STAT 580– Final Project

Wengting Zhao Xingche Guo
Yueying Wang Jingru Mu

May 2, 2017

1 Summary of Matrix Completion

In many applications measured data can be represented in a matrix $\mathbf{Z}_{m\times n}$, for which only a relatively small number of entries are observed. If Z with numerous missing entries, it causes the problem of matrix completion. There is a vast literature that proposed compuation methods on imputing methods in recent years. Here we will mainly focus on a popular matrix completion algorithm called Soft-Impute. This algorithm iteratively replaces the missing elements with those obtained from a soft-thresholded SVD.

Soft-Impute is proposed for the nuclear norm regularized least square problem (or call it *spectral regularization*):

$$\underset{M}{\text{minimize}} \left\{ \frac{1}{2} \sum_{(i,j)\in\Omega} (z_{ij} - m_{ij})^2 + \lambda \|M\|_* \right\}, \tag{1}$$

where Ω is the set of observed entries includes all $m \cdot n$ pairs $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$, and $\|M\|_*$ is the nuclear norm, or the sum of the singular values of M. λ is a tuning parameter that must be chosen from the data. *Spectral regularization* allows for solutions $\hat{\mathbf{Z}}$ that do not fit the observed entries exactly, thereby reducing potential overfitting in the case of noise entries.

Define

$$[P_\Omega(\mathbf{Z})]_{ij} = \begin{cases} z_{ij} & \text{if } (i,j) \in \Omega \\ 0 & \text{if } (i,j) \notin \Omega \end{cases}, \quad P_\Omega^\perp(\mathbf{Z}^{old}) = \mathbf{Z}^{old} - P_\Omega(\mathbf{Z}^{old})$$

Given the singular value decomposition $\mathbf{W} = \mathbf{U}\mathbf{D}\mathbf{V}^T$ of a rank r matrix $\mathbf{W}$, we define its soft-thresholded version as

$$S_\lambda \equiv \mathbf{U}\mathbf{D}_\lambda\mathbf{V}^T \quad \text{where} \quad \mathbf{D}_\lambda = \text{diag}[(d_1 - \lambda)_+, \ldots, (d_r - \lambda)_+] \tag{2}$$

(1) could be solved by using this operator and the following algorithm.

1. Initialize $\mathbf{Z}^{old} = \mathbf{0}$ and create a decreasing grid $\lambda_1 > \cdots > \lambda_K$.
2. For each $k = 1, \ldots, K$, set $\lambda = \lambda_k$ and iterate until convergence:

   Compute $\hat{\mathbf{Z}} \leftarrow \mathbf{S}_\lambda(P_\Omega(\mathbf{Z}) + P_\Omega^\perp(\mathbf{Z}^{old}))$
   Update $\mathbf{Z}^{old} \leftarrow \hat{\mathbf{Z}}_\lambda$
3. Output the sequence of solutions $\hat{\mathbf{Z}}_{\lambda_1}, \ldots, \hat{\mathbf{Z}}_{\lambda_k}$

It has been showed by Mazumder et al.(2010) that the sequence Z_λ^k generated via *soft-impute* conveges aysmptotically, i.e., as $k \to \infty$ to a minimizer of the objective function $f_\lambda(\mathbf{Z}) \equiv \frac{1}{2} \sum\limits_{(i,j)\in\Omega} (z_{ij} - m_{ij})^2 + \lambda\|M\|_*$

2 Various SVD methods/Implementations

2.1 Implement Soft-Impute in C via LAPACK.

The code that implement *Soft-Impute* in C is attached.
There are two command line arguments which you need to input. The first one is the data file. Here, the first line of the input data file is the dimension of the data, and the rest is dataset arranged by column (just like the format of $lena256$). The second argument is λ, the tuning parameter. And the result we get by C is the same as the result we conduct soft-impute in R via R internal svd function.

2.2 Soft-impute Implementation in R and Comparisons

2.2.1 Implement Soft-Impute in R

We implement soft-impute in R. There are 5 methods to conduct singular value decomposition algorithms.

- If $svd.method = 1$, we use R internal *svd* function.
- If $svd.method = 2$, we use *irlba* function in irlba package.
- If $svd.method = 3$, we use *propack.svd* function in svd package.
- If $svd.method = 4$, we use *svd* function in RcppArmadillo.
- If $svd.method = 5$, we use *svd_econ* function in RcppArmadillo.

2.2.2 Compare Various Singular Value Decomposition Algorithms in R

The SVD provides a solution to the rank-r matrix approximation problem. Suppose $r \leq rank(Z)$, and let D_r be a diagonal matrix with all but the first r diagonal entries of the diagonal matrix D set to zero. We denote $\hat{\mathbf{Z}}_r = UD_rV^T$ as the rank-r SVD of $\mathbf{Z}$.

Then we design simulation experiments to compare these five SVD algorithms' performances. In this section, we present the results of tests that was performed on the five methods of different matrices without missing values and all svd functions were asked to return full rank (except for *irlba* since it cannot be implemented for full rank, we set full rank minus one singular values to estimate the approximate speed.) Also, we give the comparision the performance except RcppArmadillo on rank-r SVD by setting rank(U)=rank(V) equals the full rank minus r.

In this section, we compare their speeds in three cases: small matrix(20×20), middle matrix(80×80) and large matrix(300×300).

1. For small size matrix $A_{20\times 20}$, if we conduct svd implementations with full rank, by comparing their speeds, we can see that *RcppArmadillo* is the fastest among these five methods. If we set r=5, R internal svd perform best among the methods that except RcppArmadillo. Because the two methods in RcppArmadillo are required rank(U)=rank(A), we won't discuss them in rank-r case.
(In each table, The unit of time consuming is $10^{-8}s$, '*' means the roughly time of *irlba*)

methods	R internal svd	propack.svd	*armasvd*	*armasvd_econ*	*irlba*
rank(U)=rank(V)=20	257.186	437.833	192.406	206.593	889.8563($*$)
rank(U)=rank(V)=5	220.378	385.498	–	–	344.666

2. For a middle size matrix $B_{80\times 80}$, if we conduct svd implementations with full rank, comparisons of speed among these methods are as follows : *armasvd_econ* > *armasvd* > *svd* > *propack.svd.*

   When we set rank of U and V as 15 for instance. The speeds among svd, propack.svd, irlba are *irlba* > *svd* > *propack.svd.*

methods	R internal svd	propack.svd	*armasvd*	*armasvd_econ*	*irlba*
rank(U)=rank(V)=80	3.551×10^5	9.805×10^5	2.367×10^5	2.287×10^5	$6.86 \times 10^5(*)$
rank(U)=rank(V)=15	2206.698	7477.954	–	–	964.002

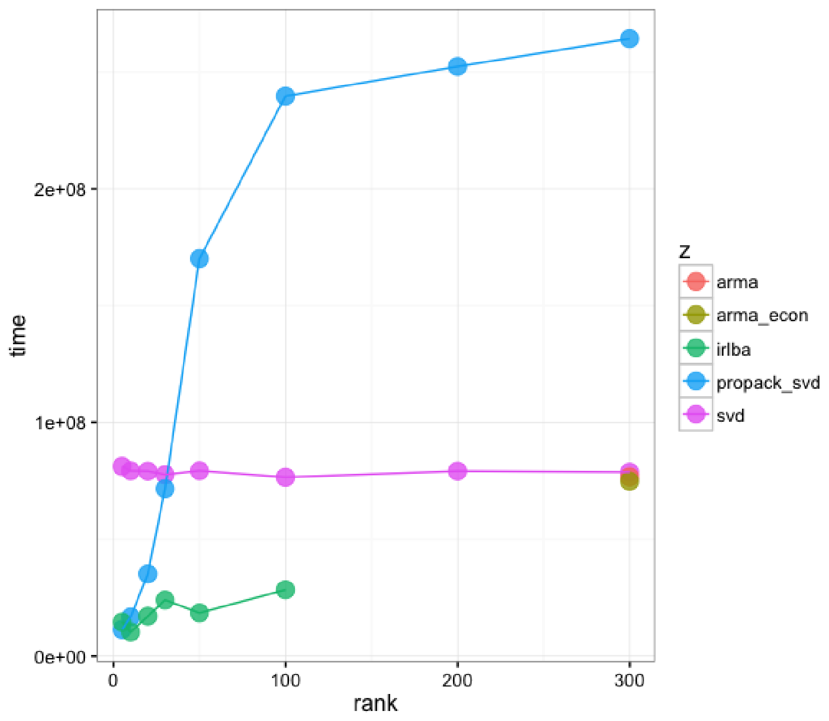
3. For large matrix $C_{300\times 300}$, if we conduct svd implementations with full rank, comparisons of speed among these five methods are as follows : *armasvd* $\geq$ *armasvd_econ* > *svd* > *propack.svd*(irlba method cannot get approximate full rank svd for large matrix).

   When the rank of U and V as 30, the speeds among svd, propack.svd, irlba are *irlba* > *svd* > *propack.svd.* When the rank of U is 15, the speeds among svd, propack.svd, irlba are *irlba* > *propack.svd* > *svd.*

methods	R internal svd	propack.svd	*armasvd*	*armasvd_econ*	*irlba*
rank(U)=rank(V)=300	9.289×10^6	4.303×10^7	8.959×10^6	8.988×10^6	–
rank(U)=rank(V)=30	8.098×10^6	9.972×10^6	–	–	2.745×10^6
rank(U)=rank(V)=15	8.057×10^6	2.515×10^6	–	–	1.564×10^6

In conclusion,

Figure 1: *Time Against Rank* (300 × 300 Matrix)



i. The speed of each method would rapid decrease as the matrices increase in size.

ii. For full rank case, the *svd* and *svd_econ* function in RcppArmadillo are the fastest svd method, their speeds are quite close; R internal svd function follows; then are the propack.svd function, and *irlba*. But as the matrix gets bigger, *irlba* becomes faster than the propack.svd.

iii. For rank-r SVD cases, the larger the matrix is, the relatively faster the irlba method is. And irlba could perform faster with the lower rank of the matrix.

2.2.3 More Discussion about Matrix Completion

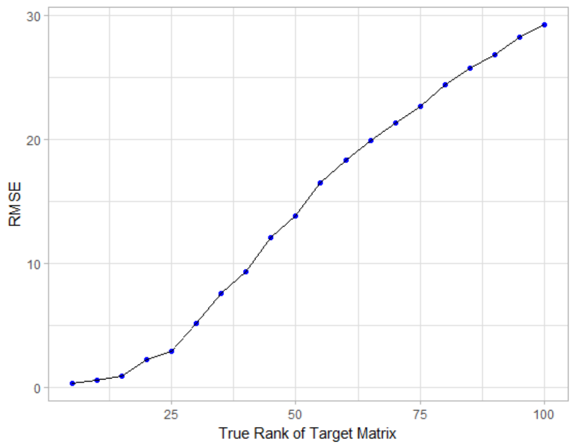
Does the true rank of target matrix have an effect on RMSE?

To find if the true rank of our target matrix influences the error of matrix completion using SoftImpute method, we try to simulate random matrix of various rank as our target matrix. For each matrix we randomly sample 20% of elements as missing data. These 20% of data would be used to calculate the RMSE to show the performance of matrix completion procedure. Among the rest 80% of elements, 75% of them are used as training set and 25% are used as validation set. The procedures we went through here are exactly like we did in the application "Lena". λ grid is $0, 10, \cdots, 180, 190, 200$.

We generate random matrix $M_{150\times100}$ of at most rank k using 2 random matrices by $M_{150\times100} = U_{150\times k} \cdot V_{k\times100}$.

Here Figure 2 is a plot of RMSE against the true rank of target matrix. We can find that as the ture rank increase, the RMSE would increase. Here we use the *armadillo* as the computation tool for efficiency.

Figure 2: RMSE vs True Rank of Target Matrix



3 Application

In this section, we will apply *Soft-Impute* to perform matrix completions.

3.1 Lena

Armadillo as the implemention tool to reconstructed the image using the remaining 60% pixels. Figure 3 is the original Lena image. Then we choose 40% of the pixels randomly as missing pixels, and plot the incomplete figure as Figure 4.

To select the tuning parameter λ, we design a grid of λ from 500 to 0 and seperate the remaining dataset further into two subsets as training set and validation set. Then we use the validation set to perform hold-out validation for the selection of λ from the grid. The best λ we selected is 135 with the validation RMSE 22.99827. The test RMSE is 19.47053 by comparing the reconstructed pixels and the original pixels at the missing locations. Figure 5 shows the λ validation RMSE against the grid of λ and Figure 6 is the reconstructed image by choosing the $\lambda = 135$. Note that the rank of the original image is 254, and the rank of the reconstructed image is 95. Here we use Armardillo as the computation tool for efficiency.

Figure 3: Original Picture

Figure 4: Picure with 40% Missing Pixels

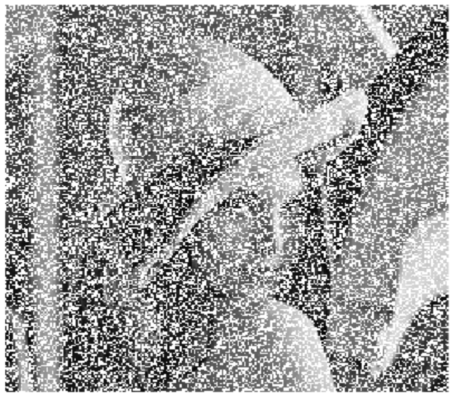


Figure 5:

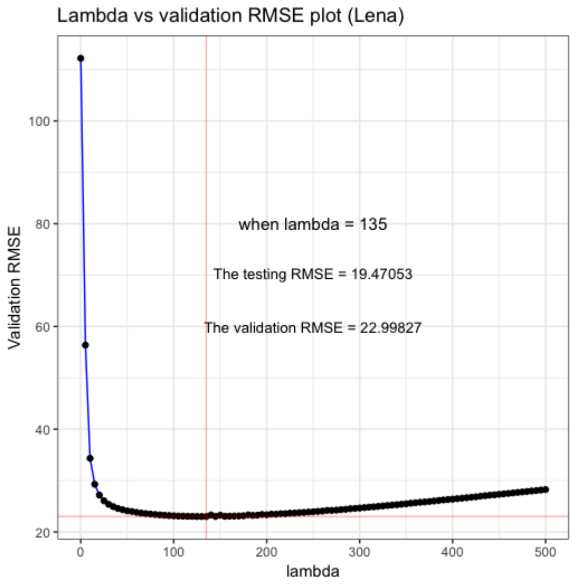


Figure 6: Picure reconstructed

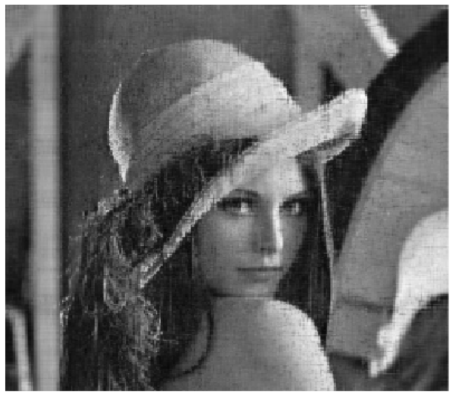


Table 1: The rank of the reconstructed image (lena)

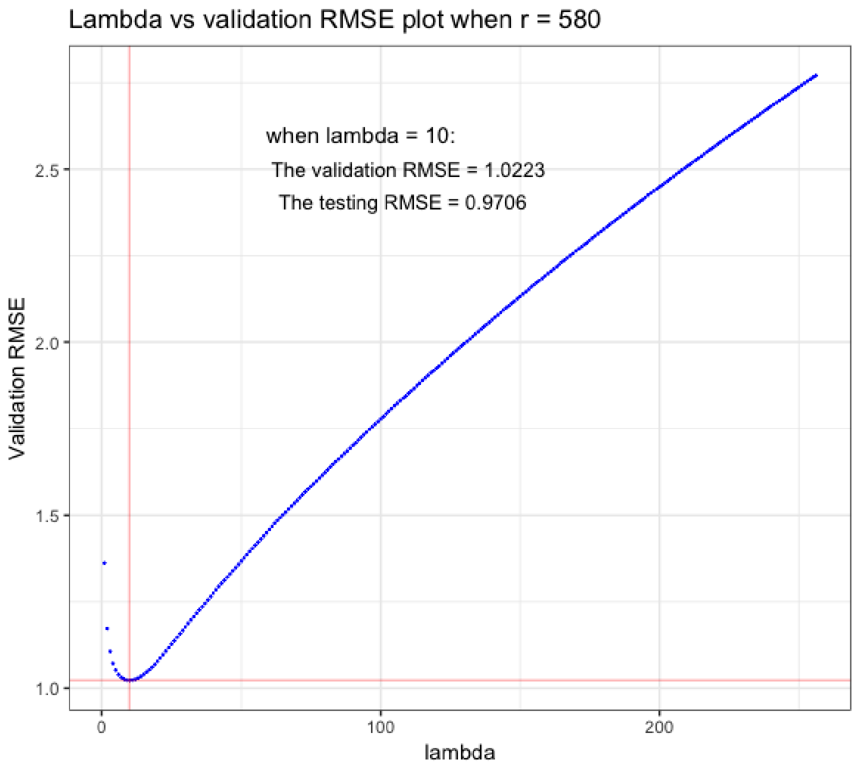
Figure	Original	Reconstructed
rank	254	95

3.2 100K

In this section, we will apply *Soft-Impute* with various implementations of SVD to the MovieLens 100K dataset. We randomly choose 70% of the dataset as the training set, and 15% as validation set, then the ramaining part as the test set.

- First we design a grid of λ (256, 255, ..., 1) and see a roughly results by using R internal SVD method. Figure 7 shows the RMSE against λ. From the plot

Figure 7: RMSE against λ (*R internal SVD, tol*=10^{-5})



  we can see the the best λ falls into the interval (6,14).

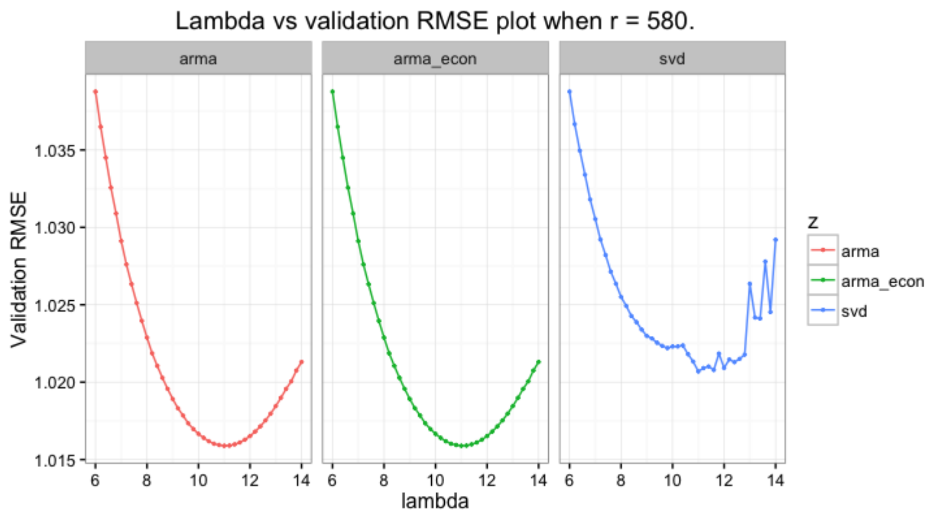
- Then we shrink the range of the grid of λ to (14, 13.8, 13.6, ..., 6.2, 6). Then apply R internal **svd** function and *RCppArmadillo* to select the best λ and compare their performance.

Table 2: Compare the Speeds with Various Methods (Movie)

Methods	R internal svd (tol=10^{-6})	R internal svd (tol=10^{-5})	armasvd	armarsvd.econ
λ	11.4	11	11	11
validation RMSE	1.0202	1.0207	1.0159	1.0159
test RMSE	0.978	0.9721	0.9721	0.9721
time (in hours)	10.23	9.40	7.07	5.20

All of these three methods give the same best $\lambda = 11$ and test RMSE if we set tolerance as 10^{-5}. From the table above, we can find that *armasvd.econ* is the fastest one among these methods. The best choice of λ would be slightly different when we set diffrent tolerance. Figure 8 compares the validation RMSE against the λ by using three methods.

Figure 8: RMSE against λ (*Comparison*)



4 (Bonus) Accelerating Soft-Impute

To compare the speed of the two versions of *Soft-Impute*: original and the Accelerated PG version, we use the data "Lena" and choose one set of decreasing λ grid, $200, 190, \cdots, 110, 100$. We only intend to compare the speed of *Soft-Impute* versions, therefore we randomly choose 40% of data as missing elements and use the other data to conduct only the two *Soft-Impute* algorithms without the validation and test procedures in application part. The svd method we use here is RcppArmadillo.

The time results we obtain using microbenchmark are as follow.

Table 3: Time for Accelerated PG SoftImpute(Seconds)

Algorithm	min	mean	median	max
APG SoftImpute	4.439280	4.719863	4.672192	5.260436
Original SoftImpute	5.031398	5.199136	5.380307	5.953136

It can be noticed that the Accelerated PG version of SoptImpute algorithm is indeed faster then the original version. The median of time for APG algorithm is about 87% of the original version.

5 Appendix–Codes

2.1 Soft-Impute in C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define EPS 0.001

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);


int main(int argc, char *argv[]){
FILE *f;

if (argc != 3){
printf("Error in Format.\n");
printf(" data: data file\n");
printf(" lambda: a tuning parameter\n");
return 1;
}

/* open the file */
if((f = fopen(argv[1], "r")) == NULL){
printf("The data file cannot be opened.\n");
return 1;
}

/* get dimensions of the data, set number of rows by N, columns by P */
int N, P;
fscanf(f, "%d", &N);
fscanf(f, "%d\n", &P);

/* read the data */
int i=0,j,k;
double X[N*P];
while( fscanf(f, "%lf", X+i)==1 ){
i++;
}

/* get lambda to conduct soft-impute */
char *lambda1 = argv[2];
```

```
double lambda = atof(lambda1);

/* set initial Zold */
double Zold[N*P], Zpro[N*P], Zlam[N*P], Znew[N*P];
for(i=0; i<N*P; i++){
Zold[i] = 0;
Zpro[i] = 0;
}

double ratio = 1, Fdiff, Fold;
char *jobu;
char *jobvt;
jobu = "A";
jobvt ="A";
int m = N, n=P, lwork=5*N, info;
double s[P], u[N*P], vt[P*P], work[lwork];

while(ratio > EPS){
/* get the orthogonal projection matrix on missing value */
for(i=0; i<N*P; i++){
if(fabs(X[i])< 1.0E-6){
Zpro[i] = Zold[i];
}
}

/* add two projection matrix together */
for(i=0; i<N*P; i++){
Zlam[i] = X[i] +Zpro[i];
}

/* get the SVD of Zlam */
dgesvd_(jobu, jobvt, &m, &n, Zlam, &m, s, u, &m, vt, &n, work, &lwork, &info);


for(i=0; i<P; i++){
if(s[i]-lambda > 1.0E-6){
s[i] = s[i] -lambda;
}
else{
s[i] = 0;
}
}


/* the principal component scores U(S-lambda)  */
for (j=0; j<P; j++){
for (i=0; i<N; i++){
u[j*N+i] = u[j*N+i] * s[j];
}
}

/* calculate U%*%S%*%VT  */
for(i=0; i<N*P; i++){
Znew[i] = 0;
}
for (j=0; j<P; j++){
for (i=0; i<N; i++){
for(k=0; k<P; k++){
Znew[j*N+i] += u[k*N+i] * vt[j*P+k];
}
}
}

/* get the squared F-norm of the difference between Znew and Zold */
Fdiff = 0, Fold = 0;
for(i=0; i<N*P; i++){
Fdiff += (Znew[i]-Zold[i])*(Znew[i]-Zold[i]);
Fold += Zold[i]*Zold[i];
}
ratio = Fdiff/Fold;

/* replace Zold by Znew */
for(i=0; i<N*P; i++){
Zold[i] = Znew[i];
}
}

for(i=0; i<N; i++){
for(j=0; j<P; j++){
printf("%lf\t", Znew[i*P+j]);
}
printf("\n");
}

return 0;
}
```

2.2

Soft-Impute in R(without parallel computing)

```
library(Rcpp)
library(RcppArmadillo)

code_full <- 'List armasvd(arma::mat & X){
  List out;
  arma::mat U, V;
  arma::vec D;
  arma::svd(U, D, V, X);
  out["u"] = U;
  out["d"] = D;
  out["v"] = V;
  return out;
}'

cppFunction(code=code_full, depends="RcppArmadillo")

code_econ <- 'List armasvd_econ(arma::mat & X){
  List out;
  arma::mat U, V;
  arma::vec D;
  arma::svd_econ(U, D, V, X);
  out["u"] = U;
  out["d"] = D;
  out["v"] = V;
  return out;
}'

cppFunction(code=code_econ, depends="RcppArmadillo")



SoftImpute.np <- function(Z, Lambda, r = min(ncol(Z),nrow(Z)), e = 0.001,
                          svd.method = c(1,2,3,4,5), empty_is_na = TRUE){
  Soft_thres<-function(x,lambda) {
    do.call( what = c,  args = lapply(x-lambda, function(x){max(x,0)}) )
  }
  Soft <- function(X, r, lambda, svd.method = c(1, 2, 3, 4, 5)){
    if (svd.method == 1){
      s <- svd(X, nu = r, nv = r)
    }
    if (svd.method == 2){
      s <- irlba::irlba(X, nu = r, nv = r)
    }
    if (svd.method == 3){
      s <- svd::propack.svd(X, neig = r)
      r <- length(s$d)
    }
    if (svd.method == 4){
      s <- armasvd(X)
      s$d <- as.vector(s$d)
      s$u <- s$u[,1:r]
      s$v <- s$v[,1:r]
    }
    if (svd.method == 5){
      s <- armasvd_econ(X)
      s$d <- as.vector(s$d)
      s$u <- s$u[,1:r]
      s$v <- s$v[,1:r]
    }
    U <- s$u
    V <- s$v
    D_lambda <- Soft_thres(s$d[1:r], lambda)
    X_lambda <- U %*% diag(D_lambda, nrow=r) %*% t(V)
    return(X_lambda)
  }

  if (empty_is_na){
  P_obs1 <- function(X, Z){
    X[is.na(Z)] <- 0
    return(X)
    }
  }
  else
    {
    P_obs1 <- function(X, Z){
      X[Z==0] <- 0
      return(X)
      }
    }
  m <- nrow(Z)
  n <- ncol(Z)
  Z_old <- matrix(0,nrow = m, ncol = n)
  k <- length(Lambda)
  Z_hat <- list()
  Z_orig <- P_obs1(Z,Z)
  for (i in 1:k){
    lambda <- Lambda[i]
```

```
    while (1){
      Z_new <- Soft( Z_orig-P_obs1(Z_old, Z)+Z_old , r , lambda , svd.method)
      if (norm(Z_new-Z_old, type = "F")/norm(Z_old, type = "F") < e)
        break;
      Z_old <- Z_new
    }
  Z_hat <- c(Z_hat, list(Z_new))
  }
  return(Z_hat)
}
```

Soft-Impute in R(with parallel computing)

```
SoftImpute.para <- function(Z, Lambda, r = min(ncol(Z),nrow(Z)), e = 0.001,
                       svd.method = c(1,2,3), empty_is_na = TRUE){
######Set parallel computing
  library(doParallel)
  cores <- detectCores() - 1
  cl <- makeCluster(cores)
  registerDoParallel(cl)

########## (Function) SVD Algorithm & Soft-thresholded SVD
  Soft_thres<-function(x,lambda) {
    do.call( what = c,  args = lapply(x-lambda, function(x){max(x,0)}) )
  }


  Soft <- function(X, r, lambda, method = c(1, 2, 3, 4)){
    if (method == 1){
      s <- svd(X, nu = r, nv = r)
    }
    if (method == 2){
      s <- irlba::irlba(X, nu = r, nv = r)
    }
    if (method == 3){
      s <- svd::propack.svd(X, neig = r)
    }
    U <- s$u
    V <- s$v
    D_lambda <- Soft_thres(s$d[1:r], lambda)
    X_lambda <- U %*% diag(D_lambda, nrow=r) %*% t(V)
    return(X_lambda)
  }
########################################################
```

```
###### (Function) Assign the matrix element to 0 at the missing positions.
  if (empty_is_na){
    P_obs1 <- function(X, Z){
      X[is.na(Z)] <- 0
      return(X)
    }
  }
  else
  {
    P_obs1 <- function(X, Z){
      X[Z==0] <- 0
      return(X)
    }
  }
##########################################################################



###### Soft-Impute Algorithm
  m <- nrow(Z)
  n <- ncol(Z)
  k <- length(Lambda)
  Z_old <- matrix(0,nrow = m, ncol = n)
  Z_orig <- P_obs1(Z,Z)


  Z_hat <- foreach(i = 1:k) %dopar%{
    lambda <- Lambda[i]
    while (1){
      Z_new <- Soft( Z_orig-P_obs1(Z_old, Z)+Z_old , r , lambda ,
                 method = svd.method)
      if (norm(Z_new-Z_old, type = "F")/norm(Z_old, type = "F") < e)
        break;
      Z_old <- Z_new
    }
    return(Z_new)
  }
############################


######End Parallel Computing
  stopCluster(cl)
###########################

  return(Z_hat)
```

```
}
```

Compare SVD algorithms

```
######For testing speed
set.seed(580580)
library(microbenchmark)

## small matrix
A1 <- matrix(rep(0,400), ncol = 20)
for (i in 1:5){
  a <- rnorm(20)
  b <- rnorm(20)
  A1 <- A1 + a%*%t(b)
}

## armasvd  <= armasvd_econ <= svd < propack.svd < irlba  (full rank)
microbenchmark( svd(A1,nu=20,nv=20),
                svd::propack.svd(A1, neig=20),
                armasvd(A1),
                armasvd_econ(A1),
                irlba::irlba(A1,nu=19, nv=19)
                )

#svd <= irlba << propack.svd
microbenchmark( svd(A1,nu=5,nv=5),
                svd::propack.svd(A1, neig=5),
                irlba::irlba(A1,nu=5, nv=5)
)


## middle matrix
A2 <- matrix(rep(0,6400), ncol = 80)
for (i in 1:15){
  a <- rnorm(80)
  b <- rnorm(80)
  A2 <- A2 + a%*%t(b)
}

## armasvd_econ  <= armasvd <= svd << irlba << propack.svd (full rank)
microbenchmark( svd(A2,nu=80,nv=80),
                svd::propack.svd(A2, neig=80),
                armasvd(A2),
                armasvd_econ(A2),
                irlba::irlba(A2,nu=79,nv=79))
```

```
## irlba << svd << propack.svd
microbenchmark( svd(A2,nu=15,nv=15),
                svd::propack.svd(A2, neig=15),
                irlba::irlba(A2,nu=15,nv=15))


## large matrix
A3 <- matrix(rep(0,90000), ncol = 300)
for (i in 1:60){
  a <- rnorm(300)
  b <- rnorm(300)
  A3 <- A3 + a%*%t(b)
}

## armasvd <= armasvd_econ  <= svd << propack.svd (full rank)
microbenchmark( svd(A3,nu=300,nv=300),
                svd::propack.svd(A3, neig=300),
                armasvd(A3),
                armasvd_econ(A3))

## irlba << svd <= propack.svd
microbenchmark( svd(A3,nu=30,nv=30),
                svd::propack.svd(A3, neig=30),
                irlba::irlba(A3,nu=30,nv=30))

## irlba < propack.svd < svd
microbenchmark( svd(A3,nu=15,nv=15),
                svd::propack.svd(A3, neig=15),
                irlba::irlba(A3,nu=15,nv=15))

## irlba <= propack.svd << svd
microbenchmark( svd(A3,nu=7,nv=7),
                svd::propack.svd(A3, neig=7),
                irlba::irlba(A3,nu=7,nv=7))
### svd
svd_speed <- microbenchmark( svd(A3,nu=5,nv=5),
                             svd(A3,nu=10,nv=10),
                svd(A3,nu=20,nv=20),
                svd(A3,nu=30,nv=30),
                svd(A3,nu=50,nv=50),
                svd(A3,nu=100,nv=100),
                svd(A3,nu=200,nv=200),
                svd(A3,nu=300,nv=300))
```

```
svd_speed_mean <- tapply(svd_speed$time, svd_speed$expr,mean)
names(svd_speed_mean)<-c(5,10,20,30,50,100,200,300)

### propack_svd
propack_speed <- microbenchmark( svd::propack.svd(A3, neig=5),
                                 svd::propack.svd(A3, neig=10),
                             svd::propack.svd(A3, neig=20),
                             svd::propack.svd(A3, neig=30),
                             svd::propack.svd(A3, neig=50),
                             svd::propack.svd(A3, neig=100),
                             svd::propack.svd(A3, neig=200),
                             svd::propack.svd(A3, neig=300))

propack_speed_mean <- tapply(propack_speed$time, propack_speed$expr,mean)
names(propack_speed_mean)<-c(5,10,20,30,50,100,200,300)

### irlba
irlba_speed <- microbenchmark(irlba::irlba(A3,nu=5,nv=5),
                                irlba::irlba(A3,nu=10,nv=10),
                                irlba::irlba(A3,nu=20,nv=20),
                              irlba::irlba(A3,nu=30,nv=30),
                                irlba::irlba(A3,nu=50,nv=50),
                                irlba::irlba(A3,nu=100,nv=100)
                                )

irlba_speed_mean <- tapply(irlba_speed$time, irlba_speed$expr,mean)
names(irlba_speed_mean)<-c(5,10,20,30,50,100)

### arma
arma_speed <- microbenchmark(armasvd(A3))
arma_speed_mean <- mean(arma_speed[,2])
names(arma_speed_mean)<-300

### arma_econ
arma_econ_speed <- microbenchmark(armasvd_econ(A3))
arma_econ_speed_mean <- mean(arma_econ_speed[,2])
names(arma_econ_speed_mean)<-300



x <- c(c(5,10,20,30,50,100,200,300),c(5,10,20,30,50,100,200,300),
        c(5,10,20,30,50,100),300,300)
z <- as.factor(  c(  rep("svd",8),rep("propack_svd",8),
        rep("irlba",6),"arma","arma_econ"  )  )
y <- c(svd_speed_mean, propack_speed_mean,irlba_speed_mean,
       arma_speed_mean, arma_econ_speed_mean)
d <- data.frame(x = x, y = y, z = z)


library(ggplot2)
ggplot(data = d, aes(x = x, y = y, colour = z))+
  geom_point(size = 4, alpha = 0.8)+
  geom_line()+
  xlab("rank")+
  ylab("time")+
  theme_bw()
```

Compare RMSE with target matrices of different rank

```
### Generate random matrix of rank r (most likely)

## Simulate random matrix U and V, with elements in U and V
##follow a discrete distribution
randU <- matrix(sample(x=0:5, size=150*100,
            replace = TRUE, prob = c(0.5,rep(0.1,5))), ncol = 150)
randV <- matrix(sample(x=0:5, size=100*100,
            replace = TRUE,prob = c(0.5,rep(0.1,5))), ncol = 100)

#randU <- matrix(rnorm(150*100, mean = 1, sd = 1),ncol = 150)
#randV <- matrix(rnorm(100*100, mean = 1, sd = 1),ncol = 100)

library(Matrix)

 # rankMatrix(randZ, method = "tolNorm2")
Rseq <- seq(5, 100, 5)
R <- c()
for(k in Rseq){
  set.seed(580580)
  randZ <- randU[,1:k] %*% randV[1:k,]
  o <- L.valid(randZ, train=0.6, valid = 0.2, test = 0.2,
           l.grid = seq(0,200,10), method = 4, m.valid = 4)
  R <- c(R, o[[4]])
}

library(ggplot2)
ggplot(data.frame(rank = Rseq, RMSE = R),aes(x=rank, y = RMSE)) +
  xlab("True Rank of Target Matrix") +
  geom_point(colour = "blue") + geom_line() + theme_light()
```

3.1 Application - Lena

```
####################################################################
#lena256 data

lena <- scan( "C:\\Users\\thinkpad\\Dropbox\\STAT 580\\FinalProject\\Rcode\\lena256")
lena <- lena[-c(1,2)]
lena <- matrix(data = lena, nrow = 256)
lena <- lena[,256:1]

image(z=lena, col=gray(1:256/256), axes=FALSE)
######################################################################
# Valid & Test

eval.l <- function(Orig.M, Est.M, test){
  # test is the flag that label all the test or validation data
  RMSE <- sqrt(sum((Orig.M[test] - Est.M[test])^2)/length(test))
  return(RMSE)
}
L.valid <- function(data, train, valid, test, l.grid, method=c(1,2,3,4,5),
                    r.valid = min(ncol(data),nrow(data)), m.valid=c(1,2,3)){
  N <- length(data)
  q <- length(l.grid)
  flag <- sample(1:N)
  n.miss <- round(N * test)
  n.valid <- round(N * valid)
  n.train <- N - (n.miss + n.valid)
  f.miss <- flag[1:n.miss]
  f.valid <- flag[n.miss+1:n.valid]
  f.train <- flag[(N-n.train+1): N]
  tr.M <- data
  tr.M[c(f.miss, f.valid)] <- NA
  # tr.M is the training matrix
  Train.out <- SoftImpute.para(tr.M, l.grid, r = r.valid, svd.method = m.valid)
  Valid.RMSE <- lapply(Train.out, FUN = eval.l, Orig.M = data, test = f.valid)
  Valid.RMSE <- do.call(Valid.RMSE, what = c)
  k <- which.min(Valid.RMSE)
  lambda <- l.grid[k]
  test.M <- data
  test.M[f.miss] <- NA
#  image(z=test.M, col=gray(1:256/256), axes=FALSE)
  Out <- SoftImpute.np(test.M, lambda, svd.method = method)
#  image(Out[[1]], col=gray(1:256/256), axes=FALSE)
  RMSE <- eval.l(data, Out[[1]], f.miss)
  cat("The lambda we choose from:", l.grid, "\n")
  cat("The RMSE of Validation Set:", Valid.RMSE, "\n")
  return(list(lambda=lambda, Missing = test.M, Estimated_Matrix= Out[[1]], RMSE = RMS
}
```

3.2 Application - MovieLens

```
eval.m <- function(Orig.M, Est.M, test){
  # test is the flag that label all the test or validation data
  RMSE <- sqrt(sum((Orig.M[test] - Est.M[test])^2)/length(test))
  return(RMSE)
}

########################################
# Test and Validation
M.valid <- function(data, valid, test, l.grid, method=c(1,2,3,4,5),m.valid=c(1,2,3),
                    r.valid, r.test, e = 1e-3){

  N <- nrow(data)
  q <- length(l.grid)
  flag <- sample(1:N)
  n.test <- round(N * test)
  n.valid <- round(N * valid)
  n.train <- N - (n.test + n.valid)
  f.test <- flag[1:n.test]
  f.valid <- flag[n.test+1:n.valid]
  f.train <- flag[(N-n.train+1): N]

  data.o <- Matrix::sparseMatrix(i = data[,1], j = data[,2],
                                 x = data[,3])
  data.o <- as.matrix(data.o)
  # data.o is the original matrix

  data1 <- data
  data1[c(f.test,f.valid),3] <- 0
  data.v <- Matrix::sparseMatrix(i = data1[,1], j = data1[,2],
                      x = data1[,3])
  data.v <- as.matrix(data.v)
  # data.v is the training matrix

  data2 <- data
  data2[f.test,3] <- 0
  data.t <- Matrix::sparseMatrix(i = data2[,1], j = data2[,2],
                                 x = data2[,3])
  data.t <- as.matrix(data.t)
  # data.t is the testing matrix
  test.ij <- which(data.o != data.t)
  test_valid.ij <- which(data.o != data.v)
  valid.ij <- setdiff(test_valid.ij,test.ij)

  Train.out <- SoftImpute.para(data.v, l.grid, svd.method = m.valid,
                               r = r.valid, empty_is_na = FALSE)
  Valid.RMSE <- lapply(Train.out, FUN = eval.m, Orig.M = data.o, test = valid.ij)
  Valid.RMSE <- do.call(Valid.RMSE, what = c)
  k <- which.min(Valid.RMSE)
  lambda <- l.grid[k]


  Out <- SoftImpute.np(data.t, lambda, svd.method = method,
                       r = r.test, empty_is_na = FALSE, e = e)
  RMSE <- eval.m(data.o, Out[[1]], test.ij)
  cat("The lambda we choose from:", l.grid, "\n")
  cat("The RMSE of Validation Set:", Valid.RMSE, "\n")
  return(list(lambda=lambda, RMSE = RMSE, Estimated_Matrix= Out[[1]]))
}
```

4 Accelerated PG *Soft-Impute*

```
library(Rcpp)
library(RcppArmadillo)

code_full <- 'List armasvd(arma::mat & X){
List out;
arma::mat U, V;
arma::vec D;
arma::svd(U, D, V, X);
out["u"] = U;
out["d"] = D;
out["v"] = V;
return out;
}'

cppFunction(code=code_full, depends="RcppArmadillo")

code_econ <- 'List armasvd_econ(arma::mat & X){
List out;
arma::mat U, V;
arma::vec D;
arma::svd_econ(U, D, V, X);
out["u"] = U;
```

```
out["d"] = D;
out["v"] = V;
return out;
}'

cppFunction(code=code_econ, depends="RcppArmadillo")

SoftImpute.APG.np <- function(Z, Lambda, r = min(ncol(Z),nrow(Z)), e = 0.001,
                          svd.method = c(1,2,3,4,5), empty_is_na = TRUE){
  ####  Lambda should be a descending sequence
  Soft_thres<-function(x,lambda) {
    do.call( what = c,  args = lapply(x-lambda, function(x){max(x,0)}) )
  }
  Soft <- function(X, r, lambda, svd.method = c(1, 2, 3, 4, 5)){
    if (svd.method == 1){
      s <- svd(X, nu = r, nv = r)
    }
    if (svd.method == 2){
      s <- irlba::irlba(X, nu = r, nv = r)
    }
    if (svd.method == 3){
      s <- svd::propack.svd(X, neig = r)
      r <- length(s$d)
    }
    if (svd.method == 4){
      s <- armasvd(X)
      s$d <- as.vector(s$d)
      s$u <- s$u[,1:r]
      s$v <- s$v[,1:r]
    }
    if (svd.method == 5){
      s <- armasvd_econ(X)
      s$d <- as.vector(s$d)
      s$u <- s$u[,1:r]
      s$v <- s$v[,1:r]
    }
    U <- s$u
    V <- s$v
    D_lambda <- Soft_thres(s$d[1:r], lambda)
    X_lambda <- U %*% diag(D_lambda, nrow=r) %*% t(V)
    return(X_lambda)
  }

  if (empty_is_na){
    P_obs1 <- function(X, Z){
      X[is.na(Z)] <- 0
      return(X)
    }
  }
  else
  {
    P_obs1 <- function(X, Z){
      X[Z==0] <- 0
      return(X)
    }
  }
  m <- nrow(Z)
  n <- ncol(Z)
  Z_0 <- matrix(0,nrow = m, ncol = n)
  Z_old <- Z_0
  k <- length(Lambda)
  Z_hat <- list()
  Z_orig <- P_obs1(Z,Z)
  A <- Z_0
  for (i in 1:k){
    lambda <- Lambda[i]
    t_old <- 1
    A <- Z_0
    Z_old <- Z_0
    while (1){
      Z_new <- Soft( Z_orig-P_obs1(A, Z)+A , r , lambda , svd.method)
      t_new <- (1 + sqrt(1 + 4* t_old^2))/2
      A <- Z_new + ((t_old - 1)/t_new) * (Z_new - Z_old)
      if (norm(Z_new-Z_old, type = "F")/norm(Z_old, type = "F") < e)
        break;
      Z_old <- Z_new
    }
    Z_hat <- c(Z_hat, list(Z_new))
    Z_0 <- Z_new
  }
  return(Z_hat)
}
```