main-Copy1

November 20, 2019

1 ADS Project 4 Group 8

1.0.1 Team members: Zihan Zhou (zz2573), Suzy Gao (zg2333), Justine Zhang (yz3420), Jason Gao (yg2583)

2 1. Introduction

This project implements a collaborative filtering algorithm for recommender systems. It uses **Alternating Least Squares**[1] as matrix factorization method, imposing **magnitudes** and **bias & intercepts** regularization[2], and comparing performances using **K − Nearest − Neighbor** (**KNN**) and **Kernel Ridge Regression** as postprocessing method[3].

3 2. Data Preprocessing

3.1 Required Packages

[1]:
```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')
```

3.2 Original Data

[2]:
```
data = pd.read_csv('../data/ml-latest-small/ratings.csv')
data.head(10)
```

[2]:

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931
5	1	70	3.0	964982400
6	1	101	5.0	964980868
7	1	110	4.0	964982176
8	1	151	5.0	964984041

```
9       1      157     5.0  964984100
```

3.3 Transfer Data into Required Shape

[3]:
```
R = data.pivot_table(values='rating', columns = 'movieId', index='userId')
R.shape
R = R.to_numpy()
```

4 3. Matrix Factorization

After preprocessing the given data, we obtained a rating matrix $R_{610\times 9724}$, $R[i,j]$ = rating of movie j given by user i. We want to factorize $R_{n\times m} = U_{n\times d}M_{d\times m}$, where n = number of users, m = number of movies, d = number of hidden features. Due to severe sparsity of R, SVD method does not work in this case.

Here we implement the **Alternating Least Squares** algorithm to solve for a feasible solution of (U, M). The objective function to be minimized is

$$\sum_{(i,j)\in I}\left(r_{ij} - (b_{ij} + u_i^\top m_j)\right)^2 + \lambda\left(\sum_i n_{u_i}(\|u_i\|^2 + b_i^2) + \sum_j n_{m_j}(\|m_j\|^2 + b_j^2)\right)$$

- n_{u_i}: number of ratings given by user i
- n_{m_j}: number of ratings of movie j
- $b_{ij} = \mu + b_i + b_j$
  - μ: overall average rating
  - b_i: observed deviations of user i from the average
  - b_j: observed deviations of movie j from the average

Let T denote the modified rating matrix, where $t_{ij} = r_{ij} - \mu - b_i - b_j$. The aim is to minimize the difference between R and UM under regularizations. Here we penalize magnitudes to prevent overfitting, and we take bias and intercepts into consideration, since some users give higher ratings than others, and some movies receive higher ratings than others.

The **Alternating Least Squares** algorithm is as follows:

1. Initialize matrix M by assigning the average rating for each movie as the first row, and small random numbers for the remaining entries;
2. Fix M, solve U by minimizing the objective function;
3. Fix U, solve M by minimizing the objective function;
4. Repeat steps 2 and 3 until the improvement on objective function is less than a stopping criterion.

We can show after some mathematical calculation that, the optimal U given M is

$$u_i = A_i^{-1}V_i, \forall i$$

where

$$A_i = M_{I_i}M_{I_i}^\top + \lambda n_{u_i} I, V_i = M_{I_i}T^\top(i, I_i)$$

Here I_i denotes the set of movies that user i rated, M_{I_i} denotes the sub-matrix of M where columns $j \in I_i$ are selected, and $T(i, I_i)$ is the row vector where columns $j \in I_i$ of the i-th row of T is taken.

