Movielens_Project_Report

Curios_i

04/05/2021

Contents

1 Introduction

Though recommender systems are not new, the interest in a movie rating recommender system was fueled by Netflix challenge announced in 2006 and finally won by BellKor's Pragmatic Chaos team in 2009. The goal of this project is to create a movie recommendation system using the movielens dataset. The movielens dataset comprises of 10M movie ratings from Netflix data as compared to more than 100M used in Netflix challenge.

To build a recommender system, this report will discuss two solutions. Please note that this is a learning project and not a research project. Although the Solution#2 is more efficient and robust, it is kind of plug and play, that's why Solution#1 is also included which dissects the main underlying theory of a recommender system. In the end, we shall compare both solutions and their limitations. Though both solutions have achieved the target RMSE, we shall still emphasize that the intent of this project and report is learning and sharing, not winning any competition. So, sit back relax and enjoy...

This report and the code is also available from github repository.

2 Data Analysis

2.1 Grabbing the data

The 10M movie rating data is downloaded from this link. Here is the code to convert this data into a data frame and then divide into a edx and validation data set.

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

#tcrossprod() is used from library Matrix, while
#funkSVD() is used from library recommenderlab

library(tidyverse)
library(caret)
library(data.table)
library(Matrix)
library(recommenderlab)
library(knitr)

# 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 <- 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")

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

Next, we divide edx data frame into edx_train and edx_test using the following code so that we can tune and test our module without touching the validation data frame. But, before doing that, we increase the virtual memory to 200,000MB so that it can handle this huge amount of data.

```
memory.limit(size=200000) #sets virtual memory size to 200,000Mb to process large data
set.seed(1, sample.kind="Rounding")
#Partition edx into 80% edx_train and 20% edx_test so
#that we can tune the model without touching the validation data
test_index <- createDataPartition(y = edx$rating, times = 1, p = 0.2, list = FALSE)
edx_train<-edx[-test_index,]
temp<-edx[test_index,]

#make sure that the movieId and userId in edx_train are also in edx_test
edx_test<-temp%>%semi_join(edx_train,by="movieId")%>%semi_join(edx_train,by="userId")
#Add removed records from edx_test back to edx_train
removed<-anti_join(temp,edx_test)
edx_train<-rbind(edx_train,removed)
#Remove temporary variables to free up the memory
rm(removed,temp,test_index)
```

2.2 Analyzing the Data and Deciding the Model

The next step is to analyze the data and decide which model to use. However, before doing that, we need to decide our loss function. We shall use root-mean-square-error as our loss function. So, let's define our loss function in the following code:

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

There are two main challenges with data in this project

- The size of data
- The nature of the model

The first temptation is to use a machine learning model on this data, like linear regression, logistic regression, LDA, QDA etc. The problem with this approach is that userId and movieId are just arbitrary variables. In fact, userId was used by Netflix to protect the privacy of users. There is no cause and effect relationship between userId vs rating or movieId vs rating. In fact, if I take userId from 1 to 1000 and 1001 to 2000 and swap them, it should not affect the prediction model at all. To prove this point, let's see the correlation.

```
cor(edx$userId,edx$rating)
```

```
## [1] 0.002313643
```

```
cor(edx$movieId,edx$rating)
```

