Collaborative Filtering with Matrix Factorization

Group 8

11/21/2019

Contents

Part 1: Implementing Gradient Descent with Probabilistic Assumptions 1
Part 2: Implementing Alternating Least Squares 4
Part 3: Post-processing and Evaluation for Alternating Least Squares 6
Part 4: Evalutation 9

Abstract

In this project, we implemented two algorithms for collaborative filtering from scratch: Gradient Descent with Probabilistic Assumptions (A2 in the assignment) and Alternating Least Squares (A3). Afterwards, we used Kernel Ridge Regression (P3) for post-processing. Our results indicate that KRR is a significant improvement for the algorithms. Also, we found that the more latent factors we use, the lower the final RMSE. Overall, the ALS outperforms GD with Probabilistic Assumptions for all levels of latent factors on the the test data.

Note on the scope of this report This report outlines matrix factorization algorithms with 100 latent factors. We also conducted the same procedure for 10 as well as 50 factors but only load these results in Part 4 and do not run them in this .Rmd.

Data loading and preparation

We prepare the data by first shuffling it. Then we split the data to train and test by a ratio of 80:20 and save them as .CSV file. We want to use exactly the same data for each algorithm for the sake of a fair comparison.

```
rating_path = '../data/ml-latest-small/ratings.csv'

rating_pd = pd.read_csv(rating_path)
num_user = np.unique(rating_pd.values[:, 0]).shape[0]
num_movie = np.unique(rating_pd.values[:, 1]).shape[0]

#df_shuffled = rating_pd.sample(frac=1).reset_index(drop=True)
#train_set = df_shuffled.iloc[:int(train_frac * len(df_shuffled))]
#train_set.to_csv(path_or_buf=train_path, index=False)
#test_set = df_shuffled.iloc[int(train_frac * len(df_shuffled)):]
#test_set.to_csv(path_or_buf=test_path, index=False)

train_path = '../output/train_set.csv'
test_path = '../output/test_set.csv'
```

Part 1: Implementing Gradient Descent with Probabilistic Assumptions

Set constant values

For both algorithms we are implementing we use three levels of latent factors: 10, 50 and 100. At first we have to define the constant values for GD with Probabilistic Assumptions including the number of latent factors. We set the mean for multivariate normal distribution we initialize to 0 and sigma to 0.5.

```
train_frac = 0.8
latent_dim = 100 # can be changed to implement algorithm with other factor levels
sigma = 0.5
sigma_p = 0.5
sigma_q = 0.5
mu = 0
```

1.1: Create the matrices

Afterwards, we want to create the matrices as well as a movie_ID dictionary for post-processing later.

```
train_rating_pd = pd.read_csv(train_path)

r_matrix = np.zeros((num_user, num_movie))
p_matrix = np.random.normal(mu, sigma_p, (num_user, latent_dim))
q_matrix = np.random.normal(mu, sigma_q, (num_movie, latent_dim))

movie_dic = {}
idx = 0
for movie_id in np.unique(train_rating_pd.values[:, 1]):
    movie_dic[movie_id] = idx
    idx += 1

movie_IDD = set(rating_pd['movieId'].unique().tolist())
train_ID = set(train_rating_pd['movieId'].unique().tolist())
not_in_training_ID = movie_IDD - train_ID

for index, row in train_rating_pd.iterrows():
    r_matrix[int(row['userId'] - 1), movie_dic[row['movieId']]] = row['rating']

sorted_dic = sorted(movie_dic.items(), key=operator.itemgetter(1))

# Saving the movie_ID dictionary
with open('../output/movie_indexes.csv', 'w') as f:
    f.write('matrix_index,movie_id\n')
    for movie_id, index in sorted_dic:
        f.write('{},{}\n'.format(index, int(movie_id)))
```

```
index = len(sorted_dic)
with open('../doc/movie_indexes.csv', 'a') as f:
    for movie_id in not_in_training_ID:
        f.write('{},{}\n'.format(index, int(movie_id)))
        index += 1
```