Similarly, given U, the optimal M is

$$m_j = A_j^{-1} V_j, \forall j$$

where

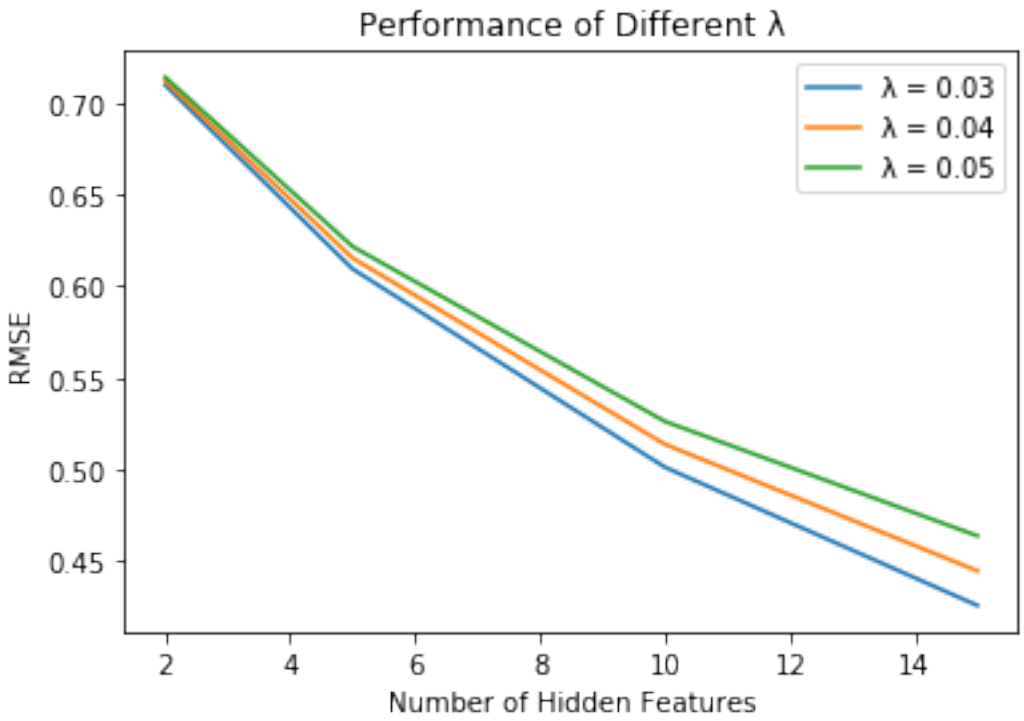
$$A_i = U_{I_j} U_{I_j}^\top + \lambda n_{m_j} I, V_i = U_{I_j} T(I_j, j)$$

Next we implemented the algorithm and evaluated its performance under different d's and λ's.

[4]:
```
%run ../lib/ALS.py
import matplotlib.pyplot as plt
import numpy as np
```

First we choose an appropriate d.

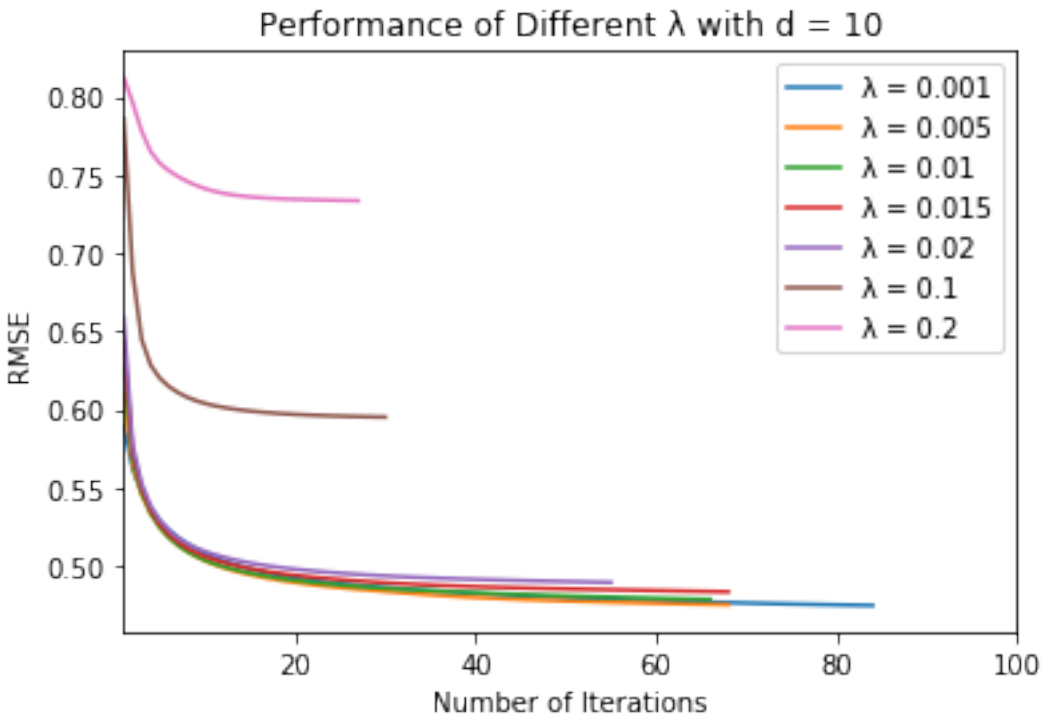
[5]:
```
lambda_list = [0.03, 0.04, 0.05]
d_list = [2, 5, 10, 15]
plt.figure()
for l in lambda_list:
    rmse = []
    label = ' = ' + str(l)
    for d in d_list:
        (U,M), serror = ALS(R, d = d, Lambda = l, seed = 2019)
        rmse.append(serror[-1])
    plt.plot(d_list, rmse, label = label)
plt.xlabel('Number of Hidden Features')
plt.ylabel('RMSE')
plt.title('Performance of Different ')
plt.legend()
plt.show()
```



