Project 4 Report

Xudong Guo, Sixing Hao, Chang Qu, Samuel Unger, Chen Wang

In this project, we applied the following methods for collaborative filtering:

- factorization algorithm: Stochastic Gradient Descent
- regularization: Temporal Dynamics
- postpocessing: KNN & Kernel Ridge Regression

Our objective is to compare the postporcessing process, and evaluate their results.

Step 1 Load Data and Train-test Split

```
library(dplyr)
library(tidyr)
library(ggplot2)
library(anytime)
library(ggplot2)
data <- read.csv("../data/ml-latest-small/ratings.csv")
set.seed(0)

# Function for dividing time into bins, and add bins to the original data

bins <- function(nbin, data=data){
  data$time <- anytime(data$timestamp) %>% format("%Y/%m/%d")
  time <- unique(data$time) %>% sort()
  bin_size <- round(length(time)/nbin)
  bin_label <- rep(1:nbin, each=bin_size)
  bin_label <- bin_label[1:length(time)]
  time_label <- data.frame(time, bin_label)
  time_label[is.na(time_label)] <- nbin
  bin <- merge(data, time_label, by="time") %>% select(-time)
  return(bin)
}

# Bin size is suggested by Koren, author of the paper on Temporal Dynamics

data_bins <- bins(30, data=data)
test_idx <- sample(1:nrow(data), round(nrow(data)/5, 0))
train_idx <- setdiff(1:nrow(data), test_idx)
data_train <- data_bins[train_idx,]
data_test <- data_bins[test_idx,]
```

Step 2 Matrix Factorization

Step 2.1 Algorithm and Regularization

- Algorithm: Stochastic Gradient Descent (A1)
- Regularizations: Temporal Dynamics (R3)

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

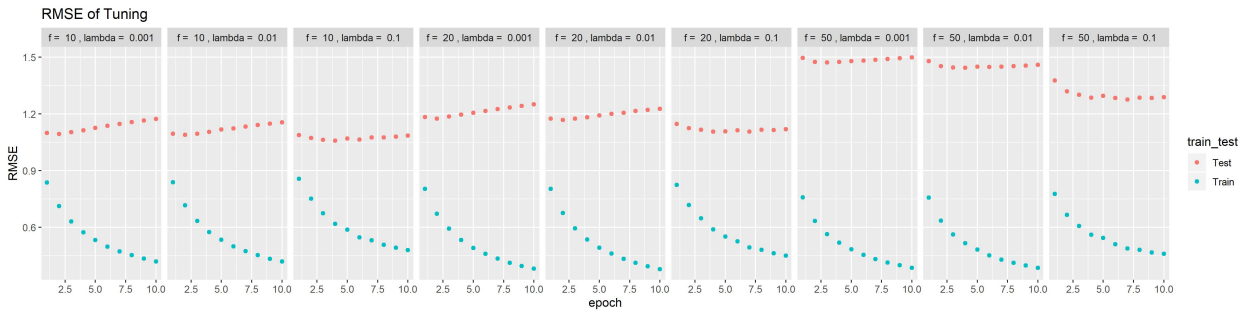
Step 2.2 Parameter Tuning

We used cross validation to tune the 2 hyperparameters: f and lambda, and select the optimal combination by choosing the one with lowest test RMSE

```
source("../lib/cross_validation.R")
f_list <- c(10, 20, 50)
l_list <- seq(-3,-1,1)
f_l <- expand.grid(f_list, l_list)
```

```
result_summary <- array(NA, dim = c(4, 10, nrow(f_l)))
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_bins, K = 3, 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.Rdata")
```

```
load("../output/rmse.Rdata")

rmse <- data.frame(rbind(t(result_summary[1,,]), t(result_summary[2,,])),
                   train_test = rep(c("Train", "Test"), each = 9),
                   par = rep(paste("f = ", f_l[,1], ", lambda = ", 10^f_l[,2]), times = 2)) %>%
  gather("epoch", "RMSE", -train_test, -par)

rmse$epoch <- as.numeric(gsub("X", "", rmse$epoch))

cv.result <- ggplot(rmse, aes(x = epoch, y = RMSE, col = train_test)) + geom_point() +
  facet_grid(~par)+ggtitle("RMSE of Tuning")
ggsave(plot=cv.result, filename = "cv_result.jpg",height = 100, width = 400, units="mm")
```

