MovieLens Prediction Project

Praveen Srivatsa

January 6, 2020

Introduction

This project uses the MovieLens movie rating data set to improve the recommendation algorithm. The data set contains ratings of over 10000 movies across 20 genres and about 70000 users who have given a total of about 10 million ratings. (NOTE : The original dataset from Movielens had about 27000 movies and 138,000 users with about 20 million ratings. The dataset used here - the 10M dataset - is a subset of the same with about 10M ratings)

Predicting movie ratings can be done in multiple different ways. At the onset, there are a few movies - blockbusters - that are rated high by many people. This cuts across people's preferences, genres, age groups etc. Then there are preferences of the users which determines their ratings. Thirdly, people tend to rate movies that they like and do not give many ratings to those that they dont. This introduces a bias in the data set which is more skewed towards positive ratings.

An additional complexity is the classification of the genres. As we will see in detail, the genres are a | seperated set of tags which essentially consists of a combination of genres. While the distinct set of genres is reported as over 700, in reality there are only 20 genres. This introduces a skew in the analysis using genres that we will try to work around in this project.

In this project 10% of the dataset is set aside as a validation dataset. All the algorithms are trained on the remaining 90% of the dataset and the predictions are then compared against the values in the validation dataset that was initially set aside. This ensures that the algorithms are not overtrained. This projects focuses on reducing the the residual mean squared error (RMSE) on the validation set.

Data Setup

Downloading and Processing the Data

First step in any data analysis project is to download and process the dataset. We can download the MovieLens 10M data from the Grouplens website. As the dataset is about 65MB and the processing takes time, we can cache the outout on the local file system. This ensures that subsequent runs take significantly less time making the process efficient.

```r
# Note: this process could take a couple of minutes

options(digits = 8)
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")
```

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

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

#Set the working directory. This ensures that both R and RStudio use the same
#working folder for the files.
#NOTE : This works on Windows, change it for Mac/Linux.
setwd("c:\\temp")
edxfile <- "edx.file"

# Check for download. If exists, just load it, else download, process and save it.
if(!file.exists(edxfile))
{
  dl <- tempfile()
  download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

  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))

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

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

  # Validation set will be 10% of MovieLens data

  set.seed(1, sample.kind="Rounding")
  # if using R 3.5 or earlier, use `set.seed(1)` instead

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

  # Make sure userId and movieId in validation set are also in edx set
  validation <- temp %>%
     semi_join(edx, by = "movieId") %>%
     semi_join(edx, by = "userId")

  # Add rows removed from validation set back into edx set
  removed <- anti_join(temp, validation)
  edx <- rbind(edx, removed)

  rm(dl, ratings, movies, test_index, temp, movielens, removed)
  save(edx, validation, file = edxfile)
} else {
  load(edxfile)
}
```

Explore the MovieLens Data

MovieLens dataset initially had 10 million ratings (rows) with 6 variables (columns) which were partitioned into *edx* and *validation* sets containing ~9M and ~1M ratings respectively.

We can see the sets are in tidy format:

```
edx %>% as_tibble()
```

```
## # A tibble: 9,000,055 x 6
##    userId movieId rating timestamp title                  genres
##     <int>   <dbl>  <dbl>     <int> <chr>                  <chr>
##  1      1     122      5 838985046 Boomerang (1992)       Comedy|Romance
##  2      1     185      5 838983525 Net, The (1995)        Action|Crime|Thriller
##  3      1     292      5 838983421 Outbreak (1995)        Action|Drama|Sci-Fi|T...
##  4      1     316      5 838983392 Stargate (1994)        Action|Adventure|Sci-...
##  5      1     329      5 838983392 Star Trek: Generation... Action|Adventure|Dram...
##  6      1     355      5 838984474 Flintstones, The (199... Children|Comedy|Fanta...
##  7      1     356      5 838983653 Forrest Gump (1994)    Comedy|Drama|Romance|...
##  8      1     362      5 838984885 Jungle Book, The (199... Adventure|Children|Ro...
##  9      1     364      5 838983707 Lion King, The (1994)  Adventure|Animation|C...
## 10      1     370      5 838984596 Naked Gun 33 1/3: The... Action|Comedy
## # ... with 9,000,045 more rows
```

