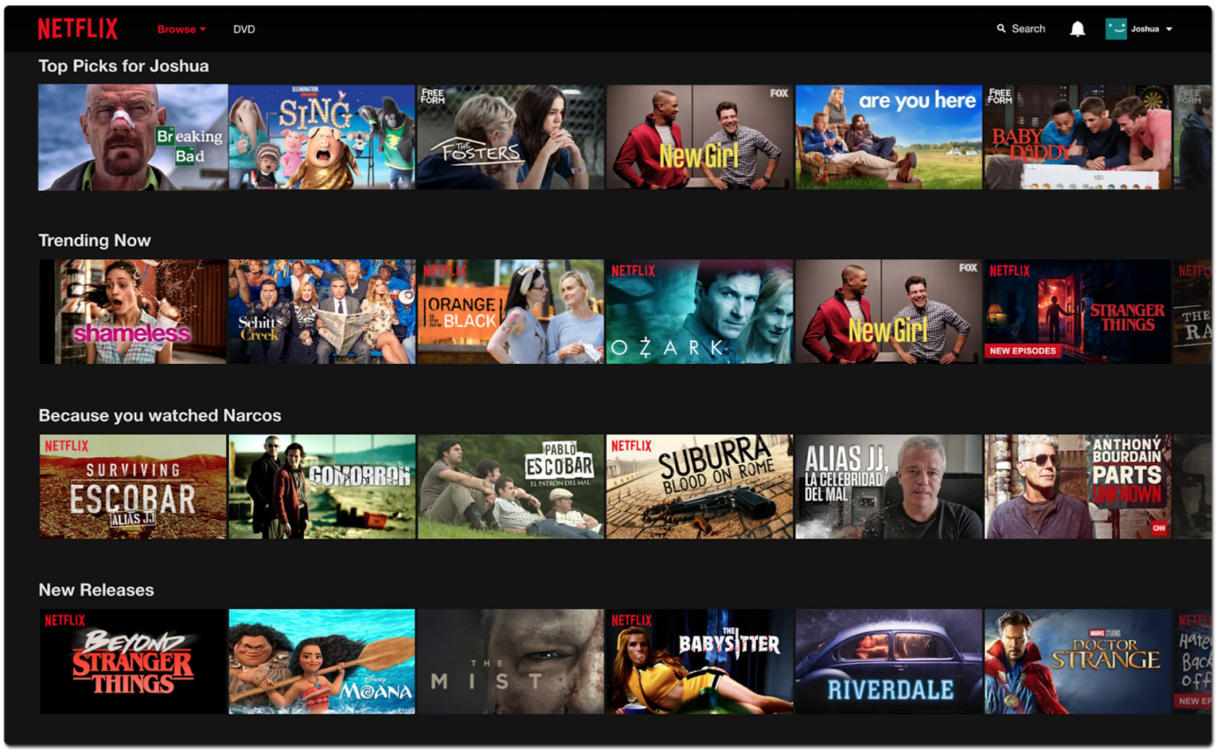
Project 4 Collaborative Filtering Algorithms Evaluation

Ran Lu, Haoyu Shang, Rui Wang, Qin Zhang, Tianya Zhang

2020/04/21

Overview



In this project, we made a collaborative filtering system used to automatically recommend movies to different users based on the rating data from Netflix.

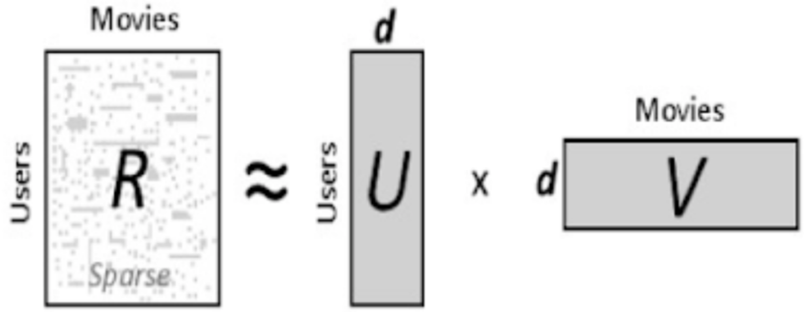
The methods we used are alternating lease squares and Gradient Descent with Probabilistic Assumption. Besides, we also use KNN in the postprocessing to increase our accuracy.

