movielens-project

Steve Kruit

2022-10-20

MovieLens project

1. Overview

Goal

The goal of this project is to create a movie recommendation system by building a model that predicts the rating a given user will give a selected movie, using the MovieLens dataset. This project builds on the code already provided and seeks to improve on the prediction methods used earlier in the course by reducing the RMSE of the model's predictions.

Dataset

The 10M version of the MovieLens dataset is used for this project. This dataset includes 10 million movie ratings applied to 10,000 movies by 72,000 users. The dataset includes the following columns:

- **userId:** Unique ID number for each user
- **movieId:** Unique ID number for each movie
- **rating:** Numerical rating out of 5 given by a user for a movie
- **timestamp:** Timestamp recording the time the review was submitted
- **title:** Movie title and year
- **genres:** String listing all genres the movie fits into

Key steps

The MovieLens dataset is partitioned into two sets:

- **edx** is used to train the model
- **validate** is used to test the final model's accuracy

The **edx** set is further partitioned into **train_set** and **test_set** to train and validate the model. **train_set** is used to train prediction models based on different variables and **test_set** is used to test the effect of the prediction models on the RMSE of predicted movie ratings. Exploratory data visualisation is also used to assess whether different variables appear to affect the movie ratings.

The following variables are considered for this analysis:

- The mean rating given by users for a given movie
- The mean rating given by a particular user for other movies

- Time taken between a movie's release and the time of the review
- Time taken between a user's first ever reivew and the given review
- Movie genres
- The mean rating given by similar users for a given movie
- The mean rating given by a particular user for similar movies

The effect of each of the above variables on the RMSE of predictions is assessed and a final model is built based on variables that minimise RMSE.

As models include a larger number of variables, calculation times tend to increase markedly. As such, consideration has also been given to methods that reduce calculation times.

The final model is tested on the **validation** set to assess whether the new methods introduced reduce the RMSE of predictions.

2. Method

The **edx** and **validation** sets are created using the code below (as given in the project instructions).

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

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

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

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

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

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

# # if using R 3.6 or earlier:
# movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
#                                            title = as.character(title),
#                                            genres = as.character(genres))
# if using R 4.0 or later:
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
                                           genres = as.character(genres))


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

# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
```

```
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)
```

We will split the **edx** set into train and test sets. First we check how many rows are in the dataset to determine what percentage we can use as the test set.

```
nrow(edx)
```

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

As we have over 9 million rows, we can use as little as 10% of the **edx** set for our test set and still have close to a million rows of data. The code below partitions the **edx** set into **train_set** (90%) and **test_set** (10%).

```
# set random seed and use data partitioning to create train and test sets from the edx dataset
set.seed(7623, sample.kind = "Rounding")
test_ind <- createDataPartition(edx$rating, times = 1, p = 0.1, list = FALSE)
train_set <- edx %>% slice(-test_ind)
temp <- edx %>% slice(test_ind)

# Make sure userId and movieId in test set are also in train set
test_set <- temp %>%
  semi_join(train_set, by = "movieId") %>%
  semi_join(train_set, by = "userId")

# Add rows removed from test set back into train set
removed <- anti_join(temp, test_set)
train_set <- rbind(train_set, removed)
rm(temp, removed)
```

We define the RMSE function we will be using to evaluate our predictions (as was done in the 'Recommendation Systems' section of the textbook).

```
RMSE <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

We start similarly to the comprehension check in section 6.2 of the Machine Learning course, first by calculating the mean of all ratings and fitting a naive Bayesian model with the function below (as in 6.2, we won't use 'hat' notation in the code).

$Y_{u,i} = \mu + e_{u,i}$

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

```
## [1] 3.512565
```

As in section 6.2, we will be comparing the RMSE of different approaches as we go along.

```
rmse_results <- tibble(method = "Just the average", RMSE = RMSE(test_set$rating, mu))
rmse_results
```

```
## # A tibble: 1 x 2
##   method            RMSE
##   <chr>            <dbl>
## 1 Just the average  1.06
```

Regularised movie and user effect models

The first 2 models we will include are the regularised movie and user effect models that were demonstrated to reduce the RMSE of predictions in section 6.2. We will calculate the RMSE using these models, and use this as the baseline by which we will assess whether we can improve the RMSE over the methods already provided in the course material. We will also introduce functions to improve calculation speeds and reduce duplication of code.

Model 1: Regularised movie effects Our first model will calculate the mean rating for each movie and apply a movie bias term b_i, giving the following equation:

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

Section 6.2 demonstrated that the use of penalised least squares regularisation improved model accuracy, so we define functions to calculate b_i for a given value of λ and then predict the values of $Y_{u,i}$.

```
calculate_b_i <- function(training_data, lambda, m=mu){
  training_data %>%
    group_by(movieId) %>%
    summarise(b_i = sum(rating - m)/(n() + lambda))
}

predict_model_1 <- function(training_data, test_data, lambda, m=mu){
  b_i <- calculate_b_i(training_data, lambda)
  test_data %>%
    left_join(b_i, by = "movieId") %>%
    mutate(pred = m + b_i) %>%
    pull(pred)
}
```

As in section 6.2, we will use cross-validation to choose the tuning parameter λ. We then run the regularised movie effects prediction model using the optimal value of λ. We also record the calculation time.

```
# record start time so we can monitor how long the calculation takes
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
library(lubridate)
```

```
time1 <- now()

# test RMSE with lambda values from 0 to 10, in increments of 0.25
lambdas <- seq(0, 10, 0.25)
rmses <- sapply(lambdas, function(l){
  predictions <- predict_model_1(train_set, test_set, l)
  return(RMSE(predictions, test_set$rating))
})
lambda <- lambdas[which.min(rmses)]

# and then run the model using the optimal lambda and calculate the RMSE
predicted_ratings <- predict_model_1(train_set, test_set, lambda)
model_1_rmse <- RMSE(predicted_ratings, test_set$rating)

# record the time after calculations have completed
time2 <- now()

print(lambda)
```

```
## [1] 2
```

```
print(model_1_rmse)
```

```
## [1] 0.9448493
```

