***

HARVARD DATA SCIENCE WITH R PROFESSIONAL CERTIFICATE PROGRAM

CAPSTONE PROJECT: MOVIELENS PROJECT

AUTHOR: ALBERTUS ERWIN SUSANTO 阎余文 (ID)

***

A. INTRODUCTION

This capstone project is the final part of Harvard Data Science with R Professional Certificate program. It aims to build a recommendation system that uses a set of variables to predict public rating of a movie. The dataset, called MovieLens, has been prepared by Prof. Rafael Irizarry, the course instructor. He took a small subset from a Netflix movie rating database with over 20 million ratings over 27,000 movies rated by more than 138,000 users, which was generated by GroupLens research lab. That subset is available in the dslabs package and is the foundation of this project.

Regarding the recommendation model building process, it starts with an initial data wrangling and descriptive statistics of the data. Once it is done, we would divide the dataset into a train set and a test set. We then build the model from the train set and later test it using the test set, choosing some parameters that have predicting value. We then compare different models to see how well it works, using the root mean squared error (RMSE) as our loss function, which can be interpreted as similar to standard deviation.

After a series of cross-validation, we then will use the optimal model to do prediction using the validation set or as it is called “final hold-out test”. We will then make a conclusion at the end.

For better navigation through this report, below is the contents list. Note that the numbering in this report and the numbering in the code is done just the exactly the same for better cross-checking between the two.

CONTENTS LIST

B. DATA PREPARATION PROCESS

Data cleaning process, or also called data wrangling is basically the first process to do with data analysis. We prepare our data before we build our model. The first step involves setting up the environment by ensuring that necessary packages are installed and loaded, particularly `tidyverse` and `caret`. These packages provide essential tools for data manipulation and partitioning, which are key for the subsequent steps.

Next, the dataset is downloaded and unzipped if it doesn't already exist locally. The dataset used here is the **MovieLens 10M dataset**, which consists of ratings and movie information. Once downloaded, the ratings data is loaded and split into its respective columns (userId, movieId, rating, timestamp) using the `str_split` function. This step essentially reads the raw ratings data and converts it into a structured data frame. The data is then transformed to ensure the appropriate types for each column: userId and movieId are integers, rating is numeric, and timestamp is an integer.

```r
# Libraries
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")

library(tidyverse)
library(caret)

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

options(timeout = 120)

dl <- "ml-10M100K.zip"
if(!file.exists(dl))
  download.file("https://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

ratings_file <- "ml-10M100K/ratings.dat"
if(!file.exists(ratings_file))
  unzip(dl, ratings_file)

movies_file <- "ml-10M100K/movies.dat"
if(!file.exists(movies_file))
  unzip(dl, movies_file)

ratings <- as.data.frame(str_split(read_lines(ratings_file), fixed("::"), simplify = TRUE),
                         stringsAsFactors = FALSE)
head(ratings)
colnames(ratings) <- c("userId", "movieId", "rating", "timestamp")
ratings <- ratings %>%
  mutate(userId = as.integer(userId),
         movieId = as.integer(movieId),
         rating = as.numeric(rating),
         timestamp = as.integer(timestamp))

movies <- as.data.frame(str_split(read_lines(movies_file), fixed("::"), simplify = TRUE),
                        stringsAsFactors = FALSE)
colnames(movies) <- c("movieId", "title", "genres")
movies <- movies %>%
  mutate(movieId = as.integer(movieId))
```

After cleaning both the ratings and movies data, the two are combined into a single dataset called `movielens` using a left join on the `movieId`. This step merges the rating information with the respective movie titles and genres.

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

With the complete `movielens` dataset ready, the next step is to partition it into training and test sets. A random 10% of the data is set aside as the test set, using `createDataPartition` to maintain the distribution of ratings.

```
# Final hold-out test set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding") # if using R 3.6 or later

# set.seed(1) # if using R 3.5 or earlier
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]
```

To avoid issues where the final hold-out test set contains users or movies not present in the training set, the `semi_join` function is used to filter the set. This step ensures that only the users and movies present in the training data (edx) are retained in the final hold-out test set (`final_holdout_test`). This is critical because if the model encounters a user or movie in the test set that it hasn't seen in the training set, it won't be able to make predictions. We are going to predict the user’s rating of a certain movie in the test set, but it should appear in the training set first, or we are going to predict the movie’s rating in the test set, but it should appear in the training set first.

```
# Make sure userId and movieId in final hold-out test set are also in edx set
final_holdout_test <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")
```

Since some users or movies in temp may be removed if they don't have a corresponding match in edx, we will pull those back to the edx set.

```
# Add rows removed from final hold-out test set back into edx set
removed <- anti_join(temp, final_holdout_test)
edx <- rbind(edx, removed)

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

C. QUIZ AS ENTRY TO THE DATASET

Here is the quiz and the code to get to the answers of each quiz questions. It is provided here since this quiz will help us better understanding the data set first before delving deeper into its patterns and creating our model. Please note that some of the code and the answer to these quizzes will be described again in the data exploration and visualization part.

1. How many rows and columns are there in the edx dataset?

```
# Number of rows:
nrow(edx)

# Number of columns:
ncol(edx)
colnames(edx)
sapply(edx, class)
```

2. How many zeros were given as ratings in the edx dataset?

```
# How many threes were given as ratings in the edx dataset?
sum(edx$rating == 0)

# How many threes were given as ratings in the edx dataset?
sum(edx$rating == 3)
```

3. How many different movies are in the edx dataset?

```
# Different Movies
edx %>% distinct(movieId) %>% count()
```

4. How many different users are in the edx dataset?

```
# Different Users
edx %>% distinct(userId) %>% count()
```

5. How many movie ratings are in each of the following genres in the edx dataset?

```
# Drama
edx %>% filter(str_detect(genres, "Drama")) %>% count()
# Comedy
edx %>% filter(str_detect(genres, "Comedy")) %>% count()
# Thriller
edx %>% filter(str_detect(genres, "Thriller")) %>% count()
# Romance
edx %>% filter(str_detect(genres, "Romance")) %>% count()
```

6. Which movie has the greatest number of ratings?

```
# Movie with Greatest Number of Ratings
edx %>%
  group_by(movieId) %>%
  summarize(title = first(title), number_ratings = n()) %>%
  arrange(desc(number_ratings)) %>%
   slice(1)

# To have a full list:
# install.packages("openxlsx") # Install the openxlsx package if you don't have it
library(openxlsx)
movie_ratings_list <- edx %>%
  group_by(movieId) %>%
  summarize(title = first(title), number_ratings = n()) %>%
  mutate(rank = rank(desc(number_ratings))) %>%
  arrange(desc(number_ratings))
write.xlsx(movie_ratings_list,
           file = "movie_ratings.xlsx",
           sheetName = "Ratings Summary",
           rowNames = FALSE)
```

7. What are the five most given ratings in order from most to least?

```
# The Five Most Given Ratings in Descending Order
edx %>%
  group_by(rating) %>%
  summarize(number_of_instances = n()) %>%
  arrange(desc(number_of_instances))
```

8. True or False: In general, half star ratings are less common than whole star ratings (e.g., there are fewer ratings of 3.5 than there are ratings of 3 or 4, etc.).

```
# Half Star & Whole Star Count
rating_counts <- edx %>%
  mutate(rating_category = ifelse(rating %% 1 == 0, "whole", "half")) %>%
  group_by(rating_category) %>%
  summarize(count = n())
rating_counts
```

D. DATA EXPLORATION AND VISUALIZATION

We now want to make some data exploration first to better understand the dataset. The very first thing we need to to is to get to know of the data.

"How many rows does it contain?
How many columns and what are their names?"

```
# Number of rows:
nrow(edx)

# Number of columns:
ncol(edx)
colnames(edx)
sapply(edx, class)
```

We find that there are 9,000,055 rows and 6 columns. Below are details of the columns:

Column Names	*Data Type*	*Information*	*Sample*
userId	integer	Id of each user	1
movieId	integer	Id of each movie	122
rating	numeric	the rating given of a certain user to a certain movie (1-5)	5
timestamp	integer	the time the rating was given	838985046
title	character	the title of the movie rated	Boomerang (1992)
genres	character	the genre of the move rated	Comedy|Romance

"How many different movies and users are in the edx dataset?"

```
# How many different movies are in the edx dataset?
edx %>% distinct(movieId) %>% count()

# How many different users are in the edx dataset?
edx %>% distinct(userId) %>% count()1
```

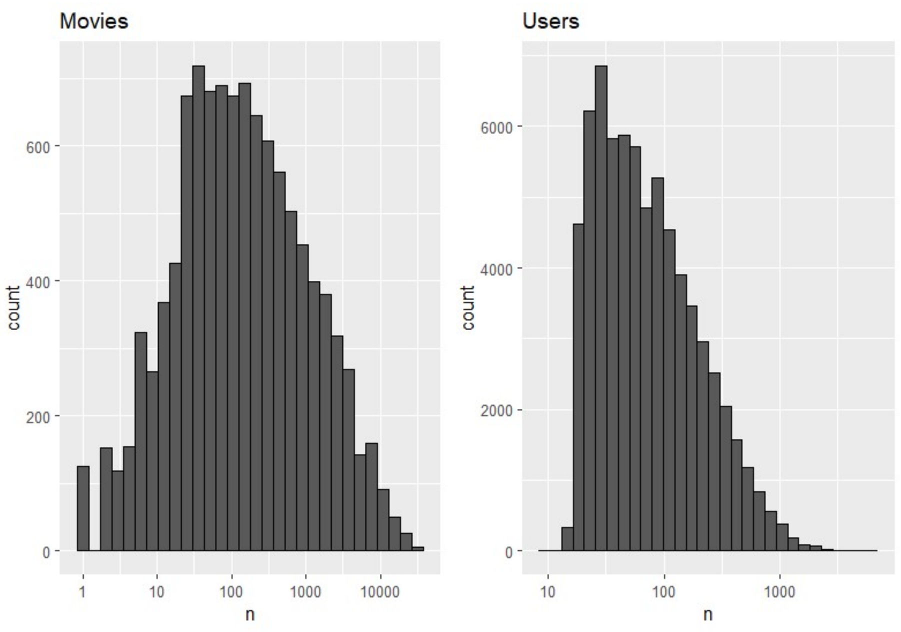
We find there are 10,677 different movies and 69,878 different users in the dataset.

"How many movies and users are matched?"