From the plot we can see that the larger the number of hidden features we choose, the better the performance of factorization is. We choose $d = 10$ as it keeps a balance between good performance and computation cost.

Next we do parameter tuning on λ, i.e. coefficient of the penalty term.

[6]:
```
lambda_list = [0.001, 0.005, 0.01, 0.015, 0.02, 0.1, 0.2]
plt.figure()
plt.xlim(1,100)
for l in lambda_list:
    (U,M), serror = ALS(R, d = 10, Lambda = l, seed = 2019)
    label = ' = ' + str(l)
    plt.plot(list(range(1, len(serror)+1)), serror, label = label)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('RMSE')
plt.title('Performance of Different  with d = 10')
plt.show()
```



We can see from the plot that $\lambda = 0.005$ is an appropriate choice. It achieves a low RMSE while taking less than 70 iteration steps to converge.

Next we do a train-test-split on the data (with 80% as training set), and evaluate different postprocessing methods.

5 4. Train-Test-Split

[7]:
```
%run ../lib/train_test_split.py
```

[8]:
```
train, test = train_test_split(R, train_size = 0.8, seed = 2019)
```

```
###### Size of training set: 0.800032  ######
```

[9]:
```
(U,M), serror = ALS(train, d = 10, Lambda = 0.005, seed = 2019)
```

6 5. Postprocessing

6.1 5.1. Baseline

First we take a look at the baseline method, which is to use $U \times M$ as predictions (of the modified rating matrix) directly.

[10]:
```
%run ../lib/modify.py
```

[11]:
```
Rhat = np.matmul(U,M)
R0 = modify(R)
train0, test0 = train_test_split(R0, 0.8, 2019)

print(RMSE(train0, Rhat))
print(RMSE(test0, Rhat))
```

```
###### Size of training set: 0.800032  ######
0.44436816964341996
1.1690024326349446
```

RMSE achieved on testing set is 1.169. Next we try to improve the predicting accuracy by postprocessing methods.

6.2 5.2. K-Nearest-Neighbor (KNN)

The first method is KNN, which is to predict user i's rating on movie j, based on the K nearest movies that user i has rated, to movie j. The distance defined here is cosine similarity using each movie's feature vector obtained above (i.e. each column of M):

$$s(v_i, v_j) = \frac{v_i^\top v_j}{\|v_i\|\|v_j\|}, -1 \leq s(v_i, v_j) \leq 1$$

We select K movies that have the highest similarities with the objective movie.

[12]:
```
%run ../lib/knn.py
```

[13]:
```
train0 = nan_to_zero(train0)
test0 = nan_to_zero(test0)
n = train0.shape[0]

# store all the similarity calculations in an array ss to reduce complexity
print("######### Start Calculating the similarity between movies #########")
ss = []
for i in range(n):
    #print(i+1,"/",n)
    simi = similarity(i,train0,test0,M)
    ss.append(simi)
```

```
######### Start Calculating the similarity between movies #########
```

6.2.1 5.2.1. $K = 1$

First we choose $K = 1$, i.e. pick the movie that is most similar to the objective movie, and use its rating as prediction.