```
validation %>% as_tibble()
```

```
## # A tibble: 999,999 x 6
##    userId movieId rating  timestamp title                      genres
##     <int>   <dbl>  <dbl>      <int> <chr>                      <chr>
##  1      1     231    5    838983392 Dumb & Dumber (1994)       Comedy
##  2      1     480    5    838983653 Jurassic Park (1993)       Action|Adventure|...
##  3      1     586    5    838984068 Home Alone (1990)          Children|Comedy
##  4      2     151    3    868246450 Rob Roy (1995)             Action|Drama|Roma...
##  5      2     858    2    868245645 Godfather, The (1972)      Crime|Drama
##  6      2    1544    3    868245920 Lost World: Jurassic Par... Action|Adventure|...
##  7      3     590    3.5 1136075494 Dances with Wolves (1990) Adventure|Drama|W...
##  8      3    4995    4.5 1133571200 Beautiful Mind, A (2001)  Drama|Mystery|Rom...
##  9      4      34    5    844416936 Babe (1995)                Children|Comedy|D...
## 10      4     432    3    844417070 City Slickers II: The Le... Adventure|Comedy|...
## # ... with 999,989 more rows
```

Data Exploration

Each row represents a rating given by one user to one movie. We can see the number of unique users that provided ratings and how many unique movies were rated in the *edx* set. We can also see the total ratings that were given.

