. We then add those records back into the training data to help build our model.

```
# remove rows without a movie or user match
validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

# Add removed rows to training data
removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)
```

Now that we've separated out our training and validation data, let's add a date field - rounded to the nearest month - to our training data.

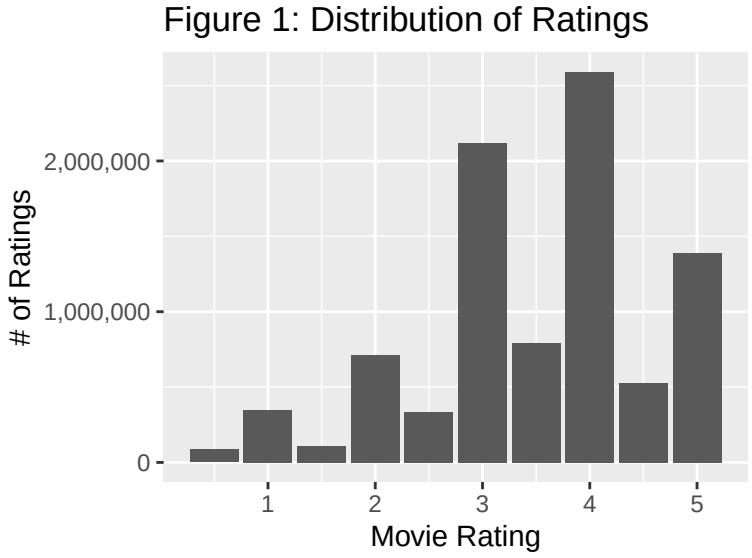
```
edx <- edx %>%
  mutate(date = round_date(as_datetime(timestamp), unit = "month"))
```

**Data Exploration and Visualization:**

To better understand our data, let's explore it.

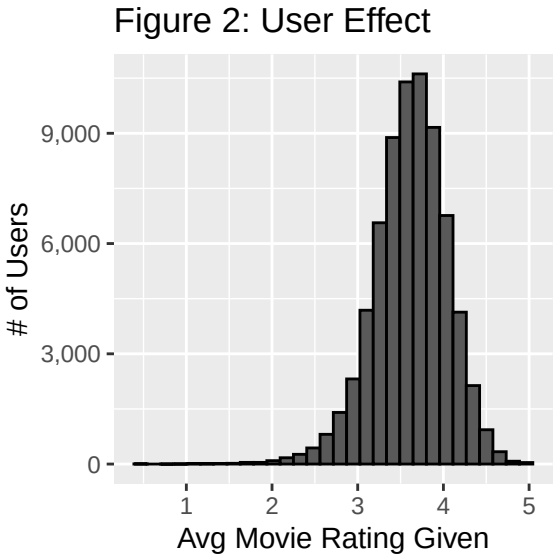
A few simple commands show us that there are exactly 9,000,055 ratings in our training data. These ratings were made by 69,878 different users across 10,677 different movies.

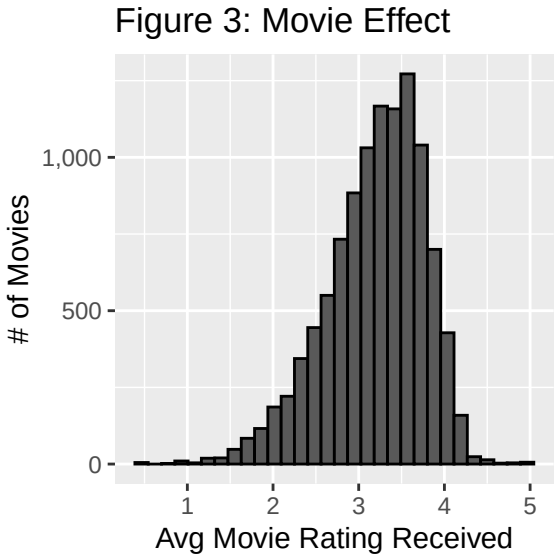
In Figure 1, we see that the distribution of ratings tends toward whole number ratings, with 4-star ratings being most common.




