2. Various SVD methods and Comparison

2.1 Method introduction

2.1.1 R internal svd function

The main functions used are the LAPACK routines DGESDD and ZGESDD. LAPACK is from <http://www.netlib.org/lapack> and its guide is listed in the references. [[1]](#footnote-1)

2.1.2 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 and we need that vector in soft-impute algorithm. So we are not going to use those two functions. We only use “propack.svd” to implement the soft-impute algorithm.

PROPACK does SVD via the implicitly restarted Lanczos bidiagonalization with partial reorthogo- nalization.[[2]](#footnote-2)

2.1.3 RcppArmadillo

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[[3]](#footnote-3). We use svd in “Armadillo” to implement the soft-impute algorithm.

2.1.4 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. [[4]](#footnote-4)

The maximum number of singular values that can be calculated in this function is

where is the dimension of the matrix.

To find the suitable number of singular values in each iteration of the 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 Comparisons

There are many aspects that can reflect performance of a method such as training 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 training error. And in our simulation experiments, they do have the same training error so we are not going to compare the training error. We are going to compare the speed of each method in our experiments.

By learning the soft-impute algorithm, we think that three factors may influence the speed. They are dimension of the matrix, missing rate and true rank of the matrix. In our experiment, we let one variable vary and keep other two variables the same to say how each variable affect the performance.

To compare the speed of each method, we use “microbenchmark” to calculate the time usage and take the median as the result.

2.2.1 Change dimension (missing rate = 0.5, true rank = 8)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimension | 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 |

Table 2.1: Time usage under missing rate = 0.5 and true rank = 8 with different dimensions

In general, the result shows that the computation speed of “R internal svd” and “RcppArmadillo” methods are similar and are the fastest method. When the matrix size is relatively large (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 relative difference with those two 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 usage of this method will go up dramatically as we can see in Table 2.1.

2.2.2 Change missing rate (dimension = (100,100), true 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 |

Table 2.2: Time usage under dimension = (100,100), true rank = 8 with different missing rate

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.

“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 (eg 100\*100), the “irlba” method is the fastest method.

2.2.3 Change true rank (dimension = (100,100), missing rate=0.5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| True 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 |

Table 2.3: Time usage under dimension = (100,100), missing rate=0.5 with different missing rate

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

After comparing different methods, we decide to use “R internal svd” to solve this problem. To design a grid of , 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 . 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 3.1:

 

Figure 3.1 Figure 3.2 Figure 3.3

We find that the minimal of the RMSE1 corresponding to . Then we do further search. We use as the second vector. After calculating the RMSE in the validation set we get the Figure 3.2. This time we find the minimal of the RMSE corresponding to . We continue our search to find the best . We use as the third vector. The RMSE can be seen in the Figure 3.3. In Figure 3.3, we can see that the minimal of RMSE3 corresponding to which is the best we choose.

As we can see, by using quantile, we only need s to select the turning parameter. In this way, we can reduce the total number of s to be considered and therefore save a lot of time.

Figure 3.4 is the image with 40% pixels randomly missing. And Figure 3.5 is the image after using the soft-impute to fix.

The total rank in the reconstructed image is 83.



Figure 3.4 Figure 3.5

1. R help [↑](#footnote-ref-1)
2. https://cran.r-project.org/web/packages/svd/svd.pdf [↑](#footnote-ref-2)
3. https://cran.r-project.org/web/packages/RcppArmadillo/RcppArmadillo.pdf [↑](#footnote-ref-3)
4. https://cran.r-project.org/web/packages/irlba/irlba.pdf [↑](#footnote-ref-4)