Soft-Impute Matrix Completion Project Report

Dapeng Hu, Qinglong Tian, Haozhe Zhang, Min Zhang
Department of Statistics, Iowa State University, Ames, IA 50011

1 Summary

The purpose of matrix completion is to solve the problems of the following type: given data in the form of an $m \times n$ matrix $Z = \{Z_{ij}\}$, find an approximating matrix $\widehat{Z}$ that imputes or fills in missing entries in Z. There are wide applications of matrix completion. One popular example is the movie-rating problem in the "Netflix" Competition, where the data is the basis for a recommender system.

The matrix completion problem is ill-specified unless we impose additional constraints on the unknown matrix Z, and one common choice is a rank constraint. Suppose that we observe all entries of the matrix Z indexed by the subset $\Omega \subset \{1, \ldots, m\} \times \{1, \ldots, n\}$. One approach is to consider estimators based on optimization problems of the form

$$\widehat{Z} = \arg\min_{M} ||Z - M||_F^2 \ \text{ subject to rank}(M) \leq \delta, \tag{1}$$

where $|| \cdot ||_F^2$ is the Frobenius norm of a matrix. However, this rank-minimization problem is computationally intractable (NP-hard), and cannot be solved in general even for moderately large matrices. Alternatively, we consider the following small modification to (1):

$$\widehat{Z} = \arg\min_{M} ||Z - M||_F^2 \ \text{ subject to } ||M||_* \leq \delta, \tag{2}$$

where $|| \cdot ||_*^2$ is the nuclear norm, i.e., the sum of the sigular values. Under many situations, the nuclear norm is an effective convex relaxation to the rank constraint, and hence problem (2) is convex.

Soft-Impute Algorithm was proposed and studied in [5] to solve problem (2). It was shown in [5] that Soft-Impute Algorithm is guaranteed to converge at least sub-linearly, meaning that $O\left(\frac{1}{\epsilon}\right)$ iterations are sufficient to compute a solution that is ϵ-close to the global optimum. The procedure for Soft-Impute Algorithm is introduced briefly as follows. First considering the case where there is no missing data, to solve (2), we simply compute the SVD of Z, soft-threshold the singular values by λ, and reconstruct the matrix. The singular value decomposition (SVD) provides an effective method here. Then, we start with an initial guess for the missing values, compute the (full rank) SVD, and then soft-threshold its singular values by λ. We reconstruct the corresponding SVD approximation and obtain new estimates for the missing values. This process is repeated until convergence.

2 Various SVD Implementations and Comparison

2.1 Brief Introduction to SVD Implementations

- **C code via LAPACK**:
  The core part of the c code for implementing SVD is the LAPACK routine DGESVD and DGEMM. The first argument of the C main function is filename, and the second argument is λ used in Soft-Impute Algorithm. The format of the input data is like the given Lena dataset. The first row is the dimension of the matrix with N is the row, and P is the column. The second row is the matrix arranged by column.

- **R internal svd function**:
  The LAPACK routines DGESDD and ZGESDD are used. The performance of R internal svd function essentially depends on the performance of LAPACK.

- **svd Package**:
  There are three functions in the svd package that are related to the singular-value decomposition. Because "trlan.svd" and "ztrlan.svd" will not return the right singular vectors which we need in soft-impute algorithm. So we are not going to use the two functions. We only use "propack.svd" to implement the soft-impute algorithm. PROPACK does SVD via the implicitly restarted Lanczos bidiagonalization with partial reorthogonalization.

- **RcppArmadillo Package**:
  The "RcppArmadillo" package includes the header files from the "Armadillo" library, which is a templated C++ linear algebra library. Various matrix decompositions are provided through optional integration with LAPACK and ATLAS libraries. We use svd in "Armadillo" to implement Soft-Impute Algorithm.

- **irlba package**:
  We use "irlba" function in the "irlba" package. The augmented implicitly restarted Lanczos bidiagonalization algorithm (IRLBA) finds a few approximate largest singular values and corresponding singular vectors of a sparse or dense matrix using a method of Baglama and Reichel. It is a fast and memory-efficient way to compute a partial SVD. To find the suitable number of singular values in each iteration of Soft-Impute Algorithm, we compare the number of thresholded singular values with the maximum allowed number of singular values then take the minimum of these two values as the suitable number in each iteration.

2.2 Simulation Study

There are many aspects that can reflect the performance of one method, such as cross-validation error, speed and memory usage. In our comparison, these four methods are just different implementations of the same algorithm, so they should have the same cross-validation error. And in our simulation experiments, they do have the same errors, so we are not going to compare the CV errors. We are going to compare the speed of each method in our experiments.

There are three factors that may influence the speed. They are matrix dimension, missing rate and true matrix rank. In our experiment, we let one variable vary and keep other two variables fix. To compare the speed of each method, we use "microbenchmark" to calculate the time consumption and take the median as the result.

2.2.1 Comparison under different matrix dimensions

Table 1: Time consumption under different matrix dimensions with missing rate = 0.5 and true matrix rank = 8

No. of rows	No. of columns	R interval svd	propack.svd	RcppArmadillo	irlba
20	20	0.00246	0.00513	**0.00232**	0.00335
50	50	0.02001	0.03667	0.01876	**0.01680**
100	100	0.12742	0.20482	**0.12269**	0.13500
500	500	18.0418	28.8572	**17.0435**	72.1405
1000	1000	**154.7145**	237.7374	164.2245	1197.6262

In general, the result shows that the computation speed of R internal svd and RcppArmadillo methods are similar, and they are the fastest methods. When the matrix size is relatively large (e.g., 1000×1000), the R internal svd performs a little better than RcppArmadillo. Otherwise, RcppArmadillo is a little better. The propack.svd method is slower than the two methods mentioned above. While the difference becomes less and less as the dimension becomes larger and larger. As for the irlba method, when the dimension is relatively small (≤ 100), this method performs similarly compared with R internal svd and RcppArmadillo methods. But when the matrix dimension is relatively large, the time consumption of this method will go up dramatically as we can see in Table 1.

2.2.2 Comparison under different missing rates

Table 2: Time consumption under different missing rates with matrix dimension = 100×100 and true matrix rank = 8

Missing rate	R interval svd	propack.svd	RcppArmadillo	irlba
0.4	0.10531	*	**0.10293**	0.13194
0.5	0.13925	0.21978	**0.13327**	0.14786
0.6	0.14494	0.23622	0.14057	**0.12816**
0.7	0.19295	0.30568	0.18452	**0.13670**
0.9	0.28146	*	0.26986	**0.09553**

As a whole, the R internal svd and RcppArmadillo perform similarly while RcppArmadillo is a little bit faster than R internal svd. The propack.svd method is the most unstable method in this situation. When the missing rate is too large or too small, this method fails to execute. Even though the missing rate is suitable, this method is also the slowest one among all the methods. The irlba method performs better as the missing rate goes up. For other three methods, they use more time as the missing rate goes up. While the irlba method is different. When the missing rate is large (≥ 0.6) and the dimension is medium (e.g., 100×100), the irlba method is the fastest method.

2.2.3 Comparison under different matrix ranks

Table 3: Time consumption under different matrix ranks with matrix dimension = 100×100 and missing rate = 0.5

True matrix rank	R interval svd	propack.svd	RcppArmadillo	irlba
4	0.11331	0.18107	0.10620	0.10551
8	0.12308	0.19143	0.11715	0.12893
12	0.14849	0.23911	0.14536	0.16806
16	0.16822	*	0.17642	0.23704

All of the methods use more time as the true rank goes up. RcppArmadillo method is the fastest although it is just a little faster than R internal svd method. The propack.svd is still the slowest and most unstable method because when the true rank is 16 it fails to execute again. For the irlba method, when the true rank is small(eg ≤ 8), the speed is very competitive. When the true rank is large, it becomes slower than R internal svd and RcppArmadillo method.

3 Application

3.1 Lena Image Data

After comparing different methods, we decide to use R internal svd to solve this problem. To design a grid of λ, we first use svd to decompose the training matrix and get the singular values of it . Then we use the quantile of the singular values of the training matrix to design the grid of λ. There are two reason why we do in this way. The first is that we have no idea about the range of so we need some reference to specify. The second reason is that by using quantile we can reduce the number of search times. We first use (The subscript represents the quantile) as the vector. And calculate the RMSE in the validation set and we get the Figure 1. We find that the minimal of the RMSE corresponding to 150. Then we do further search. After calculating the RMSE in the validation set we get the Figure 2. This time we find the minimal of the RMSE corresponding to 180 . We continue our search to find the best. The RMSE can be seen in the Figure 3. In Figure 3, we can see that the minimal of RMSE corresponding to 168, which is the best. In this way, we can reduce the total number of iterations and therefore save a lot of time.

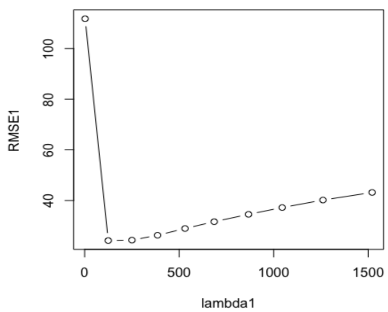

Figure 1

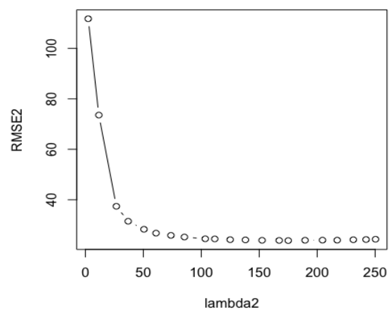

Figure 2

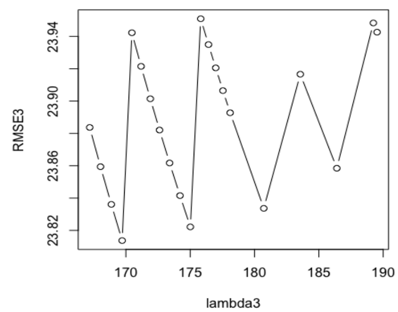

Figure 3

Figure 4 is the image with 40% pixels randomly missing. And Figure 5 is the image after using the soft-impute to fix. The total rank in the reconstructed image is 83.

Figure 4: Imputed Lena Image

Figure 5: Original Lena Image

3.2 MovieLens Dataset

We randomly split MovieLens 100K Dataset into three parts: training set (70%), validation set (15%) and test set (15%). The validation set was used to perform hold-out validation for the selection of λ. Here, we applied Soft-Impute Algorithm to MovieLens 100K Dataset using the implemention of R internal svd function and the implementation of propack.svd in "svd" package. We compared their speed and performance.

The selection of optimal values of λ is critical to the performance of Soft-Impute Algorithm. Because exhausted searching is extremely time consuming, we used a multi-step approach. First, we did singular value decomposition of the original matrix (missing entries imputed by zeros) and obtain a vector of singular values. We can select from a sequence of quantiles of these singular values (e.g., 5%, 6%, 7%, ..., 98%, 99%). The selected value of λ is denoted by λ^*. Then, we select another value of λ from $[\lambda^* - \delta, \lambda^* + \delta]$, a local region of λ^*. We continue repeating the procedure until the selected value attains a pre-specified resolution. Figure 6 illustrates the selection procedure introduced above using the implementation of R internal svd function. For instance, in the first step, $\lambda = 11.70$ has the smallest RMSE for validation set. Then, we selected 12.33 from $[6.70, 16.70]$. After four steps, our final value of optimal λ is 12.27333.

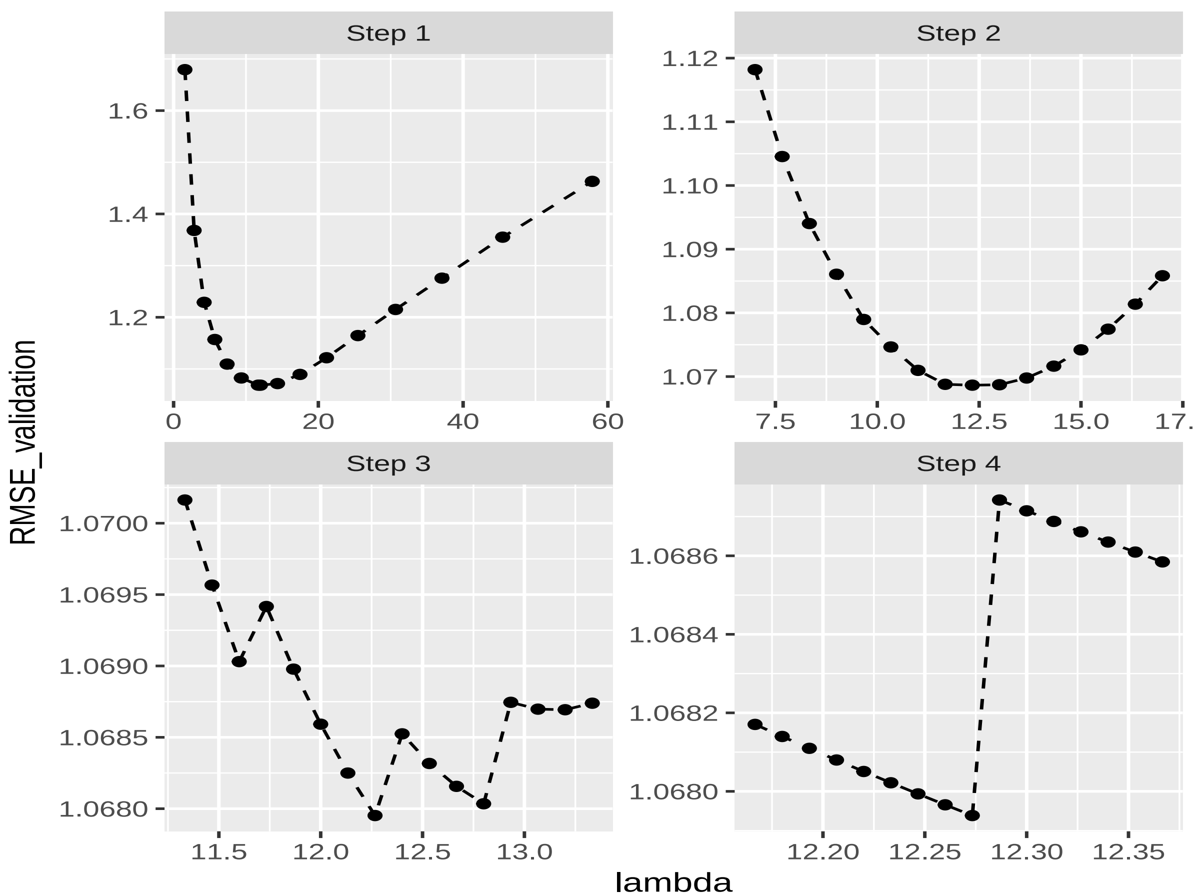

Figure 6: Selection of λ

The speed performance of svd implementations depends on the value of λ. We compared the speed performance of R internal svd function and propack.svd in "svd" package under different values of λ. After investigating Figure 7, we can conclude that the implementation with R internal svd function is faster than the implementation with rpack.svd function.

For MovieLens 100K Dataset, after hold-out validation, the optimal value of $\boldsymbol{\lambda}$ is 12.27333. The RMSE for testing set is 1.067938, while the RMSE for validation set is 1.00216.

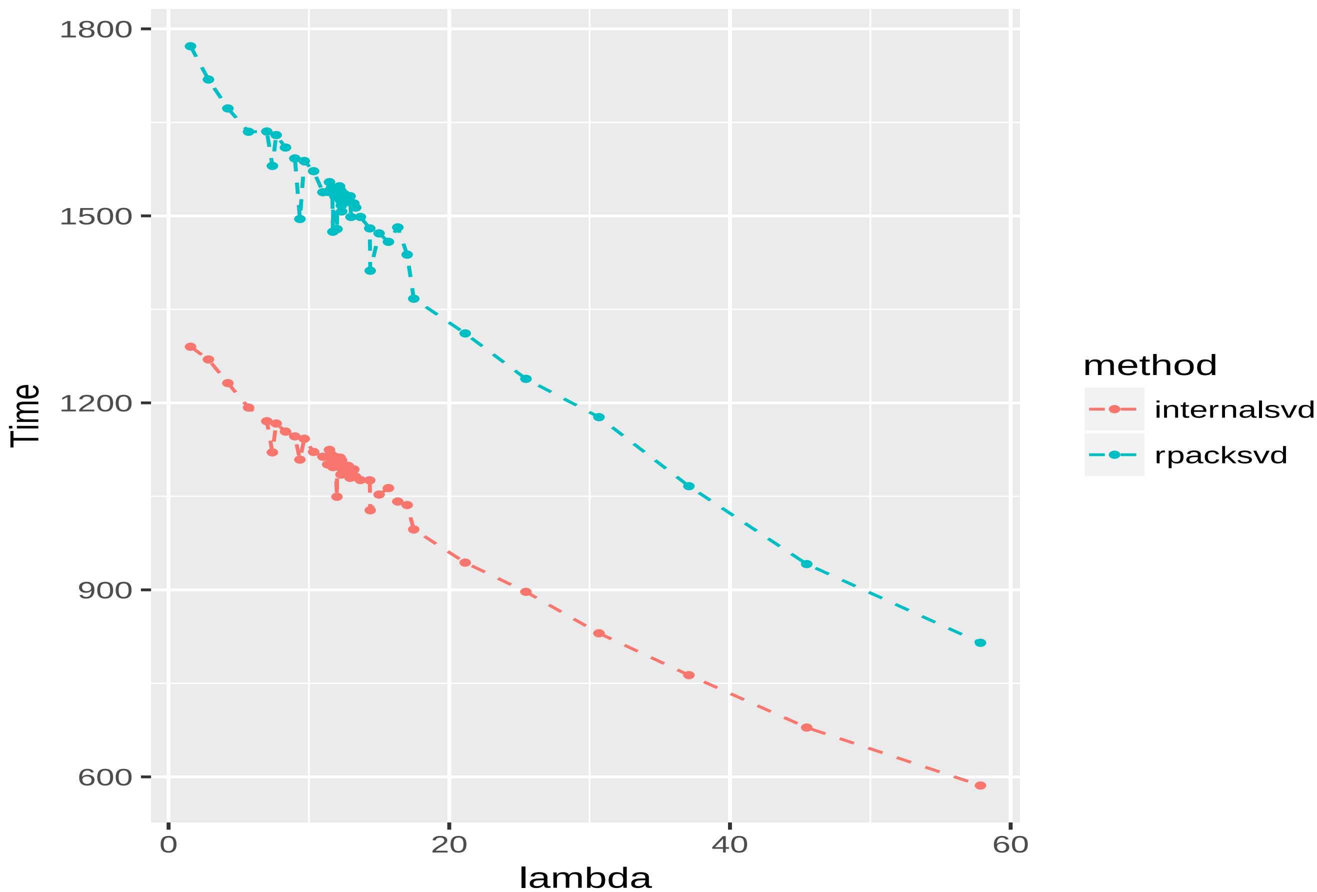

Figure 7: The time consumption for two implementations under different values of λ

4 Accelerating Soft-Impute

Let us consider a general setting of convex composite minimization problem

$$\min_{\boldsymbol{x}} F(\boldsymbol{x}); = f(\boldsymbol{x}) + g(\boldsymbol{x}),$$

where $f(\boldsymbol{x})$ is convex and L-smooth (i.e., gradient is Lipschitz continuous), and $g(\boldsymbol{x})$ is convex and non-smooth. The proximal operation of g is easy to calculate here. The proximal gradient method [2] is used to solve this problem: for each iteration $k = 0, 1, 2, \ldots,$

$$\boldsymbol{x}_{k+1} = \text{prox}_{\gamma_k, g}(\boldsymbol{x}_k - \gamma_k \nabla f(\boldsymbol{x}_k)) = \arg\min_{\boldsymbol{x}} \left[\frac{1}{2\gamma_k} \|\boldsymbol{x} - \{\boldsymbol{x}_k - \gamma_k \nabla f(\boldsymbol{x}_k)\}\|_2^2 + g(\boldsymbol{x}) \right]$$

The following theorem shows that the proximal gradient method attains the convergence rate of $O\left(\frac{1}{k}\right)$.

<u>Theorem</u> **1** *Proximal gradient method with fixed step size $\gamma_k = \frac{1}{L}$ satisfies the following convergence rate*

$$F(\boldsymbol{x}_k) - F(\boldsymbol{x}^*) \le \frac{L||\boldsymbol{x}_0 - \boldsymbol{x}^*||_2^2}{2k}.$$

Originally developed in [6] and [1], we can accelerate the proximal gradient method as follows:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \operatorname{prox}_{\gamma_k, g}(\boldsymbol{y}_k - \gamma_k \nabla f(\boldsymbol{y}_k)) \\ \boldsymbol{y}_{k+1} &= \boldsymbol{x}_{k+1} + \beta_k(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k). \end{aligned}$$

A common choice for β: $\beta_k = \frac{\lambda_k - 1}{\lambda_k + 1}$ where $\lambda_0 = 0$, $\lambda_{k+1} = \frac{1+\sqrt{1+4\lambda_k^2}}{2}$. The acceleration for this choice of β is called the Fast Iterative Soft Thresholding Algorithm (FISTA) derivation [1]. It was shown in [1] that

<u>Theorem</u> **2** *The sequences $\boldsymbol{x}_k$, $F(\boldsymbol{x}_k)$ generated via FISTA with either a constant or back-tracking (with ratio $\alpha \ge 1$) stepsize rule satisfy*

$$F(\boldsymbol{x}_k) - F(\boldsymbol{x}^*) \le \frac{2\alpha L||\boldsymbol{x}_0 - \boldsymbol{x}^*||_2^2}{k^2}.$$

