Fast Randomized PCA for Sparse Data

Xu Feng   FX17@MAILS.TSINGHUA.EDU.CN
Yuyang Xie   XYY18@MAILS.TSINGHUA.EDU.CN
Mingye Song   SONGMY16@MAILS.TSINGHUA.EDU.CN
Wenjian Yu∗   YU-WJ@TSINGHUA.EDU.CN
Jie Tang   JIETANG@TSINGHUA.EDU.CN
BNRist, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

Abstract

Principal component analysis (PCA) is widely used for dimension reduction and embedding of real data in social network analysis, information retrieval, and natural language processing, etc. In this work we propose a fast randomized PCA algorithm for processing large sparse data. The algorithm has similar accuracy to the basic randomized SVD (rPCA) algorithm (Halko et al., 2011), but is largely optimized for sparse data. It also has good flexibility to trade off runtime against accuracy for practical usage. Experiments on real data show that the proposed algorithm is up to 9.1X faster than the basic rPCA algorithm without accuracy loss, and is up to 20X faster than the svds in Matlab with little error. The algorithm computes the first 100 principal components of a large information retrieval data with 12,869,521 persons and 323,899 keywords in less than 400 seconds on a 24-core machine, while all conventional methods fail due to the out-of-memory issue.

Keywords: Principle Component Analysis; Singular Value Decomposition; Randomized Algorithms

1. Introduction

In machine learning applications, principal component analysis (PCA) is widely used for dimension reduction of input data. It often behaves as a preprocessing step for no matter supervised or unsupervised learning methods. For the problems of social network analysis, information retrieval, natural language processing (NLP), and even recommender system, where input data matrix is usually a sparse one, PCA or the equivalent truncated SVD is also applied. For example, the latent semantic analysis (LSA) (Deerwester et al., 1990), which builds dense representation of document in NLP, includes PCA as a major step. However, the application of PCA to real-world problems which require processing large data accurately, often costs prohibitive computational time. Accelerating the PCA for large sparse data is of an absolute necessity.

The standard method for performing PCA is calculating truncated singular value decomposition (SVD). For sparse matrix, this is usually implemented with svds in Matlab (Lehoucq et al., 1998), or lansvd in PROPACK (Larsen, 2004) which is an accelerated version of svds. However, if the dimensions of matrix are large and more than dozens of principle components/directions are needed, these conventional methods would induce large computational expense or even fail due to

---

excessive memory cost. An alternative is the randomized method for PCA, which has gained a lot of attention in recent years. The idea of randomized matrix method is mainly using random projection to identify the subspace capturing the dominant actions of a matrix (Halko et al., 2011; Yu et al., 2018). Then, a near-optimal low-rank decomposition of the matrix can be computed, so that we can further obtain an approximate PCA. A comprehensive presentation of the relevant techniques and theories is in Halko et al. (2011). This randomized technique has been extended to compute PCA of data sets that are too large to be stored in RAM (Yu et al., 2017), or to speed up the distributed PCA (Liang et al., 2014). For general SVD or PCA computation, the approaches based on it have also been proposed (Voronin and Martinsson, 2015; Li et al., 2017). They outperform the conventional techniques for calculating a few of principle components. Recently, a compressed SVD (cSVD) algorithm was proposed in Benjamin Erichson et al. (2017), which is based on a variant of the method in Halko et al. (2011) but runs faster for image and video processing applications. Another idea for computing PCA of large data is performing eigenvalue decomposition to the product of the data matrix's transpose and itself. However, it only has benefit while handling low-dimensional data (less than several thousands in dimension).

Although there are a lot of work on randomized PCA techniques, they are mostly for processing dense data. Compared with the deterministic methods, they involve the same or fewer floating-point operations (*flops*), and are more efficient for large high-dimensional dense data by exploiting modern computing architectures. While for large sparse data in real world, these benefits may not exist. Investigating the randomized PCA technique for large sparse data and comparing it with other existing techniques are of great interest.

In this work, we first analyze the adaptability of some acceleration skills for the basic randomized PCA (rPCA) algorithm to sparse data, followed by theoretical proofs and computational cost analysis. Then, we propose a modified power iteration scheme which allows odd number of passes over data matrix and thus provides more flexible trade-off between runtime and accuracy. We also devise a technique to efficiently handle the data matrix with more columns than rows, which is ignored in existing work. To wrap them up, we propose a fast randomized PCA algorithm for sparse data (*frPCA*) and its variant algorithm *frPCAt*, suitable for the data matrices with more columns and more rows, respectively. Theoretical analysis is performed to reveal how the efficiency of the proposed algorithms varies with the sparsity of data, the power iteration parameter, and the number of principle components wanted. In the section of experimental results, we first validate the accuracy and efficiency of the proposed algorithms with some synthetic data. The results show that it is up to 9.1X faster than the basic rPCA algorithm and 20X faster than `svds`, with negligible loss of accuracy. Then, real large data in social network, information retrieval and recommender system problems are tested. The results show the proposed algorithm is up to 8.7X faster than the basic rPCA. And, it successfully handles the largest case in less than 400 seconds and with 23 GB memory, while the `svds` fails due to out-of-memory issue (requesting more than 32 GB memory).

For reproducibility, the codes and test data in this work will be shared on GitHub (`https://github.com/XuFengthucs/frPCA_sparse`).

2. Preliminaries

In algorithm description, the Matlab conventions are used for specifying row/column indices of a matrix and some operations on sparse matrix.

2.1. Singular Value Decomposition and PCA

A standard method for performing PCA is to calculate truncated SVD. The SVD of $\mathbf{A} \in \mathbb{R}^{m \times n}$ is:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}, \tag{1}$$

where $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \cdots]$ and $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \cdots]$ are orthogonal matrices which represent the left and right singular vectors, respectively. The diagonal matrix $\mathbf{\Sigma}$ contains the singular values $(\sigma_1, \sigma_2, \cdots)$ of $\mathbf{A}$ in descending order. Suppose that $\mathbf{U}_k$ and $\mathbf{V}_k$ are matrices with the first k columns of $\mathbf{U}$ and $\mathbf{V}$ respectively, and $\mathbf{\Sigma}_k$ is the diagonal matrix containing the first k singular values of $\mathbf{A}$. Then, the truncated SVD of $\mathbf{A}$ can be represented as:

$$\mathbf{A} \approx \mathbf{A}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^{\mathrm{T}}. \tag{2}$$

Notice that $\mathbf{A}_k$ is the best rank-k approximation of the initial matrix $\mathbf{A}$ in either spectral norm of Frobenius norm (Eckart and Young, 1936).

The approximation properties of SVD explain the equivalence between the truncated SVD and PCA. Suppose each row of matrix $\mathbf{A}$ is an observed data. The matrix is assumed to be centered, i.e., the mean of all rows is a zero row vector. Then, the leading left singular vectors $\mathbf{u}_i$ are the principal components. Particularly, $\mathbf{u}_1$ is the first principal component.

