Movielens Project edx-PH125.9x

Jin Xin

Sep, 2022

```
#libraries are used in this report
library(tidyverse)
library(data.table)
library(caret)
library(Matrix)
library(doParallel)
library(glmnet)
library(RcppArmadillo)
```

Executive summary

We are going to create a movie recommendation system in this project. This system can predict the possible rating a movie likely to receive from a user. We use the RMSE (root mean square error) to evaluate our algorithm performance, and the final achieved prediciton RMSE is 0.8053019. The central concept of our algorithm is based on the Netflix competition winning "Latent Factor" method (Koren, Bell, and Volinsky 2009), but a rather vanilla version, we also take the advantage of feature selecting in a sparse dataset of the "Elastic Net" algorithm to derive the user's genres information.

Introduction

This project is inspired by the $1 million prize competition hosted by Netflix in 2006. The rule of winning is to achieve a 10% improvement on RMSE, comparing to Netflix own in house algorithm. But in this project, we just use a fraction of Netflix competition dataset (we use the 10m version against the original size of 100m (Töscher and Jahrer 2009)). Our goal in here is simply to achieve a RMSE < 0.86490 between our prediction and validation set ratings.

In the following sessions, we will have a macro view on the dataset first, then we will explain details of the model development to achieve the final goal.

Data exploration

First of all, we use the following code to download and prepare the dataset we need. The code in below is provided by edx-HarvardX PH125.9x teaching team. To successfully obtain the "edx" and "validation" dataset, we need to follow the instructions inside the code chunk below according to our own computer setup.

```
############################################################
# Create edx set, validation set (final hold-out test set)
############################################################
```

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

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

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

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

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

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

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


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

# Validation set will be 10% of MovieLens data
# if using R 3.5 or earlier, use `set.seed(1)`
set.seed(1, sample.kind="Rounding")
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1,
                                  list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]

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

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

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

We will use the "edx" data to develop and train our model, then apply the trained model on the "validation" data to predict the *rating with respect to the user/movie pairs*, and evaluate the prediction against the real ratings by the *RMSE* method. So let's look at the macro structure of the dataset first.

Here is the macro view of "edx" dateset:

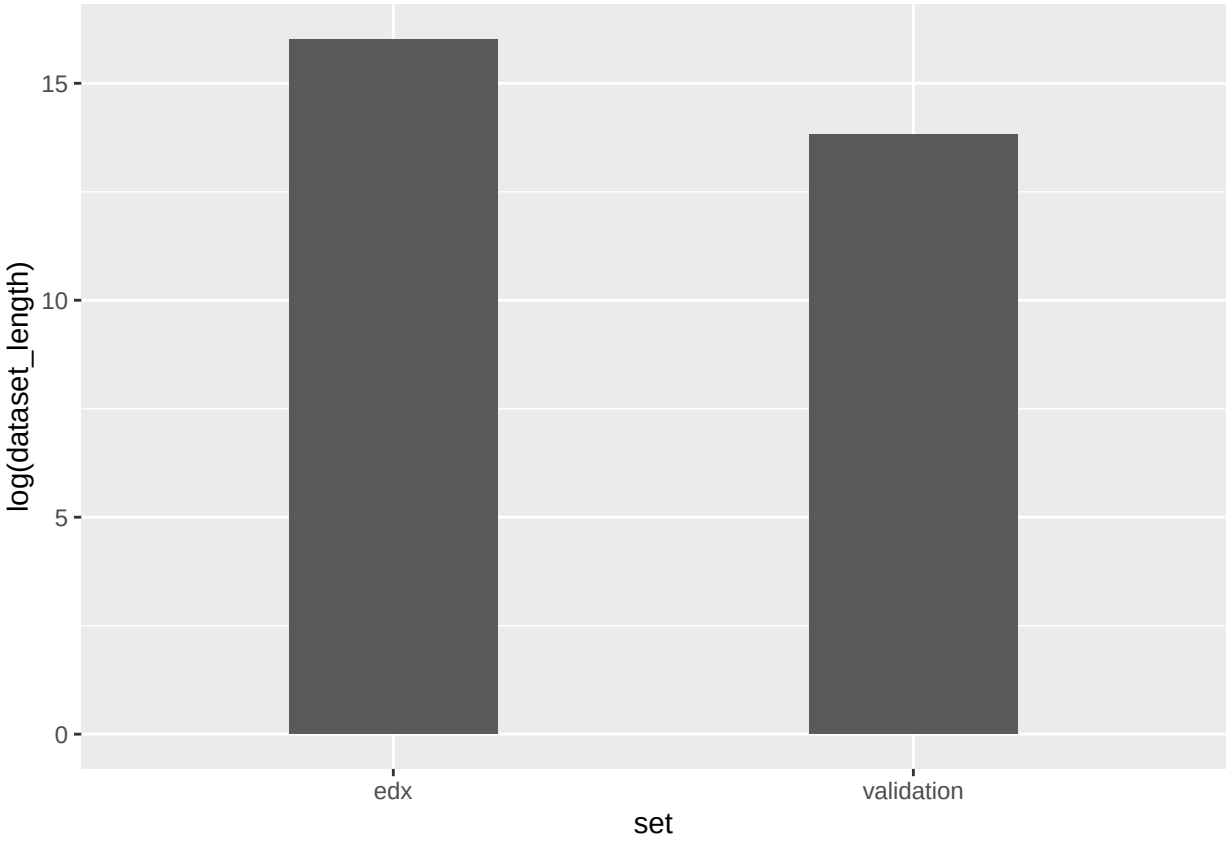
userId	movieId	rating	timestamp	title	genres
1	122	5	838985046	Boomerang (1992)	Comedy|Romance
1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller
1	316	5	838983392	Stargate (1994)	Action|Adventure|Sci-Fi
1	329	5	838983392	Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-Fi
1	355	5	838984474	Flintstones, The (1994)	Children|Comedy|Fantasy

Here is the macro view of "validation" dataset:

userId	movieId	rating	timestamp	title	genres
1	231	5	838983392	Dumb & Dumber (1994)	Comedy
1	480	5	838983653	Jurassic Park (1993)	Action|Adventure|Sci-Fi|Thriller
1	586	5	838984068	Home Alone (1990)	Children|Comedy
2	151	3	868246450	Rob Roy (1995)	Action|Drama|Romance|War
2	858	2	868245646	Godfather, The (1972)	Crime|Drama
2	1544	3	868245920	Lost World: Jurassic Park, The (Jurassic Park 2) (1997)	Action|Adventure|Horror|Sci-Fi|Thriller

Comparing the "edx" and "validation" dataset, we can tell the "edx" has the size of 9,000,055 $\times$ 6, is just a longer version of "validation" with the size of 999,999 $\times$ 6.

set	size
edx	9000055
validation	999999



When we look into the details of "edx" dataset, we have:

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

In these 6 headers, we can classify them into three groups: user info, movie info, and ratings (the user-movie joint info). For user info, we have "userId", "timestamp". This "timestamp" is showing the information about when the movie is rated by the user, and in the Unix Epoch Time format (Harper and Konstan 2016).

For movie info, we have "movieId", "title", and "genres". One thing need to pay attention is that, one movie can be categarised in multiple "genres".

let's look at one example to have a more straight understanding.

```
#we convert the timestamp column over the entire 'edx' set first
sample <- edx[, timestamp :=
                as.Date(as.POSIXct(timestamp, origin="1970-01-01"))][9,]

knitr::kable(sample)
```

userId	movieId	rating	timestamp	title	genres
1	364	5	1996-08-02	Lion King, The (1994)	Adventure|Animation|Children|Drama|Musical

```
rm(sample)
```

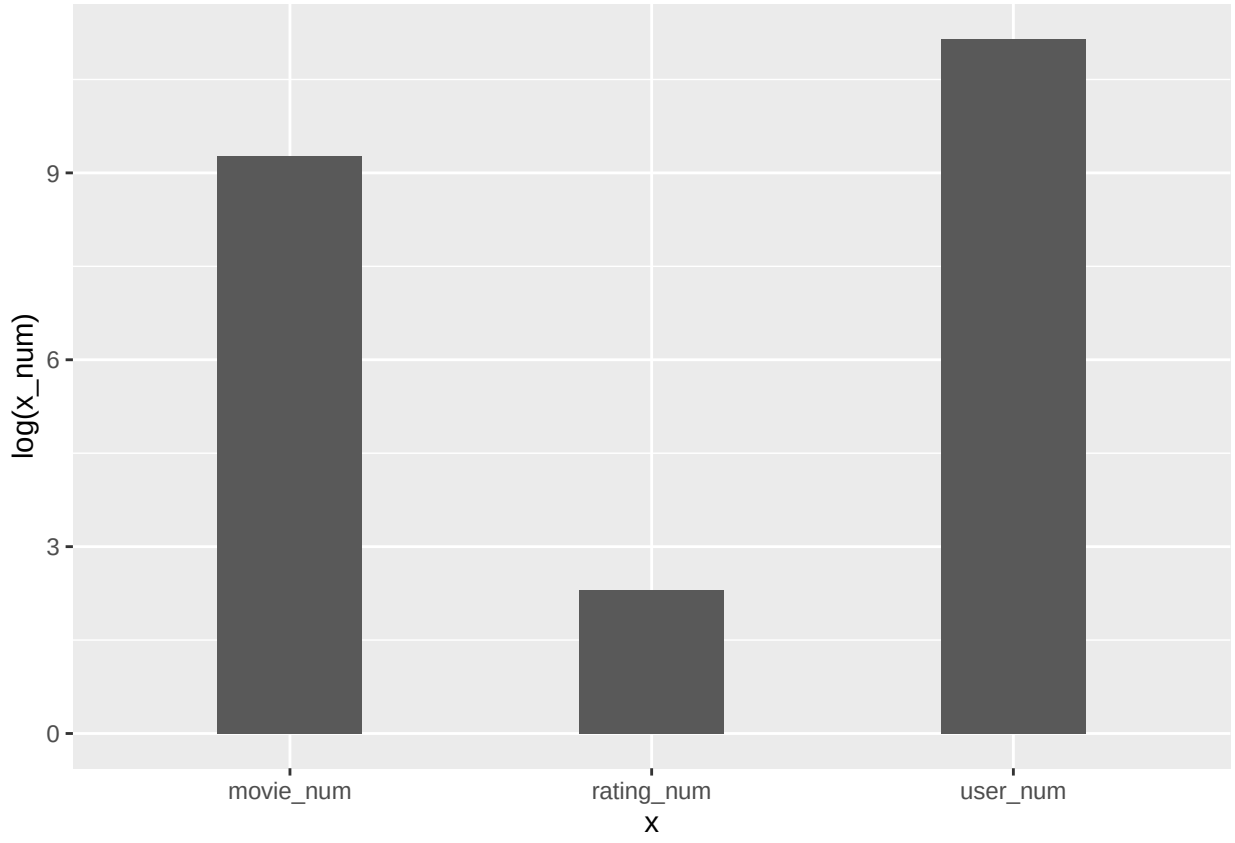
We can tell this user by "userId" = 1 rated the movie "The Lion King" by "movieId" = 364 for 5 stars, on the date of 1996-08-02. This movie "The Lion King" was released in 1994, and the movie is categorised in genres of Adventure, Animation, Children, Drama, Musical.