Soft-Impute Algorithm is a special example of proximal gradient method:

$$\min_Z \underbrace{\frac{1}{2}||W - Z||_F^2}_{f(Z)} + \underbrace{\lambda||Z||_*}_{g(Z)}.$$

Lemma 1 in [5] shows that the solution to the above optimization problem is given by $\widehat{Z} = \boldsymbol{S}_\lambda(W)$ where $\boldsymbol{S}_\lambda(W) = UD_\lambda V'$ with $D_\lambda = diag\{(d_1 - \lambda)_+, \ldots, (d_r - \lambda)_+\}$, UDV' is the SVD of W, $D = diag(d_1, \ldots, d_r)$, and $t_+ = \max(t, 0)$. $\boldsymbol{S}_\lambda(W)$ is a type of soft-thresholding operator.

As we have shown, Soft-Impute Algorithm has a convergence rate of $O\left(\frac{1}{k}\right)$, which for large k is worse than the Accelerated Soft-Impute Algorithm with rate $O\left(\frac{1}{k^2}\right)$. However, the numeric results in [5] show that Soft-Impute Algorithm performs favorably compared with the accelerated version illustrated in Figure 5 of [5].

References

[1] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[2] Ronald E Bruck Jr. An iterative solution of a variational inequality for certain monotone operators in hilbert space. *Bulletin of the American Mathematical Society*, 81(5):890–892, 1975.

[3] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[4] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. *Statistical learning with sparsity*. CRC press, 2015.

[5] Rahul Mazumder, Trevor Hastie, and Robert Tibshirani. Spectral regularization algorithms for learning large incomplete matrices. *Journal of machine learning research*, 11(Aug):2287–2322, 2010.

[6] Yurii Nesterov et al. Gradient methods for minimizing composite objective function, 2007.

[7] Tae-Hyun Oh, Yasuyuki Matsushita, Yu-Wing Tai, and In So Kweon. Fast randomized singular value thresholding for nuclear norm minimization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4484–4493, 2015.

[8] Raymond KW Wong and Thomas Lee. Matrix completion with noisy entries and outliers. *arXiv preprint arXiv:1503.00214*, 2015.

APPENDIX

C code via LAPACK

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define Epsilon 0.001
#define Iteration 10000

void dgesvd_(char *jobu, char *jobvt, int *m, int *n, double *a, int *lda, double *s, double
     *u,
             int *ldu, double *vt, int *ldvt, double *work, int *lwork, int *info);
void dgemm_(char *transa, char *transb, int *m, int *n, int *k, double *alpha, double *a,
    int *lda,
            double *b, int *ldb, double *beta, double *c, int *ldc);

int main(int argc, char *argv[]) {
    //The 1st argument is the file name; the 2nd argument is lambda
    char *filename = argv[1];
    FILE *f = fopen(filename, "r");
    double lambda = atof(argv[2]);
    int N, P, i=0, j, id=0;

    //In the dataset, the first 2 observations are N and P
    fscanf(f, "%d", &N);
    fscanf(f, "%d\n", &P);
    double data[N*P], X[N*P], Zcomp[N*P], Ztemp[N*P], Znew[N*P], Zold[N*P];

    // Read in the dataset
    while (!feof(f)) {
        fscanf(f, "%lf*[^lf]", &data[i]);
        i++;
    }

    //Transpose(data) gives X
    for (j=0; j<P; j++) {
        for (i=0; i<N; i++) {
            X[j*N+i] = data[i*P+j];
        }
    }
    //Initialize Zold with 0
    for (i=0; i>N*P; i++) {
        Zold[i] = 0;
    }

    while (id < Iteration) {
        //Compute the complementary matrix of Zold
        for (i=0; i<N*P; i++) {
            if (X[i]==0) {
                Zcomp[i]=Zold[i];
            } else {
                Zcomp[i] = 0;
            }
        }

        //Add X and Zcomp up
        for (i=0; i<N*P; i++) {
            Ztemp[i] = X[i] + Zcomp[i];
        }

        char jobu='A', jobvt='A';
        double S[P], U[N*N], VT[P*P], work[5*N];
        int lwork=5*N, info;
        // Do SVD to Ztemp
        dgesvd_(&jobu, &jobvt, &N, &P, Ztemp, &N, S, U, &N, VT, &P, work, &lwork, &info);
```