1.2: Implement the algorithm

We then define the functions for the algorithm.

```
def MSE (r, p, q):
    p_qt_matrix = np.dot(p, q.T)
    result = r - p_qt_matrix
    I = np.zeros_like(r)
```

```
    I[r != 0] = 0.5
    result = np.power(result, 2)
    result = np.multiply(I, result)
    return np.sum(result)

def L2_loss(sig1, sig2, matrix):
    result = np.power(matrix, 2)
    result = np.sum(result)
    constant = sig1 / float(sig2 * 2)
    return constant * result

def p_q_derivative (r, p, q, sigma, sigma_p, sigma_q):
    p_qt_matrix = np.dot(p, q.T)
    residual = r - p_qt_matrix
    I = np.zeros_like(r)
    I[r != 0] = 1
    residual = np.multiply(I, residual)
    p_derivative = -1.0 * np.dot(residual, q) + sigma / sigma_p * p
    q_derivative = -1.0 * np.dot(residual.T, p) + sigma / sigma_q * q
    return p_derivative, q_derivative
```

1.3: Run the algorithm to reach the threshold

We are runnig the model using a dynamic learning rate. We find out the threshold by running the model.

```
big_lr = 0.001
small_lr = 0.0001
cnt = 0

error = MSE(r_matrix, p_matrix, q_matrix) + L2_loss(sigma, sigma_q, q_matrix) + L2_loss(sigma, sigma_p,

while (error >= 238):
    break ## not to run again

    if error > 300:
        learning_rate = big_lr
    else:
        learning_rate = small_lr

    p_derivative, q_derivative = p_q_derivative (r_matrix, p_matrix, q_matrix, sigma, sigma_p, sigma_q)
    p_matrix = p_matrix - learning_rate * p_derivative
    q_matrix = q_matrix - learning_rate * q_derivative
    error = MSE(r_matrix, p_matrix, q_matrix)
    cnt += 1
    # if cnt % 100 == 0:
    #     print(error)

# pd.DataFrame(p_matrix).to_csv("../output/A2_p_100.csv")
# pd.DataFrame(q_matrix).to_csv("../output/A2_q_100.csv")
```

1.4: Compute the predicted r matrix

Based on p and q which we estimated, we can now calculate the predicted r matrix

```
p = pd.read_csv('../output/A2_p_100.csv')
q = pd.read_csv('../output/A2_q_100.csv')

r = np.dot(p.values[:,1:],q.values[:,1:].T)

# pd.DataFrame(r).to_csv("../output/A2_r_100.csv")
```

Part 2: Implementing Alternating Least Squares

```
data <- read.csv("../data/ml-latest-small/ratings.csv")
data_train <- read.csv("../output/train_set.csv")
data_test <- read.csv("../output/test_set.csv")

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

2.1: Initialize matrix and define function for Alternating Least Squares

```
RMSE <- function(rating, est_rating){
  sqr_err <- function(obs){
    sqr_error <- (obs[3] - est_rating[as.character(obs[1]), as.character(obs[2])])^2
    return(sqr_error)
  }
  return(sqrt(mean(apply(rating, 1, sqr_err))))
}

ALS <- function(f = 10, lambda = 5, max.iter=20, data, train=data_train, test=data_test) {

  # Step 1: Initialize Movie Matrix and User Matrix
  Movie <- matrix(runif(f*I, -1, 1), ncol = I)
  colnames(Movie) <- levels(as.factor(data$movieId))
  movie.average <- data %>% group_by(movieId) %>% summarize(ave=mean(rating))
  Movie[1,] <- movie.average$ave

  User <- matrix(runif(f*U, -1, 1), ncol = U)
  colnames(User) <- levels(as.factor(data$userId))

  movie.id <- sort(unique(data$movieId))
  train_RMSE <- c()
  test_RMSE <- c()

  for (l in 1:max.iter){

  # Step 2: Fix M, Solve U
  for (u in 1:U) {

    User[,u] <- solve (Movie[,as.character(train[train$userId==u,]$movieId)] %*%
```

```
      t(Movie[,as.character(train[train$userId==u,]$movieId)]) + lambda * diag(f)) %*%
      Movie[,as.character(train[train$userId==u,]$movieId)] %*% train[train$userId==u,]$rating}