Recommendation System: Matrix Factorization

There are a wide range of algorithms to generate recommendations. While undoubtedly user-based or item-based collaborative filtering methods are simple and intuitive, matrix factorization technique is usually more efficient and effective because this provide an access to discover the latent features underlying the interactions between users and items. Matrix factorization is simply a basic mathematical tool for processing the data and matrices and therefore can be used in lots of different cases and scenarios to discover the potential connection between the data

MATRIX FACTORIZATION TECHNIQUES

$$R_{ij} = Q_j^T P_i$$



We have M movies and N users, and denote Q and P as latent movie and user feature matrices. Adopt a probabilistic linear model with Gaussian observation noise, we define the conditional distribution over the observed ratings:

$$p(R|P, Q, \sigma^2) = \prod_{i=1}^{N} \prod_{j=1}^{M} [\mathcal{N}(R_{ij}|Q_i^T P_j, \sigma^2)]^{I_{ij}}$$

with

$$p(P|\sigma_P) = \prod_{j=1}^{M} \mathcal{N}(P_j|0, \sigma_P^2 \mathbf{I}) \qquad p(Q|\sigma_Q) = \prod_{i=1}^{N} \mathcal{N}(Q_i|0, \sigma_Q^2 \mathbf{I})$$

I_{ij} is the indicator function that is equal to 1 if user i rated movie j and 0 otherwise

Then maximizing the log-posterior over movie and user features with hyperparameters is equivalent to minimizing the sum of squared errors objective function with quadratic regularization terms:

$$E = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{M} I_{ij} (R_{ij} - Q_i^T P_j)^2 + \frac{\sigma^2}{2\sigma_Q^2} \sum_{i=1}^{N} ||Q_i||_F^2 + \frac{\sigma^2}{2\sigma_P^2} \sum_{j=1}^{M} ||P_j||_F^2$$

Our recommendation system is composed of three main methods: Gradient descent Model, ALS model and KNN techniques to increase the accuracy

Step 1 Load Data and Train-test Split

```
library(dplyr)
library(tidyr)
library(ggplot2)
data <- read.csv("../data/ml-latest-small/ratings.csv")

set.seed(123)
test_idx <- sample(1:nrow(data), round(nrow(data)/5, 0))
train_idx <- setdiff(1:nrow(data), test_idx)
data_train <- data[train_idx,]
data_test <- data[test_idx,]
```

Gradient descent Model

We are going to update each iteration with the following:

$$P_j^{(t+1)} = P_j^{(t)} + \text{lrate} \sum_{i=1}^{N} I_{ij}(R_{ij} - Q_i^{T(t)} P_j^{(t)}) Q_i^{T(t)} - \frac{\sigma^2}{\sigma_P^2} P_j^{(t)}$$

$$Q_i^{(t+1)} = Q_i^{(t)} + \text{lrate} \sum_{j=1}^{M} I_{ij}(R_{ij} - Q_i^{T(t)} P_j^{(t)}) P_j^{(t)} - \frac{\sigma^2}{\sigma_Q^2} Q_i^{(t)}$$

```
gradesc_pmf <- function(f = 10,
                        sigma_p = 1, sigma_q = 1, sigma = 0.1, lrate = 0.01, max.iter = 100, stopping.deriv = 0.01,
                        data, train, test){
  set.seed(0)
  #random assign value to matrix p and q
  p <- mvrnorm(f,mu = rep(0,U),Sigma = sigma_p*diag(U))
  colnames(p) <- as.character(1:U)
  #q <- mvrnorm(f,mu = rep(0,I),Sigma = sigma_q*diag(I))
  q <- matrix(rnorm(f*I, 0, sqrt(sigma_q)), ncol = I, nrow = f)
  colnames(q) <- levels(as.factor(data$movieId))
```

```
for(l in 1:max.iter){
  sample_idx <- sample(1:nrow(train), nrow(train))
  #loop through each training case and perform update
  for (s in sample_idx){

    u <- as.character(train[s,1])

    i <- as.character(train[s,2])

    r_ui <- train[s,3]

    e_ui <- r_ui - t(q[,i]) %*% p[,u]

    grad_q <- e_ui %*% p[,u] - (sigma/sigma_q) * q[,i]

    if (all(abs(grad_q) > stopping.deriv, na.rm = T)){
      q[,i] <- q[,i] + lrate * grad_q
    }

    grad_p <- e_ui %*% q[,i] - (sigma/sigma_p) * p[,u]

    if (all(abs(grad_p) > stopping.deriv, na.rm = T)){
      p[,u] <- p[,u] + lrate * grad_p
    }
  }
```

```
```{r}
U <- length(unique(data$userId))
I <- length(unique(data$movieId))
source("../lib/Matrix_Factorization_A2.R")
```
source("../lib/cross_validation_A2.R")
f_l <- cbind(f = c(5, 10, 20, 5, 10, 20, 5, 10, 20),
             sigma_p = c(1, 1, 1, 0.5, 0.5, 0.5, 1.5, 1.5, 1.5),
             sigma_q = c(1, 1, 1, 0.5, 0.5, 0.5, 1.5, 1.5, 1.5))
```