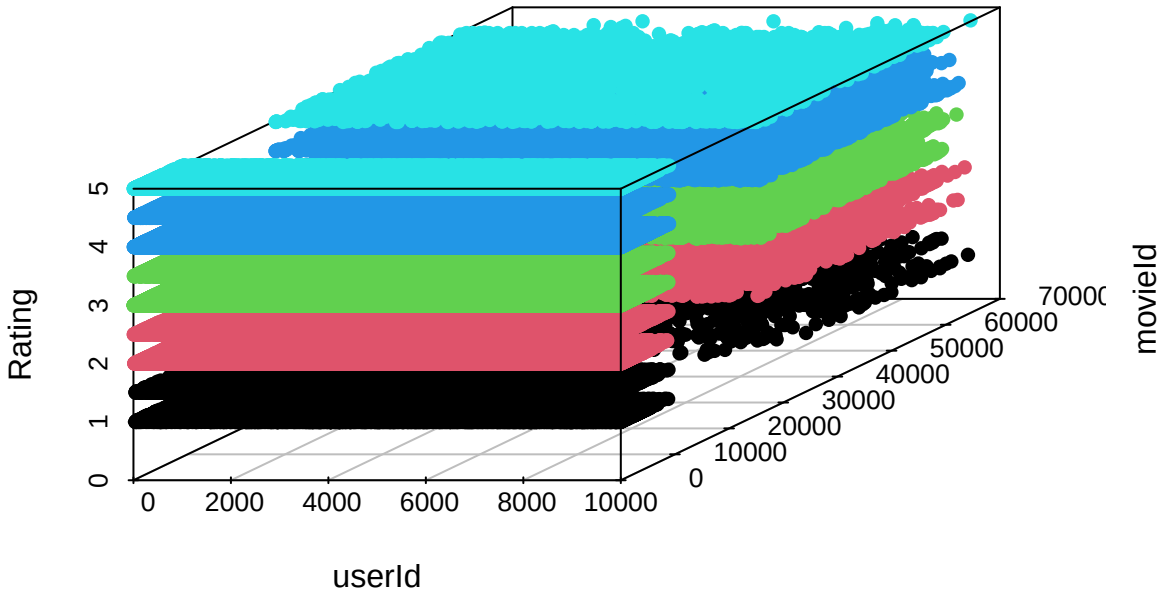
```
## [1] -0.006535696
```

Another issue is the size of the data. For such a large dataset if you try to use any model using normal r package (e.g. caret), it will crash your computer, or you need a very large computing resource, which is out of reach of most of the students. One solution to this problem is to use stochastic gradient descent algorithm.

In fact, we created a liner regression model of this dataset using "SGD" package, but the RMSE obtained was greater than 1.5, for the reasons explained above, so we are not even including that in this report.

Now, to have a visual sense of dataset, let's first dissect the edx_train into a smaller dataset and then make a 3D plot.

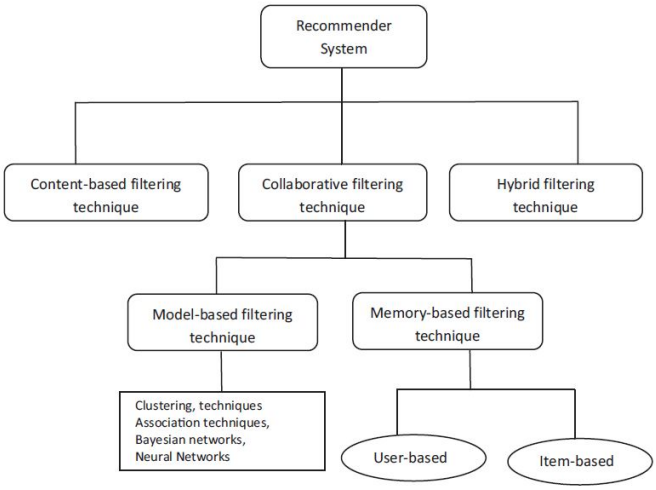
```
train_small<-edx_train%>%filter(userId<10000)
if(!require(scatterplot3d)) install.packages("scatterplot3d")
library(scatterplot3d)
scatterplot3d(x=train_small$userId,z=train_small$rating,y=train_small$movieId,
color = train_small$rating,xlab="userId",ylab="movieId",zlab="Rating",pch=16)
```



It is clear from the plot that applying a normal machine learning technique as described above will not be very successful on this data.

2.3 Recommender System Modeling Techniques

Following chart shows well known recommender system modeling techniques.



2.3.1 Conent Based Filtering

Content based technique emphasizes more on the analysis of the attributes of items in order to generate predictions. When items, like web pages, publications and news are to be recommended, content-based filtering technique is the most successful. Content based filtering requires a lot of meta data about items. Since in the case of movielens data the only meta data somewhat useful is the movie genera, we are not considering content-based filtering in this project.

[3] chapter 3 discusses an example of movie reviews either positive or negative based on 50,000 reviews from IMDB data set using deep learning. The same model can be extended to predict movie ratings from 1-5 based on reviews. As mentioned above, we don't have enough data to apply this filtering in this project.

2.3.2 Collborative Filtering

Collaborative filtering is a prediction technique for content that cannot easily and adequately be described by metadata, such as movies and music. Collaborative filtering technique works by building a database (user-item matrix) of preferences for items by users. It then matches users with relevant interest and preferences by calculating similarities between their profiles to make recommendations. Such users build a group called neighborhood. An user gets recommendations to those items that he/she has not rated before but that were already positively rated by users in his neighborhood. The technique of collaborative filtering can be divided into two categories: memory-based and model-based.

2.3.2.1 Memory Based Collaborative Filtering Memory-based CF can be achieved in two ways through user-based and item-based techniques. User based collaborative filtering technique calculates similarity between users by comparing their ratings on the same item and it then computes the predicted rating for an item by the active user as a weighted average of the ratings of the item by users similar to the active user where weights are the similarities of these users with the target item. Item-based filtering techniques compute predictions using the similarity between items and not the similarity between users.

