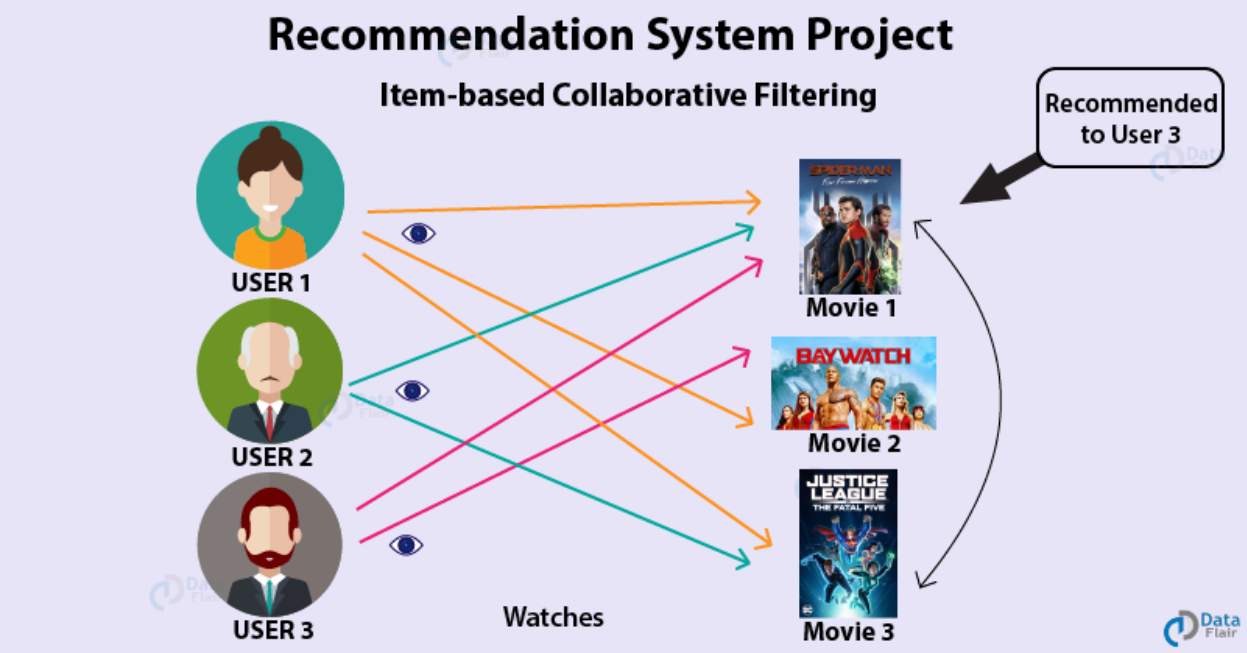
Applied Data Science - Algorithm Implementation and Evaluation

Contents

Introduction 2
Section 1: Model 1 Steps and Output 2
Step 1 - Loading the libraries and data, and splitting the test and train dataset 2
Step 2 - Matrix Factorization 3
Step 3 - Postprocessing with KNN 9
Section 2: Model 2 Steps and Output 10
Step 1 - Load Data and Train-test Split 10
Step 2 - Matrix Factorization 11
Step 3 - Post-processing with KNN 13
Step 4 - Evaluation 15
Section 3 - Evaluation and Conclusion 16

Team Members:
- Chairuna, Marsya (mc4813@columbia.edu)
- Duong, Tram (ttd2111@columbia.edu)
- Gong, Saier (sg3772@columbia.edu)
- Lin, Hongshan (hl3353@columbia.edu)
- Wang, Mengchen (mw3371@columbia.edu)



Introduction

In this project, Group 2 implement matrix factorization by focusing on alternating least square algorithms and KNN as post-processing, in which we try to evaluate how regularization will impact the prediction results. We will use RMSE results for both train and test dataset as the basis of our evaluation. We will compare the performance between the following models:

- *Model 1:* Alternating Least Squares (ALS) with Temporal Dynamics (TD) Regularization and K-nearest Neighbors (KNN) Post-processing
- *Model 2:* Alternating Least Squares (ALS) with K-nearest Neighbors (KNN) Post-processing

The objective of this project is to produce a prediction of users' movies preference based on the ratings given respective users.

The structure of this report is as follows:

- *Section 1:* Model 1 Steps and Output
- *Section 2:* Model 2 Steps and Output
- *Section 3:* Evaluation and Conclusion

Section 1: Model 1 Steps and Output

Step 1 - Loading the libraries and data, and splitting the test and train dataset

Libraries

```
# Loading libraries
library(plyr)
library(dplyr)
library(tidyr)
library(ggplot2)
library(lubridate)
library(SnowballC)
```

Data

```
data <- read.csv("../data/ml-latest-small/ratings.csv") %>% mutate(Date=as.Date(as.POSIXct(timestamp,or
source(file = "../lib/data_split(need modify).R")
```

Split the test and train dataset

```
# set.seed(0)
# test_idx <- sample(1:nrow(data), round(nrow(data)/5, 0))
# train_idx <- setdiff(1:nrow(data), test_idx)
set.seed(0)
split<-train_test_split(data = data,train_ratio = 0.8)
data_train <- split$train
data_test <- split$test
```

Step 2 - Matrix Factorization

Firstly, we will define a function to calculate RMSE.

```
RMSE<-function(rating,est_rating){
  a<-sqrt(mean((rating-est_rating)^2))
  return(a)
}
```

Step 2.1 - Alternative Least Square Algorithm with Temporal Dynamics Regulation Terms In this step, we give initial values to parameter b_u, b_i and α_u, and user feature matrix p and movie feature matrix q.

The objective function can be written as follows:

$min \sum (r_{ij} - (q_i^T p_u + \mu + b_i + b_u + \alpha_u dev_u(t))) + \sum \lambda(||q_i||^2 + ||p_u||^2 + b_i^2 + b_u^2 + \alpha_u^2)$

We then update these parameters ad matrix in each iteration and get a convergent result in RMSE.

```
##tu is the data frame of the mean date of rating of each user in 'data' dataset.
als.td<-function(f=10,lambda=0.1,max.iter=5,data,data_train,data_test){
  data<-data %>%
    arrange(userId,movieId)
  tu<-data %>%
  group_by(userId) %>%
  summarise(meanDate=round_date(mean(Date),'day'))

beta<-0.4
#f <-10
#max.iter <-3
#lambda <-0.5

train<-data_train %>%
  arrange(userId,movieId) %>%
  mutate(diff=as.numeric(Date-tu$meanDate[userId])) %>%
  mutate(dev=sign(diff)*(as.numeric( abs(diff)^beta)) )

mu<-mean(train$rating)

#train1<-train %>%
#  mutate(rating=rating-mu)

test<-data_test %>%
  arrange(userId,movieId) %>%
  mutate(diff=as.numeric(Date-tu$meanDate[userId])) %>%
  mutate(dev=sign(diff)*(as.numeric(abs(diff)))^beta)

U<-length(unique(data$userId))
I<-length(unique(data$movieId))

#initialize q
guodu<-data %>%
  group_by(movieId) %>%
  summarise(row_1=mean(rating-mu))
```

```r
#q<- rbind(guodu$row_1,matrix(data=runif((f-1)*I,min = -1,max = 1),nrow = f-1,ncol = I))
set.seed(0)
q<-matrix(data=runif(f*I,-1,1),nrow = f,ncol = I)
colnames(q)<-as.character(sort(unique(data$movieId)))#movie feature matrix

#initialize bu
bu<-rep(0,U)
names(bu)<-as.character(sort(unique(data$userId)))

#initialize alpha.u
alpha.u<-rep(0,U)
names(alpha.u)<-as.character(sort(unique(data$userId)))

#initialize bi
set.seed(0)
bi<-runif(I,-0.5,0.5)
names(bi)<-as.character(sort(unique(data$movieId)))

#set.seed(0)
p<-matrix(nrow = f,ncol = U)
colnames(p)<-as.character(sort(unique(train$userId)))#user feature matrix

train.rmse<-c()
test.rmse<-c()

for (l in 1:max.iter) {

       ####update p
  for (u in 1:U) {
    #the movie user u rated
    I.u<-train %>%
      filter(userId==u)

    q.I.u<-q[,as.character(I.u$movieId)] #the movie feature u has rated
    R.u<-I.u$rating

    A<-q.I.u %*% t(q.I.u)+lambda * nrow(I.u) *diag(f)
    V<-q.I.u %*% (R.u-bu[as.character(I.u$userId)]-bi[as.character(I.u$movieId)]-mu-alpha.u[as.characte

    p[,u]<-solve(A,tol = 1e-50) %*% V

  }

    ###update bi
 # for (m in 1:I) {
#    U.m<-train %>%
#      filter(movieId==as.numeric(names(bi)[m]))
#
#
#   pred.R.m<-c()
#   for (j in 1:nrow(U.m)) {
#
#     pred.R.m[j]<-as.numeric(t(q[,m]) %*% p[,U.m$userId[j] ]    )
```

```
#
#
#   }
#
#   ###new bi
#   bi[m]<-(sum(U.m$rating)-sum(pred.R.m)-sum(bu)-nrow(U.m)*mu-sum(alpha.u[U.m$userId]*U.m$dev)  )/(nro
#
#
#
#  }

   ###update alpha.u

  for (u in 1:U) {
    I.u<-train %>%
      filter(userId==u)


    pred.R.u<-c()
    for (i in 1:nrow(I.u)) {
      pred.R.u[i]<- as.numeric(t(p[,u]) %*% q[,as.character(I.u$movieId[i])]) #+mu+ bu[as.character(u)]
        #bi[as.character(I.u$movieId[i])]+ alpha.u[as.character(u)] * I.u$dev[i]

    }

    ##new alpha.u
    alpha.u[u]<-(sum(I.u$rating*I.u$dev)-sum(bi[as.character(I.u$movieId[i])]*I.u$dev)-sum(mu*I.u$dev)
                           - sum(pred.R.u*I.u$dev))/(sum(I.u$dev^2)+lambda)

  }

  ###update bu
  for (u in 1:U) {
    I.u<-train %>%
      filter(userId==u)

    pred.R.u<-c()
    #calculate predict rating of user u
    for (i in 1:nrow(I.u)) {
      pred.R.u[i]<- as.numeric(t(p[,u]) %*% q[,as.character(I.u$movieId[i])])
        }

    ##new bu
    bu[u]<-(sum(I.u$rating) - sum(pred.R.u)-sum(bi)-nrow(I.u)*mu-sum(alpha.u[u]*I.u$dev))/(nrow(I.u)+la
    }

  ###update q movie feature

  for (m in 1:I) {
    U.m<-train %>%
      filter(movieId==as.numeric(names(bi)[m]))
```

```
    p.U.m<-p[,U.m$userId] #the user feature who has rated movie m
    R.m<-U.m$rating

    A<-p.U.m %*% t(p.U.m)+lambda * nrow(U.m) *diag(f)
    V<-p.U.m %*% as.matrix(R.m-bu[U.m$userId]-bi[as.character(U.m$movieId)]-mu-alpha.u[U.m$userId]*U.m$d

    q[,m]<-solve(A,tol = 1e-50) %*% V
  }

  R_hat<-as.data.frame(t(p) %*% q)

  R_hat<-R_hat %>%
    mutate(userId=as.numeric(names(bu))) %>%
    pivot_longer(cols = names(bi)[1]:names(bi)[length(bi)],
                 names_to = "movieId",
                 values_to = "est2") %>%
    mutate(movieId=as.numeric(movieId))

  train.with.est<-train %>%
    mutate(est1=mu+bi[as.character(movieId)]+bu[userId]+alpha.u[userId]*dev) %>%
    left_join(R_hat) %>%
    mutate(est_rating=est1+est2)

  train.result<-RMSE(train$rating,train.with.est$est_rating)
  train.rmse[l]<-train.result

  test.with.est<-test %>%
    mutate(est1=mu+bi[as.character(movieId)]+bu[as.character(userId)]+alpha.u[as.character(userId)]*dev)
    left_join(R_hat) %>%
    mutate(est_rating=est1+est2) %>%
    filter(!is.na(est_rating))

  test.result<-RMSE(test.with.est$rating,test.with.est$est_rating)
  test.rmse[l]<-test.result

}

result.tibble<-tibble(train_rmse=train.rmse,
                      test_rmse=test.rmse)

return(r=list(p=p,q=q,mu=mu,bu=bu,bi=bi,alpha.u=alpha.u,rmse=result.tibble))
}

#r0<-als.td(f=10,lambda = 0.1,max.iter = 5,data,data_train,data_test)
#save(r0,file = "example_in_als+td.RData")
```

Here, we show an example of $f = 10, \lambda = 0.1, max.iter = 5$ of RMSE, both training and test.

```
load("../output/example_in_als+td.RData")
print(r0$rmse)
```

```
## # A tibble: 5 x 2
##   train_rmse test_rmse
```

```
##         <dbl>     <dbl>
## 1       0.900     0.989
## 2       0.660     0.898
## 3       0.621     0.898
## 4       0.606     0.906
## 5       0.598     0.914
```

Step 2.2 - Parameter Tuning In this step, we will tune the parameters using cross validation.

```
source( '../lib/cv_saier.R')
```

Cross Validation

```
####cross validation


f_list <- seq(10, 20, 10)
l_list <- seq(-2, -1, 1)
f_l <- expand.grid(f_list, l_list)
K=5

#train_summary<-matrix(0,nrow = 5,ncol = 4)
#test_summary<-matrix(0,nrow = 5,ncol = 4)
#for(i in 1:nrow(f_l)){
#    par <- paste("f = ", f_l[i,1], ", lambda = ", 10^f_l[i,2])
#    cat(par, "\n")
#    current_result <- cv.function(data_train=data_train, K = 5, f = f_l[i,1], lambda = 10^f_l[i,2])
#    train_summary[,i]<-current_result$mean_train_rmse
#    test_summary[,i]<-current_result$mean_test_rmse
#
#    #print(train_summary)
#    #print(test_summary)
#
#}

#print(train_summary)
#print(test_summary)
```

Loading the train summary and test summary to get the results

```
#save(train_summary,file = "train_summary.RData")

#save(test_summary,file = "test_summary.RData")
load("../output/train_summary.RData")
load("../output/test_summary.RData")

#colnames(test_summary)<-c("test10_0.01","test20_0.01","test10_0.1","test20_0.1")

print(train_summary)
```

```
##      train10_0.01 train20_0.01 train10_0.1 train20_0.1
## [1,]    0.6938345    0.5790885   0.8786546   0.8238524
```

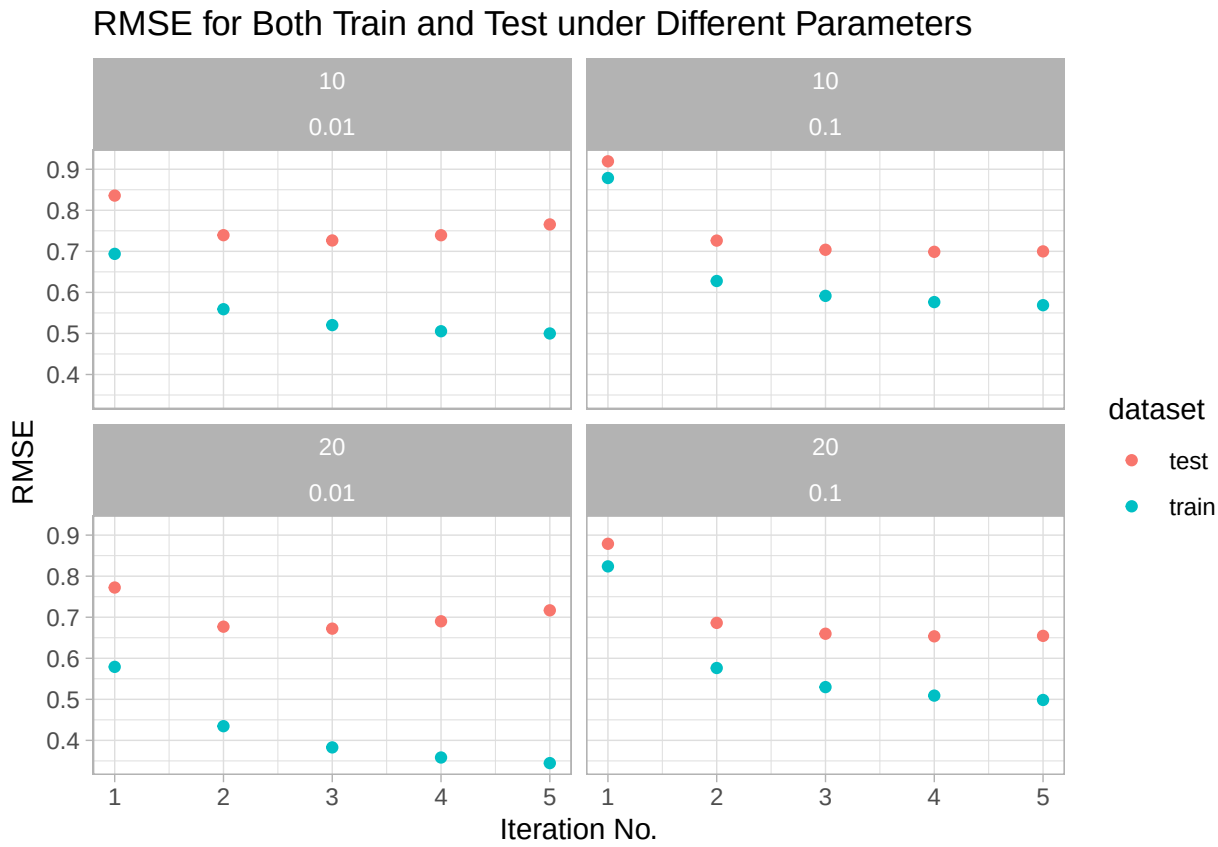
```
## [2,]    0.5590424    0.4345692   0.6277711   0.5762773
## [3,]    0.5203303    0.3829107   0.5915101   0.5297911
## [4,]    0.5053706    0.3583057   0.5763731   0.5090470
## [5,]    0.4999065    0.3446382   0.5688656   0.4984395
```

```
print(test_summary)
```

```
##      test10_0.01 test20_0.01 test10_0.1 test20_0.1
## [1,]   0.8358058   0.7723412  0.9193069  0.8788701
## [2,]   0.7393705   0.6769988  0.7261995  0.6861949
## [3,]   0.7264820   0.6721865  0.7038050  0.6597834
## [4,]   0.7392118   0.6900889  0.6988539  0.6534687
## [5,]   0.7655292   0.7168115  0.6998979  0.6546065
```

In this step, we tried to plot the RMSE values of both train and test under different parameters, which are f and λ. As illustrated from the plot, we can observe that the RMSE are more stable when we reached 3 iterations.

```
# Creating the plot
s.summary<-tibble(iteration=rep(1:5,4),
                  f=c(rep(10,10),rep(20,10)),
                  lambda=c(rep(0.01,5),rep(0.1,5),rep(0.01,5),rep(0.1,5)),
                  train=c(train_summary[,1],train_summary[,3],train_summary[,2],train_summary[,4]),
                  test=c(test_summary[,1],test_summary[,3],test_summary[,2],test_summary[,4])) %>%
  pivot_longer(cols = train:test,names_to = "dataset",values_to = "rmse")

q<-ggplot(data = s.summary,mapping = aes(x=iteration,y=rmse))+
  geom_point(mapping = aes(color=dataset))+
  facet_wrap(f~lambda)+
  labs(title = "RMSE for Both Train and Test under Different Parameters",
       x="Iteration No.",
       y="RMSE")+
  theme_light()
q
```



Before we move into post-processing with KNN, we will observe the values of RMSE under different aforementioned parameters, f and λ, for both train and test dataset. Based on the results below, we can conclude that the RMSE values for both train and test dataset are at their lowest for $f = 20, \lambda = 0.01$, therefore, we will proceed with those values for post-processing.

```
print(colMeans(train_summary))
```

```
## train10_0.01 train20_0.01  train10_0.1  train20_0.1
##    0.5556969    0.4199025    0.6486349    0.5874815
```

```
print(colMeans(test_summary))
```

```
## test10_0.01 test20_0.01  test10_0.1  test20_0.1
##   0.7612799   0.7056854   0.7496126   0.7065847
```

Step 3 - Postprocessing with KNN

In this last part, we will run the model with optimized parameters, $f = 20, \lambda = 0.01$.

```
#r <-als.td(f=20,lambda=0.01,max.iter = 5,data = data,data_train = data_train,data_test = data_test)
#save(r,file = "optimize_rmse_and_parameters.RData")
load("../output/optimize_rmse_and_parameters.RData")
print(r$rmse)
```

Step 3.1 - Run the Model with Optimized Parameters

```
## # A tibble: 5 x 2
##   train_rmse test_rmse
##        <dbl>     <dbl>
## 1      0.629     0.768
## 2      0.477     0.652
## 3      0.427     0.638
## 4      0.403     0.643
## 5      0.390     0.656
```

```
source("../lib/knn_saier.R")
load("../output/answer_train.RData")
print(answer.train$rmse)
```

Step 3.2 - Postprocessing with KNN

```
## [1] 0.7929424
```

```
load("../output/answer.RData")
print(answer$rmse)
```

```
## [1] 0.8734229
```

```
(tibble(train=answer.train$rmse,
        test=answer$rmse))
```

```
## # A tibble: 1 x 2
##   train  test
##   <dbl> <dbl>
## 1 0.793 0.873
```

Based on the table above, we observe RMSE results of **0.793** for train dataset and **0.873** for test dataset.

Section 2: Model 2 Steps and Output

After observing the performance of model 2, we will run Model 2, that is without regularization steps.

Step 1 - Load Data and Train-test Split

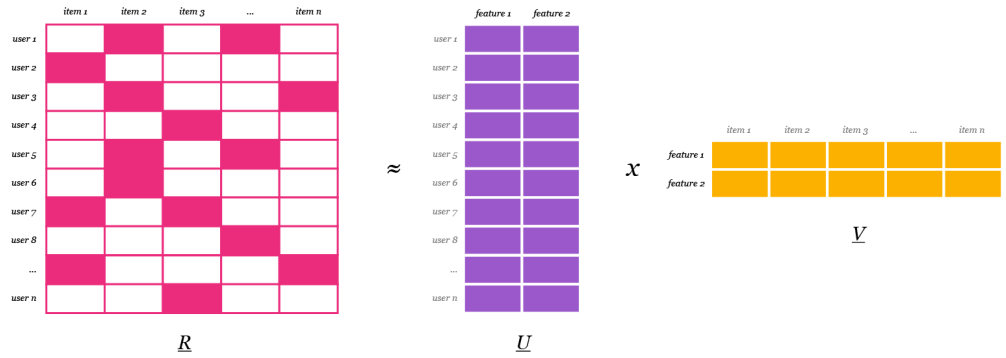
The data sets provided by Netflix consists hundred thousand movie ratings by users (on a 1–5 scale). This is an explicit feedback dataset.The users are explicitly telling us how they would rate a movie. The main data file consists of a tab-separated list with user-id (starting at 1), item-id (starting at 1), rating, and timestamp as the four fields.

```
library(dplyr)
library(tidyr)
library(ggplot2)
data <- read.csv("../data/ml-latest-small/ratings.csv")
set.seed(0)
## shuffle the row of the entire dataset
data <- data[sample(nrow(data)),]
## get a small dataset that contains all users and all movies
unique.user<-duplicated(data[,1])
unique.movie<-duplicated(data[,2])
index<-unique.user & unique.movie
all.user.movie <- data[!index,]

## split training and test on the rest
rest <- data[index,]
test_idx <- sample(rownames(rest), round(nrow(data)/5, 0))
train_idx <- setdiff(rownames(rest), test_idx)

## combine the training with the previous dataset, which has all users and all movies
data_train <- rbind(all.user.movie, data[train_idx,])
data_test <- data[test_idx,]

## sort the training and testing data by userId then by movieId,
## so when we update p and q, it is less likely to make mistakes
data_train <- arrange(data_train, userId, movieId)
data_test <- arrange(data_test, userId, movieId)
```

Step 2 - Matrix Factorization

With our training and test ratings matrices in hand, we can now move towards training a recommendation system. Matrix factorization is the state-of-the-art solution for sparse data problem. In the case of collaborative filtering, matrix factorization algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices. One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items.

```
knitr::include_graphics("../figs/matrix.png")
```



Step 2.1 - Initialize matrix and define function for Alternating Least Squares

Here we try ALS without regularization at first.

A3. Alternating Least Squares Section 3.1

ALS is an iterative optimization process where we for every iteration try to arrive closer and closer to a factorized representation of our original data.

For ALS minimiztion, we hold one set of latent vectors constant. For this example, we'll pick the Movie vectors. We then take the derivative of the loss function with respect to the other set of vectors (the user vectors). We set the derivative equal to zero (we're searching for a minimum) and solve for the non-constant vectors (the user vectors). Now comes the alternating part: With these new, solved-for user vectors in hand, we hold them constant, instead, and take the derivative of the loss function with respect to the previously constant vectors (the Movie vectors). We alternate back and forth and carry out this two-step dance until convergence.

```
U <- length(unique(data$userId))
I <- length(unique(data$movieId))
source("../lib/ALS.R")
```

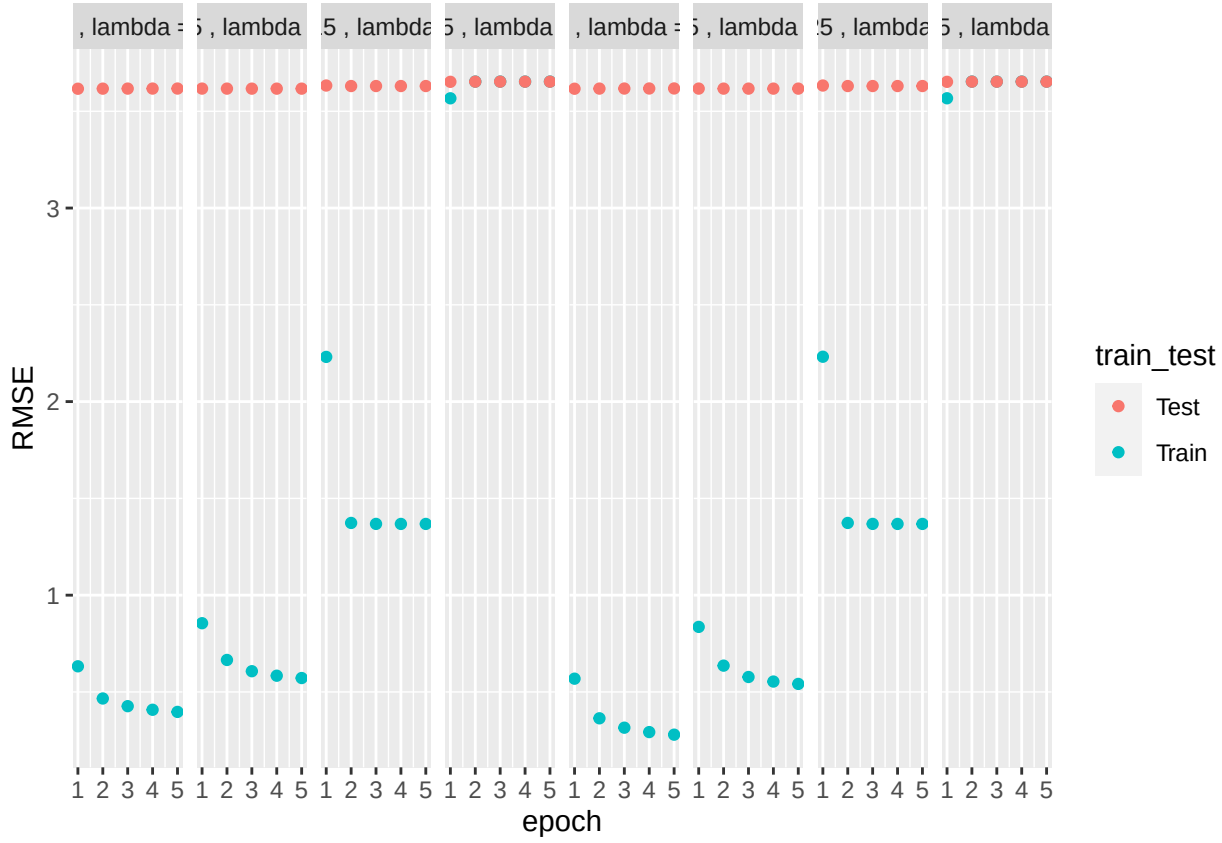
Step 2.2 - Parameter Tuning In this step, we are tuning the parameters, such as the dimension of factor f and the penalty parameter λ by cross-validation.

```
source("../lib/cr_ALS.R")
f_list <- seq(15, 25, 10)
l_list <- seq(-2,1,1)
f_l <- expand.grid(f_list, l_list)
```

```
#result_summary <- array(NA, dim = c(nrow(f_l), 10, 8))
#run_time <- system.time(for(i in 1:nrow(f_l)){
#    par <- paste("f = ", f_l[i,1], ", lambda = ", 10^f_l[i,2])
#    cat(par, "\n")
#    current_result <- cv.function(data, K = 3, f = f_l[i,1], lambda = 10^f_l[i,2])
#    result_summary[,,i] <- matrix(unlist(current_result))
#    print(result_summary)

#})
#save(result_summary, file = "../output/rmse_als_a33.Rdata")
```

It takes a while to run this chunk so we have saved the output and can be loaded directly. As it is shown in the graph, We choose the iteration number as 5. Best parameters in cv is $(f = 25, \lambda = 0.1)$.

```
load("../output/rmse_als_a3.Rdata")
rmse <- data.frame(rbind(t(result_summary[1,1:5,]), t(result_summary[2,1:5,])), train_test = rep(c("Tra
rmse$epoch <- as.numeric(gsub("X", "", rmse$epoch))
rmse %>% ggplot(aes(x = epoch, y = RMSE, col = train_test)) + geom_point() + facet_grid(~par)
```



Step 3 - Post-processing with KNN

After matrix factorization, postporcessing will be performed to improve accuracy. The referenced papers:

P2:Postprocessing SVD with KNN Section 3.5

We used KNN to updated all rating. For example we choose an unseen movie j for a certain user u from matrix q and we calculate the cosine similarity to compare the latent factors of the movie j with the latent factor of other movies. We choose the most similar movie i and calculate the average rating of movie i. Finally we update the rating of movie j by user u with the mean of its similar movies.

```
#result <- ALS(f = 25, lambda =0.1,
                    max.iter = 5, data = data, train = data_train, test = data_test)

#save(result, file = "../output/mat_fac_als_a33.RData")
```

```
load("../output/mat_fac_als_a33.RData")
#library(lsa)

#ratingmean<-function(data){
#  mean(data$rating)
#}
#rating<-ddply(data,.(movieId),ratingmean)
#rating$index<-c(1:nrow(rating))
#l_distance<-cosine(result$Movie)
#diag(l_distance)<-0
#cos_id<-rep(1:I,ncol=I)
#knn_r<-rep(1:I,ncol=I)
#for (i in 1:I){
#    cos_id[i]<-which.max(rank(l_distance[,i]))
#    knn_r[i]<-rating[rating$index==cos_id[i],]$V1
#}
#save(knn_r, file = "../output/knn_r.RData")
```

Step 3.1 - KNN Post-processing

Step 3.2 - Linear Regression for Training dataset After post-processing, we use linear regression and treat these output from previous step (ALS, regularization and post-processing) as input to predict the expectation movie rating and calculate the rmse.

```
load("../output/knn_r.RData")
data_train$movieindex<-dense_rank(data_train$movieId)
r<-rep(0,nrow(data_train))
for (i in 1:nrow(data_train)){
  rowindex<-data_train$userId[i]
  columindex<-data_train$movieindex[i]
  qi <- as.matrix(result$User[,rowindex])
  qj <- as.matrix(result$Movie[,columindex])
  r[i]<-t(qi)%*%qj
  }
w<-knn_r[data_train$movieindex]
```

```
data_train_linear<-as.data.frame(cbind(data_train$rating,r,w))
fit<-lm(V1~r+w,data=data_train_linear)
exp_rating<-predict(fit,data_train_linear)
(rmse_adj <- sqrt(mean((data_train_linear$V1 - exp_rating)^2)))
```

```
## [1] 0.5232906
```

```
cat("The RMSE train of the adjusted model is", rmse_adj)
```

```
## The RMSE train of the adjusted model is 0.5232906
```

```
data_test$movieindex<-dense_rank(data_test$movieId)
r1<-rep(0,nrow(data_test))
for (i in 1:nrow(data_test)){
  rowindex<-data_test$userId[i]
  columindex<-data_test$movieindex[i]
  qi <- as.matrix(result$User[,rowindex])
  qj <- as.matrix(result$Movie[,columindex])
  r1[i]<-t(qi)%*%qj
}
w1<-knn_r[data_test$movieindex]
```

```
data_test_linear<-as.data.frame(cbind(data_test$rating,r1,w1))
exp_rating_test<-predict(fit,data_test_linear)
rmse_adj_test <- sqrt(mean((data_test_linear$V1 - exp_rating_test)^2))
cat("The RMSE test of the adjusted model is", rmse_adj_test)
```