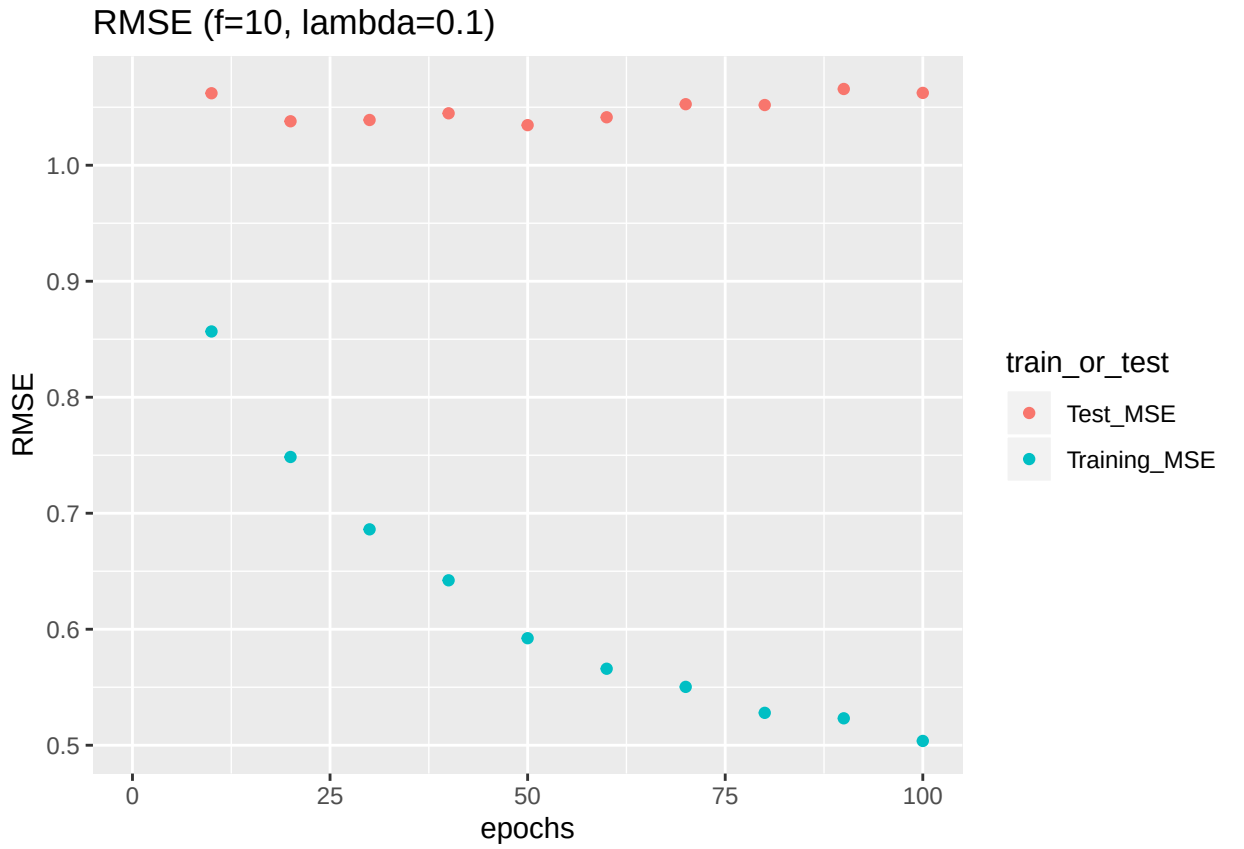


[CV]