  # Step 3: Fix U, Solve M
  for (i in 1:I) {
    Movie[,i] <- solve (User[,train[train$movieId==movie.id[i],]$userId] %*%
      t(User[,train[train$movieId==movie.id[i],]$userId]) + lambda * diag(f)) %*%
      User[,train[train$movieId==movie.id[i],]$userId] %*% train[train$movieId==movie.id[i],]$rating
  }

    # Summarize
    cat("iter:", l, "\t")
    est_rating <- t(User) %*% Movie
    colnames(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)

  }
  ratings <- t(as.matrix(User))%*%as.matrix(Movie)
  return(list(p = User, q = Movie, r= ratings, train_RMSE = train_RMSE, test_RMSE = test_RMSE))
}
```

2.2: Get the r and q matrix for 100 latent factors

```
#the r matrix and q matrix for factor of 100, lambda of 5 and RMSE
als3= ALS(f = 100, lambda = 5, max.iter=10, data, train=data_train, test=data_test) # f can be changed
```

```
## iter: 1  training RMSE: 1.19625  test RMSE: 1.499583
## iter: 2  training RMSE: 0.5495955    test RMSE: 1.752284
## iter: 3  training RMSE: 0.4508212    test RMSE: 1.653252
## iter: 4  training RMSE: 0.4134006    test RMSE: 1.557571
## iter: 5  training RMSE: 0.3926967    test RMSE: 1.478213
## iter: 6  training RMSE: 0.3796243    test RMSE: 1.416365
## iter: 7  training RMSE: 0.3708124    test RMSE: 1.369893
## iter: 8  training RMSE: 0.3645669    test RMSE: 1.3356
## iter: 9  training RMSE: 0.3599595    test RMSE: 1.310444
## iter: 10     training RMSE: 0.3564575    test RMSE: 1.291959
```

```
mat5= als3$q
mat6=t(as.matrix(als3$p))%*%as.matrix(als3$q)
write.csv(mat5, file = "../output/A3_q_100.csv")
write.csv(mat6, file = "../output/A3_r_100.csv")
```

Part 3: Post-processing and Evaluation for Alternating Least Squares

As anticipated eariler, the post-processing is done here exemplarily for ALS with 100 latent factors. The process for other levels of latent factors as well as for the GD with Probabilistic Assumptions is similar. There are only few marginal differences:

- Use the respective q matrix produced by the original algorithms
- As the q-matrices produced by the two algorithms are slighlty different, the initial data manipulation is updated accordingly
- The optimal ranges across which to test the tuning parameter lambda are different depending on the amount of factors included

```
#read output and data
train_set<-read.csv("../output/train_set.csv")
test_set<-read.csv("../output/test_set.csv")
q<-read.csv("../output/A3_q_100.csv",header= FALSE)
rating.a3<-read.csv("../output/A3_r_100.csv",header=FALSE)
```

3.1: Prepare for kernel ridge regression input

We need to transform the results from ALS to the form that we can put into kernel ridge regression. First, in order to implement krr on each user separately, we need to the split rating data for the 610 users. Second, each column of the ALS-generated q matrix represents a movie. We should extract certain column of q matrix corresponding to the movie(movieid) users rating and then combine them to build 610 different transformed new q matrices. And then, nomarlize each column of new q matrix to get X (which will be our predictor in kernel ridge regression.

