Capstone Project - Movielens

Poon HY

10/21/2021

1 INTRODUCTION

The aim of this project is to develop a movie recommendation model using the Movielens dataset.

The dataset is downloaded from the website www.grouplens.org. There are a number of versions of the Movielens datasets on this site. We are specifically using the Movielens 10M, which has about 10 million observations.

The code below downloads the dataset, separates it into two sets ("edx" and "validation"), and further separates the edx set into a train dataset ("train_set") and test dataset ("test_set"). The 2 datasets **train_set** and **test_set** will be used to develop the recommendation model to predict movie ratings in the **validation** dataset as a final test of the model.

```
# Install packages as required
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")
if(!require(recosystem)) install.packages("recosystem", repos = "http://cran.us.r-project.org")
if(!require(Rcpp)) install.packages("Rcpp", 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

# Download Movielens 10M dataset into a temp file named dl
dl <- tempfile()
download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

# Extract ratings file which has 4 columns of userId, movieId, ratting and timestamp
ratings <- fread(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                 col.names = c("userId", "movieId", "rating", "timestamp"))

# Extract movies file which has 3 columns of movieId, title and genres
movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")

movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(movieId),
                                           title = as.character(title),
```

```
                                           genres = as.character(genres))

# Combine the 2 files into 1 dataset named movielens
movielens <- left_join(ratings, movies, by = "movieId")

# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding")
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)

# Split edx into training and test sets
set.seed(1, sample.kind="Rounding")
index <- createDataPartition(y = edx$rating, times = 1, p = 0.2,
                             list = FALSE)
train_set <- edx[-index,]
test_set <- edx[index,]

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

So now we have 3 relevant datasets: **train_set**, **test_set** and **validation**, which have the following numbers of rows:

```
##            Number_of_rows
## train_set         7200043
## test_set          1799966
## validation         999999
```

Each dataset consists of 5 columns as listed below:

```
## [1] "userId"    "movieId"   "rating"    "timestamp" "title"     "genres"
```

2 ANALYSIS

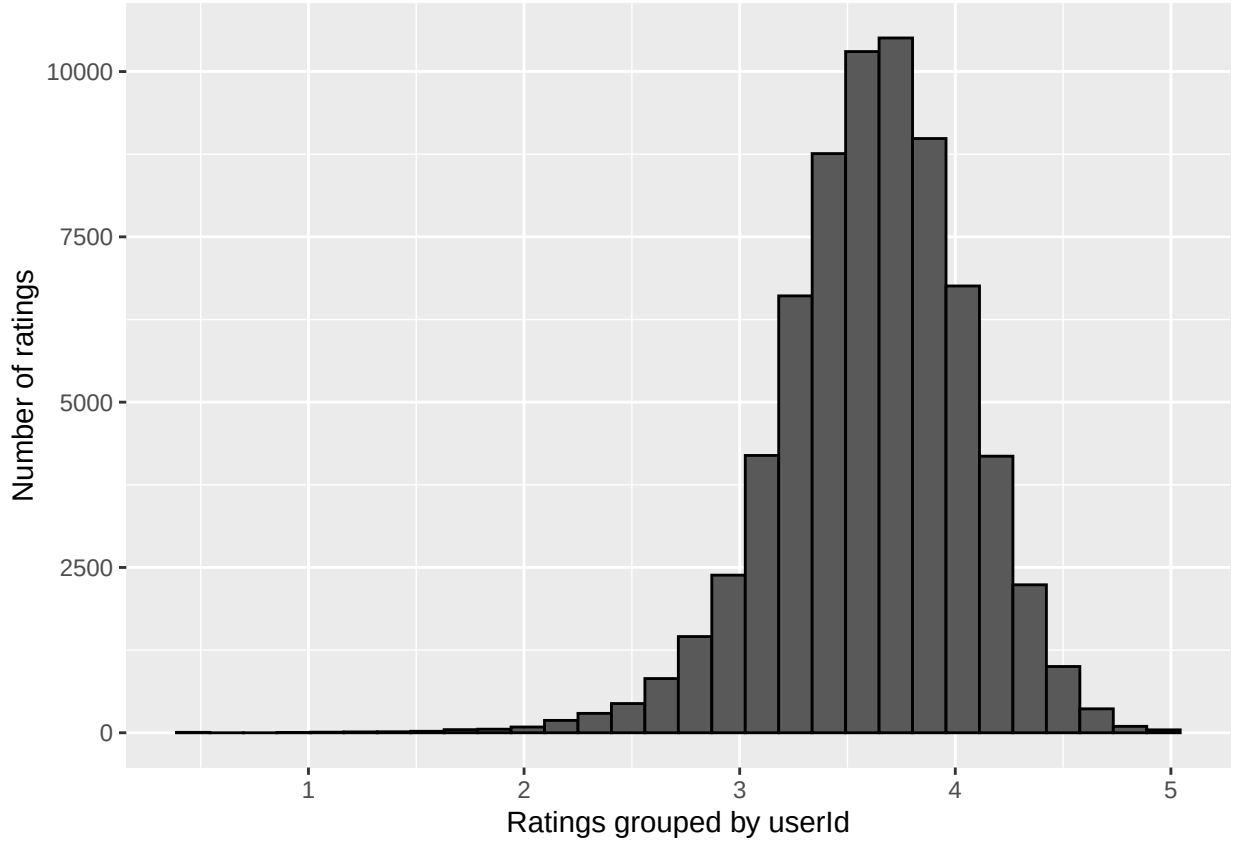
2.1 Users

I will begin the analysis of train_set by focusing on the column "userId". From the code below, we know there are 69878 unique raters in train_set.