In fact, the average movie rating is 3.51.

Examining the distribution of average movie ratings, by user (Figure 2), shows that some users tend to rate movies more generously than others. And, similarly, we see that some movies also tend to be rated more favorably than others (Figure 3). So, it appears that there is both a *user effect* and a *movie effect*.




Examining the distribution of average movie ratings, by genre (Figure 4), reveals that there is also variability

in how different genres are rated. Plotting movie ratings over time using a smoothing function shows that timing, too, has an effect on ratings (albeit, to a lesser extent). And so it appears that there is also a *genre effect* and a *time effect*.

Figure 4: Genres Effect

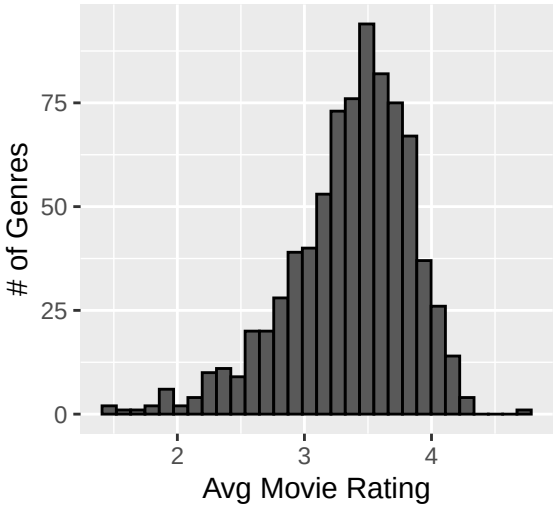
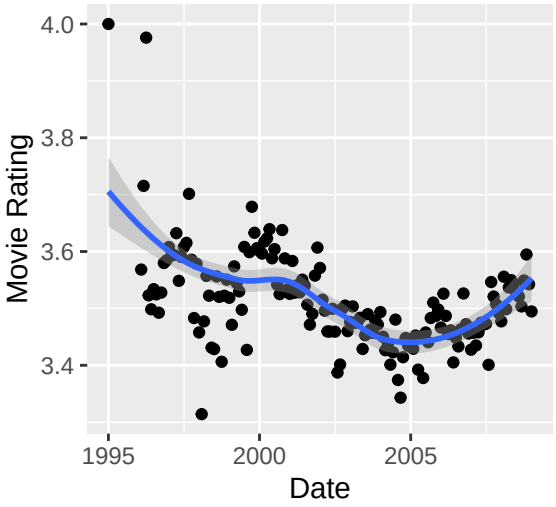



Figure 5: Time Effect




Let's look at the best- and worst-rated genres to better understand the genre effect. In Figure 6, we see the top- and bottom-rated genres seem to have a huge genre effect, but have gotten barely any ratings. Furthermore, the genre combinations seem quite unusual.

In Figure 7, we only plot the top and bottom-rated categories for *common* genres, those with at least 25,000 ratings. Now we see a more expected list of genres with average ratings that are not nearly as polarizing. So, we see that uncommon genres with fewer ratings are more susceptible to having a very large estimated genre effect.