```r
edx %>%
  summarize(total_users = n_distinct(userId),
            total_movies = n_distinct(movieId),
            total_ratings = n())
```

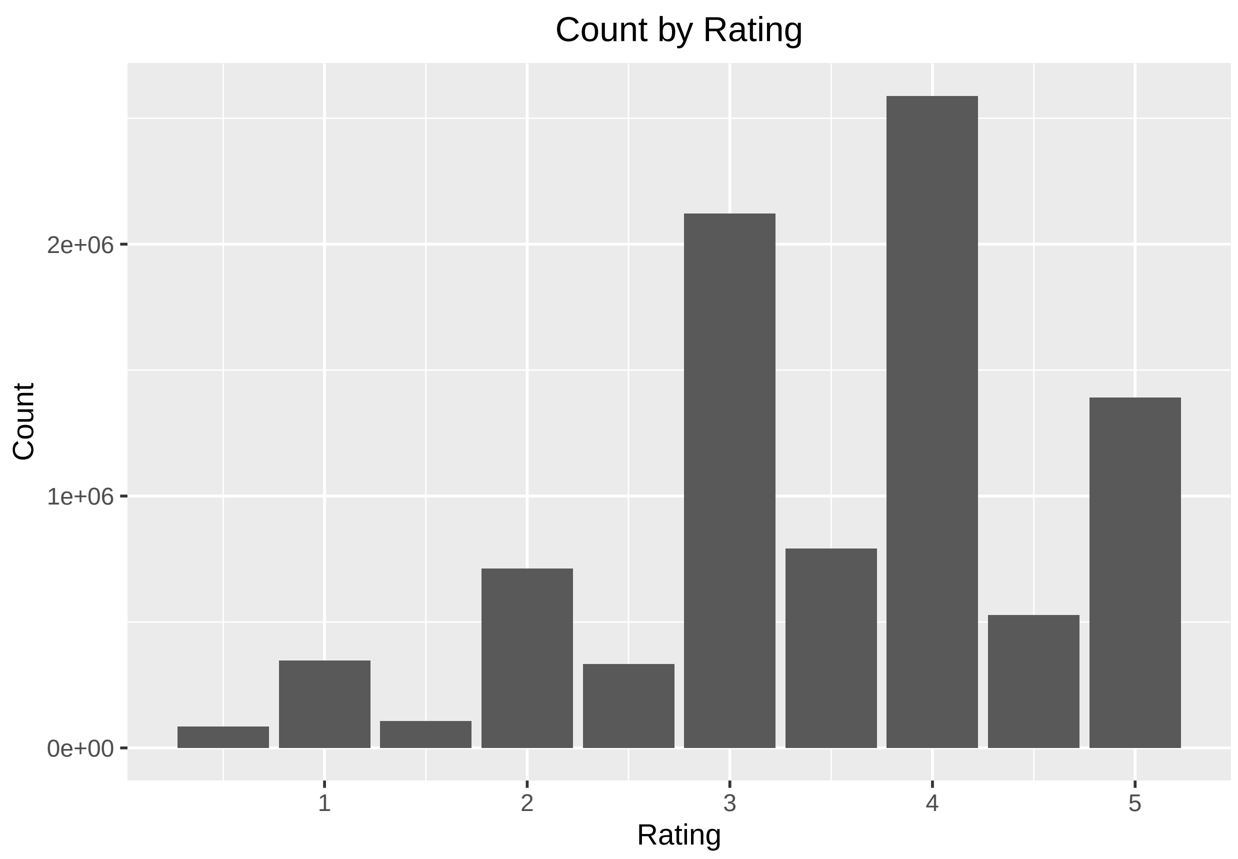
```
##   total_users total_movies total_ratings
## 1       69878        10677       9000055
```

We notice that we have about 70K users and 10K movies, but we only have 9M ratings out of a possible 700M combinations - which means that not all users rated all movies.

Distributions

Let's look at some of the general properties of the data to better understand the challenges. The first thing we can notice is the count by rating.

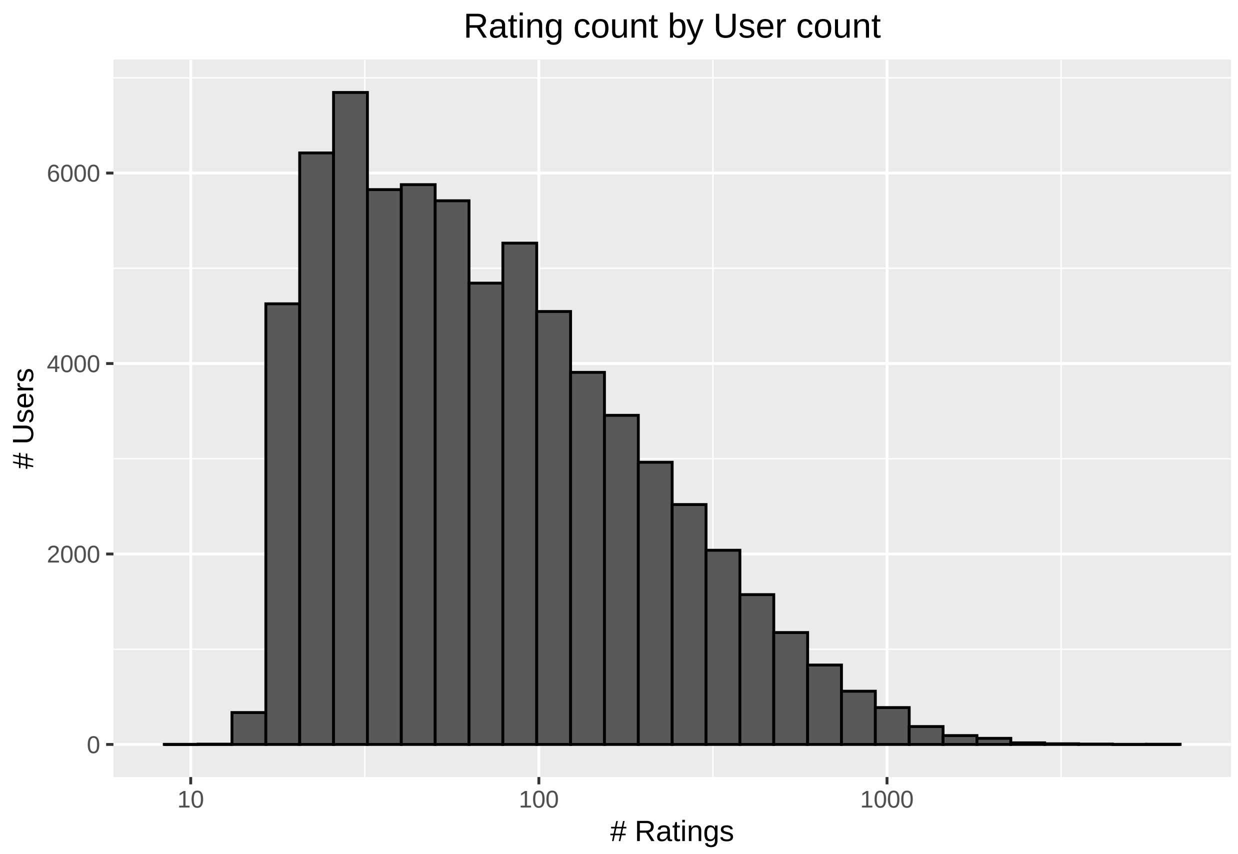
```r
count_byrating <- edx %>% group_by(rating) %>% summarize(count = n())
count_byrating %>% ggplot(aes(x=rating,y=count)) +
  geom_bar( aes(y = count), stat = "identity", position = "identity") +
  xlab("Rating") +
  ylab("Count") +
  ggtitle("Count by Rating") +
  theme(plot.title = element_text(hjust = 0.5))
```



We notice that 4,3 and 5 are the highest rated ratings. This is an indication that people who might not like a movie, might choose to NOT rate, thus leading to low count for the lower ratings. Also full numbered ratings (1,2,3,4,5) are significantly higher than the mid numbered (0.5,1.5,2.5,3.5,4.5) ratings. This is an interesting observation.

Our next observation is that some users tend to give a lot of ratings and are very active.