Here is the train and test RMSE for selected set of hyper parameters

```
load(file = "../output/mat_fac.RData")

RMSE <- data.frame(epochs = seq(10, 100, 10),
                   Training_MSE = result$train_RMSE,
```

```
                  Test_MSE = result$test_RMSE) %>%
  gather(key = train_or_test, value = RMSE, -epochs)

ggplot(RMSE, aes(x = epochs, y = RMSE,col = train_or_test)) +
  geom_point() + scale_x_discrete(limits = seq(10, 100, 10)) +
  xlim(c(0, 100)) +
  ggtitle("RMSE (f=10, lambda=0.1)")
```



Step 3 Postprocessing

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

P2:Postprocessing SVD with KNN Section 3.5

P3:Postprocessing SVD with kernel ridge regression Section 3.6

```
result <- gradesc(f = 10, lambda = 0.1,lrate = 0.01, max.iter = 100, stopping.deriv = 0.01,
                  data = data_bins, train = data_train, test = data_test)

save(result, file = "../output/mat_fac.RData")
write.csv(result$p, file = "../output/p.csv")
write.csv(result$q, file = "../output/q.csv")
write.csv(result$b_user, file = "../output/b_user.csv")
write.csv(result$b_movie, file = "../output/b_movie.csv")
write.csv(result$b_bin, file = "../output/b_bin.csv")
write.csv(result$mu, file = "../output/mu.csv")
```

Postprocessing SVD with KNN

```
# import neccessary library

from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LinearRegression
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from itertools import chain
import math
import time
```

```
# import dataset
q = pd.read_csv('../output/q.csv')
q.drop(['Unnamed: 0'], axis = 1, inplace=True)

# convert dataframe into a matrix
q_mat = q.to_numpy()
q_mat = np.transpose(q_mat)

# calculate mean rating for each movie
rating = pd.read_csv('../data/ml-latest-small/ratings.csv')
rating = rating[['movieId','rating']]
y = rating.groupby(['movieId']).mean()
```

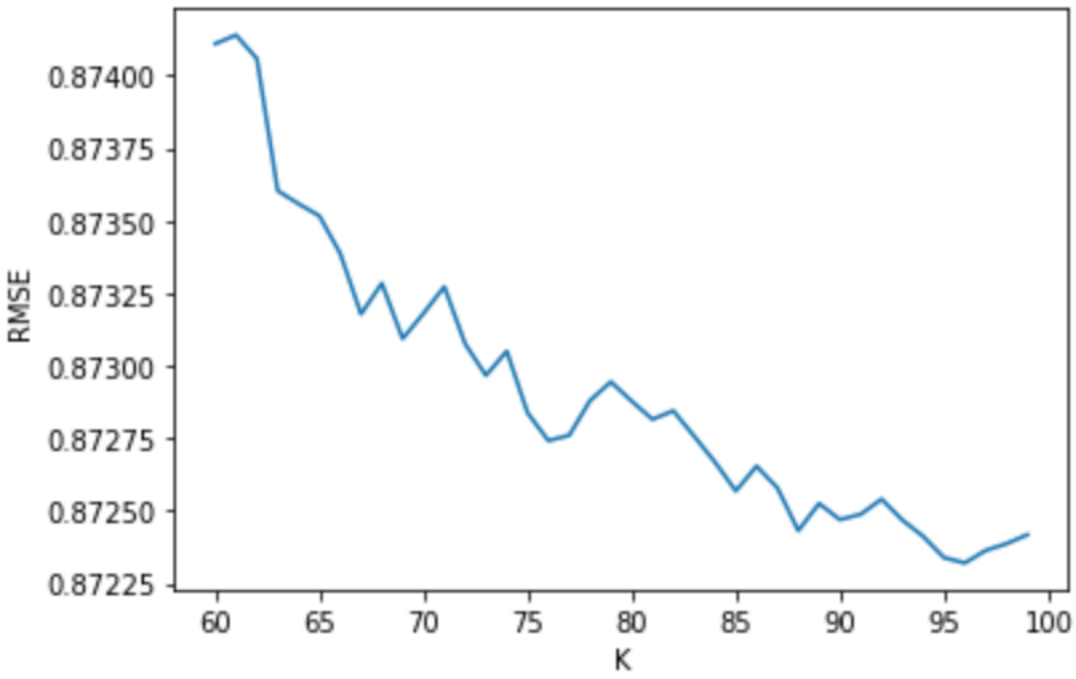
Next, we will use Grid search cross validation to choose the optimal K

```
# create a KNN regression model
knn_model = KNeighborsRegressor(metric = 'cosine', algorithm='brute')

# define parameters
param_grid = {'n_neighbors': np.arange(60, 100)}

# create grid search CV
knn_gscv = GridSearchCV(knn_model, param_grid, cv=5, scoring = 'neg_mean_squared_error')

# fit to our data
knn_gscv.fit(q_mat, y)

# the best knn parameter
knn_gscv.best_params_

# RMSE of our KNN model
math.sqrt(-1*knn_gscv.best_score_)
```