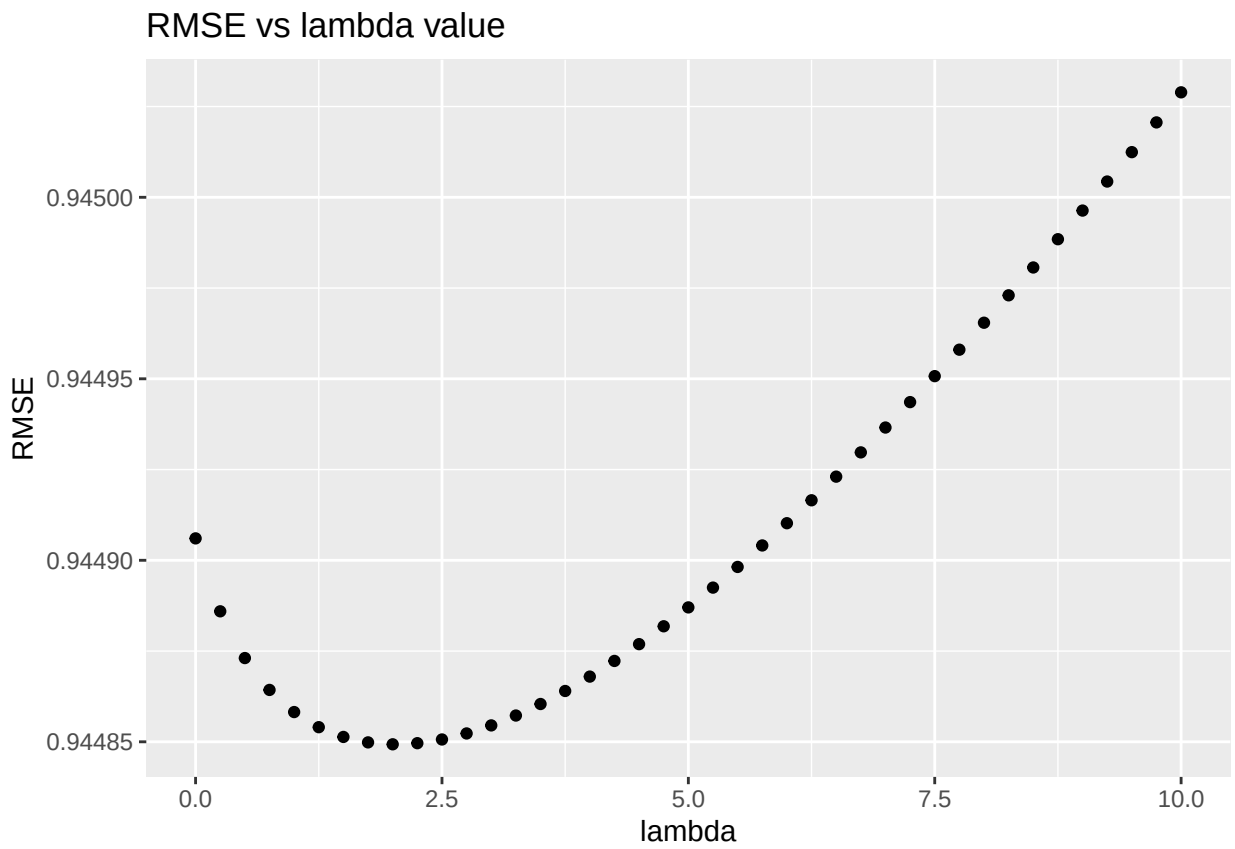
It takes some time to perform the required calculations:

```
time2 - time1
```

```
## Time difference of 1.300221 mins
```

We examine 41 different values of λ during cross-validation. If we look at the RMSE for each of the values of λ graphically, we see that the RMSE values form a parabolic shape, indicating that we can potentially reduce the calculation time if we are a bit smarter about which values of λ we examine.

```
tibble(lambda = lambdas, RMSE = rmses) %>%
  ggplot(aes(lambda, RMSE)) +
  geom_point() +
  ggtitle("RMSE vs lambda value")
```



To reduce run time, we will implement an algorithm that first assesses λs in increments of 2, then assesses λs an increment of 1 away from the best of these λs, then 0.5 from the best of these, then 0.25 from the best of these - thus selecting the best λ at a sensitivity of 0.25 while only examining a total of $5 + 2 + 2 + 2 = 11$ different values of λ (compared with the original 41).

First we define a function to calculate RMSEs for a given list of λs using a specified prediction function (so it can be replicated for each of the models we assess).

```
calculate_RMSEs <- function(prediction_function, lambdas, training_data, test_data){
  sapply(lambdas, function(l){
    predictions <- prediction_function(training_data, test_data, l)
    return(RMSE(predictions, test_data$rating))
  })
}
```

And then define a function to return the best λ and corresponding RMSE for a given list of λs.

```
best_lambda <- function(prediction_function, lambdas, training_data, test_data, lambda_rmse){
  best <- lambda_rmse
  rmses <- calculate_RMSEs(prediction_function, lambdas, training_data, test_data)
  if(min(rmses) < lambda_rmse$rmse){
    best$rmse <- min(rmses)
    best$lambda <- lambdas[which.min(rmses)]
  }
  best
}
```

Finally, we define the function to narrow in on the best λ by examining small subsets of the total list of the sequence seq(0, 10, 0.25).

```
optimise_lambda <- function(prediction_function, training_data, test_data){
  best <- tibble(lambda = 0, rmse = 9999)
  lambdas <- seq(1, 9, 2)
  best <- best_lambda(prediction_function, lambdas, training_data, test_data, best)
  lambdas <- c(best$lambda - 1, best$lambda + 1)
  best <- best_lambda(prediction_function, lambdas, training_data, test_data, best)
  lambdas <- c(best$lambda - 0.5, best$lambda + 0.5)
  best <- best_lambda(prediction_function, lambdas, training_data, test_data, best)
  lambdas <- c(best$lambda - 0.25, best$lambda + 0.25)
  best <- best_lambda(prediction_function, lambdas, training_data, test_data, best)
  best
}
```

The above function also returns the RMSE for each value of λ, which saves us from re-running the model using the correct λ to find the value of the lowest RMSE. The function could be written more elegantly as a recursive function but as we're looking at the same range of possible λs every time here we can leave the code as is!

We now confirm that the algorithm returns the same values for λ and the RMSE, and record the calculation time.

```
time1 <- now()
new_method <- optimise_lambda(predict_model_1, train_set, test_set)
time2 <- now()

print(new_method)
```

```
## # A tibble: 1 x 2
##   lambda  rmse
##    <dbl> <dbl>
## 1      2 0.945
```

```
print(new_method$rmse == model_1_rmse)
```

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

```
print(new_method$lambda == lambda)
```

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

We get the same results and have substantially reduced the calculation time.

```
time2 - time1
```

```
## Time difference of 21.21052 secs
```

We will therefore use the algorithmic approach to assess all models from now on.