```
# Library for Showing Grids of Graphs
library(gridExtra)

# Movie's Histogram
p1 <- edx %>%
  count(movieId) %>%
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  ggtitle("Movies")
```

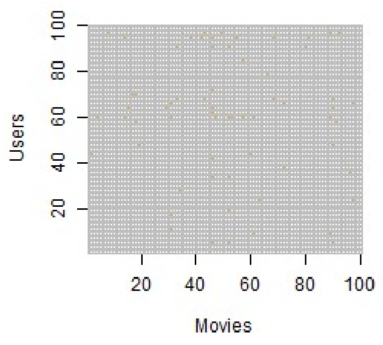
```
# User's Histogram
p2 <- edx %>%
  count(userId) %>%
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  ggtitle("Users")

# Show the two Histograms
grid.arrange(p1, p2, ncol = 2)
```

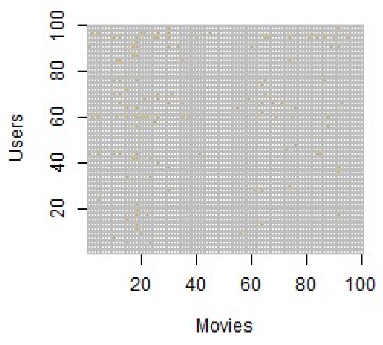
We see that from the histogram charts below, some movies are rated more than the other movies and some users rate more movies than the other users do.

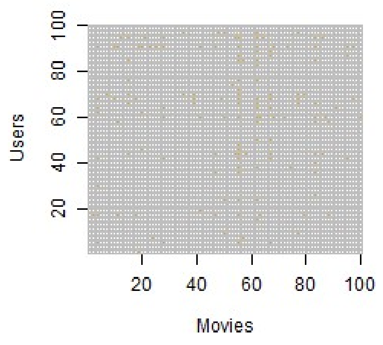


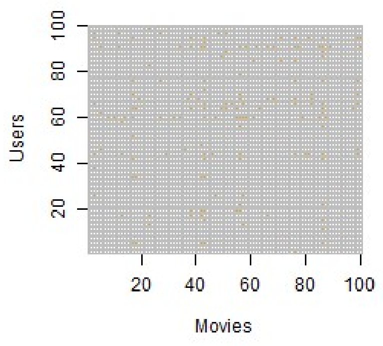
Now we want to know better the data as to how many movies are rated by how many users.

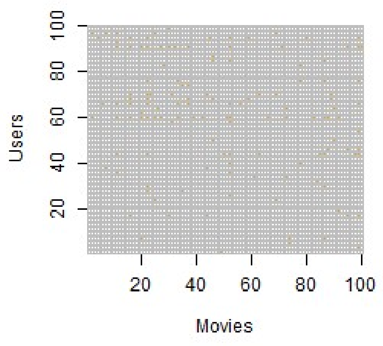
```
par(mfrow = c(2, 3))
users <- sample(unique(edx$userId), 100)

r1 <- edx %>% filter(userId %in% users) %>%
  dplyr::select(userId, movieId, rating) %>%
  mutate(rating = 1) %>%
  pivot_wider(names_from = movieId, values_from = rating) %>%
  (\(mat) mat[, sample(ncol(mat), 100)])()%>%
  as.matrix() %>%
  t() %>%
  image(1:100, 1:100,. , xlab="Movies", ylab="Users")
abline(h=0:100+0.5, v=0:100+0.5, col = "grey")

r2 <- edx %>% filter(userId %in% users) %>%
  dplyr::select(userId, movieId, rating) %>%
  mutate(rating = 2) %>%
  pivot_wider(names_from = movieId, values_from = rating) %>%
  (\(mat) mat[, sample(ncol(mat), 100)])()%>%
  as.matrix() %>%
  t() %>%
  image(1:100, 1:100,. , xlab="Movies", ylab="Users")
abline(h=0:100+0.5, v=0:100+0.5, col = "grey")

r3 <- edx %>% filter(userId %in% users) %>%
  dplyr::select(userId, movieId, rating) %>%
  mutate(rating = 3) %>%
  pivot_wider(names_from = movieId, values_from = rating) %>%
  (\(mat) mat[, sample(ncol(mat), 100)])()%>%
  as.matrix() %>%
  t() %>%
  image(1:100, 1:100,. , xlab="Movies", ylab="Users")
abline(h=0:100+0.5, v=0:100+0.5, col = "grey")

r4 <- edx %>% filter(userId %in% users) %>%
  dplyr::select(userId, movieId, rating) %>%
  mutate(rating = 4) %>%
  pivot_wider(names_from = movieId, values_from = rating) %>%
  (\(mat) mat[, sample(ncol(mat), 100)])()%>%
  as.matrix() %>%
  t() %>%
  image(1:100, 1:100,. , xlab="Movies", ylab="Users")
  abline(h=0:100+0.5, v=0:100+0.5, col = "grey")

r5 <- edx %>% filter(userId %in% users) %>%
  dplyr::select(userId, movieId, rating) %>%
  mutate(rating = 5) %>%
  pivot_wider(names_from = movieId, values_from = rating) %>%
  (\(mat) mat[, sample(ncol(mat), 100)])()%>%
  as.matrix() %>%
  t() %>%
  image(1:100, 1:100,. , xlab="Movies", ylab="Users")
  abline(h=0:100+0.5, v=0:100+0.5, col = "grey")

  par(mfrow = c(1, 1))
```











The yellow dots show movie and user matches. Our recommendation system is equal to predicting the values of the grey area, which are ratings of are movies that are not rated or users that do not rate certain movies.

What are the top ten most rated movies?

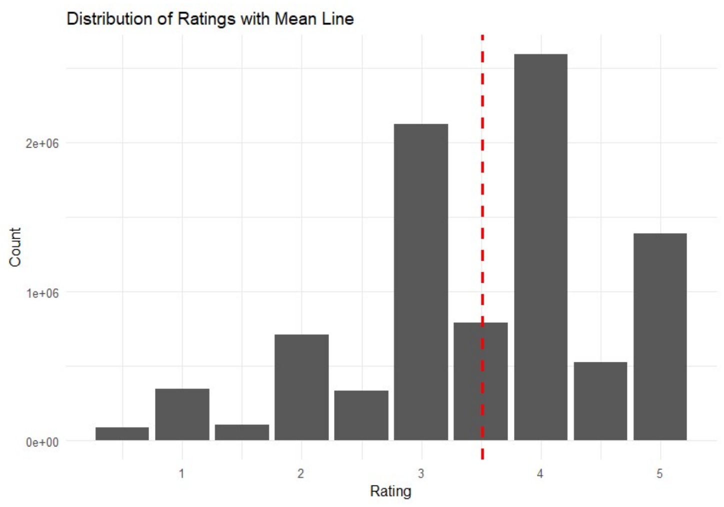
```
# Add openxlsx Library to Transfer the Result to Excel
library(openxlsx)

# Creating the List
movie_ratings_list <- edx %>%
  group_by(movieId) %>%
  summarize(title = first(title), number_ratings = n()) %>%
  mutate(rank = rank(desc(number_ratings))) %>%
  arrange(desc(number_ratings))

# Tranfering to Excel
write.xlsx(movie_ratings_list,
           file = "movie_ratings.xlsx",
           sheetName = "Ratings Summary",
           rowNames = FALSE)
```

rank	movieId	title	number_ratings
1	296	Pulp Fiction (1994)	31362
2	356	Forrest Gump (1994)	31079
3	593	Silence of the Lambs, The (1991)	30382
4	480	Jurassic Park (1993)	29360
5	318	Shawshank Redemption, The (1994)	28015
6	110	Braveheart (1995)	26212
7	457	Fugitive, The (1993)	25998
8	589	Terminator 2: Judgment Day (1991)	25984
9	260	Star Wars: Episode IV - A New Hope (1977)	25672
10	150	Apollo 13 (1995)	24284

"What is the rating mostly given by people?"

Now we want to check each variable or regressor or predictors that may have predicting values. If a variable has quite a good diversity of value and it has somehow correlation with the rating, we might than see some predicting value from that variable. We start from the mean value of rating and its distribution.

```
# Checking the mean and distribution of rating value
rating_distribution <- edx %>%    # Calculate the distribution of ratings
  group_by(rating) %>%
  count()

mean_rating <- mean(edx$rating) # Calculate the mean of the ratings

rating_distribution %>%  # Plot the distribution using ggplot2 and add a vertical line for the mean
  ggplot(aes(x = rating, y = n)) +
  geom_bar(stat = "identity") +       # Use stat = "identity" because counts are pre-calculated
  geom_vline(aes(xintercept = mean_rating), color = "red", linetype = "dashed", size = 1) +  # Add mean line
  labs(x = "Rating", y = "Count", title = "Distribution of Ratings with Mean Line") +  # Add labels
  theme_minimal()                     # Apply a clean theme
```

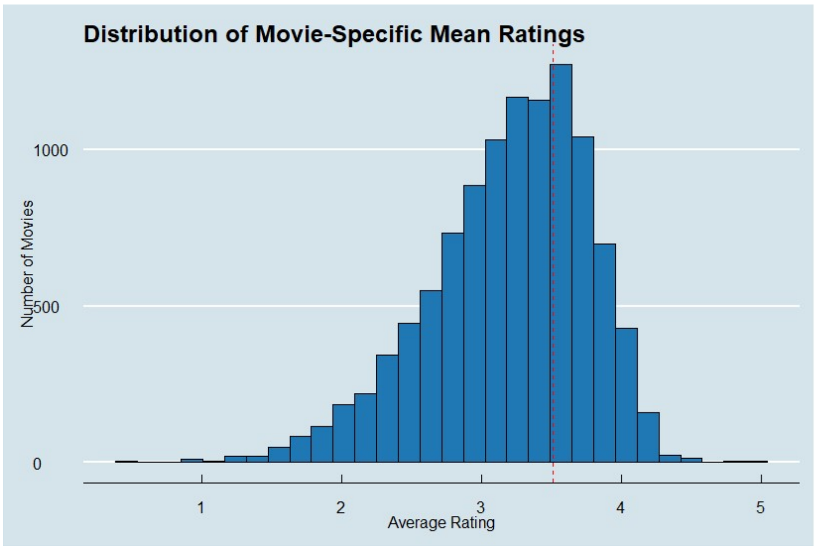


The graph shows that the mean of rating given is 3.5 while the mode is 4. Mean of rating will be our first regressor or predictor in the recommendation model.

"Is there an obvious difference of rating among different movies,
indicating a movie-specific rating mean (movie-effect),
as well as user-specific rating mean (user-effect)?"

We are now going to check if some movies tend to have higher or lower rating and some users have tendency to give higher or lower rating on movies. We start with the movie-specific rating mean.

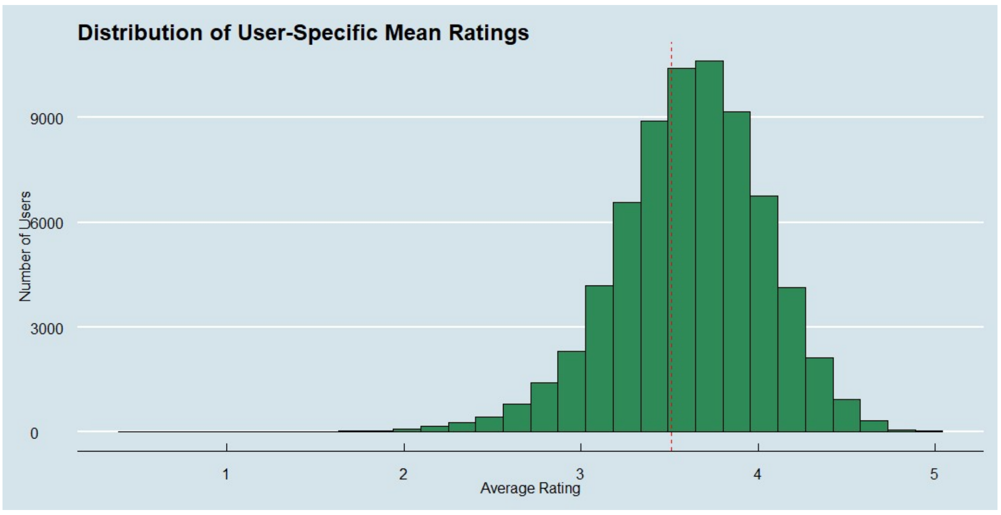
```
# Overall mean rating
mean_rating