[14]:
```
k = 1
pred_train = knn(k, train0, test0, M, ss)
pred_test = knn(k, train0, test0, M, ss)
print('Training RMSE:', RMSE(train0, pred_train))
```

```
print('Testing RMSE:', RMSE(test0, pred_test))
```

```
Training RMSE: 0.8188360959482575
Testing RMSE: 1.0857543086234802
```

The performance is slightly better than baseline method.

6.2.2 5.2.2. $K > 1$

When we choose $K > 1$, the prediction would be a weighted average of ratings of the K most similar movies, as proposed by [4]. More specifically the formula for predicting user i's rating on movie j is

$$P_{i,j} = \frac{\sum_{k \in I_i^K(j)} s(v_j, v_k) R_{i,k}}{\sum_{k \in I_i^K(j)} s(v_j, v_k)}$$

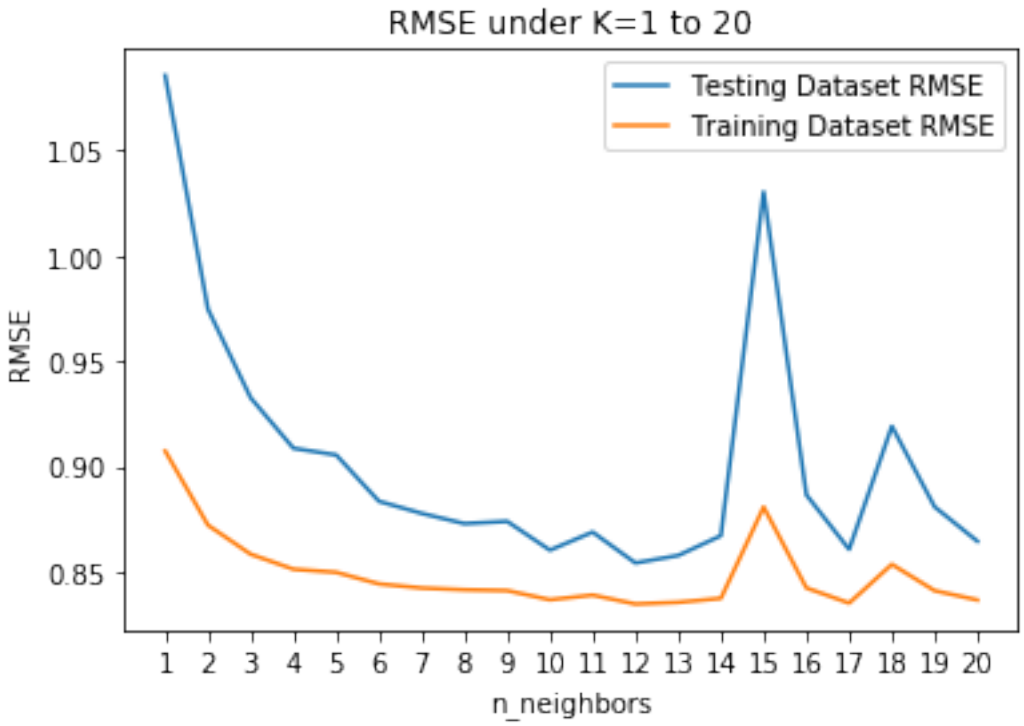
Here $I_i^K(j) = \{k : k \text{ belongs to the } K \text{ most similar movies to movie } j \text{ and user } u \text{ has rated } k\}$.

One problem that would possibly arise is that since similarities lie between -1 and 1, there would be numerical issues when the denominator is close to 0.

Next we evaluate the performance under different k's.