In our Solution#1 we have used memory based collaborating filtering along with regularization on top of overall average rating.

2.3.2.2 Model Based Techniques Model based techniques employ previous ratings to learn a model in order to improve the performance of collaborative filtering. These techniques can quickly recommend a set of items for the fact that they use pre-computed model. The most popular model based technique in recommended system is Singular Value Decomposition (SVD). In both our Solution#1 and Solution#2 matrix factorization by SVD is used as a model based CF filtering.

3 Solution#1

Solution#1 is based on [1] and [2]. The model for predicted ratings have four components, calculated from edx_train

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

1. μ - mean of ratings
2. b_i - average rating of a movie i, with L2 regularization
3. b_u - average rating given by a user I, with L2 regularization
4. $r_{u,i}$ - residual ratings calculated from above.

5. $\xi_{u,i}$ - All rest of the variations not covered by the model.

We can write

$$pred = r_{u,i} + \xi_{u,i} = Y_{u,i} - \mu + b_i + b_u$$

In our computation we shall use variable `pred` to represent the left hand side of the equation and will use matrix factorization to estimate that. `pred` represents the fact that group of users have similar rating patterns for group of movies, i.e. it is the interaction between users and movies. We shall estimate it by using singular value decomposition.L

The first step is to calculate the first component, i.e. μ - mean of ratings

```
mu<-mean(edx_train$rating)
```

Next step is to calculate values of b_i and b_u using L2 regularization. We shall use equations given in [1] and [2].

$$b_i(\lambda) = \frac{1}{\lambda + n_i} \sum_{a=1}^{n_i} (Y_a - \mu)$$

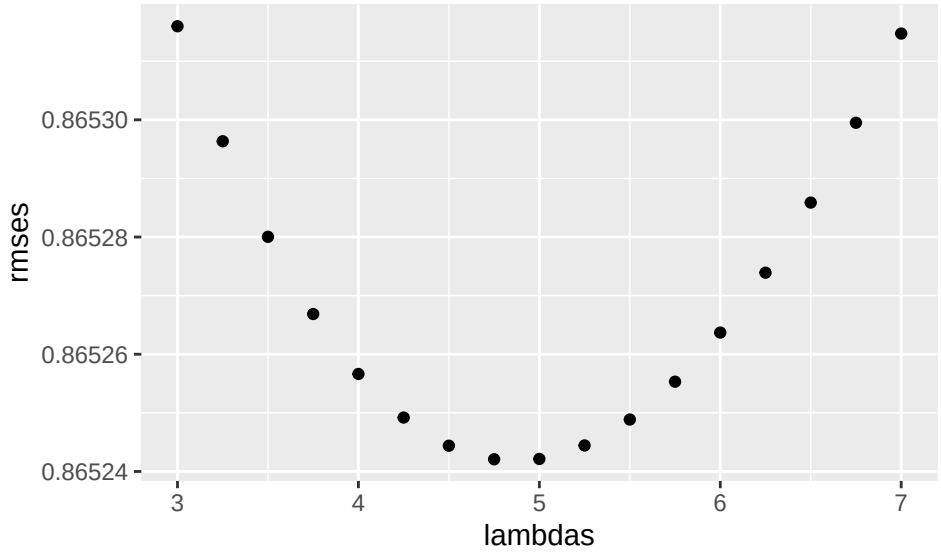
$$b_u(\lambda) = \frac{1}{\lambda + n_u} \sum_{a=1}^{n_u} (Y_a - \mu - b_i)$$

Since b_i and b_u are function of λ, we first define an range of lambdas and then find rmses for every value in the lambdas vector.

```
lambdas <- seq(3, 7, 0.25)
# calculates rmses for the above values of lambdas to tune the model
rmses<-sapply(lambdas,function(l){
  b_i<-edx_train%>%group_by(movieId)%>%summarise(b_i=sum(rating-mu)/(n()+l))
  b_u<-edx_train%>%left_join(b_i,by="movieId")%>%
    group_by(userId)%>%
    summarise(b_u=sum(rating-b_i-mu)/(n()+l))
  predicted_ratings_test<-edx_test%>%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,edx_test$rating))})
```

Now, plot of rmses vs lambdas

```
library(ggplot2)
qplot(lambdas,rmses)
```



We find the the minimum rmse at this point is

```
min(rmses)
```

```
## [1] 0.8652421
```

at the value of lambda

```
lambda<-lambdas[which.min(rmses)]
lambda
```

```
## [1] 4.75
```

Now we shall calculate b_i and b_u for edx_train. Those b_i and b_u are then used to calculate predicted ratings for edx_train and edx_test. Predicted ratings for edx_test is later used along with funk svd predictions to tune the model.