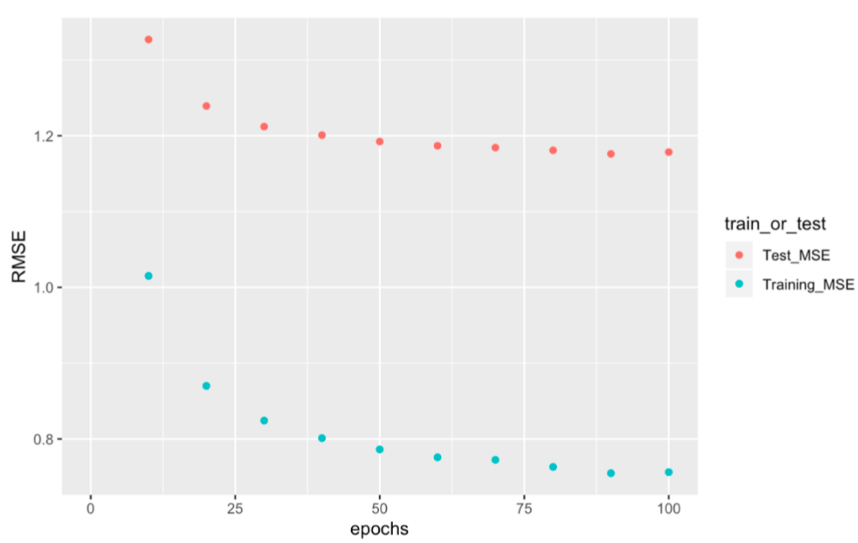
Then we tune parameters through cross-validation

f = 5

sigma_p = 0.5

sigma_q = 0.5

lrate = 0.01



ALS model

Because both qi and pj are unknowns, the J function is not convex. However, if we fix one of the unknowns, the optimization problem becomes quadratic and can be solved optimally (using normal equation). Thus, ALS technique rotates between fixing qi 's and pu's. When all pu's are fixed, the system recomputes the qi's by solving a least-squares problem, and vice versa. Again, this is an iterative process, but suitable for parallelization. For example, in ALS the system computes each pu independently of the other user factors (so we can solve normal equations for different users in parallel). The same holds for calculating item factors.

ALS with Tikhonov regulation (penalizes large parameters)

$$\min_{Q,P} f(Q,P) = \sum_{i,j \in K} (R_{ij} - Q_i^T P_j)^2 + \lambda(\sum_i n_{Q_i} ||Q_i||^2 + \sum_j n_{P_j} ||P_j||^2)$$

During each iteration, fix Q to solve P, and fix P to solve Q by:

$$P_i = (Q_{I_i} Q_{I_i}^T + \lambda n_{P_i} E)^{-1} Q_{I_i} R^T(i, I_i)$$
$$Q_j = (P_{I_j} P_{I_j}^T + \lambda n_{Q_j E})^{-1} P_{I_j} R(I_j, j)$$