# Movie-specific mean rating
movie_avg_ratings <- edx %>%
  group_by(movieId) %>%
  summarize(movie_mean = mean(rating))

# Compare with overall mean
head(movie_avg_ratings)

# Plot the distribution of movie-specific means
ggplot(movie_avg_ratings, aes(x = movie_mean)) +
  geom_histogram(bins = 30, color = "black", fill = "#1f77b4") +
  geom_vline(xintercept = mean_rating, linetype = "dashed", color = "red") +
  labs(title = "Distribution of Movie-Specific Mean Ratings", x = "Average Rating", y = "Number of Movies") +
   theme_economist()
```



From the graph above it is obvious that some movies are indeed tend to have higher ratings compared to the mean of the entire set, while some have lower rating points. The red dotted line in the graph above shows the mean of the entire set.

What about each user's rating? Any tendency of giving higher or lower ratings?

```
# User-specific mean rating
user_avg_ratings <- edx %>%
  group_by(userId) %>%
  summarize(user_mean = mean(rating))

# Compare with overall mean
head(user_avg_ratings)

# Plot the distribution of user-specific means
ggplot(user_avg_ratings, aes(x = user_mean)) +
  geom_histogram(bins = 30, color = "black", fill = "#2E8B57") +
  geom_vline(xintercept = mean_rating, linetype = "dashed", color = "red") +
  labs(title = "Distribution of User-Specific Mean Ratings", x = "Average Rating", y = "Number of       Users") +
   theme_economist()
```



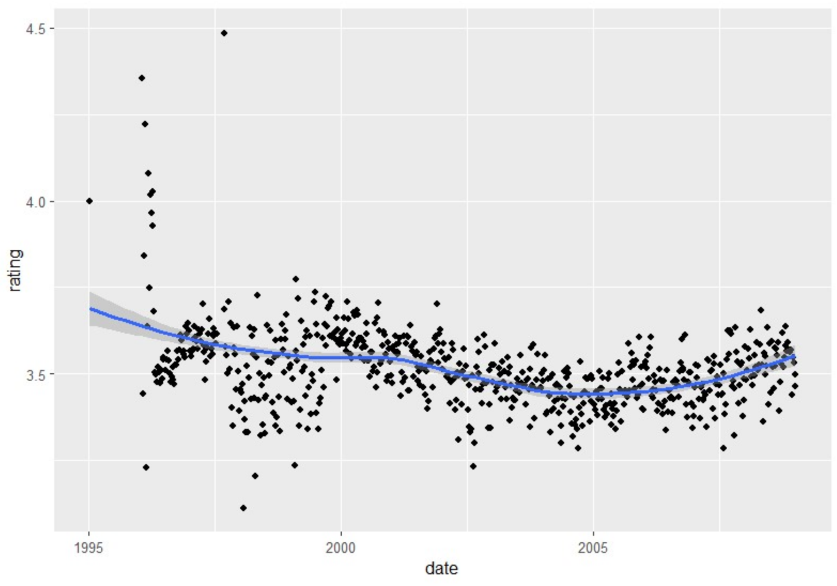
Similar to the previous graph, the red dotted line shows the mean of ratings given by all users in the entire set. The graph shows that mostly people give higher rating than the mean.

This shows that there is indeed movie-effect and user-effect that could be taken into the model.

"Is there a trend/fluctuation in the rating given from time-to-time?"

Now we are going to check if there is any trend in the rating given from time-to-time.

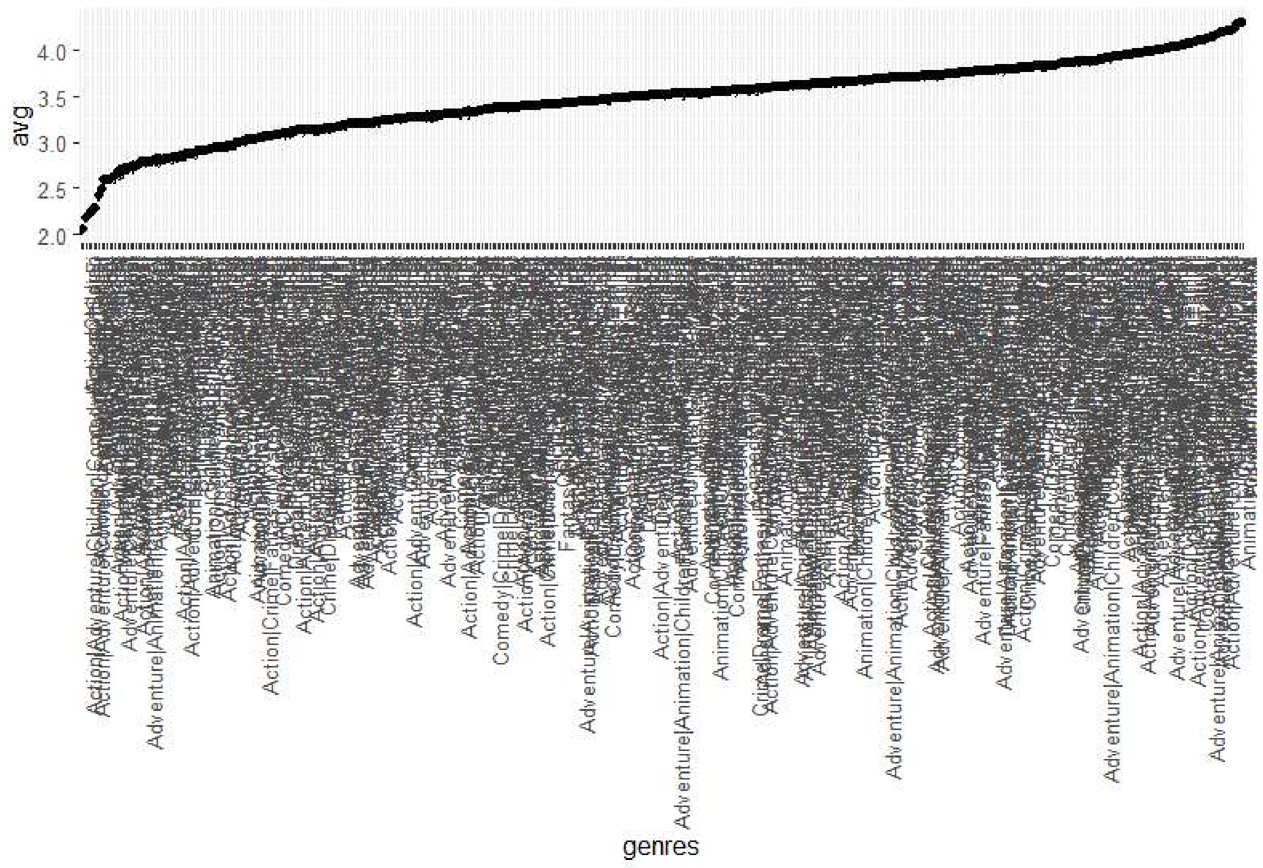
```
# Seeing if there's time effect
edx %>% mutate(date = round_date(date, unit = "week")) %>% # we first transform the format of time
  group_by(date) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(date, rating)) +
  geom_point() +
  geom_smooth()
```



Here we see that there's some evidence of a time effect in the plot, though it is not very strong. Hence time can be another valuable predictor of rating.

“Is there obvious difference of rating among different genres, indicating the effects of genre on movie rating?”

```
# Considering genre effect
genre_effect <- edx %>% group_by(genres) %>%
  summarize(n = n(), avg = mean(rating), se = sd(rating)/sqrt(n())) %>%
  filter(n >= 1000) %>%
  mutate(genres = reorder(genres, avg))

genre_effect %>%
  ggplot(aes(x = genres, y = avg, ymin = avg - 2*se, ymax = avg + 2*se)) +
  geom_point() +
  geom_errorbar() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))

best_rated_genre <- genre_effect %>% arrange(desc(avg)) %>% slice(1)  # Top-rated genre
best_rated_genre
least_rated_genre <- genre_effect %>% arrange(avg) %>% slice(1)       # Least-rated genre
least_rated_genre
```



We find that the best rated movies in average are of genre “Drama|Film-Noir|Romance”, with a total of 2989 movies, average rating of 4.30 and standard deviation of 0.0145. While the worst rated movies in average are of genre “Action|Children”, with total of 3922 movies, average of 2.04 in ratings, and a standard deviation of 0.0174.

The plot above shows that there is strong evidence of a genre effect. This means that genre might be a good predictor for rating.

E. BUILDING OUR MODEL

We will start building our prediction models, step-by-step. We will check the result of each model and compare their effectiveness in predicting the rating.

The first step to do that is by dividing our data set into train set and test set. We use 80% of the data set for the train set and the remaining 20% for the test set. Here’s the code:

```
# Partitioning the Data into Train and Test Set
test_index <- createDataPartition(y = edx$rating, times = 1,
                                  p = 0.2, list = FALSE)
train_set <- edx[-test_index,]
test_set <- edx[test_index,]
test_set <- test_set %>%
  semi_join(train_set, by = "movieId") %>%
   semi_join(train_set, by = "userId")
```

Our measure of effectiveness is the RMSE (root mean squared error). We use the code below to create the RMSE function:

```
# RMSE Function as the Evaluation Measure of Models
RMSE <- function(true_ratings, predicted_ratings){
    sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

1. Model 1: Predicting Using the Mean

Our first model is very simple. We will just predict the rating using the mean of the entire train set, which is indeed crude and naïve, but it should be better than guessing randomly between the scale 0-5.

