HarvardX - MovieLens 10M - Capstone Project

Chris Mann

6/13/2019

Introduction

As part of the HarvardX Data Science Certificate program, this All Learner- Capstone project focuses on the creation of a movie recommendation system using the **MovieLens 10M** dataset (publicly available from the GroupLens Research Lab).

All files can be downloaded from GitHub.

The goal of this project is to utilize the tools and techniques covered in the course to achieve a **Root Mean Squared Error (RMSE)** on a segregated test dataset that is smaller than: **RMSE <= 0.87750**

Although we are only using the 10 Million record MovieLens dataset, the number of records, unfortunately, is still too large to be practically manipulated on most laptops and desktops. The dataset size combined with R's current limitations in the areas of parallel computing, disk-based virtualization, and limited GPU support led to many algorithms quickly exhausting even 64Gb RAM. Many of the even common algorithms (K-Nearest Neighbors, Naive-Bayes, SVM, etc.) implemented in R and wrapped by the Caret package require all of the training data to be stored in system memory resulting in tuning and training attempts being aborted when trying to allocate 100s of Gbs of RAM. These limitations severely restricted both the algorithms and validation approaches available.

Although resource limitations are the norm in Data Science, this course isn't focused on the available techniques to handle "Big Data." It is my sincere hope that future cohorts will be allowed to use the smaller **1M MovieLens** dataset so that they can focus on exploring the algorithms and not dealing with RAM limitations.

This project covers the utilization of the following techniques, which all fit within resource limits and deliver RMSE values smaller than the project target:

- Standard & Regularized Linear Regression
- XGB – Extreme Parallel Tree Boosting (xgBoost)
- RECO - Recommender w/ Matrix Factorization (recosystem)

Best RMSE Results by Method:

```
## Loading required package: knitr

## Loading required package: kableExtra

## Loading required package: scales

## Loading required package: tidyverse

## Registered S3 methods overwritten by 'ggplot2':
##   method         from
##   [.quosures     rlang
##   c.quosures     rlang
##   print.quosures rlang

## -- Attaching packages ---------------------------------------------------------------------------- tidy

## v ggplot2 3.1.1     v purrr   0.3.2
## v tibble  2.1.2     v dplyr   0.8.1
## v tidyr   0.8.3     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------------------------------------------ tidyverse_
## x readr::col_factor() masks scales::col_factor()
## x purrr::discard()    masks scales::discard()
## x dplyr::filter()     masks stats::filter()
## x dplyr::group_rows() masks kableExtra::group_rows()
## x dplyr::lag()        masks stats::lag()
```

```
## Loading required package: lubridate
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```

```
## Loading required package: tinytex
```

Method	Best RMSE
Reg LR: Movie + User + Time + Genre Effects Model:	0.8643084
XGB Mixed Tree 50 Boost Rnds:	0.8539179
RECO 1000 X 100:	0.7740506

MovieLens 10M Dataset Overview

Load and Create the Train & Test Datasets

Creation of the MovieLens 10M training dataset (edx) and the 10% random holdout testing dataset (validation) is accomplished using the course provided R script (below):

NOTE: if using R version 3.6.0, be sure to use: set.seed(1, sample.kind = "Rounding") instead of set.seed(1)

```
#####################################
# Create edx set and validation set
#####################################

# Note: this process could take a couple of minutes

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")
```

```
## Loading required package: caret
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

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

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

ratings <- read.table(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
```

```
                      col.names = c("userId", "movieId", "rating", "timestamp"))

movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))

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

# Validation set will be 10% of MovieLens data

# set.seed(1) # if using R 3.6.0: set.seed(1, sample.kind = "Rounding")
set.seed(1, sample.kind = "Rounding")
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
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)
```

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```
edx <- rbind(edx, removed)

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

Examining the MovieLens Data

```
class(edx)
```

```
## [1] "data.frame"
```

```
glimpse(edx)
```

```
## Observations: 9,000,055
## Variables: 6
## $ userId    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ movieId   <dbl> 122, 185, 292, 316, 329, 355, 356, 362, 364, 370, 37...
## $ rating    <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5...
## $ timestamp <int> 838985046, 838983525, 838983421, 838983392, 83898339...
## $ title     <chr> "Boomerang (1992)", "Net, The (1995)", "Outbreak (19...
## $ genres    <chr> "Comedy|Romance", "Action|Crime|Thriller", "Action|D...
```

The **edx** dataset is a Data Frame with 9,000,055 observations of 6 variables:

- **rating:** User defined rating from 0 to 5 in 0.5 increments. This will be our target variable. Technically this is ordinal and could be used as categorical, but we will be treating it as a continuous variable since our evaluation method is RMSE
- **userId:** ID of the User giving the Rating observation
- **movieId:** ID of the Movie being Rated
- **timestamp:** POSIX datetime INT from 1 Jan 1970
- **title:** Movie Title
- **genres:** Character string of Genre categories each separated by the pipe "|" character

```
# Create dataframe with additional User rating information:
#       Average (u_avg), StDev (u_std), Number of Reviews (u_numrtg)
user_data <- edx %>% group_by(userId) %>%
  summarize(u_avg = mean(rating),
            u_std = sd(rating),
            u_numrtg = as.numeric(n()))

# Create dataframe with additional Movie rating information:
#       Average (m_avg), StDev (m_std), Number of Reviews (m_numrtg)
movie_data <- edx %>% group_by(movieId) %>%
  summarize(m_avg = mean(rating),
            m_std = sd(rating),
            m_numrtg = as.numeric(n()))

# Number of unique Users
nrow(user_data)
```

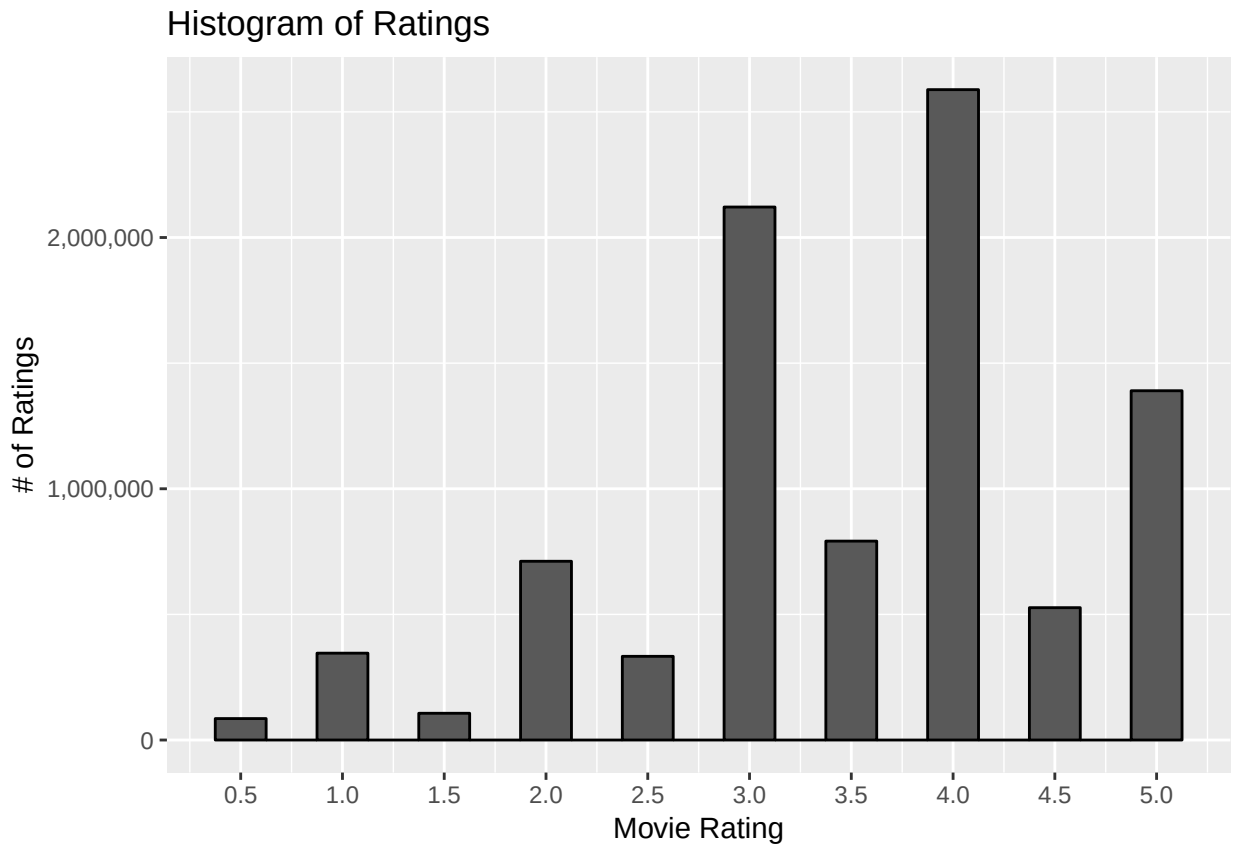
```
## [1] 69878
```

```
# Number of unique Movies
nrow(movie_data)
```

```
## [1] 10677
```

The dataset contains the ratings of 10,677 Movies by 69,878 Users

```
# Plot the distribution of Ratings in the Training set

edx %>%
  ggplot(aes(x= rating)) +
  geom_histogram(binwidth = 0.25, color = "black") +
  scale_x_continuous(breaks=seq(0, 5, 0.5)) +
  scale_y_continuous(labels=comma) +
  labs(x="Movie Rating", y="# of Ratings") +
  ggtitle("Histogram of Ratings")
```



Movie ratings aren't evenly distributed and half-ratings (x.5) are less frequent than whole number ratings.

More exploration of the characteristics of the MovieLens dataset will be explored as we develop the Linear Regression model.

Recommender System Methodology

As mentioned in the Introduction, this project will develop and produce three different recommendation systems that all achieve an RMSE less than our target.

- Standard & Regularized Linear Regression
- XGB – Extreme Parallel Tree Boosting (xgBoost)
- RECO - Recommender w/ Matrix Factorization (recosystem)

Standard & Regularized Linear Regression

As the first approach, we will build and test both a simple Linear Regression model and a Regularized Linear Regression model. This will be done by successively adding mean distributions based upon each of the following variables in the dataset:

- Rating (mu) - Starting with an overall average of all ratings (mu)
- Movie (b_i) - Adding the Average rating of each individual Movie
- User (b_u) - Adding the Average rating of each individual User
- Time (w_u) - Adding the weekly average by converting the timestamp into a weekly date
- Genre (g_u) - Adding the average for each unique genre string. ex: "Drama|Action"

Linear Regression Model Creation

```
# Create working copied of the Training (edx) and Test (validation) datasets
train_set <- edx
test_set <- validation

