STAT 580 Final Report

Yifan Zhu, Gang Han, Lijin Zhang, Lingnan Yuan

Department of Statistics, Iowa State University

1 Summary of Soft-Impute

First we want to fit a matrix $\widehat{\mathbf{Z}}$ that approximates $\mathbf{Z}$ in a suitable sense. By doing this, we can find an approximation that has simple structure or fill in any missing entries in $\mathbf{Z}$ which is our purpose.[1]

Our general approach is to consider estimators based on optimization problem of the form:

$$\widehat{\mathbf{Z}} = \arg \min_{\mathbf{M} \in \mathbb{R}^{m \times n}} \|\mathbf{Z} - \mathbf{M}\|_F^2 \qquad \text{subject to} \qquad \Phi(\mathbf{M}) \leq c, \tag{1}$$

$\Phi(\cdot)$ is a constraint function that encourages $\hat{\mathbf{Z}}$ to be sparse in some general sense. There are some kinds of constraint methods, one is the singular value decomposition that we'll discuss later.

The SVD provides a solution to the rank-q matrix matrix approximation problem. Then our optimization problem (1) has the form:

$$\arg \min_{rank(\mathbf{M})=r} \|\mathbf{Z} - \mathbf{M}\|_F. \tag{2}$$

It has a closed form solution $\widehat{\mathbf{Z}}_r = UD_rV^T$. It's sparse in the sense that all but r singular values are 0 and it provides an effective method for matrix completion.

Suppose our observed entries of the matrix $\mathbf{Z}$ indexed by the subset $\Omega \subset 1, \cdots, m \times 1, \cdots, n$. A natural approach is to seek the lowest rank approximating matrix $\widehat{\mathbf{Z}}$. But forcing the estimate $\mathbf{M}$ to interpolate each of the observed entries z_{ij} will be too harsh and can lead to overfitting. So it would be better to allow some errors. Consider the optimization problem:

$$\arg \min_{\text{rank}(\mathbf{M}) \leq r} \sum_{(i,j) \in \Omega} (z_{ij} - m_{ij})^2. \tag{3}$$

However, the optimization problem (3) is nonconvex, the exact solutions are in general not available. The nuclear norm is a convex relaxation of the rank of a matrix, so we can give a convex relaxation with noise:

$$\arg \min_{\mathbf{M}} \left\{ \frac{1}{2} \sum_{(ij) \in \Omega} (z_{ij} - m_{ij})^2 + \lambda \|\mathbf{M}\|_* \right\}. \tag{4}$$

Algorithm 1 SOFT-IMPUTE FOR MATRIX COMPLETION[2]

```
1: Initialize $Z^{\text{old}} = 0$ and create a decreasing grid $\lambda_1 > \cdots > \lambda_n$
2: for each $k = 1, \cdots, K$, set $\lambda = \lambda_k$ and iterate until convergence: do
3:     Compute $\widehat{\mathbf{Z}}_\lambda \leftarrow S_\lambda(P_\Omega(\mathbf{Z} + P_\Omega^\perp(Z^{\text{old}})))$.
4:     Update $Z^{\text{old}} \leftarrow \widehat{\mathbf{Z}}_\lambda$
5: end for
6: Output the sequence of solutions $\widehat{\mathbf{Z}}_{\lambda_1}, \cdots, \widehat{\mathbf{Z}}_{\lambda_K}$.
```

Then we can use Soft-Impute Algorithm to solve this problem.

Define the projection operator $P_\Omega : \mathbb{R}^{m\times n} \mapsto \mathbb{R}^{m\times n}$ as follows:

$$[P_\Omega(\mathbf{Z})]_{ij} = \begin{cases} z_{ij} & \text{if} \quad (i,j) \in \Omega \\ 0 & \text{if} \quad (i,j) \notin \Omega \end{cases} \tag{5}$$

Define the soft-thresholded version of SVD as:

$$S_\lambda(\mathbf{W}) \equiv \mathbf{U}\mathbf{D}_\lambda\mathbf{V}^\mathbf{T} \quad \text{where} \quad \mathbf{D}_\lambda = \text{diag}[(d_1 - \lambda)_+, \cdots, (d_r - \lambda)_+]. \tag{6}$$

2 Comparison of SVD implementations

We've got different functions of doing soft-impute implementations using 4 svd methods. Now we want to compare which of these methods is the fastest and which of these methods is the most accurate.

First of all, we simulated several matrices using the underlying model:

$$\mathbf{Z}_{m\times n} = \mathbf{U}_{\mathbf{m\times r}}\mathbf{V}^\mathbf{T}_{\mathbf{r\times n}} + \epsilon$$

Where $\mathbf{U}$ and $\mathbf{V}$ are random matrices with standard normal Gaussian entries, and ϵ is i.i.d. Guassian. Ω is uniformly random over the indices of the matrix with p % of missing entries. Define the SNR, test error and training error as:

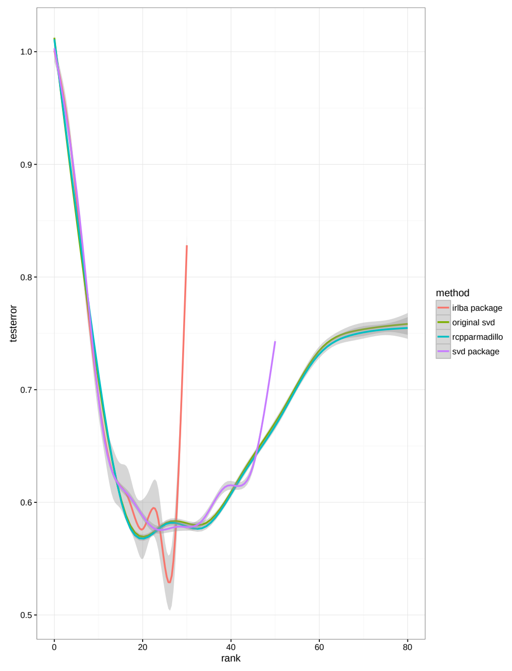
$$\text{SNR} = \sqrt{\frac{\text{var}(UV^T)}{\text{var}(\epsilon)}}$$

$$\text{Test error} = \frac{\|P_\Omega^\perp(UV^T - \widehat{Z})\|_F^2}{\|P_\Omega^\perp(UV^T)\|_F^2}$$

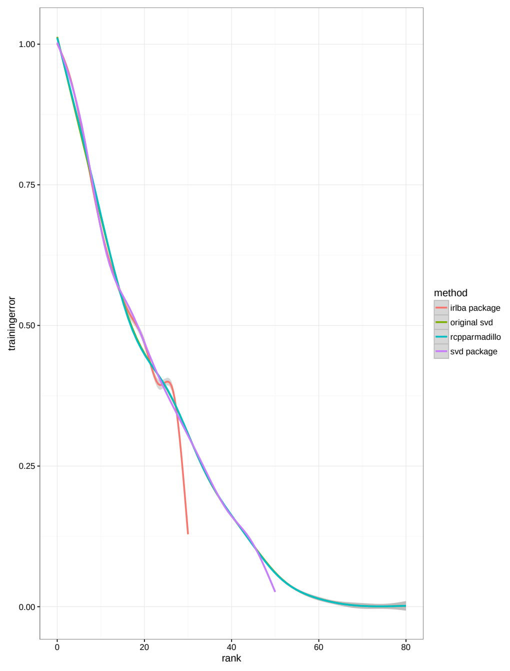
$$\text{Training error} = \frac{\|P_\Omega(Z - \widehat{Z})\|_F^2}{\|P_\Omega(Z)\|_F^2}$$

In the following figures, we show the training and test error for the 4 methods as a function of rank.