Finally, we add the RMSE to our results table.

```
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularised Movie Effect Model",
                                 RMSE = model_1_rmse ))
rmse_results
```

```
## # A tibble: 2 x 2
##   method                          RMSE
##   <chr>                          <dbl>
## 1 Just the average               1.06
## 2 Regularised Movie Effect Model 0.945
```

Model 2: Regularised movie and user effects Model 2 will calculate the mean rating for each user and apply a user bias term b_u. We define functions to calculate b_i for a given value of λ and then predict the values of $Y_{u,i} = \mu + b_i + b_u + e_{u,i}$.

```
calculate_b_u <- function(training_data, lambda, b_i, m=mu){
  training_data %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - mu - b_i)/(n()+lambda))
}

predict_model_2 <- function(training_data, test_data, lambda, m=mu){
  b_i <- calculate_b_i(training_data, lambda)
  b_u <- calculate_b_u(training_data, lambda, b_i)
  test_data %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
    pull(pred)
}
```

We determine the optimal value of λ and calculate the RMSE using Model 2 and the optimal λ, then add to our table of results.

```
optimal <- optimise_lambda(predict_model_2, train_set, test_set)
model_2_rmse <- optimal$rmse
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularised Movie + User Effects Model",
                                 RMSE = model_2_rmse ))
rmse_results
```

```
## # A tibble: 3 x 2
##   method                                RMSE
##   <chr>                                <dbl>
## 1 Just the average                     1.06
## 2 Regularised Movie Effect Model       0.945
## 3 Regularised Movie + User Effects Model 0.866
```

As expected, the regularised movie and user effects model substantially improves the RMSE of predictions, as it did in section 6.2. The goal now is to try to improve upon this model using our own methods.

To add to the recommendation system, we will firstly consider time and genre effects (as suggested in the section 6.2 Comprehension Check).

Time effects

We will start with time, which we will split into 2 different variables:

- Time taken between a user's first ever review and the given review
- Time taken between a movie's release and the time of the review

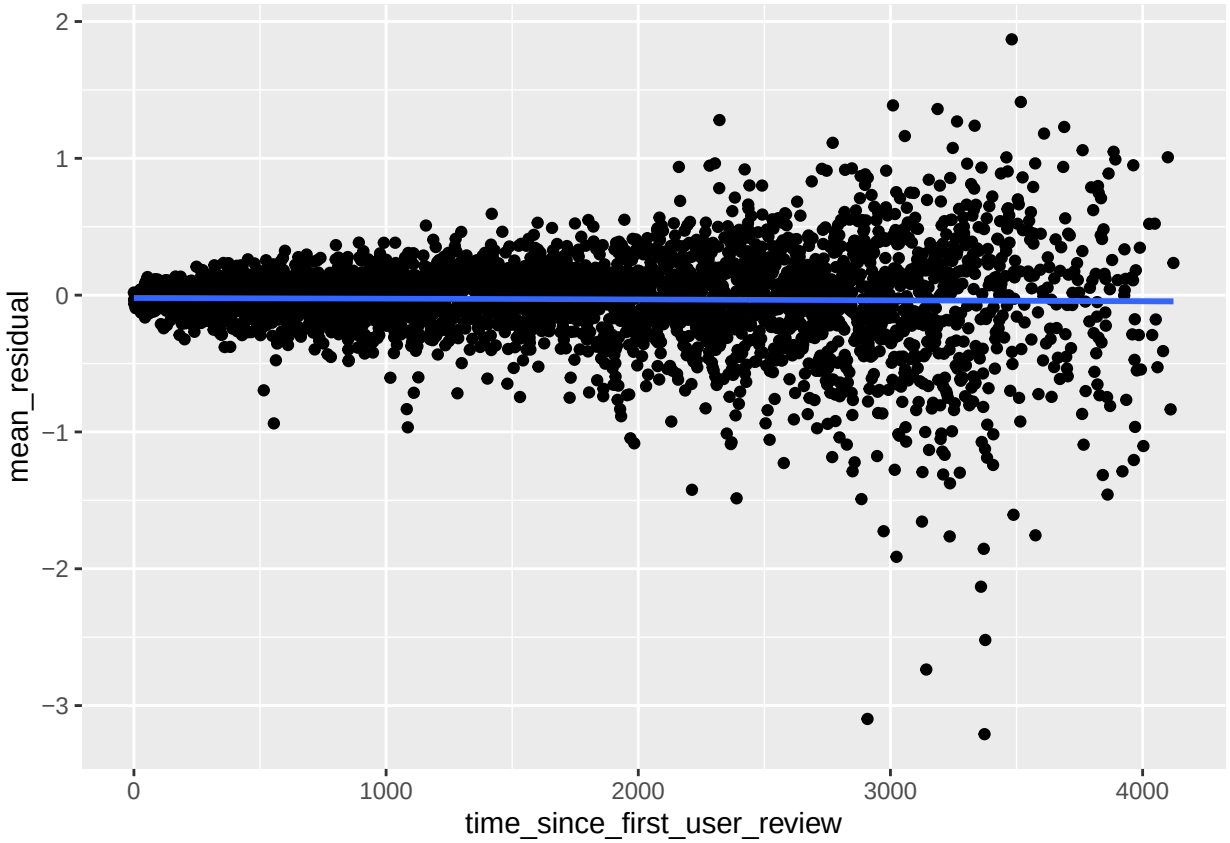
We record the timestamps for first review of each movie and first review by each user.

```
# get timestamp of first review for each movie
first_reviews_by_movie <- edx %>%
  group_by(movieId) %>%
  summarise(first_movie_review = min(timestamp, na.rm = TRUE))

# and timestamp of first review by each user
first_reviews_by_user <- edx %>%
  group_by(userId) %>%
  summarise(first_user_review = min(timestamp, na.rm = TRUE))
```

First we will consider if there is a relationship between user ratings and the time taken between a rating and the user's first ever rating - i.e. do user ratings get more or less generous over time. The chart below shows the mean residuals for our test set as time since first user review increases.

```
test_set %>%
  left_join(first_reviews_by_user, by = "userId") %>%
  mutate(time_since_first_user_review = round((timestamp - first_user_review) / (60 * 60 * 24)),
         pred = predicted_ratings,
         res = rating - pred) %>%
  group_by(time_since_first_user_review) %>%
  summarise(mean_residual = mean(res)) %>%
  ggplot(aes(time_since_first_user_review, mean_residual)) +
  geom_point() +
  geom_smooth()
```



There does not appear to be a relationship between time since first user review and mean ratings. Of course, on an individual level some users may get more or less generous over time but in aggregate there seems to be no evidence of a trend. We could try to build individual models for each user but that would be very computationally expensive so we will not attempt to incorporate this measure in our final model.