Step 3.3 - Prediction

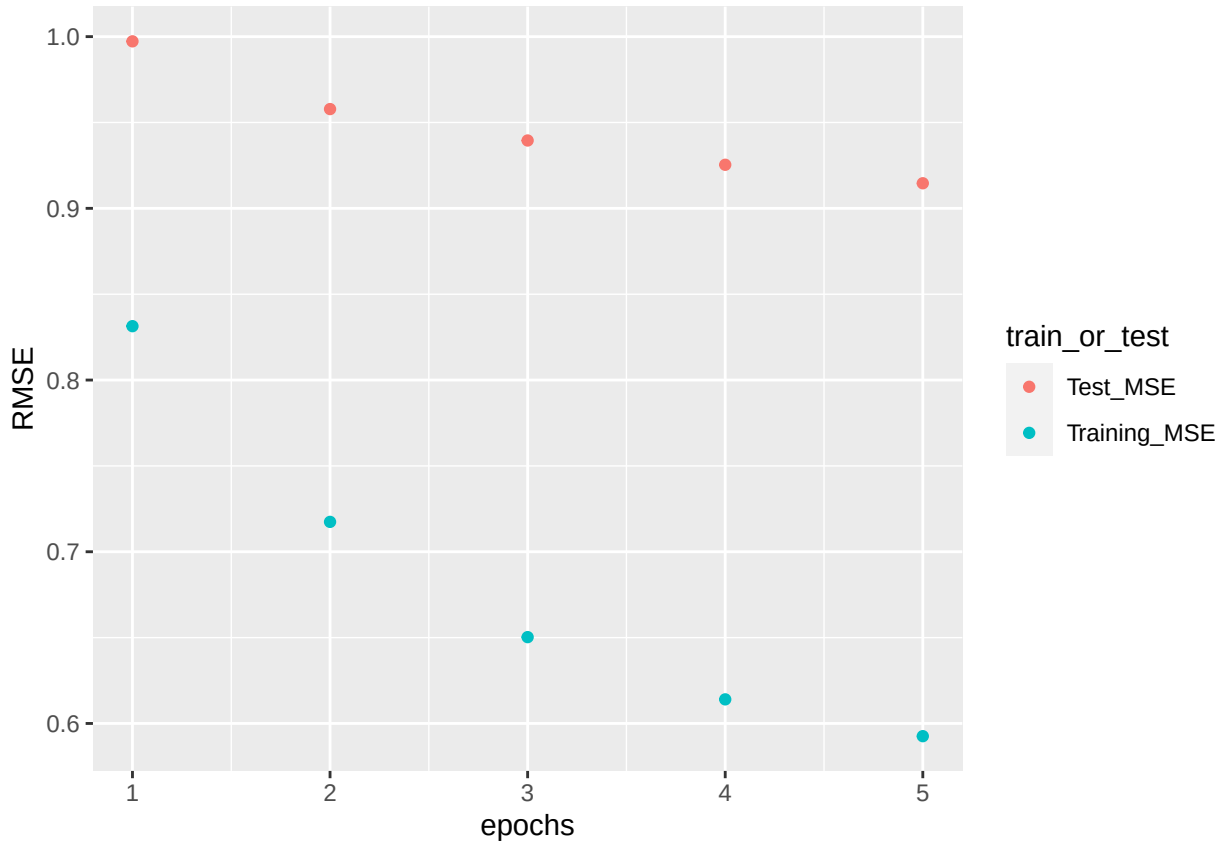
```
## The RMSE test of the adjusted model is 1.370012
```

Step 4 - Evaluation

You should visualize training and testing RMSE by different dimension of factors and epochs (One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE).

Before postprocessing, our accuracy for train and test datasets are 0.59 and 0.9135011.After postprocessing, our accuracy for train and test datasets are 0.5892697 and 1.342821. After postprocessing, the training rmse decrease but testing rmse increase which means we have problem with overfitting. Therefore our algorithms before postprocessing works well and we do not need postprocessing for Alternative Least Squares algorithms.

```
library(ggplot2)
RMSE <- data.frame(epochs = seq(1, 5, 1), Training_MSE = result$train_RMSE, Test_MSE = result$test_RMSE
RMSE %>% ggplot(aes(x = epochs, y = RMSE,col = train_or_test)) + geom_point() + scale_x_discrete(limits
```



Section 3 - Evaluation and Conclusion

The summary of the model performances is as follows.

```
##     Model         Method  f lambda Train RMSE Test RMSE
## 1 Model 1 ALS + TD + KNN 20   0.01      0.793     0.873
## 2 Model 2      ALS + KNN 10    0.1      0.591     1.344
```

- As we can observe from the table above, Model 2 has a lower RMSE value for train dataset (0.589) compared to Model 1 (0.793).
- However, Model 1 performs better for test dataset based on the test RMSE value of 0.873 vs. model 2 of 1.343. Therefore, we can conclude that **Model 1 (with regularization) has a better fit for test data compared to model 2 (withour regularization)**.
- This conclusion is reasonable since regularization reduces overfitting, which might be the reason why Model 1 performs better in test dataset.