After searching, we found that when K=96, it will give us the smallest RMSE, which is 0.872. We want to visualize the result of cross validation.

```
# get RMSE for each K
all_rmse = np.sqrt(-1*knn_gscv.cv_results_['mean_test_score'])
k = range(60,100)

# visualize RMSE w.r.t K
plt.plot(k, all_rmse)
```

```
plt.xlabel('K')
plt.ylabel('RMSE')
```



We see that RMSE is the smallest when $K = 96$.

Now we want to combine all the component we had so far, including $p_i^T q_i$, b_i, b_u, $b_{i,Bin(t)}$, and KNN, to construct a linear regression function to make predictions for the ratings.

```
# import data
p = pd.read_csv('../output/p.csv')
b_user = pd.read_csv('../output/b_user.csv')
b_movie = pd.read_csv('../output/b_movie.csv')
mu = pd.read_csv('../output/mu.csv')
b_bin = pd.read_csv('../output/b_bin.csv')

# clean data
p.drop(['Unnamed: 0'], axis = 1, inplace=True)
b_user.drop(['Unnamed: 0'], axis = 1, inplace=True)
b_movie.drop(['Unnamed: 0'], axis = 1, inplace=True)
b_bin.drop(['Unnamed: 0'], axis = 1, inplace=True)

# convert into matrix
p_mat = p.to_numpy()
b_user_mat = b_user.to_numpy()
b_movie_mat = b_movie.to_numpy()
b_bin_mat = b_bin.to_numpy()

# comput interaction matrix
interaction = np.matmul(q_mat, p_mat)
```

In our case, our data has three dimension, which are 9724 movies, 610 users, and 30 bins, we want to flatten each component into a $(9724 * 610 * 30) * 1$ column vector in order to constrcut the linear regression function.

```
# flatten interaction matrix
inter_flat = []

for col in range(interaction.shape[1]):
    inter_flat.append(interaction[:,col].tolist())

li = list(chain.from_iterable(inter_flat))

# repeat for 30 times
inter_flat_time = []
for i in range(30):
    for ele in li:
        inter_flat_time.append(ele)
```

```
# repear b_user for every movie
b_user_flat = []
for ele in b_user_mat.tolist():
    for i in range(9724):
        b_user_flat.append(ele)

b_user_flat = list(chain.from_iterable(b_user_flat))

# repeat for 30 times
b_user_flat_time = []
for i in range(30):
    for ele in b_user_flat:
        b_user_flat_time.append(ele)
```

```
# repear b_movie for every user
b_movie_flat = []
for i in range(610):
    b_movie_flat.append(b_movie_mat[:,0].tolist())

b_movie_flat = list(chain.from_iterable(b_movie_flat))

# repeat for 30 times
b_movie_flat_time = []
for i in range(30):
    for ele in b_movie_flat:
        b_movie_flat_time.append(ele)
```

```
# repear KNN for every user
rating_pred_flat = []
for i in range(610):
    rating_pred_flat.append(rating_pred[:,0].tolist())

rating_pred_flat = list(chain.from_iterable(rating_pred_flat))

# repeat for 30 times
rating_pred_flat_time = []
for i in range(30):
    for ele in rating_pred_flat:
        rating_pred_flat_time.append(ele)
```

```
# flatten b_bin for 30 bins
b_bin_mat = np.transpose(b_bin_mat)
b_bin_flat = []

for col in range(30):
    for i in range(610):
        b_bin_flat.append(b_bin_mat[:,col].tolist())

b_bin_flat = list(chain.from_iterable(b_bin_flat))
```

```
# calcualte y vec for linear regression
y_mat = y.to_numpy()
y_flat = []
for i in range(610):
    y_flat.append(y_mat[:,0].tolist())

y_flat = list(chain.from_iterable(y_flat))

# repeat for 30 times
y_flat_time = []
for i in range(30):
    for ele in y_flat:
        y_flat_time.append(ele)
```

```
# combine all component into X matrix
inter_df = pd.DataFrame(inter_flat_time, columns = ['Interaction'])
b_user_df =  pd.DataFrame(b_user_flat_time, columns = ['b_user'])
b_movie_df =  pd.DataFrame(b_movie_flat_time, columns = ['b_movie'])
b_bin_df =  pd.DataFrame(b_bin_flat, columns = ['b_bin'])
knn_df =  pd.DataFrame(rating_pred_flat_time, columns = ['KNN'])

X = pd.concat([inter_df, b_user_df, b_movie_df, b_bin_df, knn_df], axis = 1, sort=False)
```