Figure 6: Best/Worst Genres

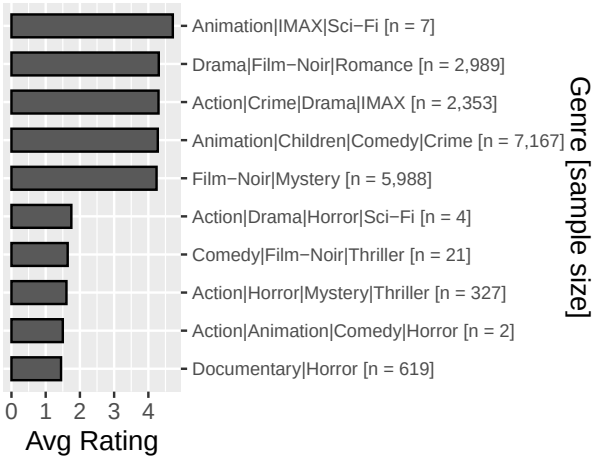
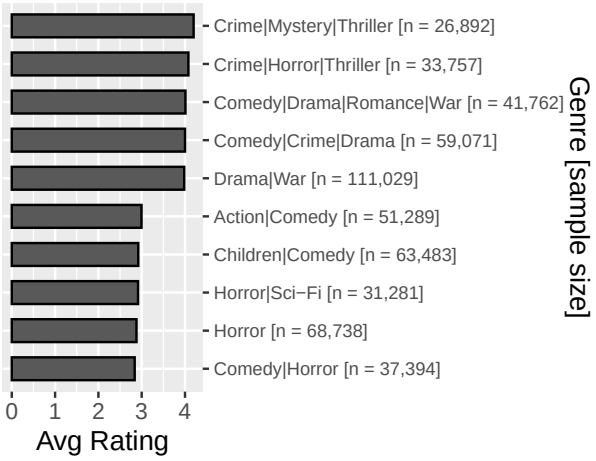



Figure 7: Best/Worst Common Genres




Examining the movie effect and user effect (not presented here), reveals a similar trend.

**Insights Gained:**

Through data exploration, we've revealed that movie ratings aren't completely random. We know that there is an effect associated with the **user** making the rating, the **movie** being rated, the movie's **genre**, and (to a lesser extent) the **time** the movie was rated.

Furthermore, we know that users, movies, and genres with fewer ratings are more susceptible to having an

average rating near one of the polar extremes. For the time effect, we use a smoothing function to avoid that pitfall, but we will need to use another approach to address it for users, movies, and genres (which are categorical; not continuous).

**Modeling Approach:**

With these insights, we can devise a modeling approach. To start, our best estimate for any given rating is the overall mean across all ratings: 3.51.

Of course, for a given user, movie, genre, and time, we know we can make a better estimate than that. However, we need to make we sure that we don't overtrain our model with large estimated effects for a user, movie, or genre that has few ratings.

Regularization (Ridge Regression, in our case) offers a way to do this by introducing a penalty term, *lambda*, to our error calculation. By adding this fixed parameter, we penalize estimates that are less important to the overall model; typically those with smaller sample size. In other words, imposing this penalty reduces the estimated coefficients of each effect that we model - and disproportionately so for users, movies, and genres with fewer ratings.

The purpose of regularization is to make our model more generalizable, so that it will perform better in the "real world" once we expand outside of our training data.

Using regularization, we will estimate each effect one-by-one, controlling for the previous effects at each step in the process. In order, we'll estimate the movie effect, the user effect, the genre effect, and finally the time effect. We will apply penalty terms to the movie, user, and genre effect, but using a smoothing function to estimate the time effect.

We use k-fold crossvalidation on our training data to tune our parameters. Specifically, we search for the optimal penalty term (lambda) and the optimal smoothing range (span, for the time effect).

We'll use the following functions to calculate our RMSE and RSS (Residual Sum of Squares):

```
RMSE <- function(actuals, predictions){
  sqrt(mean((actuals - predictions)^2))
}

RSS <- function(actuals, predictions) {
  sum((actuals - predictions)^2)
}
```