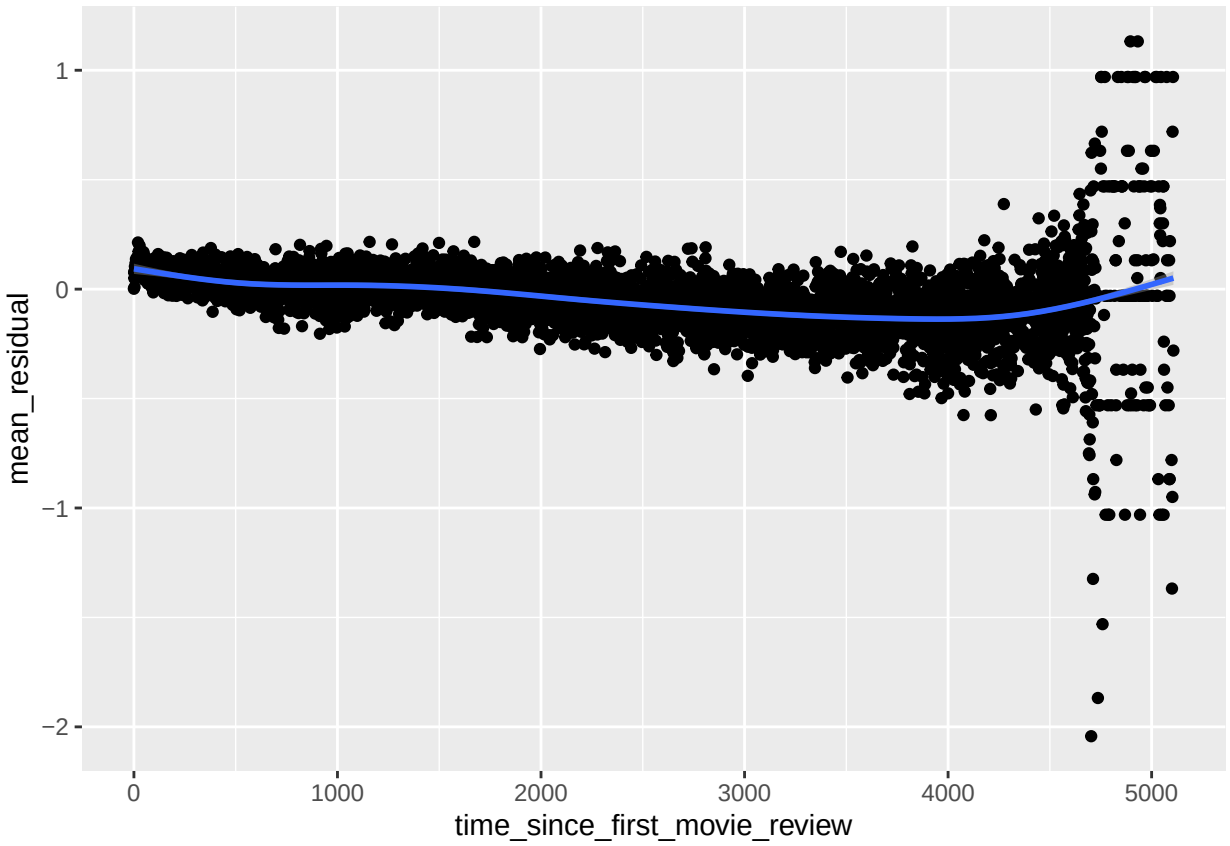
Now we will consider whether there is a relationship between time since a movie's release and rating scores. This is a little tricky because our dataset does not include the release date of movies. The title does include the release year but this doesn't tell us *when* in the year a movie was released - so we will use the timestamp of the earliest review in the **edx** set as a proxy for the release date.

To be certain that we are actually recording the earliest review for each movie we would need to note the timestamp of the first review using both the **edx** *and* **validation** datasets - otherwise we will have an incorrect value where the earliest review happens to be in the **validation** set. However, we have been instructed not to use the **validation** set at all for model training. To be cautious, we calculate time elapsed based on the first reviews found in the **edx** dataset - noting that there will be some resulting inaccuracies if we use this time-based model on the validation set for our final calculation.

Similar to above, we check if there appears to be a visual relationship between time and ratings.

```
test_set %>%
  left_join(first_reviews_by_movie, by = "movieId") %>%
  mutate(time_since_first_movie_review = round((timestamp - first_movie_review) / (60 * 60 * 24)),
         pred = predicted_ratings,
         res = rating - pred) %>%
  group_by(time_since_first_movie_review) %>%
  summarise(mean_residual = mean(res)) %>%
  ggplot(aes(time_since_first_movie_review, mean_residual)) +
```

```
  geom_point() +
  geom_smooth()
```



There appears to be a slight time effect based on time since first movie review.

Model 3: Add effect of time since first movie review We will fit a model to the training data using the loess method (the same method used to show the trend line in the geom_smooth() function used in the graph above) based on time since first movie review.

```
loess_b_ti <- function(training_data, b_i, b_u, first_reviews_by_movie, seed = 142){
  set.seed(seed, sample.kind = "Rounding")
  training_data %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(first_reviews_by_movie, by = "movieId") %>%
    mutate(pred = mu + b_i + b_u,
           error = rating - pred,
           days_since_first_movie_review = round((timestamp - first_movie_review) / (60 * 60 * 24))) %>%
    group_by(days_since_first_movie_review) %>%
    summarise(avg_error = mean(error)) %>%
    loess(avg_error ~ days_since_first_movie_review, data = .)
}

calculate_b_ti <- function(b_ti_loess, range){
  tibble(days_since_first_movie_review = range) %>%
```

```
    mutate(b_ti = predict(b_ti_loess, days_since_first_movie_review))
}

predict_model_3 <- function(training_data, test_data, lambda, first_reviews_by_movie, m=mu){
  b_i <- calculate_b_i(training_data, lambda)
  b_u <- calculate_b_u(training_data, lambda, b_i)
  b_ti_loess <- loess_b_ti(training_data, b_i, b_u, first_reviews_by_movie)
  max_days <- bind_rows(training_data, test_data) %>%
    left_join(first_reviews_by_movie, by = "movieId") %>%
    mutate(days_since_first_movie_review = round((timestamp - first_movie_review) / (60 * 60 * 24))) %>%
    summarise(max = max(days_since_first_movie_review)) %>%
    pull(max)
  b_ti <- calculate_b_ti(b_ti_loess, 0:max_days)
  test_data %>%
    left_join(first_reviews_by_movie, by = "movieId") %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    mutate(days_since_first_movie_review = round((timestamp - first_movie_review) / (60 * 60 * 24)),
           # set days since to zero in cases where they are negative using validation set
           days_since_first_movie_review = ifelse(days_since_first_movie_review < 0,
                                                  0,
                                                  days_since_first_movie_review)) %>%
    left_join(b_ti, by = "days_since_first_movie_review") %>%
    mutate(pred = mu + b_i + b_u + b_ti) %>%
    pull(pred)
}
```