```r
edx %>%
  count(userId) %>%
  ggplot(aes(n)) +
  geom_histogram(color = "black") +
  scale_x_log10() +
  xlab("# Ratings") +
  ylab("# Users") +
  ggtitle("Rating count by User count") +
  theme(plot.title = element_text(hjust = 0.5))
```



As we can notice, a large set of users (about 7000) have rated over 50 movies, while there are a few users who have rated over a 1000 movies and are very active on the site. With these insights, lets proceed to run predictions on our dataset.

Data Analysis

In order to validate the accuracy of our predictions, we can use the confusionMartix and determine the accuracy, the specificity and the sensitivity. We can even compute the weighted F1 score. But a better way to measure our algorithm is to compute the loss function to determine how close our predictions were to the actual ratings from the validation set.

Loss Function

We use residual mean squared error (RMSE) as the metric to determine how good our algorithm works. If we define $y_{m,u,g}$ as the rating for movie m by user u with genre g and denote out prediction with $\hat{y}_{m,u,g}$. The RMSE is then defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{m,u,g}(\hat{y}_{m,u,g} - y_{m,u,g})^2}$$

with N being the number of user/movie/genre combinations and the sum occurring over all these combinations.

Let's write a function that computes the RMSE for vectors of ratings and their corresponding predictors:

```r
# rmsefunction for vectors of ratings and their corresponding predictors
getrmse <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

First approach: Simple model

Let's start with a simple model. Lets assume that movie,user and genre preferences do not contribute to the ratings. Thus the mean ratings across movies is the predicted rating for any other movie.

This model can be depicted as such.

$$Y_{m,u,g} = \mu + \epsilon_{m,u,g}$$

with $\epsilon_{m,u,g}$ as errors sampled from the same distribution centered at 0 and μ the "true" rating for all movies. We know that the estimate that minimizes the RMSE is the least squares estimate of μ and, in this case, is the average of all ratings:

```r
edx_mean <- mean(edx$rating)
edx_mean
```

```
## [1] 3.5124652
```

We see that the average ratings of all movies is around 3.512 on a scale of 0 to 5.If we predict all unknown ratings with $\hat{\mu}$ we obtain the following RMSE:

```r
rmse01 <- getrmse(validation$rating, edx_mean)

all_rmses <- data_frame(method = "(rmse01) Simple approach", rmse = rmse01)
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.0612018

With this simple approach, we get a RMSE above 1. This means that just using the average of the ratings to predict the ratings of the other movies results in a very bad loss. Lets see how we can improve upon this prediction.

Second approach: Modelling movie effects

The first approach just assumed that future ratings are based on past ratings. It did not consider that some movies will be rated higher than others. Taking the mean ratings per movie and using this as a basis to predict the rating of other movies adds on the movie effect making the predictions better. This can be depicted as

$$Y_{m,u,g} = \mu + b_m + \epsilon_{m,u,g}$$

```r
avg_bymovie <- edx %>%
  group_by(movieId) %>%
  summarise(mov_mean = mean(rating - edx_mean))

mean_bymovie <- edx_mean + validation %>%
  left_join(avg_bymovie, by='movieId') %>%
  pull(mov_mean)

rmse02 <- getrmse(validation$rating,mean_bymovie)

all_rmses <- bind_rows(all_rmses,
            data_frame(method="(rmse02) Movie Effect", rmse = rmse02))
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866

By grouping the ratings on the basis of movies, our loss came down to below 1. It is now at 0.944 However it is still not too good. Lets see if we can add on other effects to bring down our loss.

Third approach: Modelling user effects

In the last approach, we did not consider the user effect. Some users rate some movies better and others dont. So using just the average of the movie ratings to predict future movie ratings is not good enough. We have to add the user effect into the algorithm.