And make our predictions and estimate our error using the `predict_ratings` function. `predict_ratings` takes in two datasets (one for training, one for testing) and two parameters (the lambda penalty term and span smoothing parameter). It builds a model on the training data, makes predictions on the test (or crossvalidated) data, and then calculates the model error.

```
predict_ratings <- function(train_data, new_data, lambda, span) {

  # avg rating in training data
  mu_hat <- mean(train_data$rating)

  # movie effect (regularized)
  b_i <- train_data %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu_hat)/(n()+lambda))

  # user effect (regularized)
  b_u <- train_data %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
```

```
  summarize(b_u = sum(rating - b_i - mu_hat)/(n()+lambda))

# genre effect (regularized)
b_g <- train_data %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  group_by(genres) %>%
  summarize(b_g = sum(rating - b_i - b_u - mu_hat)/(n()+lambda))

## time effect (smoothed model, using gam)

# append movie, user, genres effect to training data
train_data <- train_data %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  left_join(b_g, by="genres")

# train model to estimate smoothed time effect
# (CONTROLLING for the movie, user, genre effects)
train_loess_b_t <- train(
  rating - b_i - b_u - b_g ~ date, # control for the other effects
  method = "gamLoess",
  tuneGrid=data.frame(span = span, degree = 1),
  trControl = trainControl(method = "none"),
  data = train_data)

# drop movie, user, genre effect columns from training data
train_data <- within(train_data, rm(b_i, b_u, b_g))

# use TRAINED gam model to PREDICT rating for each date in test data
# note: we have FINISHED the gam model training.
# this is NOT using the test data to train the model.
new_data['b_t_hat'] <- predict(train_loess_b_t, newdata=new_data)

# calculate b_t using smoothed values predicted by trained gam model
# do NOT regularize the time effect (it's already smoothed)
b_t <- new_data %>%
  group_by(date) %>%
  summarize(b_t = sum(b_t_hat - mu_hat)/n())

# make predictions on test data
predicted_ratings <-
  new_data %>%
  left_join(b_i, by = "movieId") %>%
  left_join(b_u, by = "userId") %>%
  left_join(b_g, by = "genres") %>%
  left_join(b_t, by = "date") %>%
  mutate(pred = mu_hat + b_i + b_u + b_g + b_t) %>%
  pull(pred)

# return parameter values and accuracy stats
# note: the RSS is used to calculate the pooled RMSE in k-fold crossvalidation
return(data.frame(
```

```
    lambda=lambda,
    span=span,
    rmse=RMSE(new_data$rating, predicted_ratings),
    rss=RSS(new_data$rating, predicted_ratings),
    n=nrow(new_data)
  ))

}
```

We'll also define two more functions, `crossvalidate` and `summarize_cv_stats` (not presented here) that split our training data into folds for crossvalidation and calculate the pooled RMSE associated with each unique parameter combination (lambda, span).

The last steps we take before building our model are to assign a fold index to each row in on our training data for crossvalidation...

```
k <- 5 # number of folds
folds <- createFolds(y = edx$rating, k = k, list = FALSE) # assign each row to a fold
```

...and to specify our tuning grid to optimize lambda and span.

```
grid <- expand.grid(
  fold_index = seq(from=1, to=k, by=1),
  lambda = seq(from=4, to=6, by=1),
  span = seq(from=0.05, to=0.15, by=0.05))
```

Now, we build and tune our model, using crossvalidation on our training data.

```
# call `crossvalidate()` and store results for each tune/fold
cv_fold_stats <- mapply(crossvalidate,
                        fold_index=grid$fold_index,
                        lambda=grid$lambda,
                        span=grid$span,
                        MoreArgs = list(data=edx)
                        )

# summarize croosvalidated results for each tune
cv_results <- summarize_cv_stats(cv_fold_stats)
```

Inspecting our RMSE for each parameter in our tune grid shows us the combination of lambda and span that resulted in the lowest error on our crossvalidated data.