Predicted ratings for edx_train are used to calculate the residual ratings. Funk svd matrix factorization is applied on those residual edx_train ratings to build the model.

Here is the code:

```
# calculates b_i and b_u for edx_train. Since all users and movies in edx_test and
#validation are present in edx_train, these values of b_i and b_u are used later  to
#calculate the predicted ratings for those datasets
b_i<-edx_train%>%group_by(movieId)%>%
  summarise(b_i=sum(rating-mu)/(n()+lambda))
b_u<-edx_train%>%left_join(b_i,by="movieId")%>%
  group_by(userId)%>%
  summarise(b_u=sum(rating-b_i-mu)/(n()+lambda))
# calculates predicted ratings for the edx_test based on b_i and b_u
predicted_ratings_test<-edx_test%>%
  left_join(b_i,by="movieId")%>%
  left_join(b_u,by="userId")%>%
  mutate(pred=mu+b_i+b_u)%>%
```

```
  pull(pred)
#calculates predicted ratings for edx_train, so that we can calculate
#residuals for further matrix factorization
predicted_ratings_train<-edx_train%>%
  left_join(b_i,by="movieId")%>%
  left_join(b_u,by="userId")%>%
  mutate(pred=mu+b_i+b_u)%>%
  pull(pred)
#calculated residual ratings after subtracting the predicted ratings
#from edx_train ratings
edx_train<-edx_train%>%
  mutate(predicted_ratings_train=predicted_ratings_train,
         resid=rating-predicted_ratings_train)
```

Now, we shall convert `resid` into a matrix, where each row will represent a user and each column will represent a movie.

```
y<-edx_train %>% select(userId,movieId,resid) %>% spread(movieId,resid) %>% as.matrix()
rownames(y)<- y[,1]
y<-y[,-1]
```

3.1 Funck SVD

A really smart realization made by the guys who entered the Netflix's competition (notably Simon Funk) was that the users' ratings weren't just random guesses. Raters probably follow some logic where they weight the things they like in a movie (a specific actress or a genre) against things they don't like (long duration or bad jokes) and then come up with a score.

That process can be represented by a linear formula of the following kind:

$$R_{u,i} = UV^T$$

Here,

U is a matrix of rows equal to number of users (or number of rows of $r_{u,i}$) and k columns

V is a matrix of rows equal to number of items/movies (or number of columns of $r_{u,i}$) and k columns

k is number of features we want to extract

$R_{u,i}$ is a user rating matrix, with u users and i ratings.

In a recommender system database, every user doesn't rate every movie, so $R_{u,i}$ is a sparse matrix, where most of the elements are unknown.

U and V can be found in such a way that the square error difference between their cross product and known rating in the user-item matrix is minimum.

If we consider u as a row vector of U matrix and v as a row vector of V matrix, we can write

$$\text{expected rating} = \hat{r_{u,i}} = uv^T$$

Our goal is to minimize the following for each known rating.

$$min(u,v)\sum(r_{u,i} - uv^T)$$

For our model to be able to generalize well and not over-fit the training set, we introduce a penalty term to our minimization equation. This is represented by a regularization factor γ multiplied by the square sum of the magnitudes of user and item vectors, in case of L2 regularization (ridge regression)

$$min(u,v)\sum(r_{u,i} - uv^T) + \gamma(||u||^2 + ||v||^2)$$

While in case of L1 regularization,

$$min(u,v)\sum(r_{u,i} - uv^T) + \gamma(||u|| + ||v||)$$

funkSVD is a function from `recommenderlab` library. In the funkSVD function, we use L1 regularization, where all elements of U and V matrices are set to zero initially. Following is the usage of the function with default values of arguments.

funkSVD(x, k = 10, gamma = 0.015, lambda = 0.001, min_improvement = 1e-06, min_epochs = 50, max_epochs = 200, verbose = FALSE)

Arguments

Parameter	Description
x	a matrix, potentially containing NAs.
y	number of features (i.e. rank of approximation)
gamma	regularization term
lambda	learning rate
min_improvement	required minimum improvement per iteration
min_epochs	minimum number of iterations per feature
max_epochs	maximum number of iterations per feature
verbos	show progress

3.2 Tuning funkSVD model

Since running funkSVD requires a lot of computational power and memory, we shall only use k = 3 features since that will give us less than the target RMSE. Using more features may definitely improve the RMSE.