So our job is to use these user and movie info to predict the "rating", and the "rating" is ranged between:

```
range(edx$rating)
```

```
## [1] 0.5 5.0
```

We see the range is starting from 0.5, but implicitly the floor of the rating range is **0**. Why does this implicit **0** rating assumption may stand? Let us look at number of user/movie paired ratings we have in "edx":



Theoretically, there should be:

```
uniqueN(edx$userId) * uniqueN(edx$movieId)
```

```
## [1] 746087406
```

this number of total ratings, but instead we only got 9,000,055 in our dataset. Of course, this is very reasonable, since it is not possible for having everyone watches all movies, and it is also possible for one has watched the movie, but did not rate it. In other word, we can view this as a big matrix with the dimension of user_num: 69878 by movie_num: 10677, however this matrix is highly sparse with only rating_num:

9,000,055 of non zero elements, and the reset are all zeros. Mathematically, we call this kind of matrix a "sparse matrix".

From the code snippet at very beginning helped us get the "edx" and "validation" datasets, we can tell the "validation" data is intentionally built based on users and movies also included in "edx", which we use for our algorithm development.

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

That "userId" and "movieId" matching process makes the rating matrix of "validation" being exactly the same as "edx", so we can further specify our job to be predicting the "rating" at the corresponding "userId"-"movieId" position in the big sparse matrix, based on the known elements are given by "edx" dataset. Then it becomes to estimate an element value inside a sparse matrix problem.

After we identified our problem mathematically, one of the most popular algorithm to solve this problem is "Latent Factor" method by Koren, Bell, and Volinsky (2009). We will imitate this $1 million algorithm, but in a rather simple way, and combining with an extra step of using "Elastic Net" to search user's genres info, which is not included in the dataset.

Methods Details

Before we get into the details of our algorithm, we will build some basic theoretical understanding on it first.

Theoretical Foundations

The central idea of our algorithm, as well as the one by Koren, Bell, and Volinsky (2009), is matrix factorisation. More specifically, it is the idea of PCA or SVD in linear algebra, but on the opposite way. As described by Bell, Koren, and Volinsky (2007), PCA is the idea of rank minimisation for a dense matrix with no missing entry, but our user-movie rating matrix is highly sparse with most entries are missing, so we are going the opposite way of rank reduction, instead we are estimating the rank up from zero. To justify our approach, let's break it into two parts: 1, rating matrix factorisation; 2, factor search.

Because PCA is just a slim version of SVD in theory, let's build some foundation on SVD first. Strang (2016) tells us any matrix can be factorised by SVD, even the rank 1 matrix:

$$A = U\Sigma V^T$$

In the above SVD formula, assume A is a m(rows) by n(cols) matrix, where U is a m(rows) by m(cols) square matrix composed by singular vectors in A's column space R^m, which captures the feature information of the rows of A. The V is a n(rows) by n(cols) square matrix composed by singular vectors in A's row space R^n, which captures the feature information of the columns of A. Lastly, the Σ is a m(rows) by n(cols) diagonal matrix contains f number of singular values of A with the reset min(m,n) - f are all 0's, where f $\leq$ min(m, n). Here is an example of m = 5, f = 3, n = 4 to demonstrate the Σ:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This number f is critical, since it defines how many critical factor scales (aka. singular value) is contained in A. We also mentioned before the PCA, a slim version SVD, can help us reduce the rank of A without

compromising critical information according Eckart-Young theorem (Strang 2019). So we can use this f, or some $k \leq f$ more specifically, to shrink the U and V into m by k, and n by k matrix respectively, and this will still enable us to recover a good estimate of A with a reduced size U, V through the matrix multiplication. Here we only abstracted the key idea of SVD (PCA) to help us build a theoretical foundation of our algorithm, for further details on the topic, the book by Strang (2016) gives a comprehensive explanation. Now let's apply the standard SVD formula into our scenario, and rewrite it as:

$$R = PIQ^T$$

Then R is a user_num(rows) by movie_num(cols) rating matrix, P is a user_num(rows) by f(cols) matrix, I is a f(rows) by f(cols) identity matrix, Q is a movie_num(rows) by f(cols) matrix. And the f is the key to derive our "latent factor" algorithm. We are assuming the f number is known to us, and there are same f hidden factors pertained to each user and each movie, according to the matrix multiplication in above, for each element in R can be arrived by:

$$r_{ui} = \vec{p_u}^T \cdot 1 \cdot \vec{q_i}$$

The 1 between $\vec{p_u}^T$ and $\vec{q_i}$ can be neglected. we also need to clarify that the $\vec{p_u}^T$ is the u-th row of matrix P in above, and same for $\vec{q_i}^T$ is the i-th row of matrix Q, so the transpose on Q returns the $\vec{q_i}$ back to vertical. And the length for every $\vec{p_u}$ and $\vec{q_i}$ is the same to be f. With the above understanding, we can remake the R matrix to be more convenient to serve our purpose:

$$R_{lfm} = P^T Q$$

Where P is a factor_num by user_num matrix, and Q is a factor_num by movie_num matrix, so R_{lfm} is a user_num by movie_num matrix, and for notation convenient, we will use R to indicate R_{lfm} in reset of the report.

By inspecting the R matrix equation, to make a good estimation on any R element r_{ui}, we need the corresponding $\vec{p_u}$ and $\vec{q_i}$. So how can we get this latent factor vector for each user and movie? Bell, Koren, and Volinsky (2007) solved it by the idea of minimising the loss function $\frac{1}{2}\|P^T Q - R\|_F^2$ by alternating gradient descent, and the alternating iterations here beautifully solved the problem of both P and Q are unknown to us at beginning of learning, the iteration will be demonstrated like this:

$$P_{t+1} = P_t - \underbrace{Q_t \cdot (P_t^T Q_t - R)^T}_{\nabla P_t}$$

$$Q_{t+1} = Q_t - \underbrace{P_t \cdot (P_t^T Q_t - R)}_{\nabla Q_t}$$

The above $P_t \to Q_{t+1} \to P_{t+2}$ alternations are not efficient with our sparse matrix scenario though, we just use them as a demonstration, and we will tweak them to fit our purpose later, when we get into the details of our calculation. For more details on gradient descent and why it works, the book by Strang (2019) covers all the essentials. To have an intuitive understanding about gradient descent, geometrically we are having a convex loss function, because $\frac{1}{2}\|P^T Q - R\|_F^2 \geq 0$ is positive semidefinite, ideally a rather simple case is this loss function > 0 a positive definite matrix, then it is a bowl opening up (convex up), the bottom point inside the bowl is the minimum of the loss function we are working hardly to touch. On the other hand, at this bottom point, our target RMSE function is minimised:

$$RMSE_{matrix} = \|P^T Q - R\|_F$$

Up to now, we have built enough understanding to proceed with our algorithm, but we still miss one important part, the number f of latent factors to construct each user and movie vector, we will leave it at the end inside the last step of the computation. With the above foundation, we are ready to get into the computations now.

Centralisation

Instead applying our latent factor algorithm directly, we need to centralise user-movie rating matrix first. Traditionally, we only centeralise the measurements vectors in SVD. In our rating matrix, there are tow kinds of measurements, users and movies. If we treat each user as a measurement, then each movie is a sample point, conversely, we treat each movie as a measurement, then each user is a sample point.

$$R = \begin{bmatrix} \vec{User_1}^T \\ \vec{User_2}^T \\ \vdots \\ \vec{User_u}^T \end{bmatrix} = \begin{bmatrix} r_{11} & \dots & r_{1i} \\ \vdots & \ddots & \vdots \\ r_{u1} & \dots & r_{ui} \end{bmatrix} = \begin{bmatrix} \vec{Movie_1} & \vec{Movie_2} & \dots & \vec{Movie_i} \end{bmatrix}$$

So in our rating matrix, we have to centralise in both row and column directions simultaneously. To stress the simultaneous centralisation, we are not performing the de-mean on one direction then the other, instead we treat the rating r_{ui} to be on its own r-scale in a third dimension apart from the user-movie dimension, because r_{ui} is the result of some mapping function $f(\vec{user_u}, \vec{movie_i})$. One more thing to consider is the sparsity of our rating matrix, there are very few r_{ui} are known to us, and we don't know whether the user watched the movie, or she just could not be bothered to rate it. So instead imputing the missing value in the matrix with 0's, we just exclude them from our computation in here. The other reason of doing so is, we will otherwise get a 0 mean, because those 0's adding no value to the sum of all ratings, but making our divisor enormously large.

```
#global mean
g_mean <- mean(edx$rating)
```

The "global mean" we just get is the centre of all the available ratings, so it is neither the mean of any user nor mean of any movie. But we can assume this global mean is the centre of all the means of every user, as well as cenre of all the means of each movie, alternatively we can call it the mean of all users' means, and mean of the all movies' means, so we can estimate each user's real mean by $User_u\_mean = g\_mean + u\_bias$.

How can we find the best u_bias for each user? We aim to minimise the $RMSE$:

$$RMSE = \sqrt{\frac{\sum_{ui \in N} (\hat{r}_{ui} - r_{ui})^2}{N}}$$

For the sake of simplicity, we will only work on the inner part (squared error), and construct our loss function:

$$Loss = \frac{1}{2N} \sum_{ui \in N} (\hat{r}_{ui} - r_{ui})^2$$

we are aiming for the best u_bias so that RMSE can be minimised with the set up of $\hat{r}_{ui} \sim g\_mean + u\_bias$.

And what's more, there are active users who rates every movie she watched, and lazy users who rarely rate, so we introduce the regularisation factor λ to avoid the overfitting risk, and the new regularised loss function becomes:

$$Loss = \frac{1}{2} \sum_{i \in N} (\hat{r}_{ui} - r_{ui})^2 + \frac{\lambda}{2N} \sum_{i \in N} b_u^2$$

We dropped the u from ui in the Σ sign, because we are estimating each user's bias individually based on all the movies rated by her. We also dropped the N from the first part of $\frac{1}{2N}$ to make our computation formula of b_u to be consistent with the Professor Irizarry's text book (Irizarry 2022), some may prefer to keep it, that is just personal choice, it will only alter the λ result, but the final b_u should not vary much.