Figure 8: Crossvalidated RMSE by lambda (color=span)

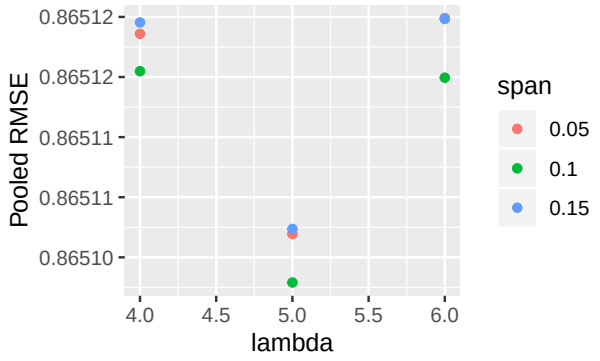
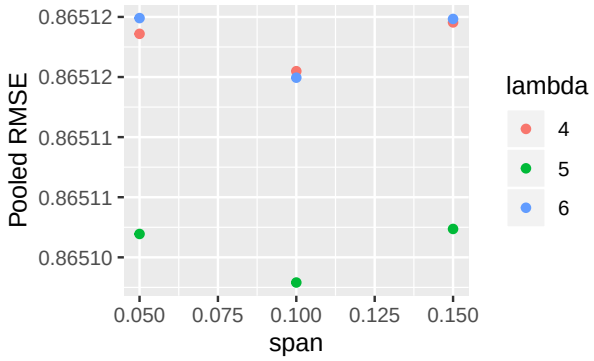



Figure 9: Crossvalidated RMSE by span (color=lambda)



We store the information from the best performing model to apply to our validation dataset.

```
best_index <- which.min(cv_results$pooled_rmse) # index of best model
best_lambda <- cv_results$lambda[best_index]
best_span <- cv_results$span[best_index]
```

As we saw in Figures 8 and 9, our optimal parameters are lambda = 5 and span = 0.1.

Before we evaluate our model on the final validation data, we need to add a date field rounded to the nearest month (as we did for the training data).

```
validation <- validation %>%
  mutate(date = round_date(as_datetime(timestamp), unit = "month"))
```

And now let's make our final predictions and calculate our model error:

```
test_results <- predict_ratings(edx, validation, best_lambda, best_span)
```

## Results

Examining the results of our model on the validation dataset shows that it has achieved an RMSE of **0.86441**, as shown below.

```
round(test_results$rmse,digits=5)
```

```
## [1] 0.86441
```

Which is to say the error we typically make when predicting a movie rating is about 0.864 stars.[2] Overall, there is room for improvement, but it's a good base from which to build off of. Our predictions are typically within at least one star of the actual rating.

Another measure to consider is generalizability. We introduced a penalty term, lambda, in the hopes that it would make our model more generalizable. Comparing our final RMSE of 0.86441 to our crossvalidated RMSE of 0.8651, we see that they are quite similar, and that the final RMSE is even slightly *better* than the crossvalidated RMSE.

Since RMSE is a random variable, this is entirely possible, and actually a good sign that our model is, in fact, generalizable.

## Conclusion

**Summary:**

In conclusion, using movie ratings data from GroupLens research lab we successfully created a movie recommendation system. With a good level of accuracy, our model is able to predict how a user will rate a movie of a given genre at a given point in time.

Our model is generalizable in that it predicts just as well on new data as it does on the data it was trained on. It is ready for use out in the wild.

**Limitations:**

The model is limited by a few factors. It does not factor in that certain groups of movies may be rated similarly through factors analysis. It also engineered few new features; in fact, `date` was the only new feature created. Thirdly, parameter tuning could have been more exhaustive to find the very precise parameter values, *lamda* and *span*, that would further minimize the RMSE on our crossvalidated data. The model's parameter tuning was limited due to computational constraints.

**Future work:**

It is believed that by addressing the limitations presented above, RMSE could be even further improved. Specifically, the addition of factors analysis, more feature engineering, and hyper-parameter tuning are suggested. In particular, feature engineering may be a particularly under-developed area of research in this modeling. More processing of genres (e.g. dummy-encoding for individual genres) or even evaluation of movie titles through Natural Language Processing (NLP) could prove fruitful to future improvements.

[2] We can interpret RMSE as the standard deviation of our errors (the difference between our model's predictions and the actual observed values).