Theoretically, this model assumes the same rating for all movies and users with all the differences among them explained by a random variation or an error term. The mathematical model is as follows:

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

- $Y_{u,i}$ = the value of rating of user u and movie i.
- μ = the mean of rating of all movies by al users.
- $e_{u,i}$ = the error term/difference in rating value of user u and movie i that occurs by random variation (it is independent and its distribution is centered at zero).

Here is our code:

```
# 1. First Model - Predicting Simply Using the Mean
# We count the mean:
mu_hat <- mean(train_set$rating)
mu_hat

# We see the RMSE of using mean in the test set:
model1_rmse <- RMSE(test_set$rating, mu_hat)
model1_rmse
```

To have a comparison to see if using the mean is better than anything, we try using the median of the rating scale, which is 2.5 to predict the rating.

```
# We compare the use of mean in predicting the result
# with using the median of the scale (not the median of the ratings given)
# which is 2.5
median_scale_prediction <- rep(2.5, nrow(test_set))
model0_rmse <- RMSE(test_set$rating, median_scale_prediction)
```

Then we check the result of the two models, one using the mean of the ratings, another using the median of the scale.

```
# We input it into our collection of results of models RMSEs
rmse_results <- data_frame(method = "Constant Value 2.5", RMSE = model0_rmse)
rmse_results <- bind_rows(rmse_results,
                          data_frame(method = "Using the Data Set Mean", RMSE = model1_rmse))
rmse_results

# We export the result to an excel sheet:
# install.packages("writexl") --- in case haven't been installed
library(writexl)
```

```
write_xlsx(rmse_results, "rmse_results.xlsx")
```

The result is:

Method	RMSE
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023

Well, that shows that using the Data Set Mean model (Model 1) is still better than using only the constant value of 2.5. The smaller the RMSE is, the closer we are to the real value in the test set, which means we make better prediction of the rating. We should do better than that though. Proceed!

2. Model 2: Predicting Using the Set Mean and Mean Difference of Each Movie's Rating

Our second model will take into account the fact that some movies are generally rated higher than other movies (movie-effect). By including this consistent difference into our equation, we would get a more accurate prediction. Here is the updated model, with term b_i representing the average rating for movie i:

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

- $Y_{u,i}$ = the value of rating of user u and movie i.
- μ = the mean of rating of all movies by al users.
- $e_{u,i}$ = the error term/difference in rating value of user u and movie i that occurs by random variation (it is independent and its distribution is centered at zero).
- b_i = the average difference of rating of movie i

Here's our code:

```
# 2. Second Model - Predicting Using the Movie Effect---------------------------

# The course in Machine Learning introduced us to this code below,
# with a caveat that it will take longer time. So we'll just leave it.
# fit <- lm(rating ~ as.factor(userId), data = movielens)

# We use the simpler way of predicting using movie effect
mu <- mean(train_set$rating)
movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu)) # here we count the mean of the difference of rating mean of movies with the mean of the
                                                  entire train set ratings

# Let's check the value of b_i
movie_avgs
movie_avgs %>% qplot(b_i, geom ="histogram", bins = 30, data = ., color = I("black"))

# Input the b_i into the test_set
test_set_with_bi <- test_set %>%
  left_join(movie_avgs, by = 'movieId')

# We predict the rating in the test_set
predicted_ratings <- mu + test_set_with_bi$b_i

# Evaluate the efficacy of this model
model2_rmse <- RMSE(predicted_ratings, test_set$rating)

# We input it into our collection of results of models RMSEs
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method="Movie Effect Model",
                                     RMSE = model2_rmse ))


# We export the result to an excel sheet:
write_xlsx(rmse_results, "rmse_results.xlsx")
```

The result shows that adding the movie effect (b_i) into our model enables us to make better prediction, with lower RMSE.

Method	**RMSE**
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222

3. Model 3: Predicting Using the Set Mean, Movie Effect, and User Effect

Our third model will further add the user effect into our calculation. User effect takes into account the fact that some people tend to give a higher rating while some people give a lower rating for any kinds of movies.

Note that our third model would be build on top of the second model that already takes into account the movie effect, not separately.

In mathematical form, our model is like this:

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

- $Y_{u,i}$ = the value of rating of user u and movie i.
- μ = the mean of rating of all movies by al users.
- $e_{u,i}$ = the error term/difference in rating value of user u and movie i that occurs by random variation (it is independent and its distribution is centered at zero).
- b_i = the average difference of rating of movie i
- b_u = the average difference of rating given by user u

Here is our code:

```
# 3. Third Model - Predicting Using the User Effect + Movie Effect -------------

# Again, the course in Machine Learning introduced us to this code below,
# with a caveat that it will take longer time. So we'll just leave it.
# lm(rating ~ as.factor(movieId) + as.factor(userId))

# We use the simpler way of calculating the user effect (after deducing the movie effect)
user_avgs <- train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i)) # note that we take away the movie effect to have a                more precise
                                                         calculation of the user effect.

# Input the b_u into the test_set
# note that in our third model, we also take into our prediction the movie effect, so we also include it
test_set_with_bi_bu <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId')

# We predict the rating in the test_set
predicted_ratings <- test_set_with_bi_bu %>%
  mutate(pred = mu + b_i + b_u) %>%
   pull(pred)

# We evaluate the model
model3_rmse <- RMSE(predicted_ratings, test_set$rating)

# We input into our rmse collection
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method="Movie + User Effects Model",
                                     RMSE = model3_rmse ))
```

```
# We export the result to an excel sheet:
write_xlsx(rmse_results, "rmse_results.xlsx")
```

The result shows that it is indeed better than just taking into account the movie effect model. See below:

Method	**RMSE**
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752

4. Model 4: Including Genre Effect

We see that some genres of movies are better rated than others. We are going to include that into our equation, considering the presence of genre effect.

$$Y_{u,i,g} = \mu + b_i + b_u + b_g + \epsilon_{u,i,g}$$

- $Y_{u,i}$ = the value of rating of user u and movie i.
- μ = the mean of rating of all movies by al users.
- $e_{u,i}$ = the error term/difference in rating value of user u and movie i that occurs by random variation (it is independent and its distribution is centered at zero).
- b_i = the average difference of rating of movie i
- b_u = the average difference of rating of user u
- $b_{g(i)}$ = the average difference of rating of genre g given movie i

Here's the code:

```
# 4. Model 4: Genre Effect -----------------------------------------------------
# Step 1: Calculate the genre effect (b_g) for combined genres
genre_avgs <- train_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  group_by(genres) %>%
  summarize(b_g = mean(rating - mu - b_i - b_u))

# Ensure the genre effect was calculated
head(genre_avgs)

# Checking the genre effect using histogram
genre_avgs %>% ggplot(aes(b_g)) +
  geom_histogram(binwidth = 0.01, fill = "blue", color = "black") +
  labs(title = "Distribution of Genre Effect (b_g)", x = "Genre Effect (b_g)", y = "Frequency")

# Step 2: Merge genre effect into test set
# Note: We need to separate multiple genres in the test set too
test_set_with_bi_bu_bg <- test_set %>%
  left_join(movie_avgs, by = "movieId") %>%
  left_join(user_avgs, by = "userId") %>%
  left_join(genre_avgs, by = "genres")

# Step 3: Predict ratings in the test set using movie, user, time, and genre effects
predicted_ratings <- test_set_with_bi_bu_bg %>%
  mutate(pred = mu + b_i + b_u + b_g) %>%
  pull(pred)

# Step 4: Evaluate the RMSE for Model 5 (Movie + User + Genre Effects Model)
model4_rmse <- RMSE(predicted_ratings, test_set$rating)


# Step 5: Append the Model 5 RMSE result to our collection
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = "Movie + User + Genre Effects",
                                     RMSE = model4_rmse))
```

```
rmse_results

# Step 6: Export the result to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")
```

The result shows an amelioration, though not very significant:

Method	RMSE
Constant Value 2.5	1.466059
Using the Data Set Mean	1.060704
Movie Effect Model	0.943714
Movie + User Effects Model	0.866162
Movie + User + Genre Effects	0.865824

5. Model 5: Including Time Effect

In the data exploration and visualization part we have found that there is a time-trend in ratings shown, though very small. We then would like to include that time effect into our equation, trying to get a better prediction of our movie rating.

The mathematical form of our model shows like this:

$$Y_{u,i,g} = \mu + b_i + b_u + b_g + f(d_{u,i}) + \epsilon_{u,i,g}$$

- $Y_{u,i}$ = the value of rating of user u and movie i.
- μ = the mean of rating of all movies by al users.
- $e_{u,i}$ = the error term/difference in rating value of user u and movie i that occurs by random variation (it is independent and its distribution is centered at zero).
- b_i = the average difference of rating of movie i
- b_u = the average difference of rating given by user u
- $b_{g(i)}$ = the average difference of rating of genre g
- *f* is a function of $\text{date}_{u,i}$

To implement our model, we will explore various methods for calculating the time effect. Our first approach involves using the Generalized Additive Model (GAM). Next, we will employ the KSmooth method, applying a box kernel with several bandwidth options. We are opting out of the normal kernel due to time constraints and the substantial memory demand it places on the system. Additionally, we anticipate that the time effect may ultimately have limited utility in improving our results relative to the computational cost. The third approach uses the LOESS method, with iterations over different spans.

5.1. Using GAM

The Generalized Additive Model (GAM) allows for a flexible approach to modelling the time effect, as it estimates smooth functions for each covariate independently. For our purposes, GAM will enable us to model non-linear trends in time by fitting a smooth spline over the data points. By incorporating a spline basis for the time variable, we can capture seasonal and temporal fluctuations, which might otherwise be overlooked with simpler linear models. This method is particularly helpful when we expect the relationship between time and the target variable to vary continuously. By tuning the smoothness parameter, we can control the trade-off between bias and variance in the model.

$$f(x) = \alpha + \sum_{j=1}^{p} f_j(x_j) + \epsilon$$

Above is the function of the GAM:

- f(x) is the fitted function.
- α is the intercept.
- $f_j(x_j)$ represents smooth functions applied to each predictor x_j.
- ϵ is the error term.