The derived formula to perform the following b_u calculation is:

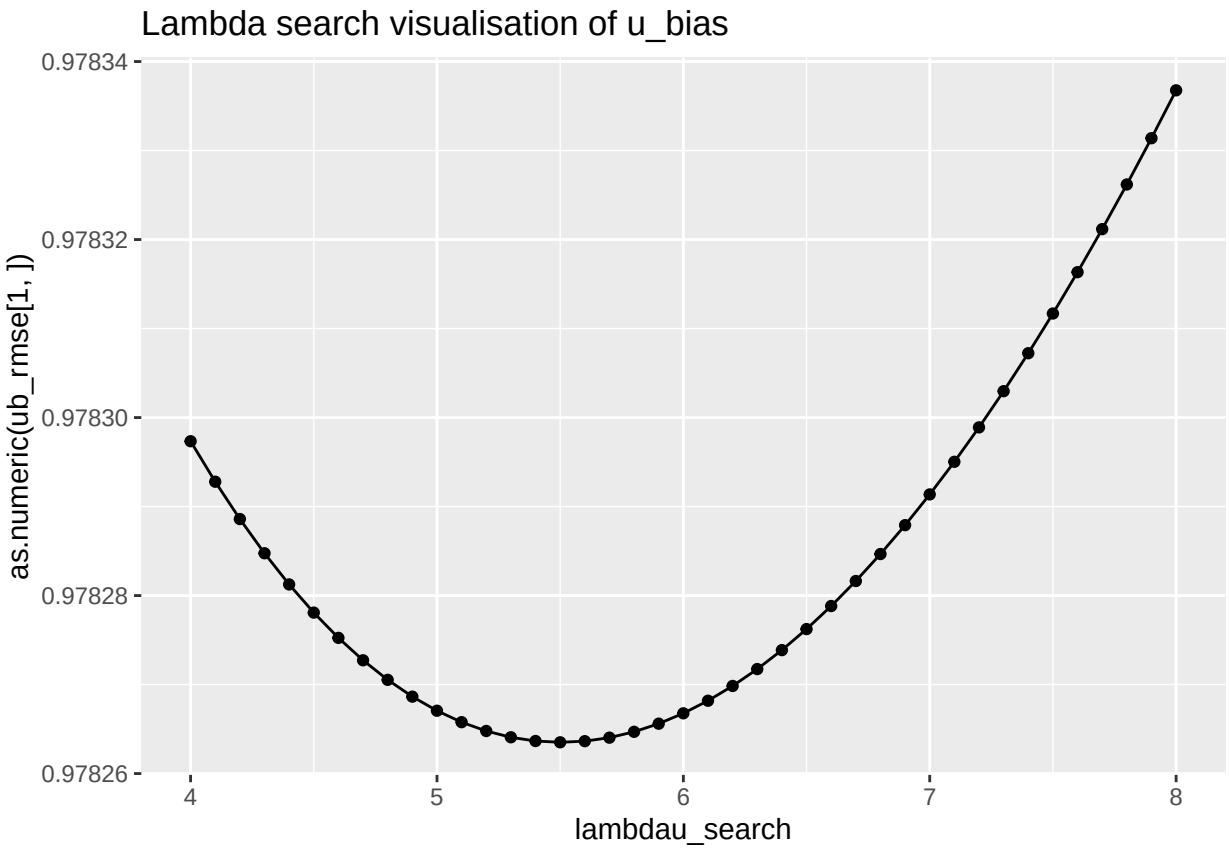
$$b_u = \frac{\sum_{i \in N} (r_i - \bar{r})}{\lambda + N}$$

We also employ a 5-fold cross validation method to overcome potential overfitting.

```
#make 5-folds cross validation sets

set.seed(2, sample.kind= "Rounding")

ind_cv <- createFolds(edx$userId, k = 5)

#utilize the multicores on our computer to parallel the loop
#I am using total cores -1 to avoid system freeze
#number of cores only need to be registered once globally

registerDoParallel(cores = 3) #register 3 mores for parallel computing

train_cv <- foreach(k = 1:5) %dopar% {edx[-ind_cv[[k]],]}

test_cv <- foreach(k = 1:5, .packages = "tidyverse") %dopar% {
    edx[ind_cv[[k]],] %>%
    semi_join(train_cv[[k]], by = "movieId") %>%
    semi_join(train_cv[[k]], by = "userId")}

#We did not put back the dropped out data back to train_cv,
#because they are eventually showing up in other folds.
#This is just personal choice, we can put them back,
#but it won't cause major change in result.

rm(ind_cv)
```

We use the generated folds in above to search the best λ:

```
lambdau_search <- seq(4, 8, 0.1)

#try all the possible lambdas, and result in a RMSE table

ub_tune <- foreach(l = lambdau_search, .combine = "cbind.data.frame") %:%
  foreach(k = 1:5, .combine = "c", .packages = "data.table") %dopar% {
    u_bias <- train_cv[[k]][
      , .(u_bias = sum(rating - g_mean) / (l + .N)), by = .(userId)]

    pred <- u_bias[test_cv[[k]], on = .(userId)][
      , .(err = g_mean + u_bias - rating)]

    sqrt(mean(pred$err * pred$err))
  }

setDT(ub_tune)
setnames(ub_tune, as.character(lambdau_search))

#take the mean value over each column
#to get the final cross validation rmse result for each lambda
ub_rmse <- ub_tune[, lapply(.SD, mean)]
```




```
#take the lambda with the best cross validation RMSE
#apply it to the u_bias formula
lambda_u <- lambdau_search[which.min(ub_rmse[1,])]

u_bias <- edx[, .(u_bias = sum(rating - g_mean) / (lambda_u + .N)),
                        by = .(userId)]

rm(ub_rmse, ub_tune, lambdau_search, lambda_u)
```

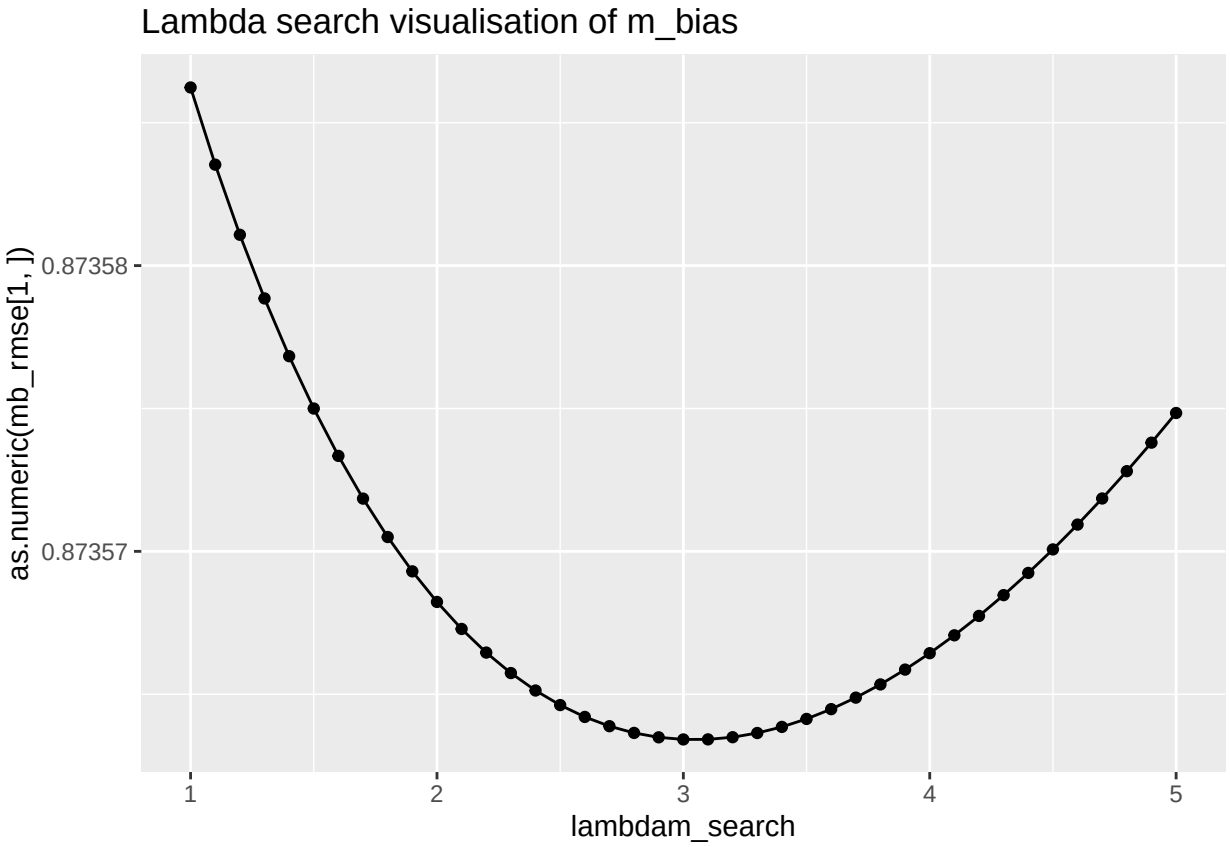
We can use the same method to find the bias of each movie, so that our mean is adjusted not only toward each user real mean, but also toward each movie real mean. So that $\hat{r}_{ui} \sim g\_mean + u\_bias + m\_bias$, and the b_m is:

$$b_m = \frac{\sum_{u \in N} (r_u - \bar{r} - b_u)}{\lambda + N}$$

From the above formula we can tell, the movie's real mean varies on each user. It is not like user's mean is reflecting each user's own perspective, but movie's real mean is a combination of all watchers different perspectives, so the above formula takes the rater's perspective into account.

```
lambdam_search <- seq(1, 5, 0.1)

#again try all the possible lambdas, and result in a RMSE table

mb_tune <- foreach(l = lambdam_search, .combine = "cbind.data.frame") %:%
  foreach(k = 1:5, .combine = "c", .packages = "data.table") %dopar% {
    m_bias <- u_bias[train_cv[[k]], on = .(userId)][
```

```
      , .(m_bias = sum(rating - g_mean - u_bias) / (l + .N)),
      by = .(movieId)]

    pred <- m_bias[u_bias[test_cv[[k]],
                          on = .(userId)],
                   on = .(movieId)][
                     , .(err = g_mean + u_bias + m_bias - rating)]

    sqrt(mean(pred$err * pred$err))
  }

setDT(mb_tune)
setnames(mb_tune, as.character(lambdam_search))

#take the mean value over each column
#to get the final cross validation rmse result for each lambda
mb_rmse <- mb_tune[, lapply(.SD, mean)]
```




```
#take the lambda with the best cross validation rmse
#apply it to the u_bias formula
lambda_m <- lambdam_search[which.min(mb_rmse[1,])]

m_bias <- edx[u_bias, on = .(userId)][
  , .(m_bias = sum(rating - g_mean - u_bias) / (lambda_m + .N)),
  by = .(movieId)]
```

```
rm(mb_tune, mb_rmse, lambdam_search, lambda_m, train_cv, test_cv)
```

By now, we have finished the centralisation process under $\bar{r}_{ui} = \bar{r}_g + b_u + b_m$. We have to stress that the mean estimation $\bar{r}_{ui}$ *varies on different user-movie combinations*, so it can be generalised across all the elements of our Rating matrix, this value can always be generated if we have history on the specified user and movie. This generalisation property makes it to be versatile in prediction.