n_{P_i} and n_{Q_j} are the numbers of ratings of user i and movie j respectively
E is the $n_f \times n_f$ identity matrix
Q_{I_i} is the sub-matrix of M where columns $j \in I_i$ are selected
$R(i, I_i)$ is the row vector where columns $j \in I_i$ of the i-th row of R is taken
$R(I_j, j)$ is the column vector where rows $i \in I_i$ of the j-th column of R is taken

```
als <- function(f = 10, lambda = 0.3, max.iter=1, data, train, test){
  # random assign value to matrix p and q
  p <- matrix(runif(f*U, -1, 1), ncol = U)
  colnames(p) <- as.character(1:U)
  q <- matrix(runif(f*I, -1, 1), ncol = I)
  colnames(q) <- levels(as.factor(data$movieId))
  rate <- data %>% select(movieId, rating) %>% group_by(movieId) %>% summarise(rate_avg = mean(rating))
  q[1,] <- rate$rate_avg
```

```r
# fix q, compute p
  for(u in 1:U){
    user <- train[train$userId==u,]
    M.u <- q[,as.character(user$movieId)]
    A.u <- M.u%*%t(M.u) + lambda*nrow(user)*diag(f)
    R.u <- user$rating
    V.u <- M.u%*%R.u
    p[,u] <- solve(A.u)%*%V.u
  }

# fix p, compute q
  for(i in 1:I){
    movie <- train[train$movieId==colnames(q)[i],]
    U.i<-p[,as.character(movie$userId)]
    A.i<-U.i%*%t(U.i)+lambda*nrow(movie)*diag(f)
    R.i<-movie$rating
    if (length(R.i)==1){
      V.i<-U.i*R.i
      q[,i]<-solve(A.i)%*%V.i
    }
    if (length(R.i)>1){
      V.i<-U.i%*%R.i
      q[,i]<-solve(A.i)%*%V.i
    }
  }
```

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

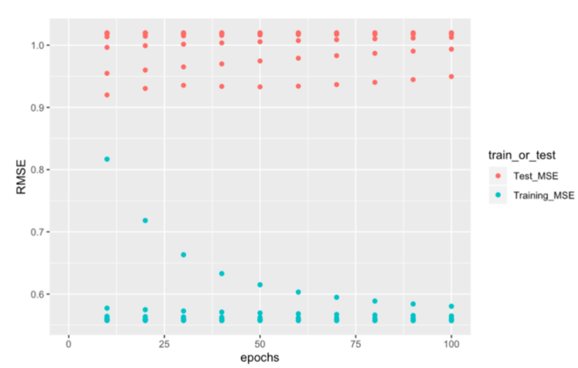
```r
source("../lib/cross_validation_ALS.R")
f_list <- seq(10, 20, 10)
l_list <- seq(-1, 0, 1)
f_l <- expand.grid(f_list, l_list)
```

```{r}
result_summary <- array(NA, dim = c(nrow(f_l), 10, 4))
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.als.function(data, K = 5, f = f_l[i,1], lambda = 10^f_l[i,2])
  result_summary[,,i] <- matrix(unlist(current_result), ncol = 10, byrow = T)
  print(result_summary)

})

save(result_summary, file = "../output/rmse_als.Rdata")
```

Then we adjust the parameters:

f = 5

lambda = 0.1



As we can see, the train RMSE is 0.557, test RMSE is 1.017. Both of them are improved comparing with sample code.