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

Simple Overall Average

Create and Display the RMSE results of using the overall Average (mu) of all ratings:

```
#*****************************************************************************
# Linear Regression - Using just the overall average
#   Yu,i = mu
#*****************************************************************************

# Calculate the overall average rating
mu <- mean(train_set$rating)
mu
```

```
## [1] 3.512465
```

```
# Evaluate the performance of simply guessing the overall average
rmse <- RMSE(test_set$rating, mu)
rmse
```

```
## [1] 1.061202
```

```
# Save the RMSE result to display later
LR_rmse_results <- tibble(Method = "LR: Base Mean Model",
                          RMSE = rmse)
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE
LR: Base Mean Model	1.061202

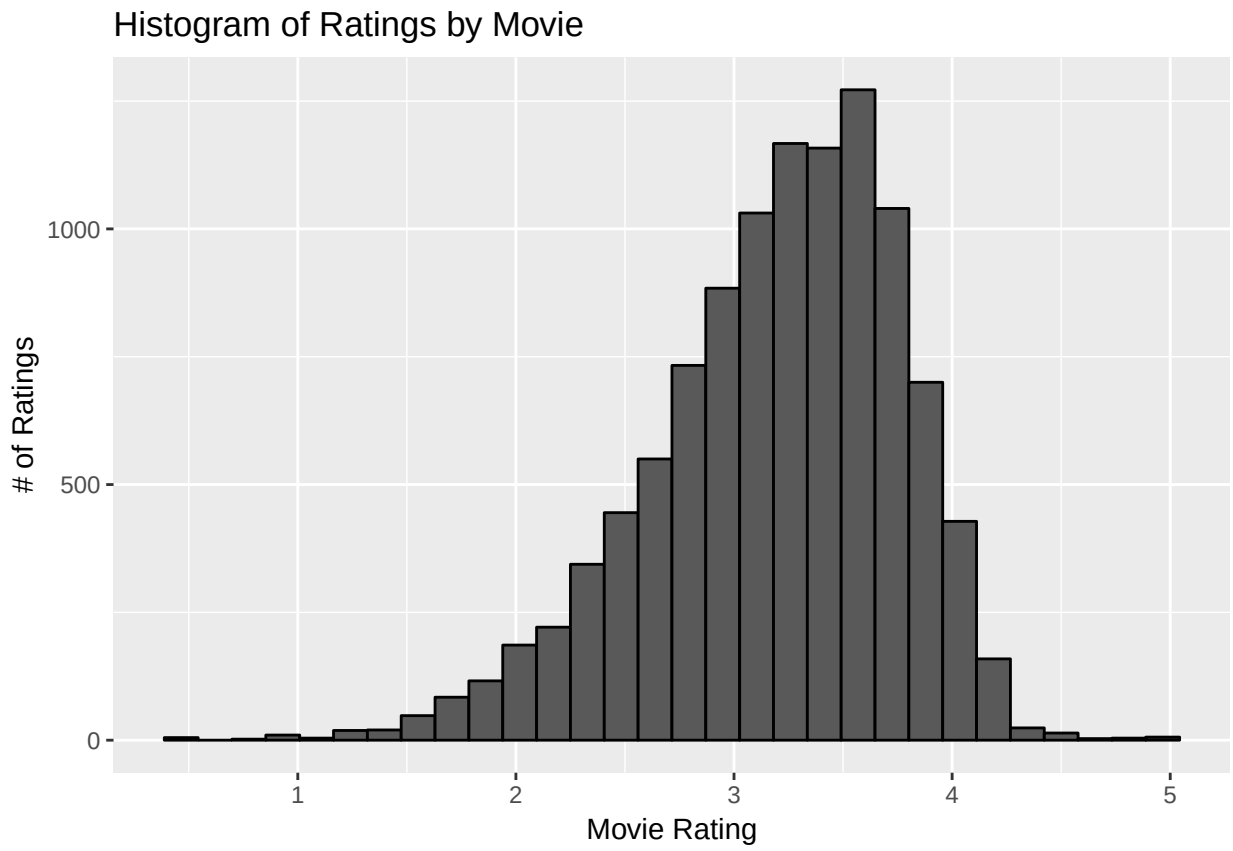
Using just the overall mean (mu), we do surprisingly well considering that we're using the simplest model available. I tried several "advanced" classification methods against this problem and the RMSE was over 3. Granted, classification adds in additional error when using RMSE as the objective function (each prediction gets binned into a 0.5 increment range).

Movie Effect

To enhance the LR model, we will take a look at the rating for each Movie. Plotting the distribution of Movie ratings (below) shows that ratings are not uniformly distributed and that the average of each movie should add predictive capabilities.

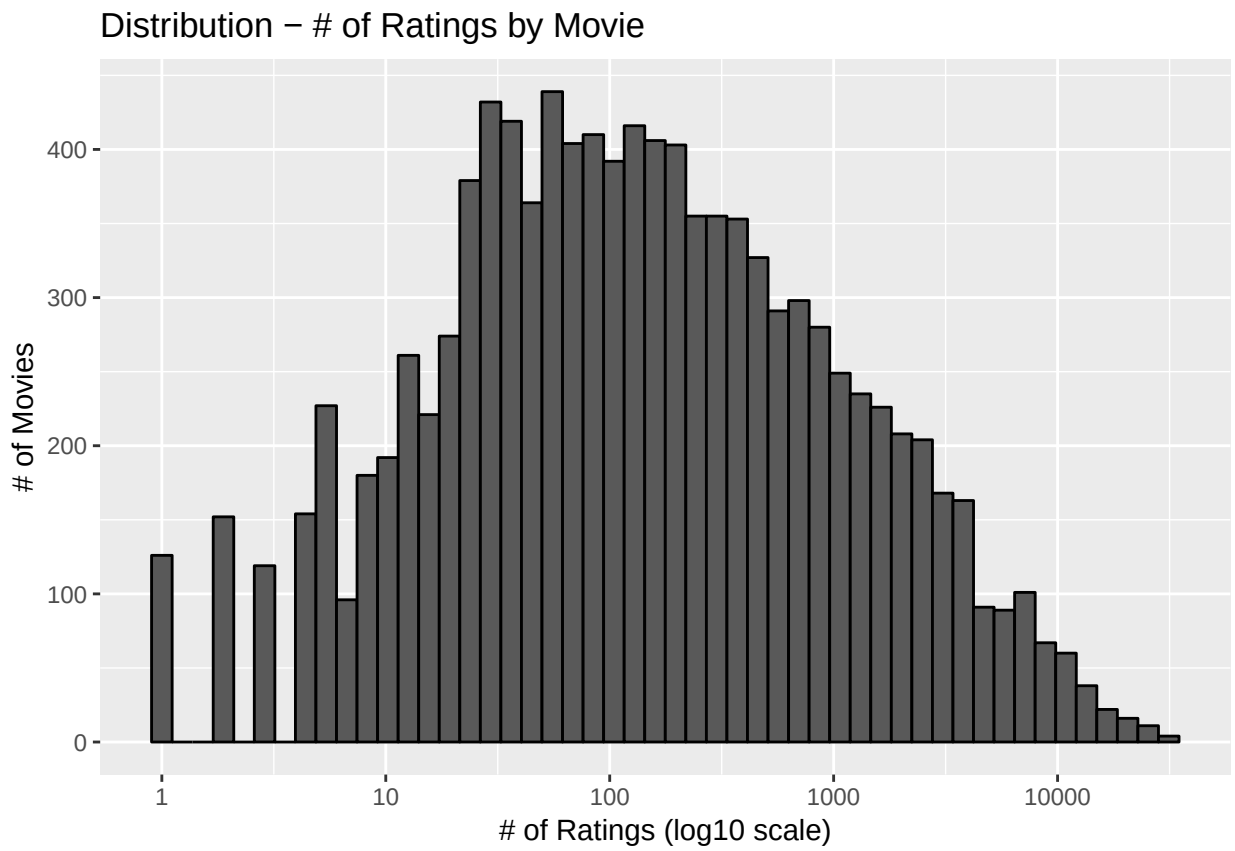
```
# Plot the distribution of Ratings by MovieID in the Training set

train_set %>%
  group_by(movieId) %>%
  summarize(b_u = mean(rating)) %>%
  #filter(n()>=100) %>%
```

```
  ggplot(aes(b_u)) +
  geom_histogram(bins = 30, color = "black") +
  labs(x="Movie Rating", y="# of Ratings") +
  ggtitle("Histogram of Ratings by Movie")
```



The plot below shows that number of Ratings an individual Movie receives varies greatly. This fits intuitively in that there are many popular movies that get seen by millions of people and other movies that are flops and don't get seen much.

```
# Plot the log distribution of # of Ratings per Movie

movie_data %>%
  ggplot(aes(x = m_numrtg)) +
  geom_histogram(bins = 50, color = "black") +
  scale_x_log10() +
  labs(x="# of Ratings (log10 scale)", y="# of Movies") +
  ggtitle("Distribution - # of Ratings by Movie")
```



We will add a term to include the average Movie rating to the general simple mean, thereby adding the "Movie Effect:"

```
#*******************************************************************************
# Linear Regression - Base Average Model + Movie Effect:
#   Yu,i = mu + movie_avgs$b_i
#*******************************************************************************

# Calcualte the average rating for each Movie and subtract the mean
movie_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu))

# Calculate the Predicted ratings on the Test set
predicted_ratings <- mu + test_set %>%
  left_join(movie_avgs, by='movieId') %>%
  pull(b_i)

# Calculate the RMSE
rmse <- RMSE(predicted_ratings, test_set$rating)
LR_rmse_results <- bind_rows(LR_rmse_results,
                             tibble(Method="LR: Movie Effect Model",
                                    RMSE = rmse))
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

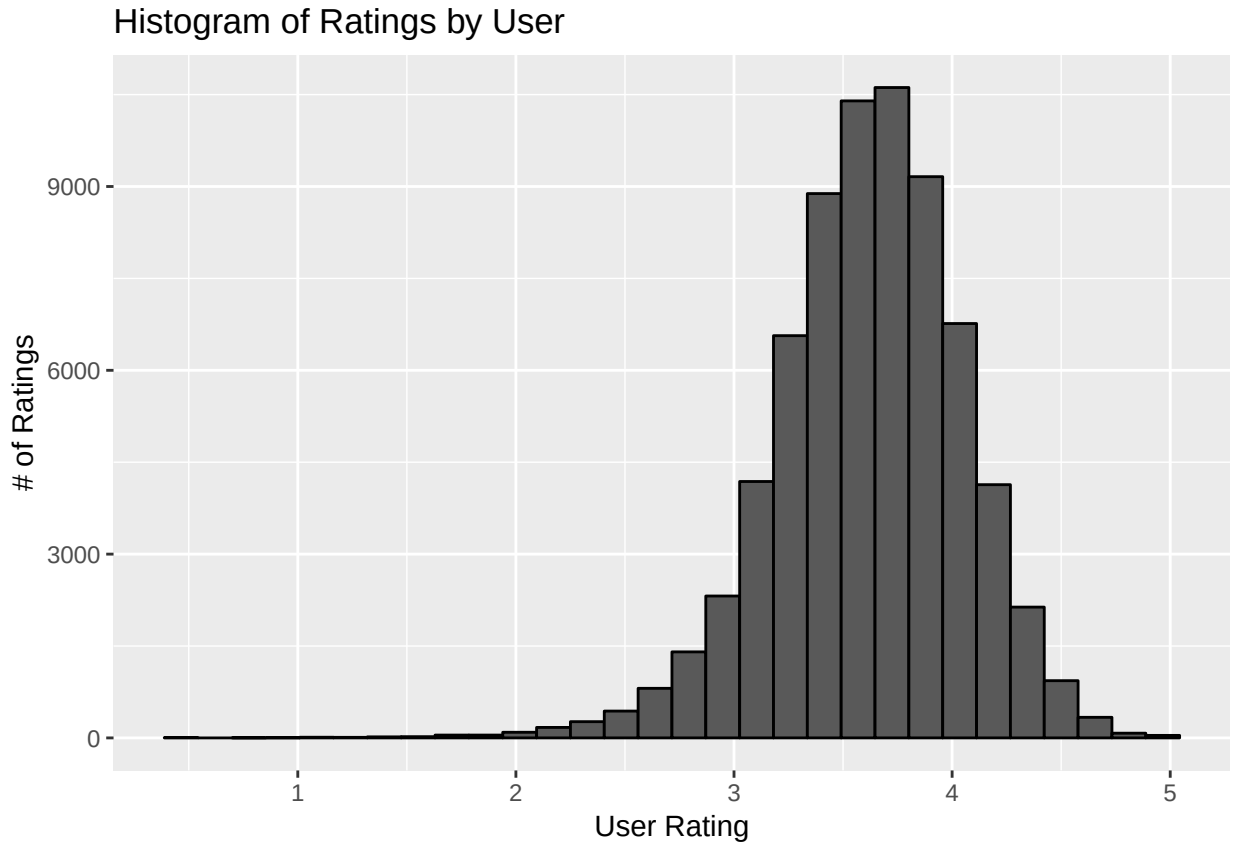
Method	RMSE
LR: Base Mean Model	1.0612018
LR: Movie Effect Model	0.9439087

Adding the Movie average shows improvement, but still isn't better than our target.