Genres Bias

userId	movieId	rating	timestamp	title	genres
1	364	5	1996-08-02	Lion King, The (1994)	Adventure|Animation|Children|Drama|Musical

We use the beginning example to remind ourselves that, the "genres" information of each movie are given, simultaneously we can make an assumption that each user also has some preferences toward some particular genres, we call it user genre bias g_u. Instead of being a single numeric value, we assume that $\vec{g_u}$ is a vector, why? That vector assumption is inspired from the movie genres. In above "The Lion King" example, we see the genres are categorised as:

```
## [1] "Adventure|Animation|Children|Drama|Musical"
```

One movie has multiple genres, but we can not quantify which genre is the dominant one. This multi-genres attribute is also applicable to users, like the same example we just used, we can not tell the user "1" rated "The Lion King" for "5" star is because she likes the the adventure part, the animation , or the musical part of the movie, we don't know what her dominant genre taste is. And it is also possible like the case in movie, she likes all the 5 genres in the movie, but out of these genres she might have stronger preference on musical genre over the rest.

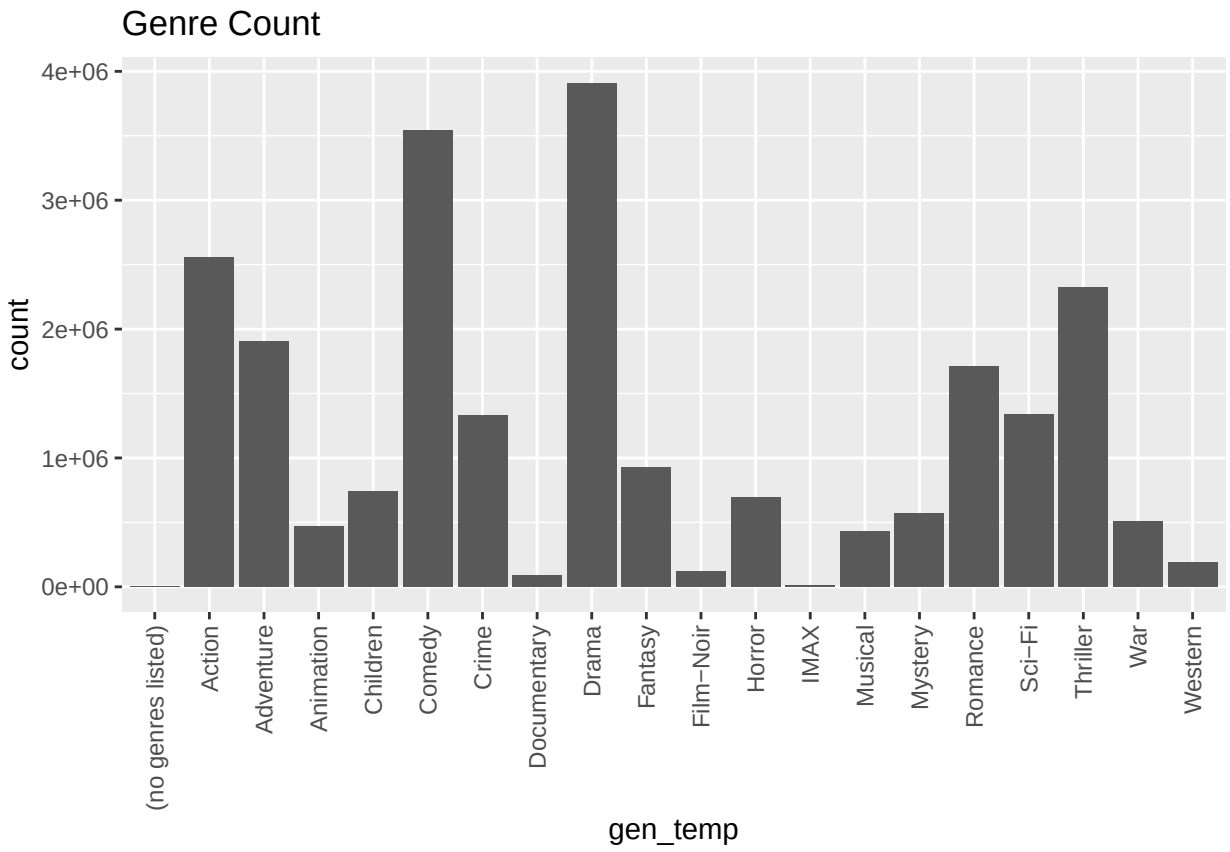
Now we have enough reason to not only take the user genre $\vec{g_u}$ as a vector, but also treat the movie genre as a vector $\vec{g_m}$ too. So to combine them together, we assume there is a genre bias $b_g = \vec{g_m} \cdot \vec{g_u}$. Next let's find the $\vec{g_m}$ for each movie, then use those movie $\vec{g_m}$ to find $\vec{g_u}$ for each user, after all we can use them to construct the b_g in regard to different "userId"-"movieId" pairs in our prediction.

Since we are given the genres of movie already, so we start with constructing each movie's genre vector $\vec{g_m}$ first. Let's begin with some data cleaning on the genres column to remove the "|".

```
#every genres[[k]] contains all the genres names of one movie
genres <- str_split(edx$genres, "\\|")

#a char vector has all the genres names without duplication
gen_cat <- genres %>% unlist() %>% unique()
```

We then can have a visualisation on genres count, the top Hollywood capital favored genres are drama comedy and action. We will get a residual mean for each genre in next. The residual mean is based on the residual by subtracting the $\bar{r_g}$(global mean), b_u, and b_m from the r_{ui}ratings.

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



For any genre x, it's residual mean is calculated as:$\bar{X}_{gen} = \frac{1}{N}\sum_{x \in N}(\bar{r_g} + b_u + b_m - r_x)$. One thing to mention that, this movie genres mean is highly biased, for the "The Lion King" example, it is categorised in 5 different genres, so this movie will be used in all those 5 genres mean calculation.

```
n <- length(gen_cat)

#calculate mean of each genre
#one movie can be repeatedly used in multi genres mean calculation
gen_mean <- foreach(g = 1:n, .combine = "c",
                    .packages = "data.table") %dopar% {
                      u_bias[m_bias[edx, on = .(movieId)],
                             on = .(userId)][genres %like% gen_cat[g],
                              mean(g_mean + u_bias + m_bias - rating)]}
rm(n)

m_id <- unique(edx$movieId)
m_n <- length(m_id)

#a list contains genres char vector for each movieId without duplication
m_gen <- foreach(i = 1:m_n, .packages = c("stringr", "data.table")) %dopar% {
  gens <- edx[movieId == m_id[i], genres][1]
  str_split(gens, "\\|") %>% unlist()
}

names(m_gen) <- m_id
```

The next code chunk may be a bit confusing at beginning, what it does is to create a data frame that contains all the movie's genres means. We can see from the "The Lion King" example, the movie column has it's categorised genres rows filled with that genre's mean just calculated in above, and the rest genres rows are zero.

movieId	title	genres
364	Lion King, The (1994)	Adventure|Animation|Children|Drama|Musical

We only show 10 of the genres in below for demonstration.

```
##      gen_cat          364
## 1     Comedy 0.0000000000
## 2    Romance 0.0000000000
## 3     Action 0.0000000000
## 4      Crime 0.0000000000
## 5   Thriller 0.0000000000
## 6      Drama 0.0001968037
## 7     Sci-Fi 0.0000000000
## 8  Adventure 0.0003959867
## 9   Children 0.0008421342
## 10   Fantasy 0.0000000000
```

```
#convert the movies genres name char vector into a numeric vector
#each element corresponds to the index number in the genres names vector
#the gen_cat we constructed in genre name cleaning code chunk
ind <- foreach(g = 1:m_n, .packages = "foreach") %dopar% {
  foreach(i = 1:length(m_gen[[g]]), .combine = "c",
          .packages = "stringr") %do% {
            str_which(gen_cat, m_gen[[g]][i])}}

gen <- data.frame(gen_cat)

#the m_n was defined in last chunk.
j <- 1
while(j <= m_n){
  gen[, j+1] <- gen_mean
  gen[-ind[[j]], j+1] <- 0
  j <- j + 1
}

colnames(gen)[-1] <- m_id

rm(j, genres, ind, m_gen, gen_cat, gen_mean, m_n)
```

As we stated at the beginning of this secession, we are looking for $\vec{g_m}$ and $\vec{g_u}$, so now we have got the $\vec{g_m}$ for each movie in the columns of above table. Now we start to look for the $\vec{g_u}$ for each user.

We made an assumption in above that, each genre bias $b_g = \vec{g_m} \cdot \vec{g_u}$, so linear algebraically we can treat the elements of $\vec{g_u}$ are the coefficients of the elements of $\vec{g_m}$, then each user-movie pair $b_g = \sum_{i=1}^{20} g_{ui} \times g_{mi}$. To find the g_{ui}, we have got the g_{mi} already, the b_g is in the residual of $\bar{r_g} + b_u + b_m - r_x$, under the assumption of $r_{ui} \sim \bar{r_g} + b_u + b_m + b_g$.

We will use "Elastic-Net" method to fit the $b_g = \vec{g_m} \cdot \vec{g_u}$ to get the wanted $\vec{g_u}$. We need to have some intuitive understanding with "Elastic-Net" first, before we proceed further.

Let's look at the loss function structure of "Elastic-Net" first (Friedman, Hastie, and Tibshirani 2010):

$$Loss_{en} = \frac{1}{2N}\sum_{m\in N}\|y_{ui} - \beta_{u0} - \vec{x}_m^T\vec{\beta}_u\|_F^2 + \lambda\Big[(1-\alpha)\frac{\|\vec{\beta}_u\|_F^2}{2} + \alpha\sum_{j\in N}\|\vec{\beta}_j\|_2\Big]$$

This loss function may look a bit scary at the first glance, but indeed it is just a normal quadratic loss $\frac{1}{2N}\sum_{m\in N}\|y_{ui} - \beta_{u0} - \vec{x}_m^T\vec{\beta}_u\|_F^2$ with a "lasso-ridge hybrid" regularisation term. In the previous b_u and b_m part, we have had lots of experiences with the ridge regression regularisation already, which is evolved into $\frac{\lambda(1-\alpha)}{2}\|\beta\|_F^2$ over here, then the last part is a lasso $\lambda\cdot\alpha\sum_{j\in 20}\|\beta_j\|_2$. So geometrically, this "lasso-ridge" hybrid regularisation form is a shape between a ridge circle and a lasso diamond on a two dimension contour, it is more like a diamond lasso with a slight rounded sticking out vertex. Have a geometric understanding can help us better grasp the reason for us choosing it for feature fitting. The more detailed explanations can be found in the original paper by Zou and Hastie (2005).