$$Y_{m,u,g} = \mu + b_m + b_u + \epsilon_{m,u,g}$$

where b_u is a user effect. This now adds on the effect of a user rating a great movie high vs the user rating a great movie low.

```r
avg_byuser <- edx %>%
  left_join(avg_bymovie, by="movieId") %>%
  group_by(userId) %>%
  summarise(user_mean = mean(rating - edx_mean - mov_mean))
```

```r
mean_byuser <- validation %>%
  left_join(avg_bymovie, by='movieId') %>%
  left_join(avg_byuser, by='userId') %>%
  mutate(user_mean = edx_mean + mov_mean + user_mean) %>%
  pull(user_mean)

rmse03 <- getrmse(validation$rating,mean_byuser)

all_rmses <- bind_rows(all_rmses,
            data_frame(method="(rmse03) Movie-User Effect", rmse = rmse03))
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866
(rmse03) Movie-User Effect	0.86534882

Now this is looking better and our rmse is now at 0.865. Notice that using multiple effeccts like movie and user brings down the loss substantially. However as we noticed earlier, not all users rate all movies and some users are very active while others are not. This skews our data set and we can try to adjust for the same.

Fourth model: Regularization Effect

Regularization

As we noticed from our initial exploration, users tend to give higher ratings and not give lower ratings. Also some users give many (over 1000) ratings while other just give a few (50). So we could have a movie with 100s of good ratings giving it a high average rating, while others might have just a few good ratings, but might end up with a higher average.

In order to normalize the effect sizes, we can use a tuning parameter that fit the model using the following model:

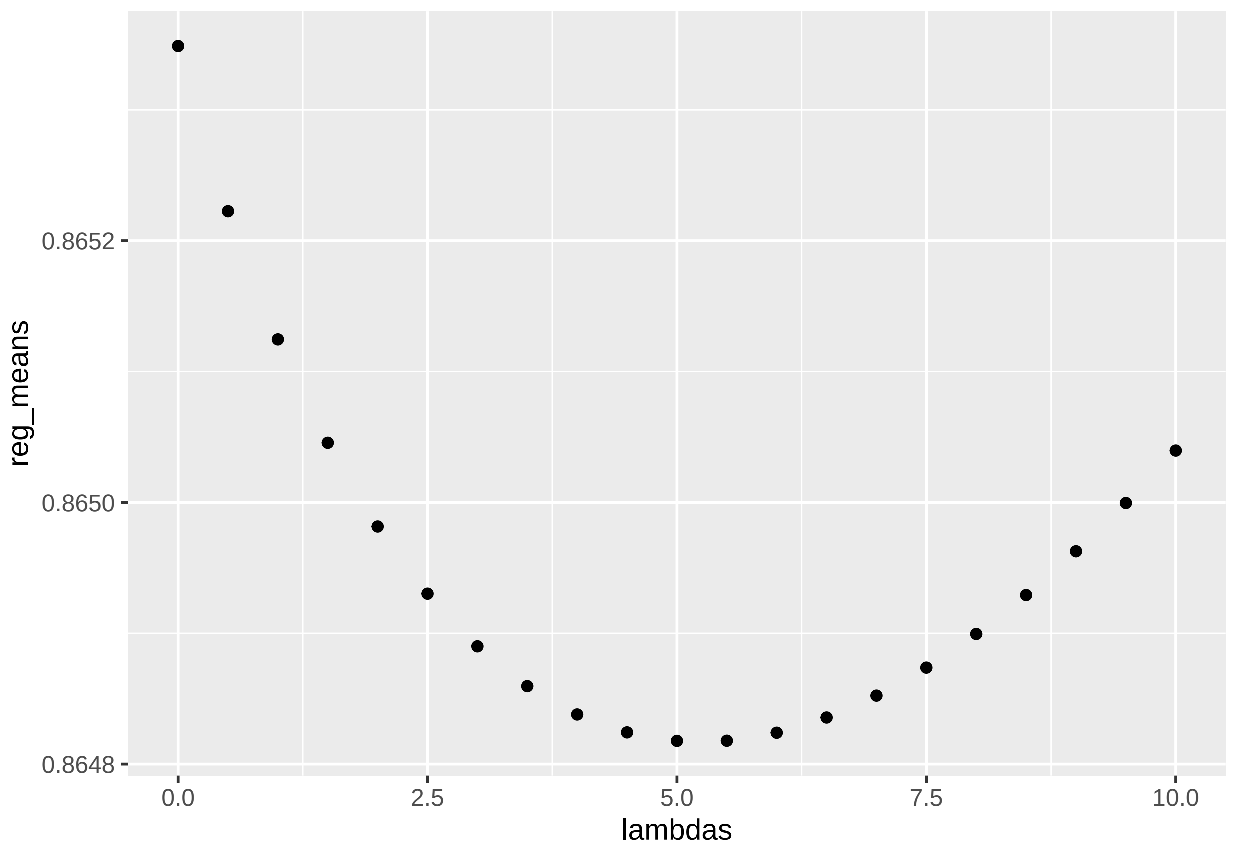
$$\frac{1}{\lambda + n_i} \sum_{u=1}^{n_i} (Y_{m,u,g} - \hat{\mu})$$

where n_i is the number of ratings made for movie i. This approach will have our desired effect: when our sample size n_i is very large, a case which will give us a stable estimate, then the penalty λ is effectively ignored since $n_i + \lambda \approx n_i$. However, when the n_i is small, then the estimate $\hat{b}_i(\lambda)$ is shrunken towards 0 but when n_i is more we the estimate starts moving away from the actual values.

Regularizing movie and user effects

We can use regularization to estimate movie effect as well as the user effects. Note that λ is a tuning parameter. We can use cross-validation to choose it.