User Effect

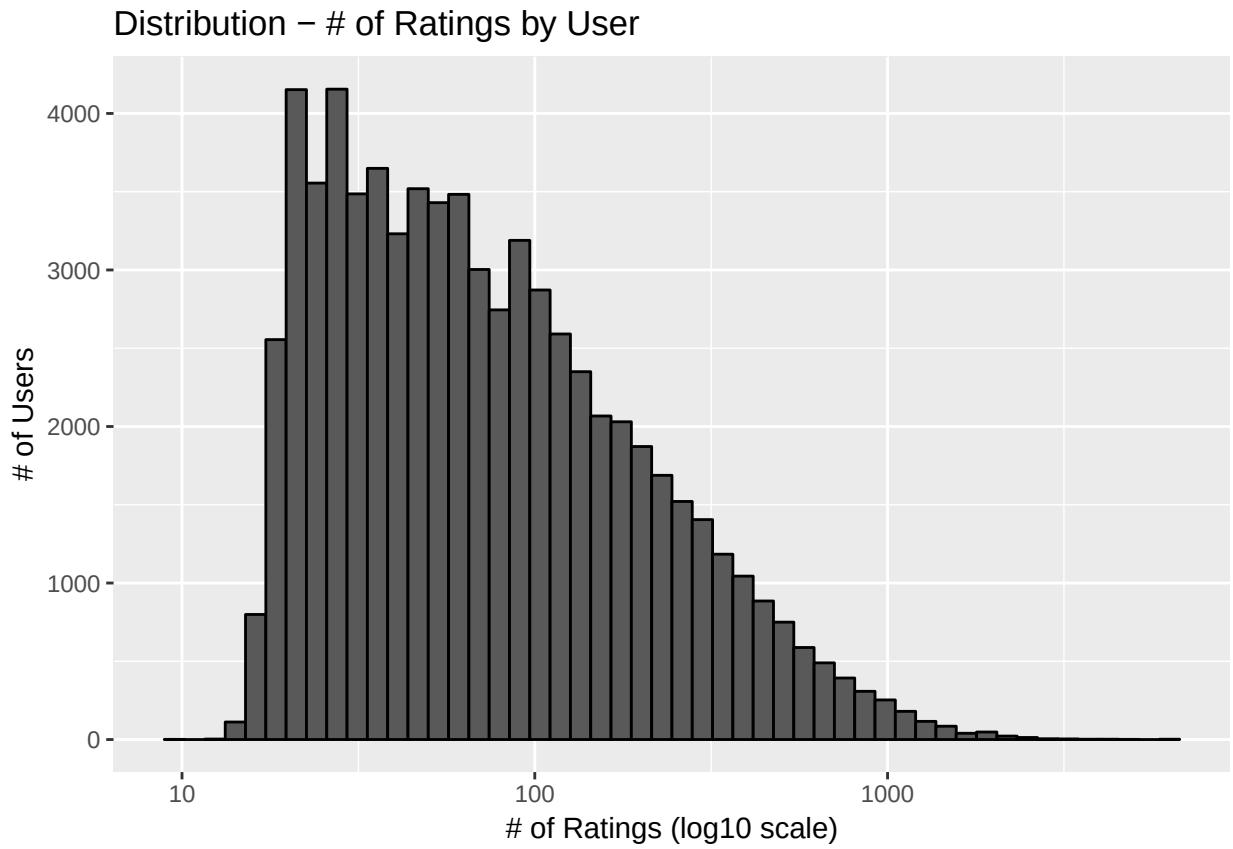
Taking a look at the distribution of User ratings (below) we see that just like Movie ratings, the Ratings by Users varies appreciably.

```
# Plot the distribution of Ratings by User ID in the Training set

train_set %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating)) %>% filter(n()>=100) %>%
  ggplot(aes(b_u)) +
  geom_histogram(bins = 30, color = "black") +
  labs(x="User Rating", y="# of Ratings") +
  ggtitle("Histogram of Ratings by User")
```




```
# Plot the log distribution of # of Ratings per User

user_data %>%
  ggplot(aes(x = u_numrtg)) +
  geom_histogram(bins = 50, color = "black") +
  scale_x_log10() +
  labs(x="# of Ratings (log10 scale)", y="# of Users") +
```

```
  ggtitle("Distribution - # of Ratings by User")
```



We can see that the distribution of Ratings per User also varies with some users rating many more movies than others. This makes sense given that some people are very passionate about movies as well as sharing their opinions and ratings with others.

Next, we will add the "User Effect" by taking the average rating of each user and combining it to the current Mean + Movie model:

```
#*********************************************************************************
# Average + Movie Effect + User Effect:
#   Yu,i = mu + movie_avgs$b_i + user_avgs$b_u
#*********************************************************************************

# Calculate average rating for each User
user_avgs <- train_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))

# Predict ratings on the Test set
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)
```

```r
# Calcualte the RMSE
rmse <- RMSE(predicted_ratings, test_set$rating)
LR_rmse_results <- bind_rows(LR_rmse_results,
                             tibble(Method="LR: Movie + User Effects Model",
                                    RMSE = rmse))
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

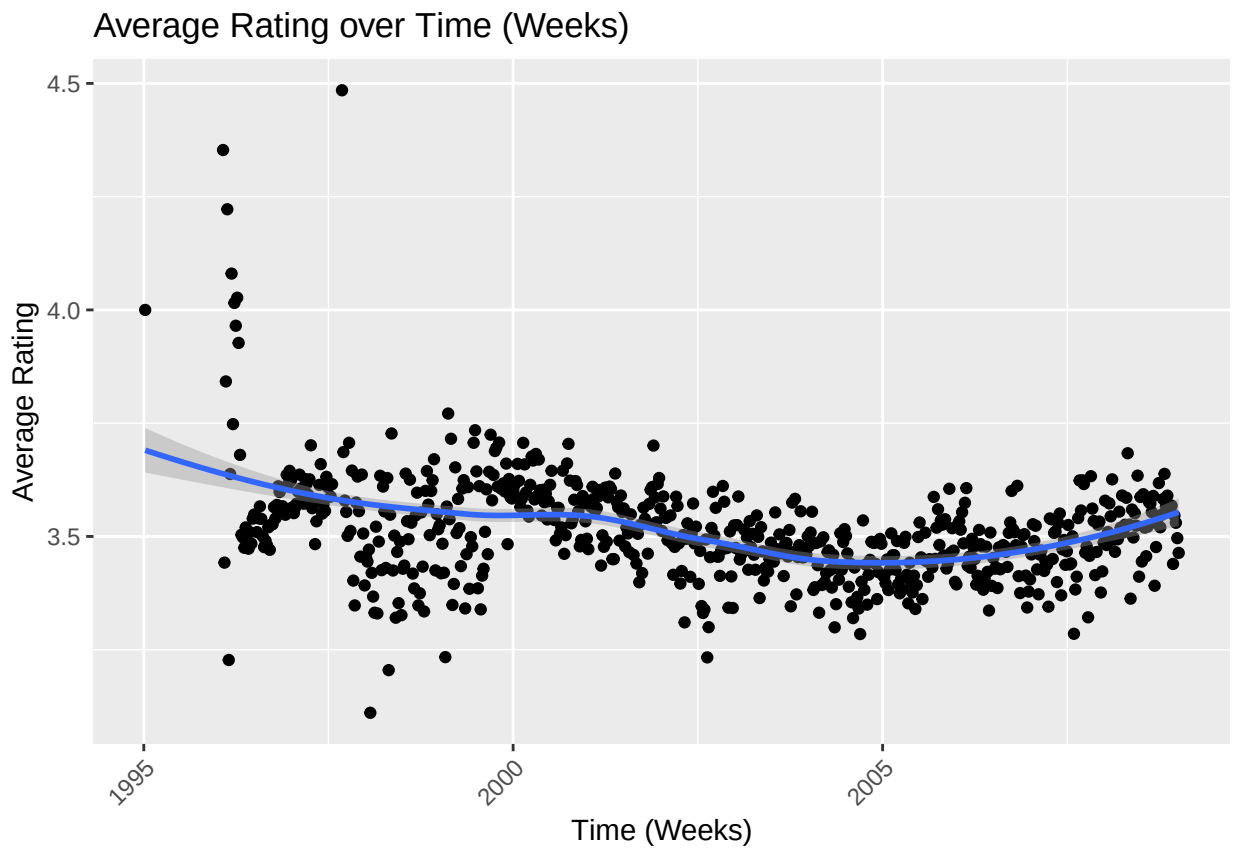
Method	RMSE
LR: Base Mean Model	1.0612018
LR: Movie Effect Model	0.9439087
LR: Movie + User Effects Model	0.8653488

Adding the User mean adjustment to the overall Mean and Movie has reduced the RMSE below the target. We could stop here, but let's add in the effects for the ratings varying by Time as well as ratings varying by Genre.

Time Effect

Converting the **timestamp** variable into a weekly date and plotting the average rating over time, there does appear to be a slight Time Effect.

```r
# Plot the Average Rating by Week

train_set %>%
  mutate(date = round_date(as_datetime(timestamp), unit = "week")) %>%
  group_by(date) %>%
  summarize(rating = mean(rating)) %>%
  ggplot(aes(date, rating)) +
  geom_point() +
  geom_smooth(method="loess") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(x="Time (Weeks)", y="Average Rating") +
  ggtitle("Average Rating over Time (Weeks)")
```



Adding in the Time Effect:

```
##################################################################################
# Average + Movie + User + Time Effect:
#   Yu,i = mu + movie_avgs$b_i + user_avgs$b_u + week_avgs$w_u
##################################################################################

# Calculate the average rating by Week
week_avgs <- train_set %>%
  mutate(week = round_date(as_datetime(timestamp), unit = "week")) %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  group_by(week) %>%
  summarize(w_u = mean(rating - mu - b_i - b_u))

# Predict Ratings on the Test set
predicted_ratings <- test_set %>%
  mutate(week = round_date(as_datetime(timestamp), unit = "week")) %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(week_avgs, by='week') %>%
  mutate(pred = mu + b_i + b_u + w_u) %>%
  pull(pred)

# Evaluate the RMSE
rmse <- RMSE(predicted_ratings, test_set$rating)
LR_rmse_results <- bind_rows(LR_rmse_results,
```

```
                          tibble(Method="LR: Movie + User + Time Effects Model",
                                 RMSE = rmse))
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