Now we will split the data into testing and trainning data, with 20 of the data being testing data.

```
# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y_flat_time, test_size=0.2, random_state=42)
```

```
# fit linear regression
start = time.time()
reg = LinearRegression().fit(X_train,y_train)
end = time.time()

# calculate training time
model_train_tm = end-start
print('model training took: {}'.format(model_train_tm))
```

The training job takes around 2420.9919040203094 seconds.

```
# get parameters of the linear regression
reg.coef_

# make predictions
y_pred = reg.predict(X_test)
```

Our estimated linear regression function is $r_{ij} = -0.000871 * p_i^T q_j - 0.000117 * b_u + 0.902 * b_i + 0.0205 *$

$b_{i,Bin(t)} + 0.717 * KNN$

```
# compute MSE
mse = (((y_pred - np.array(y_test))**2).sum()) / len(y_test)

# compute RMSE
RMSE = math.sqrt(mse)
RMSE
```

In the end, we got $RMSE$ of 0.701, which is a huge improvement from the one without the post-processing.

Postprocessing SVD with Kernel Ridge Regression

```
# import neccessary library
import time as time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from google.colab import drive
from sklearn.model_selection import GridSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.preprocessing import normalize
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
```

```
p = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/output/
q = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/output/
rating = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/da
b_bin = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/ou
mu = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/output
b_movie = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/
b_user = pd.read_csv('https://raw.githubusercontent.com/TZstatsADS/fall2019-project4-sec2-grp4/master/ou
q.drop(['Unnamed: 0'], axis = 1, inplace = True)
q.index = range(1, 11)
rating.index = range(1,100837)
#import dataset
```

```
q_transpose = q.T
q_transpose_normalized = q_transpose / np.linalg.norm(q_transpose, axis = 1, keepdims = True)
#transpose and normalize q matrix
q_transpose_normalized.reset_index(inplace = True)
#preparing for later-on dataset merge
q_transpose_normalized.rename(columns = {"index": "movieId"}, inplace = True)
#rename from "index" to "movieId"
```

In this step, we normalize each q_i, that is $X_i = \frac{q_i}{||q_i||}$.

```
rating['movieId'] = rating['movieId'].astype(int)
#change movieId into integer
q_transpose_normalized['movieId'] = q_transpose_normalized['movieId'].astype(int)
#change movieId into integer
```

```
rating_merge = pd.merge(rating, q_transpose_normalized, on = 'movieId', how = 'left')
#merge 'rating' and 'q_transpose_normalized' datasets
dict_of_users = {k: v for k, v in rating_merge.groupby('userId')}
#separating all users into 610 cat
dict_of_users.get(1)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
```

```
#checking if user_1 is in the dictionary
```

By using dictionary, we are easier to access each userId.

```
def RMSE(n_users, method):
  vecList = list()
  for i in range(1, n_users + 1):
    X = dict_of_users.get(i)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y = dict_of_users.get(i)[['rating']]
    pred = KernelRidge(kernel = method).fit(X, y).predict(X)
    sq_diff = np.square(np.array(pred) - np.array(y))
    mean_sq_diff = sq_diff.mean()
    root_mean_sq_diff = np.sqrt(mean_sq_diff)
    vecList.append(root_mean_sq_diff)
  return vecList
#for iteration return RMSE
```

```
def List_Sum_Function(list, length):
   if (length == 0):
     return 0
   else:
     return list[length - 1] + List_Sum_Function(list, length - 1)
#define function of sum of list values
```

```
## Train kernel ridge ##
train, test = train_test_split(rating_merge, test_size = 0.2)
#spliting dataset 20% test 80% train
dict_of_users_train = {k: v for k, v in train.groupby('userId')}
#separating all users into 610 cat
dict_of_users_test = {k: v for k, v in test.groupby('userId')}
#separating all users into 610 cat

dict_of_users_train.get(1)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
#checking if user_1 is in the train dictionary
dict_of_users_test.get(1)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
#checking if user_1 is in the test dictionary
```

We split the dataset with 80% training and 20% testing.

```
def RMSE_train(n_users, kernel, alpha, gamma, degree):
  vector_train = list()
  for m in range(1, n_users + 1):
    X_train = dict_of_users_train.get(m)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y_train = dict_of_users_train.get(m)[['rating']]
    pred = KernelRidge(alpha = alpha, gamma = gamma, degree = degree, kernel = kernel).fit(X_train, y_t
    sq_diff = np.square(np.array(pred) - np.array(y_train))
    mean_sq_diff = sq_diff.mean()
    root_mean_sq_diff = np.sqrt(mean_sq_diff)
    vector_train.append(root_mean_sq_diff)
  return vector_train
#define RMSE train function

def RMSE_test(n_users, kernel, alpha, gamma, degree):
  vector_test = list()
  for m in range(1, n_users + 1):
```

```
    X_train = dict_of_users_train.get(m)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y_train = dict_of_users_train.get(m)[['rating']]
    X_test = dict_of_users.get(m)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y_test = dict_of_users.get(m)[['rating']]
    pred = KernelRidge(alpha = alpha, gamma = gamma, degree = degree, kernel = kernel).fit(X_train, y_t
    sq_diff = np.square(np.array(pred) - np.array(y_test))
    mean_sq_diff = sq_diff.mean()
    root_mean_sq_diff = np.sqrt(mean_sq_diff)
    vector_test.append(root_mean_sq_diff)
  return vector_test
#define RMSE test function
```

```
## Kernel Ridge Tuning Parameters ##
KernelList = ["rbf", "polynomial", "laplacian", "linear", "sigmoid"]
GammaList = [0.00001, 0.0001, 0.01]
AlphaList = [0.00001, 0.0001, 0.01]

vecList2_train = list()
vecList2_test = list()
for i in KernelList:
  for j in GammaList:
    for k in AlphaList:
        RMSE_total_train = List_Sum_Function(RMSE_train(610, kernel = i, alpha = k, gamma = j, degree =
        RMSE_total_test = List_Sum_Function(RMSE_test(610, kernel = i, alpha = k, gamma = j, degree = 3
        vecList2_train.append(RMSE_total_train)
        vecList2_test.append(RMSE_total_test)
#finding out the best parameters for both training and testing datasets
#laplacian mechanism with gamma = 0.01 and alpha = 0.0001 is the optimal parameters for our kernel ridg
```

By comparing performance among “rbf”, “polynomial”, “laplacian”, “linear”, “sigmoid” kernels, alpha ranging from 0.00001 to 0.01 and gamma ranging from 0.00001 to 0.01, we decide to use laplacian mechanism with gamma = 0.01 and alpha = 0.0001. While in general, “linear” kernels have the worst performance and “sigmoid” does not have stable RMSE.

```
## Train Kernel Ridge Final Model ##
def train_model(datasets, n_users, kernel, alpha, gamma, degree):
  vecList_train = list()
  for i in range(1, n_users + 1):
    X = datasets.get(i)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y = datasets.get(i)[['rating']]
    X_overall = q_transpose_normalized[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    pred = KernelRidge(alpha = alpha, gamma = gamma, degree = degree, kernel = kernel).fit(X, y).predic
    vecList_train.append(pred)
  return vecList_train

t0 = time.time()
train_model(dict_of_users_train, 610, kernel = 'laplacian', alpha = 0.0001, gamma = 0.01, degree = 3)
training_time = time.time() - t0
training_time
#training time = 45.278982400894165s
```

The training time is 45.278982400894165s.

```
def RMSE_aggregate(datasets):
  vecList_agg = list()
```

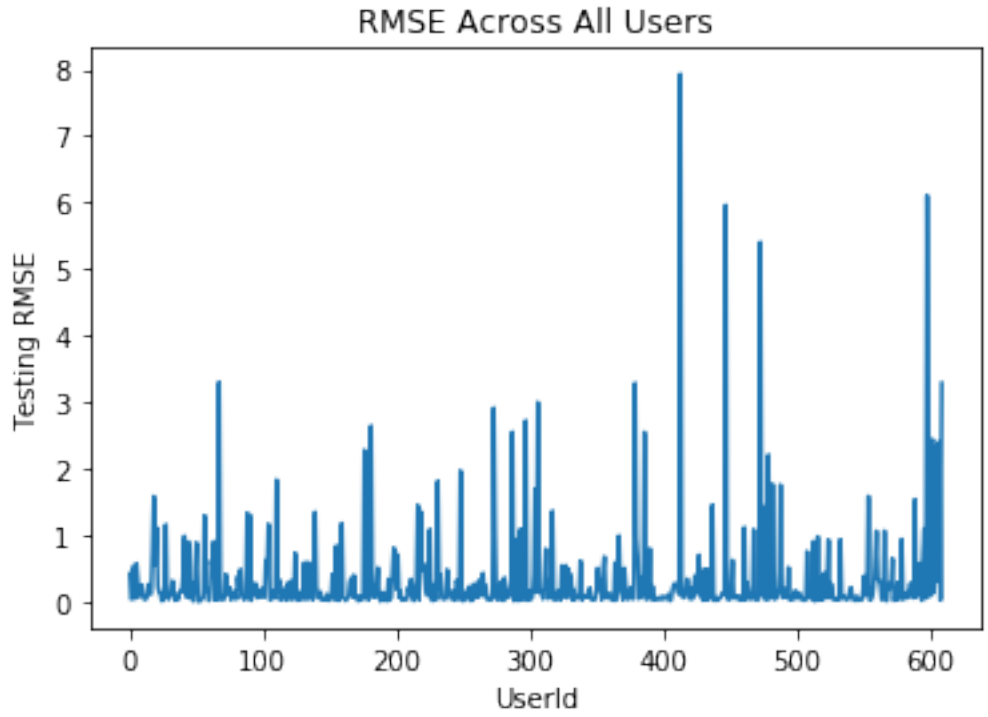
```
  for i in range(1, 610 + 1):
    X = datasets.get(i)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y = datasets.get(i)[['rating']]
    X_overall = dict_of_users.get(i)[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
    y_overall = dict_of_users.get(i)[['rating']]
    pred = KernelRidge(alpha = 0.0001, gamma = 0.01, degree = 3, kernel = 'laplacian').fit(X, y).predic
    sq_diff = np.square(np.array(pred) - np.array(y_overall))
    sq_diff_sum = np.sum(sq_diff)
    vecList_agg.append(sq_diff_sum)
  return vecList_agg

np.sqrt(List_Sum_Function(RMSE_aggregate(dict_of_users), len(RMSE_aggregate(dict_of_users)))/610)
#check the total RMSE of KRR
```