```r
lambdas <- seq(0, 10, 0.5)

reg_means <- sapply(lambdas, function(l){
  ravg_mov <- edx %>%
    group_by(movieId) %>%
    summarize(rmov_mean = sum(rating - edx_mean)/(n()+l))

  ravg_user <- edx %>%
    left_join(ravg_mov, by="movieId") %>%
    group_by(userId) %>%
    summarize(ruser_mean = sum(rating - rmov_mean - edx_mean)/(n()+l))

  rmean_regularized <-
      validation %>%
      left_join(ravg_mov, by = "movieId") %>%
      left_join(ravg_user, by = "userId") %>%
      mutate(reg_mean = edx_mean + rmov_mean + ruser_mean) %>%
      pull(reg_mean)

  return(getrmse(validation$rating,rmean_regularized))
})

qplot(lambdas, reg_means)
```



As we notice from the plot, the optimal λ is 5. Using this, lets us compute our rmse considering the movie and the user effect, but regularized with the tuning parameter.

```r
# lambda that minimizes rmse
lambda <- lambdas[which.min(reg_means)]

# calculate rmse after regularizing movie + user effect from previous models
rmse04 <- min(reg_means)

all_rmses <- bind_rows(all_rmses,
            data_frame(method="(rmse04) Regularized Movie-User Effect", rmse = rmse04))
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866
(rmse03) Movie-User Effect	0.86534882
(rmse04) Regularized Movie-User Effect	0.86481775

By regularizing the effect of the movie-user combination with a tuning parameter, we have brought down the loss to 0.8648 which is better than the desired target. While this is great, lets see if we can do better. So far we have not considered the effect of the genre at all. Would that help?

Fifth model: Genre Effect

Many users have preferred genres and their ratings will vary based on their genre preferences. While this might not affect some of the blockbuster movies which result in higher ratings across user and genre preferences, it could have a significant impact for all the other movies. Lets explore and find out. So let's compute :

$$Y_{m,u,g} = \mu + b_m + b_u + b_g + \epsilon_{m,u,g}$$

where b_g is a genre-specific effect.

```r
avg_bygenres <- edx %>%
  left_join(avg_bymovie, by="movieId") %>%
  left_join(avg_byuser, by="userId") %>%
  group_by(genres) %>%
  summarise(genres_mean = mean(rating - edx_mean - mov_mean - user_mean))

mean_bygenre <- validation %>%
  left_join(avg_bymovie, by='movieId') %>%
  left_join(avg_byuser, by='userId') %>%
  left_join(avg_bygenres, by='genres') %>%
  mutate(genres_mean = edx_mean + mov_mean + user_mean + genres_mean) %>%
  pull(genres_mean)

rmse05 <- getrmse(validation$rating,mean_bygenre)

all_rmses <- bind_rows(all_rmses,
            data_frame(method="(rmse05) Genre Effect", rmse = rmse05))
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866
(rmse03) Movie-User Effect	0.86534882
(rmse04) Regularized Movie-User Effect	0.86481775
(rmse05) Genre Effect	0.86494689

Well by bringing in the Genre effect, the loss has INCREASED to 0.8649. This does not look good. Lets take a look at the genres.

```
edx %>% summarize(total_users = n_distinct(userId),
                  total_movies = n_distinct(movieId),
                  total_genres = n_distinct(genres))
```

```
##   total_users total_movies total_genres
## 1       69878        10677          797
```

If we explore the data, we realize that it reports nearly 800 genres!!! However on closer examination we realize that multiple genres are marked against a record and these are separated by a |. In reality there are only about 20 genres. So we have been using the effect of a combination of the genres and not the actual genres making our algorithm less efficient. Lets see what we can do about that.

Sixth model: Independent Genre Effect

So lets split up the genres into independent records. We can do this with the separate_rows function which breaks up the record by any given character.

NOTE : Splitting the Genre increases the dataset size substantially. This used over 34GB RAM and errored out on a regular desktop with only about 16GB RAM. So be sure to run/test this on a system with enough resources (8 vCPU, 56 GB RAM recommended). Even with that the entire report generation took nearly an hour

```
setwd("c:\\temp")
edxfile_g <- "edx_g.file"
if(!file.exists(edxfile_g)){
    validation_g <- validation %>%  separate_rows(genres, sep = "\\|")
    edx_g <- edx %>% separate_rows(genres, sep = "\\|")
    save(edx_g, validation_g, file = edxfile_g)
} else {
    load(edxfile_g)
}

edx_g %>% summarize(total_users = n_distinct(userId),
                    total_movies = n_distinct(movieId),
                    total_genres = n_distinct(genres))
```