Because of this "lasso-ridge" hybrid structure, making it to be specially useful in feature selection under a sparse situation (Friedman, Hastie, and Tibshirani 2010). That property is just a perfect fit in our scenario, we have a sparse rating matrix, we want to fit the data with the selected genres feature to construct a $\vec{g_u}$ for each user. We can easily implement this method by the 'glmnet' package.

To use the 'glmnet' we need to make some clarifications in the above formula in our purpose:
1. the y_{ui} is the rating element in a "movieId"x"userId" rating matrix, which is just a transpose of R we used in above.
2. $\vec{x}_m^T$ is a row vector with its elements are all 20 genres (including the "no genres listed": empty genres for that movie) we have in the dataset, but with only that categorised genres of the movie filled with the respective genres means, and reset genres slots are all zero. Simply put, they are just rows of "gen" matrix we have constructed in above.
3. β_0 and $\vec{\beta}_u$ is a set of value to be returned by the fitting, they are in together to represent one single user's genres structure. We used the word structure to stress that, the $\vec{\beta}_u$ has the same length as $\vec{x}_m$, we also assume the user's genres preferences can be categorised in a mixture of these 20 genres we have. And the β_{u0} here is to be treated as a central point of all the genres to a user.
4. The λ is acting as the traditional penalty coefficient to prevent overfitting. The nice thing about using 'glmnet' package is that, it tunes the λ by cross validation automatically, and applies it to get us a well regularised fitting result.
5. α is a weighting factor to provide the flexibility for us to maneuver the penalty term to lean toward a pure ridge regression ($\alpha = 0$), or a pure lasso ($\alpha = 1$), we just use the hybrid form here ($\alpha = 0.5$).
6. The last thing we need to pay attention is the group lasso $\alpha\sum_{j\in N}\|\beta_j\|_2$, the N is total number of movies we have, the $\|\vec{\beta}_j\|_2$ is the ι_2 norm for each user-movie paired rating with a fixed "userId", because we use movie history to estimate user genres, and the length of each $\vec{\beta}_j$ is 20, which is the total number of genres we have.

After we clarified the data structure, then we will proceed to the calculations in below. One thing to note that, if you are about to run the code, be prepared it can take you about 20 hours or more, that is the amount of time I got on my windows machine.

The next code chunk will give us a residual table with each non NA element is filled by $\bar{r_g} + b_u + b_m - r_{ui}$. We get the NA's in the rest, so we are making a big "movieId" by "userId" sparse table.

```
residual_train <- u_bias[m_bias[edx, on = .(movieId)]
                         , on = .(userId)][
                           , .(resid = g_mean + u_bias + m_bias - rating),
                           by = .(userId, movieId)]

rtable <- dcast(residual_train, userId ~ movieId, value.var = "resid")

movieId <- names(rtable[,-1])
```

```
rtable_tr <- transpose(rtable, keep.names = "movieId",
                       make.names = "userId")

rtable_tr$movieId <- as.numeric(rtable_tr$movieId)

m_id_dt <- data.table(movieId = m_id)

rtable <- rtable_tr[m_id_dt, on = .(movieId)]

rm(rtable_tr, movieId, m_id_dt)

#turn the previous made movie by user table into a sparse matrix
rtable_y <- setnafill(rtable[,-1], type = "const", fill = 0)
rtable_y <- as(as.matrix(rtable_y), "sparseMatrix")

#turn the movie genres table in above into a sparse matrix
#we use the result movie by genres mean matrix as the predictor in model
gen_x <- as(as.matrix(gen[,-1]), "sparseMatrix")
gen_x <- t(gen_x)

#break the big rating matrix into user batches with size of 1000
set.seed(3, sample.kind = "Rounding")
ind_y<- createFolds(1: rtable_y@Dim[2], k = ceiling(rtable_y@Dim[2]/1000))

k <- length(ind_y)

#an empty list container to be filled by the fitting results in next
u_beta <- list()

#alpha = 0.5 to make the model to be a standard "Elastic Net"
#the "mse" defines our loss function to be quadratic
#record the best result \beta with the "lambda.min"
for(k in 1:k){
  fit <- cv.glmnet(gen_x, rtable_y[, ind_y[[k]]],
                   family = "mgaussian",
                   type.measure = "mse",
                   nfolds = 5, alpha = 0.5,
                   parallel = TRUE, trace.it = TRUE)
  u_beta[[k]] <- coef(fit, s= "lambda.min")
  rm(fit)
  gc()
}

rm(k, ind_y)

u_beta <- unlist(u_beta)
```

```
#each u_beta$"userId" is a 21x1 vector for the userId = k
#1st element named interception is \beta_0
#the 2nd to 21th elements named as V1 - V20 is the vector \beta_u

u_beta$"1"
```

```
## 21 x 1 sparse Matrix of class "dgCMatrix"
##                            1
## (Intercept) -0.0003314782
## V1          -0.8264282329
## V2          -0.8526669824
## V3          -4.2757890417
## V4          -1.3346781237
## V5          -0.9836084615
## V6           2.3497552442
## V7          -0.7515988033
## V8           1.1775253830
## V9          -3.8585053737
## V10         -1.6285328838
## V11          3.6571686258
## V12         -1.4407546161
## V13         -0.2262983186
## V14          1.3141850984
## V15          2.0628985891
## V16         -0.1886050341
## V17          0.2625385818
## V18         -0.0524170985
## V19          .
## V20          .
```

Latent Factors

Up to now, we have done data centralisation, and used the movie genres mean to derive the user genres, which can be used to construct $b_g = \vec{g}_m \cdot \vec{\beta}_u + \beta_{0u}$, so for each pair of 'userId' and 'movieId', we can make a very rough prediction based on $\hat{r}_{ui} \sim \bar{r}_g + b_u + b_i + b_g$. This subsection will get us the last element lf_{ui} to complete our model $\hat{r}_{ui} \sim \bar{r}_g + b_u + b_i + b_{gui} + lf_{ui} + \epsilon$.

At the very beginning of this secession, we have had a mathematical justification for this model about why it can help us achieve the result we want. We now can construct a model to serve our purpose on that mathematical ground.

$$Loss = \frac{1}{2}(\vec{p}_u \cdot \vec{q}_i - rsid_{ui})^2 + \lambda(\|\vec{p}_u\|_2 + \|\vec{q}_i\|_2)$$

$$rsid_{ui} = \bar{r}_g + b_u + b_i + b_g - r_{ui}$$

We will derive the best $\vec{p}_u$ and $\vec{q}_i$ for each user and movie respectively, so that our loss function is minimised. To note that, instead of the R, P, Q matrix, we use the vectors and single rating element values in our loss function, this kind of set up will be more convenient for our sparse case. So the learning iterations are also in a vector form to fit our needs.

$$\vec{p}_{t+1} = \vec{p}_t - lr \cdot \nabla_p L$$

$$\vec{q}_{t+1} = \vec{q}_t - lr \cdot \nabla_q L$$

$$\nabla_p L = err \cdot \vec{q}_t + \lambda \cdot \vec{p}_t$$

$$\nabla_q L = err \cdot \vec{p}_t + \lambda \cdot \vec{q}_t$$

$$err = \vec{p}_t \cdot \vec{q}_t - rsid_{ui}$$

This set formulas describe the whole calculation process, but in a reverse order. This way of ordering can more clearly explain themselves. To be success in computation, we also need to tune some hyperparameters first:

1. **f**: is the factor length for each $\vec{p}_u$ and $\vec{q}_i$, which we mentioned in the beginning of the "Theoretical Foundations" subsection without a solution. We will tune it first to construct $\vec{p}_u$ and $\vec{q}_i$, so that we can

proceed.
2. **lr**: is the learning rate, in some textbook it is also called step size, which will be more intuitive, since it controls how big the stride we are taking for each step in searching. There is a trade off, if the size is too big, not only we will never reach the minimum point of the bowl, but also the result can explode by keep climbing to the infinite maximum. But if we take too small stride each step, then it may take forever to reach the minimum point. So academically, researchers and practitioners are constantly searching and developing new ways of constructing this step size, someone calls it the art in machine learning. But we will only use the most brutal way to tune it by cross validation.
3. λ: is playing as a penalty coefficient to prevent from overfitting in loss function as usual, but it also decide how good we can estimate the lowest point in the loss function bowl. If the **lr** controls the speed of searching/learning in general, then λ controls the precision of learning.

Now we are ready to proceed to the calculation details. the following code will help us to get the $residual_{ui}$ for each user-movie paired ratings in "edx" dataset, after that we use the prepared data to tune those hyperparameters just described.