Besides that, there are other hyperparameters, like epochs, gamma, lambda and min_improvement. We shall keep the learning rate lambda default, however, for regularization, we shall tune our model for three values of gamma. We define a corresponding vector f_rmse and initialize it to store the RMSE outcomes of three models.

```
# gammas are L1 regularization terms in svd gradient descent algorithm
# we shall calculate rmses for each of these gammas and will determine which one gives optimal tuning
gammas<-c(0.015,0.025,0.035)
# f_rmses stores rmses calculated for respective values of gammas
f_rmses<-c(1,1,1)
```

3.2.1 For gamma = 0.015

As mentioned above, funkSVD function returns us two `fsvd$U` and `fsvd$V` matrices. Cross product of these two matrices gives us our prediction matrix for residuals. We add those predictions to predicted_ratings_test. Remember that

$$predicted\_ratings\_test = \mu + b_i + b_u$$

```
#############################################################################
# runs the Simon Funk's gradient descent algorithm to factorize matrix of residuals
# this will take several hours
fsvd<-funkSVD(y, k=3, gamma=gammas[1], lambda=0.001, verbose=TRUE)
# y_hat is prediction matrix from SVD
y_hat_0.015<-tcrossprod(fsvd$U,fsvd$V)
# Assigns row and column names to prediction matrix so that predicted rating can be pulled from the mat
rownames(y_hat_0.015)<-rownames(y)
colnames(y_hat_0.015)<-colnames(y)
# creates a placeholder for pred = predicted ratings for residuals from SVD
pred<-rep(0,length(edx_test$userId))
# fill vector pred from the prediction matrix for respective userId and movieId
for(i in 1:length(edx_test$userId)){
  pred[i]<-y_hat_0.015[as.character(edx_test$userId[i]),as.character(edx_test$movieId[i])]}
# calculated predicted ratings form edx_test for that particular gamma
predicted_ratings_0.015<-predicted_ratings_test+pred
# calculates rmse for that particular gamma
f_rmses[1]<-RMSE(predicted_ratings_0.015,edx_test$rating)
#[1] 0.8260231
```

The RMSE calculated for gamma=0.015 is

```
f_rmses[1]
```

```
## [1] 0.8260231
```

3.2.2 For gamma = 0.025

As mentioned above, funkSVD function returns us two `fsvd$U` and `fsvd$V` matrices. Cross product of these two matrices gives us our prediction matrix for residuals. We add those predictions to predicted_ratings_test. Remember that

$$predicted\_ratings\_test = \mu + b_i + b_u$$

```
#############################################################################
# runs the Simon Funk's gradient descent algorithm to factorize matrix of residuals
fsvd<-funkSVD(y, k=3, gamma=gammas[2], lambda=0.001, verbose=TRUE)
# y_hat is prediction matrix from SVD
y_hat_0.025<-tcrossprod(fsvd$U,fsvd$V)
# Assigns row and column names to prediction matrix so that predicted rating can be pulled from the mat
rownames(y_hat_0.025)<-rownames(y)
colnames(y_hat_0.025)<-colnames(y)
pred<-rep(0,length(edx_test$userId))
# fill vector pred from the prediction matrix for respective userId and movieId
for(i in 1:length(edx_test$userId)){
  pred[i]<-y_hat_0.025[as.character(edx_test$userId[i]),as.character(edx_test$movieId[i])]}
# calculated predicted ratings form edx_test for that particular gamma
predicted_ratings_0.025<-predicted_ratings_test+pred
# calculates rmse for that particular gamma
f_rmses[2]<- RMSE(predicted_ratings_0.025,edx_test$rating)
#0.8257668
```

The RMSE calculated for gamma=0.025 is

```
f_rmses[2]
```

```
## [1] 0.8257668
```

3.2.3 For gamma = 0.035

As mentioned above, funkSVD function returns us two `fsvd$U` and `fsvd$V` matrices. Cross product of these two matrices gives us our prediction matrix for residuals. We add those predictions to predicted_ratings_test. Remember that

$$predicted\_ratings\_test = \mu + b_i + b_u$$

```
##############################################################################
# runs the Simon Funk's gradient descent algorithm to factorize matrix of residuals
fsvd<-funkSVD(y, k=3, gamma=gammas[3], lambda=0.001, verbose=TRUE)
y_hat_0.035<-tcrossprod(fsvd$U,fsvd$V)
rownames(y_hat_0.035)<-rownames(y)
colnames(y_hat_0.035)<-colnames(y)
pred<-rep(0,length(edx_test$userId))
for(i in 1:length(edx_test$userId)){
  pred[i]<-y_hat_0.035[as.character(edx_test$userId[i]),as.character(edx_test$movieId[i])]}
predicted_ratings_0.035<-predicted_ratings_test+pred
f_rmses[3]<- RMSE(predicted_ratings_0.035,edx_test$rating)
#[1] 0.8258410
```