We use the functions defined above to calculate the RMSE using Model 3 and add to our table of results.

```
# predict new ratings
predicted_ratings <- predict_model_3(train_set, test_set, lambda, first_reviews_by_movie)

# test RMSE
model_3_rmse <- RMSE(predicted_ratings, test_set$rating)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Movie + User Effects + Time Effects Model",
                                 RMSE = model_3_rmse ))
rmse_results
```

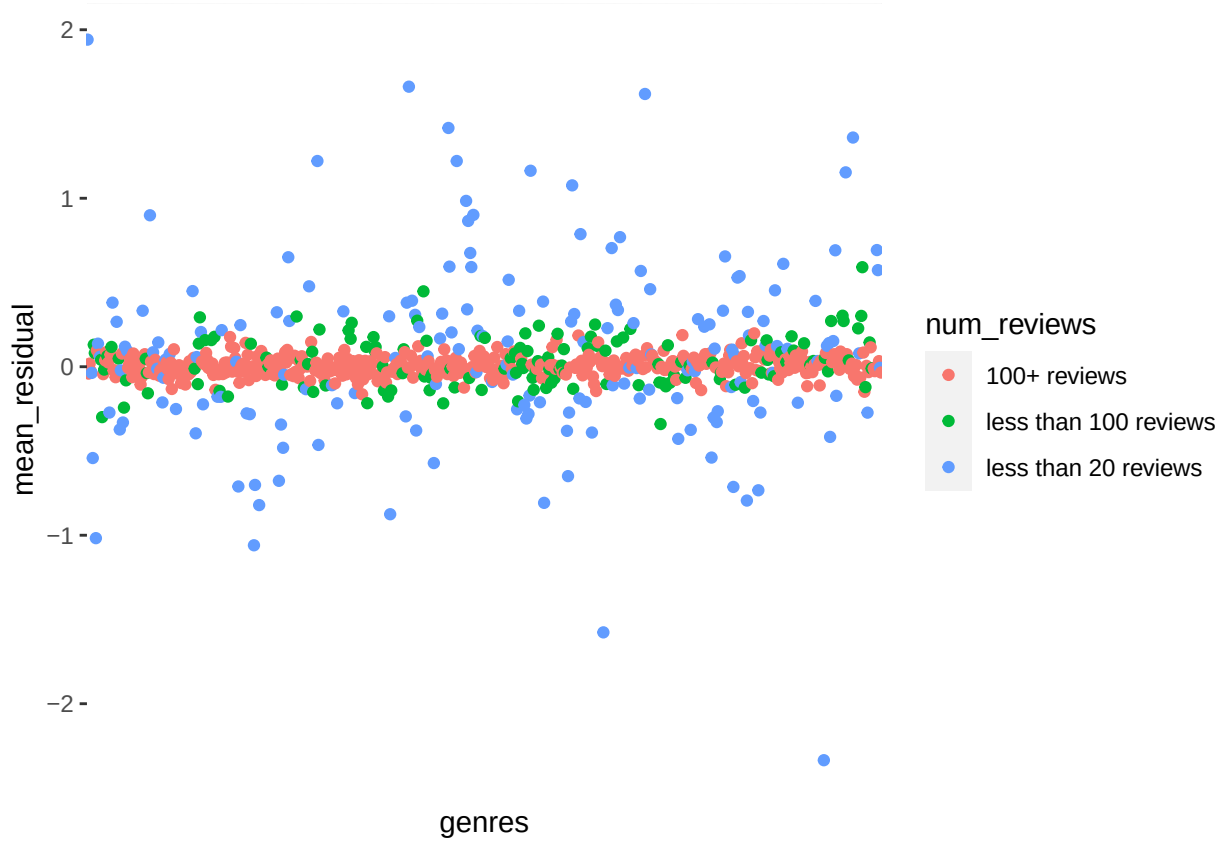
```
## # A tibble: 4 x 2
##   method                                  RMSE
##   <chr>                                  <dbl>
## 1 Just the average                       1.06
## 2 Regularised Movie Effect Model         0.945
## 3 Regularised Movie + User Effects Model 0.866
## 4 Movie + User Effects + Time Effects Model 0.866
```

Improvements from this model appear to be negligible. When we also consider the likelihood of inaccuracies presenting when we use the **validation** set, there appears to be little value added by including the model. We therefore will not include any time effects in our final model.

Genre effects

To determine whether we will include genre effects in our final model, we visually examine residuals for each combination of genres in the data.

```
test_set %>%
  mutate(pred = predicted_ratings,
         res = rating - pred) %>%
  group_by(genres) %>%
  summarise(n = n(), mean_residual = mean(res, na.rm = TRUE)) %>%
  ungroup() %>%
  mutate(num_reviews = ifelse(n < 20, "less than 20 reviews", "less than 100 reviews"),
         num_reviews = ifelse(n >= 100, "100+ reviews", num_reviews)) %>%
  ggplot(aes(genres, mean_residual, color = num_reviews)) +
  geom_point() +
  theme(axis.text.x = element_blank(),
        axis.ticks.x = element_blank())
```



Residuals are typically around zero, except where a particular genre has a small number of reviews (and therefore more variance). As there doesn't seem to be evidence of any genre bias (after accounting for user and movie bias) we don't include genre effects in our final model.

Group effects

Conceptually, it seems likely that people with similar taste in movies will give similar scores for the same movie, and similar movies will receive similar scores from users (as outlined in textbook section 33.11). We

will try clustering similar movies and users together to see if we can improve our recommendation model.