```
#data transformation and get input of krr for A3
train_split<-split(train_set,train_set$userId)
movie<-as.vector(unlist(c(q[1,])))
q<-as.matrix(q[-1,])
```

```
# Data transformation for A2 is just slightly different since the relative q-matrix is transposed

#train_split<-split(train_set,train_set$userId)
#train_split1<-split(train_set,train_set$movieId)
#length(train_split1)
#movie<-as.vector(unlist(c(q[1,])))
#q<-as.matrix(q[-1,])
```

```
new_q_split<-list()
for (k in 1:length(train_split)){
  new<-c()
for (i in 1:dim(train_split[[k]])[1]){
 new<-cbind(new,q[,which(movie==train_split[[k]]$movieId[i])])}
 new_q_split[[k]]<-new}

normal<-function(a){return(a/sqrt(sum(a^2)))}

#if the sum of squares of a vector is 0, the normalized vector should be 0.
q_trans<-apply(q,2,normal)
q_trans[which(is.na(q_trans))]<-0

x_split<-list()
```

```r
for (k in 1:length(train_split)){
  x_split[[k]]<-apply(new_q_split[[k]],2,normal)}

data_split<-list()
for (k in 1:length(train_split)){
  data_split[[k]]<-cbind(train_split[[k]]$rating,t(x_split[[k]]))}

#save(data_split,file = "../output/data_split1.RData")
```

3.2: Tuning parameter for kernel ridge regression

We set the kernel as Gaussian (in line with the paper) and use cross validation to tune lambda so to minimise RMSE.

```r
#write a function to do cross validation and tune parameter lambda in krr
cv.krr <- function(data, kfold, lam){
  set.seed(123)
  data.x <- as.matrix(data[,-1])
  data.y <- data[,1]
  n <- nrow(data.x)
  cv.id <- createFolds(1:n, k = kfold)
  cv.tuning <- c()
  for (j in cv.id){
    #Split Data in train and validation sets
    x.train.cv <- data.x[-j,]
    y.train.cv <- data.y[-j]
    x.validation.cv <- data.x[j,]
    y.validation.cv <- data.y[j]
    #Run Model
    mod.cv <- krr(x = x.train.cv,  y.train.cv, lambda = lam)
    #Estimate predictin of validation test
    pred.cv <- predict(mod.cv, x.validation.cv)
    #Calculate RMSE
    rmse.cv <- sqrt(mean((y.validation.cv - pred.cv)^2))
    cv.tuning <- cbind(cv.tuning, rmse.cv)
    cv.mean <- mean(cv.tuning)
    }
  return(cv.mean)
}
```

```r
#find best lambda

# The following lambdas should be compared when optimizing for 100 factors
# However, when optimising for 50 and 10 factors, c(0.55, 0.6, 0.65, and 0.7) are more appropriate

lambdas <- c(0.50, 0.55, 0.60)

rmse_tune <- data.frame(lambdas=lambdas,rmse=rep(0,length(lambdas)))
for (i in 1:length(lambdas)){
  m <- lapply(data_split, cv.krr, 5, lambdas[i])
  rmse_tune[i,2] <-  sum(unlist(m))
}
```

```
best_lambda <- rmse_tune %>%
  filter(rmse == min(rmse))
best_lambda <- best_lambda$lambda
```

3.3: Train kernel ridge regression and get prediction

0.5 is the best λ for this model. Therefore, we use $\lambda = 0.5$ to train 610 kernel ridge regression models. And then get a prediction matrix with dimension of 610*9724 for krr.

```
#use tuned lambda to train 610 users model
train_model<-vector(mode="list",length=length(data_split))
for(i in 1:length(data_split)){
  train_model[[i]]<-krr(data_split[[i]][,-1],data_split[[i]][,1], best_lambda)}
#get prediction in a matrix with dimension 610*9724
pred_rating<-matrix(0,nrow=length(data_split),ncol=dim(q)[2])
for (i in 1:length(data_split)){
  pred_rating[i,]<-predict(train_model[[i]],t(q_trans))}

#save(pred_rating,file = "../output/pred_rating4.RData")
```

```
rating.a3<-rating.a3[-1,]
colnames(rating.a3)<-c(as.character(movie))
rownames(rating.a3)<-c(1:610)
colnames(pred_rating)<-c(as.character(movie))
rownames(pred_rating)<-c(1:610)
```

```
#function to cacluate mse
mea <- function(data,test){
  movies<-data$movieId
  users<-data$userId
  pred<-as.numeric(t(test[match(c(as.character(users)),rownames(test)),match(c(as.character(movies)),co
  return(mean((data$rating-pred)^2))
}
```

3.4: Compute weighted average of algrithms predicition and krr prediction

We combined the predictions together and see if using weighted average will help further minimize RMSE. And we used cross validation to get the best weight.

For factor numbers of 10, we find the best weight is 0.7. For 50 and 100 factors, the best weight is 1, which means krr is a substantive improvement of ALS.