The built-in function svds in Matlab is a common choice to compute truncated SVD. It is based on a Krylov subspace iterative method and is especially efficient for sparse matrix. For a dense matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, svds costs $O(mnk)$ *flops* for computing rank-k truncated SVD. The cost becomes $O(nnz(\mathbf{A})k)$ *flops* when $\mathbf{A}$ is sparse, where $nnz(\cdot)$ means the number of nonzero elements. lansvd in PROPACK (Larsen, 2004) is also an efficient program, written in Matlab/Fortran, for computing the dominant singular values/vectors of a sparse matrix. lansvd can cost two to three times less CPU time than svds. However, there is no parallel version of lansvd, so that its actual runtime on a modern computer is often longer than that of svds.

2.2. The Basic Randomized Algorithm for PCA

Previous work has shown that the randomized methods have advantages for solving the least linear squares problem and low-rank matrix approximation (Drineas and Mahoney, 2016). The method for low-rank approximation mainly relies on the random projection to identify the subspace capturing the dominant actions of matrix $\mathbf{A}$, which can be realized by multiplying $\mathbf{A}$ with a random matrix on its right or left side to obtain the subspace's orthogonal basis matrix $\mathbf{Q}$. Then, the low-rank approximation in form of $\mathbf{QB}$ is computed and further results in the approximate truncated SVD (Halko et al., 2011). Because $\mathbf{Q}$ has much fewer columns than $\mathbf{A}$, this method reduces the computational time. It derives the basic randomized PCA (rPCA) algorithm, described as Algorithm 1.

In Alg. 1, $\mathbf{\Omega}$ is a Gaussian i.i.d matrix. Although other kinds of random matrix can replace $\mathbf{\Omega}$ to reduce the computational cost of $\mathbf{A\Omega}$, they bring some sacrifice on accuracy. The s in Alg. 1 is an oversampling parameter which enables $\mathbf{\Omega}$ with more than k columns for better accuracy. s is a small integer such as 5 or 10. With $\mathbf{Q}$, we have $\mathbf{A} \approx \mathbf{QB} = \mathbf{QQ}^{\mathrm{T}}\mathbf{A}$. By performing the economic SVD on the $(k + s) \times n$ matrix $\mathbf{B}$ the approximate truncated SVD of $\mathbf{A}$ is obtained. To improve the accuracy of the approximation, power iteration (PI) scheme can be applied (Halko et al., 2011), i.e., Steps 3~6. It is based on that matrix $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}$ has exactly the same singular vectors as $\mathbf{A}$, but its singular values decay more quickly. Therefore, performing the randomized QB procedure

---

Algorithm 1 basic rPCA

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$, rank parameter k, power parameter p
Output: $\mathbf{U} \in \mathbb{R}^{m \times k}, \mathbf{S} \in \mathbb{R}^{k \times k}, \mathbf{V} \in \mathbb{R}^{n \times k}$
  1: $\mathbf{\Omega} = \text{randn}(n, k + s)$
  2: $\mathbf{Q} = \text{orth}(\mathbf{A\Omega})$
  3: **for** $i = 1, 2, \cdots, p$ **do**
  4:    $\mathbf{G} = \text{orth}(\mathbf{A}^{\mathrm{T}}\mathbf{Q})$
  5:    $\mathbf{Q} = \text{orth}(\mathbf{AG})$
  6: **end for**
  7: $\mathbf{B} = \mathbf{Q}^{\mathrm{T}}\mathbf{A}$
  8: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{B})$
  9: $\mathbf{U} = \mathbf{QU}$
 10: $\mathbf{U} = \mathbf{U}(:, 1 : k), \mathbf{S} = \mathbf{S}(1 : k, 1 : k), \mathbf{V} = \mathbf{V}(:, 1 : k)$

---

on $(\mathbf{AA}^{\mathrm{T}})^p \mathbf{A}$ can achieve better accuracy. The orthonormalization operation "orth($\cdot$)" is used to alleviate the round-off error in the floating-point computation. It can be implemented with a call to a packaged QR factorization (e.g., `qr(X, 0)` in Matlab).

The basic rPCA algorithm with the PI scheme has the following guarantee (Halko et al., 2011; Musco and Musco, 2015):

$$||\mathbf{A} - \mathbf{QQ}^{\mathrm{T}}\mathbf{A}|| = ||\mathbf{A} - \mathbf{USV}^{\mathrm{T}}|| \leq (1 + \varepsilon)||\mathbf{A} - \mathbf{A}_k||, \qquad (3)$$

with a high probability. Here, $\mathbf{A}_k$ is the best rank-k approximation of $\mathbf{A}$.

Assuming that multiplying an $m \times n$ sparse matrix $\mathbf{A}$ and a $n \times l$ dense matrix costs $C_{mul}nnz(\mathbf{A})l$ *flops*, performing QR factorization of an $m \times n$ matrix costs $C_{qr}mn \min(m, n)$ *flops* and performing the economic SVD on an $m \times n$ matrix costs $C_{svd}mn \min(m, n)$ *flops*, we can analyze the *flop* count of Alg. 1. Using FC_1 to denote it, we have:

$$\text{FC}_1 = pC_{qr}nl^2 + (p+1)C_{qr}ml^2 + (2p+2)C_{mul}nnz(A)l + C_{mul}mlk + C_{svd}nl^2. \qquad (4)$$

3. Methodology

3.1. The Ideas for Acceleration

Because many real data can be regarded as sparse matrix, accelerating the basic rPCA algorithm for sparse matrix is the focus. In Alg. 1, the matrix multiplication in Steps 2 and 7 occupy the majority of computing time if $\mathbf{A}$ is dense. However, this is not true for sparse matrix, and therefore optimizing other steps will bring substantial acceleration.

In existing work, some ideas have been proposed to accelerate the basic rPCA algorithm. In Voronin and Martinsson (2015), the idea of using eigendecomposition to compute SVD in Step 8 of Alg. 1 was proposed. It was pointed out that in the power iteration, orthonormalization after each matrix multiplication is not necessary. In Li et al. (2017), the power iteration was accelerated by replacing QR factorization with LU factorization, and the Gaussian matrix was replaced with the random matrix with uniform distribution. In Benjamin Erichson et al. (2017), the rPCA algorithm without power iteration was discussed for dense matrix in image or video processing problem. It employs a variant of the basic rPCA algorithm, where the random matrix is multiplied to the left of

A. The algorithm was accelerated by using sparse random matrices and using eigendecomposition to obtain the orthonormal basis of the subspace.

For handling sparse $\mathbf{A}$, we just use the Gaussian matrix for $\mathbf{\Omega}$, because other matrices may cause $\mathbf{A\Omega}$ rank-deficient. The useful ideas for faster randomized PCA for sparse matrix are:

- Use the eigendecomposition for computing economic SVD of $\mathbf{B}$,
- Replace the orthonormal $\mathbf{Q}$ with the left singular vector matrix $\mathbf{U}$,
- Perform $\mathbf{LU}$ factorization in the power iteration,
- Perform orthonormalization after every other matrix-matrix multiplication in power iteration.