Here's our code:

```
# 5. Model 5 - Predicting Using the User + Movie + Genre + Time Effect ----------
# 5. Model 5.1: Using GAM -------------------------------------------------------
# Step 1: Add a week number to each rating in the train and test sets
train_set <- train_set %>%
  mutate(weekNum = (timestamp - min(timestamp)) %/% (7 * 24 * 60 * 60) + 1)

test_set <- test_set %>%
  mutate(weekNum = (timestamp - min(timestamp)) %/% (7 * 24 * 60 * 60) + 1)

# Step 2: Fit a smooth curve to the ratings as a function of time (weekNum)
fit <- mgcv::gam(rating ~ s(weekNum, bs = "cs"),
                 family = gaussian(), data = train_set) # apply smoothing

# Step 3: Evaluate the fitted curve for each week number
r <- seq(1, max(train_set$weekNum))
f_t <- mgcv::predict.gam(fit, data.frame(weekNum = r)) - mu
rm(fit)

# Step 4: Plot the fitted curve
ggplot(data.frame(weekNum = r, f_t), aes(weekNum, f_t)) +
  geom_line() +
  xlab('t, Week number') +
  ylab((r'[$f(t)$]'))

# Step 5: Recalculating the effects
# 5.1: Movie Effect
movie_effect_t <- train_set %>%
  mutate(f_t = f_t[weekNum]) %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu - f_t))

# 5.2: User Effect
user_effect_t <- train_set %>%
  left_join(movie_effect_t, by = "movieId") %>%
  mutate(f_t = f_t[weekNum]) %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i - f_t))

head(user_effect_t)


# 5.3: Genre Effect
genre_effect_t <- train_set %>%
  left_join(movie_effect_t, by = "movieId") %>%
  left_join(user_effect_t, by = "userId") %>%
  mutate(f_t = f_t[weekNum]) %>%
  group_by(genres) %>%
  summarize(b_g = mean(rating - mu - b_i - b_u - f_t))

# Step 6: Obtain predictions for the test set using the new model
predicted_ratings <- test_set %>%
  mutate(f_t = f_t[weekNum]) %>%
  left_join(movie_effect_t, by = "movieId") %>%
  left_join(user_effect_t, by = "userId") %>%
  left_join(genre_effect_t, by = "genres") %>%
  mutate(pred = mu + b_i + b_u + b_g + f_t) %>%
 pull(pred)

# Step 7: Compute RMSE for the new model
model_rmse <- RMSE(predicted_ratings, test_set$rating)

# Step 8: Append the RMSE result to the collection of RMSEs
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = "Movie + User + Genre + Time Effects",
                                     RMSE = model_rmse))

# Step 9: Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")
```

The result shows that this method is not the most effective way to predict the time effects for the recommendation model.

Method	RMSE
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752
Movie + User + Genre Effects	0.865412
Movie + User + Genre + Time Effects (GAM)	0.865955

5.2. Using Bin-Smoothing with Box Kernel

In bin-smoothing with a box kernel, the time data is divided into discrete bins, with the assumption that all observations within each bin share a similar time effect. This approach allows us to smooth the time effect by averaging the values within each bin. A box kernel, which gives equal weight to all data points within the bin, is straightforward to implement and less computationally demanding than more complex kernels, such as the normal kernel. While we're not using the normal kernel in this instance, it's worth noting that a normal kernel would weigh points closer to the centre of the bin more heavily, potentially giving a more refined result but at the cost of higher computation. We will test several bandwidths for the box kernel to balance the trade-off between smoothness and responsiveness to short-term fluctuations.

$$\hat{f}(x_0) = \frac{1}{N_0} \sum_{i \in A_0} Y_i$$

Even though we are not using the Normal Kernel but let me here provide a theoretical comparison with the box kernel method. A normal kernel would weight observations closer to the target point x more heavily, allowing for a smooth estimate over the data.

$$\hat{f}(x_0) = \sum_{i=1}^{N} w_0(x_i) Y_i$$

Here’s our code:

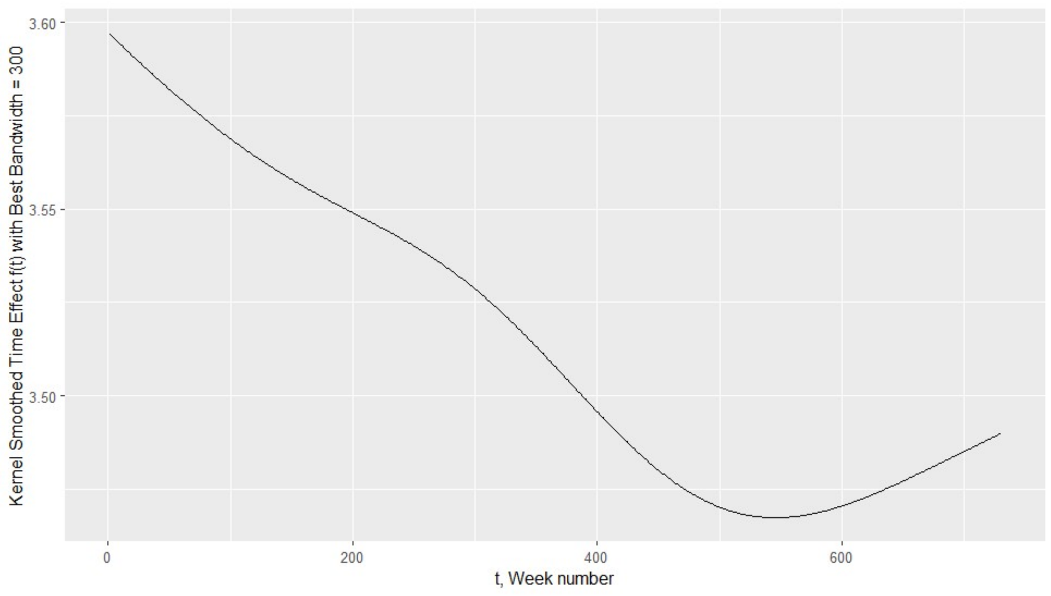
```
# 5. Model 5.2: Using KSmooth Bandwidth ----------------------------------------
# Step 1: Add a week number to each rating in the train and test sets
train_set_byweek <- train_set %>%
  mutate(weekNum = (timestamp - min(timestamp)) %/% (7 * 24 * 60 * 60) + 1) %>%
  group_by(weekNum) %>%
  summarize(y = mean(rating)) %>%
  arrange(weekNum)

test_set <- test_set %>%
  mutate(weekNum = (timestamp - min(timestamp)) %/% (7 * 24 * 60 * 60) + 1)

# Step 2: Iterate over possible bandwidth values for kernel smoothing
bandwidth_values <- seq(50, 500, by = 50)  # Adjust range and increment as necessary
best_rmse <- Inf
best_bandwidth <- NA

for (bandwidth in bandwidth_values) {
  # Apply kernel smoothing with the current bandwidth
  smoothed_train <- ksmooth(x = train_set_byweek$weekNum, y = train_set_byweek$y, kernel =     "normal", bandwidth =
   bandwidth)

  # Generate smoothed time effect for the train and test sets
  smoothed_train_fitted <- approx(smoothed_train$x, smoothed_train$y, xout =  train_set$weekNum)$y
  smoothed_test_fitted <- approx(smoothed_train$x, smoothed_train$y, xout = test_set$weekNum)$y

  # Calculate movie, user, and genre effects using the current time effect
  movie_effect_t <- train_set %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(movieId) %>%
    summarize(b_i = mean(rating - mu - f_t))

  user_effect_t <- train_set %>%
    left_join(movie_effect_t, by = "movieId") %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(userId) %>%
    summarize(b_u = mean(rating - mu - b_i - f_t))

  genre_effect_t <- train_set %>%
    left_join(movie_effect_t, by = "movieId") %>%
    left_join(user_effect_t, by = "userId") %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(genres) %>%
    summarize(b_g = mean(rating - mu - b_i - b_u - f_t))

  # Obtain predictions for the test set
  predicted_ratings <- test_set %>%
    mutate(f_t = smoothed_test_fitted) %>%
    left_join(movie_effect_t, by = "movieId") %>%
    left_join(user_effect_t, by = "userId") %>%
    left_join(genre_effect_t, by = "genres") %>%
    mutate(pred = mu + b_i + b_u + b_g + f_t) %>%
    pull(pred)

  # Calculate RMSE for the current bandwidth
  model_rmse <- RMSE(predicted_ratings, test_set$rating)
```

```
 # Check if this bandwidth yields a lower RMSE
 if (model_rmse < best_rmse) {
   best_rmse <- model_rmse
   best_bandwidth <- bandwidth
 }
}

# Step 3: Store the best result
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = paste("Movie + User + Genre + KSmooth Time Effects (bandwidth =", best_bandwidth,
")"),
                                     RMSE = best_rmse))

# Step 4: Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")

# Step 5: Plot the best kernel-smoothed curve using the optimal bandwidth
smoothed_train_best <- ksmooth(x = train_set_byweek$weekNum, y = train_set_byweek$y, kernel = "normal", bandwidth =
best_bandwidth)
ggplot(data.frame(weekNum = smoothed_train_best$x, f_t = smoothed_train_best$y),
       aes(weekNum, f_t)) +
 geom_line() +
 xlab('t, Week number') +
 ylab(paste('Kernel Smoothed Time Effect f(t) with Best Bandwidth =', best_bandwidth))
```

It produces a better result than the GAM method, though still relatively insignificant improvement.

Method	RMSE
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752
Movie + User + Genre Effects	0.865412
Movie + User + Genre + Time Effects (GAM)	0.865955
Movie + User + Genre + KSmooth Time Effects (box bandwidth = 300)	0.865001

The optimum box kernel bandwidth turns out to be 300 (which means 300 weeks).