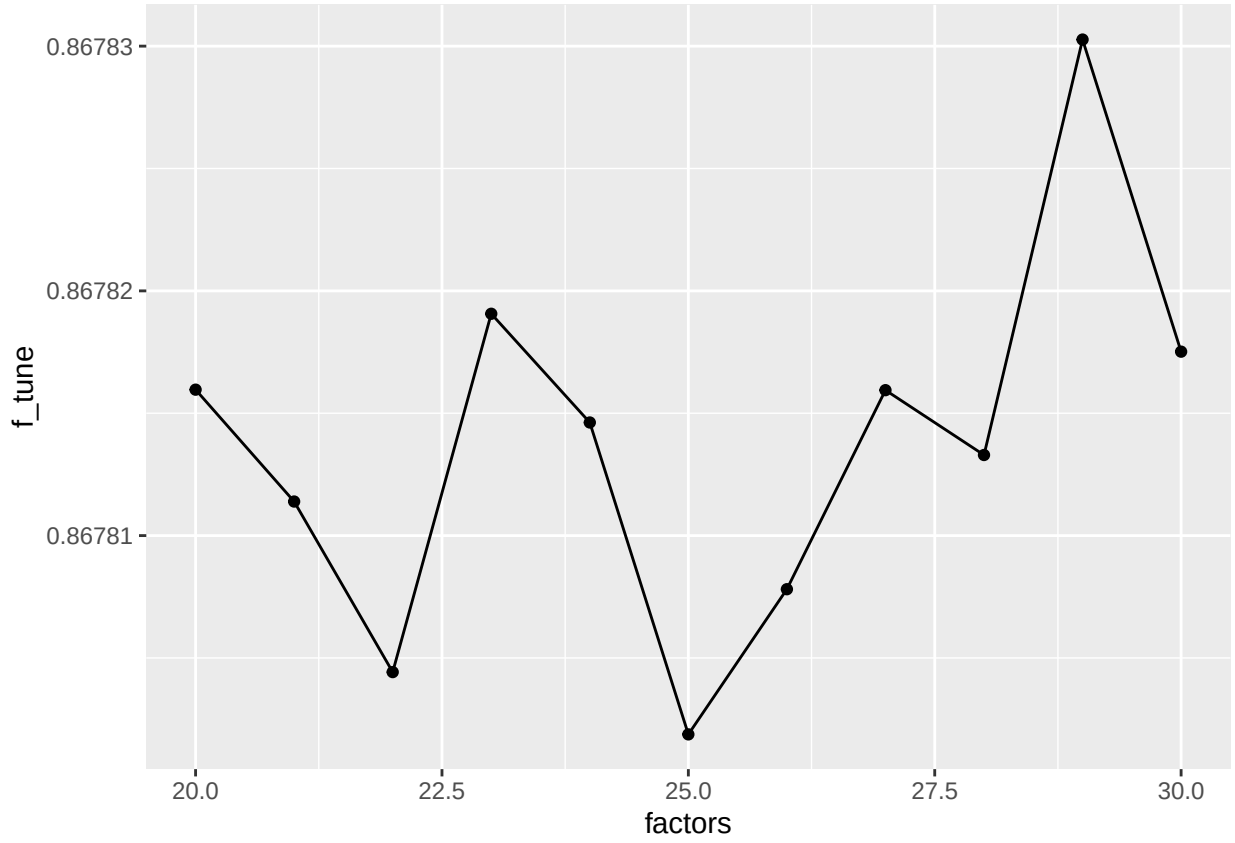
```
uid_train <- edx$userId %>% as.character()
mid_train <- edx$movieId %>% as.character()

#compose gen_bias for the user-movie pair in edx trainning set
ml <- length(edx$movieId)
gen_bias_train <- foreach(i = 1:ml, .combine = "c",
                          .packages = "Matrix") %dopar% {
  gen[,mid_train[i]] %*% u_beta[[uid_train[i]]][-1] +
                              u_beta[[uid_train[i]]][1]}

rm(ml, uid_train, mid_train)

#create a new column called "err" = rsid
#!!be aware the "gen_bias" is subtracted in here, because it was trained using
#residual = mean + b_u + b_m - rating!!
resid_gen <- m_bias[u_bias[edx[, .(userId, movieId, rating)][
  , gen_bias := gen_bias_train], on = .(userId)], on = .(movieId)][
    , err := g_mean + u_bias + m_bias - gen_bias - rating]

#turn the 'err' = rsid column into a wide table with size of user# X movie#
rtable_gen <- dcast(resid_gen, userId ~ movieId, value.var = "err")

#save the ordered userId and movieId in training matrix
#will be used for naming the trained output user and movie matrix
uid_gen <- rtable_gen[,1]
mid_gen <- names(rtable_gen)[-1]

#turn the above table into a sparse matrix, which is more RAM friendly
rtable_gen <- setnafill(rtable_gen[,-1], fill = 0)
rtable_gen <- Matrix(as.matrix(rtable_gen), "sparseMatrix")

R <- rtable_gen@x #training resid
U_i <- rtable_gen@i + 1 #user index of each resid
M_j <- rep(1:rtable_gen@Dim[2], diff(rtable_gen@p)) #movie index of each resid

rm(resid_gen, rtable_gen)
```

```
#####hyperparameters tuning#####
#create train-test set(1 fold)
set.seed(0, sample.kind = "Rounding")

Rid_cv <- createFolds(1:length(R), 5)
Rid_cv <- Rid_cv[[1]] #(just use 1 fold)

train_R <- R[-Rid_cv]
test_R <- R[Rid_cv]

#saved in place for future loop
tr_n <- length(train_R)
tst_n <- length(test_R)

#train-test userid, movieid sets
Ui_tr <- U_i[-Rid_cv]
Ui_tst <- U_i[Rid_cv]

Mj_tr <- M_j[-Rid_cv]
Mj_tst <- M_j[Rid_cv]
```

In hyperparameter tuning, we use **Stochastic Gradient Descent** instead of going through all the 9,000,055 ratings, because SGD is super efficient in terms of learning speed, but not great at convergence. So we only use SGD to find good parameter estimates, but use full size gradient descent (GD) in the final learning. The SGD has exactly the same iterations as normal GD described in above, but only train on a small sample (batch) of the full dataset in every epoch.

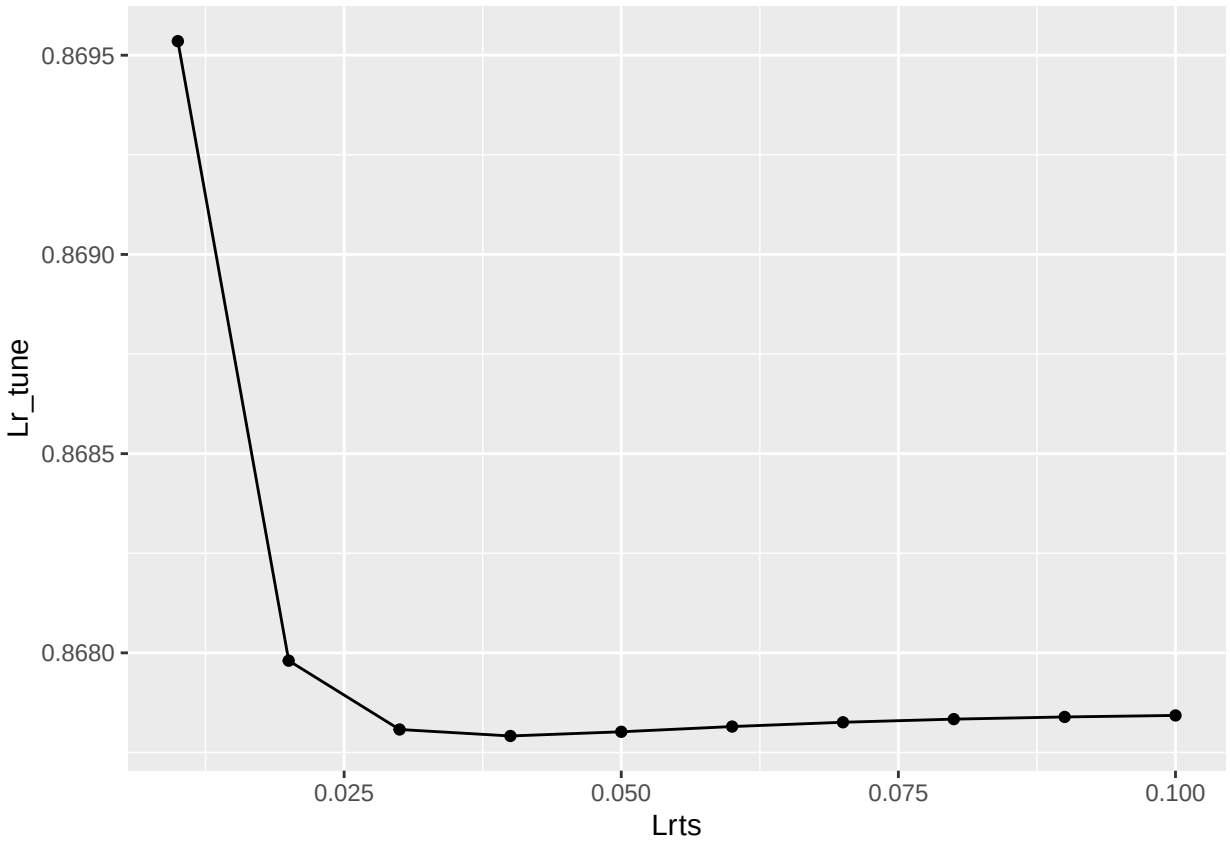
```
###factor length###
#these try on figures are randomly chosen
lambda <- 1
L_rate <- 0.05

#we will iterate batch_size samples each time, and repeat epochs times
epochs <- 1000
batch_size <- 10000

factors <- seq(20, 30, 1)
f_tune <- foreach(f = factors, .combine = "c") %dopar% {
  set.seed(3, sample.kind = "Rounding")
  P <- matrix(runif(f*rtable_gen@Dim[1], 0, 1), nrow = f)
  set.seed(4, sample.kind = "Rounding")
  Q <- matrix(runif(f*rtable_gen@Dim[2], 0, 1), nrow = f)

  for (t in 1:epochs){

    batch_id <- sample(1:tr_n, batch_size, replace = FALSE)

    for (ui in batch_id){

      err_ui <- c(P[, Ui_tr[ui]] %*% Q[, Mj_tr[ui]] - train_R[ui])
      nabla_p <- err_ui * Q[, Mj_tr[ui]]  + lambda * P[,Ui_tr[ui]]
      nabla_q <- err_ui * P[, Ui_tr[ui]]  + lambda * Q[,Mj_tr[ui]]
```

```
      P[, Ui_tr[ui]] <- P[, Ui_tr[ui]] - L_rate * nabla_p
      Q[, Mj_tr[ui]] <- Q[, Mj_tr[ui]] - L_rate * nabla_q
    }
  }
  err <- sapply(1:tst_n, function(j){
    P[, Ui_tst[j]] %*% Q[, Mj_tst[j]] - test_R[j]
  })
  rm(P, Q)
  sqrt(mean(err * err))
}

f_opt <- factors[which.min(f_tune)] #save the tuned factor length for future

rm(f_tune, factors) #can be kept for plotting
```