[19]:
```
Ks = np.arange(1, 21, 1)
rmse_test = np.zeros(len(Ks))
rmse_train = np.zeros(len(Ks))

for i, k in enumerate(Ks):
    #print('K:',k)
    pred_test = knn(k,train0,test0,M,ss)
    error_test = RMSE(test0, pred_test)
    rmse_test[i] = error_test

    pred_train = knn(k,train0,train0,M,ss)
    error_train = RMSE(train0, pred_train)
    rmse_train[i] = error_train

print('######## Lowest RMSE:', min(rmse_test))
#Generate plot
plt.plot(Ks, rmse_test, label = 'Testing Dataset RMSE')
plt.plot(Ks, rmse_train, label = 'Training Dataset RMSE')


plt.legend()
plt.xlabel('n_neighbors')
plt.ylabel('RMSE')
plt.title('RMSE under K=1 to 20')
plt.xticks(Ks)
plt.savefig('../figs/knn.png')
```

```
######## Lowest RMSE: 0.8542394216110797
```



The lowest RMSE 0.854239 is achieved at $K = 12$, which improves baseline method by $1 - \frac{0.854239}{1.169} = 26.9\%$.

6.3 5.3. Kernel Ridge Regression (KRR)

Another postprocessing method is Kernel Ridge Regression using only movie feature M, and discard user feature U.

For any user i, let $y = i^{th}$ row of rating matrix R, with missing values omitted, i.e. y is the vector of ratings of movies rated by user i. Let X be a matrix of observations - each row of X is a normalized vector of features of one movie j rated by user i. We can predict user i's rating on movie j using ridge regression:

$$\hat{y}_j = x_j^\top \hat{\beta} = x_j^\top (X^\top X + \lambda I)^{-1} X^\top y$$

where x_j is the feature vector of movie j. Another form is

$$\hat{y}_j = \hat{\beta}^\top x_j = y^\top (XX^\top + \lambda I)^{-1} X x_j^\top$$

Imposing kernel trick on this formula leads to KRR.

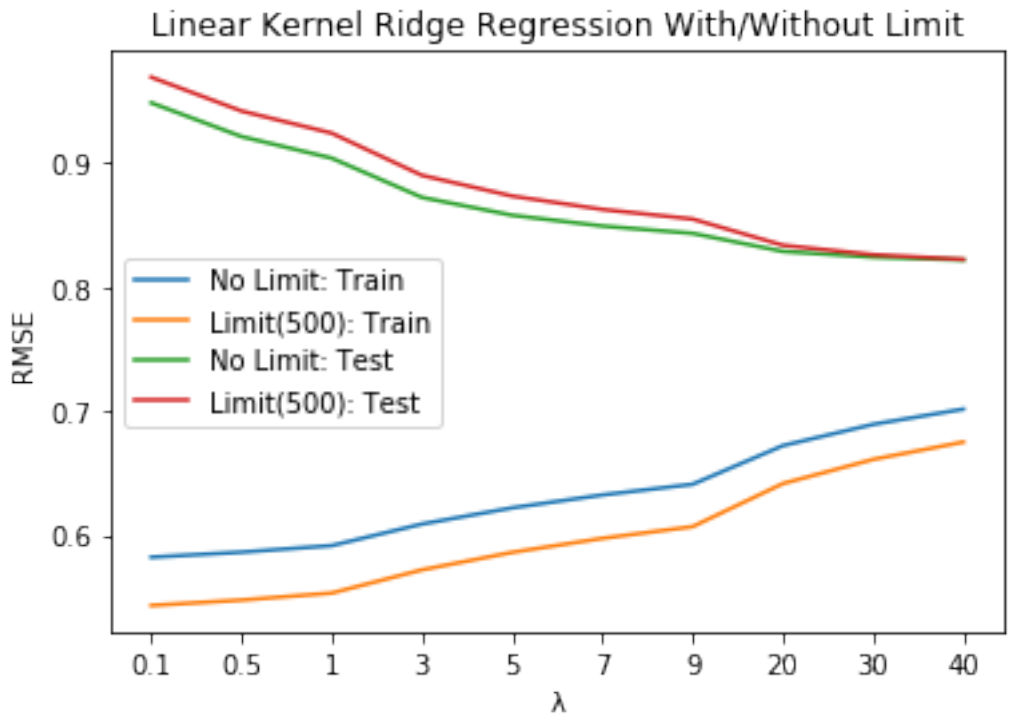
We apply KRR to every user and take the average of errors.