The RMSE calculated for gamma=0.035 is

```
f_rmses[3]
```

```
## [1] 0.825841
```

3.3 Calculating Final RMSE for Validation

After calculating RMSEs for all three values of gammas, we select the gamma which gave us lowest RMSE.

```
#calculates the optimal gamma from rmses results
gamma_min<-gammas[which.min(f_rmses)]
gamma_min
```

```
## [1] 0.025
```

```
# assigns the final prediction matrix from the min rmse gamma
y_hat_final<-y_hat_0.025
```

Next, we calculate predicted ratings for validation set using μ, b_i and b_u from edx_train.

```
# predicted ratings for validation set is first calculated from mu, b_i and b_u from edx_train
predicted_ratings_valid<-
  validation%>%left_join(b_i,by="movieId")%>%
  left_join(b_u,by="userId")%>%
  mutate(pred=mu+b_i+b_u)%>%
  pull(pred)
```

From here, we calculate first RMSE before applying SVD residual calculations

```
# This finds rmse before applying SVD to residual
RMSE(validation$rating,predicted_ratings_valid)
```

```
## [1] 0.8657012
```

Following code now calculates validation RMSE

```
# A residual vector placeholder for each user in the validation
pred<-rep(0,length(validation$userId))
# pulls the predicted value of residual from the prediction matrix
for(i in 1:length(validation$userId)){
  pred[i]<-y_hat_final[as.character(validation$userId[i]),as.character(validation$movieId[i])]}
#calculates the final predicted ratings for validation set
predicted_ratings_valid<-predicted_ratings_valid+pred
# calculates final rmse
RMSE1<-RMSE(predicted_ratings_valid,validation$rating)
RMSE1
```

```
## [1] 0.8252377
```

3.4 Results

The results of solution#1 in a table form:

```
library(knitr)
result1<-data.frame(method="Solution#1",RMSE=RMSE1)
kable(result1)
```

method	RMSE
Solution#1	0.8252377

4 Solution #2

We used gradient descent algorithm in fundSVD function for matrix factorization. The problem with this solution is that it requires a lot of computational power and cannot be be parallelized. In [2], authors presented a fast parallel SG method, FPSG, for shared memory systems. By dramatically reducing the cache-miss rate and carefully addressing the load balance of threads, FPSG is more efficient than state-of-the-art parallel algorithms for matrix factorization. R's `recosystem` package is based on this FPSG

algorithm, which is much faster than funkSVD of `recommenderlab` and requires less memory.

FPSG uses the standard thread class in C++ implemented by pthread to do the parallelization. The description of C++ code is beyond the scope of this study.

In the first step, we shall load the `recosystem package` and will initialize a `reco` object.

```
install.packages("recosystem")
```

```
library(recosystem)
r<-Reco()
```

4.1 Model Tuning

Next, we tune the model for 5 features or 10 features #with 3 different values of learning rates (0.05,0.1,0.15) for 20 iterations and 5 fold cross validation. Please note that since the tune function is itself using cross validation, we are not breaking edx dataset into train and test.

```
opts<-r$tune(data_memory(edx$userId,edx$movieId,
    rating = edx$rating,index1 = TRUE),
    opts = list(dim=c(5,10),lrate=c(0.05,0.1,0.15),niter=20,nfold=5,verbose=FALSE))
```

Opts$min gives us the optimized tuning parameters.

```
opts$min
```

```
## $dim
## [1] 10
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0.1
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.01
##
## $lrate
## [1] 0.15
##
## $loss_fun
## [1] 0.8098363
```

