Practical Work 3

CAO Anh Quan
M2 D&K Factorization-Based Data Modeling

January 29, 2018

Contents

1 Project Structure

- Factorization_TP3_REPORT.pdf: The report for this project.
- dsgd_mf.cpp: Contains the implementation for distributed SGD algorithm.

- compute_rmse.cpp: Compute the root mean square error using the output from the sgd dsgd_mf.cpp file.
- plot_rmse.py: Plot the result from the compute_rmse.cpp program.
- sgd.sh: Bash script file:
  1. Compile and run the dsgd_mf.cpp.
  2. Compile and run the compute_rmse.cpp
  3. Plot the errors returned from the compute_rmse.cpp
- test_rank.sh: Test the dsgd_mf.cpp with different rank values.
- test_step_size.sh: Test the dsgd_mf.cpp with different step_size values.

2 Complete the file dsgd_mf_template.cpp

2.1 Function initalizeGSLMatrix

```
void initalizeGSLMatrix(gsl_matrix * M, double mult)
{
    int nrows = M->size1;
    int ncols = M->size2;

    for (int i = 0; i < nrows; i++)
      {
          for (int j = 0; j < ncols; j++)
          {
              //i and j'th entry of m1
            double value = ((double)rand() / (((double)RAND_MAX)) ) * mult;
              M->data[i * M->tda + j] = value;
          }
      }

}
```

2.2 Function computeDivDiff

```
void computeDivDiff(mtxElm * X, int dataBlockSize, mtxElm * div_term,
    gsl_matrix * Z1, gsl_matrix * Z2)
{
    int s3 = Z1->size2;


    for(int i =0; i<dataBlockSize; i++)
    {
        int curI1   = X[i].i;
```

```
        int curI2   = X[i].j;
        double curX = X[i].val;

        double curXhat = 0;
        //Compute the corresponding curXhat
        for (int k = 0; k < s3; k++) {
            double z1 = Z1->data[curI1 * Z1->tda + k];
            double z2 = Z2->data[k * Z2->tda + curI2];
            curXhat += z1 * z2;
        }
        div_term[i].i = curI1;
        div_term[i].j = curI2;
        div_term[i].val = (curXhat-curX);

    }

}
```

2.3 Function computeZ1update

```
void computeZ1update(mtxElm * div_term, int dataBlockSize, gsl_matrix * Z2,
    gsl_matrix * Z1update)
{
    int s3 = Z2->size1;

    for(int i =0; i<dataBlockSize; i++)
    {
        int curI1   = div_term[i].i;
        int curI2   = div_term[i].j;
        double curVal = div_term[i].val;


        //Compute Z1update (the gradient wrt Z1)
        for (int k = k; k < s3; k++) {
            Z1update->data[curI1 * Z1update->tda + k] += curVal * Z2->data[k *
    Z2->tda + curI2];
        }
    }


}
```

2.4 Function computeZ2update

```
void computeZ2update(mtxElm * div_term, int dataBlockSize, gsl_matrix * Z1,
    gsl_matrix * Z2update)
{
```

```
    int s3 = Z1->size2;

    for(int i =0; i<dataBlockSize; i++)
    {
        int curI1   = div_term[i].i;
        int curI2   = div_term[i].j;
        double curVal = div_term[i].val;

        //Compute Z2update (the gradient wrt Z2)
        for (int k = k; k < s3; k++) {
            Z2update->data[k * Z2update->tda + curI2] += curVal * Z1->data[
    curI1 * Z1->tda + k];
        }
    }
}
```

2.5 Update Z1 and Z2 with Gradient Descent

```
// Update Z1 with Gradient Descent
for (int i = 0; i < curS1; i++)
{
    for (int k = 0; k<s3; k++)
    {
        Z1block->data[i * Z1block->tda + k] -= eta * Z1update->data[i *
    Z1update->tda + k];
    }
}

// Update Z2 with Gradient Descent
for (int k = 0; k < s3; k++)
{
    for (int j = 0; j < curS2; j++)
    {
        Z2block->data[k * Z2block->tda + j] -= eta * Z2update->data[k *
    Z2update->tda + j];
    }
}
```

2.6 Communicate the blocks of Z2

2.6.1 Compute the Source and Destination for sending and receiving Z2's blocks

```
int dest = (processId - 1 + numBlocks) % numBlocks;
int src  = (processId + 1 + numBlocks) % numBlocks;
```