Postprocessing with KNN

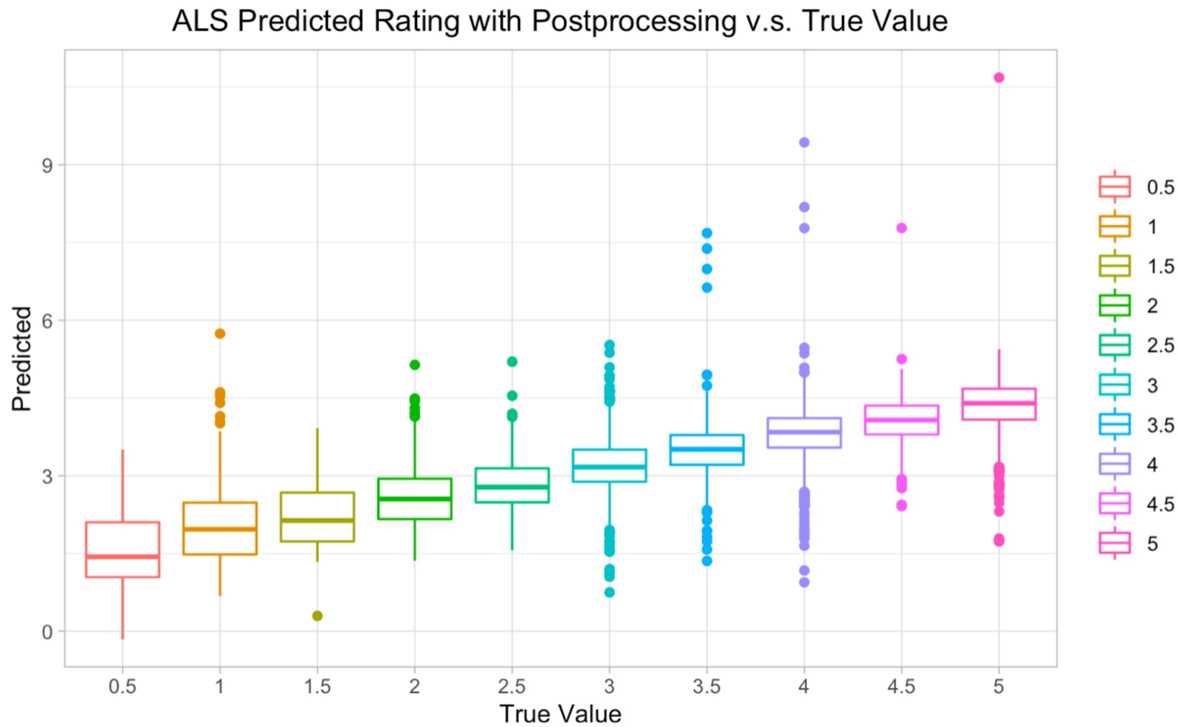
In post processing part, we use KNN to improve our prediction accuracy.
Define similarity between movies j and j2 as cosine similarity between vectors vj and vj2

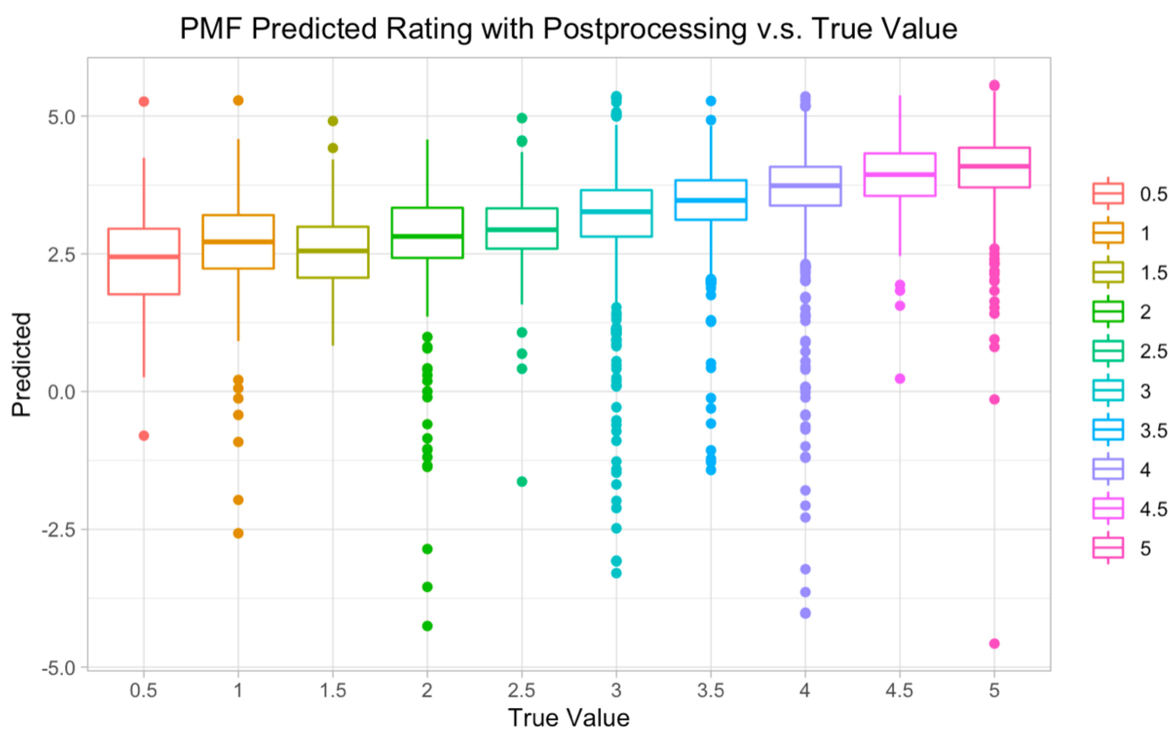
$$s(v_j, v_{j_2}) = \frac{v_j^T v_{j_2}}{||v_j||||v_{j_2}||}$$

```
if(!require("lsa")){
  install.packages("lsa")
}
library(lsa)
library(tidyverse)
# cos_mat_q <- cosine(q)
# save(cos_mat_q, file = "../output/cos_pmf.RData")
load("../output/cos_pmf.RData")

colnames(cos_mat_q) <- colnames(q)
rownames(cos_mat_q) <- colnames(q)
index <- rep(NA, ncol(cos_mat_q))
for (i in 1:ncol(cos_mat_q)) { # find the index of the nearest neighbor movie
  vec <- cos_mat_q[,i]
  index[i] <- order(vec, decreasing = TRUE)[2]
}

original_rating <- t(q) %*% p # The estimated rating before post processing

mean_rating_knn <- data %>% group_by(movieId) %>% summarize(mean_rating_knn = mean(rating))
mean_rating_knn_movieId <- as.character(unique(data_train$movieId))
knn_rating <- cbind(mean_rating_knn_movieId,mean_rating_knn[index[which(colnames(cos_mat_q)%in%mean_rating_knn_movieId)],2])
```

```
train_userID <- as.character(data_train$userId)
train_movieID <- as.character(data_train$movieId)
predictor_before <- vector()
knn_predictor <- vector()
for(w in 1:nrow(data_train)){ # extract the predictor for the set before and after postprocessing
  predictor_before <- c(predictor_before, original_rating[train_movieID[w], train_userID[w]])
  knn_predictor <- c(knn_predictor, knn_rating[train_movieID[w],2])
}
y_knn <- data_train$rating
train_knn <- cbind(predictor_before = predictor_before, knn_predictor = knn_predictor, y_knn = y_knn)
train_knn <- na.omit(train_knn)
train_knn <- as.data.frame(train_knn)

knn_model <- lm(y_knn ~ knn_predictor + predictor_before)
```