First we will make clusters of similar movies. Because we have a large dataset, we will need to filter it down somewhat to make the calculation workable. Determining exactly how to filter the data took some trial and error, but in the end we will limit **train_set** to the top 1,000 users with the most movie reviews, and then the movies that received at least 10 reviews from those top 1,000 users. These values have been chosen because including more users could lead to very long calculation times, and including movies with fewer reviews could lead to the k-clustering algorithm (which we will be using for this analysis) failing to converge.

```
# filter by top 1,000 users
user_list <- train_set %>%
  group_by(userId) %>%
  summarise(n = n()) %>%
  top_n(1000, n) %>%
  pull(userId)

# further filter by only choosing movies with 10+ reviews
num_movies <- train_set$movieId %>%
  unique() %>%
  length()
num_movies_filtered <- train_set %>%
  filter(userId %in% user_list) %>%
  group_by(movieId) %>%
  filter(n() >= 10) %>%
  ungroup() %>%
  .$movieId %>%
  unique() %>%
  length()
print(paste0("Number of movies: ", num_movies))
```

```
## [1] "Number of movies: 10677"
```

```
print(paste0("Number of movies after filtering: ", num_movies_filtered))
```

```
## [1] "Number of movies after filtering: 8531"
```

As we can see, this approach still leaves us with around 80% of all movies - we will put all of the remaining movies into a group of their own after we perform the clustering. To use the k-clustering algorithm we will need to convert our data to a matrix and replace any NAs with zeroes.

```
# create matrix of all movies with 10+ reviews and our top 1,000 users
x <- train_set %>%
  filter(userId %in% user_list) %>%
  group_by(movieId) %>%
  filter(n() >= 10) %>%
  ungroup() %>%
  select(movieId, userId, rating) %>%
  pivot_wider(names_from = userId, values_from = rating)

# convert to matrix of users and movie scores
row_names <- x$movieId
x <- x[,-1] %>% as.matrix()
x <- sweep(x, 2, colMeans(x, na.rm = TRUE))
```

```
x <- sweep(x, 1, rowMeans(x, na.rm = TRUE))
rownames(x) <- row_names

# convert NAs to zero
x_0 <- x
x_0[is.na(x_0)] <- 0
```

Now that we have our matrix, we will use k-clustering to create 10 groups of movies (this should take around 30 seconds to compute).

```
# Use k-clustering to group movies into 10 groups
if(!require(stats)) install.packages("stats", repos = "http://cran.us.r-project.org")
library(stats)
set.seed(4334)
k <- kmeans(x_0, centers = 10, nstart = 10, iter.max = 100) # gets error message if not enough iteratio
groups <- k$cluster
movie_groups <- tibble(movieId = as.numeric(names(groups)), movie_group = groups)
```

We view a selection of movies from the first group.

```
train_set %>%
  left_join(movie_groups, by = "movieId") %>%
  group_by(movieId) %>%
  summarise(title = first(title), movie_group = first(movie_group)) %>%
  filter(movie_group == 1) %>%
  pull(title) %>%
  .[1:10]
```

```
##  [1] "Nixon (1995)"
##  [2] "City of Lost Children, The (Cité des enfants perdus, La) (1995)"
##  [3] "Dead Man Walking (1995)"
##  [4] "Richard III (1995)"
##  [5] "Postman, The (Postino, Il) (1994)"
##  [6] "Antonia's Line (Antonia) (1995)"
##  [7] "Bottle Rocket (1996)"
##  [8] "Chungking Express (Chóngqìng Senlín) (1994)"
##  [9] "Flirting With Disaster (1996)"
## [10] "Belle de jour (1967)"
```

The movies in group 1 tend to be critically acclaimed dramas. Now we look at a selection from group 2.

```
train_set %>%
  left_join(movie_groups, by = "movieId") %>%
  group_by(movieId) %>%
  summarise(title = first(title), movie_group = first(movie_group)) %>%
  filter(movie_group == 2) %>%
  pull(title) %>%
  .[1:10]
```

```
##  [1] "Jumanji (1995)"
##  [2] "Grumpier Old Men (1995)"
```

```
##  [3] "Father of the Bride Part II (1995)"
##  [4] "Ace Ventura: When Nature Calls (1995)"
##  [5] "Money Train (1995)"
##  [6] "Copycat (1995)"
##  [7] "Assassins (1995)"
##  [8] "Powder (1995)"
##  [9] "Mortal Kombat (1995)"
## [10] "Broken Arrow (1996)"
```

Movies in this group tend to be aimed at a broader audience, with a selection of popular comedies and lowbrow action movies. We won't go through each of the remaining groups but they also tend to have a thematic link of some sort, giving us confidence that the groups we have selected could be useful.

Next we will group the users. We filter **train_set** to include the top 500 movies with the most reviews, and then the users that have reviewed at least 10 of the top 500 films.

```
# first whittle down to top movies
movie_list <- train_set %>%
  group_by(movieId) %>%
  summarise(n = n()) %>%
  top_n(500, n) %>%
  pull(movieId)

# further filter by only choosing users with 10+ reviews
num_users <- train_set$userId %>%
  unique() %>%
  length()
num_users_filtered <- train_set %>%
  filter(movieId %in% movie_list) %>%
  group_by(userId) %>%
  filter(n() >= 10) %>%
  ungroup() %>%
  .$userId %>%
  unique() %>%
  length()
print(paste0("Number of users: ", num_users))
```

```
## [1] "Number of users: 69878"
```

```
print(paste0("Number of users after filtering: ", num_users_filtered))
```

```
## [1] "Number of users after filtering: 66590"
```

Using this approach we still account for the vast majority of users. As with the movies that weren't selected, the users not considered for clustering will be put into a group of their own when we make our predictions later on. As we did before, we will need to create a matrix to perform the k-clustering algorithm.

```
# create matrix of the top movies reviews and users with at least 10 reviews
x <- train_set %>%
  filter(movieId %in% movie_list) %>%
  group_by(userId) %>%
  filter(n() >= 10) %>%
  ungroup() %>%
```

```
  select(movieId, userId, rating) %>%
  pivot_wider(names_from = movieId, values_from = rating)

row_names <- x$userId
x <- x[,-1] %>% as.matrix()
x <- sweep(x, 2, colMeans(x, na.rm = TRUE))
x <- sweep(x, 1, rowMeans(x, na.rm = TRUE))
rownames(x) <- row_names

# convert NAs to zero
x_0 <- x
x_0[is.na(x_0)] <- 0
```

As there are many more users than there were movies, the k-clustering function will take some time to compute - around 8.5 minutes on my machine. We will again create 10 groups, but we will use a different k-clustering algorithm (using algorithm = "MacQueen") to avoid errors.

```
# Use k-clustering to group movies into 10 groups
set.seed(4217)
k <- kmeans(x_0, centers = 10, nstart = 10, iter.max = 1000, algorithm = "MacQueen") # gets error messa
# also need to use MacQueen algorithm to avoid error 'Quick-TRANSfer stage steps exceeded maximum'
groups <- k$cluster
user_groups <- tibble(userId = as.numeric(names(groups)), user_group = groups)
```

Unlike with the groups of movies, we cannot simply look at the user groups to sense-check the groupings, so we will be reliant on the RMSE of our predictions to judge whether the groups used are appropriate.

Firstly, we will test for any movie group effects on ratings.

Model 4: Additional movie group effects Similarly to the earlier calculations for b_i and b_u, we will calculate a new bias term $b_{u,I}$ that measures the bias a user u may have for movies from movie group I. We will again use regularisation in our calculation. Our model now has the formula:

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

We create functions to calculate $b_{u,I}$ and then predict ratings using the new formula for $Y_{u,i}$.

```
calculate_b_uI <- function(training_data, lambda, b_i, b_u, mov_groups, m=mu){
  training_data %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(mov_groups, by = "movieId") %>%
    mutate(movie_group = ifelse(is.na(movie_group), 99, movie_group)) %>% # assign all ungrouped movies
    group_by(userId, movie_group) %>%
    summarise(b_uI = sum(rating - mu - b_i - b_u)/(n()+lambda))
}

predict_model_4 <- function(training_data, test_data, lambda,
                            mov_groups = movie_groups, m=mu){
  b_i <- calculate_b_i(training_data, lambda)
  b_u <- calculate_b_u(training_data, lambda, b_i)
  b_uI <- calculate_b_uI(training_data, lambda, b_i, b_u, mov_groups)
  test_data %>%
    left_join(b_i, by="movieId") %>%
```

```
    left_join(b_u, by="userId") %>%
    left_join(mov_groups, by = "movieId") %>%
    mutate(movie_group = ifelse(is.na(movie_group), 99, movie_group)) %>% # assign all ungrouped movies
    left_join(b_uI, by = c("userId", "movie_group")) %>%
    mutate(b_uI = ifelse(is.na(b_uI), 0, b_uI),
           pred = mu + b_i + b_u + b_uI) %>%
    pull(pred)
}
```

We use the functions above to calculate the optimal lambda and associated RMSE for our new regularised movie + user + movie group effects model. This calculation will take a few minutes.

```
best <- optimise_lambda(predict_model_4, train_set, test_set)
model_4_rmse <- best$rmse
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularised Movie + User + Movie Group Effects Model",
                                 RMSE = model_4_rmse ))
rmse_results
```

```
## # A tibble: 5 x 2
##   method                                              RMSE
##   <chr>                                              <dbl>
## 1 Just the average                                   1.06
## 2 Regularised Movie Effect Model                     0.945
## 3 Regularised Movie + User Effects Model             0.866
## 4 Movie + User Effects + Time Effects Model          0.866
## 5 Regularised Movie + User + Movie Group Effects Model 0.840
```

The new model has noticeably improved the RMSE of our predictions, so we will include movie group effects in our final model. We now investigate user group effects.

Model 5: Additional user group effects As above, we will calculate a new bias term $b_{i,U}$ that measures the bias a user group U may have for movie i. We will again use regularisation in our calculation. Our model now has the formula:

$Y_{u,i} = \mu + b_i + b_u + b_{u,I} + b_{i,U} + e_{u,i}$

We create functions to calculate $b_{i,U}$ and then predict ratings using the new formula for $Y_{u,i}$.

```
calculate_b_iU <- function(training_data, lambda, b_i, b_u, b_uI,
                           mov_groups, u_groups, m=mu){
  training_data %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(mov_groups, by = "movieId") %>%
    left_join(u_groups, by = "userId") %>%
    mutate(movie_group = ifelse(is.na(movie_group), 99, movie_group),
           user_group = ifelse(is.na(user_group), 99, user_group)) %>% # assign all ungrouped movies an
    left_join(b_uI, by = c("userId", "movie_group")) %>%
    group_by(movieId, user_group) %>%
    summarise(b_iU = sum(rating - mu - b_i - b_u - b_uI)/(n()+lambda))
}
```

```
predict_model_5 <- function(training_data, test_data, lambda, mov_groups = movie_groups,
                            u_groups = user_groups, m=mu){
  # calculate bias terms from training data
  b_i <- calculate_b_i(training_data, lambda)
  b_u <- calculate_b_u(training_data, lambda, b_i)
  b_uI <- calculate_b_uI(training_data, lambda, b_i, b_u, mov_groups)
  b_iU <- calculate_b_iU(training_data, lambda, b_i, b_u, b_uI, mov_groups, u_groups)
  # use bias terms to predict test data ratings
  test_data %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(mov_groups, by = "movieId") %>%
    left_join(u_groups, by = "userId") %>%
    mutate(movie_group = ifelse(is.na(movie_group), 99, movie_group),
           user_group = ifelse(is.na(user_group), 99, user_group)) %>% # assign all ungrouped movies an
    left_join(b_uI, by = c("userId", "movie_group")) %>%
    left_join(b_iU, by = c("movieId", "user_group")) %>%
    mutate(b_uI = ifelse(is.na(b_uI), 0, b_uI),
           b_iU = ifelse(is.na(b_iU), 0, b_iU),
           pred = mu + b_i + b_u + b_uI + b_iU) %>%
    pull(pred)
}
```

We use the functions above to calculate the optimal lambda and associated RMSE for our new regularised movie + user + movie group + user group effects model. This calculation will take a few minutes.

```
best <- optimise_lambda(predict_model_5, train_set, test_set)
model_5_rmse <- best$rmse
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularised Movie + User + Movie Group + User Group Effects Mo
                                 RMSE = model_5_rmse ))
rmse_results
```

```
## # A tibble: 6 x 2
##   method                                                      RMSE
##   <chr>                                                      <dbl>
## 1 Just the average                                           1.06
## 2 Regularised Movie Effect Model                             0.945
## 3 Regularised Movie + User Effects Model                     0.866
## 4 Movie + User Effects + Time Effects Model                  0.866
## 5 Regularised Movie + User + Movie Group Effects Model       0.840
## 6 Regularised Movie + User + Movie Group + User Group Effects Model 0.832
```

Once again, our RMSE has improved. We will therefore use Model 5 (regularised movie + user + movie group + user group effects model) as our final model.

Using the final model to predict ratings in the validation set Now that we have determined the content of our final model, we will train the model using the full **edx** dataset and use this model to predict ratings in the **validation** dataset. This will again take a few minutes.

So far our prediction functions have inputted both the training and test data sets, trained the model on the training set and tested on the test set. However, for testing on the **validation** set we will perform the steps

individually to eliminate any concerns that the **validation** set may have been used to train the model when inputted into a function that both trains and tests the model.