```
        // Compute Dlambda
        for (i=0; i<P; i++) {
            if (S[i]-lambda>0) {
                S[i] = S[i] - lambda;
            } else {
                S[i] = 0;
            }

        }
        //Compute Znew
        double UD[N*P], alpha=1.0, beta=0.0;
        char transa='N', transb='N';
        for (j=0; j<P; j++) {
            for (i=0; i<N; i++) {
                UD[N*j+i] = U[N*j+i] * S[j];
            }
        }
        dgemm_(&transa, &transb, &N, &P, &P, &alpha, UD, &N, VT, &P, &beta, Znew, &N);


        //Campare the ratio and Epsilon
        double Nnorm=0, Dnorm=0, ratio;
        for (i=0; i<N*P; i++) {
            Nnorm = Nnorm + pow(Znew[i]-Zold[i],2);
            Dnorm = Dnorm + pow(Zold[i], 2);
        }
        ratio = Nnorm/Dnorm;
        if (ratio<Epsilon) break;
        //Update Zold with Znew
        for (i=0; i<N*P; i++) {
            Zold[i] = Znew[i];
        }
        id++;
    }
   //Print out the final result
    for (i=0; i<N; i++) {
        for (j=0; j<P; j++) {
            printf("%lf", Znew[j*N+i]);
        }
        printf("\n");
    }

    return 0;
}
```

../Codes/Min/SoftImpute.c

R internal svd function

```
soft_thres <- function(x,y){
  ifelse((x-y)>0, x-y, rep(0,length(x)))
}

softimpute_internalsvd <- function(mat_train, lambda, thres_ratio = 10^(-3), mat_validation
    = NULL){
  starting_time <- Sys.time()
  na_index <- is.na(mat_train)
  Z_old <- mat_train
  Z_old[na_index] <- 0
  ratio <- 1 + thres_ratio
  while(ratio>thres_ratio){
    mat_train[na_index] <- Z_old[na_index]
    svd_result <- svd(mat_train)
    Z_new <- svd_result$u%*%diag(soft_thres(svd_result$d, lambda))%*%t(svd_result$v)
    ratio <- sum((Z_old-Z_new)^2)/sum((Z_old)^2)
```

```
    #print(ratio)
    Z_old <- Z_new
  }
  end_time <- Sys.time()
  if(is.null(mat_validation))
    return(list(Z_new, end_time - starting_time))
  else{
    return(c(sqrt(mean((mat_validation - Z_new)[!is.na(mat_validation)]^2)),
             as.numeric(end_time - starting_time, units = "secs")))
  }
}
```

../Codes/haozhe/softimpute_internalsvd.R

svd package

```
library(svd)

soft_thres <- function(x,y){
  ifelse((x-y)>0, x-y, rep(0,length(x)))
}

softimpute_rpacksvd <- function(mat_train, lambda, thres_ratio = 10^(-3), mat_validation =
    NULL){
  starting_time <- Sys.time()
  na_index <- is.na(mat_train)
  Z_old <- mat_train
  Z_old[na_index] <- 0
  ratio <- 1 + thres_ratio
  while(ratio>thres_ratio){
    mat_train[na_index] <- Z_old[na_index]
    svd_result <- propack.svd(mat_train)
    Z_new <- svd_result$u%*%diag(soft_thres(svd_result$d, lambda))%*%t(svd_result$v)
    ratio <- sum((Z_old-Z_new)^2)/sum((Z_old)^2)
    #print(ratio)
    Z_old <- Z_new
  }
  end_time <- Sys.time()
  if(is.null(mat_validation))
    return(list(Z_new, end_time - starting_time))
  else{
    return(c(sqrt(mean((mat_validation - Z_new)[!is.na(mat_validation)]^2)),
             as.numeric(end_time - starting_time, units = "secs")))
  }
}
```

../Codes/haozhe/softimpute_rpacksvd.R

RcppArmadillo Package

```
library(inline)
library(RcppArmadillo)

SoftImpute_Rcpp <- function(x,lambda,epsilon=0.00001,maxiter=1000){
  z_old <- matrix(0, nrow = dim(x)[1], ncol=dim(x)[2])
  i <- 0
  while(1){
    svd1 <- arma_svd(x+(x==0)*z_old)
    D_lambda <- diag(as.vector((svd1$d-lambda>0)*(svd1$d-lambda)))
    z_new <- svd1$u %*% D_lambda %*% t(svd1$v)
    if(sum((z_new-z_old)^2)/sum(z_old^2)<epsilon){
```