Before postprocessing, our accuracy for train datasets of gradient model is 1.179. After postprocessing, our accuracy for test datasets are 0.985. And the train datasets of ALS before is 1.017 and after is 0.664 which means after KNN, the accuracy of two model has improved a lot.


ALS Predicted Rating with Postprocessing v.s. True Value
Predicted
True Value
0
3
6
9
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5


PMF Predicted Rating with Postprocessing v.s. True Value
Predicted
True Value
-5.0
-2.5
0.0
2.5
5.0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Evaluation

```
    cat("epoch:", l, "\t")
    est_rating <- t(q) %*% p
    rownames(est_rating) <- levels(as.factor(data$movieId))

    train_RMSE_cur <- RMSE(train, est_rating)
    cat("training RMSE:", train_RMSE_cur, "\t")
    train_RMSE <- c(train_RMSE, train_RMSE_cur)

    test_RMSE_cur <- RMSE(test, est_rating)
    cat("test RMSE:",test_RMSE_cur, "\n")
    test_RMSE <- c(test_RMSE, test_RMSE_cur)
    }
  return(list(p = p, q = q, train_RMSE = train_RMSE, test_RMSE = test_RMSE))
}
```

	Gradient Descent with Probabilistic Assumptions	Alternative Least Squares
RMSE Before postprocessing	Train 0.756 Test 1.179	Train 0.557 Test 1.017
RMSE After postprocessing	Test 0.985	Test 0.664

From the RMSE table above, we can see the RMSEs for Gradient Descent with Probabilistic Assumption for both train and test are larger than the RMSEs for Alternative Least Squares. Moreover during the our running process, we found out that the model training time for Gradient Descent with Probabilistic Assumptions is much longer than that of ALS

After doing post processing, we can see the RMSE for Gradient Descent with Probabilistic Assumption and ALS model for test dataset has a obvious improvement, which means the postprocessing is effective.