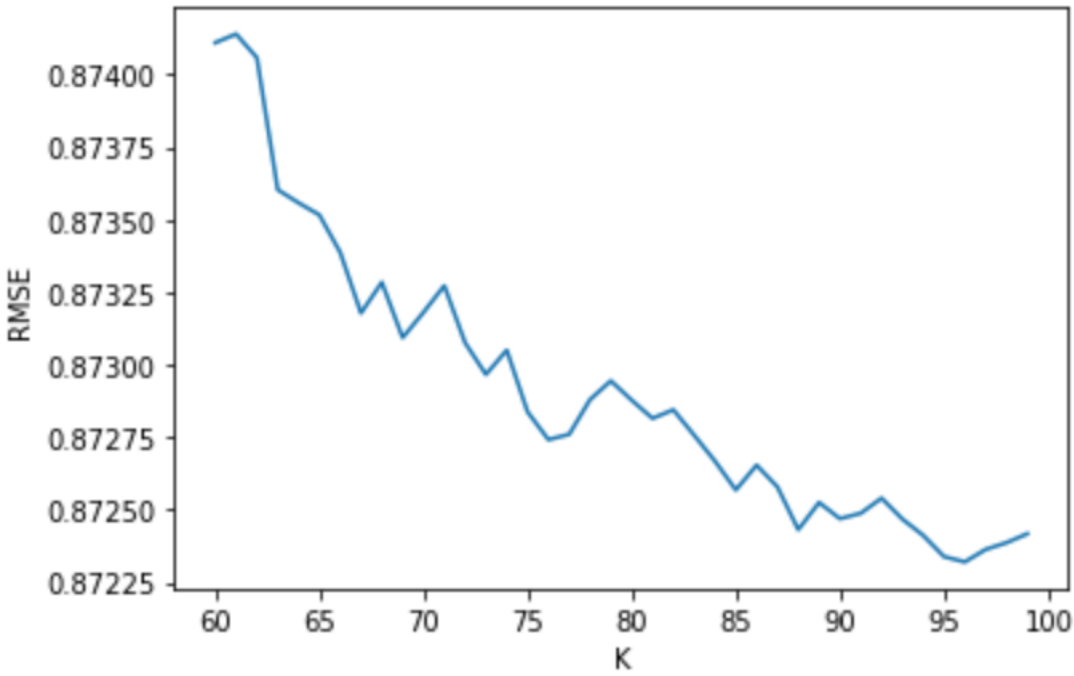
The RMSE of Kernel Ridge Regression is 0.7806703963573395.

```
np.shape(train_model(dict_of_users_train, 610, kernel = 'laplacian', alpha = 0.0001, gamma = 0.01, degr
train_model(dict_of_users_train, 610, kernel = 'laplacian', alpha = 0.0001, gamma = 0.01, degree = 3)
#check dimension of the final recommender system which is 610 * 9724
```

Finally, we output the recommendation system (estimated ratings for each user and each movie) which is a 610 * 9724 dimension matrix.

Step 4 Evaluation





The post-processing KNN decreases the RMSE from 1.062 to 0.7007 which is better than the performance of Kernel Ridge Regression which has RMSE 0.78067.

The training time for KNN when fitting linear regression is around 40 minutes and which for training kernel ridge is 45.279 seconds.

Step 5: Future Work

In the dataset, we noticed that some users do not rate many movies, and some movies do not have enough rating. This might cause some problem, as when we split time into 30 bins, some users and movies will not have enough appearance, thus making our prediction on these observations less accurate. This might further influence our prediction on other observations.

To address this issue, we should subset the original data, to include movies and users that has at least a certain amount of appearances. Considering our target of prediction, which is recommendation based on rating prediction, it's reasonable not to include movies with rare appearance, and users with very few rating records shouldn't be our focus as well. By using the data after subset, we should have a more reasonable prediction.

Also, when sampling train and test data, we also need to make sure that, all bins are being sampled, and so does movies and users. It is inevitable for users and movies which have less record to be evaluated less accurately, but we need to mitigate this issue.