Tuning Parameter	Value
Features	10
Learning rate	0.15

4.2 Model Trainig

Following will train edx dataset based on above tuning parameters and 100 iterations.

```
r$train(data_memory(edx$userId,edx$movieId,rating = edx$rating,index1 = TRUE),
        opts = c(opts$min,nthread=1,niter=100))
```

```
## iter      tr_rmse          obj
##    0       0.9531   1.0512e+07
##    1       0.8625   9.0436e+06
##    2       0.8346   8.6027e+06
##    3       0.8229   8.4195e+06
##    4       0.8149   8.3029e+06
##    5       0.8087   8.2156e+06
##    6       0.8034   8.1463e+06
##    7       0.7987   8.0886e+06
##    8       0.7943   8.0350e+06
##    9       0.7900   7.9870e+06
##   10       0.7860   7.9429e+06
##   11       0.7824   7.9034e+06
##   12       0.7791   7.8696e+06
##   13       0.7760   7.8363e+06
##   14       0.7734   7.8099e+06
##   15       0.7709   7.7848e+06
##   16       0.7687   7.7618e+06
##   17       0.7669   7.7442e+06
##   18       0.7650   7.7257e+06
##   19       0.7634   7.7099e+06
##   20       0.7620   7.6948e+06
##   21       0.7605   7.6794e+06
##   22       0.7594   7.6702e+06
##   23       0.7583   7.6589e+06
##   24       0.7573   7.6486e+06
##   25       0.7564   7.6406e+06
##   26       0.7555   7.6316e+06
##   27       0.7546   7.6227e+06
##   28       0.7539   7.6157e+06
##   29       0.7531   7.6077e+06
##   30       0.7524   7.6018e+06
##   31       0.7518   7.5957e+06
##   32       0.7511   7.5884e+06
##   33       0.7506   7.5835e+06
##   34       0.7500   7.5784e+06
##   35       0.7496   7.5752e+06
##   36       0.7491   7.5699e+06
##   37       0.7486   7.5645e+06
##   38       0.7481   7.5601e+06
##   39       0.7478   7.5569e+06
##   40       0.7473   7.5526e+06
##   41       0.7470   7.5490e+06
##   42       0.7466   7.5458e+06
##   43       0.7463   7.5432e+06
##   44       0.7459   7.5387e+06
##   45       0.7456   7.5355e+06
```

```
##   46       0.7453   7.5335e+06
##   47       0.7450   7.5298e+06
##   48       0.7447   7.5273e+06
##   49       0.7445   7.5247e+06
##   50       0.7442   7.5217e+06
##   51       0.7439   7.5190e+06
##   52       0.7437   7.5176e+06
##   53       0.7435   7.5147e+06
##   54       0.7432   7.5128e+06
##   55       0.7431   7.5119e+06
##   56       0.7428   7.5083e+06
##   57       0.7425   7.5057e+06
##   58       0.7424   7.5038e+06
##   59       0.7422   7.5027e+06
##   60       0.7420   7.5013e+06
##   61       0.7418   7.4989e+06
##   62       0.7416   7.4964e+06
##   63       0.7415   7.4953e+06
##   64       0.7413   7.4936e+06
##   65       0.7412   7.4925e+06
##   66       0.7410   7.4911e+06
##   67       0.7408   7.4890e+06
##   68       0.7407   7.4876e+06
##   69       0.7405   7.4867e+06
##   70       0.7404   7.4852e+06
##   71       0.7403   7.4837e+06
##   72       0.7401   7.4820e+06
##   73       0.7400   7.4806e+06
##   74       0.7398   7.4796e+06
##   75       0.7397   7.4788e+06
##   76       0.7396   7.4774e+06
##   77       0.7395   7.4767e+06
##   78       0.7394   7.4754e+06
##   79       0.7393   7.4743e+06
##   80       0.7391   7.4726e+06
##   81       0.7391   7.4719e+06
##   82       0.7389   7.4707e+06
##   83       0.7388   7.4690e+06
##   84       0.7388   7.4693e+06
##   85       0.7386   7.4681e+06
##   86       0.7385   7.4660e+06
##   87       0.7384   7.4660e+06
##   88       0.7383   7.4650e+06
##   89       0.7383   7.4648e+06
##   90       0.7382   7.4630e+06
##   91       0.7381   7.4624e+06
##   92       0.7380   7.4619e+06
##   93       0.7379   7.4605e+06
##   94       0.7378   7.4603e+06
##   95       0.7377   7.4590e+06
##   96       0.7377   7.4583e+06
##   97       0.7376   7.4575e+06
##   98       0.7375   7.4564e+06
##   99       0.7375   7.4564e+06
```

4.3 Results

Now we use the predict function to get predicted ratings for the validation set.

```
predicted_ratings<-r$predict(data_memory(validation$userId,
        validation$movieId,rating = NULL,index1 = TRUE),out_memory())
```

Finally the RMSE function gives the root mean square error.

```
RMSE2<-RMSE(predicted_ratings,validation$rating)
RMSE2
```

```
## [1] 0.7934708
```

4.4 Important Note about Solution#2

Please note that since FPSG algorithm is using parallel processing, RMSE results may not be 100% reproducible. That means, if someone else runs this code, it might give a slightly different RMSE.