```
# Number of unique userIds in train_set
length(unique(train_set$userId))
```

When I plotted the distribution of ratings by users with the code below, we see that most of the ratings are around 3.5 to 4.

```
# Plot b_u = average user rating
train_set %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating)) %>%
  ggplot(aes(b_u)) +
  geom_histogram(bins = 30, color = "black") +
  xlab("Ratings grouped by userId") + ylab("Number of ratings")
```

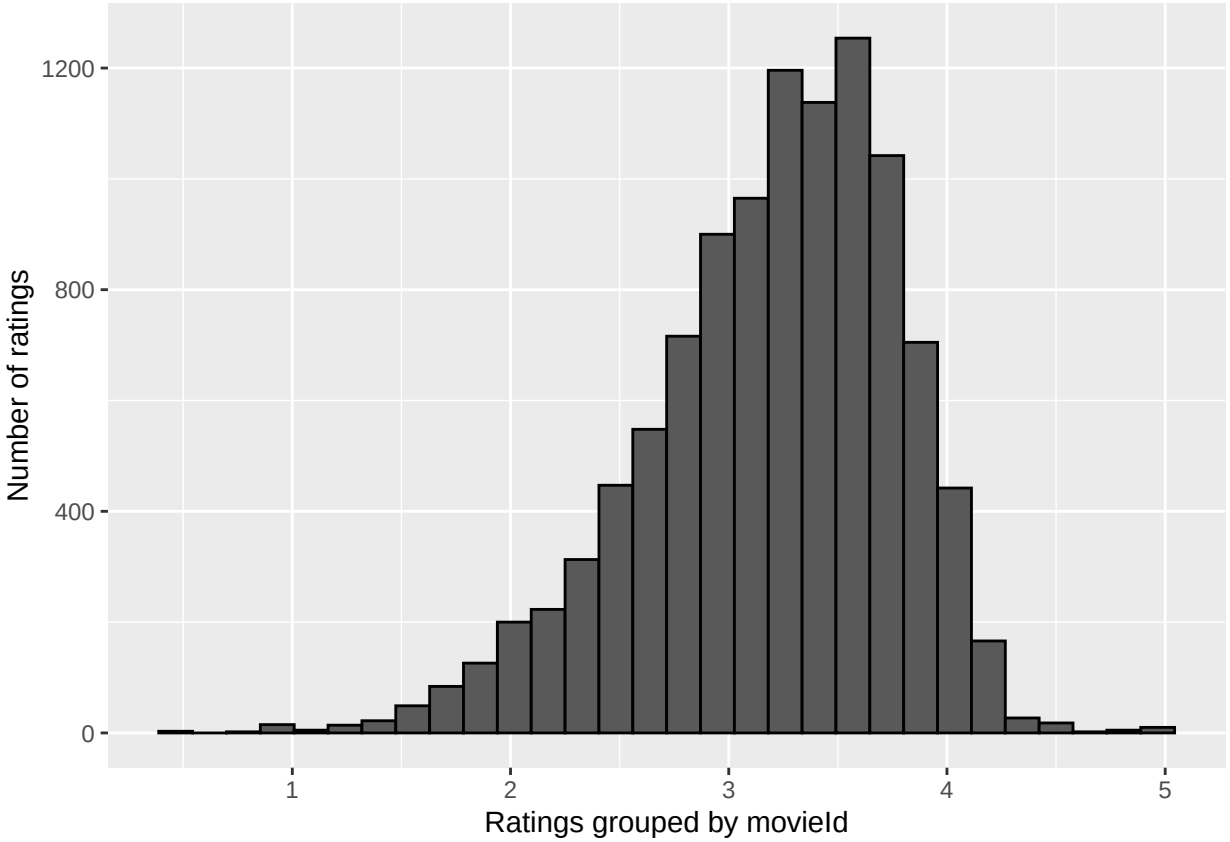


2.2 Movie ID

Next, I looked at movieId. In train_set, there are 10637 unique movies. I plotted the distribution of ratings of movies with the code below:

```
# Plot b_i = average movie rating
train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating)) %>%
  ggplot(aes(b_i)) +
```

```
  geom_histogram(bins = 30, color = "black") +
  xlab("Ratings grouped by movieId") + ylab("Number of ratings")
```



As in the user ratings, the histogram is skewed to the right. Most movies were rated 3 or 3.5.

2.3 Genres

The number of unique values in the genres column is 797. I also listed the first 6 genres entries in train_set to give us a sense of what they look like.

```
# List first 6 entries of genres column in train_set
head(train_set$genres)
```

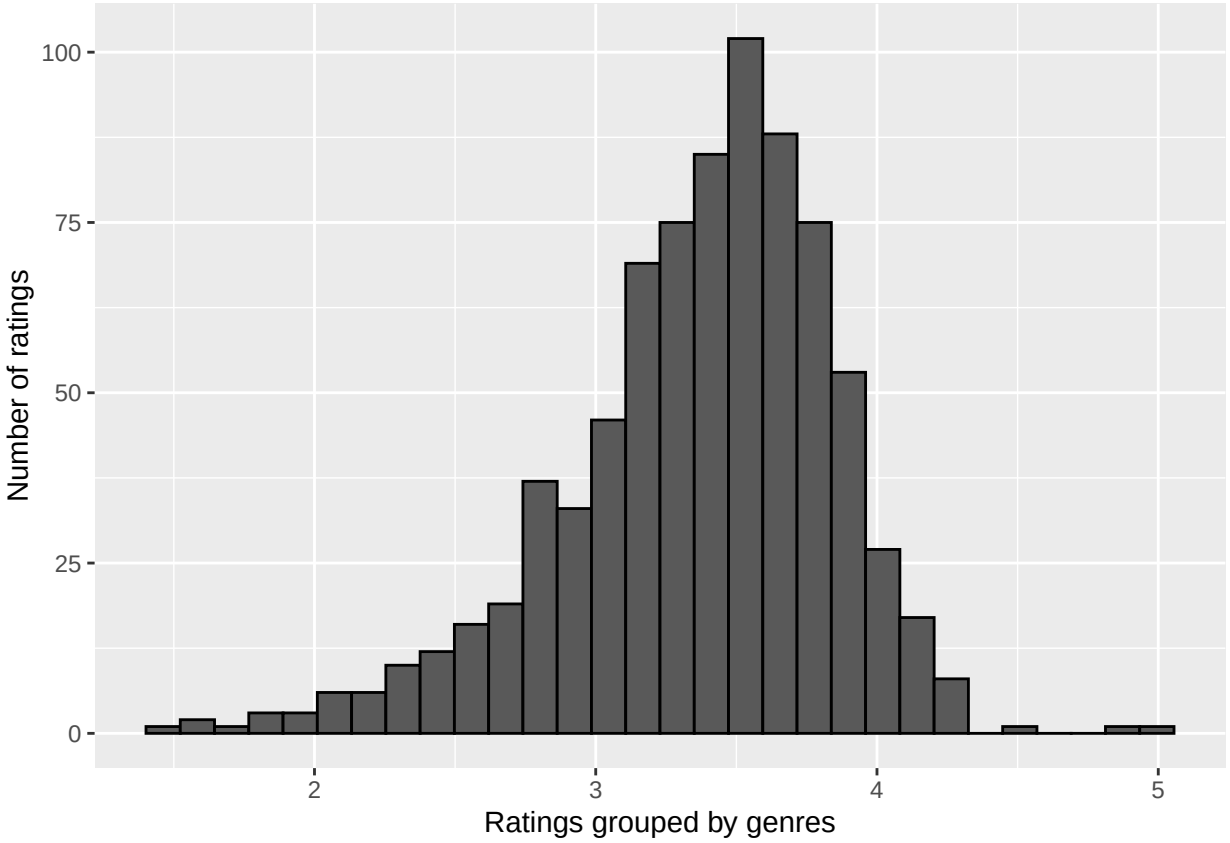
```
## [1] "Action|Crime|Thriller"
## [2] "Action|Adventure|Sci-Fi"
## [3] "Action|Adventure|Drama|Sci-Fi"
## [4] "Children|Comedy|Fantasy"
## [5] "Adventure|Animation|Children|Drama|Musical"
## [6] "Action|Romance|Thriller"
```

There are so many unique values because different combinations of genres are counted as unique. So in the sample listed above, the 6 entries are all considered unique because of the combination of different genres. The question is whether we should split up the genres and count them separately. However, doing so might change the nature of the classification (for example, an action-crime-thriller might be quite different from an

action-romance-thriller), and it was not clear if there is a right way for the individual genres to be weighted. Hence, I decided to keep them as it is.

The shape of the distribution of ratings when grouped by genres (plotted below) looks similar to that of userId and movieId.

```
# Plot b_g = average genres rating
train_set %>%
  group_by(genres) %>%
  summarize(b_g = mean(rating)) %>%
  ggplot(aes(b_g)) +
  geom_histogram(bins = 30, color = "black") +
  xlab("Ratings grouped by genres") + ylab("Number of ratings")
```

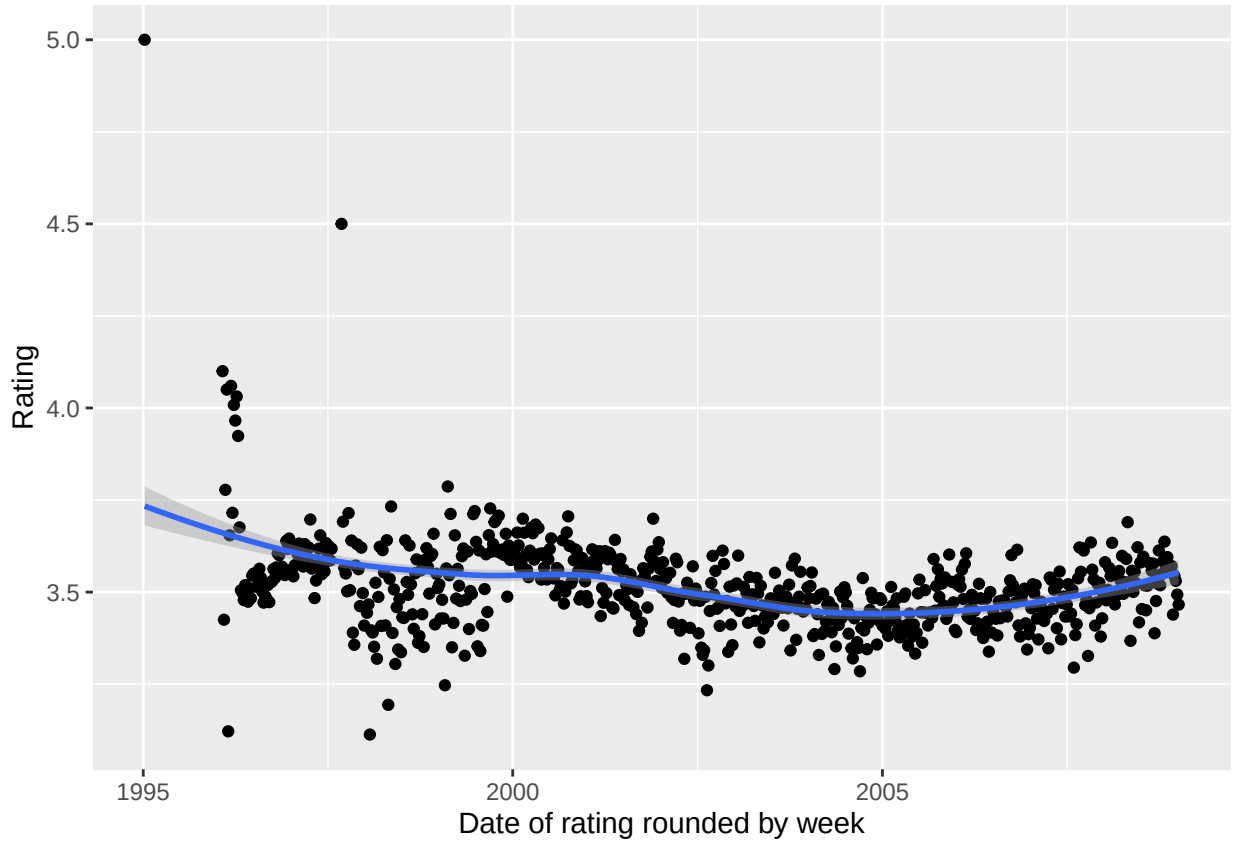


2.4 Timestamp - Time of rating

I now looked across time, specifically the time of rating, which is given by the column timestamp. I rounded up timestamp into blocks of weeks and plot the ratings across the whole period, as follows:

```
# Load lubridate
library(lubridate)
# Convert timestamp to week
train_set <- mutate(train_set, rating_date = as_datetime(timestamp))
train_set %>% mutate(rating_date = round_date(rating_date, unit = "week")) %>%
  # Plot ratings over time
```

```
  group_by(rating_date) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(rating_date, rating)) +
  geom_point() +
  geom_smooth(formula = y ~ x, method = "loess") +
  xlab("Date of rating rounded by week") + ylab("Rating")
```



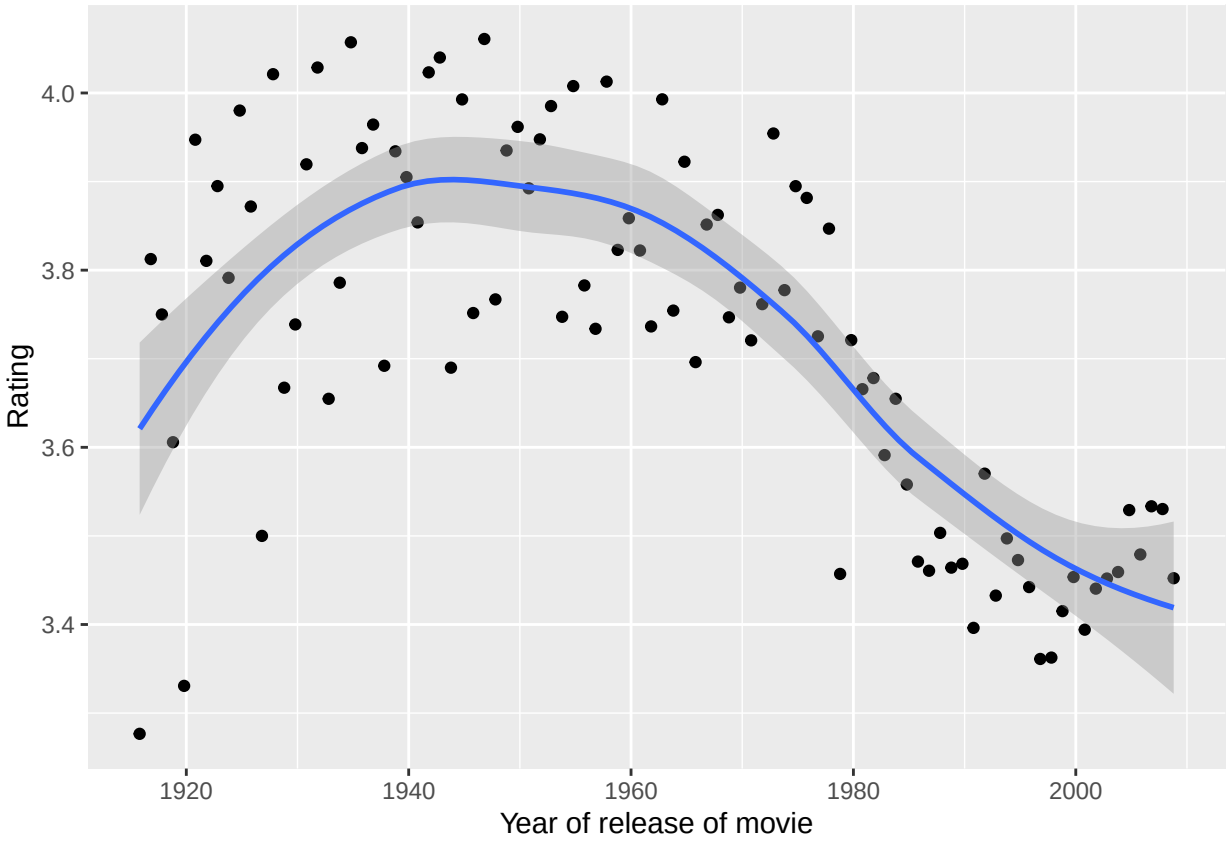
The curve is rather flat with little variation, even less so if we disregard the tail to the left, when ratings were much fewer and skewed by outliers.

2.5 Year of release

I also looked at another time element - the time when the movie was released. This information was not given in a separate column, but the year of release is given in parentheses at the end of the title. I extracted the year of release from the title column and plotted ratings over the year of release, as follows:

```
library(stringr)
# Extract year of movie release from title, which were the 4 digits from the -5 to -2 positions of the
train_set <- train_set %>% mutate(release_year = as.Date(str_sub(title, -5, -2), format="%Y"))
# Plot ratings over year of release
train_set %>%
  group_by(release_year) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(release_year, rating)) +
  geom_point() +
```

```
  geom_smooth(formula = y ~ x, method = "loess") +
  xlab("Year of release of movie") + ylab("Rating")
```



Plotting ratings over year of release looks more meaningful than over time of rating as there is greater variation. Note that the ratings are higher for older movies in general, although ratings do go down for movies released before 1940. This makes sense as users, who only started rating the movies in 1995, would probably pick old movies or remember them if they are well known or known to be good, i.e. what we call "classics".

So far, there seem to be meaningful variations in the ratings when grouped by userId, movieId, genres and release_year, variations we could use to construct a ratings prediction model.

3 METHODS FOR CONSTRUCTING PREDICTION MODELS

3.1 Loss function

To measure the accuracy of the predictions, the root mean squared error (RMSE) as defined in the code below will be calculated for predicted ratings against actual ratings in test_set and, eventually, the validation set.

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

3.2 Assume same rating for all movies

In the simplest recommendation model, we assume all movies are rated the same, i.e the average rating. This is represented by the equation:

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

where $Y_{u,i}$ is the rating for user u and movie i, μ is the mean rating, and $\epsilon_{u,i}$ represents the independent errors.

We calculate μ, or mu, with the following code:

```
# Assume all movies have same (average) rating
mu <- mean(train_set$rating)
mu
```

```
## [1] 3.512482
```

The RMSE for this approach is:

```
# Calculate RSME for just assuming average rating
rmse_avg <- RMSE(test_set$rating, mu)
rmse_avg
```

```
## [1] 1.059904
```

3.3 Include movie effects

We would want to include the effects of other factors such as movieId and userId to improve the RMSE as we have shown that these factors do influence the ratings. Starting with the movie effect, we could model the effect by the equation:

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

where b_i, or b_i, represents the bias due to the movie effect.

b_i can therefore be calculated by subtracting the rating for each movie by the overall average mu. Then I predicted the ratings for test_set and calculated the RMSE that includes movie effects, or rmse_m.

```
# Include movie effect to calculate b_i
movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu))

# Predict ratings and calculate RMSE with movie effect
predicted_ratings <- mu + test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  pull(b_i)
rmse_m <- RMSE(predicted_ratings, test_set$rating)
rmse_m
```

```
## [1] 0.9437429
```

3.4 Include movie effects with regularisation

Through a process called regularisation, I introduced a penalty term lambda that would dilute the effects of ratings which consists of very few samples, as these would skew the results. The following code picks the lambda that minimises the RMSE of the movie effect:

```
# Calculate regularised estimates of movie effect
# To do that, we pick a factor (lamda) that minimises RMSE
lambdas <- seq(0, 10, 0.25)
just_the_sum <- train_set %>%
  group_by(movieId) %>%
  summarize(s = sum(rating - mu), n_i = n())
rmses <- sapply(lambdas, function(l){
  predicted_ratings <- test_set %>%
    left_join(just_the_sum, by='movieId') %>%
    mutate(b_i = s/(n_i+l)) %>%
    mutate(pred = mu + b_i) %>%
    pull(pred)
  return(RMSE(predicted_ratings, test_set$rating))
})
lambda <- lambdas[which.min(rmses)]
```

I then worked out the RMSE including the movie effect, with regularisation.

```
# Calculate b_i with lambda value
movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n()+lambda), n_i = n())

# Predict ratings and calculate RMSE with movie effect and regularisation
predicted_ratings <- mu + test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  pull(b_i)
rmse_m <- RMSE(predicted_ratings, test_set$rating)
rmse_m
```

```
## [1] 0.9436745
```

3.5 Include user effects

I did the same with userId, calculating the average rating from the user effect (b_u), then combining the two effects (movieId with regularisation and userId) to calculate the RMSE, or rmse_m_u.

```
# Include user effect
user_avgs <- train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))

# Predict ratings and calculate RMSE with movie and user effects
predicted_ratings <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
```

```
  left_join(user_avgs, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  pull(pred)
rmse_m_u <- RMSE(predicted_ratings, test_set$rating)
rmse_m_u
```

```
## [1] 0.8657865
```

The RMSE has improved from 0.9436745 (movie effect) to 0.8657865 (movie + user effects).

3.6 Include genres effect

The process is repeated with the genres column, first calculating the average rating due to the genres effect (b_g), then combining three effects (movie, user and genres) to work out the RMSE, or rmse_m_u_g.

```
# Include genre effect
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))

# Predict ratings and calculate RMSE with movie, user and genre effects
predicted_ratings <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  mutate(pred = mu + b_i + b_u + b_g) %>%
  pull(pred)
rmse_m_u_g <- RMSE(predicted_ratings, test_set$rating)
rmse_m_u_g
```

```
## [1] 0.8654497
```

The RMSE improved, but only by a bit to 0.8654497.

3.7 Include year of release

Finally, I included the effect of the year of movie release. For this, I extracted the year of release and repeated the process to work out the RMSE due to the movie, user, genres and release-year effects.

```
# Extract year of movie release from title
library(stringr)
train_set <- train_set %>% mutate(release_year = as.Date(str_sub(title, -5, -2), format="%Y"))
test_set <- test_set %>% mutate(release_year = as.Date(str_sub(title, -5, -2), format="%Y"))

# Include release-year effect
release_year_avgs <- train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
```

```
  left_join(genre_avgs, by="genres") %>%
  group_by(release_year) %>%
  summarize(b_r = mean(rating - mu - b_i - b_u - b_g))

# Predict ratings and calculate RMSE with movie, user, genre and release-year effects
predicted_ratings <- test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  left_join(release_year_avgs, by='release_year') %>%
  mutate(pred = mu + b_i + b_u + b_g + b_r) %>%
  pull(pred)
rmse_m_u_g_r <- RMSE(predicted_ratings, test_set$rating)
rmse_m_u_g_r
```

```
## [1] 0.865278
```

Again, the improvement of the RMSE to 0.865278 is small.

3.8 Tabulation of RMSE

The results of calculating the RMSE from various effects are tabulated below.

```
##          method      RMSE
## 1      Just avg 1.0599043
## 2        +Movie 0.9436745
## 3         +User 0.8657865
## 4        +Genre 0.8654497
## 5 +Release-year 0.8652780
```

3.9 Matrix factorisation using Recosystem

I wanted to use matrix factorisation on the train_set dataset, but it was a challenge to do so as the sparse matrix of userId against movieId was too large for my computer's memory. There is, however, a package called Recosystem that enables this to be done effectively.

Recosystem requires the training and test datasets to be loaded into the computer hard disk with just 3 columns: userId, movieId and ratings. I stored these in 2 separate text files - "train_subset.txt" and "test_subset.txt". The data in the text files are then loaded into "train_reco" and "test_reco".

```
# Load necessary libraries
library(recosystem)
library(Rcpp)

# Save the first 3 columns of train_set and test_set (userId, movieId and ratings) into text files
# Leave out the row names and column names
write.table(train_set[, 1:3],"train_subset.txt",sep="\t", row.names=FALSE, col.names=FALSE)
write.table(test_set[, 1:3],"test_subset.txt",sep="\t", row.names=FALSE, col.names=FALSE)

# Draw data from text files that contain userId, movieId and ratings
train_reco <- data_file("train_subset.txt", package = "recosystem")
test_reco <- data_file("test_subset.txt",  package = "recosystem")
```

From here, a model object r is created and the best-tuned parameter is selected after plugging different values into the function r$tune, as below. I kept most of the parameters at the default values, only plugging in different values for dim (number of latent factors) and lrate (learning rate). The few values used in the code below were the ones that gave the best results after I experimented with a few numbers. Note that the functions take some time to run, although there is a bar that helpfully indicates the progress of the calculations in percentage terms.

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

```
# Create a model object
r <- Reco()

# Select suitable tuning parameters using certain parameters in r$tune
opts <- r$tune(train_reco, opts = list(dim = c(30, 40), lrate = c(0.1, 0.2),
                                       costp_l1 = 0, costq_l1 = 0,
                                       nthread = 1, niter = 10))
opts
```

```
## $min
## $min$dim
## [1] 40
##
## $min$costp_l1
## [1] 0
##
## $min$costp_l2
## [1] 0.01
##
## $min$costq_l1
## [1] 0
##
## $min$costq_l2
## [1] 0.1
##
## $min$lrate
## [1] 0.1
##
## $min$loss_fun
## [1] 0.8035956
##
##
## $res
##    dim costp_l1 costp_l2 costq_l1 costq_l2 lrate  loss_fun
## 1   30        0     0.01        0     0.01   0.1 0.8309941
## 2   40        0     0.01        0     0.01   0.1 0.8401773
## 3   30        0     0.10        0     0.01   0.1 0.8138570
## 4   40        0     0.10        0     0.01   0.1 0.8159120
## 5   30        0     0.01        0     0.10   0.1 0.8044257
## 6   40        0     0.01        0     0.10   0.1 0.8035956
## 7   30        0     0.10        0     0.10   0.1 0.8327358
## 8   40        0     0.10        0     0.10   0.1 0.8305906
## 9   30        0     0.01        0     0.01   0.2 0.9766327
## 10  40        0     0.01        0     0.01   0.2 1.0216543
```

```
## 11  30        0     0.10        0     0.01   0.2 0.9657932
## 12  40        0     0.10        0     0.01   0.2 0.9214622
## 13  30        0     0.01        0     0.10   0.2 0.8103558
## 14  40        0     0.01        0     0.10   0.2 0.8111486
## 15  30        0     0.10        0     0.10   0.2 0.8286258
## 16  40        0     0.10        0     0.10   0.2 0.8289394
```

The listing of "opts" shows the parameters (out of the few plugged in) that resulted in the smallest RMSE. The tuned parameters were used to train the model using training data from train_reco. From there, I used the model to predict ratings using test data from test_reco, and calculated the RMSE (rmse_reco) by comparing the predicted ratings to the actual ratings in test_set.

```
# Train model using training data train_reco
r$train(train_reco, opts = c(opts$min, nthread = 1, niter = 20))
```

```
## iter      tr_rmse          obj
##    0       0.9996   1.0154e+07
##    1       0.8785   8.0950e+06
##    2       0.8453   7.4998e+06
##    3       0.8221   7.1353e+06
##    4       0.8043   6.8765e+06
##    5       0.7902   6.6860e+06
##    6       0.7783   6.5388e+06
##    7       0.7683   6.4253e+06
##    8       0.7595   6.3280e+06
##    9       0.7518   6.2480e+06
##   10       0.7447   6.1802e+06
##   11       0.7383   6.1190e+06
##   12       0.7324   6.0655e+06
##   13       0.7269   6.0164e+06
##   14       0.7218   5.9743e+06
##   15       0.7172   5.9376e+06
##   16       0.7129   5.9025e+06
##   17       0.7089   5.8716e+06
##   18       0.7052   5.8438e+06
##   19       0.7017   5.8172e+06
```

```
# Predict ratings using test data test_reco
# Output goes to memory rather than a file
predicted_reco = r$predict(test_reco, out_memory())

# Calculate RMSE by comparing to test_set
rmse_reco <- RMSE(predicted_reco, test_set$rating)
rmse_reco
```

```
## [1] 0.7888844
```

4 RESULTS

The RMSE results from calculating the biases from various effects and from the Recosystem method are tabulated below:

```
##          method      RMSE
## 1      Just avg 1.0599043
## 2        +Movie 0.9436745
## 3         +User 0.8657865
## 4        +Genre 0.8654497
## 5 +Release-year 0.8652780
## 6    Recosystem 0.7888844
```

The RMSE from Recosystem is clearly lower than the rest. I will hence use the Recosystem method to predict ratings in the validation set.

Similar to what I did with train_set and test_set, I prepared the validation set for Recosystem and worked out the RMSE (rmse_val).

```
# Save the first 3 columns of validation set (userId, movieId and ratings) into text file
write.table(validation[, 1:3],"validation.txt",sep="\t", row.names=FALSE, col.names=FALSE)

# Draw data from validation.txt that contain userId, movieId and ratings
val_reco <- data_file("validation.txt", package = "recosystem")

# Predict ratings using validation set, sending results to memory rather than a file
predicted_val = r$predict(val_reco, out_memory())

# Calculate RMSE by comparing to validation set
rmse_val <- RMSE(predicted_val, validation$rating)
rmse_val
```

```
## [1] 0.7884861
```

The resulting RMSE was 0.7884861, close to that for test_set and well below the required 0.86490.

5 CONCLUSION

The calculations used to predict the ratings by accounting for biases due to various effects are simple. However, the downside is that the improvements in RMSE are small after accounting for the first effect, which was the movie effect. There is also a limit to how low the RMSE can go, i.e. there is a limit to the accuracy of the predictions using this model.

I tried using matrix factorisation, but quickly found out that a normal desktop computer could not handle the large sparse matrix of userId against movieId. Then I came across the Recosystem package that enabled the factorisation to be done, and the resulting RSME was a significant improvement.

The results, including the final RSME from the validation set, are:

```
##             method      RMSE
## 1         Just avg 1.0599043
## 2           +Movie 0.9436745
## 3            +User 0.8657865
## 4           +Genre 0.8654497
## 5    +Release-year 0.8652780
## 6       Recosystem 0.7888844
## 7 Final validation 0.7884861
```