As we can see in the figures, the soft-impute using svd methods `original svd` and `rcpparmadillo` perform very similarly. While the method of `irlba package` seems to have lowest Test error at some rank, but it seems fluctuate a lot. the methods `original svd` and `rcpparmadillo`. According to the training error, the method of `irlba package` recovers a matrix with a relatively low rank which is closer to our true rank. So it has less evidence of overfitting.



(a) Test error



(b) Training error

Figure 1: Compare the test error against the rank among the 4 methods with $p = 50\%, \text{SNR} = 1, \text{true rank} = 10, m = n = 100$.

Then we tried different properties of the generated matrix. Let $p = 0.3, 0.5, 0.8; \text{SNR} = 1, 10; \text{true rank} = 5, 10, 20.$,then we get the table that compares the running time and test error among the 4 methods.

Table 1: Performance of different methods for different problem instances in terms of test error and running time

time1	time2	time3	time4	error1	error2	error3	error4	true rank	SNR	p
2.1650	2.5915	1.3096	1.7030	0.2592	0.2592	0.2592	0.2592	5	1	0.3
2.0902	2.6631	1.4791	1.6333	0.4338	0.4338	0.4339	0.4338	10	1	0.3
2.0032	2.7237	1.6506	1.5678	0.6502	0.6502	0.6506	0.6502	20	1	0.3
1.9709	1.9810	0.6514	1.5437	0.0046	0.0046	0.0046	0.0046	5	10	0.3
1.9875	1.9486	0.6748	1.5096	0.0113	0.0113	0.0111	0.0113	10	10	0.3
2.0435	1.9584	0.6629	1.5443	0.0443	0.0432	0.0312	0.0443	20	10	0.3
2.8228	2.8239	1.3798	2.2199	0.3492	0.3492	0.3492	0.3492	5	1	0.5
2.6000	2.8008	1.6049	2.0534	0.5798	0.5798	0.5798	0.5798	10	1	0.5
2.4068	2.5195	1.7681	1.8531	0.7970	0.7970	0.7970	0.7970	20	1	0.5
2.8192	2.8098	0.9243	2.2188	0.0097	0.0097	0.0097	0.0097	5	10	0.5
2.7298	2.7674	0.9625	2.1510	0.0319	0.0319	0.0306	0.0319	10	10	0.5
2.6173	2.6013	0.9362	2.0141	0.1913	0.1896	0.1488	0.1913	20	10	0.5
4.1483	4.1360	2.3177	3.2476	0.7382	0.7382	0.7382	0.7382	5	1	0.8
4.4981	4.4856	2.5795	3.5294	0.9040	0.9040	0.9040	0.9040	10	1	0.8
3.8848	3.8947	2.4525	3.0592	0.9829	0.9829	0.9829	0.9829	20	1	0.8
3.9987	3.9948	1.5107	3.1125	0.1739	0.1739	0.1746	0.1739	5	10	0.8
4.1852	4.1783	1.9447	3.2685	0.5261	0.5261	0.5259	0.5261	10	10	0.8
4.6070	4.5803	2.4109	3.6193	0.8458	0.8458	0.8458	0.8458	20	10	0.8

This table shows that the minimum test errors among the 4 methods are quite similar, but the running time of method `irlba package` is the shortest followed by `rcpparmadillo`.

According to the comparison above, we'll use the method `irlba package` in our application part, because it's the fastest one and in the figure it turns out to have minimum test error. We'll also use `rcpparmadillo`, because it's also very fast and is more stable in the figure.

3 Application

3.1 Lena

Based the comparison between these 4 implementations, we choose the `original svd` in R to do the image imputing. The reason is that from Figure 1 we can see the `original svd` and `rcpparmadillo` seems to be more steady than `irlba package` (the gray region that represents confidence interval is smaller). Also these two methods seems to have smaller test error around the true matrix rank. Because the data for lena is not large, thus we do not care about the speed of implementations. And we can expect `original svd` and `rcpparmadillo` would give almost the same results from the comparison part, hence we choose to use `orginal svd`.

First we randomly removed the 40% of the lena and got an image with only 60% being observed. Now we use the 60% as our data to restore the image with soft-impute. In order to obtain an optimal restoration, we need to tune the parameter λ in the soft-impute algorithm. The strategy we took is to further divide the data into two parts: 70% as training data and 30% as validation data. We did the imputation with a series of λ's (we used $\lambda = 500, 490, \ldots, 10$). Then we compared the corresponding validation part of these imputed matrix with the true validation set. The error term we used is

$$\frac{\|P_{\Omega_{\text{validation}}}(Z_{\text{true}} - \hat{Z}_{\lambda})\|_F^2}{\|P_{\Omega_{\text{validation}}} Z_{\text{true}}\|_F^2} \tag{7}$$

where $\Omega_{\text{validation}}$ is the set of matrix entries that validation set has.

We chose the λ that gave us the smallest test error and applied soft-impute again on the whole data set with the optimal λ we got. In this case, with the test error we defined, the optimal λ we chose is $\lambda = 190$.

After we imputed the matrix, we wanted to see how close it is to the true one. We looked at the rank of recovered matrix and the true matrix, and also the error of the imputed one compared with the true one. The error we used is

$$\frac{\|P_{\Omega_{\text{test}}}(Z_{\text{true}} - \hat{Z}_{\lambda})\|_F^2}{\|P_{\Omega_{\text{test}}}(Z_{\text{true}})\|_F^2} \tag{8}$$

where Ω_{test} is the set of matrix entries that test set has, which is the rest 40% of the true lena image that is not taken as data.

With the optimal λ, the rank of recovered matrix is

$$\text{rank}(\hat{Z}_{\lambda_{\text{opt}}}) = 77$$

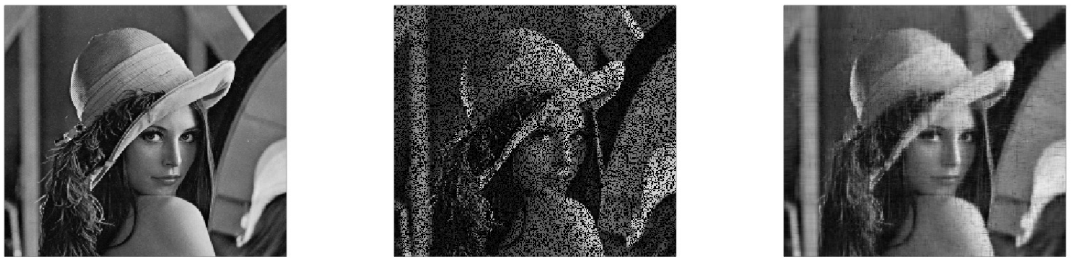
While the true rank is

$$\text{rank}(Z_{\text{true}}) = 254$$

The test error of the imputed matrix when compared with the true matrix in the rest 40% of the image is

$$\text{test error} = 0.031$$

Finally, we compare the true image, incomplete image and the restored image visually, which is shown in Figure 2.