First we train our model by calculating bias terms using the optimal value of λ already calculated for Model 5 and applying to the **edx** dataset.

```
# update mu for full edx set
mu <- mean(edx$rating)

# calculate bias terms using optimal value of lambda and the edx dataset
b_i <- calculate_b_i(edx, best$lambda)
b_u <- calculate_b_u(edx, best$lambda, b_i)
b_uI <- calculate_b_uI(edx, best$lambda, b_i, b_u, movie_groups)
b_iU <- calculate_b_iU(edx, best$lambda, b_i, b_u, b_uI, movie_groups, user_groups)
```

And then we use the bias terms calculated in the previous step to predict ratings using the **validation** set.

```
# predict ratings for validation set using the bias terms calculated with the edx set
predicted_values <- validation %>%
  left_join(b_i, by="movieId") %>%
  left_join(b_u, by="userId") %>%
  left_join(movie_groups, by = "movieId") %>%
  left_join(user_groups, by = "userId") %>%
  mutate(movie_group = ifelse(is.na(movie_group), 99, movie_group),
         user_group = ifelse(is.na(user_group), 99, user_group)) %>%
  left_join(b_uI, by = c("userId", "movie_group")) %>%
  left_join(b_iU, by = c("movieId", "user_group")) %>%
  mutate(b_uI = ifelse(is.na(b_uI), 0, b_uI),
         b_iU = ifelse(is.na(b_iU), 0, b_iU),
         pred = mu + b_i + b_u + b_uI + b_iU) %>%
  pull(pred)
```

We now check the RMSE of our final model.

```
final_model_rmse <- RMSE(predicted_values, validation$rating)
final_model_rmse
```

```
## [1] 0.8299894
```

We appear to have significantly improved the accuracy of our predictions!

3. Model results and performance

To ensure we have indeed improved prediction accuracy, we will compare the RMSE of our predictions with the RMSE of predictions made using Model 2, the regularised movie + user effects model outlined in the textbook. This calculation won't take as long as our final model, but will still be a minute or two.

```
optimal <- optimise_lambda(predict_model_2, train_set, test_set)
textbook_predicted_values <- predict_model_2(edx, validation, optimal$lambda)
textbook_rmse <- RMSE(textbook_predicted_values, validation$rating)
textbook_rmse
```

```
## [1] 0.8648177
```

Our final model RMSE of 0.8299894 has significantly improved upon the textbook model RMSE of 0.8648177.

On top of the improved RMSE, our model has also substantially reduced the calculation time compared to the approach outlined in the textbook, by introducing an algorithmic approach to selecting λ. Coding has also been tidied up pretty substantially by creating functions to perform each step of the model calculations - without doing this there would have been many more lines of code, as well as a whole lot of copy-pasting and dealing with the increased risk of typos that would accompany needing to copy-paste each step of the algorithm multiple times.

There are certainly improvements that could be made to the model outlined in this paper. In particular, we could not find an adequate way to account for time and genre effects. As the winning entries to the Netflix Challenge linked to in the coursework did account for these effects in some way, it is reasonable to conclude that there are methods not considered here that would improve model accuracy by considering these effects.

Although clustering movies and users into groups did improve model accuracy, I will not pretend that I am an expert on the k-clustering algorithm. It is certainly possible that there are different ways to filter the data before using the algorithm, or different algorithm argument values, that would improve the accuracy of the predictions made (although these may blow out calculation times!). Changing the number of groups used could also improve accuracy - this was not considered here due to the length of time required to perform the k-clustering algorithm, but somebody with more time and/or computing power could definitely experiment and find an optimal number of groups to use.

Despite these potential improvements, we are still satisfied with the overall performance of the model, as we have managed to significantly improve upon the original predictions with a relatively efficient prediction function.

4. Summary and conclusions

We have built upon the regularised movie + user effects model outlined in the course material, creating a regularised movie + user + movie group + user group effects model that improves the RMSE of predicted ratings from 0.8648177 to 0.8299894.

Our model adds to the approach described in the textbook by clustering movies/users into groups of similar movies/users and calculates bias terms to account for:

- individual users' bias for movies from each movie group ($b_{u,I}$); and
- the bias users from a particular user group have for individual movies ($b_{i,U}$).

With the bias terms above included, our model now has the form:

$Y_{u,i} = \mu + b_i + b_u + b_{u,I} + b_{i,U} + e_{u,i}$

The effect of the different model steps on the RMSE of predictions made using the **validation** dataset is shown below.

```
# get optimal lambda values for each model step
optimal1 <- optimise_lambda(predict_model_1, train_set, test_set)
# model 2 RMSE already calculated
# model 3 was not included in our final model
optimal4 <- optimise_lambda(predict_model_4, train_set, test_set)
# model 5 RMSE already calculated

# make predictions using optimal lambdas on validation set
predict1 <- predict_model_1(edx, validation, optimal1$lambda)
```

```r
predict4 <- predict_model_4(edx, validation, optimal4$lambda)

# calculate RMSEs
rmse0 <- RMSE(rep(mu, length(validation$rating)), validation$rating) # naive Bayesian model
rmse1 <- RMSE(predict1, validation$rating)
rmse2 <- textbook_rmse # already calculated
rmse4 <- RMSE(predict4, validation$rating)
rmse5 <- final_model_rmse

tibble(method=c("Just the average",
                "Regularised Movie Effects Model",
                "Regularised Movie + User Effects Model",
                "Regularised Movie + User + Movie Group Effects Model",
                "Regularised Movie + User + Movie Group + User Group Effects Model"),
                                RMSE = c(rmse0, rmse1, rmse2, rmse4, rmse5))
```

```
## # A tibble: 5 x 2
##   method                                                          RMSE
##   <chr>                                                          <dbl>
## 1 Just the average                                               1.06
## 2 Regularised Movie Effects Model                                0.944
## 3 Regularised Movie + User Effects Model                         0.865
## 4 Regularised Movie + User + Movie Group Effects Model           0.838
## 5 Regularised Movie + User + Movie Group + User Group Effects Model 0.830
```

We see that every additional step added to the model has increased the accuracy of its predictions.

Functions were created to train and test the model, with a view to tidying up the code base and reducing model run time by selecting the λ term used for regularisation in a more efficient manner.

We considered the effect of time and genre effects on ratings but were unable to improve our model's performance using these variables. It is also likely that the model's performance could be improved by further fine-tuning the parameters used, or by using other methods or algorithms not considered here.