Firstly, we formulate the eigendecomposition based SVD as an eigSVD algorithm (described in Alg. 2), where "eig($\cdot$)" computes the eigendecomposition and "spdiags($\cdot$)" is used for constructing a sparse diagonal matrix. In Alg. 2, the "diag($\cdot$)" in Step 3 is the function to transform a diagonal matrix to a vector. Step 4 is to construct a sparse diagonal matrix $\hat{\mathbf{S}} = \text{diag}(\mathbf{S})^{-1}$, where "./" is the element-wise division operator. The eigSVD algorithm's correctness is given as Lemma 1.

Algorithm 2 eigSVD

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$ ($m \geq n$)
Output: $\mathbf{U} \in \mathbb{R}^{m \times n}, \mathbf{S} \in \mathbb{R}^n, \mathbf{V} \in \mathbb{R}^{n \times n}$
 1: $\mathbf{B} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$
 2: $[\mathbf{V}, \mathbf{D}] = \text{eig}(\mathbf{B})$
 3: $\mathbf{S} = \text{sqrt}(\text{diag}(\mathbf{D}))$
 4: $\hat{\mathbf{S}} = \text{spdiags}(1./\mathbf{S}, 0, n, n)$
 5: $\mathbf{U} = \mathbf{AV}\hat{\mathbf{S}}$

Lemma 1 *The matrix $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ produced by Alg. 2 form the economic SVD of matrix $\mathbf{A}$.*

Proof Suppose $\mathbf{A}$ has SVD as (1). Since $m \geq n$,

$$\mathbf{A} = \mathbf{U}(:, 1:n)\hat{\mathbf{\Sigma}}\mathbf{V}^{\mathrm{T}} \tag{5}$$

where $\hat{\mathbf{\Sigma}}$, a square diagonal matrix, is the first n rows of $\mathbf{\Sigma}$. Eq. (5) is the economic SVD of $\mathbf{A}$. Then Step 1 computes

$$\mathbf{B} = \mathbf{A}^{\mathrm{T}}\mathbf{A} = \mathbf{V}\hat{\mathbf{\Sigma}}^2\mathbf{V}^{\mathrm{T}}. \tag{6}$$

The right-hand side of (6) is the eigendecomposition of $\mathbf{B}$. This means in Step 2, $\mathbf{D} = \hat{\mathbf{\Sigma}}^2$ and $\mathbf{V}$ is the right singular vector matrix of $\mathbf{A}$. Therefore, the values of $\mathbf{S}$ in Step 3 are the diagonal elements of $\hat{\mathbf{\Sigma}}$ and the $\hat{\mathbf{S}}$ in Step 4 equals to $\hat{\mathbf{\Sigma}}^{-1}$. In Step 5, $\mathbf{U} = \mathbf{AV}\hat{\mathbf{S}} = \mathbf{AV}\hat{\mathbf{\Sigma}}^{-1} = \mathbf{U}(:, 1:n)$. The last equality is derived from (5). This proves the lemma. ∎

According to Alg. 2, the *flop* count of the eigSVD algorithm is:

$$\text{FC}_2 = 2C_{mul}mn^2 + C_{eig}n^3. \tag{7}$$

We assume performing the eigendecomposition of an $n \times n$ matrix costs $C_{eig}n^3$ *flops*. Notice that eigSVD algorithm is especially efficient if $m \gg n$, because $\mathbf{B}$ becomes a small $n \times n$ matrix. And, the computed singular values in $\mathbf{S}$ are in ascending order. Numerical issues can arise if matrix $\mathbf{A}$ dose not have full column rank. So, eigSVD algorithm is only applicable to special situations.

Notice that "eig(·)" in Step 2 of Alg. 2 can be replaced with "eigs(·)" to compute the largest k eigenvalues/eigenvectors, so that the algorithm can produce the results of truncated SVD. This results in an eigSVDs algorithm (see Algorithm 3), which can also be used to compute PCA.

Algorithm 3 eigSVDs

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(m \geq n)$, k
Output: $\mathbf{U} \in \mathbb{R}^{m \times k}, \mathbf{S} \in \mathbb{R}^k, \mathbf{V} \in \mathbb{R}^{k \times n}$
  1: $\mathbf{B} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$
  2: $[\mathbf{V}, \mathbf{D}] = \mathrm{eigs}(\mathbf{B}, k)$
  3: $\mathbf{S} = \mathrm{sqrt}(\mathrm{diag}(\mathbf{D}))$
  4: $\hat{\mathbf{S}} = \mathrm{spdiags}(1./\mathbf{S}, 0, k, k)$
  5: $\mathbf{U} = \mathbf{A}\mathbf{V}\hat{\mathbf{S}}$

Secondly, the idea that the orthonormal $\mathbf{Q}$ can be replaced with the left singular matrix $\mathbf{U}$ can be explained with Lemma 2.

Lemma 2 *In the basic rPCA algorithm, orthonormal matrix $\mathbf{Q}$ includes a set of orthonormal basis of subspace $range(\mathbf{A}\mathbf{\Omega})$ or $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}\mathbf{\Omega})$. As long as $\mathbf{Q}$ holds this property, no matter how it is produced, the result of basic rPCA algorithm will not change.*

Proof From Step 2 of Alg. 1 we see that $\mathbf{Q}$ is an orthonormal matrix, and its columns are a set of orthonormal basis of subspace $range(\mathbf{A}\mathbf{\Omega})$. If $p > 0$, from Steps 3∼6 we can see that the orthonormal matrix $\mathbf{Q}$ includes the a set of orthonormal basis of subspace $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}\mathbf{\Omega})$. The result of the basic rPCA algorithm, is actually $\mathbf{Q}\mathbf{B} = \mathbf{Q}\mathbf{Q}^{\mathrm{T}}\mathbf{A}$, which further equals $\mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}}$. Notice that $\mathbf{Q}\mathbf{Q}^{\mathrm{T}}$ is an orthogonal projector onto the subspace $range(\mathbf{Q})$, if $\mathbf{Q}$ is an orthonormal matrix. The orthogonal projector is uniquely determined by the subspace (see (Golub and Van Loan, 1996) or Section 8.2 of (Halko et al., 2011)), i.e. $range(\mathbf{A}\mathbf{\Omega})$ or $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}\mathbf{\Omega})$. Therefore, As long as $\mathbf{Q}$ includes a set of orthonormal basis of the subspace, $\mathbf{Q}\mathbf{Q}^{\mathrm{T}}$ is identical and the basic rPCA algorithm's results will not change. ∎

Both QR factorization and SVD of a same matrix produce the orthonormal basis of its range space (column space), in $\mathbf{Q}$ and $\mathbf{U}$, respectively. Therefore, with Lemma 2, we see that $\mathbf{Q}$ can be replaced by $\mathbf{U}$ from SVD in the basic rPCA algorithm.

Thirdly, LU factorization is used in power iteration to replace QR factorization for saving runtime. This does not affecting the algorithm's correctness, which is proved in Lemma 3.

Lemma 3 *In the basic rPCA algorithm, the "orth(·)" operation in the power iteration, except the last one, can be replaced by LU factorization. This does not affect the algorithm's accuracy in exact arithmetic.*

Proof Firstly, if the "orth(·)" is not performed, the power iteration produces $\mathbf{Q}$ including a set of basis of the subspace $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}\mathbf{\Omega})$. As mentioned before, the "orth(·)" is just for alleviating the round-off error, and after using it $\mathbf{Q}$ still represents $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}\mathbf{\Omega})$.

The pivot LU factorization of a matrix $\mathbf{M}$ is:

$$\mathbf{P}\mathbf{M} = \mathbf{L}\mathbf{U}, \tag{8}$$

where $\mathbf{P}$ is a permutation matrix, and $\mathbf{L}$ and $\mathbf{U}$ are lower triangular and upper triangular matrices, respectively. Obviously, $\mathbf{M} = (\mathbf{P}^{\mathrm{T}}\mathbf{L})\mathbf{U}$, where $\mathbf{P}^{\mathrm{T}}\mathbf{L}$ has the same column space as $\mathbf{M}$. Therefore, replacing "orth($\cdot$)" with LU factorization (using $\mathbf{P}^{\mathrm{T}}\mathbf{L}$) also produces the basis of $range((\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\mathbf{A}\boldsymbol{\Omega})$. Then, based on Lemma 2, this does not affect the algorithm's result in exact arithmetic. ∎

Notice that the LU factor $\mathbf{P}^{\mathrm{T}}\mathbf{L}$ has scaled matrix entries with linearly independent columns, since $\mathbf{L}$ is a lower triangular matrix with unit diagonals and $\mathbf{P}$ just means row permutation. Therefore, it also alleviates the round-off error.

Finally, the orthonormalization or LU factorization in power iteration can be performed after every other matrix multiplication. It harms the accuracy little but remarkably reduces runtime.

3.2. A Modified Power Iteration Scheme and Handling Matrix with More Columns

For a sparse matrix $\mathbf{A}$, the power iteration in the basic rPCA algorithm (Alg. 1) is computationally expensive, because it includes the multiplication of two dense matrices. We also notice that, each time we increase the power parameter by one, two matrix multiplications are induced resulting in large increase of computation cost. This makes inconvenient trade-off between runtime and accuracy. To alleviate this issue, we here propose a modified power iteration scheme, which allows odd number of passes over $\mathbf{A}$ and thus provides more convenient performance trade-off of the rPCA algorithm.

We first observe that, if the power parameter $p > 0$, Steps 1 and 2 of Alg. 1 can be simply replaced with:

1: $\mathbf{Q} = \boldsymbol{\Omega} = \mathrm{randn}(m, k + s)$

For the same power parameter p, this reduces one pass over matrix $\mathbf{A}$. Because the singular values of $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^p$ decay more quickly than $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^{p-1}\mathbf{A}$, performing randomized QB procedure on $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^p$ is more accurate than $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^{p-1}\mathbf{A}$. It means:

$$||\mathbf{A} - \hat{\mathbf{Q}}\hat{\mathbf{Q}}^{\mathrm{T}}\mathbf{A}|| < ||\mathbf{A} - \mathbf{Q}\mathbf{Q}^{\mathrm{T}}\mathbf{A}||, \tag{9}$$

where $\hat{\mathbf{Q}}$ is the orthogonal matrix produced by $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^p\boldsymbol{\Omega}$ while $\mathbf{Q}$ is produced from $(\mathbf{A}\mathbf{A}^{\mathrm{T}})^{p-1}\mathbf{A}\boldsymbol{\Omega}$. This proves the rationality of this modification. Therefore, we can just modify Step 1 and 2 in Alg. 1 like this without other modification, to realize the odd number of passes over $\mathbf{A}$.

Another modification of Alg. 1 can be motivated by the observation that the *flop* count of Alg. 1, i.e. (4), is not favorable to the case with $m < n$, because C_{svd} is much large than other constants (C_{qr} and C_{mul}). Although we may run the algorithm to process $\mathbf{A}^{\mathrm{T}}$, the transpose of a sparse matrix is not easily obtained due to the storage format of sparse matrix.

Actually, there is a variant of the basic rPCA algorithm (Benjamin Erichson et al., 2017; Li et al., 2017), where the random matrix is multiplied to the left of $\mathbf{A}$. With the same idea, we derive an algorithm called basic rPCAt described as Alg. 4. Its flop count is:

$$\mathrm{FC}_4 = pC_{qr}ml^2 + (p + 1)C_{qr}nl^2 + (2p + 2)C_{mul}nnz(A)l + C_{mul}nlk + C_{svd}ml^2. \tag{10}$$

Therefore, we can derive that when Alg. 1 and Alg. 4 handle a sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m < n$,

$$\mathrm{FC}_1 - \mathrm{FC}_4 = (C_{svd} - C_{qr} - C_{mul})(n - m)l^2 > 0. \tag{11}$$

The reason is that C_{svd} is much larger than C_{qr} and C_{mul}. Eq. (11) shows Alg. 4 is more efficient than Alg. 1 when handling the matrix with more columns. Thus, we shall choose between them according to the matrix's dimensions, so as to achieve the best runtime performance.

Algorithm 4 basic rPCAt

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$, rank parameter k, power parameter p
Output: $\mathbf{U} \in \mathbb{R}^{m \times k}$, $\mathbf{S} \in \mathbb{R}^{k \times k}$, $\mathbf{V} \in \mathbb{R}^{n \times k}$
  1: $\mathbf{\Omega} = \text{randn}(k + s, m)$
  2: $\mathbf{Q} = \text{orth}((\mathbf{\Omega A})^{\text{T}})$
  3: **for** $i = 1, 2, \cdots, p$ **do**
  4:    $\mathbf{G} = \text{orth}(\mathbf{AQ})$
  5:    $\mathbf{Q} = \text{orth}(\mathbf{A}^{\text{T}}\mathbf{G})$
  6: **end for**
  7: $\mathbf{B} = (\mathbf{AQ})^{\text{T}}$
  8: $[\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{V}}] = \text{svd}(\mathbf{B})$
  9: $\mathbf{V} = \mathbf{Q}\hat{\mathbf{U}}$
 10: $\mathbf{U} = \hat{\mathbf{V}}(:, 1 : k), \mathbf{S} = \hat{\mathbf{S}}(1 : k, 1 : k), \mathbf{V} = \mathbf{V}(:, 1 : k)$

3.3. The Fast PCA Algorithm for Sparse Data

Based on Section 3.1 , we find out that the eigSVD procedure can be applied to the basic rPCA algorithm to produce both the economic SVD of $\mathbf{B}$ and the orthonormal $\mathbf{Q}$. Because $k + s \ll \min(m, n)$ and matrices are not rank-deficient in practice, using eigSVD induces no numerical issue. With the accelerating skills, we propose a fast rPCA algorithm (*frPCA*) for sparse matrix (described as Alg. 5), where "lu$(\cdot)$" denotes the LU factorization and its first output is "$\mathbf{P}^{\text{T}}\mathbf{L}$". "$\lfloor \cdot \rfloor$" denotes the floor function which returns the maximum integer no larger than the input value, and the pass parameter q represents the number of passes over $\mathbf{A}$ in the whole algorithm. The equivalence between the frPCA algorithm and the basic rPCA algorithm is demonstrated as Theorem 1.

Theorem 1 *The frPCA algorithm (Alg. 5) is mathematically equivalent to the basic rPCA algorithm (Alg. 1) when $p = (q - 2)/2$.*

Proof When $p = (q - 2)/2$, the number of power iteration is the same for the both algorithms. One difference between Alg. 5 and Alg. 1 is in the power iteration (the "for" loop). Based on Lemma 1 we see that eigSVD accurately produces a set of orthonormal basis. Besides, based on Lemma 2 and 3, we see the power iteration in Alg. 5 is mathematically equivalent to that in Alg. 1. The other difference is the last three steps in Alg. 5. Its correctness is due to Lemma 1 and that the singular values produces by eigSVD is in the ascending order. ∎

Below we analyze the flop count of Alg. 5. Suppose the flop count of multiplication of $\mathbf{M} \in \mathbb{R}^{m \times l}$ and $\mathbf{N} \in \mathbb{R}^{l \times l}$ is $C_{mul}ml^2$. Here, C_{mul} reflects one addition and one multiplication. If LU factorization is performed on $\mathbf{M}$, it takes $ml^2 - l^3/2$ times minus and multiplication operations, denoted by $C_{lu}(ml^2 - l^3/2)$ flop counts. If $l \ll m$, we see that LU factorization costs similar runtime as the the matrix multiplication. So, for the purpose of runtime comparison, we assume

---

Algorithm 5 frPCA

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(m \leq n)$, k, pass parameter $q \geq 2$
Output: $\mathbf{U} \in \mathbb{R}^{m \times k}$, $\mathbf{S} \in \mathbb{R}^{k}$, $\mathbf{V} \in \mathbb{R}^{n \times k}$

1: **if** q is an even number **then**
2:    $\mathbf{Q} = \text{randn}(n, k + s)$
3:    $\mathbf{Q} = \mathbf{A}\mathbf{Q}$
4:    **if** $q > 2$ **then** $[\mathbf{Q}, \sim] = \text{lu}(\mathbf{Q})$ **else** $[\mathbf{Q}, \sim, \sim] = \text{eigSVD}(\mathbf{Q})$
5: **else**
6:    $\mathbf{Q} = \text{randn}(m, k + s)$
7: **end if**
8: **for** $i = 1, 2, 3, \cdots, \lfloor \frac{q-1}{2} \rfloor$ **do**
9:    **if** $i == \lfloor \frac{q-1}{2} \rfloor$ **then**
10:      $[\mathbf{Q}, \sim, \sim] = \text{eigSVD}(\mathbf{A}(\mathbf{A}^{\mathrm{T}}\mathbf{Q}))$
11:    **else**
12:      $[\mathbf{Q}, \sim] = \text{lu}(\mathbf{A}(\mathbf{A}^{\mathrm{T}}\mathbf{Q}))$
13:    **end if**
14: **end for**
15: $[\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{V}}] = \text{eigSVD}(\mathbf{A}^{\mathrm{T}}\mathbf{Q})$
16: $ind = k + s : -1 : s + 1$
17: $\mathbf{U} = \mathbf{Q}\hat{\mathbf{V}}(:, ind), \mathbf{V} = \hat{\mathbf{U}}(:, ind), \mathbf{S} = \hat{\mathbf{S}}(ind)$

---

that $C_{mul}ml^2 \approx C_{lu}(ml^2 - l^3/2)$ in the following analysis. Considering that $l \ll \min(m, n)$, we derive the flop count of Alg. 5 for the situation with q equal an even number:

$$
\begin{aligned}
\text{FC}_5 &= (\frac{q}{2} - 1)C_{lu}(ml^2 - \frac{l^3}{2}) + qC_{mul}nnz(A)l + C_{mul}mlk + 2C_{mul}(m + n)l^2 + 2C_{eig}l^3 \\
&\approx (\frac{q}{2} - 1)C_{mul}ml^2 + qC_{mul}nnz(A)l + C_{mul}mlk + 2C_{mul}(m + n)l^2.
\end{aligned}
$$
(12)

As we will see soon, Alg. 5 is more efficient for handling matrix $\mathbf{A}$ with dimension $m < n$. So, we also propose a variant fast rPCA algorithm (denoted by *frPCAt*) through applying the accelerating skills to Alg. 4. The resulted algorithm is described as Alg. 6.

Theorem 2 *The variant fast rPCA algorithm (Alg. 6) is mathematically equivalent to the basic rPCAt algorithm (Alg. 4) when $p = (q - 2)/2$.*

Proof When $p = (q - 2)/2$, the number of power iteration is the same. The differences between Alg. 6 and Alg. 4 are also in the power iteration and the last three steps in Alg. 6. Based on Lemma 1 we see that eigSVD accurately produces a set of orthonormal basis. Besides, based on Lemma 2 and 3, we see the power iteration in Alg. 6 is mathematically equivalent to that in Alg. 4. The correctness of last three steps is due to Lemma 1 and that the singular values produces by eigSVD is in the ascending order. ∎

Algorithm 6 frPCAt

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(m \geq n)$, k, pass parameter $q \geq 2$
Output: $\mathbf{U} \in \mathbb{R}^{m \times k}$, $\mathbf{S} \in \mathbb{R}^{k}$, $\mathbf{V} \in \mathbb{R}^{n \times k}$

1: **if** q is an even number **then**
2:     $\mathbf{Q} = \text{randn}(k + s, m)$
3:     $\mathbf{Q} = (\mathbf{QA})^{\mathrm{T}}$
4:     **if** $q == 2$ **then** $[\mathbf{Q}, \sim, \sim] = \text{eigSVD}(\mathbf{Q})$ **else** $[\mathbf{Q}, \sim] = \text{lu}(\mathbf{Q})$
5: **else**
6:     $\mathbf{Q} = \text{randn}(n, k + s)$
7: **end if**
8: **for** $i = 1, 2, 3, \cdots, \lfloor \frac{q-1}{2} \rfloor$ **do**
9:     **if** $i == \lfloor \frac{q-1}{2} \rfloor$ **then**
10:       $[\mathbf{Q}, \sim, \sim] = \text{eigSVD}(\mathbf{A}^{\mathrm{T}}(\mathbf{AQ}))$
11:     **else**
12:       $[\mathbf{Q}, \sim] = \text{lu}(\mathbf{A}^{\mathrm{T}}(\mathbf{AQ}))$
13:     **end if**
14: **end for**
15: $[\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{V}}] = \text{eigSVD}(\mathbf{AQ})$
16: $ind = k + s : -1 : s + 1$
17: $\mathbf{U} = \hat{\mathbf{U}}(:, ind), \mathbf{V} = \mathbf{Q}\hat{\mathbf{V}}(:, ind), \mathbf{S} = \hat{\mathbf{S}}(ind)$

Similarly, we can analyze the *flop* count of the variant fast rPCA algorithm (Alg. 6).

$$
\begin{aligned}
\text{FC}_6 &= (\frac{q}{2} - 1)C_{lu}(nl^2 - \frac{l^3}{2}) + qC_{mul}nnz(A)l + C_{mul}nlk + 2C_{mul}(m + n)l^2 + 2C_{eig}l^3 \\
&\approx (\frac{q}{2} - 1)C_{mul}nl^2 + qC_{mul}nnz(A)l + C_{mul}nlk + 2C_{mul}(m + n)l^2.
\end{aligned}
\tag{13}
$$

Now, the difference of flop counts of Alg. 5 and 6 is

$$
\text{FC}_6 - \text{FC}_5 = (\frac{q}{2} - 1)C_{lu}(n - m)l^2 + C_{mul}(n - m)lk < 0,
\tag{14}
$$

if they are performed on $\mathbf{A} \in \mathbb{R}^{m \times n}$ $(m > n)$. It means Alg. 6 is more efficient for handling matrix with more rows. Accordingly, Alg. 5 is more efficient for handling matrix with more columns.

To evaluate how the proposed fast PCA algorithm accelerates the basic rPCA algorithm, we give the following analysis on the theoretical speedup based on flop counts. As Alg. 1 and Alg. 6 are both efficient for the situation with $m \geq n$, we analyze the ratio of flop counts of them. According to (4) and (13), the speedup ratio of the *frPCAt* algorithm to the basic rPCA algorithm is (assuming that $p = (q - 2)/2$):

$$
\begin{aligned}
\text{Sp1} &= \frac{\text{FC}_1}{\text{FC}_6} \\
&\approx \frac{(\frac{q}{2} - 1)C_{qr}nl^2 + \frac{q}{2}C_{qr}ml^2 + qC_{mul}nnz(A)l + C_{mul}mlk + C_{svd}nl^2}{(\frac{q}{2} - 1)C_{mul}nl^2 + qC_{mul}nnz(A)l + C_{mul}nlk + 2C_{mul}(m + n)l^2}.
\end{aligned}
\tag{15}
$$

Denote $t = nnz(\mathbf{A})/m$ as the average number of nonzeros per row, $\alpha = t/l$ as a sparsity parameter related to the rank parameter, and $\beta = n/m$ as a matrix shape parameter ($\beta \leq 1$). We further

derive:

$$\text{Sp1} \approx \frac{(\frac{q}{2}-1)C_{qr}\beta + \frac{q}{2}C_{qr} + qC_{mul}\alpha + C_{mul} + C_{svd}\beta}{(\frac{q}{2}+1)C_{mul}\beta + qC_{mul}\alpha + C_{mul}\beta + 2C_{mul}}. \quad (16)$$

Based on this, we have the following theorem.

Theorem 3 *The speedup ratio of the variant fast PCA algorithm (Alg. 6) to the basic rPCA algorithm (Alg. 1), Sp1, depends on the number of passes over* $\mathbf{A}$ *(denoted by q), the ratio of average number of nonzeros per row to the rank parameter l (denoted by α), and the number of columns over the number of rows (denoted by β). Sp1 becomes higher as α decreases. And,*

$$\lim_{q\to\infty} \text{Sp1} = \frac{C_{qr}\beta + C_{qr} + 2C_{mul}\alpha}{C_{mul}\beta + 2C_{mul}\alpha}, \quad (17)$$

which approaches to $2C_{qr}/C_{mul}$ *for a very sparse square matrix* $\mathbf{A}$ *(α is small and β equals 1). Here, C_{qr} and C_{mul} are the constants for the flop counts of QR factorization and matrix-matrix multiplication respectively.*

Proof Firstly, based on (15) and (16), we can derive the derivative of Sp1 with respect to α:

$$\frac{\partial \text{Sp1}}{\partial \alpha} = \frac{qC_{mul}(\text{FC}_6 - \text{FC}_1)}{\text{FC}_6{}^2} < 0, \quad (18)$$

where FC_1 and FC_6 represents the *flop* counts of Alg. 1 and 6, respectively. This means Sp1 increases with the decrease of α (which means matrix becomes sparser).

Then, when q is sufficiently large, we can derive:

$$\text{Sp1} \approx \frac{(\frac{q}{2}-1)C_{qr}\beta + \frac{q}{2}C_{qr} + qC_{mul}\alpha}{(\frac{q}{2}+1)C_{mul}\beta + qC_{mul}\alpha} \approx \frac{C_{qr}\beta + C_{qr} + 2C_{mul}\alpha}{C_{mul}\beta + 2C_{mul}\alpha}, \quad (19)$$

which results in (17). Finally, if $\alpha \to 0$ and $\beta = 1$ the speedup ratio approaches to $2C_{qr}/C_{mul}$. This is the upper bound of the speedup for a square or approximately square $\mathbf{A}$, absolutely greater than 1. ∎

A similar theorem can be derived for Alg. 5. With the theorems, we see that the proposed fast rPCA algorithm accelerates the basic rPCA algorithm without loss of accuracy. Besides, it allows odd number of passes over matrix $\mathbf{A}$, providing better trade-off between runtime and accuracy.

4. Experiments

All experiments are carried out on a Linux server with two 12-core Intel Xeon E5-2630 CPUs (2.30 GHz), and 32 GB RAM. The proposed algorithms Alg. 5 and Alg. 6 are implemented in C with MKL libraries (Int, 2018) and OpenMP derivatives for multi-thread computing. QR factorization, LU factorization and other basic linear algebra operations are realized through LAPACK routines which are automatically executed in parallel on the multi-core CPUs. svds in Matlab2016b are used as accurate truncated SVD. eigSVDs is the other algorithm to compare and is efficiently implemented in Matlab2016b. Because the lansvd in Matlab/Fortran is not well parallelized, it runs

slower than `svds` in our experiments. And, considering that $k \ll \min(m, n)$, the method calculating all the singular values/vectors by eigSVD and then making truncation is not competitive in runtime. Therefore, we do not include `lansvd` and eigSVD in the comparisons.

In the experiments, we choose Alg. 5 or 6 as the proposed fast algorithm according to the shape of test matrix. The oversampling parameter is always set $s = 5$, and all runtimes are in seconds.

4.1. Accuracy and Efficiency Validation

In this subsection, we test different sparse matrices from real data and show how the performance of the proposed algorithm is affected by various factors.

Firstly, we obtain a sparse matrix in size 90,230 × 45,115 from the MovieLens dataset (Harper and Konstan, 2016). The matrix has 97 nonzeros per row on average and is denoted by Matrix 1. Then, we randomly set some nonzero elements to zero to get two sparser matrices: Matrix 2 and 3 with 24 and 9 nonzeros per row on average, respectively. We also obtain three matrices (Matrix 4-6) from the information retrieval application "AMiner" (Ami, 2018). They are in size 647,789 × 323,896, and have about 16, 8 and 4 nonzeros per row on average, respectively. We compute the first 100 singular values and singular vectors with `svds`, eigSVDs, Alg. 1 (setting $p = 5$) and Alg. 6 (setting $q = 11$). The runtimes are listed in Table 1 and the computed singular values are drawn in Figure 1. We use "Sp2" to denote the speedup ratio to `svds`, which is different from Sp1.

Table 1: The runtimes of different PCA algorithms for matrices with different sparsity.

Algorithm	Matrix 1		Matrix 2		Matrix 3		Matrix 4		Matrix 5		Matrix 6	
	time	Sp2	time	Sp2	time	Sp2	time	Sp2	time	Sp2	time	Sp2
svds	36.0	*	25.5	*	21.0	*	178.9	*	149.5	*	131.1	*
eigSVDs	459.2	0.1	104.6	0.2	37.2	0.6	278.7	0.6	156.2	1.1	75.7	1.7
Alg.1 ($p = 5$)	13.1	2.8	10.0	2.5	9.76	2.2	99.9	1.8	90.7	1.6	84.5	1.6
Alg.6 ($q = 11$)	4.32	**8.3**	1.58	**16**	1.05	**20**	17.2	**10**	13.8	**11**	10.2	**13**

Table 1 shows that the speedup ratio of Alg. 6 increases when $nnz(\mathbf{A})$ decrease, no matter compared with `svds` or to Alg. 1. The proposed algorithm is up to **20X** faster than `svds` and **9.1X** faster than the basic rPCA algorithm (both achieved on Matrix 3). In Fig. 1, the curves of eigSVDs is not shown, as they are indistinguishable to those of `svds`. From the figure, we see that the randomized PCA algorithms are indistinguishable from `svds` at the first tens of singular values. Alg. 6 is also indistinguishable from Alg. 1. This validates the effectiveness of the proposed algorithm with odd number of passes over $\mathbf{A}$.

Fig. 2(a) shows the first principal component (i.e. $\mathbf{u}_1$) of Matrix 2 computed by `svds` and Alg. 6, which looks indistinguishable (only 1.4×10^{-10} difference in l_∞-norm). For other principal components, we calculate the correlation coefficient between the results obtained with the both methods. As shown in Fig. 2(b), the correlation coefficients are close to 1. The largest deviation occurs for the 29th principal component, with value 0.9988.

Secondly, we test the randomized algorithms with different q and p parameters. $q = 2, 4, 6, 9, 11$ and $p = 0, 1, 2, 4, 5$ are set to Alg. 6 and Alg. 1 respectively. The runtimes of the both algorithms are listed in Table 2, for computing the first 100 principal components of Matrix 2.

From the table, we see that the speedup ratio increases with q. At the same time, we plot Fig. 3 to show the curves of computed singular values. From it we see with q or p increasing the singular values approach to the accurate values. And, since the proposed algorithm allows odd number of
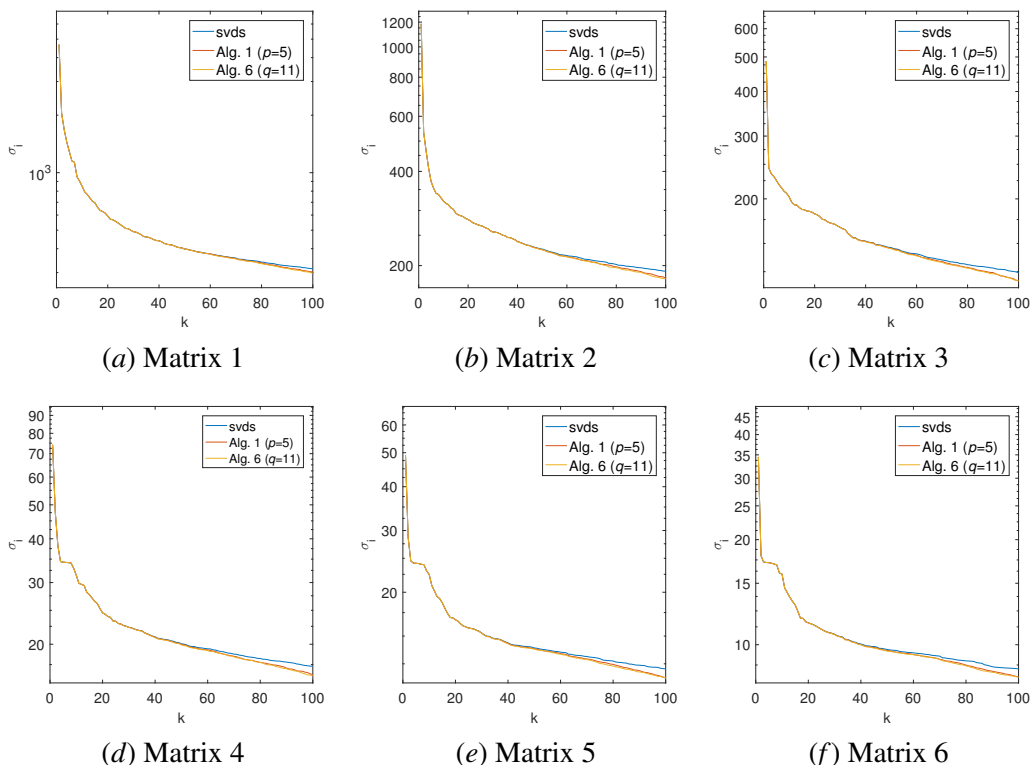
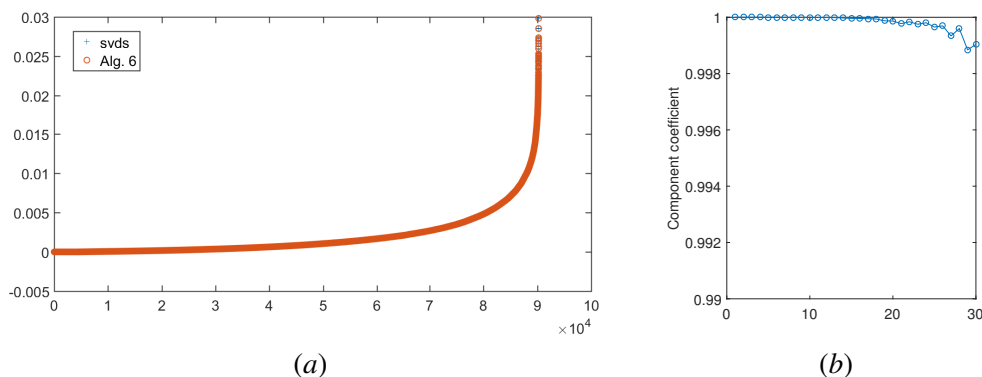Figure 1: The computed singular values for different matrices, showing the accuracy of our algorithm.
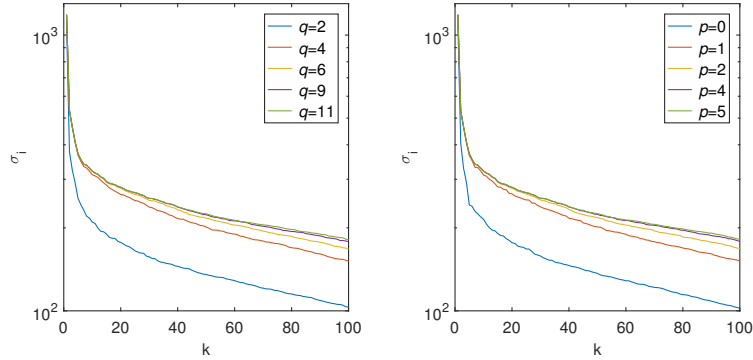

Figure 2: The accuracy of Alg. 6 ($q = 11$) on principal components of Matrix 2 (with comparison to the results from svds). (a) The numeric values (sorted) of first principal component. (b) The correlation coefficients for the first 30 principal components.

passes over **A**, it has better flexibility. If lower accuracy is allowed, the *frPCAt* algorithm runs much faster. For example, setting $q = 4$, it's actually 27X faster than svds.