[20]:
```
%run ../lib/krr.py
from sklearn.metrics import mean_squared_error
from sklearn.kernel_ridge import KernelRidge
import time
```

6.3.1 5.3.1. Linear Kernel

First we apply linear kernel, i.e. ordinary ridge regression.

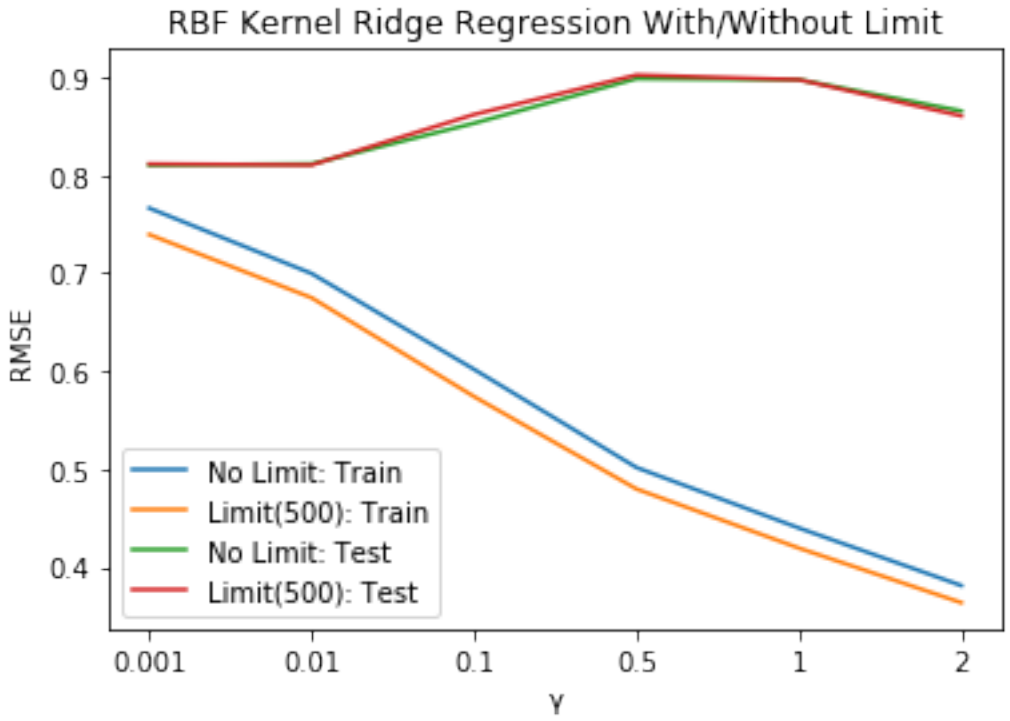
[21]:
```
n = train0.shape[0]
n_train = np.count_nonzero(train0)
n_test = np.count_nonzero(test0)
lambda_list = [0.1, 0.5, 1, 3, 5, 7, 9, 20, 30, 40]
test_error = []
test_error_l = []
train_error = []
train_error_l = []
n_rating_user = np.count_nonzero(train0,1)
sorted_user = sorted(range(len(n_rating_user)), key=lambda x: n_rating_user[x])
for l in lambda_list:
    teste = []
    teste_l = []
    traine = []
    traine_l = []
    for i in sorted_user:
        nolimit = krr_linear(i,train0,test0,M,l)
        traine.append(nolimit[0])
        teste.append(nolimit[1])
        limit = krr_linear(i,train0,test0,M,l,limit=500)
        traine_l.append(limit[0])
        teste_l.append(limit[1])
    train_error.append(np.sqrt(sum(traine)/n_train))
    train_error_l.append(np.sqrt(sum(traine_l)/n_train))
    test_error.append(np.sqrt(sum(teste)/n_test))
    test_error_l.append(np.sqrt(sum(teste_l)/n_test))

plt.figure()
plt.plot(np.arange(1,len(lambda_list)+1,1), train_error, label = 'No Limit:␣
 ↪Train')
plt.plot(np.arange(1,len(lambda_list)+1,1), train_error_l, label = 'Limit(500):␣
 ↪Train')
plt.plot(np.arange(1,len(lambda_list)+1,1), test_error, label = 'No Limit:␣
 ↪Test')
plt.plot(np.arange(1,len(lambda_list)+1,1), test_error_l, label = 'Limit(500):␣
 ↪Test')
plt.legend()
plt.xticks(np.arange(1,len(lambda_list)+1,1), (str(x) for x in lambda_list))
plt.xlabel('')
plt.ylabel('RMSE')
plt.title('Linear Kernel Ridge Regression With/Without Limit')
plt.savefig('../figs/linear.png')
```