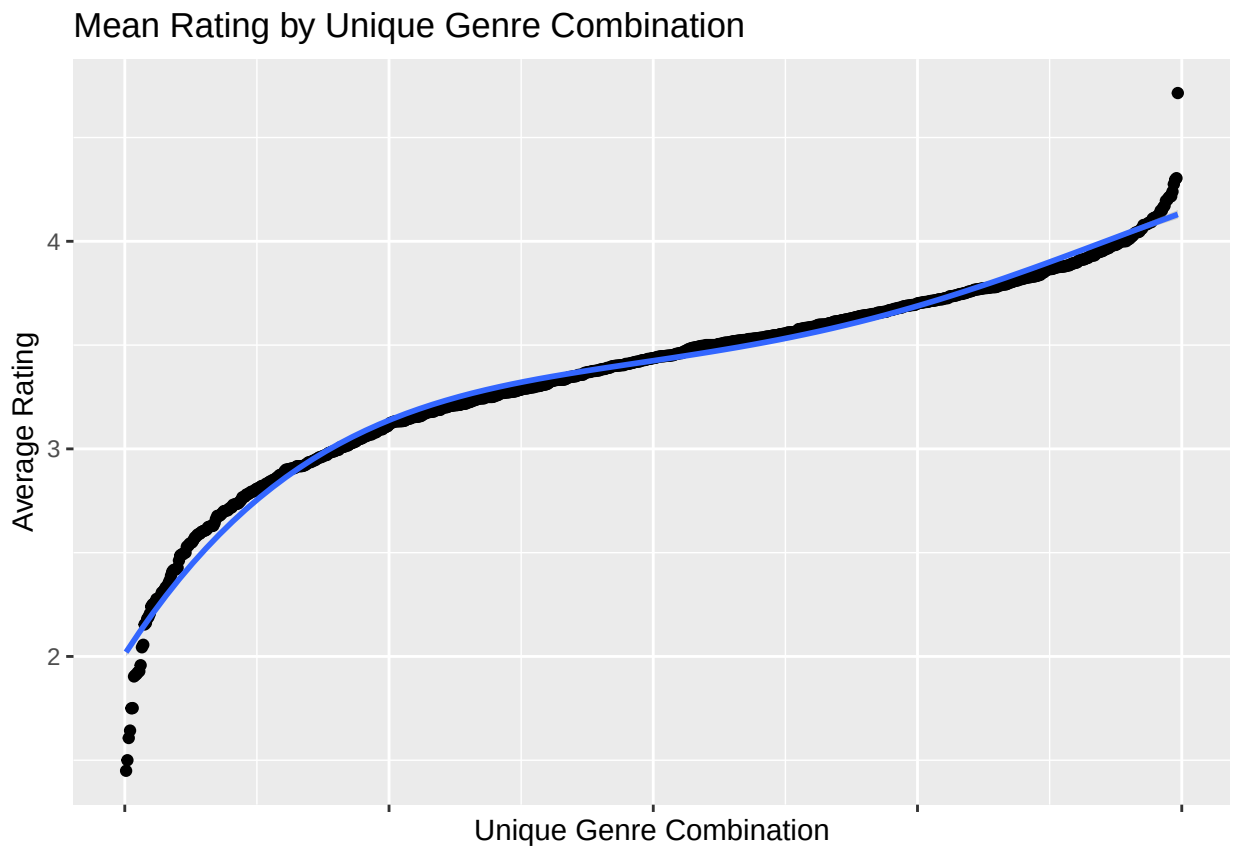
Method	RMSE
LR: Base Mean Model	1.0612018
LR: Movie Effect Model	0.9439087
LR: Movie + User Effects Model	0.8653488
LR: Movie + User + Time Effects Model	0.8652511

Adding the Time Effect further improves the RMSE, but not by much.

Genre Effect

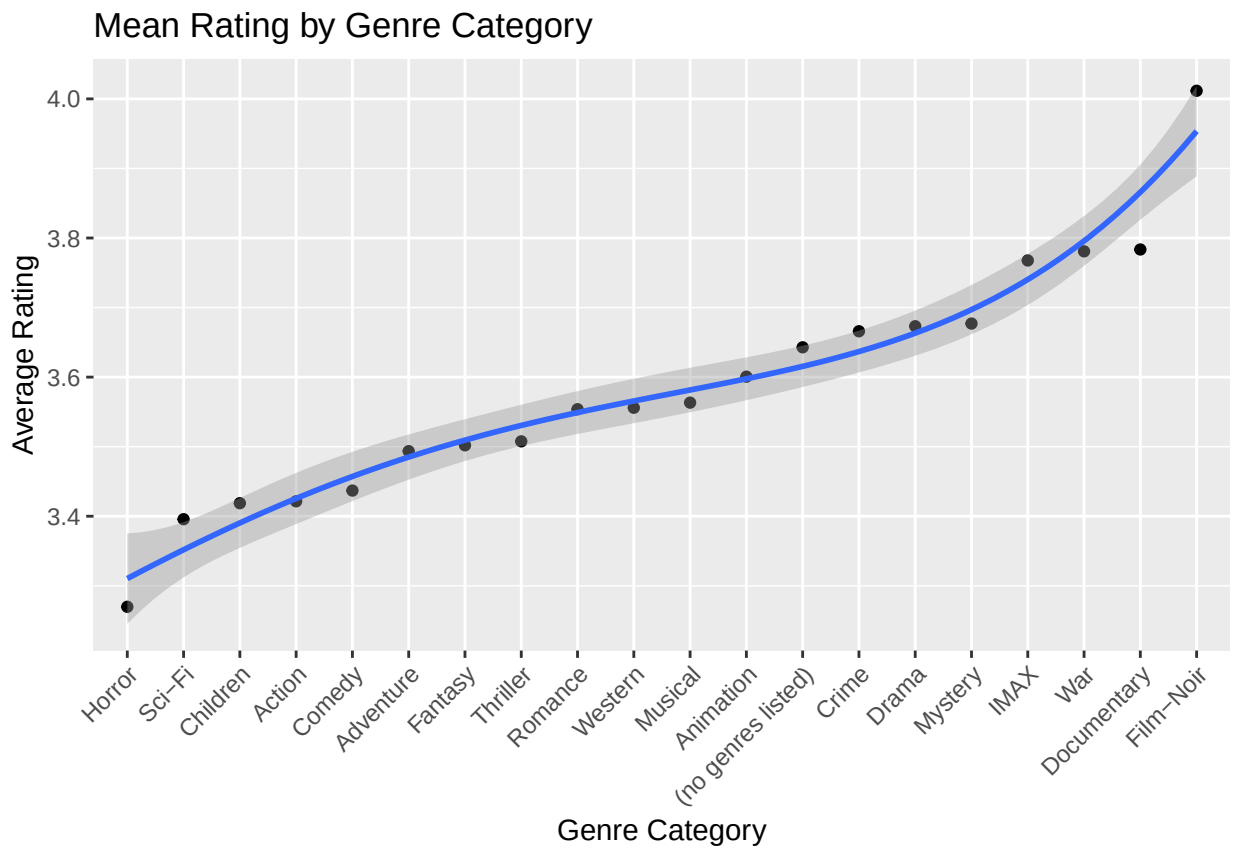
Finally, we will add in the effect for each Genre combination:

```
# Display graph of Average Rating by all unique Genre combinations

train_set %>%
  group_by(genres) %>%
  summarize(avg = mean(rating)) %>%
  ggplot(aes(x = as.numeric(reorder(genres, avg)), y = avg)) +
  geom_point() +
  geom_smooth( aes(x = as.numeric(reorder(genres, avg)), y = avg),
                   method = 'lm', formula = y ~ poly(x, 4), se = TRUE) +
  theme(axis.text.x=element_blank()) +
  labs(x="Unique Genre Combination", y="Average Rating") +
  ggtitle("Mean Rating by Unique Genre Combination")
```



The unique genre combinations (ex: "Drama|Action|Thriller") show a relatively strong variation especially at each tail.

```
# Display graph of each indivudual Genre
# First split out each individual Genre from the genres variable
genre_cats <- edx %>%
  select(genres, rating) %>%
  separate_rows(genres, sep = "\\|") %>%
  group_by(genres) %>%
  summarize(avg = mean(rating)) %>%
  arrange(avg)

# Sort the Genres by factor
genre_cats$genres <- factor(genre_cats$genres, levels = genre_cats$genres[order(genre_cats$avg)])

# Plot the Average per Genre
genre_cats %>% ggplot(aes(genres, avg)) +
  geom_point() +
  geom_smooth(aes(x = as.numeric(genres), y = avg),
              method = 'lm',
              formula = y ~ poly(x, 4),
              se = TRUE) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(x="Genre Category", y="Average Rating") +
  ggtitle("Mean Rating by Genre Category")
```




```r
# Calcualte the average rating of each Genre Combination
genre_avgs <- train_set %>%
  mutate(week = round_date(as_datetime(timestamp), unit = "week")) %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(week_avgs, by='week') %>%
  group_by(genres) %>%
  summarize(g_u = mean(rating - mu - b_i - b_u - w_u))

# Make predictions against the Test set
predicted_ratings <- test_set %>%
  mutate(week = round_date(as_datetime(timestamp), unit = "week")) %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(week_avgs, by='week') %>%
  left_join(genre_avgs, by='genres') %>%
  mutate(pred = mu + b_i + b_u + w_u + g_u) %>%
  pull(pred)

rmse <- RMSE(predicted_ratings, test_set$rating)
LR_rmse_results <- bind_rows(LR_rmse_results,
                          tibble(Method="LR: Movie + User + Time + Genre Effects Model",
                                 RMSE = rmse))
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE
LR: Base Mean Model	1.0612018
LR: Movie Effect Model	0.9439087
LR: Movie + User Effects Model	0.8653488
LR: Movie + User + Time Effects Model	0.8652511
LR: Movie + User + Time + Genre Effects Model	0.8648488

Adding the final Genre Effect further lowers the RMSE value.

However, we should be able to improve the predictions by regularizing the Linear Model.

Regularized Linear Regression Model Creation

Since we've already covered the creation of the base Linear Regression model utilizing the Overall Mean + Movie + User + Time + Genre Effects, this section will simply re-run the code above modified to use Penalized Least Squares in order to reduce the effects of outliers in the data ex: Movies with only a few Ratings that get either very high or very low Ratings. The results will be shown at the bottom.

```
#*******************************************************************************
# Regularized Linear Regression - Base Average Model + Movie Effect:
#   Yu,i = mu + movie_reg_avgs$b_i
#*******************************************************************************

# Calcualte Regularized Movie Effect (b_i)
lambda <- 3
movie_reg_avgs <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n()+lambda), n_i = n())

# Predict against Test Set
predicted_ratings <- test_set %>%
  left_join(movie_reg_avgs, by = "movieId") %>%
  mutate(pred = mu + b_i) %>%
  pull(pred)

# Calcualte RMSE
rmse <- RMSE(predicted_ratings, test_set$rating)
LR_rmse_results <- bind_rows(LR_rmse_results,
                          tibble(Method="Reg LR: Movie Effect Model",
                                 RMSE = rmse))


#*******************************************************************************
# Regularized Linear Regression - Average + Movie Effect + User Effect:
#   Yu,i = mu + b_i + b_u
#*******************************************************************************

# Find the best RMSE across range of Lambdas
lambdas <- seq(5, 5.5, 0.05)
rmses <- sapply(lambdas, function(l){
  mu <- mean(train_set$rating)
  b_i <- train_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))
  b_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
```

```
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))
  predicted_ratings <-
    test_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(pred = mu + b_i + b_u) %>%
    pull(pred)
  return(RMSE(predicted_ratings, test_set$rating))
})

LR_rmse_results <- bind_rows(LR_rmse_results,
                          tibble(Method="Reg LR: Movie + User Effect Model",
                                 RMSE = min(rmses)))


#*****************************************************************************
# Regularized Linear Regression - Average + Movie + User + Time Effect:
#   Yu,i = mu + b_i + b_u + w_u
#*****************************************************************************

# Update Training and Test datasets to include date as Week
train_set <- edx
train_set <- train_set %>% mutate(week = round_date(as_datetime(timestamp), unit = "week"))

test_set <- validation
test_set <- test_set %>% mutate(week = round_date(as_datetime(timestamp), unit = "week"))

# Find best Lambda across range
lambdas <- seq(5.4, 5.6, 0.05)
rmses <- sapply(lambdas, function(l){
  mu <- mean(train_set$rating)
  b_i <- train_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))
  b_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))
  w_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    group_by(week) %>%
    summarize(w_u = sum(rating - b_i - b_u - mu)/(n()+l))
  predicted_ratings <-
    test_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(w_u, by = "week") %>%
    mutate(pred = mu + b_i + b_u + w_u) %>%
    pull(pred)
  return(RMSE(predicted_ratings, test_set$rating))
})
```

```
LR_rmse_results <- bind_rows(LR_rmse_results,
                          tibble(Method="Reg LR: Movie + User + Time Effect Model",
                                 RMSE = min(rmses)))