2.6.2 Send and Receive the blocks of Z2

```
MPI_Sendrecv(Z2block->data, fact2BlockSize, MPI_DOUBLE,
        dest, DSGD_Z2BLOCK,
        tempBuf, fact2BlockSize, MPI_DOUBLE,
        src, DSGD_Z2BLOCK,
        MPI_COMM_WORLD, &st);

for (int i = 0; i<s3; i++)
{
    for (int j = 0; j<s2; j++)
    {
        Z2block->data[i * Z2block->tda + j] = tempBuf[i * Z2block->tda + j];
    }
}
```

2.6.3 Compute the block index of the original matrix

```
//form a new part
blockI1 = blockI1;
blockI2 = (blockI2 + 1 + numBlocks) % numBlocks;
```

3 Set the rank to 10 and the step size to 0.00001. Run the code for MovieLens 1 Million Dataset

Bash Script sgd.sh for running SGD.

```
rank=$1
eta=$2
iter=$3

echo The rank is $rank
echo The eta is $eta
echo The iter is $iter

mpicxx dsgd_mf.cpp -Wall -I/usr/local/include -L/usr/local/lib  -lgsl -
    lgslcblas -lm -o dsgd_mf
mpirun -n 4 ./dsgd_mf 3883 6040 $rank $iter $eta 1

mpicxx compute_rmse.cpp -Wall -I/usr/local/include -L/usr/local/lib  -lgsl -
    lgslcblas -lm -o compute_rmse
mpirun -n 10 ./compute_rmse 3883 6040 $rank $iter 4
python3 plot_rmse.py "$rank $eta $iter"
```

To run the distributed SGD with the rank of 10, eta of 0.00001 and iter of 20, we use the following command.

```
./sgd.sh 10 0.00001 20
```

4 Compute the RMSE by using the code compute_rmse.cpp and plot the RMSE

I rewrite the plot_rmse.m into python code because I cannot find a way to execute Matlab script in Linux command line using Octave. The python code need one input which is the **image name**, and it uses that name to **save the plot** of the RMSE.

```
from os import listdir
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import sys

# data size
s1 = 3883
s2 = 6040
s3 = 10

image_name = sys.argv[1]

outpath = './rmse'

rmse = []

files = listdir(outpath)
for i in range(len(files)):
    f = open(outpath + "/" + str(i) + ".txt", "r")
    lines = f.readlines()
    for line in lines:
        rmse.append(float(line))

data = pd.DataFrame(list(zip(list(range(len(rmse))), rmse)))
data.columns = ["Iteration", "RMSE"]


ax = sns.lineplot(x="Iteration", y="RMSE", legend="full", data=data)
plt.savefig(image_name + ".png")
```

The Result Image

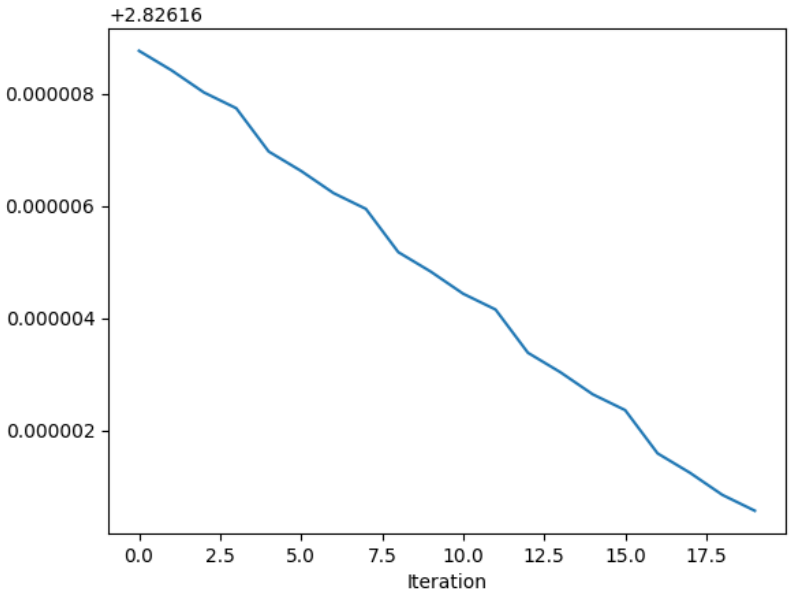


Figure 1: The RMSE values with respect to each iteration using the rank of 10 and the eta of 0.00001

According to the figure 1, we can see that the code is implemented correctly so that the RMSE decreases as expected.