```
#get the best weight
weights <- seq(0,1,0.1)
rmse_train <- data.frame(weights=weights,rmse=rep(0,length(weights)))
rating.weighted<-list()

for (i in 1:length(weights)){
  rating.weighted[[i]]<- rating.a3*(1-weights[i]) + pred_rating*weights[i]
  rating.weighted[[i]]<-as.matrix(rating.weighted[[i]])
  mean1<-mea(train_set[1:10000,],rating.weighted[[i]])
  mean2<-mea(train_set[10001:20000,],rating.weighted[[i]])
  mean3<-mea(train_set[20001:30000,],rating.weighted[[i]])
  mean4<-mea(train_set[30001:40000,],rating.weighted[[i]])
```

```
  mean5<-mea(train_set[40001:50000,],rating.weighted[[i]])
  mean6<-mea(train_set[50001:60000,],rating.weighted[[i]])
  mean7<-mea(train_set[60001:70000,],rating.weighted[[i]])
  mean8<-mea(train_set[70001:80000,],rating.weighted[[i]])
  mean9<-mea(train_set[80001:dim(train_set)[1],],rating.weighted[[i]])
  rmse_train[i,2]<-sqrt(((mean1+mean2+mean3+mean4+mean5+mean6+mean7+mean8)*10000+(dim(train_set)[1]-800
}

rmse_train
```

```
##    weights     rmse
## 1      0.0 1.396845
## 2      0.1 1.368874
## 3      0.2 1.343172
## 4      0.3 1.319871
## 5      0.4 1.299101
## 6      0.5 1.280985
## 7      0.6 1.265636
## 8      0.7 1.253156
## 9      0.8 1.243632
## 10     0.9 1.237133
## 11     1.0 1.233705
```

Part 4: Evalutation

4.1 Evaluation

We used the tuned weight to evaluate on test data and get the final RMSE for the whole A3 algorithm with krr as post processing method.

```
#get test rmse
best_weight <- match(min(rmse_train$rmse), rmse_train$rmse)
mean11<-mea(test_set[1:10000,],rating.weighted[[best_weight]])
mean21<-mea(test_set[10001:20000,],rating.weighted[[best_weight]])
mean32<-mea(test_set[20001:dim(test_set)[1],],rating.weighted[[best_weight]])
rmse_test<-sqrt(((mean11+mean21)*10000+(dim(test_set)[1]-20000)*mean32)/dim(test_set)[1])