(a) true image (b) incomplete image (c) restored image

Figure 2: Visually comparing the imputed image with the true one

We can see that the restored image is not so good. It is not like a natural image with good smoothness. Also the restored image has a much smaller rank than the true one. However, soft-impute is intended to impute low-rank matrix, and true image actually has a quite big rank. That might also be a reason why the final result is not good.

3.2 MovieLens

In this section, we use MovieLen 100K Dataset to apply our soft-impute algorithm. As the data set is not complete when we convert the original data into a matrix form, we need to divide it into three parts: 70% as training set, 15% as validation set and 15% as test set. What we did is similar as we did in lena part. We first use the training set as observed entries to impute the matrix, then we compare the validation set with the corresponding entries of the imputed matrix. Again we use (7) as the error term and we pick the λ with the smallest validation error. After we obtained the optimal λ, we use the 70% training set and 15% validation set combined as our data, or observed entries to impute the matrix. Then we compare the test set with the corresponding entries of the imputed matrix and calculate the test error using (8). Besides, we calculated the RMSE of the imputed matrix with respect to the test set in corresponding parts. The RMSE formula is

$$\text{RMSE} = \sqrt{\frac{\|P_{\Omega_{\text{test}}}(Z_{\text{true}} - \hat{Z}_{\lambda_{\text{opt}}})\|_F^2}{|\Omega_{\text{test}}|}} \tag{9}$$

As the dataset is pretty big, we chose `irlba package` and `rcpparmadillo` as two implementations to be applied to this dataset. There two implementations are relatively faster than others ,thus with the big dataset we could save a lot of time.

In choosing the optimal λ, as the big matrix takes time to impute, we used a strategy to find the optimal one using 3 grids of λ's by assuming the test error as a function of λ is unimodal. The first grid is $\lambda = 100, 90, \ldots, 10$ with step size 10. From this grid we can find the interval where the optimal λ located. We found $\lambda = 10$ was the smallest in this grid. Then we knew the optimal λ is between 20 and 0. Then we used the second grid $\lambda = 19, 16, \ldots, 4, 1$ with step size 3. We found $\lambda = 13$ is the smallest in this grid. Then we tried the third grid $\lambda = 15, 14, 12, 11$. In this grid, $\lambda = 12$ gave us the smallest test error. But comparing it with the test error when $\lambda = 13$, we found the test error for $\lambda = 13$ was smaller. Hence, the optimal λ we chose is $\lambda = 13$.

The time it took in each λ grid, the best λ for that grid and the corresponding validation error for that best λ for both methods are given below.

Table 2: Time, best λ, validation error for two methods in three grids

	method	time	best λ	validation error
1_{st} grid	`irlba package`	6278.067	10	0.07842399
	`rcpparmadillo`	43121.128	10	0.07840086
2_{nd} grid	`irlba package`	12487.161	13	0.07705321
	`rcpparmadillo`	35947.462	13	0.07707266
3_{rd} grid	`irlba package`	3673.131	12	0.07710168
	`rcpparmadillo`	20142.322	12	0.07715054

From Table 2, we can see the method `irlba  package` is a lot faster. However the best λ and validation error they got for each grid are almost the same.

Using the optimal $\lambda = 13$, we obtained the imputed matrix and compare the corresponding test set entries in the matrix with test set. The test error and RMSE we got is shown below.

Table 3: Test error and RMSE with respect to two methods

	optimal λ	test error	RMSE
`irlba package`	13	0.07705321	1.0283
`rcpparmadillo`	13	0.07707266	1.0284

The two methods gave almost the same test error and RMSE, thus the major difference between is the speed. In the case of large dataset, `irlba package` is much more efficient.

References

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

[2] 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.

A Appendix: Codes for final project

C LAPACK soft-impute implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>


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 min(int m, int n){
    if (m<n)
        return m;
    else
        return n;
}

void diag(double *s, int r, double *ds){
    int i,j;
    for (i = 0; i < r; i++){
        for (j = 0; j < r; j++){
            if (i == j)
                ds[i + j*r] = s[i];
            else
                ds[i + j*r] = 0;
        }
    }
}

double fnormsq(double *z, int m, int n){
    int i,j;
    double sum = 0;
    for (i=0; i< m; i++){
        for (j = 0; j < n; j++){
            sum = sum + z[i + j*m]*z[i + j*m];
        }
    }
    return sum;
}

double diff(double *zold, double *znew, int m, int n){
    double zold_minus_znew[m*n];
    int i,j;
    for (i = 0; i < m; i++){
        for (j = 0; j < m; j++){
            zold_minus_znew[i+j*m] = zold[i+j*m] - znew[i+j*m];
        }
    }
    return fnormsq(zold_minus_znew, m,n)/fnormsq(zold,m,n);
}
```

```
void soft_threshold(double *s, double lambda, int r){
    int i;
    for (i=0; i<r; i++){
        double s_minus_lambda = s[i] - lambda;
        if (s_minus_lambda < 0)
            s[i] = 0;
        else
            s[i] = s[i] - lambda;
    }
}

void P_omega_c(double *zold, double *P_omega_c_zold, double *omega, int
    m, int n){
    int i;
    for (i= 0; i < m*n; i++)
        P_omega_c_zold[i] = (1 - omega[i]) * zold[i];
}

void get_znew(double *znew, double *z, double *zold, double *omega,
    double lambda, int m, int n){
    double P_Z_plus_P_c_Z_old[m*n];
    double P_c_Z_old[m*n];
    P_omega_c(zold, P_c_Z_old, omega, m, n);
    int i,j;
    for (i = 0; i<m; i++){
        for (j = 0; j<n; j++){
            P_Z_plus_P_c_Z_old[i+j*m] = z[i+j*m] + P_c_Z_old[i+j*m];
        }
    }

    int r;
    r = min(m,n);

    // svd
    char jobu = 'S';
    char jobvt = 'S';
    //int m = m;
    //int n = n;
    double *A = P_Z_plus_P_c_Z_old;
    double s[r];
    double u[m * r];
    double vt[r * n];
    double *work;
    int lwork = -1;
    double lworkopt;
    int info;

        dgesvd_(&jobu, &jobvt, &m, &n, A, &m, s, u, &m, vt, &n, &
            lworkopt, &lwork, &info);

    if (info != 0)
    {
        printf("The dgesvd error %d\n", info);
    }
    else
    {
        lwork = (int) lworkopt;
        work = (double *) malloc(lwork * sizeof(double));
        assert(work != NULL);

        dgesvd_(&jobu, &jobvt, &m, &n, A, &m, s, u, &m, vt, &n, work, &
            lwork, &info);

        if (info != 0)
        {
            printf("The␣dgesvd␣error␣%d\n", info);
        }
    }
    //svd end

    soft_threshold(s, lambda, r);

    double ds[r*r];
    diag(s,r,ds);

    double us[m*r];
    char transa = 'N', transb = 'N';
    double alpha = 1;
    double beta = 0;

    // U*S_{lambda}
    dgemm_(&transa, &transb, &m, &r, &r, &alpha, u, &m, ds, &r, &beta,
        us, &m);
    // U*S_{Lambda}*Vt
    dgemm_(&transa, &transb, &m, &n, &r, &alpha, us, &m, vt, &r, &beta,
        znew, &m);
}

void soft_impute(double *zhat, double *z, double *zold, double *omega,
    double lambda, int m, int n){
    get_znew(zhat, z, zold, omega, lambda, m, n);
    zold = zhat;
    while (1){
        get_znew(zhat, z, zold, omega, lambda, m, n);
        if (diff(zold, zhat, m, n) < 1e-5)
            break;
    }
}



int main(){
    int M = 10, N = 10;
double Z_missing[10][10] =
{{0,-2.08,-1.67,-0.48,-0.52,-1.37,0.47,0.57,-0.2,-0.37},
{1.77,0,0.72,0,0,2.22,0.99,0,-2.54,3.76},
{-7.45,0,2.93,0,-0.68,0,-4.64,4.84,0,-7.13},
{3.26,0,-1.1,0,0,0,-1.34,-0.45,-2.14,0.73},
{3.46,0,0,-1.66,0,-0.83,3.65,-1.93,-1.8,3.19},
{-1.79,1.14,1.16,0,2.93,0,0,0,0,4.06},
{0,0.04,1.44,1.4,0,0,-2.14,1.52,0,-2.34},
{1.18,0.69,-0.49,0,0,0.45,4.33,-2.87,-1.26,0},
{0,0,0,-0.85,2.99,0,6.29,0,-1.53,6},
{2.16,0,0,-1.22,0,0,-1.05,0,0,0}};

double Omega[10][10] =
{{0,1,1,1,1,1,1,1,1,1},
{1,0,1,0,0,1,1,0,1,1},
{1,0,1,0,1,0,1,1,0,1},
{1,0,1,0,0,0,1,1,1,1},
{1,0,0,1,0,1,1,1,1,1},
{1,1,1,0,1,0,0,0,0,1},
{0,1,1,1,0,0,1,1,0,1},
{1,1,1,0,0,1,1,1,1,0},
{0,0,0,1,1,0,1,0,1,1},
{1,0,0,1,0,0,1,0,0,0}};

double Z_true[10][10] =
{{1.61,-2.08,-1.65,-0.48,-0.5,-1.36,0.49,0.58,-0.15,-0.42},
{1.79,4.1,0.69,-1.55,0.68,2.21,0.98,-2.93,-2.56,3.74},
{-7.43,-2.38,2.96,4.18,-0.71,-0.72,-4.64,4.82,5.29,-7.11},
{3.22,1.3,-1.05,-1.72,-1.19,0.38,-1.35,-0.43,-2.11,0.68},
{3.46,-1.02,-2.28,-1.68,0.67,-0.85,3.66,-1.94,-1.81,3.17},
{-1.76,1.18,1.16,0.47,2.92,1,5.04,-3.05,0.1,4.05},
{-2.66,0.05,1.44,1.38,-0.32,0.25,-2.13,1.51,1.6,-2.36},
{1.2,0.74,-0.47,-0.87,1.81,0.42,4.32,-2.87,-1.28,4.09},
{0.64,1.76,0.14,-0.84,3.01,1.12,6.3,-4.38,-1.52,6.05},
{2.13,1.53,-0.42,-1.23,-0.75,0.63,-1.06,-0.55,-1.62,0.73}};

double lambda[10];
int i,j,k;
// Let lambda
for (i=0; i<10; i++){
    lambda[i] = 0.1*i + 0.1;
}

double z[M*N];
double zold[M*N];
double zhat[M*N];
double ztrue[M*N];
double omega[M*N];
//double res[M][N][10];
//double error[10];


// Initialize zold as all zero
for (i=0; i<M; i++){
    for (j=0; j<N; j++){
        zold[i + M*j] = 0;
    }
}

// Initialize z as Z_missing, transpose for input, also transpose Z_true
     and save it as ztrue
for (i=0; i<M; i++){
    for (j=0; j<N; j++){
        z[i + M*j] = Z_missing[i][j];
        ztrue[i + M*j] = Z_true[i][j];
        omega[i + M*j] = Omega[i][j];
    }
```

```
}


for (k = 0; k < 10; k++){
    soft_impute(zhat, z, zold, omega, lambda[k], M, N);

    //error[k] = diff(zhat, ztrue, M, N);
    printf("%f\n", diff(ztrue, zhat, M, N));


    for (i=0; i<M*N; i++){
        zold[i] = zhat[i];
    }
}

return 0;
}
```

C part for RCpp Armadillo implementation:

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;
// [[Rcpp::export]]

List armadillo_svd(arma::mat &p){
   arma::mat U;
   arma::vec s;
   arma::mat V;

   svd(U,s,V,p);

   List ret;
   ret["U"]=U;
   ret["s"]=s;
   ret["V"]=V;


   return (ret);
}
```

All functions used in comparison and application part, includin four implementations of soft-impute:

```
########################################################################
#########              functions to do softimpute              ###########
#Do SVD and soft-threshold using method1
mysoft1 <- function(x,zold,lambda,rank.max){
  res = matrix(rep(0, nrow(x)*ncol(x)), nrow = nrow(x))
  res[!is.na(x)] = res[!is.na(x)] + x[!is.na(x)]
  res[is.na(x)]= res[is.na(x)] + zold[is.na(x)]
  s <- svd(res,nv = min(rank.max,qr(res)$rank), nu = min(rank.max,qr(res
      )$rank))
  D <- s$d[1:min(rank.max,qr(res)$rank)]
  D <- D - lambda
  D[D < 0] <- 0
  U <- (s$u)
  V <- (s$v)
```

```r
  D <- diag(D)
  res <- (U%*%D)%*%(t(V))
  return(res)
}

#Do SVD and soft-threshold using method2
require(svd)
mysoft2 <- function(x,zold,lambda,truerank){
  res = matrix(rep(0, nrow(x)*ncol(x)), nrow = nrow(x))
  res[!is.na(x)] = res[!is.na(x)] + x[!is.na(x)]
  res[is.na(x)]= res[is.na(x)] + zold[is.na(x)]
  s <- propack.svd(res,neig = truerank)
  D <- s$d
  D <- D - lambda
  D[D < 0] <- 0
  U <- (s$u)
  V <- (s$v)
  D <- diag(D)
  res <- (U%*%D)%*%(t(V))
  return(res)
}

#Do SVD and soft-threshold using method3
require(irlba)
mysoft3 <- function(x,zold,lambda,rank.max){
  res = matrix(rep(0, nrow(x)*ncol(x)), nrow = nrow(x))
  res[!is.na(x)] = res[!is.na(x)] + x[!is.na(x)]
  res[is.na(x)]= res[is.na(x)] + zold[is.na(x)]
  s <- irlba(res,nv = rank.max)
  D <- s$d[1:min(rank.max,qr(res)$rank)]
  D <- D - lambda
  D[D < 0] <- 0
  U <- (s$u)
  V <- (s$v)
  D <- diag(D)
  res <- (U%*%D)%*%(t(V))
  return(res)
}
#Do SVD and soft-threshold using method4 rcpp
require(irlba)
library(Rcpp)
library(RcppArmadillo)
sourceCpp("armadillo_svd.cpp")
mysoft4 <- function(x,zold,lambda){
  res = matrix(rep(0, nrow(x)*ncol(x)), nrow = nrow(x))
  res[!is.na(x)] = res[!is.na(x)] + x[!is.na(x)]
  res[is.na(x)]= res[is.na(x)] + zold[is.na(x)]
  s <- armadillo_svd(res)
  D <- s$s
  D <- D - lambda
  D[D < 0] <- 0
  U <- (s$U)
  V <- (s$V)
  D <- diag(c(D))
  res <- (U%*%D)%*%(t(V))
  return(res)
}
```

```
#the soft impute algorithm
mysoftimpute1 <- function (x,lambda,rank.max){
  zhat <- list()
  zold = matrix(rep(0, nrow(x) * ncol(x)), nrow = nrow(x))
  znew = mysoft1(x,zold,lambda[1],rank.max)
  zold = znew
  for (i in 1:length(lambda)){
    while(1){
      znew = mysoft1(x,zold,lambda[i],rank.max)
      diff=(sum((znew-zold)^2))/(sum(zold^2))
      if (is.na(diff)){
        break
      }
      if(diff < 10^-5){
        break
      }
      zold <- znew
    }
    zhat[[i]] <- znew
  }
  return (zhat)
}
#method 2
mysoftimpute2 <- function (x,lambda,truerank){
  zhat <- list()
  zold = matrix(rep(0, nrow(x) * ncol(x)), nrow = nrow(x))
  znew = mysoft2(x,zold,lambda[1],truerank)
  zold = znew
  for (i in 1:length(lambda)){
    while(1){
      znew = mysoft2(x,zold,lambda[i],truerank)
      diff=(sum((znew-zold)^2))/(sum(zold^2))
      if (is.na(diff)){
        break
      }
      if(diff < 10^-5){
        break
      }
      zold <- znew
    }
    zhat[[i]] <- znew
  }
  return (zhat)
}
#method 3
mysoftimpute3 <- function (x,lambda,rank.max){
  zhat <- list()
  zold = matrix(rep(0, nrow(x) * ncol(x)), nrow = nrow(x))
  znew = mysoft3(x,zold,lambda[1],rank.max)
  zold = znew
  for (i in 1:length(lambda)){
    while(1){
      znew = mysoft3(x,zold,lambda[i],rank.max)
      diff=(sum((znew-zold)^2))/(sum(zold^2))
      if (is.na(diff)){
```

```r
        break
      }
      if(diff < 10^-5){
        break
      }
      zold <- znew
    }
    zhat[[i]] <- znew
  }
  return (zhat)
}
#method 4 rcpp
mysoftimpute4 <- function (x,lambda){
  zhat <- list()
  zold = matrix(rep(0, nrow(x) * ncol(x)), nrow = nrow(x))
  znew = mysoft4(x,zold,lambda[1])
  zold = znew
  for (i in 1:length(lambda)){
    while(1){
      znew = mysoft4(x,zold,lambda[i])
      diff=(sum((znew-zold)^2))/(sum(zold^2))
      if (is.na(diff)){
        break
      }
      if(diff < 10^-5){
        break
      }
      zold <- znew
    }
    zhat[[i]] <- znew
  }
  return (zhat)
}









###################################################
#####     functions to generate matrix     #######

# function to generate low rank matrix. m and n are number of rows and
    columns of the generated matrix. Rank is the desired rank of
    generated matrix.
lrm.gen <- function(m, n, rank){
  U <- matrix(rnorm(m*rank), nrow = m)
  Vt <- matrix(rnorm(rank*n), nrow = rank)
  return(U%*%Vt)
}

# function to add noise to a matrix. X is the true matrix and SNR is the
     desired signal-to-noise ratio. Large SNR preferred for imputing.
add_noise <- function(X, SNR){
  noise <- rnorm(length(X))

  # scale noise to satisfy SNR
  noise <- sqrt(var(X[1:length(X)])/var(noise))/SNR * noise

  # shape the noise into the shape of true matrix
  dim(noise) <- dim(X)
  return(X + noise)
}

# function to generate p*100% missing postions in matrix X. Input p is
    in (0,1). Return a matrix of 0 and 1. (observed is 1).
obs.gen <- function(X, p){
  obs_ind <- sample(length(X), p*length(X))
  omega <- array(1, dim = dim(X))
  omega[obs_ind] <- 0
  return(omega)
}

# function to set unobserved postions NA
set_NA <- function(X, omega){
  X[omega == 0] <- NA
  return(X)
}

# generate low rank matrix, add noise and randomly remove
incomp.sim <- function(m,n,rank,SNR,p){
  # generate
  lrm <- lrm.gen(m,n,rank)

  # add noise
  lrm_with_noise <- add_noise(lrm, SNR)

  # omega
  omega <- obs.gen(lrm, p)

  #remove
  incomp <- set_NA(lrm_with_noise, omega)

  res <- list(incomp = incomp, true = lrm, true_wth_noise = lrm_with_
      noise, omega = omega)
  return(res)
}


###########################################################
#####        functions to do comparison      #############
#function to find test error
testerror <- function(uv,zhat,omega){
  resnum <- (norm((uv-zhat)*(1-omega), type = "F"))^2
  resden <- (norm(uv*(1-omega), type = "F"))^2
  res <- resnum/resden
  return(res)
}
#function to find training error
trainingerror <- function(z,zhat,omega){
  resnum <- (norm((z-zhat)*omega, type = "F"))^2
  resden <- (norm(z*omega, type = "F"))^2
  res <- resnum/resden
  return(res)
}
```

Plot part of comparison:

```
source("fpfunctions.R")

lambda = 100:1
teste <- rep(0,20000)
traine <- rep(0,20000)
rankzhat <- rep(0,20000)
method <- rep(c(1,2,3,4),each = 5000)
p <- 0.5
truerank <- 10
SNR <- 1
#for(p in c(0.3,0.5,0.8)){
#  for(SNR in c(1,10)){
#    for(truerank in c(5,10,20)){
###########################This is method 1 ########################
      for(i in 1:50){
        gen.matrix <- incomp.sim(100,100,truerank,SNR,p)
        uv <- (gen.matrix)$true
        z <- (gen.matrix)$true_wth_noise
        omega <- (gen.matrix)$omega
        incomp <- (gen.matrix)$incomp
        sim.matrix <- mysoftimpute1(incomp,lambda,100)
        for (j in 1:length(lambda)){
          teste[i*100 - 100 + j] <- testerror(uv,sim.matrix[[j]],omega)
          traine[i*100 - 100 + j] <- trainingerror(z,sim.matrix[[j]],
              omega)
          rankzhat[i*100 - 100 + j] <- qr(sim.matrix[[j]])$rank
        }
      }

#################This is method 2- svd package########################

for(i in 1:50){
  gen.matrix <- incomp.sim(100,100,truerank,SNR,p)
  uv <- (gen.matrix)$true
  z <- (gen.matrix)$true_wth_noise
  omega <- (gen.matrix)$omega
  incomp <- (gen.matrix)$incomp
  sim.matrix <- mysoftimpute2(incomp,lambda,75)
  for (j in 1:length(lambda)){
    teste[i*100 + 100*49 + j] <-  testerror(uv,sim.matrix[[j]],omega)
    traine[i*100 + 100*49 + j] <- trainingerror(z,sim.matrix[[j]], omega
        )
    rankzhat[i*100 + 100*49 + j] <-  qr(sim.matrix[[j]])$rank
  }
}
#################This is method 3- irlba########################
for(i in 1:50){
  gen.matrix <- incomp.sim(100,100,truerank,SNR,p)
  uv <- (gen.matrix)$true
  z <- (gen.matrix)$true_wth_noise
  omega <- (gen.matrix)$omega
  incomp <- (gen.matrix)$incomp
  sim.matrix <- mysoftimpute3(incomp,lambda,30)
  for (j in 1:length(lambda)){
    teste[i*100 + 100*99 + j] <- testerror(uv,sim.matrix[[j]],omega)
    traine[i*100 + 100*99 + j] <- trainingerror(z,sim.matrix[[j]], omega
        )
    rankzhat[i*100 + 100*99 + j] <- qr(sim.matrix[[j]])$rank
  }
}
################This is method 4- RCPP#######################
for(i in 1:50){
  gen.matrix <- incomp.sim(100,100,truerank,SNR,p)
  uv <- (gen.matrix)$true
  z <- (gen.matrix)$true_wth_noise
  omega <- (gen.matrix)$omega
  incomp <- (gen.matrix)$incomp
  sim.matrix <- mysoftimpute4(incomp,lambda)
  for (j in 1:length(lambda)){
    teste[i*100 + 100*149 + j] <- testerror(uv,sim.matrix[[j]],omega)
    traine[i*100 + 100*149 + j] <- trainingerror(z,sim.matrix[[j]],
        omega)
    rankzhat[i*100 + 100*149 + j] <- qr(sim.matrix[[j]])$rank
  }
}
df <- data.frame(testerror = teste, trainingerror = traine, rank =
    rankzhat, method = method)
save(df, file = "graphpart.Rda" )
#    }
#  }
#}



load("graphpart.Rda")
require(ggplot2)
df$method[df$method == 1] <- "original␣svd"
df$method[df$method == 2] <- "svd␣package"
df$method[df$method == 3] <- "irlba␣package"
df$method[df$method == 4] <- "rcpparmadillo"
df$method <- factor(df$method)
ggplot(df, aes(x = rank,y = testerror, color = method)) + geom_smooth()
    + theme_bw()  #label{comparison_testerror}
ggplot(df, aes(x = rank,y = trainingerror, color = method)) + geom_
    smooth() + theme_bw()
```

Table part of comparison:

```
source("fpfunctions2.R")

time_save <- c()
gen.matrix <- list()
uv <- list()
z <- list()
omega <- list()
incomp <- list()
teste_save1 <- rep(0,18)
```

```r
time_save1 <- rep(0,18)
teste_save2 <- rep(0,18)
time_save2 <- rep(0,18)
teste_save3 <- rep(0,18)
time_save3 <- rep(0,18)
teste_save4 <- rep(0,18)
time_save4 <- rep(0,18)
teste_temp <- rep(0,50)
lambda <- 500:1


for(k in 1:10){
i = 1
#######first generate 18 matrices
for(p in c(0.3,0.5,0.8)){
  for(SNR in c(1,10)){
    for(truerank in c(5,10,20)){
      gen.matrix[[i]] <- incomp.sim(100,100,truerank,SNR,p)
      uv[[i]] <- (gen.matrix[[i]])$true
      z[[i]] <- (gen.matrix[[i]])$true_wth_noise
      omega[[i]] <- (gen.matrix[[i]])$omega
      incomp[[i]] <- (gen.matrix[[i]])$incomp
##########################This is method 1 #######################
  time_start <- proc.time()
  sim.matrix <- mysoftimpute1(incomp[[i]],lambda,100)
  time_end <- proc.time()
  for (j in 1:length(lambda)){
    teste_temp[j] <- testerror(uv[[i]],sim.matrix[[j]], omega[[i]])
  }
  teste_save1[i] <- teste_save1[i] + min(teste_temp)
  time_save1[i] <- time_save1[i] + time_end[1] - time_start[1]
  ##########################This is method 2 #######################
  time_start <- proc.time()
  sim.matrix <- mysoftimpute2(incomp[[i]],lambda,50)
  time_end <- proc.time()
  for (j in 1:length(lambda)){
    teste_temp[j] <- testerror(uv[[i]],sim.matrix[[j]], omega[[i]])
  }
  teste_save2[i] <- teste_save2[i] + min(teste_temp)
  time_save2[i] <- time_save2[i] + time_end[1] - time_start[1]
  ##########################This is method 3 #######################
  time_start <- proc.time()
  sim.matrix <- mysoftimpute3(incomp[[i]],lambda,30)
  time_end <- proc.time()
  for (j in 1:length(lambda)){
    teste_temp[j] <- testerror(uv[[i]],sim.matrix[[j]], omega[[i]])
  }
  teste_save3[i] <- teste_save3[i] + min(teste_temp)
  time_save3[i] <- teste_save3[i] + time_end[1] - time_start[1]
  ##########################This is method 4 #######################
  time_start <- proc.time()
  sim.matrix <- mysoftimpute4(incomp[[i]],lambda)
  time_end <- proc.time()
  for (j in 1:length(lambda)){
    teste_temp[j] <- testerror(uv[[i]],sim.matrix[[j]], omega[[i]])
  }
  teste_save4[i] <- teste_save4[i] + min(teste_temp)
  time_save4[i] <- time_save4[i] + time_end[1] - time_start[1]
  i <- i+1
    }
  }
}
}
rank_save <- rep(c(5,10,20),6)
SNR_save <- rep(c(1,1,1,10,10,10),3)
p_save <- rep(c(0.3,0.5,0.8),each = 6)
df2 <- data.frame(time1 = time_save1/10,time2 = time_save2/10,time3 =
    time_save3/10,time4 = time_save4/10,error1 = teste_save1/10,error2 =
    teste_save2/10,
                  error3 = teste_save3/10,error4 = teste_save4/10,
                      truerank = rank_save, SNR = SNR_save, p = p_save)
save(df2,file = "comparison_testpart.Rda")



load("comparison_testpart.Rda")
df2
```