5.3. Using LOESS

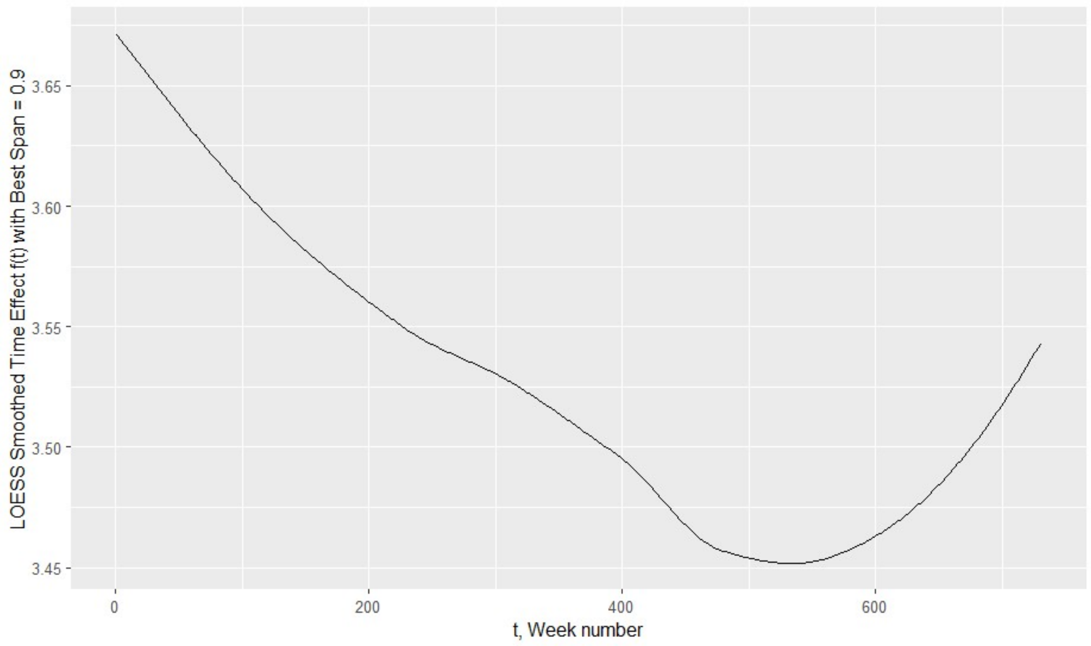
Instead of assuming that the function is approximately constant in a window, we assume the function is locally linear. We can consider larger window sizes with the linear assumption than with the constant assumption. Instead of the one-week window, we consider a larger one in which the trend is approximately linear. We're going to fitting line (degree = 1), not parabola (degree = 2) which is the default of loess.

```
# 5. Model 5.3: Time Effect Using LOESS -------------------------------------------
# Step 1: Add a week number to each rating in the train and test sets
train_set_byweek <- train_set %>%
  mutate(weekNum = (timestamp - min(timestamp)) %/% (7 * 24 * 60 * 60) + 1) %>%
  group_by(weekNum) %>%
  summarize(y = mean(rating)) %>%
  arrange(weekNum)

# Step 2: Iterate over possible span values for LOESS
span_values <- seq(0.1, 0.9, by = 0.1)  # Adjust range and increment if necessary
best_rmse <- Inf
best_span <- NA

for (span in span_values) {
  # Apply LOESS with the current span value
  loess_fit <- loess(y ~ weekNum, data = train_set_byweek, span = span)

  # Generate smoothed time effect for train and test sets
  smoothed_train_fitted <- predict(loess_fit, newdata = data.frame(weekNum = train_set$weekNum))
  smoothed_test_fitted <- predict(loess_fit, newdata = data.frame(weekNum = test_set$weekNum))

  # Calculate movie, user, and genre effects using the current time effect
  movie_effect_t <- train_set %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(movieId) %>%
    summarize(b_i = mean(rating - mu - f_t))

  user_effect_t <- train_set %>%
    left_join(movie_effect_t, by = "movieId") %>%
    mutate(f_t = smoothed_train_fitted) %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i - f_t))

 genre_effect_t <- train_set %>%
  left_join(movie_effect_t, by = "movieId") %>%
  left_join(user_effect_t, by = "userId") %>%
  mutate(f_t = smoothed_train_fitted) %>%
  group_by(genres) %>%
  summarize(b_g = mean(rating - mu - b_i - b_u - f_t))

 # Obtain predictions for the test set
 predicted_ratings <- test_set %>%
  mutate(f_t = smoothed_test_fitted) %>%
  left_join(movie_effect_t, by = "movieId") %>%
  left_join(user_effect_t, by = "userId") %>%
  left_join(genre_effect_t, by = "genres") %>%
  mutate(pred = mu + b_i + b_u + b_g + f_t) %>%
  pull(pred)

 # Calculate RMSE for the current span
 model_rmse <- RMSE(predicted_ratings, test_set$rating)

 # Check if this span yields a lower RMSE
 if (model_rmse < best_rmse) {
  best_rmse <- model_rmse
  best_span <- span
 }
}

# Step 3: Store the best result
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = paste("Movie + User + Genre + LOESS Time Effects (span =", best_span, ")"),
                           RMSE = best_rmse))

# Step 4: Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")

# Step 5: Plot the best LOESS-smoothed curve using the optimal span
loess_fit_best <- loess(y ~ weekNum, data = train_set_byweek, span = best_span)
ggplot(data.frame(weekNum = train_set_byweek$weekNum, f_t = predict(loess_fit_best, train_set_byweek)),
      aes(weekNum, f_t)) +
 geom_line() +
 xlab('t, Week number') +
  ylab(paste('LOESS Smoothed Time Effect f(t) with Best Span =', best_span))
```

The result shows that LOESS method, with span 0.9, produces more or less the same result with KSmooth method, with bandwidth of 300. The difference is very small as shown below:

Method	**RMSE**
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752
Movie + User + Genre Effects	0.865412
Movie + User + Genre + Time Effects (GAM)	0.865955
Movie + User + Genre + KSmooth Time Effects (box bandwidth = 300)	0.865001
Movie + User + Genre + LOESS Time Effects (span = 0.9)	0.865086

Here is our span range and their effects to RMSE:



6. Model 6: Using Regularization

Some movies in the dataset are rated by only a very small number of users—often just one. These ratings, often for obscure films, introduce significant uncertainty regarding their accuracy in reflecting the true quality of the movie or predicting how others might rate them. In terms of mathematical calculations, this can lead to large estimates of b, whether highly positive or negative. Such extreme estimates can increase the overall error in our model, as reflected by a higher RMSE (Root Mean Square Error), especially when these ratings are given too much weight.

Regularization offers a solution by penalizing these large estimates that stem from small sample sizes. By doing so, it adjusts for the uncertainty of the ratings, treating them with less weight compared to movies with more ratings. This method has similarities with the Bayesian approach, which shrinks predictions toward a more general mean, reducing the impact of outliers.

Our regularization equation is like this:

$$\text{Regularization: } \lambda \left(\sum_i b_i^2 + \sum_u b_u^2 + \sum_g b_g^2 \right)$$

Adding that regularization into our model produces:

$$Y_{u,i,g} = \mu + b_i + b_u + b_g + f(d_{u,i}) + \epsilon_{u,i,g} + \lambda \left(\sum_i b_i^2 + \sum_u b_u^2 + \sum_g b_g^2 \right)$$

Given the relatively insignificant result of the incorporation of time effect into our predictive model, in this regularization model we will try two kinds, first is without the time effect and the second is with the time effect (the mathematical formula above includes the time effect which is f($d_{u,i}$).

6.1. Model 6.1: Regularized Model Without Time Effect

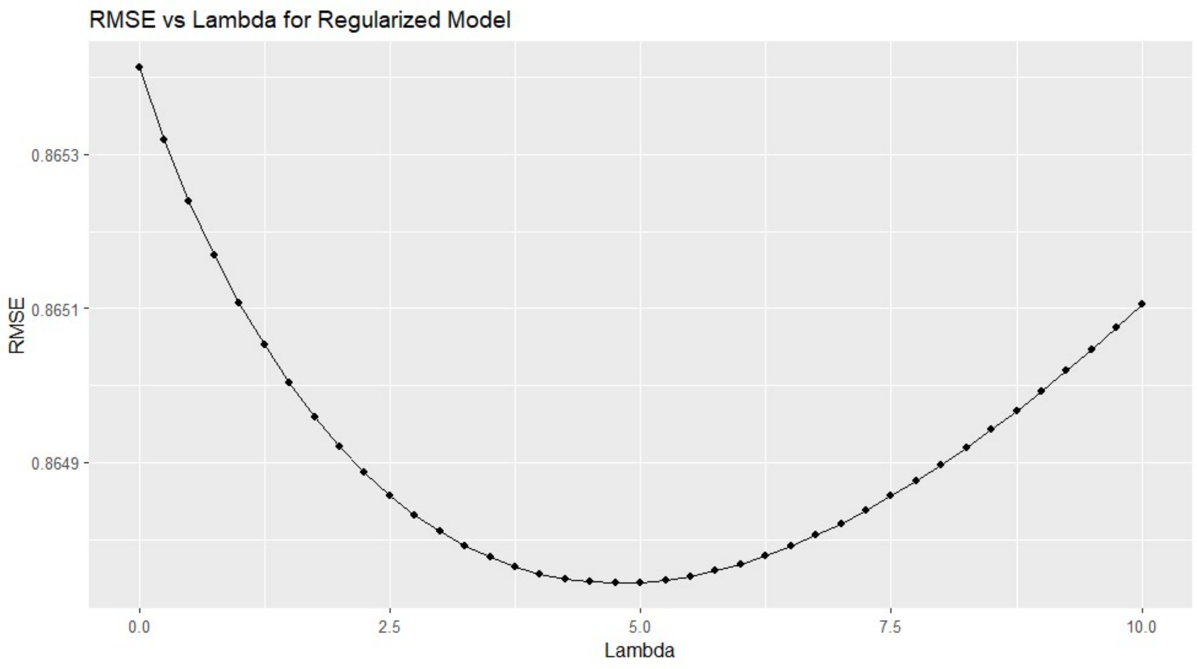
```
# 6. Model 6.1: Regularized Model Without Time Effect ----------------------------
# Step 1: Define lambda values for regularization
lambdas <- seq(0, 10, 0.25)

# Step 2: Apply regularization to the movie, user, and genre effects
regularization_rmse_results <- data.frame()

for (lambda in lambdas) {

  # Movie Effect with Regularization
  movie_avgs_reg <- train_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu) / (n() + lambda)) # Regularized

  # User Effect with Regularization
  user_avgs_reg <- train_set %>%
    left_join(movie_avgs_reg, by = "movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - mu - b_i) / (n() + lambda)) # Regularized
  # Genre Effect with Regularization
  genre_avgs_reg <- train_set %>%
    left_join(movie_avgs_reg, by = "movieId") %>%
    left_join(user_avgs_reg, by = "userId") %>%
    group_by(genres) %>%
    summarize(b_g = sum(rating - mu - b_i - b_u) / (n() + lambda)) # Regularized

  # Step 3: Predict ratings on the test set using the regularized movie, user, and genre effects
  predicted_ratings <- test_set %>%
    left_join(movie_avgs_reg, by = "movieId") %>%
    left_join(user_avgs_reg, by = "userId") %>%
    left_join(genre_avgs_reg, by = "genres") %>%
    mutate(pred = mu + b_i + b_u + b_g) %>%
    pull(pred)

  # Step 4: Calculate RMSE for each lambda
  model_rmse <- RMSE(predicted_ratings, test_set$rating)

  # Step 5: Store the RMSE results for each lambda in a separate data frame for regularization
  regularization_rmse_results <- bind_rows(regularization_rmse_results,
                                           data_frame(Method = paste("Regularized Model (lambda =", lambda, ")"),
                                                      RMSE = model_rmse))
}

# Step 6: Find the best lambda based on the lowest RMSE in regularization_rmse_results
best_lambda <- lambdas[which.min(regularization_rmse_results$RMSE)]
best_rmse <- min(regularization_rmse_results$RMSE)

# Step 7: Append the best result of the regularized model to the overall rmse_results (that contains previous models' results)
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = paste("Regularized Model with Movie, User, Genre Effects (lambda =", best_lambda, ")"),
                          RMSE = best_rmse))

# Step 8: Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")

# Step 9: Print the best lambda and the corresponding RMSE
print(paste("Optimal lambda:", best_lambda, "with RMSE:", best_rmse))

# Optional: Plot RMSE vs Lambda for Regularized Model
ggplot(regularization_rmse_results, aes(x = lambdas, y = RMSE)) +
  geom_point() +
  geom_line() +
  xlab('Lambda') +
  ylab('RMSE') +
  ggtitle('RMSE vs Lambda for Regularized Model')
```