5 Play with the rank and the step-size. What do you observe?

5.1 Play with the rank

I would like to test the ranks with the following values: 10, 20, 30, 40, 50, 60 and with eta=0.00001 and iter=20. I created a bash script file to automatically iterate over all the values and compute the sgd, then plot the RMSE.

test_rank.sh

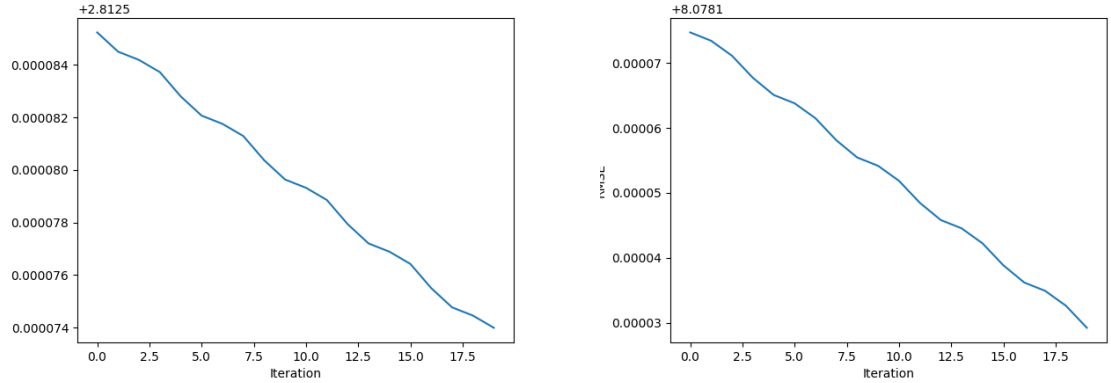
```
ranks=(10 20 30 40 50 60)
eta=0.00001
iter=20

for rank in "${ranks[@]}"; do
    ./sgd.sh $rank $eta $iter;
done
```

To excute the file, we run the command below

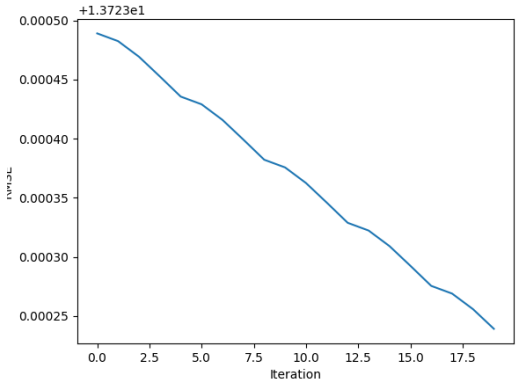
```
./test_rank.sh
```

Results

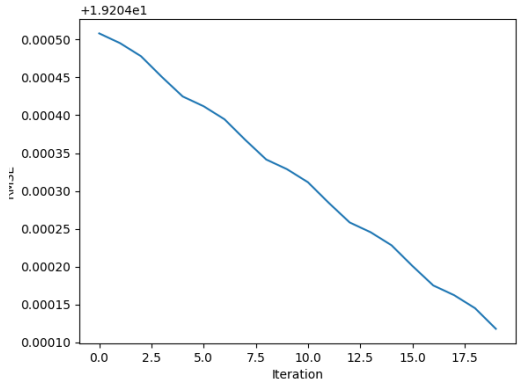


(a) Rank 10

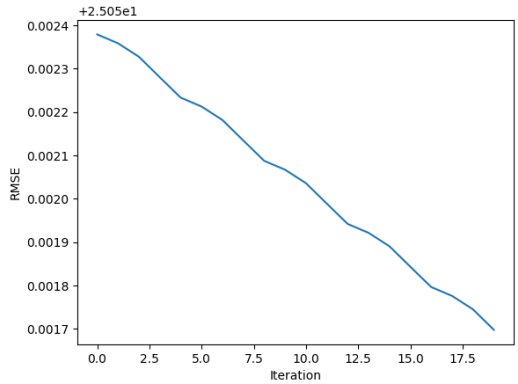
(b) Rank 20



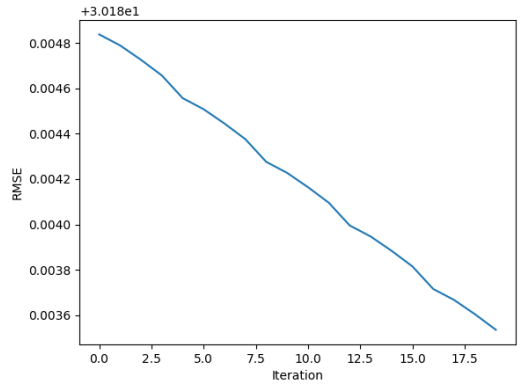
(c) Rank 30



(d) Rank 40



(e) Rank 50



(f) Rank 60

Figure 2: The change of root mean square error values concerning the number of iterations when we change the value of the **Rank**

We can see from 6 figures above that; the RMSE generally increases as we increase the rank. The reason is that when we have a higher rank, each entry in the X hat matrix will be represented by a more significant sum which increases the error. This also makes the gradient bigger which increase the convergence speed. Therefore, as I observed from the TP 2, if the rank is too big, the algorithm can fail to converge because of the too big gradient, and if we want the algorithm to converge again, we need to decrease the step size.