#*****************************************************************************
# Regularized Linear Regression - Average + Movie + User + Time + Genre Effect:
#   Yu,i = mu + b_i + b_u + w_u + g_u
#*****************************************************************************

# Find best Lambda to minimize RMSE
lambdas <- seq(5.35, 5.55, 0.05)
rmses <- sapply(lambdas, function(l){
  mu <- mean(train_set$rating)
  b_i <- train_set %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n()+l))
  b_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n()+l))
  w_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    group_by(week) %>%
    summarize(w_u = sum(rating - b_i - b_u - mu)/(n()+l))
  g_u <- train_set %>%
    left_join(b_i, by="movieId") %>%
    left_join(b_u, by="userId") %>%
    left_join(w_u, by="week") %>%
    group_by(genres) %>%
    summarize(g_u = sum(rating - b_i - b_u - w_u - mu)/(n()+l))
  predicted_ratings <-
    test_set %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    left_join(w_u, by = "week") %>%
    left_join(g_u, by = "genres") %>%
    mutate(pred = mu + b_i + b_u + w_u + g_u) %>%
    pull(pred)
  return(RMSE(predicted_ratings, test_set$rating))
})

LR_rmse_results <- bind_rows(LR_rmse_results,
                          tibble(Method="Reg LR: Movie + User + Time + Genre Effect Model",
                                 RMSE = min(rmses)))
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE
LR: Base Mean Model	1.0612018
LR: Movie Effect Model	0.9439087
LR: Movie + User Effects Model	0.8653488
LR: Movie + User + Time Effects Model	0.8652511
LR: Movie + User + Time + Genre Effects Model	0.8648488
Reg LR: Movie Effect Model	0.9438538
Reg LR: Movie + User Effect Model	0.8648170
Reg LR: Movie + User + Time Effect Model	0.8646938
Reg LR: Movie + User + Time + Genre Effect Model	0.8643084

```
#*****************************************************
```

As seen from the above results, the Regularizing using Penalized Least Squares improves every version of the model and gives the final "Best RMSE" of **0.8643084**

```
# Clean up Environment
rm(genre_avgs, lambda, lambdas, movie_avgs, movie_reg_avgs, mu, best_rmse_results,
   predicted_ratings, rmse, rmses, user_avgs, week_avgs,  genre_cats, user_data,
   movie_data, test_set, train_set)
```

XGB – Extreme Parallel Tree Boosting (xgBoost)

Since its introduction in 2015, xgBoost, along with Deep Neural Networks, have largely dominated the winning solutions in Kaggle competitions.

xgBoost (Extreme Gradient Boosting) is an R library optimized for tree boosting. Its utilization of multi-threading and regularization deliver very fast (when compared to previous Random Forest methods) and accurate predictions. xgBoost is similar to other gradient boosting frameworks but focused on parallel computation on a single machine along with efficient memory usage.

xgBoost is also very flexible, implementing both linear models and tree learning algorithms. It also supports many different objective functions, including regression, classification and ranking.

xgBoost Data Preparation

xgBoost only works with numeric vectors so we will be massaging the Training and Testing datasets by converting the categorical **genres** variable into a set of One-Hot Encoded variables. Since the Time Effect was relatively weak in the Regression models, **timestamp** is dropped along with the Movie **title** since it does not add any predictive value and the **movieId** has already provided a numeric encoding.

Create Training and Test Datasets

```
# Load required libraries
if (!require(xgboost)) install.packages('xgboost')
```

```
## Loading required package: xgboost
```

```
##
## Attaching package: 'xgboost'
```

```
## The following object is masked from 'package:dplyr':
##
##     slice
```

```
library(xgboost)

if (!require(data.table)) install.packages('data.table')
```

```
## Loading required package: data.table
##
## Attaching package: 'data.table'
## The following objects are masked from 'package:lubridate':
##
##     hour, isoweek, mday, minute, month, quarter, second, wday,
##     week, yday, year
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
## The following object is masked from 'package:purrr':
##
##     transpose
```

```r
library(data.table)

# Set seed for random sample (if needed)
#set.seed(1, sample.kind = "Rounding")

# Create new Training and Test datasets
# If you don't have at least 64Gb RAM, then use the 30% random sample below

###!!!!!  Check if this runs within 8Gb RAM

#train_set <- edx %>% select(-one_of("timestamp", "title")) %>% mutate(userId = as.numeric(userId)) %>%

train_set <- edx %>% select(-one_of("timestamp", "title")) %>% mutate(userId = as.numeric(userId))
test_set <- validation %>% select(-one_of("timestamp", "title")) %>% mutate(userId = as.numeric(userId))
```

One-Hot Encode Movie Genres

Here we turn the concatenated **genres** variable into individual columns "one-hot" encoded:

- 1 = Movie is classified as belonging to that Genre
- 0 = Movie is NOT in that Genre

Variables Added: *g_Action, g_Adventure, g_Animation, g_Children, g_Comedy, g_Crime, g_Documentary, g_Drama, g_Fantasy, g_FilmNoir, g_Horror, g_IMAX, g_Musical, g_Mystery, g_Romance, g_SciFi, g_Thriller, g_War, g_Western*

We also add one variable called *g_Count* that sums the number of Genres for each movie. Example: Drama|Action|Thriller = 3

```r
# One-hot encoding of Genres in the Training Set
train_set <- train_set %>% mutate(g_Action = if_else(grepl("Action", genres), 1, 0),
                                  g_Adventure = if_else(grepl("Adventure", genres), 1, 0),
                                  g_Animation = if_else(grepl("Animation", genres), 1, 0),
                                  g_Children = if_else(grepl("Children", genres), 1, 0),
                                  g_Comedy = if_else(grepl("Comedy", genres), 1, 0),
                                  g_Crime = if_else(grepl("Crime", genres), 1, 0),
                                  g_Documentary = if_else(grepl("Documentary", genres), 1, 0),
                                  g_Drama = if_else(grepl("Drama", genres), 1, 0),
                                  g_Fantasy = if_else(grepl("Fantasy", genres), 1, 0),
                                  g_FilmNoir = if_else(grepl("Film-Noir", genres), 1, 0),
```

```
                                  g_Horror = if_else(grepl("Horror", genres), 1, 0),
                                  g_IMAX = if_else(grepl("IMAX", genres), 1, 0),
                                  g_Musical = if_else(grepl("Musical", genres), 1, 0),
                                  g_Mystery = if_else(grepl("Mystery", genres), 1, 0),
                                  g_Romance = if_else(grepl("Romance", genres), 1, 0),
                                  g_SciFi = if_else(grepl("Sci-Fi", genres), 1, 0),
                                  g_Thriller = if_else(grepl("Thriller", genres), 1, 0),
                                  g_War = if_else(grepl("War", genres), 1, 0),
                                  g_Western = if_else(grepl("Western", genres), 1, 0))

# Add column which is a total count of the number of Genres
train_set <- train_set %>% mutate(g_Count = rowSums(select(.,g_Action:g_Western) != 0))

# One-hot encoding of Genres in the Test Set
test_set <- test_set %>% mutate(g_Action = if_else(grepl("Action", genres), 1, 0),
                                g_Adventure = if_else(grepl("Adventure", genres), 1, 0),
                                g_Animation = if_else(grepl("Animation", genres), 1, 0),
                                g_Children = if_else(grepl("Children", genres), 1, 0),
                                g_Comedy = if_else(grepl("Comedy", genres), 1, 0),
                                g_Crime = if_else(grepl("Crime", genres), 1, 0),
                                g_Documentary = if_else(grepl("Documentary", genres), 1, 0),
                                g_Drama = if_else(grepl("Drama", genres), 1, 0),
                                g_Fantasy = if_else(grepl("Fantasy", genres), 1, 0),
                                g_FilmNoir = if_else(grepl("Film-Noir", genres), 1, 0),
                                g_Horror = if_else(grepl("Horror", genres), 1, 0),
                                g_IMAX = if_else(grepl("IMAX", genres), 1, 0),
                                g_Musical = if_else(grepl("Musical", genres), 1, 0),
                                g_Mystery = if_else(grepl("Mystery", genres), 1, 0),
                                g_Romance = if_else(grepl("Romance", genres), 1, 0),
                                g_SciFi = if_else(grepl("Sci-Fi", genres), 1, 0),
                                g_Thriller = if_else(grepl("Thriller", genres), 1, 0),
                                g_War = if_else(grepl("War", genres), 1, 0),
                                g_Western = if_else(grepl("Western", genres), 1, 0))

# Add column which is a total count of the number of Genres
test_set <- test_set %>% mutate(g_Count = rowSums(select(.,g_Action:g_Western) != 0))
```

Additional Movie & User Derived Variables

In addition to the One-Hot Genre variables, we are also going to be adding additional information about each User and Movie. We will be adding:

- User: Average (u_avg), StDev (u_std), Number of Reviews (u_numrtg)
- Movie: Average (m_avg), StDev (m_std), Number of Reviews (m_numrtg)

```
# Create dataframe with additional User rating information:
#       Average (u_avg), StDev (u_std), Number of Reviews (u_numrtg)
user_data <- train_set %>%
  group_by(userId) %>%
  summarize(u_avg = mean(rating),
            u_std = sd(rating),
            u_numrtg = as.numeric(n()))

# Create dataframe with additional Movie rating information:
```

```
#       Average (m_avg), StDev (m_std), Number of Reviews (m_numrtg)
movie_data <- train_set %>%
  group_by(movieId) %>%
  summarize(m_avg = mean(rating),
            m_std = sd(rating),
            m_numrtg = as.numeric(n()))

# Merge User & Movie derived fields into the Training and Test datasets
train_set <- train_set %>% left_join(user_data, by = "userId")
train_set <- train_set %>% left_join(movie_data, by = "movieId")
test_set <- test_set %>% left_join(user_data, by = "userId")
test_set <- test_set %>% left_join(movie_data, by = "movieId")
```

Now that the Training & Test datasets have all of the additional derived variables included, the sets must be split to provide the data and the target "label" data tables. xgBoost uses its own Data Matrix format called xgb.DMatrix:

```
# xgBoost requires that the target variable, referred to as the "label" (rating) be in a separate data
# Here we create a Label and Data for the Training and Testing sets
# We are also dropping the Genres variable since it isn't numeric and no longer needed due to the One-H

train_data <- train_set %>% select(-one_of("rating", "genres")) %>% as.data.table()
train_label <- train_set %>% select("rating") %>% as.data.table()
test_data <- test_set %>% select(-one_of("rating", "genres")) %>% as.data.table()
test_label <- test_set %>% select("rating") %>% as.data.table()

# Create the xgBoost Training & Testing matricies
train_matrix = xgb.DMatrix(as.matrix(train_data), label=as.matrix(train_label))
test_matrix = xgb.DMatrix(as.matrix(test_data), label=as.matrix(test_label))
```

xgBoost - XGB Model Creation

xgBoost allows the creation of both Linear, Tree, and Mixed Linear & Tree based models. xgBoost can also create Classification trees (binary and multi-class) by changing the objective function. While xgBoost is very fast when compared to other methods, the training times needed for anything, but a simple linear model easily extend into the 30 minute to 10-20 hour range.