The result shows that regularization (without the time effect) indeed ameliorates our model, even though not a very significant improvement:

Method	**RMSE**
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752

Movie + User + Genre Effects	0.865412
Movie + User + Genre + Time Effects (GAM)	0.865955
Movie + User + Genre + KSmooth Time Effects (box bandwidth = 300)	0.865001
Movie + User + Genre + LOESS Time Effects (span = 0.9)	0.865086
Regularized Model with Movie, User, Genre Effects (lambda = 4.75)	0.864743

We find the best lambda is 4.75.



6.2. Model 6.2: Regularized Model with Time Effect (KSmooth)

Now we are going to take a look whether including time effect in our model will further improve our model:

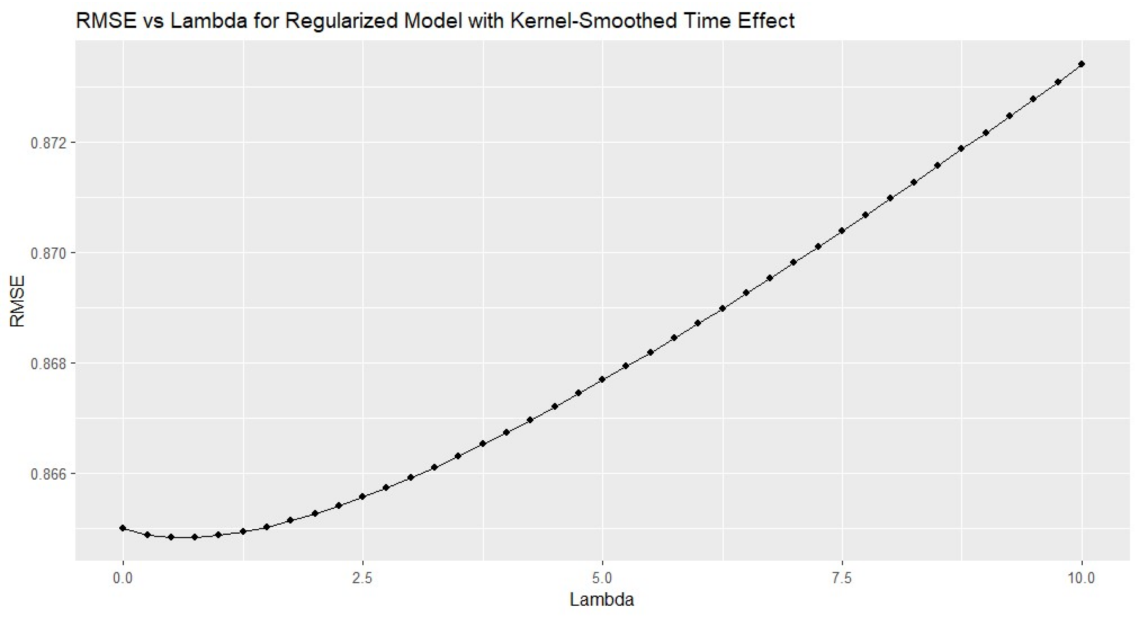
```
# 6. Model 6.2: Regularized Model with Kernel-Smoothed Time Effect ----------------
# Step 1: Define lambda values for regularization
lambdas <- seq(0, 10, 0.25)

# Step 2: Calculate the time effect using kernel smoothing (ksmooth)
train_set_byweek <- train_set %>%
  mutate(weekNum = (timestamp - min(timestamp)) %/% (7 * 24 * 60 * 60) + 1) %>%
  group_by(weekNum) %>%
  summarize(y = mean(rating)) %>%
  arrange(weekNum)

# Apply kernel smoothing on the train set
smoothed_train <- ksmooth(x = train_set_byweek$weekNum, y = train_set_byweek$y, kernel = "normal", bandwidth = 300)
# we use 300 as bandwidth since it was found as the best bandwidth for our data set at the previous model (Model 5)

# Extract fitted values for each week in the train set
smoothed_train_fitted <- approx(smoothed_train$x, smoothed_train$y, xout = train_set$weekNum)$y

# Step 3: Calculate the movie, user, and genre effects with regularization, incorporating the time effect
regularization_rmse_results <- data.frame()
```

```r
for (lambda in lambdas) {

  # Movie Effect with Regularization and Time Effect
  movie_effect_t <- train_set %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu - f_t) / (n() + lambda)) # Regularized

  # User Effect with Regularization and Time Effect
  user_effect_t <- train_set %>%
    left_join(movie_effect_t, by = "movieId") %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - mu - b_i - f_t) / (n() + lambda)) # Regularized

  # Genre Effect with Regularization and Time Effect
  genre_effect_t <- train_set %>%
    left_join(movie_effect_t, by = "movieId") %>%
    left_join(user_effect_t, by = "userId") %>%
    mutate(f_t = smoothed_train_fitted) %>%
    group_by(genres) %>%
    summarize(b_g = sum(rating - mu - b_i - b_u - f_t) / (n() + lambda)) # Regularized

  # Step 4: Use kernel smoothing to extract the smoothed time effect for the test set
  smoothed_test_fitted <- approx(smoothed_train$x, smoothed_train$y, xout = test_set$weekNum)$y

  # Step 5: Obtain predictions for the test set using regularized effects and time effect
  predicted_ratings <- test_set %>%
    mutate(f_t = smoothed_test_fitted) %>%
    left_join(movie_effect_t, by = "movieId") %>%
    left_join(user_effect_t, by = "userId") %>%
    left_join(genre_effect_t, by = "genres") %>%
    mutate(pred = mu + b_i + b_u + b_g + f_t) %>%
    pull(pred)

  # Step 6: Compute RMSE for each lambda
  model_rmse <- RMSE(predicted_ratings, test_set$rating)

  # Step 7: Store RMSE for each lambda
  regularization_rmse_results <- bind_rows(regularization_rmse_results,
                                           data_frame(Method = paste("Regularized Model (lambda =", lambda, ") with M,U,G Effects +
                                              KSmooth Time Effect (bandwidth =", best_bandwidth, ")"),
                                           RMSE = model_rmse))
}

# Step 8: Find the best lambda based on the lowest RMSE
best_lambda <- lambdas[which.min(regularization_rmse_results$RMSE)]
best_rmse <- min(regularization_rmse_results$RMSE)

# Step 9: Append the best regularized model result with time effect to the overall RMSE results
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = paste("Regularized Model with Kernel-Smoothed Time Effect (lambda =", best_lambda,
                            ")"),
                          RMSE = best_rmse))

# Step 10: Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")
```

```
# Step 11: Print the best lambda and the corresponding RMSE
print(paste("Optimal lambda:", best_lambda, "with RMSE:", best_rmse))

# Plot RMSE vs Lambda for Regularized Model with Kernel-Smoothed Time Effect
ggplot(regularization_rmse_results, aes(x = lambdas, y = RMSE)) +
  geom_point() +
  geom_line() +
  xlab('Lambda') +
  ylab('RMSE') +
   ggtitle('RMSE vs Lambda for Regularized Model with Kernel-Smoothed Time Effect')
```

It appears that including the time effect into our model does not improve it, yet increase the RMSE hence making it less accurate, though at a very small margin.

Method	RMSE
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752
Movie + User + Genre Effects	0.865412
Movie + User + Genre + Time Effects (GAM)	0.865955
Movie + User + Genre + KSmooth Time Effects (box bandwidth = 300)	0.865001
Movie + User + Genre + LOESS Time Effects (span = 0.9)	0.865086
Regularized Model with Movie, User, Genre Effects (lambda = 4.75)	0.864743
Regularized Model with Kernel-Smoothed Time Effect (lambda = 0.5)	0.864824



Since taking into account the time effect turns out to be unbeneficial to our prediction model accuracy, we decide to drop the time effect. On the next model we will continue improving our regularized model without the time effect (model 6.1).

7. Model 7: Matrix Factorization

In our exploration of model refinement, we transitioned to matrix factorization. dissects complex relationships by decomposing variables into underlying patterns, akin to how mathematical factors like 6 and 4 contribute to the product 24. This method proves highly effective for data with inherent latent factors—such as preferences in movie recommendations. Among the matrix factorization approaches, we selected Singular Value Decomposition (SVD), which elegantly captures these hidden dependencies, thereby enhancing predictive accuracy.