Furthermore, I think the higher the rank, the closer we can get to the original matrix because with a higher rank, we have more basis vectors so that we can represent more vectors in the original matrix so that we should have smaller RMSE. Thus, the high rank will also make the algorithm more sensitive to the noise.

5.2 Play with the step-size

I would like to test the step-size with the following values: 0.00001, 0.001, 0.1, 1 with rank=10 and iter=20. I created a bash script file to automatically iterate over all the values and compute the sgd, then plot the RMSE.

test_step_size.sh

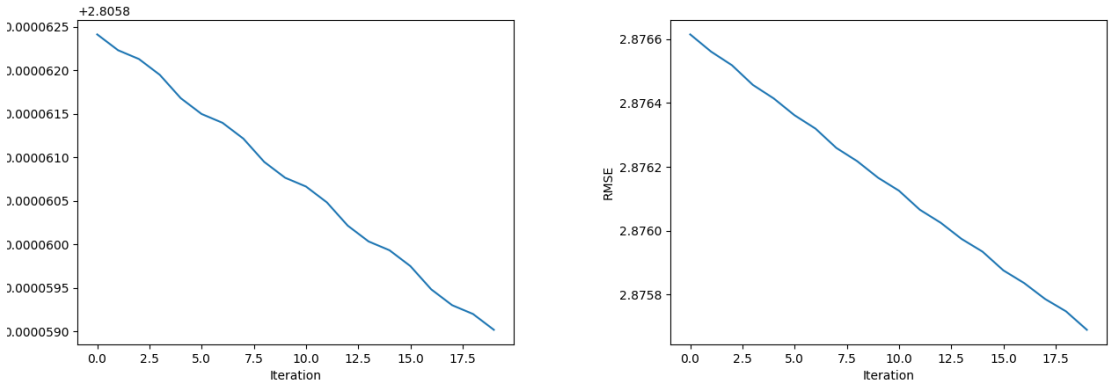
```
rank=10
etas=(0.00001 0.001 0.1 1)
iter=20

for eta in "${etas[@]}"; do
    ./sgd.sh $rank $eta $iter;
done
```

To excute the file, we run the command below

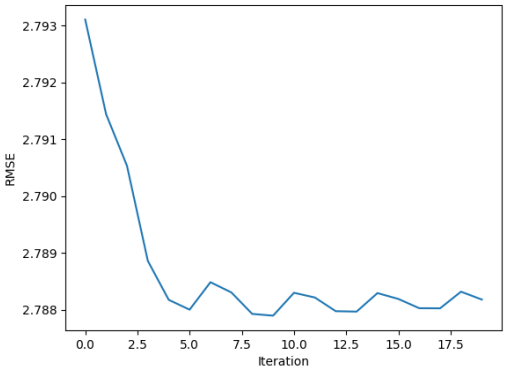
```
./test_step_size.sh
```

Results

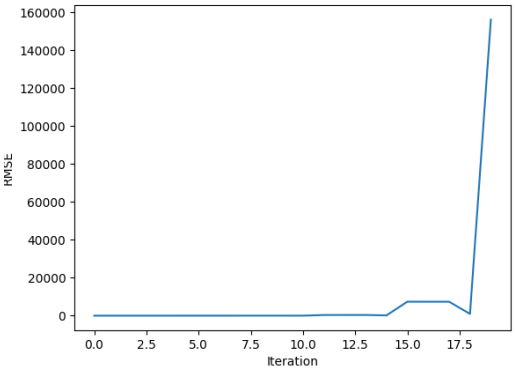


(a) eta=0.00001

(b) eta=0.001



(c) eta=0.1



(d) eta=1

Figure 3: The change of root mean square error values with respect to the number of iterations when we change the value of the **step-size**

According to 4 figures above, when we increase the eta, the algorithm converges faster. We can see from the figure 3c that with eta=1, the algorithm reaches minima in only five steps. After that, it fluctuates around that point because, at this moment, the gradient is not small enough to converge. The solution to this problem is the Learning Rate Schedules or Adaptive Learning Rate. If we have a too large eta, the algorithm cannot converge because we have an excessively large gradient. On the other hand, if the eta is too small, the algorithm will need many iterations which could be forever to converge. Therefore, we need to choose a suitable value for the **step-size** so that the algorithm can converge to the minima at decent convergence speed.