The lowest testing RMSE we obtained is 0.8218 at $\lambda = 40$, which improves baseline by 29.7%. Next we use RBF kernel with fixed $\lambda = 1$.

[23]:
```
gamma_list = [0.001, 0.01, 0.1, 0.5, 1, 2]
test_error = []
test_error_l = []
train_error = []
train_error_l = []
n_rating_user = np.count_nonzero(train0,1)
sorted_user = sorted(range(len(n_rating_user)), key=lambda x: n_rating_user[x])
for l in gamma_list:
    teste = []
    teste_l = []
    traine = []
    traine_l = []
    for i in sorted_user:
        nolimit = krr_rbf(i,train0,test0,M,1,l)
        traine.append(nolimit[0])
        teste.append(nolimit[1])
        limit = krr_rbf(i,train0,test0,M,1,l,limit=500)
        traine_l.append(limit[0])
        teste_l.append(limit[1])
    train_error.append(np.sqrt(sum(traine)/n_train))
    train_error_l.append(np.sqrt(sum(traine_l)/n_train))
    test_error.append(np.sqrt(sum(teste)/n_test))
```

```
    test_error_l.append(np.sqrt(sum(teste_l)/n_test))

plt.figure()
plt.plot(np.arange(1,len(gamma_list)+1,1), train_error, label = 'No Limit:␣
 ↪Train')
plt.plot(np.arange(1,len(gamma_list)+1,1), train_error_l, label = 'Limit(500):␣
 ↪Train')
plt.plot(np.arange(1,len(gamma_list)+1,1), test_error, label = 'No Limit: Test')
plt.plot(np.arange(1,len(gamma_list)+1,1), test_error_l, label = 'Limit(500):␣
 ↪Test')
plt.legend()
plt.xticks(np.arange(1,len(gamma_list)+1,1), (str(x) for x in gamma_list))
plt.xlabel('')
plt.ylabel('RMSE')
plt.title('RBF Kernel Ridge Regression With/Without Limit')
plt.savefig('../figs/rbf.png')
```



The lowest testing RMSE we obtained is 0.8104 at $\gamma = 0.001$, which improves baseline by 30.7%.

Remark: Although the performance of KRR is better than KNN, it may not be as reliable as KNN in a different dataset because

- There is a weird monotonicity of error with regards to tuning parameters on both kernels;
- The trends of training error and testing error are not consistent.

and we cannot explain the reason for that.

7 6. Conclusion

We significantly improved the performance compared to baseline method by postprocessing. The biggest improvement is obtained by KRR but it may not be as reliable as KNN due to its weird shape of train-test-error curve that we can not explain.

8 7. Reference

1. Y. Zhou et al., "Large-Scale Parallel Collaborative Filtering for the Netflix Prize", Proc. 4^{th} Int'l Conf. Algorithmic Aspects in Information and Management, LNCS 5034, Springer, 2008, pp. 337-348.

2. Y.Koren. "Matrix Factorization Techniques for Recommender Systems", Journal Computer, 42, No.8, 2009, pp. 30-37.

3. A.Patarek. "Improving Regularized Singular Value Decomposition for Collaborative Filtering", Proc. KDD Cup and Workshop, ACM Press, 2007, pp. 39-42.

4. Z.Wen. "Recommendation System Based on Collaborative Filtering", Stanford CS229 Project Report, 2008.