rmse_test
```

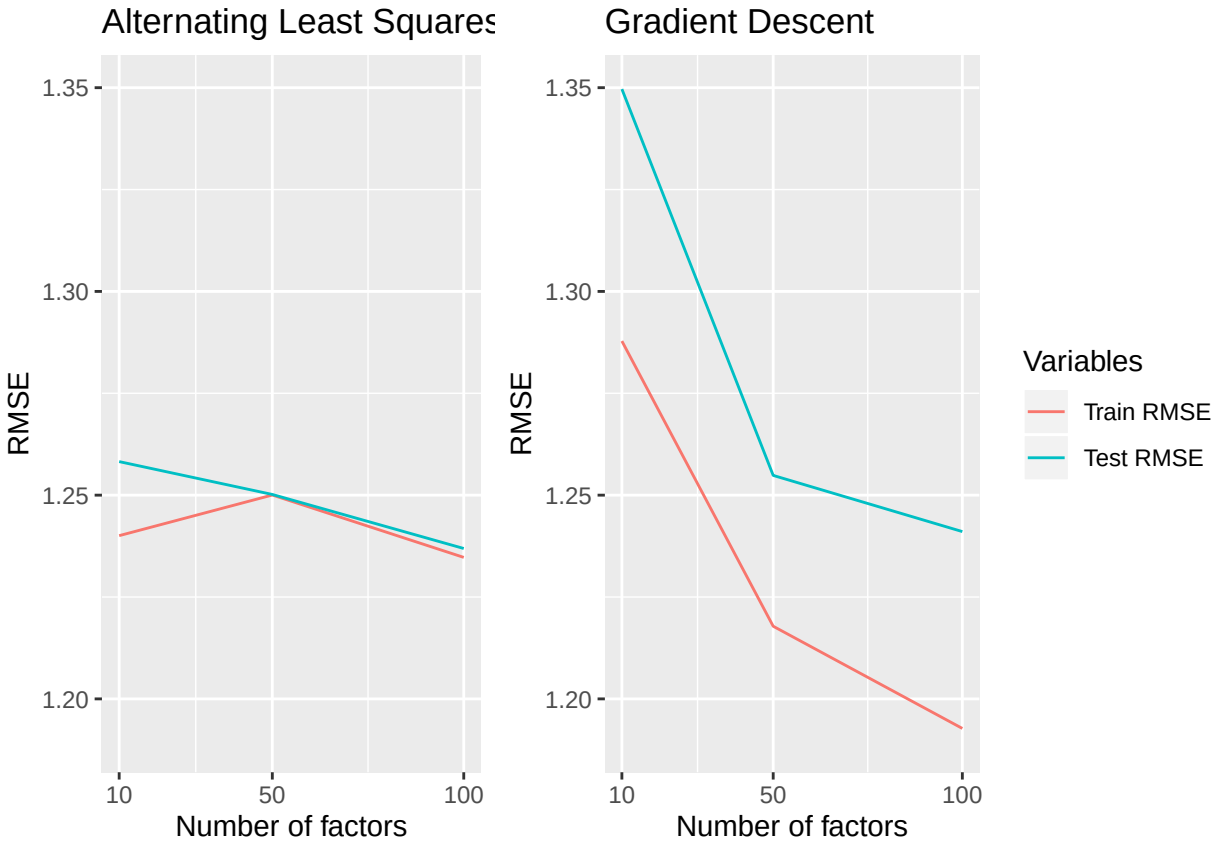
```
## [1] 1.236222
```

4.2 Visualization

After running the above code for both algorithms and for all three factor sizes (10, 50, 100), we have saved the resulting RMSEs in the files loaded below. This allows us to graphically visualise our results in the following way

```
load("../output/als_train_rmse.RData")
load("../output/als_test_rmse.RData")
load("../output/gs_train_rmse.RData")
load("../output/gs_test_rmse.RData")

factors <- c(10, 50, 100)
```

```r
als <- melt(as.data.frame(cbind(factors, als_train_rmse, als_test_rmse)), id="factors")
gs <- melt(as.data.frame(cbind(factors, gs_train_rmse, gs_test_rmse)), id="factors")

p1 <- ggplot(als, aes(x = factors, y = value, colour=variable)) +
  geom_line() +
  ggtitle("Alternating Least Squares") +
  theme(legend.position = "none") +
  ylim(1.19, 1.35) +
  ylab("RMSE") +
  scale_x_continuous("Number of factors", breaks=c(10, 50, 100))

p2 <- ggplot(gs, aes(x = factors, y = value, colour=variable)) +
  geom_line() +
  ggtitle("Gradient Descent") +
  ylim(1.19, 1.35) +
  ylab("RMSE") +
  scale_x_continuous("Number of factors", breaks=c(10, 50, 100)) +
  scale_colour_discrete(name = "Variables", labels = c("Train RMSE", "Test RMSE"))

grid.arrange(p1, p2, nrow = 1, widths = c(2.7,4))
```



We can see from the plot that for A2 as the number of factors increases, RMSE goes down for both train data and test data. But for A3, it goes up and then goes down. This maybe because the train data we split does not include all the movies. 0 will appear in q matrix and will give 0 ratings. A3 is still better. However, it is possible that as we increase the number of factor for A2 and A3, according to the decreasing trend of RMSE for A2, we may get a lower RMSE than A3.