Because of the required training times, we will only be creating one small model:

- XGB Linear 50 Boost Rnds

We will also present the RMSE results and training times for more complex models.

```
##########################################################################################
# Model: XGB Linear 50 Boost Rnds
##########################################################################################
# Set paramaters for most basic Linear tree
# These settings should train in less than 1 MINUTE with 50 Boosting Rounds

xgb_params <- list(booster = "gblinear",      # Linear boosting alg
                   objective = "reg:linear",  # Linear regression (default)
                   eval_metric = "rmse",      # rmse as objective function
                   verbosity = 3,             # Highest verbosr level - shows all debug info
                   silent = 0)                # Not silent
```

Train the xgBoost Linear model

```r
# Train the XGB tree using the currently set xgb_params
start_time <- Sys.time()                    # Record start time
xgb_model <- xgboost(params = xgb_params,
                     data = train_matrix,
                     nrounds = 50,          # Maximum number of Boosting Rounds
                     verbose = 2)           # Display pogress during Training
```

```
## [1]  train-rmse:1.165194
## [2]  train-rmse:1.040609
## [3]  train-rmse:1.000653
## [4]  train-rmse:0.980639
## [5]  train-rmse:0.968543
## [6]  train-rmse:0.960254
## [7]  train-rmse:0.954154
## [8]  train-rmse:0.949479
## [9]  train-rmse:0.945818
## [10] train-rmse:0.942905
## [11] train-rmse:0.940543
## [12] train-rmse:0.938571
## [13] train-rmse:0.936880
## [14] train-rmse:0.935382
## [15] train-rmse:0.934017
## [16] train-rmse:0.932742
## [17] train-rmse:0.931533
## [18] train-rmse:0.930374
## [19] train-rmse:0.929254
## [20] train-rmse:0.928176
## [21] train-rmse:0.927131
## [22] train-rmse:0.926124
## [23] train-rmse:0.925148
## [24] train-rmse:0.924207
## [25] train-rmse:0.923300
## [26] train-rmse:0.922427
## [27] train-rmse:0.921591
## [28] train-rmse:0.920790
## [29] train-rmse:0.920015
## [30] train-rmse:0.919276
## [31] train-rmse:0.918565
## [32] train-rmse:0.917881
## [33] train-rmse:0.917231
## [34] train-rmse:0.916599
## [35] train-rmse:0.915998
## [36] train-rmse:0.915421
## [37] train-rmse:0.914861
## [38] train-rmse:0.914326
## [39] train-rmse:0.913813
## [40] train-rmse:0.913314
## [41] train-rmse:0.912832
## [42] train-rmse:0.912369
## [43] train-rmse:0.911923
## [44] train-rmse:0.911502
## [45] train-rmse:0.911070
## [46] train-rmse:0.910667
## [47] train-rmse:0.910280
```

```
## [48] train-rmse:0.909905
## [49] train-rmse:0.909542
## [50] train-rmse:0.909188
```

```
finish_time <- Sys.time()                    # Record finish time
finish_time - start_time                     # Display total training time
```

```
## Time difference of 25.62506 secs
```

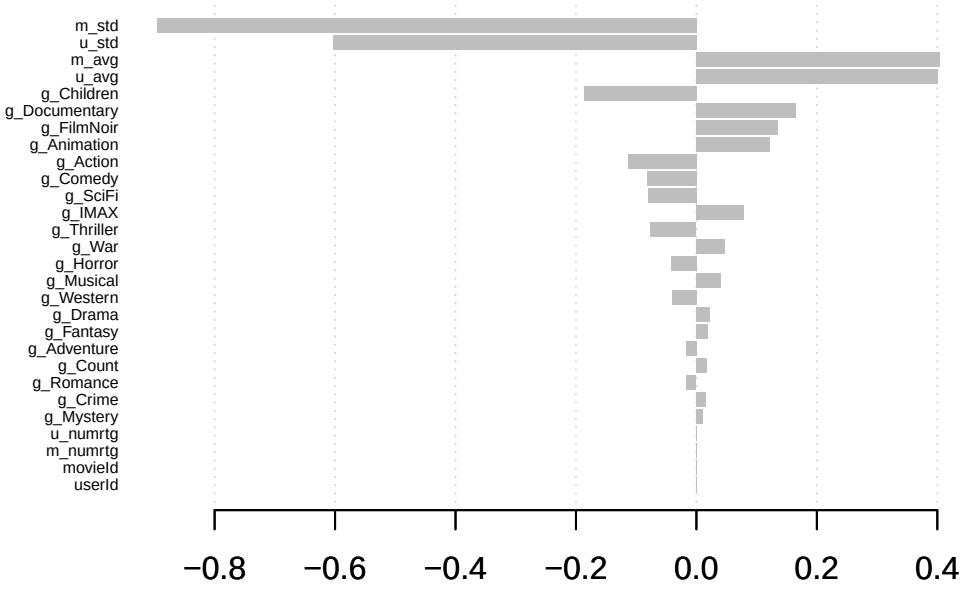
Use the trained model to predict the Test set and report the RMSE results.

```
# Use the trained model to predict the Test dataset
test_pred <- as.data.frame(predict(xgb_model , newdata = test_matrix))

# Calculate the RMSE of the Predictions
rmse <- RMSE(test_label$rating, test_pred$`predict(xgb_model, newdata = test_matrix)`)

XGB_rmse_results <- tibble(Method="XGB Linear 50 Boost Rnds",
                           RMSE = rmse,
                           Train_Time = paste(as.character(round(as.numeric(finish_time - start_time, u
kable(XGB_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
XGB Linear 50 Boost Rnds	0.9149801	25.63 secs

Once the Model is trained we can see the feature importance:

```
# Calculate and Plot the Feature Importances for the XG

# Compute feature importance matrix
importance_matrix <- xgb.importance(model = xgb_model)

# Plot the Feature Importance of the Top 10 features
xgb.plot.importance(importance_matrix)
```



Add results of the additional models to the RMSE results matrix:

```
# Create Table of Additional Model Results
XGB_rmse_results <- bind_rows(XGB_rmse_results,
                              tibble(Method="XGB Linear 1000 Boost Rnds",
                                     RMSE = 0.8820874,
                                     Train_Time = "8.131121 mins"))
XGB_rmse_results <- bind_rows(XGB_rmse_results,
                              tibble(Method="XGB Mixed Tree 5 Boost Rnds",
                                     RMSE = 0.8690746,
                                     Train_Time = "24.74677 mins"))
XGB_rmse_results <- bind_rows(XGB_rmse_results,
                              tibble(Method="XGB Mixed Tree 50 Boost Rnds",
                                     RMSE = 0.8539179,
                                     Train_Time = "20.66 HOURS"))
kable(XGB_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
XGB Linear 50 Boost Rnds	0.9149801	25.63 secs
XGB Linear 1000 Boost Rnds	0.8820874	8.131121 mins
XGB Mixed Tree 5 Boost Rnds	0.8690746	24.74677 mins
XGB Mixed Tree 50 Boost Rnds	0.8539179	20.66 HOURS

xgBoost - Observations and Additional Comments

Despite the ability to use a full Training set including many additional derived parameters, the xgBoost

models were only able to outperform the Linear Regression Models when using 50 Boosting Rounds at an expense of almost 21 HOURS of Training time on a desktop system with an Intel Quad-Core i7-6700 CPU with 64Gb of RAM.

If you would like to reproduce the result above, use the below training parmanters and number of boosting rounds:

```
#*****************************************************************************************
# Model: XGB Linear 1000 Boost Rnds
#*****************************************************************************************
# Use the same xgb_params as XGB Linear 50, but train it with 1000 boosting rounds


#*****************************************************************************************
# Model: XGB Mixed Tree 5 Boost Rnds
#*****************************************************************************************
# Use the these xgb_params and train with 5 boosting rounds
# NOTE: Training time of 25+ minutes depending upon your system

# xgb_params <- list(booster = "gbtree",
#                    objective = "reg:linear",
#                    colsample_bynode = 0.8,
#                    learning_rate = 1,
#                    max_depth = 10,
#                    num_parallel_tree = 25,
#                    subsample = 0.8,
#                    verbosity = 3,
#                    silent = 0)

#*****************************************************************************************
# Model: XGB Mixed Tree 50 Boost Rnds
#*****************************************************************************************
# Use the same xgb_params as XGB Mixed Tree 5 Boost Rnds above, but train it with 50 boosting rounds
# NOTE: Training time of 20+ HOURS depending upon your system
```

Clean up the Environment before moving to the next model.

```
rm(finish_time, importance_matrix, movie_data, rmse, start_time, test_data, test_label,
   test_matrix, test_pred, test_set, train_data, train_label, train_matrix, train_set,
   user_data, xgb_model, xgb_params)
```

Recosystem - Matrix Factorization w/ Parallel Stochastic Gradient Descent

While we had success with xgBoost, there are several "Recommender" packages available in the R ecosystem focused specifically on Collaborative Filtering. Within the Recommender system, the main task is to predict unknown entries in a rating matrix composed of Users vs Items.

One technique used to solve the recommender problem is matrix factorization where the rating matrix is successively approximated by the product of two matrices of lower dimensions.

Recosystem is a wrapper for the open-source LibFM library that implements optimized and parallelized matrix factorization. The recosystem library also provides easy to use functions to perform parameter tuning across a grid of parameter options, training against an in-memory or on-disk dataset using the tuned parameters, and prediction of results against a test set.

RECO - Data Preparation

Recosystem only uses three variables:

- **Rating (rating):** Rating score provided by User for Movie
- **User (userId):** User ID associated with the Rating
- **Movie (movieId):** Movie ID associated with the Rating

Create Training and Test Datasets

Given the few required variables, the datasets are quite small and should easily fit into system memory. We will only be keeping the **rating**, **userId**, and **movieId** columns:

```
# Load required libraries
if (!require(recosystem)) install.packages('recosystem')
```

```
## Loading required package: recosystem
```

```
library(recosystem)

# Create new Training and Testing datasets. Recosystem only uses three columns: Rating, UserId, and Mov
train_set <- edx %>% select(-one_of("genres","title","timestamp")) %>% as.matrix()
test_set <-  validation %>% select(-one_of("genres","title","timestamp")) %>% as.matrix()
```

Create the Recosystem Object

```
# Create the Reco Recommender object
r = Reco()
```

Create the Tuning Parameter Grid

One very nice feature of Recosystem, besides it's speed, is that tuning the training parameters is very easy and the overall best performing tuned parameters are stored directly in the Reco object making calling them against the Training set simple.

Below we will create a small Tuning grid as an example. This tuning grid only tunes between 2 values of **dim** (20 & 30).

NOTE: We can use nthread = X in the parameter list to define the number of threads to use. If omitted, Recosystem will try to use the maximum available. I used nthread = 8

```
#*****************************************************************************
# Tuning Grid V1 - We will only tune with two values of Dim (20 & 30)
#*****************************************************************************

opts_list = list(dim      = c(20, 30),  # Number of Latent Features
                 costp_l1 = c(0),       # L1 regularization cost for User factors
                 costp_l2 = c(0.01),    # L2 regularization cost for User factors
                 costq_l1 = c(0),       # L1 regularization cost for Movie factors
                 costq_l2 = c(0.1),     # L2 regularization cost for Movie factors
                 lrate    = c(0.1),     # Learning Rate - Aprox step size in Gradient Descent
                 niter    = 10,         # Number of Iterations for Training (Not used in Tuning)
                 nfolds   = 5,          # Number of Folds for CV in Tuning
                 verbose  = FALSE,      # Don't Show Progress
                 nthread  = 1)          #!!! Can be set to higher values for Tuning, but MUST be set to
                                        #    for Training or the results are not reproducible
```

Tune the Model using the Parameter Grid

As noted, if you want, you can use the **nthread** paramater, but it can't be used as a Training parameter. It requires additional investigation, but **nthread** must be set to 1 during Training or the results cannot be reproduced. There must be some randomization in the parallelization that does not rely on setting the Seed value.