5 Conclusion

We see that both of our solutions have achieved good RMSEs, less than the project requirement. Since FPSG method is much faster, we used 10 features in the matrix factorization and achieved a better RMSE. In solution 1, we used only 3 features for matrix factorization, however, we believe that we could have achieved much better RMSE if we have used more features. Following is the comparison of RMSEs achieved by both models.

```
result2<-rbind(result1,data.frame(method="Solution#2",RMSE=RMSE2))
kable(result2)
```

method	RMSE
Solution#1	0.8252377
Solution#2	0.7934708

The original Netflix competition was launched in 2006 and winners were announced in 2009. Since then this topic has attracted a lot of attention and significant improvements have been made in recommender system modeling. The main issue, as discussed in our solution#1 is the heavy computational cost to handle linear algebraic operations on a huge dataset. However, we see that the computational cost was significantly reduced in solution#2 by using parallel computing.

5.1 Who won the competition?

The obvious answer comes into mind is Bellkor. Following is the trend of Netflix's revenue growth over the years.

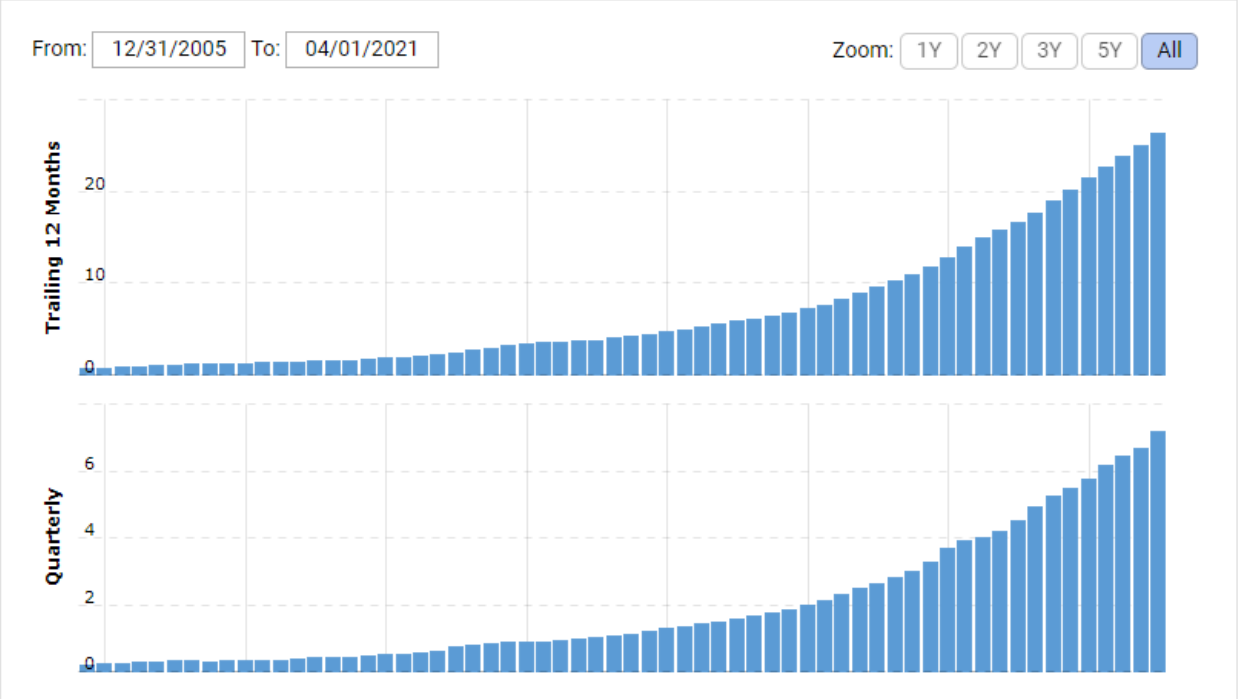


Figure 1: Netflix Revenue Growth

This shows that the real beneficiary of all these kind of initiatives was actually Netflix itself. The $1M price over three years of hard work divided among 7 persons doesn't look like an attractive story, compared to the the benefits Netflix reaped out of it.

The practical challenge for any such recommender system is that the competition was done on a static data, whereas the challenge in the real world recommender system is to deal with data changing all the time, i.e. always new users adding up and new movies coming in. We couldn't find how Netflix is using recommender system on real time data and at what frequency they are retraining their models, though we have some insight from LinkedIn offline batch computing and online serving story.

6 Bibliography

[1] Yehuda Koren. The BellKor Solution to the Netflix Grand Prize

[2] Rafael A. Irizarry. Introduction to Data Science

[3] François Chollet and Joseph J. Allaire. Deep Learning with R

[4] WEI-SHENG CHIN, YONG ZHUANG, YU-CHIN JUAN, CHIH-JEN LIN. A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems