Numerical Linear Algebra

The SVT Algorithm

Nick Morris

05/13/2017

Implementation

The following input parameters were used for the svt() function, when implementing the SVT algorithm:

1. sample.set:
    a. a matrix with three columns: row index (column 1), column index (column 2), value (column 3)
    b. each row of sample.set corresponds to one entry of the unknown low rank matrix
2. matrix.size:
    a. indicates the number of rows and columns of the square low rank unknown matrix
3. step.size:
    a. a scalar used to decide how to converge onto the completion of the low rank unknown matrix
    b. too large of a value may constantly jump back and forth over the point of convergence
    c. too low of value may take too many iterations to reach the point of convergence
4. tolerance:
    a. a scalar used to decide if the algorithm has converged
5. threshold:
    a. the value that will be used to subtract from singular values at each iteration
6. increment:
    a. a scalar used to increase the number of required singular values to compute, until we find a singular value $<$ threshold
7. max.iterations:
    a. a scalar used to stop the algorithm
    b. if max.iterations have occurred then this means we did not satisfy the tolerance condition for convergence
8. the.matrix:
    a. if the true low rank unknown matrix is given then we compute:
        i. relative error (full.relative.error)
        ii. relative reconstruction error (relative.error)
        iii. best possible error (best.error)
    b. otherwise we compute:
        i. relative reconstruction error (relative.error)
9. noise.aware:
    a. if set to TRUE: tolerance = 1.05 * tolerance

The svt() function starts off with error checking to make sure sample.set, matrix.size, and max.iterations have been inputted properly. Then the function computes values for step.size, tolerance, threshold, and increment if they were not already given by the user. The values computed for these four parameters correspond to the suggested values in the SVT paper. The matrices $P_\Omega(M)$ and Y were implemented to be sparse matrices, meaning that there was space saved for RAM by ignoring zero elements. The Lanczos bidiagonalization algorithm was used for computed the truncated SVD of Y at each iteration. The package irlba provided the function irlba() which does this truncated SVD computation, and it has functionality to extend the truncation to be larger if more singular values need to be computed. The number of SVD iterations on Y iterates, rank of X iterates, and relative errors of X iterates are kept track of. Then convergence is checked for, if this has not occurred then the next Y iterate is computed and the loop restarts with SVD computations.

The svt() function has the following outputs:

1. X:
    a. the estimate of the unknown low rank matrix M
2. duration:
    a. the time in seconds it took for the algorithm to converge
3. iterations:
    a. the number of iterations it took for the algorithm to converge
4. relative.errors:
    a. the relative reconstruction error of X at each iteration → $||P_\Omega(M - X_i)||_F$ / $||P_\Omega(M)||_F$
        i. where X_i is X at iteration i
5. svd.iterations:
    a. the number of iterations it took for the Lanczos bidiagonalization algorithm to find all singular values less than threshold, at each iteration of Y
6. X.ranks:
    a. the rank of X at each iteration
7. full.relative.errors:
    a. the relative error of X at each iteration → $||M - X_i||_F$ / $||M||_F$
8. best.errors:
    a. the best possible error at each iteration of X → $||M - M_i||_F$ / $||M||_F$
        i. where M_i is the best rank-k approximation of X for k = rank(X_i)

Table 5.1

Five replications for each row was run under different seeds, and then averaged together. The 1000 x 1000 matrices are solved faster and have less iterations under my implementation, when compared to Table 5.1 in the SVT paper. Most of the 5000 x 5000 matrices are solved faster and have less iterations under my implementation. The higher rank 10000 x 10000 matrices are solved faster, whereas the lower rank matrices are solved slower. The 10000 x 10000 matrices have less iterations. The 20000 x 20000 and 30000 x 30000 matrices were not able to be completed due to the size of the workspace reaching memory capacity. See the picture below for details. This issue can be fixed by exporting each replicate to its own file such that the workspace doesn't get over sized with large matrix objects.

n	r	m/d_r	m/n^2	Time (sec)	Iterations	Relative Error
1000	10	6	0.12	8.2836	74.2	0.000980979
1000	50	4	0.39	35.4432	73.33333	0.000941444
1000	100	3	0.57	112.321	81.8	0.000974057
5000	10	6	0.024	156.1123	77	0.000974254
5000	50	5	0.1	401.9136	69.4	0.000965844
5000	100	4	0.158	1052.8099	79	0.00098196
10000	10	6	0.012	957.0854	78	0.000974762
10000	50	5	0.05	1547.0116	71	0.000968368
10000	100	4	0.08	3354.2353	81.2	0.000984525
20000	10	6	0.006	6933.3364	78	0.000986195
20000	50	5	0.025	NA	NA	NA
30000	10	6	0.004	NA	NA	NA

```
Error: cannot allocate vector of size 6.7 Gb
In addition: Warning messages:
1: In M(n = 30000, r = 10, seeds = c(42, 5)) :
  Reached total allocation of 16271Mb: see help(memory.size)
2: In M(n = 30000, r = 10, seeds = c(42, 5)) :
  Reached total allocation of 16271Mb: see help(memory.size)
3: In M(n = 30000, r = 10, seeds = c(42, 5)) :
  Reached total allocation of 16271Mb: see help(memory.size)
4: In M(n = 30000, r = 10, seeds = c(42, 5)) :
  Reached total allocation of 16271Mb: see help(memory.size)
```

Table 5.2

Five replications for each row was run under different seeds, and then averaged together. Each of the delta settings require less iterations across all tau settings, under my implementation. The final rank of my estimated matrix X is much higher when tau = 2000, and similar when tau ≠ 2000.

tau	delta	Iteration Mean	Iteration Std. Dev.	Rank Mean
2000	6.666667	134.4	103.9966346	134.6
3000	6.666667	119.4	100.4006972	10
4000	6.666667	93.6	1.3416408	10
5000	6.666667	112.6	1.9493589	10
6000	6.666667	131.4	2.0736441	10
2000	10	274.4	407.4841101	117.8
3000	10	113.6	144.4499913	10.6
4000	10	62.4	0.8944272	10
5000	10	74.8	1.3038405	10
6000	10	87.2	1.6431677	10
2000	13.333333	DNC	DNC	DNC
3000	13.333333	478.2	349.9359941	10
4000	13.333333	236.6	426.7537932	10
5000	13.333333	56	1.2247449	10
6000	13.333333	65	1.2247449	10

Figure 5.1

The convergence of each matrix is like the convergence in Figure 5.1 of the SVT paper. This does not fully support the interesting phenomenon of the rank of successive X iterates to be non-decreasing, but shows evidence for the rank of most successive X iterates to be non-decreasing.

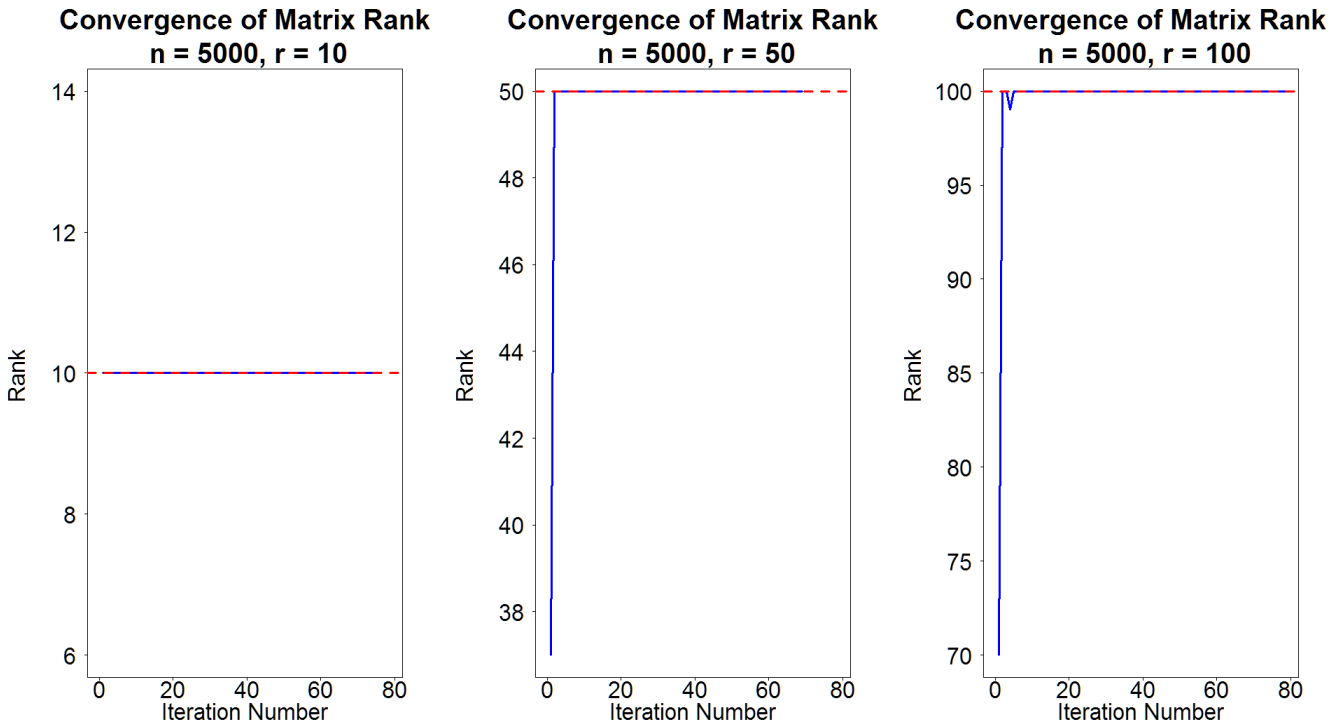

Table 5.5

The city-to-city geo-distance data was loaded from the TSP package in R. The number of iterations and the time it took to solve each matrix, across both algorithms, was higher under my implementation. The best possible relative error and relative error are like the results in Table 5.5 of the SVT paper.

Algorithm	Rank, i	Iterations	Time (sec)	$\|\|M - M_i\|\|_F / \|\|M\|\|_F$	$\|\|M - X^{opt}_i\|\|_F / \|\|M\|\|_F$
SVT	1	264	8.0028	0.409141	0.4091424
SVT	2	682	21.0912	0.1894546	0.1894586
SVT	3	1569	48.8436	0.1158721	0.1158801
Noise Aware	1	261	8.0496	0.409141	0.4091426
Noise Aware	2	673	20.9996	0.1894546	0.189459
Noise Aware	3	1546	47.973	0.1158721	0.1158809

Figure 5.2

In the SVT and noise aware variant (NVA) implementations, Relative Error converges onto Best Possible Relative Error for every step down taken by Best Possible Relative Error, which is like the results in the SVT paper. The matrix rank convergence climbs upward in rank but my convergence ends at rank 3 which maps to the best rank-3 approximation of the city-to-city distance matrix. For SVT and NAV, the Relative Residual Error drops below Best Possible Relative Error which makes sense given that Relative Residual Error is computed based on entries of X that correspond to the Ω sample set. Whereas Relative Error and Best Possible Relative Error are computed based on all entries, and it also makes sense that Relative Error never drops below Best Possible Relative Error.