Lena:

```r
############################################################
#####        functions to do comparison      ##############
############################################################
#####        functions to do comparison      ##############
#function to find test error. Compare over the rest 40% of image, which
    is not taken as data.
testerror <- function(uv,zhat,omega){
  resnum <- (norm((uv-zhat)*(1-omega), type = "F"))^2
  resden <- (norm(uv*(1-omega), type = "F"))^2
  res <- resnum/resden
  return(res)
}
#function to find training error. Compare over validation set.
trainingerror <- function(z,zhat,omega){
  resnum <- (norm((z-zhat)*omega, type = "F"))^2
  resden <- (norm(z*omega, type = "F"))^2
  res <- resnum/resden
  return(res)
}

##############################################################
############ original svd in R################################
#a) R interval svd function
z_new_r_svd<-function(z,z_old,lambda){
  r_zero<-row(z)[which(z!=0)]
  c_zero<-col(z)[which(z!=0)]
  z_old[cbind(r_zero,c_zero)]<-0
  p<-z+z_old
  s<-svd(p)
  D<-diag(s$d)
  U<-(s$u)
  V<-(s$v)
  D<-D-lambda*diag(length(s$d))
  D[D<0]<-0
  z_new<-U%*%D%*%t(V)
  return (z_new)
}
r_svd<-function(z,lambda){
  z_hat<-array(rep(0,nrow(z)*ncol(z)*length(lambda)),dim=c(nrow(z),ncol(
      z),length(lambda)))
  z_old<-matrix(rep(0,nrow(z)*ncol(z)),nrow=nrow(z))
  for (i in 1:length(lambda)){
    z_new<-z_new_r_svd(z,z_old,lambda[i])
    diff=1
    while(diff>10^-5){
      z_old<-z_new
      z_new<-z_new_r_svd(z,z_old,lambda[i])
      diff<-sum((z_new-z_old)^2)/(sum(z_old^2)) }
    z_hat[,,i]<-z_new}
  return (z_hat)
}




############################################################
################ read in data################################
lena_test<-read.table("~/Documents/isu/STAT␣580/lena/test.txt")
lena_training<-read.table("~/Documents/isu/STAT␣580/lena/training.txt")
lena_validation<-read.table("~/Documents/isu/STAT␣580/lena/validating.
    txt")
lena_test_omega<-read.table("~/Documents/isu/STAT␣580/lena/test_omega.
    txt")
lena_validation_omega<-read.table("~/Documents/isu/STAT␣580/lena/
    validating_omega.txt")
lena_training_omega<-read.table("~/Documents/isu/STAT␣580/lena/trainng_
    omega.txt")
lena<-matrix(scan("~/Documents/isu/STAT␣580/lena/lena256",skip=1),nrow
    =256)

lena_test[is.na(lena_test)]<-0

lena_training[is.na(lena_training)]<-0

lena_validation[is.na(lena_validation)]<-0
lena_validation_omega[is.na(lena_validation_omega)]<-0
lena_training<-as.matrix(lena_training)
lena_test<-as.matrix(lena_test)
lena_validation<-as.matrix(lena_validation)
lena_validation_omega<-as.matrix(lena_validation_omega)
lena_test_omega<-as.matrix(lena_test_omega)
lena_training_omega<-as.matrix(lena_training_omega)
lena<-as.matrix(lena)

lambda=seq(500,10,-10)

lena_training_hat<-array(rep(0,nrow(lena_training)*ncol(lena_training)*
    length(lambda)),dim=c(nrow(lena_training),ncol(lena_training),length(
    lambda)))
lena_training_error<-rep(0,length(lambda))
```

```
for (i in 1:length(lambda)){
  lena_training_hat[,,i]<-r_svd(lena_training,lambda[i])
  lena_training_error[i]<-trainingerror(lena_validation,lena_training_
      hat[,,i], lena_validation_omega)

}
best_lambda<-lambda[which.min(lena_training_error)]
best_lambda
#190
min(lena_training_error)
#0.04585135
lena_test_hat<-r_svd(lena_test,best_lambda)[,,1]

lena_test_error<-testerror(lena,lena_test_hat,lena_test_omega)
lena_test_error
#0.03104175
library(Matrix)
rankMatrix(lena)
rankMatrix(lena_test_hat)
#original image
image(lena[256:1,256:1],col = grey(seq(0, 1, length = 256)))

#before pic amendment
image(lena_test[256:1,256:1],col = grey(seq(0, 1, length = 256)))

#after pic amendment
image(lena_test_hat[256:1,256:1],col = grey(seq(0, 1, length = 256)))
```

The first grid part of λ selection in MovieLens ($\lambda = 100, 90, \ldots, 10$):

Using `irlba package`

```
##########################################################
#####        functions to do comparison      #############
##########################################################
#####        functions to do comparison      #############
#function to find test error
testerror <- function(uv,zhat,omega){
  resnum <- (norm((uv-zhat)*(1-omega), type = "F"))^2
  resden <- (norm(uv*(1-omega), type = "F"))^2
  res <- resnum/resden
  return(res)
}
#function to find training error
trainingerror <- function(z,zhat,omega){
  resnum <- (norm((z-zhat)*omega, type = "F"))^2
  resden <- (norm(z*omega, type = "F"))^2
  res <- resnum/resden
  return(res)
}

############################################################
############ irlba################################
require(irlba)
z_new_i<-function(z,z_old,lambda){
    r_zero<-row(z)[which(z!=0)]
    c_zero<-col(z)[which(z!=0)]
    z_old[cbind(r_zero,c_zero)]<-0
    p<-z+z_old
    s<-irlba(p,nv=100)
    D<-diag(s$d)
    U<-(s$u)
    V<-(s$v)
    D<-D-lambda*diag(length(s$d))
    D[D<0]<-0
    z_new<-U%*%D%*%t(V)
    return (z_new)
}

irlba_svd<-function(z,lambda){
 z_hat<-array(rep(0,nrow(z)*ncol(z)*length(lambda)),dim=c(nrow(z),ncol(z
     ),length(lambda)))
 z_old<-matrix(rep(0,nrow(z)*ncol(z)),nrow=nrow(z))
for (i in 1:length(lambda)){
    z_new<-z_new_i(z,z_old,lambda[i])
    diff=1
    while(diff>10^-5){
        z_old<-z_new
        z_new<-z_new_i(z,z_old,lambda[i])
        diff<-sum((z_new-z_old)^2)/(sum(z_old^2))
        if (is.na(diff)){
        break
      }
    }
z_hat[,,i]<-z_new
}
return (z_hat)
}


############################################################
################ read in data##################################
m_test<-read.table("test.txt")
m_training<-read.table("trainng.txt")
m_validation<-read.table("validating.txt")
m_test_omega<-read.table("test_omega.txt")
m_validation_omega<-read.table("validating.txt")
m_training_omega<-read.table("trainng_omega.txt")
m_validation2<-read.table("validating_2.txt")
m_validation2_omega<-read.table("validating_2_omega.txt")

m_test[is.na(m_test)]<-0

m_training[is.na(m_training)]<-0

m_validation[is.na(m_validation)]<-0
m_validation2[is.na(m_validation2)]<-0
m_validation_omega[is.na(m_validation_omega)]<-0
m_validation2_omega[is.na(m_validation2_omega)]<-0
m_training<-as.matrix(m_training)
m_test<-as.matrix(m_test)
m_validation<-as.matrix(m_validation)
m_validation2<-as.matrix(m_validation2)
m_validation_omega<-as.matrix(m_validation_omega)
```

```r
m_validation2_omega<-as.matrix(m_validation2_omega)
m_test_omega<-as.matrix(m_test_omega)
m_training_omega<-as.matrix(m_training_omega)


lambda=seq(100,10,-10)




############comparing lambda##################
lambda_com<-function(td,vd,lambda,td_omega){
td_hat<-array(rep(0,nrow(td)*ncol(td)*length(lambda)),dim=c(nrow(td),
    ncol(td),length(lambda)))
td_error<-rep(0,length(lambda))
for (i in 1:length(lambda)){
    td_hat[,,i]<-irlba_svd(td,lambda[i])
    td_error[i]<-trainingerror(vd,td_hat[,,i], td_omega)
}
best_lambda<-lambda[which.min(td_error)]
min_td_error<-td_error
return (list(best_lambda,min_td_error))
}

#ts<-proc.time()
l<-lambda_com(m_training,m_validation,lambda,m_validation_omega)
t<-system.time(lambda_com(m_training,m_validation,lambda,m_validation_
    omega))
#t<-proc.time()-ts


best_lambda<-unlist(l[1])

min_td_error<-unlist(l[2])

o<-list(t,best_lambda,min_td_error)
save(o, file = "m_irlba100_10.Rda" )
```