```
##   total_users total_movies total_genres
## 1       69878        10677           20
```

```r
edx_g %>% group_by(genres) %>% summarize(count = n())
```

```
## # A tibble: 20 x 2
##    genres               count
##    <chr>                <int>
##  1 (no genres listed)       7
##  2 Action             2560545
##  3 Adventure          1908892
##  4 Animation           467168
##  5 Children            737994
##  6 Comedy             3540930
##  7 Crime              1327715
##  8 Documentary          93066
##  9 Drama              3910127
## 10 Fantasy             925637
## 11 Film-Noir           118541
## 12 Horror              691485
## 13 IMAX                  8181
## 14 Musical             433080
## 15 Mystery             568332
## 16 Romance            1712100
## 17 Sci-Fi             1341183
## 18 Thriller           2325899
## 19 War                 511147
## 20 Western             189394
```

We notice now that there are only 20 genres. Lets us run our algorithm with the genre effect and see how it performs.

```r
avg_byg <- edx_g %>%
  left_join(avg_bymovie, by="movieId") %>%
  left_join(avg_byuser, by="userId") %>%
  group_by(genres) %>%
  summarise(genres_mean = mean(rating - edx_mean - mov_mean - user_mean))

mean_byg <- validation_g %>%
  left_join(avg_bymovie, by='movieId') %>%
  left_join(avg_byuser, by='userId') %>%
  left_join(avg_byg, by='genres') %>%
  mutate(genres_mean = edx_mean + mov_mean + user_mean + genres_mean) %>%
  pull(genres_mean)

rmse06 <- getrmse(validation_g$rating,mean_byg)

all_rmses <- bind_rows(all_rmses,
          data_frame(method="(rmse06) Split Genre Effect", rmse = rmse06))
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866
(rmse03) Movie-User Effect	0.86534882

method	rmse
(rmse04) Regularized Movie-User Effect	0.86481775
(rmse05) Genre Effect	0.86494689
(rmse06) Split Genre Effect	0.86313336

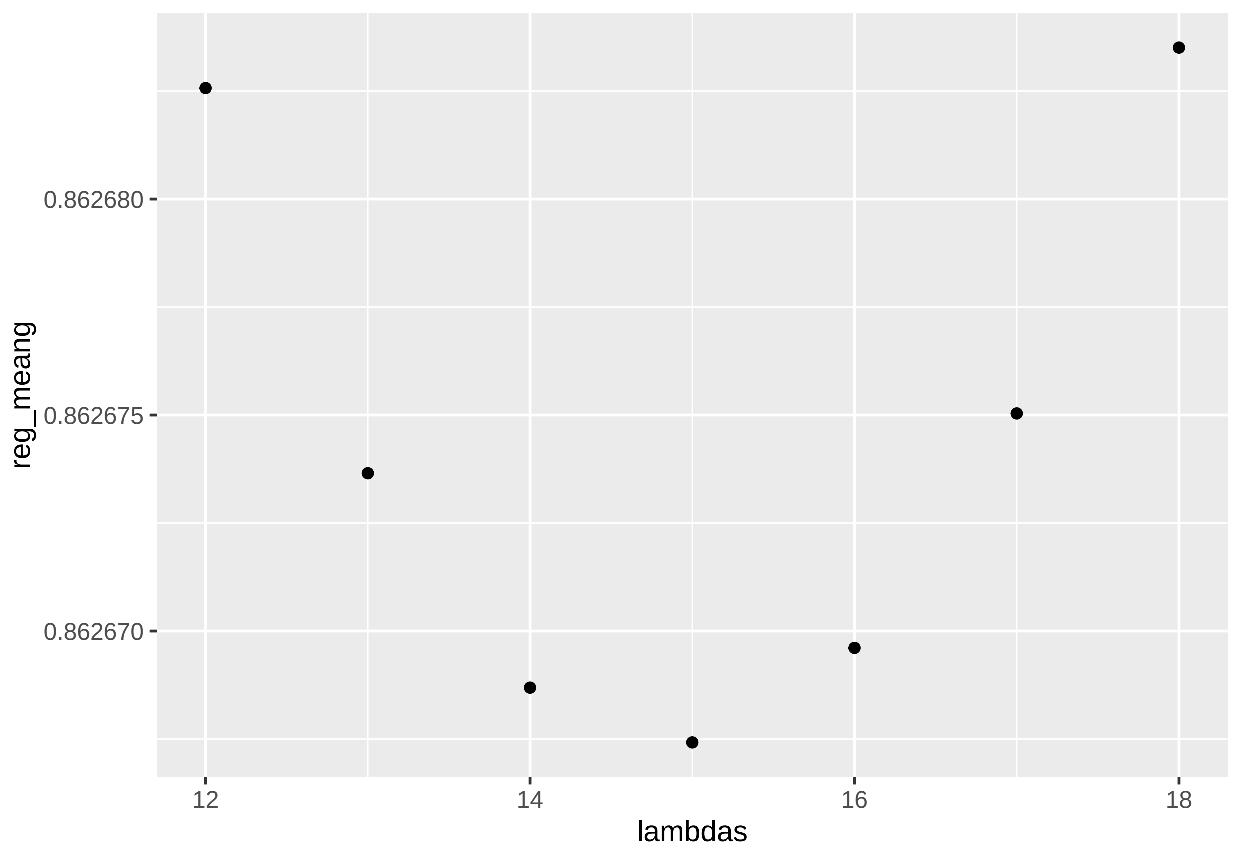
This is now better. With an rmse of 0.8631 it is better than the regularization effect with movie and user effects. But can we regularize the effect of the genre too? Lets find out.