Next, we will tune the model given the parameter grid above:

```
#*****************************************************************************
# TUNE the Model - This will take around 2 minutes if executed
#*****************************************************************************

start <- Sys.time()

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

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
opts_tune = r$tune(data_memory(train_set[, 1], train_set[, 2], train_set[, 3]),
                   opts = opts_list)

finish <- Sys.time()
tune_time <- finish - start
tune_time
```

```
## Time difference of 1.865986 mins
```

```
opts_tune$min
```

```
## $dim
## [1] 30
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0.01
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.1
##
## $lrate
## [1] 0.1
##
## $loss_fun
## [1] 0.7980156
```

We see that the Tuned options chose 30 for the **dim** variable. Many iterations were performed to optimize the other model parameters. For the sake of time, we won't perform additional optimizations, but we will look later at the results of using Tuning and 5-fold Cross-Validation in order to find a semi-optimal number of Factors (__dim__)

Train the Model using the Best Parameters

Once the parameters have been optimized, we Train the model:

```
#*****************************************************************************
# TRAIN the Model over 50 Iterations
#*****************************************************************************
start <- Sys.time()

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

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
r$train(data_memory(train_set[, 1], train_set[, 2], train_set[, 3]),
        opts = c(opts_tune$min,
                 niter    = 50,     # Train over 50 Iterations
                 nthread = 1))      # Must be set to 1 for training
```

```
## [12:14:20] ======== Monitor: Learner ========
## [12:14:20] EvalOneIter: 0.585234s, 50 calls @ 11704us
## [12:14:20] GetGradient: 0.631294s, 50 calls @ 12625us
## [12:14:20] PredictRaw: 0.24759s, 50 calls @ 4951us
## [12:14:20] UpdateOneIter: 25.0049s, 50 calls @ 500098us
## [12:14:20] ======== Monitor: GBLinear ========
## [12:14:20] DoBoost: 24.126s, 50 calls @ 482520us
## [12:14:20] PredictBatch: 0.41116s, 101 calls @ 4070us
## [12:14:20] PredictBatchInternal: 4.43905s, 52 calls @ 85366us
## iter      tr_rmse          obj
##    0       0.9728  1.2038e+007
##    1       0.8743  9.9168e+006
##    2       0.8400  9.1975e+006
##    3       0.8168  8.7618e+006
##    4       0.8001  8.4699e+006
##    5       0.7879  8.2666e+006
##    6       0.7782  8.1091e+006
##    7       0.7703  7.9908e+006
##    8       0.7638  7.8966e+006
##    9       0.7581  7.8206e+006
##   10       0.7531  7.7527e+006
##   11       0.7488  7.6981e+006
##   12       0.7448  7.6488e+006
##   13       0.7412  7.6066e+006
##   14       0.7378  7.5673e+006
##   15       0.7347  7.5337e+006
##   16       0.7318  7.5029e+006
##   17       0.7292  7.4756e+006
##   18       0.7268  7.4509e+006
##   19       0.7245  7.4269e+006
##   20       0.7224  7.4071e+006
##   21       0.7204  7.3876e+006
##   22       0.7185  7.3703e+006
##   23       0.7168  7.3543e+006
##   24       0.7151  7.3386e+006
##   25       0.7136  7.3237e+006
##   26       0.7122  7.3113e+006
##   27       0.7109  7.2996e+006
```

```
##   28       0.7096  7.2882e+006
##   29       0.7083  7.2778e+006
##   30       0.7072  7.2663e+006
##   31       0.7061  7.2576e+006
##   32       0.7051  7.2498e+006
##   33       0.7041  7.2415e+006
##   34       0.7032  7.2335e+006
##   35       0.7023  7.2251e+006
##   36       0.7015  7.2192e+006
##   37       0.7007  7.2133e+006
##   38       0.6999  7.2070e+006
##   39       0.6991  7.1995e+006
##   40       0.6985  7.1955e+006
##   41       0.6978  7.1891e+006
##   42       0.6971  7.1841e+006
##   43       0.6965  7.1796e+006
##   44       0.6959  7.1743e+006
##   45       0.6954  7.1694e+006
##   46       0.6948  7.1653e+006
##   47       0.6943  7.1617e+006
##   48       0.6938  7.1576e+006
##   49       0.6933  7.1548e+006
```

```
finish <- Sys.time()
train_time <- finish - start
train_time
```

```
## Time difference of 1.230515 mins
```

Predict the Test Set

After training, we use the trained model to predict the Ratings in the Test Set and calculate the RMSE:

```
#*********************************************************************************
# PREDICT the Test Ratings
#*********************************************************************************
start <- Sys.time()

pred <- r$predict(data_memory(test_set[, 1], test_set[, 2], test_set[, 3]), out_memory())

finish <- Sys.time()
pred_time <- finish - start
pred_time
```

```
## Time difference of 1.022871 mins
```

Calculate the RMSE

```
#*********************************************************************************
# Calculate the RMSE
#*********************************************************************************

rmse <- RMSE(test_set[,3],pred)

RECO_rmse_results <- tibble(Method="V1 - 30 x 50",
                            RMSE = rmse,
```

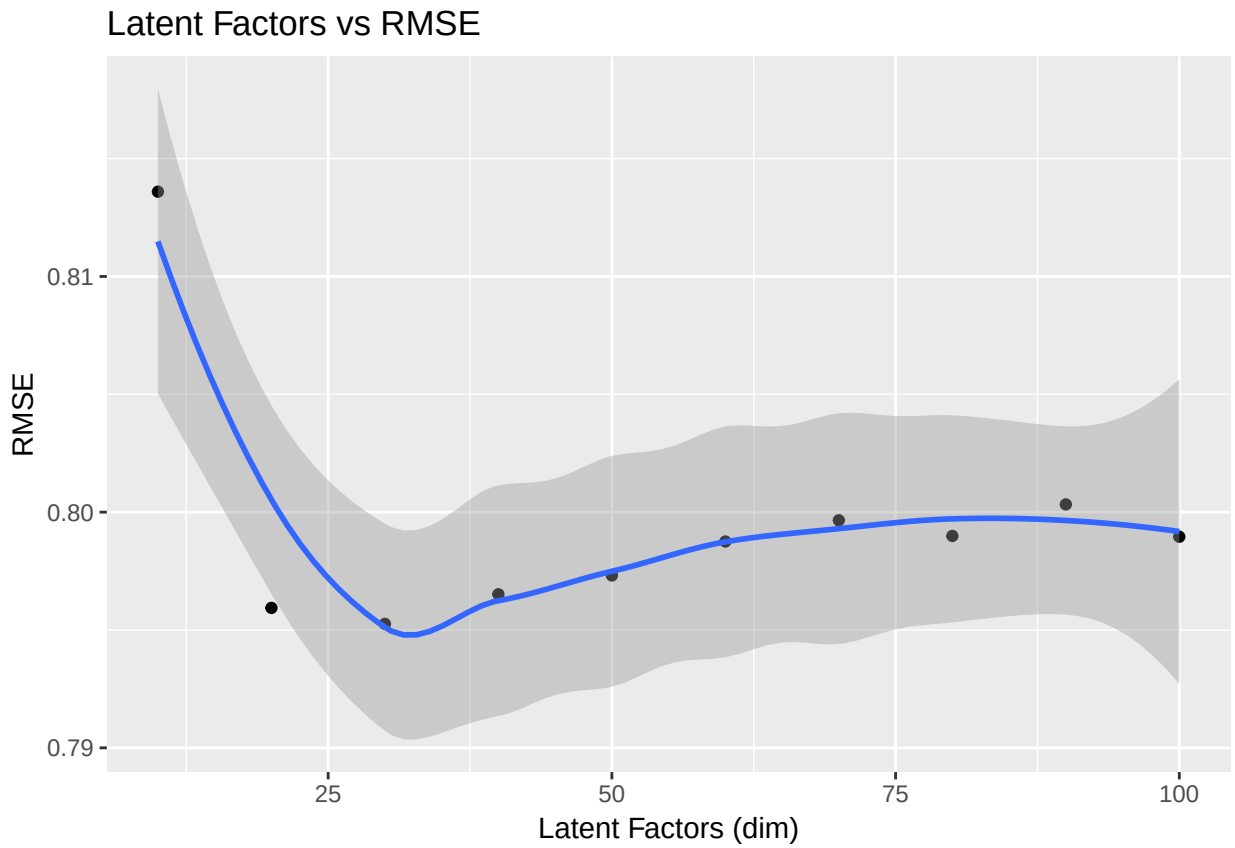
```
                            Train_Time = round(as.numeric(train_time, units="mins"), 2))
kable(RECO_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
V1 - 30 x 50	0.7808364	1.23

As mentioned previously, we will now look at the 5-Fold Cross Validation results using the Tuning function to predict the RMSE across values of the **dim** parameter between 10 and 100:

```
# We will not be executing this code because it takes too long.  Instead we will simply
# create a matrix of the results and plot the values,
#
#*********************************************************************************
# Tune the DIM parameter - FOR 10-100 and then 10-40 and Graph
# !!!!  NOTE !!!!  THESE WILL TAKE A LONG TIME TO RUN
#*********************************************************************************
# Tune & Graph Latent Factors (dim) from 10 to 100 by 10
#*********************************************************************************
#
# opts_list = list(dim      = c(seq(10, 100, 10)),    # Number of Latent Features
#                  costp_l1 = c(0),       # L1 regularization cost for User factors
#                  costp_l2 = c(0.01),    # L2 regularization cost for User factors
#                  costq_l1 = c(0),       # L1 regularization cost for Movie factors
#                  costq_l2 = c(0.1),     # L2 regularization cost for Movie factors
#                  lrate    = c(0.1),     # Learning Rate - Aprox step size in Gradient Descent
#                  nfold    = 5,          # Number of folds for Cross Validation
#                  nthread  = 8,          # Set Number of Threads to 1 *** Required to get Reproducible
#                  niter    = 50,         # Number of Iterations
#                  verbose  = FALSE)      # Don't Show Fold Details
#
# start <- Sys.time()
# set.seed(1, sample.kind = "Rounding")
# opts_tune <- r$tune(data_memory(train_set[, 1], train_set[, 2], train_set[, 3]),
#                     opts = opts_list)
# finish <- Sys.time()
# tune_time <- finish - start
#
# opts_tune$min
#
# opts_tune$res %>%
#   ggplot(aes(dim, loss_fun)) +
#   geom_point() +
#   geom_smooth(method="loess") +
#   labs(x="Latent Factors (dim)", y="RMSE") +
#   ggtitle("Latent Factors vs RMSE")
#*********************************************************************************


CV_10_100_50 <- data.frame("dim" = c(seq(10, 100, 10)),
                           "loss_fun" = c(0.8135956,
                                          0.7959371,
                                          0.7952601,
                                          0.7965165,
                                          0.7973126,
```

```
                                          0.7987548,
                                          0.7996540,
                                          0.7989897,
                                          0.8003280,
                                          0.7989563))