There are several methods of matrix factorization actually, but given the limited knowledge and time, we will only use the SVD model as explained by Funk (2006). The other methods include:

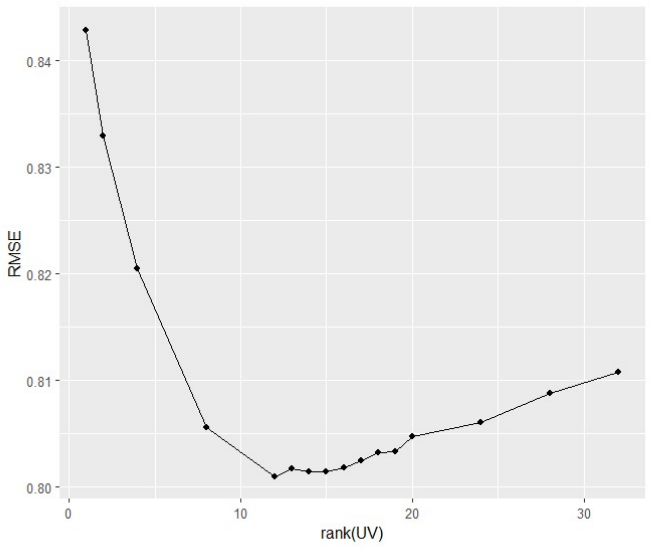
1) Singular Value Decomposition (SVD)
2) Non-negative Matrix Factorization (NMF)
3) QR Decomposition
4) Cholesky Decomposition
5) Eigenvalue Decomposition
6) LU Decomposition
7) CUR Decomposition
8) Latent Dirichlet Allocation (LDA)
9) Independent Component Analysis (ICA)
10) Tensor Factorization
11) Sparse Matrix Factorization
12) Alternating Least Squares (ALS)
13) Bayesian Matrix Factorization (BMF)
14) Factorization Machines (FMs)
15) Matrix Completion (MC)

Mathematically represented, our model is:

$$Y_{u,i} = \mu + b_{1,i} + b_{2,u} + b_{3,g} + UV^T + \epsilon_{u,i}$$

where each of the bias terms or effects $b_{1,i}$, $b_{2,u}$, $b_{3,}$and the matrix factorization term UV^T are regularized.

Using iterative tuning, we optimized the number of latent features to 12, which yielded the best RMSE performance among several configurations. This tuning process involved progressively increasing the feature count, recalculating RMSE each time to pinpoint the feature set that balanced complexity with accuracy. See the graph below on UV and the RMSE relationship.



Here is our code for the model 7:

```r
# 7. Model 7: Matrix Factorization with Singular Value Decomposition ------------
# Load necessary libraries
library(Rcpp)
library(RcppArmadillo)
library(dplyr)
library(purrr)
library(ggplot2)

# Load the C++ file
Rcpp::sourceCpp("C:/Users/HP/Downloads/Harvard DS/svd.cpp")

# Step 1: Prepare residuals from the previous best model
mu <- mean(train_set$rating)  # Assume `mu` is the global mean rating

previous_train <- train_set %>%
  left_join(movie_avgs_reg, by = "movieId") %>%
  left_join(user_avgs_reg, by = "userId") %>%
  left_join(genre_avgs_reg, by = "genres") %>%
  mutate(pred = mu + b_i + b_u + b_g) %>%
  pull(pred)
residuals_train <- as.numeric(train_set$rating - previous_train)

# Test set predictions for the previous model
previous_test <- test_set %>%
  left_join(movie_avgs_reg, by = "movieId") %>%
  left_join(user_avgs_reg, by = "userId") %>%
  left_join(genre_avgs_reg, by = "genres") %>%
  mutate(pred = mu + b_i + b_u + b_g) %>%
  pull(pred)

# Create unique indices for userId without gaps
Uidx <- numeric(max(train_set$userId))
Uidx[unique(train_set$userId)] <- seq_along(unique(train_set$userId))

# Create unique indices for movieId without gaps
Vidx <- numeric(max(train_set$movieId))
Vidx[unique(train_set$movieId)] <- seq_along(unique(train_set$movieId))

# Step 2: Define matrix factorization function with parameters
funk <- function(Uidx, Vidx, residuals, nFeatures, steps = 500,
                 regCoef = 0.02, learningRate = 1e-3) {
  funkCpp(Uidx[train_set$userId] - 1,  # Convert to 0-based indexing
          Vidx[train_set$movieId] - 1,
          residuals,
          nFeatures, steps, regCoef, learningRate)
}

# Step 3: Tune latent features
set.seed(1)
if(!file.exists('funk_tuning.Rdata')) {
  nFeatures <- c(1, 2, 4, 8, seq(12, 20), 24, 28, 32)
  rmses <- sapply(nFeatures, function(nF) {
    message(nF, ' features')

    # Run matrix factorization with the current feature count
    set.seed(1)
    funkResult <- funk(Uidx, Vidx, residuals_train, nFeatures = nF, steps = 500)
    U <- funkResult$U
    V <- funkResult$V
    # Predict ratings for the test set
    predicted_ratings_funk <- test_set %>%
      mutate(pred = previous_test +
               map2_dbl(userId, movieId, function(u, v) U[Uidx[u], ] %*% V[Vidx[v], ])) %>%
      pull(pred)

    # Calculate RMSE for current number of features
    rmse <- RMSE(predicted_ratings_funk, test_set$rating)
    message(rmse, '\n')
    return(rmse)
  })

  save(nFeatures, rmses, file = 'funk_tuning.Rdata')
}

# Load RMSE results if file exists
if (file.exists('funk_tuning.Rdata')) {
  load('funk_tuning.Rdata')
}

# Plot RMSE against the number of MF features
par(cex = 0.7)
qplot(nFeatures, rmses, xlab = 'rank(UV)', ylab = 'RMSE', geom = c('point','line'))

# Find the optimal number of features
optimal_nfeatures <- nFeatures[which.min(rmses)]
optimal_nfeatures

# Run matrix factorization with the optimal number of features
set.seed(1)
if (!file.exists('funk.Rdata')) {
  funkResult <- funk(Uidx, Vidx, residuals_train, nFeatures = nFeaturesOpt, steps = 500)
  save(nFeaturesOpt, funkResult, file = 'funk.Rdata')
}

# Load matrix factorization data from file
load('funk.Rdata')

U <- funkResult$U
V <- funkResult$V

# Predict ratings for the test set
predicted_ratings_funk <- test_set %>%
  mutate(pred = previous_test +
           map2_dbl(userId, movieId, function(u, v) U[Uidx[u], ] %*% V[Vidx[v], ])) %>%
  pull(pred)

# Compute and print RMSE
mf_rmse <- RMSE(predicted_ratings_funk, test_set$rating)
mf_rmse

# Step 9: Append the best regularized model result with time effect to the overall RMSE results
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = paste("Matrix Factorization (features =", optimal_nfeatures, ") with Regularized Movie, User,
                          Genre Effect"),
                          RMSE = mf_rmse))
```

```
# Step 10: Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")
```

The result is:

Method	**RMSE**
Constant Value 2.5	1.465932
Using the Data Set Mean	1.060023
Movie Effect Model	0.943222
Movie + User Effects Model	0.865752
Movie + User + Genre Effects	0.865412
Movie + User + Genre + Time Effects (GAM)	0.865955
Movie + User + Genre + KSmooth Time Effects (box bandwidth = 300)	0.865001
Movie + User + Genre + LOESS Time Effects (span = 0.9)	0.865086
Regularized Model with Movie, User, Genre Effects (lambda = 4.75)	0.864743
Regularized Model with Kernel-Smoothed Time Effect (lambda = 0.5)	0.864824
Matrix Factorization (features = 12) with Regularized Movie, User, Genre Effect	0.801769

F. RESULT OF FINAL TESTING

We will not test the efficacy of our model on the final_holdout_test set. The final or the best model we use will be our model 7 - matrix factorization which is based on the regularized model that takes into account movie, user, and genre effects (model 6.1).

Here is our code:

```
# F. FINAL TESTING USING FINAL HOLDOUT TEST
# Rechecking the final_holdout_test
head(final_holdout_test)

# Note that some users and movieid in the final_holdout_test would not be present in the our model.
# This is because after creating the final_holdout_test, even though we made sure that it has the same users and movieId in the edx dataset,
# we still partitioned the edx data set into train and test set, making some users or movies absent in the train set that we used to build our model.
# Since we cannot alter the final_holdout_test, we then will just assign 0 to the b_i, b_u, and b_g when unmatchings are found.

# Generate final predictions for the FINAL HOLDOUT TEST dataset
predicted_ratings_fht <- final_holdout_test %>%
  left_join(movie_avgs_reg, by = "movieId") %>%
  left_join(user_avgs_reg, by = "userId") %>%
  left_join(genre_avgs_reg, by = "genres") %>%
  mutate(
    # Replace any NA values in b_i, b_u, and b_g with 0
    b_i = ifelse(is.na(b_i), 0, b_i),
    b_u = ifelse(is.na(b_u), 0, b_u),
    b_g = ifelse(is.na(b_g), 0, b_g),
    pred = mu + b_i + b_u + b_g +
      map2_dbl(userId, movieId, function(u, v) {
        # Check if userId and movieId have valid indices in Uidx and Vidx
        if (!is.na(Uidx[u]) && Uidx[u] > 0 && !is.na(Vidx[v]) && Vidx[v] > 0) {
          U[Uidx[u], ] %*% V[Vidx[v], ]
        } else {
          0  # Return 0 if index is missing
        }
      })
  ) %>%
  pull(pred)

# Compute RMSE
fht_rmse <- RMSE(predicted_ratings_fht, final_holdout_test$rating)
fht_rmse

# Append the final holdout test RMSE to the overall RMSE results
rmse_results <- bind_rows(rmse_results,
                          data_frame(Method = paste("Final Holdout Test - Matrix Factorization with", optimal_nfeatures, "features"),
                          RMSE = fht_rmse))

# Export the updated RMSE results to an Excel file
write_xlsx(rmse_results, "rmse_results.xlsx")
```

Our model produces RMSE of 0.801405139. The resulting RMSE underscores the model's precision, outperforming previous approaches and validating matrix factorization's effectiveness in this domain.

Final Holdout Test - Matrix Factorization with 12 features	0.801405139

G. CONCLUSION

In summary, our analysis journey—beginning from baseline models that only uses mean, and gradually taking into account the movie effect, user effect, and genre effect. We tried including the time effect in our model but later found out that it does not really help improve our accuracy. We later advance our model through regularization and lastly through matrix factorization. This process highlights the evolving sophistication in predictive modelling for movie recommendations. The matrix factorization model, enhanced with user, movie, and genre effects, emerged as the optimal approach, significantly reducing RMSE and capturing intricate user preferences.

Despite these achievements, limitations exist given that our RMSE is still relatively big with 0.80 in the end. The last model, which is matrix factorization is also relied heavily on substantial computational resources, coupled with the challenge it cannot overcome when dealing with sparsely represented users or movies. Unfortunately, further looking for more accurate model will surely take more complexity in it, though in real projects might worth trying.

H. DISCLAIMER & REFERENCES

This project was completed independently and is not plagiarized. It references the three sources listed below. The use of generative AI was minimal, limited solely to proofreading and troubleshooting.

Chan, Y. (2022). "SVD.pp."

Funk, S. (2006). "Netflix Update: Try This at Home."

Irizarry, R. (n.d.). "Introduction to Data Science - Data Analysis and Prediction Algorithms with R."

***