Using rcpparmadillo

```r
##########################################################
#####        functions to do comparison      #############
##########################################################
#####        functions to do comparison      #############
#function to find test error
testerror <- function(uv,zhat,omega){
  resnum <- (norm((uv-zhat)*(1-omega), type = "F"))^2
  resden <- (norm(uv*(1-omega), type = "F"))^2
  res <- resnum/resden
  return(res)
}
#function to find training error
trainingerror <- function(z,zhat,omega){
  resnum <- (norm((z-zhat)*omega, type = "F"))^2
  resden <- (norm(z*omega, type = "F"))^2
  res <- resnum/resden
  return(res)
}

###################################################################
############ Rcpp Armadillo################################
library(Rcpp)
library(RcppArmadillo)
sourceCpp("armadillo_svd.cpp")
z_new_armadillo_svd<-function(z,z_old,lambda){
    r_zero<-row(z)[which(z!=0)]
    c_zero<-col(z)[which(z!=0)]
    z_old[cbind(r_zero,c_zero)]<-0
    p<-z+z_old
    p<-armadillo_svd(p)
    D<-diag(as.vector(p$s))
    U<-(p$U)
    V<-(p$V)
    D<-D-lambda*diag(length(p$s))
    D[D<0]<-0
    Da<-matrix(rep(0,dim(U)[2]*dim(V)[2]),nrow=dim(U)[2])
    Da[1:dim(D)[1],1:dim(D)[2]]<-Da[1:dim(D)[1],1:dim(D)[2]]+D
    D<-Da
    z_new<-U%*%D%*%t(V)
    return (z_new)
}
armadillo_cpp_svd<-function(z,lambda){
 z_hat<-array(rep(0,nrow(z)*ncol(z)*length(lambda)),dim=c(nrow(z),ncol(z
     ),length(lambda)))
 z_old<-matrix(rep(0,nrow(z)*ncol(z)),nrow=nrow(z))
for (i in 1:length(lambda)){
        z_new<-z_new_armadillo_svd(z,z_old,lambda[i])
        diff=1
    while(diff>0.00001){
        z_old<-z_new
        z_new<-z_new_armadillo_svd(z,z_old,lambda[i])
        diff<-sum((z_new-z_old)^2)/(sum(z_old^2))
        if (is.na(diff)){
        break
      }
    }
z_hat[,,i]<-z_new}
return (z_hat)
}

###################################################################
################ read in data###################################
m_test<-read.table("test.txt")
m_training<-read.table("trainng.txt")
m_validation<-read.table("validating.txt")
m_test_omega<-read.table("test_omega.txt")
m_validation_omega<-read.table("validating.txt")
m_training_omega<-read.table("trainng_omega.txt")
m_validation2<-read.table("validating_2.txt")
m_validation2_omega<-read.table("validating_2_omega.txt")

m_test[is.na(m_test)]<-0

m_training[is.na(m_training)]<-0
```

```
m_validation[is.na(m_validation)]<-0
m_validation2[is.na(m_validation2)]<-0
m_validation_omega[is.na(m_validation_omega)]<-0
m_validation2_omega[is.na(m_validation2_omega)]<-0


m_training<-as.matrix(m_training)
m_test<-as.matrix(m_test)
m_validation<-as.matrix(m_validation)
m_validation2<-as.matrix(m_validation2)
m_validation_omega<-as.matrix(m_validation_omega)
m_validation2_omega<-as.matrix(m_validation2_omega)
m_test_omega<-as.matrix(m_test_omega)
m_training_omega<-as.matrix(m_training_omega)


lambda=seq(100,10,-10)




############comparing lambda##################
lambda_com<-function(td,vd,lambda,td_omega){
td_hat<-array(rep(0,nrow(td)*ncol(td)*length(lambda)),dim=c(nrow(td),
    ncol(td),length(lambda)))
td_error<-rep(0,length(lambda))
for (i in 1:length(lambda)){
    td_hat[,,i]<-armadillo_cpp_svd(td,lambda[i])
    td_error[i]<-trainingerror(vd,td_hat[,,i], td_omega)
}
best_lambda<-lambda[which.min(td_error)]
min_td_error<-td_error
return (list(best_lambda,min_td_error))
}

#ts<-proc.time()
l<-lambda_com(m_training,m_validation,lambda,m_validation_omega)
t<-system.time(lambda_com(m_training,m_validation,lambda,m_validation_
    omega))
#t<-proc.time()-ts


best_lambda<-unlist(l[1])

min_td_error<-unlist(l[2])

o<-list(t,best_lambda,min_td_error)
save(o, file = "p_cpp_100_10.Rda" )
```

Imputing matrix using optimal λ and getting test error and RMSE:

```
best_lambda <- 13

m_test_hat_rcpp<-armadillo_cpp_svd(m_test,best_lambda)[,,1]

save(m_test_hat,file="m_test_hat_rcpp.Rda")
```

```
m_test_hat_irlba<-irlba_svd(m_test,best_lambda)[,,1]

save(m_test_hat_irlba,file="m_test_hat_irlbaRda")
```

```
load("m_test_hat_irlba.Rda")
m_test_hat_irlba <- m_test_hat
load("m_test_hat_cpp.Rda")
m_test_hat_cpp <- m_test_hat
test_set <- read.table("./Movie␣data/validating_2.txt")
test_set <- as.matrix(test_set)
test_set[is.na(test_set)] <- 0
test_set_omega <- read.table("./Movie␣data/validating_2_omega.txt")
test_set_omega <- as.matrix(test_set_omega)


# result from irlba
test_error_irlba <- (norm(m_test_hat_irlba*test_set_omega - test_set, "F
    "))^2/(norm(test_set, "F"))^2

RMSE_irlba <- sqrt((norm(m_test_hat_irlba*test_set_omega - test_set, "F"
    ))^2/(sum(test_set_omega)))

# result from Rcpp
test_error_cpp <- (norm(m_test_hat_cpp*test_set_omega - test_set, "F"))
    ^2/(norm(test_set, "F"))^2

RMSE_cpp <- sqrt((norm(m_test_hat_cpp*test_set_omega - test_set, "F"))^2
    /(sum(test_set_omega)))
```