CV_10_100_50 %>%
  ggplot(aes(dim, loss_fun)) +
  geom_point() +
  geom_smooth(method="loess") +
  labs(x="Latent Factors (dim)", y="RMSE") +
  ggtitle("Latent Factors vs RMSE")
```

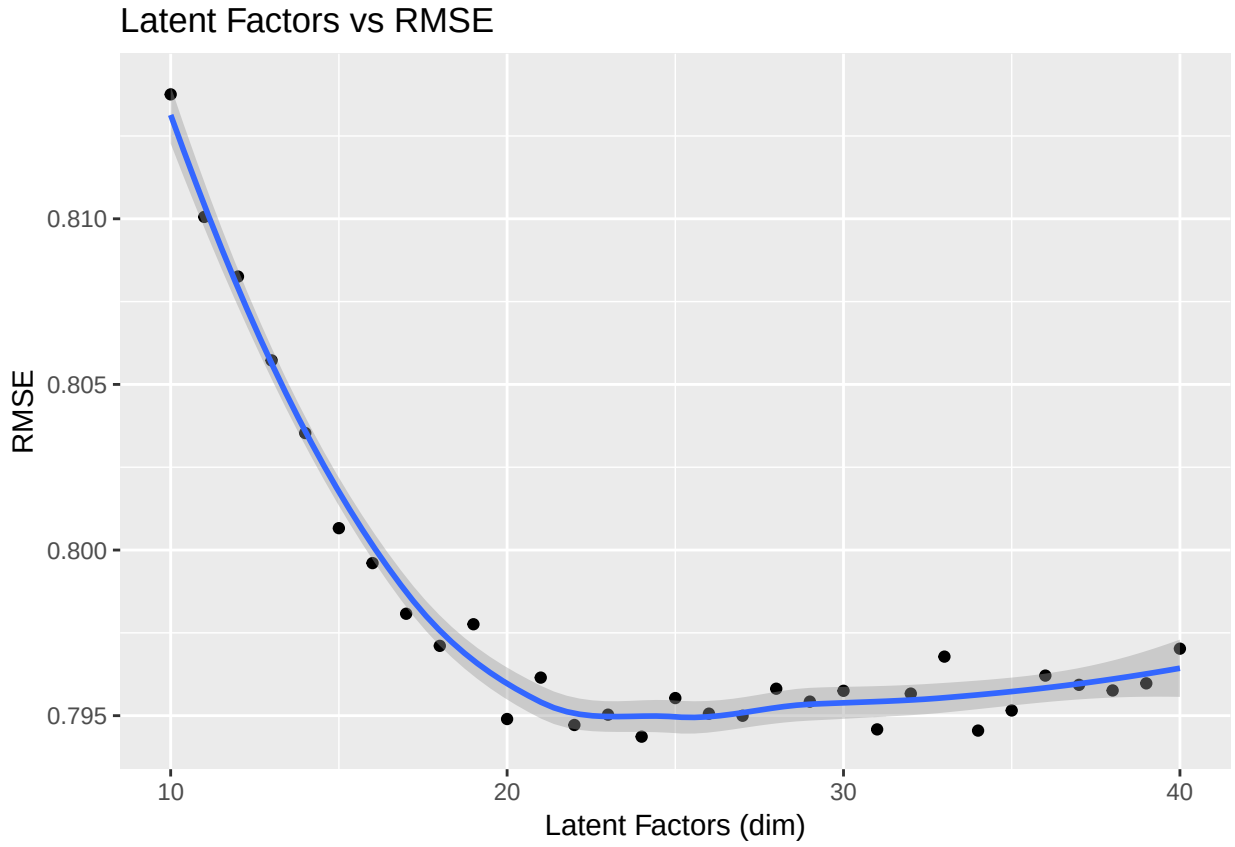


We see that the predicted minimum is somewhere between 20 & 30, so we'll re-run the Tuning between 10 and 40 to get a better look:

```
#*******************************************************************************
# Tune the DIM parameter - FOR 10-100 and then 10-40 and Graph
# !!!!  NOTE !!!!  THESE WILL TAKE A LONG TIME TO RUN
#*******************************************************************************
# Tune & Graph Latent Factors (dim) from 10 to 40
#*******************************************************************************
#
# opts_list = list(dim      = c(seq(10, 40, 2)),   # Number of Latent Features
#                  costp_l1 = c(0),       # L1 regularization cost for User factors
#                  costp_l2 = c(0.01),    # L2 regularization cost for User factors
#                  costq_l1 = c(0),       # L1 regularization cost for Movie factors
```

```
#                   costq_l2 = c(0.1),     # L2 regularization cost for Movie factors
#                   lrate    = c(0.1),     # Learning Rate - Aprox step size in Gradient Descent
#                   nfold    = 5,          # Number of folds for Cross Validation
#                   nthread  = 8,          # Set Number of Threads to 1 *** Required to get Reproducible
#                   niter    = 50,         # Number of Iterations
#                   verbose  = FALSE)      # Don't Show Fold Details
#
# start <- Sys.time()
# set.seed(1, sample.kind = "Rounding")
# opts_tune <- r$tune(data_memory(train_set[, 1], train_set[, 2], train_set[, 3]),
#                     opts = opts_list)
# finish <- Sys.time()
# tune_time <- finish - start
#
# opts_tune$min
#
# opts_tune$res %>%
#   ggplot(aes(dim, loss_fun)) +
#   geom_point() +
#   geom_smooth(method="loess") +
#   labs(x="Latent Factors (dim)", y="RMSE") +
#   ggtitle("Latent Factors vs RMSE")
#*********************************************************************************


CV_10_40_50 <- data.frame("dim" = c(10:40),
                          "loss_fun" = c(0.8137556,
                                         0.8100568,
                                         0.8082561,
                                         0.8057245,
                                         0.8035323,
                                         0.8006602,
                                         0.7996062,
                                         0.7980771,
                                         0.7971078,
                                         0.7977603,
                                         0.7948986,
                                         0.7961489,
                                         0.7947183,
                                         0.7950298,
                                         0.7943660,
                                         0.7955314,
                                         0.7950606,
                                         0.7949973,
                                         0.7958134,
                                         0.7954243,
                                         0.7957506,
                                         0.7945857,
                                         0.7956658,
                                         0.7967824,
                                         0.7945495,
                                         0.7951564,
                                         0.7962097,
```

```
                                        0.7959298,
                                        0.7957610,
                                        0.7959754,
                                        0.7970225))

CV_10_40_50 %>%
  ggplot(aes(dim, loss_fun)) +
  geom_point() +
  geom_smooth(method="loess") +
  labs(x="Latent Factors (dim)", y="RMSE") +
  ggtitle("Latent Factors vs RMSE")
```



From this, **dim = 24** gives the smallest RMSE, so we will re-Train the model at **dim = 24** and see how it compares to the current RMSE result:

```
#*******************************************************************************
# Train "Optimal" RMSE - (dim = 24)  V2 - 24 x 50
#*******************************************************************************
opts_list = list(dim      = c(24),      # Number of Latent Features
                 costp_l1 = c(0),       # L1 regularization cost for User factors
                 costp_l2 = c(0.01),    # L2 regularization cost for User factors
                 costq_l1 = c(0),       # L1 regularization cost for Movie factors
                 costq_l2 = c(0.1),     # L2 regularization cost for Movie factors
                 lrate    = c(0.1),     # Learning Rate - Aprox step size in Gradient Descent
                 nfold    = 5,          # Number of folds for Cross Validation
                 nthread  = 1,          # Set Number of Threads to 1 *** Required to get Reproducible R
                 niter    = 50,         # Number of Iterations
```

```
                verbose  = FALSE)      # Don't Show Fold Details

start <- Sys.time()

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

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
r$train(data_memory(train_set[, 1], train_set[, 2], train_set[, 3]),
        opts = c(opts_list,
                 nthread = 1))

finish <- Sys.time()
train_time <- finish - start

start <- Sys.time()

pred <- r$predict(data_memory(test_set[, 1], test_set[, 2], test_set[, 3]), out_memory())
finish <- Sys.time()

pred_time <- finish - start
pred_time
```

```
## Time difference of 49.28689 secs
```

```
rmse <- RMSE(test_set[,3],pred)

RECO_rmse_results <- bind_rows(RECO_rmse_results,
                               tibble(Method="V2 - 24 x 50 (Optimal DIM)",
                                      RMSE = rmse,
                                      Train_Time = round(as.numeric(train_time, units="mins"), 2)))
kable(RECO_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
V1 - 30 x 50	0.7808364	1.23
V2 - 24 x 50 (Optimal DIM)	0.7830790	0.99

```
#*****************************************************************************
```

Although the model with **dim = 24** produced the minimal RMSE in the Cross-Validation Tuning, we now see that it actually has a worse RMSE than the initial model with **dim = 30**. As a test, we will create a very Large model with **dim = 1000** and train it over 100 iterations to see what the actual RMSE is on the Test set. The code is provided below but will not be run because it takes almost 1 Hour to complete. The resulting RMSE will simply be added to the current results table.

```
#*****************************************************************************
# Train Very Large (dim = 1000) -  V3 - 1000 x 100
# !!! NOTE !!!  This takes almost 1 Hour to Run
#*****************************************************************************
#
# opts_list = list(dim      = c(1000),    # Number of Latent Features
#                  costp_l1 = c(0),       # L1 regularization cost for User factors
#                  costp_l2 = c(0.01),    # L2 regularization cost for User factors
#                  costq_l1 = c(0),       # L1 regularization cost for Movie factors
#                  costq_l2 = c(0.1),     # L2 regularization cost for Movie factors
```

```
#                   lrate    = c(0.1),     # Learning Rate - Aprox step size in Gradient Descent
#                   nfold    = 5,          # Number of folds for Cross Validation
#                   nthread  = 1,          # Set Number of Threads to 1 *** Required to get Reproducible
#                   niter    = 100,        # Number of Iterations
#                   verbose  = TRUE)       # Show Fold Details
#
# start <- Sys.time()
#
# set.seed(1, sample.kind = "Rounding")
# r$train(data_memory(train_set[, 1], train_set[, 2], train_set[, 3]),
#         opts = c(opts_list,
#                   nthread = 1))
#
# finish <- Sys.time()
# train_time <- finish - start
#
# start <- Sys.time()
#
# pred <- r$predict(data_memory(test_set[, 1], test_set[, 2], test_set[, 3]), out_memory())
# finish <- Sys.time()
#
# pred_time <- finish - start
# pred_time
#
# rmse <- RMSE(test_set[,3],pred)
#
#*********************************************************************************

# Add the actual results to the table
RECO_rmse_results <- bind_rows(RECO_rmse_results,
                               tibble(Method="V3 - 1000 x 100",
                                      RMSE = 0.7740506,
                                      Train_Time = 52.92))
kable(RECO_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
V1 - 30 x 50	0.7808364	1.23
V2 - 24 x 50 (Optimal DIM)	0.7830790	0.99
V3 - 1000 x 100	0.7740506	52.92

```
#*********************************************************************************
```

These results show that a very large **dim** value for Latent Factors still delivers an increase in performance. It is unknown what the limit is for the number of Latent Factors, but this model has produced the best RMSE result so far.

Results

Below are the tabulated results of all methods. We find that all of the approaches achieve an RMSE value below our target.

- Standard & Regularized Linear Regression

```
kable(LR_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE
LR: Base Mean Model	1.0612018
LR: Movie Effect Model	0.9439087
LR: Movie + User Effects Model	0.8653488
LR: Movie + User + Time Effects Model	0.8652511
LR: Movie + User + Time + Genre Effects Model	0.8648488
Reg LR: Movie Effect Model	0.9438538
Reg LR: Movie + User Effect Model	0.8648170
Reg LR: Movie + User + Time Effect Model	0.8646938
Reg LR: Movie + User + Time + Genre Effect Model	0.8643084

- XGB – Extreme Parallel Tree Boosting (xgBoost)

```
kable(XGB_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
XGB Linear 50 Boost Rnds	0.9149801	25.63 secs
XGB Linear 1000 Boost Rnds	0.8820874	8.131121 mins
XGB Mixed Tree 5 Boost Rnds	0.8690746	24.74677 mins
XGB Mixed Tree 50 Boost Rnds	0.8539179	20.66 HOURS

- RECO - Recommender w/ Matrix Factorization (recosystem)

```
kable(RECO_rmse_results) %>%
  kable_styling(full_width = FALSE, position = "left")
```

Method	RMSE	Train_Time
V1 - 30 x 50	0.7808364	1.23
V2 - 24 x 50 (Optimal DIM)	0.7830790	0.99
V3 - 1000 x 100	0.7740506	52.92

Conclusion

This Capstone MovieLens 10M project has applied three different techniques toward the development of a Movie Recommendation algorithm. Each approach has generated an RMSE below our target.

It is interesting to see that the Regularized Linear Regression performed almost as well as the xgBoost ensemble model. Given the popularity of xgBoost, I was surprised that it didn't perform better especially when training a very large model for over 20 hours.

The speed and excellent performance of the Recosystem Matrix Factorization was a surprise result. Using only three variables, it by far outperformed the other methods. Also of interest and asubject for further investigation was the fact that the continued addition of Latent Factors to the Matrix Factorization model led to ever improving RMSE results.