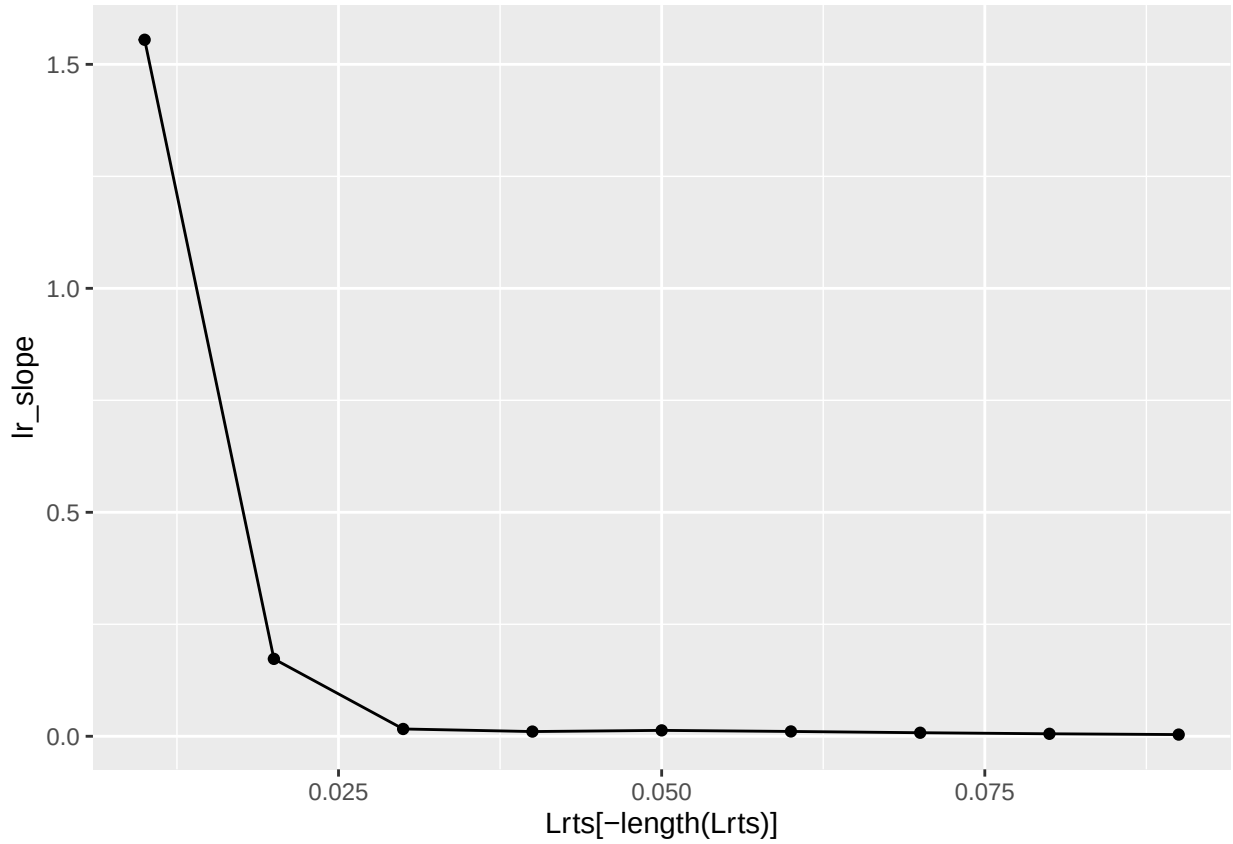

```
###learning rate###
rm(L_rate)

Lrts <- seq(0.01, 0.1, 0.01)
Lr_tune <- foreach(L_rate = Lrts, .combine = "c") %dopar% {
  set.seed(3, sample.kind = "Rounding")
  P <- matrix(runif(f_opt*rtable_gen@Dim[1], 0, 1), nrow = f_opt)
  set.seed(4, sample.kind = "Rounding")
  Q <- matrix(runif(f_opt*rtable_gen@Dim[2], 0, 1), nrow = f_opt)
```

```
  for (t in 1:epochs){

    batch_id <- sample(1:tr_n, batch_size, replace = FALSE)

    for (ui in batch_id){

      err_ui <- c(P[, Ui_tr[ui]] %*% Q[, Mj_tr[ui]] - train_R[ui])
      nabla_p <- err_ui * Q[, Mj_tr[ui]]  + lambda * P[,Ui_tr[ui]]
      nabla_q <- err_ui * P[, Ui_tr[ui]]  + lambda * Q[,Mj_tr[ui]]

      P[, Ui_tr[ui]] <- P[, Ui_tr[ui]] - L_rate * nabla_p
      Q[, Mj_tr[ui]] <- Q[, Mj_tr[ui]] - L_rate * nabla_q
    }
  }
  err <- sapply(1:tst_n, function(j){
    P[, Ui_tst[j]] %*% Q[, Mj_tst[j]] - test_R[j]
  })
  rm(P, Q)
  sqrt(mean(err * err))
}
```



The learning-rate tuning rmse plot does not have a turning point, so instead we will locate the best rate by their slopes, when the tuning rmse slopes get smaller, the improvement by trying the next learning rate fades. Then we can define the first learning rate, whose slope drops under certain level (we choose slope $<$ 0.1 in here), to be our choice of optimal learning rate.

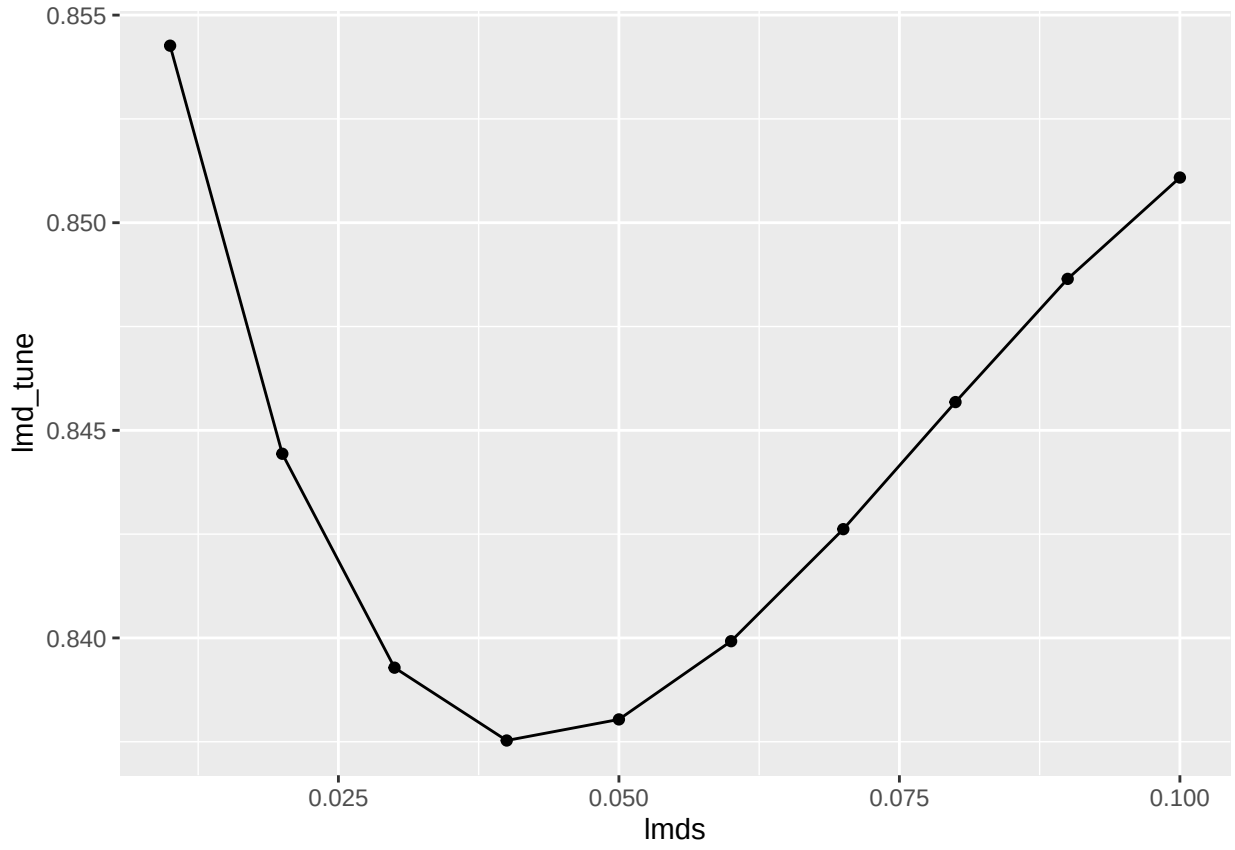
```
lr_slope <- sapply(1:(length(Lr_tune) - 1), function(k){
  abs(Lr_tune[k+1] - Lr_tune[k]) / 0.001})

#set our selection level at 0.1
#a smaller number, but will slow down the final learning process
L_rate_opt <- Lrts[which(lr_slope < 0.1)][1]

rm(Lrts, Lr_tune, lr_slope)#can be kept for plotting
```



In next, we will use the the full GD method instead of SGD to find the best λ, but there is one challenge in computation cost. As mentioned before, the λ decides how close our learning result can be to the real minimum point, for the best result, we use the full gradient descent (GD) method instead of SGD to tune the *lambda* to mimic our final learning computation process, by saying mimic is because we will only train for very few epochs. But the full size training data is too big, so the computation cost is very high. In addition, the R is known for slow loop, therefore we are using the 'RcppArmadillo' package to write the gradient descent loop in C++, and source the C++ code back to R as a function. This C++ function is named as 'gdtune()' in below code chunk, the C++ code can be found in Appendix at the very end of this report. The manual to use the C++ code can be found on the 'Rcpp' package cran page.

```
###lambda###
rm(lambda)

#create P, Q with tuned already factor length = f_opt
set.seed(3, sample.kind = "Rounding")
P <- matrix(runif(f_opt*rtable_gen@Dim[1], 0, 1), nrow = f_opt)
set.seed(4, sample.kind = "Rounding")
```

```r
Q <- matrix(runif(f_opt*rtable_gen@Dim[2], 0, 1), nrow = f_opt)

lmds <- seq(0.01, 0.1, length.out = 10)

#https://dirk.eddelbuettel.com/code/rcpp.armadillo.html
#https://www.rcpp.org/ for C++ in R maunal details
lmd_tune <- sapply(lmds, function(lmd){
  gdtune(P = P, Q = Q, ytr = train_R, ytst = test_R,
         Uitr = Ui_tr, Mjtr = Mj_tr, Uitst = Ui_tst, Mjtst = Mj_tst,
         L_rate = L_rate_opt, lambda = lmd, epochs = 5)
  })#function returns err^2

lmd_tune <- sqrt(lmd_tune / length(test_R))#turn the results into rmse
```




```r
#take the the lowest point in the valley to be our lambda choice
lmd_opt <- lmds[which.min(lmd_tune)]

rm(P, Q, lmds, lmd_tune)#save for plot
```

By now, we have tuned all the important hyperparameters to be ready to give us an expected good result. Rigorously speaking, we have not done all the parameters yet, that is because our training dataset size allowing us to risk the epochs (iteration steps) number by guessing. The industrial practice may require the epochs to be tuned as well, because too large epochs may cause the model to be over trained, over training will also increase prediction error like overfitting (Nielsen 2015).