```
      break
    }
    if(i>maxiter){
      break
    }
    z_old <- z_new
    i <- i+1
  }
  return(z_new)
}
```

../Codes/haozhe/softimpute_rcpp.R

irlba package

```
library("irlba")
library("Matrix")
SoftImpute_irlba <- function(x,lambda,epsilon=0.00001,maxiter=1000){
  z_old <- matrix(0, nrow = dim(x)[1], ncol=dim(x)[2])
  i <- 0
  n <- trunc(min(dim(x))/2-1)
  D_lambda <- 0
  while(1){
    if(i==0){
      svd1 <- irlba(x+(x==0)*z_old,n)
    } else{
      D <- diag(D_lambda)
      m <- length(D[D>0])
      svd1 <- irlba(x+(x==0)*z_old,min(n,m))
    }
    D_lambda <- diag((svd1$d-lambda>0)*(svd1$d-lambda))
    z_new <- svd1$u %*% D_lambda %*% t(svd1$v)
    if(sum((z_new-z_old)^2)/sum(z_old^2)<epsilon){
      break
    }
    if(i>maxiter){
      break
    }
    z_old <- z_new
    i <- i+1
  }
  return(z_new)
}
```

../Codes/haozhe/softimpute_irlba.R

Lena Data

```
setwd("~/Desktop/ISU/STAT580/FinalProject")
lenadata <- scan(file = "lena256")
lenatemp <- lenadata[-c(1,2)]
lenavec <- NULL
for (j in 1:lenadata[1]){
  for (i in 1:lenadata[2]){
    lenavec[(j-1)*lenadata[1]+i] <- lenatemp[(lenadata[1]-i)*lenadata[1]+j]
  }
}
lena <- matrix(lenavec,ncol = lenadata[2], nrow = lenadata[1], byrow = T)
image(lena,axes=FALSE ,col = grey(seq(0,1,length=256)))
lenasample <- lena
set.seed(12345)
```

```
lenasample[sample(length(lena),size = trunc(0.4*length(lena)))] <- 0
sampleseq <- seq(1,length(lena))
sampleindex <- sampleseq*(lenasample!=0)
nonzerosize <- length(sampleindex[sampleindex!=0])
training_index <- sort(sample(sampleindex[sampleindex!=0],size = trunc(0.7*nonzerosize)))
validation <- lenasample
validation[training_index] <- 0
validation_index <- sampleseq*(validation!=0)
training <- lenasample*(validation==0)
validation1 <- validation[validation_index]

lambda <- svd(training)
lambda <- lambda$d
lambda1 <- sort(quantile(lambda,probs = seq(0,0.9,0.1),names = F),decreasing = T)


RMSE1 <- NULL
for(i in 1:length(lambda1)){
  S1 <- SoftImpute_svd(training,lambda1[i])
  S1_validation <- S1[validation_index]
  RMSE1[i] <- sqrt(sum((validation1-S1_validation)^2)/length(validation1))
}
plot(lambda1,RMSE1,type = "b")

minRMSE_index <- length(lambda1)-order(RMSE1)[1]
lambda2 <- sort(quantile(lambda,probs = seq(minRMSE_index/10-0.1,minRMSE_index/10+0.1,0.01),
    names = F),decreasing = T)
RMSE2 <- NULL
for(i in 1:length(lambda2)){
  S1 <- SoftImpute_svd(training,lambda2[i])
  S1_validation <- S1[validation_index]
  RMSE2[i] <- sqrt(sum((validation1-S1_validation)^2)/length(validation1))
}
plot(lambda2,RMSE2,type = "b")

minRMSE_index2 <- length(lambda2)-order(RMSE2)[1]
lambda3 <-sort(quantile(lambda,probs = seq(minRMSE_index2/100-0.01,minRMSE_index2/
    100+0.01,0.001),names = F),decreasing = T)
RMSE3 <- NULL
for(i in 1:length(lambda3)){
  S1 <- SoftImpute_svd(training,lambda3[i])
  S1_validation <- S1[validation_index]
  RMSE3[i] <- sqrt(sum((validation1-S1_validation)^2)/length(validation1))
}
plot(lambda3,RMSE3,type = "b")
lambda_best <- lambda3[order(RMSE3)[1]]



S1 <- SoftImpute_svd(lenasample,lambda_best)
image(lenasample,axes=FALSE ,col = grey(seq(0,1,length=256)))
image(S1,axes=FALSE ,col = grey(seq(0,1,length=256)))
```

../Codes/Dapeng/lena.R