Seventh model: Independent Genre Effect with Regularization

We noticed the impact of regularization on the movie and user combination. Lets extend our genre approach and regularize the same to see its impact on our rmse.

This takes a very long time to run, so be careful with choose too many tuning parameters. The seq was limited to run between 12 and 18 here using internal tests that indicated the best value

```r
lambdas <- seq(12, 18, 1)

reg_meang <- sapply(lambdas, function(l){

  ravg_movg <- edx_g %>%
    group_by(movieId) %>%
    summarize(rmov_meang = sum(rating - edx_mean)/(n()+l))

  ravg_userg <- edx_g %>%
    left_join(ravg_movg, by="movieId") %>%
    group_by(userId) %>%
    summarize(ruser_meang = sum(rating - rmov_meang - edx_mean)/(n()+l))

  ravg_byg <- edx_g %>%
    left_join(ravg_movg, by="movieId") %>%
    left_join(ravg_userg, by="userId") %>%
    group_by(genres) %>%
    summarise(rgenres_meang = mean(rating - rmov_meang - ruser_meang - edx_mean))

  rmean_regularizedg <- validation_g %>%
    left_join(ravg_movg, by='movieId') %>%
    left_join(ravg_userg, by='userId') %>%
    left_join(ravg_byg, by='genres') %>%
    mutate(reg_meang = edx_mean + rmov_meang + ruser_meang + rgenres_meang) %>%
    pull(reg_meang)

  return(getrmse(validation_g$rating,rmean_regularizedg))
})

qplot(lambdas, reg_meang)
```



As the plot shows us, the effect of the lambda is best at 15. Lets use this to compute the rmse with regularization of the genre+user+movie effect.

```r
# lambda that minimizes rmse
lambda <- lambdas[which.min(reg_meang)]

# calculate rmse after regularizing movie + user effect from previous models
rmse07 <- min(reg_meang)

all_rmses <- bind_rows(all_rmses,
            data_frame(method="(rmse07) Regularized Split Genre Effect", rmse = rmse07))
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866
(rmse03) Movie-User Effect	0.86534882
(rmse04) Regularized Movie-User Effect	0.86481775
(rmse05) Genre Effect	0.86494689
(rmse06) Split Genre Effect	0.86313336
(rmse07) Regularized Split Genre Effect	0.86266742

This gives great results and our rmse is now at **0.8626**

Conclusion

We can inspect the RMSEs for the various model trained from simple approach (of predicting the mean rating regardless of the movie, user or genre) to regularized movie and user effects (by finding the optimal value of tuning parameter λ) to the approach of splitting the genres and regularizing their effect.

The final results of the RMSEs for various approaches are as follows:

```r
all_rmses %>% knitr::kable()
```

method	rmse
(rmse01) Simple approach	1.06120181
(rmse02) Movie Effect	0.94390866
(rmse03) Movie-User Effect	0.86534882
(rmse04) Regularized Movie-User Effect	0.86481775
(rmse05) Genre Effect	0.86494689
(rmse06) Split Genre Effect	0.86313336
(rmse07) Regularized Split Genre Effect	0.86266742

As we can see from the results we started with a simple but ineffective approach. The next 3 approches built upon it by taking into consideration the movie effect, the user effect and then regularizing this model using a tuning parameter. This gave us an acceptable rmse score of *0.8648177*

However, we noticed a significant benefit when we broke up the genre into independent observations. This was however very heavy on computation and needed much higher computing resources. By taking the independent genres effect and regularizing the same, we ended with a final rmse score of *0.8626674*