This final piece of code to put all the tuned hyperparameters together is also written in C++, and sourced as function 'gd()', the C++ code is put in Appendix at very end of report.

```
#the user matrix P, and movie matrix Q are generated inside the 'gd()' function
#the seed for C++ sourced function need to be set in R
#loop the model for 50 epochs
set.seed(5, sample.kind = "Rounding")
pq <- gd(U_i = U_i, M_j = M_j, y = R,
         u_n = rtable_gen@Dim[1], m_n = rtable_gen@Dim[2],
         factor_n = f_opt, L_rate = L_rate_opt,
         lambda = lmd_opt, epochs = 50)
```

The 'gd()' function returns a list we named as **pq**, which contains trained user matrix P with each column representing an 'userId', and movie matrix Q with each column representing a "movieId". We can use the code below to check any "nan" value in them, the "nan" is the result of training explosion, which means either the learning rate or the λ is too big, then we need to re-tune them.

```
sum(is.nan(pq$P))
```

```
## [1] 0
```

```
sum(is.nan(pq$Q))
```

```
## [1] 0
```

We need to name the matrix P, Q's columns by "userId" and "movieId", which will be convenient in the final prediction composition. We have the ordered "userId" and "movieId" vectors recorded in the rsid matrix code part in above. Because we trained them accordingly with each r_{ui}'s corresponding "userId" and "movieId", we can apply the same order "userId" and "movieId" from rating table directly on P, Q.

```
#the uid_gen and mid_gen are saved from the previous data preparation chunk
colnames(pq$P) <- uid_gen$userId %>% as.character()
colnames(pq$Q) <- mid_gen
```

By the end of this session, we have got all the elements in our prediction model $\hat{r}_{ui} = \bar{r}_g + b_u + b_m + b_{gui} + lf_{ui} + \epsilon$, where both b_{gui} and lf_{ui} have to be composed accordingly with the predicting rating's "userId" and "movieId".

Final Result

As described in above, to make a prediction, we have to know the "userId" and "MovieId" of the to be predicted rating, then we can compose the b_{gui} and lf_{ui} accordingly, and put these "userId" and "movieId" specified elements into the prediction model. So We need to use the "userId" and "movieId" from each rating in the "validation" dataset, to call the respective trained elements and vectors, the following prediction composition code chunk is implemented by this concept.

```
#get the 'userId' and 'movieId' from validation dataset
uid_val <- validation$userId %>% as.character()
mid_val <- validation$movieId %>% as.character()

i <- length(validation$rating)
```

```
#use the 'userId' and 'movieId' to call
#the respective movie and user gen vectors to construct gen_bias(b_gui)
gen_bias <- foreach(i = 1:i, .combine = "c") %dopar% {
  gen[,mid_val[i]] %*% u_beta[[uid_val[i]]][-1] + u_beta[[uid_val[i]]][1]
}

#call the user/movie id respective latent factor vectors for lf_ui
f_gd <- foreach(i = 1:i, .combine ="c") %dopar% {
  pq$P[, uid_val[i]] %*% pq$Q[, mid_val[i]]
}

#prediction composition and get the error for RMSE calculation
#!!be aware the "gen_bias" and "f_gd" are subtracted in prediction composition
#by the reason that both of them were trained from the residuals
#mean + u_b + m_b - rating & mean + u_b + m_b - g_b - rating!!
pred <- m_bias[u_bias[validation[, .(userId, movieId, rating)][
  , ":="(gen_bias = gen_bias, f_gd = f_gd)], on = .(userId)],
  on = .(movieId)][
    , ":="(pred = pred <- g_mean + u_bias + m_bias - gen_bias - f_gd,
           err = pred - rating)]
```

*the real code in calculation is $\bar{r}_g + b_u + b_m - b_{gui} - lf_{ui}$, because we trained them from the residuals of

$$b_{gui} \sim \bar{r}_g + b_u + b_m - r_{ui} \rightarrow r_{ui} \sim \bar{r}_g + b_u + b_m - b_{gui}$$

$$lf_{ui} \sim \bar{r}_g + b_u + b_m - b_{gui} - r_{ui} \rightarrow r_{ui} \sim \bar{r}_g + b_u + b_m - b_{gui} - lf_{ui}$$

.

We now use the prediction error from above calculation to get the Final RMSE:

```
sqrt(mean(pred$err * pred$err))
```

```
## [1] 0.8053019
```

The goal of prediction RMSE:0.8053019 $<$ 0.86490 is achieved.

Conclusion

To summarise what we have done to reach our prediction RMSE goal, the whole process is broken into 4 big parts: data exploration, problem identification, model construction, and code implementation. In the data exploration, we get a picture of the dataset, then we simplified the multi-attributes rating into a user-movie rating matrix, so by inspiration of the "Latent Factor" algorithm of Koren, Bell, and Volinsky (2009), our goal becomes a matrix factorisation estimation problem on a mathematical ground. Around this concept, we further broke down those attributes into user factors and movie factors, and use them to build a model that is versatile enough to predict any user-movie paired rating.

Beside the "Latent Factor" as the core in our algorithm, we employed an "Elastic Net" algorithm with the given movie genres to derive genres vector for each user. Although we can use "Latent Factor" directly on the ratings without exploiting the hidden user genres info, but this extra approach is believed not only can improve the "Latent Factor" learning speed afterward, but also can improve a recommendation system performance with more user attributes data, such as movie search histories, and if this recommendation system is used in such as Amazon Prime Video, then we will have much richer user info connected with

their shopping behaviour, then this user genres info from shopping history can be used for classifying them into different groups, then we could have a group bias, which can reduce computation cost. And for new customers we can provide a better service to meet their broad tastes without throwing darts blindly. We have to bare in mind that, the real world data is much larger and complex, so reduce computation cost can improve data efficiency to companies, as well as customer experiences.

Even though our prediction RMSE is much better than the target, we can not guarantee the same result in the real world, since we are playing in a naive dataset. Especially in the case of newly on board customers, we have no watch history about them, then we are really throwing darts in dark to push the movies she might like. So except for the taste information we argued in above, we can also use the timestamp info that was neglected by us in report, we can push the contemporary popular movies to them to test the taste.

At the very end of this report, we put a table of RMSE path of our model to demonstrate how effectiveness of the "Latent Factor" algorithm in a recommendation system is.

contributors	RMSE
g_mean	1.061202
g_mean + u_bias	0.9779359
g_mean + u_bias + m_bias	0.880098
g_mean + u_bias + m_bias + gen_bias	0.8762853
g_mean + u_bias + m_bias + gen_bias + latent_factor	0.8053019

Refnereces

Bell, Robert, Yehuda Koren, and Chris Volinsky. 2007. "Modeling Relationships at Multiple Scales to Improve Accuracy of Large Recommender Systems." In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '07*. ACM Press. https://doi.org/10.1145/1281192.1281206.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2010. "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software* 33 (1): 1–22. https://www.jstatsoft.org/v33/i01/.

Harper, F. Maxwell, and Joseph A. Konstan. 2016. "The MovieLens Datasets." *ACM Transactions on Interactive Intelligent Systems* 5 (4): 1–19. https://doi.org/10.1145/2827872.

Irizarry, Rafael A. 2022. "Introduction to Data Science." *Rafalab*. https://rafalab.github.io/dsbook/.

Koren, Yehuda, Robert Bell, and Chris Volinsky. 2009. "Matrix Factorization Techniques for Recommender Systems." *Computer* 42 (8): 30–37. https://doi.org/10.1109/mc.2009.263.

Nielsen, Michael A. 2015. *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/index.html.

Strang, Gilbert. 2016. *Introduction to Linear Algebra*. 5th ed. Wellesley, MA: Wellesley-Cambridge Press.

———. 2019. *Linear Algebra and Learning from Data*. Wellesley, MA: Wellesley-Cambridge Press.

Töscher, Andreas, and Michael Jahrer. 2009. "The BigChaos Solution to the Netflix Grand Prize," January.

Zou, Hui, and Trevor Hastie. 2005. "Regularization and Variable Selection via the Elastic Net." *Journal of the Royal Statistical Society. Series B (Statistical Methodology)* 67 (2): 301–20. http://www.jstor.org/stable/3647580.

Appendix

```
###this code chunk represent the function gdtune() is written in C++
##this function returns the sum of squared error for each lambda
###see the manual in below cran link for manual to use C++ coded function in R
###https://cran.r-project.org/web/packages/Rcpp/index.html
```

```
// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export]]
double gdtune(mat P, mat Q,
              Rcpp::NumericVector ytr,
              Rcpp::NumericVector ytst,
              Rcpp::NumericVector Uitr,
              Rcpp::NumericVector Uitst,
              Rcpp::NumericVector Mjtr,
              Rcpp::NumericVector Mjtst,
              double L_rate, double lambda,
              int epochs){
  vec err(ytst.size());
  double se;

  for(int i = 0; i < epochs; i++){
    for(int j = 0; j < ytr.size(); j++){
      int ui = Uitr(j) - 1;
      int mj = Mjtr(j) - 1;
      double err = dot(P.col(ui), Q.col(mj)) - ytr(j);

      vec nabla_P_temp = err * Q.col(mj) + lambda * P.col(ui);
      vec nabla_Q_temp = err * P.col(ui) + lambda * Q.col(mj);

      P.col(ui) -= L_rate * nabla_P_temp;
      Q.col(mj) -= L_rate * nabla_Q_temp;
    }
  }

  for(int k = 0; k < ytst.size(); k++){
    int ui = Uitst(k) - 1;
    int mj = Mjtst(k) - 1;
    err(k) = dot(P.col(ui), Q.col(mj)) - ytst(k);
  }

  se = dot(err, err);

  return se;
}
```

```
###this is the final gd() function code in C++
##this function returns a list contains factor_length by user_num matrix P
##and factor_length by movie_num matrix Q

// [[Rcpp::depends(RcppArmadillo)]]

#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export]]
```

```
Rcpp::List gd(Rcpp::NumericVector U_i,
              Rcpp::NumericVector M_j,
              Rcpp::NumericVector y,
              int u_n, int m_n, int factor_n,
              double L_rate, double lambda, int epochs){

  mat P(factor_n, u_n, fill::randu);
  mat Q(factor_n, m_n, fill::randu);

  for(int i = 0; i < epochs; i++){
    for(int j = 0; j < y.size(); j++){
      int ui = U_i(j) - 1;
      int mj = M_j(j) - 1;
      double err = dot(P.col(ui), Q.col(mj)) - y(j);

      vec nabla_P_temp = err * Q.col(mj) + lambda * P.col(ui);
      vec nabla_Q_temp = err * P.col(ui) + lambda * Q.col(mj);

      P.col(ui) -= L_rate * nabla_P_temp;
      Q.col(mj) -= L_rate * nabla_Q_temp;
    }
  }

  return Rcpp::List::create(Rcpp::Named("P") = P,
                            Rcpp::Named("Q") = Q);
}
```