Lastly, we construct matrices by modifying the dimensions of matrix. We only keep the first 107,966 columns of Matrix 6 to obtain Matrix 7 and the first 107,966 rows of Matrix 6 to get Matrix 8. For them we test different PCA algorithms. The results are listed in Table 3. From it we see that

Table 2: The runtimes of the basic rPCA algorithm and the proposed algorithm with different q values.

Algorithm	$q = 2$		$q = 4$		$q = 6$		$q = 9$		$q = 11$	
	time	Sp1	time	Sp1	time	Sp1	time	Sp1	time	Sp1
Alg.1 ($p = \lfloor (q-1)/2 \rfloor$)	2.03	*	3.70	*	5.32	*	8.48	*	10.0	*
Alg.6	0.69	2.9	0.94	3.9	1.04	5.1	1.32	6.4	1.58	6.3

(a) frPCAt ($q = 2, 4, 6, 9, 11$)   (b) basic rPCA ($p = 0, 1, 2, 4, 5$)

Figure 3: The computed singular values of basic rPCA algorithm and proposed algorithm with different q values ($p = \lfloor (q-1)/2 \rfloor$).

eigSVDs (Alg. 3) is more efficient than `svds` when m is much larger than n. And, Alg. 5 runs faster than Alg. 6, if when $m < n$. This validates the analysis in Section 3.3.

Table 3: The runtimes of different PCA algorithms for matrices with different dimensions.

Algorithm	Matrix 6 647,789×323,896		Matrix 7 647,989×107,966		Matrix 8 107,966×323,896	
	time	Sp2	time	Sp2	time	Sp2
svds	131.1	*	100.4	*	51.3	*
eigSVDs	75.7	1.7	16.1	6.3	47.0	1.1
Alg.1 ($p = 5$)	84.5	1.6	59.6	1.7	35.6	1.4
Alg.5 ($q = 11$)	14.2	12	7.19	14	2.78	**18**
Alg.6 ($q = 11$)	10.2	**13**	5.91	**18**	3.79	14

4.2. Results on Real Large Data

In this subsection, we test the proposed algorithm with three large real datasets. The first one is a large matrix from MovieLens in size $270,896 \times 45,115$ with 97 nonzeros per row on average. The Aminer person-keyword matrix is the second dataset, in the largest size $12,869,521 \times 323,899$, with 16 nonzeros per row on average. The last one is a social network matrix from SNAP (Leskovec and Krevl, 2014) in size $82,168 \times 82,168$ with 12 nonzeros per row on average. We computed the first 100 principal components/directions. The runtimes of different algorithms are listed in Table 4.

In the experiment, `svds` and eigSVDs failed on the Aminer dataset due to the limit of memory. The rPCA algorithms performed very well. The speedup ratio of Alg. 6 to `svds` is up to 13X. Assuming that $C_{qr} = 5C_{mul}$ in practice, we can see the Sp1 on MovieLens and SNAP are 3.7 and

Table 4: The runtimes of different PCA algorithms for real large sparse matrices.

Sparse Data	svds	eigSVDs	Alg. 1 ($p=5$)	Alg. 6 ($q=12$)		
	time	time	time	time	Sp1	Sp2
MovieLens	108.4	566.2	34.8	12.5	2.8	8.5
Aminer	*	*	1448.3	398.7	3.6	*
SNAP	22.7	124.4	14.8	1.74	8.7	13

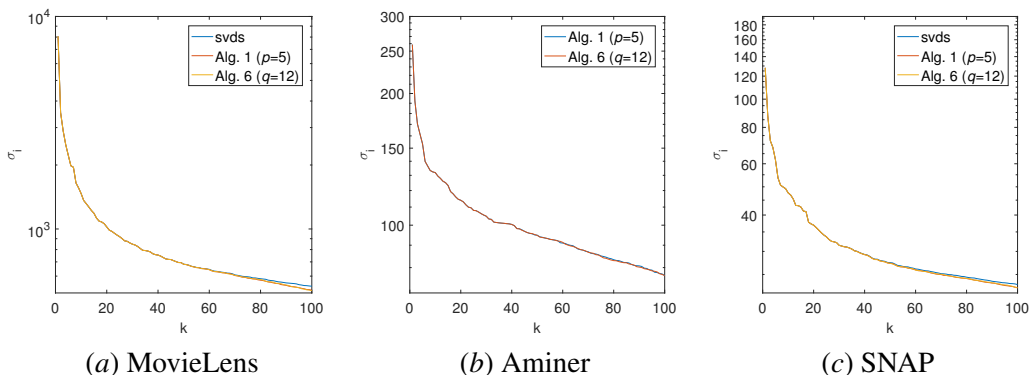

(a) MovieLens   (b) Aminer   (c) SNAP

Figure 4: The computed singular values of MovieLens, Aminer and SNAP matrix.

8.3 according to Eq. (19). They approximate the Sp1 in Table 4, which validates the analysis in Theorem 3. In Figure 4, we plot the computed singular values, showing the good accuracy of the proposed algorithm. The memory costs of `svds`, Alg. 1 and Alg. 6 are 1.1 GB, 1.0 GB and 0.87 GB on MovieLens and 0.58 GB, 0.55 GB and 0.35 GB on SNAP, respectively. They suggest that the rSVD algorithms need less memory than `svds`. For the largest dataset, Aminer, the memory costs of Alg. 1 and Alg. 6 are 25 GB and 23GB, respectively, while `svds` fails due to out-of-memory.

5. Conclusions

A fast randomized PCA algorithm including several techniques is proposed for sparse matrix. It is faster than `svds` and the basic rPCA algorithm. Its speedup ratio is up to 20X to `svds` and 9.1X to the basic rPCA algorithm. On real data from information retrieval, recommender system and network analysis, the proposed frPCA algorithm performs well, while `svds` and eigSVDs algorithm may fail due to large memory cost. The frPCA algorithm runs up to 13X faster than `svds` and 8.7X faster than basic rPCA algorithm for the network analysis dataset with little accuracy loss.

References

Aminer. https://www.aminer.cn, 2018.

Intel parallel studio xe cluster edition for linux. https://software.intel.com/en-us/intel-parallel-studio-xe, 2018.

N. Benjamin Erichson, Steven L. Brunton, and J. Nathan Kutz. Compressed singular value decomposition for image and video processing. In *Proc. IEEE International Conference on Computer Vision (ICCV)*, pages 1880–1888, Oct. 2017.

S. Deerwester, S. T. Dumais, G. W. Furnas, et al. Indexing by latent semantic analysis. *Journal of the American Society for Information Science & Technology*, 41(6):391–407, 1990.

Petros Drineas and Michael W Mahoney. Randnla: randomized numerical linear algebra. *Communications of the ACM*, 59(6):80–90, 2016.

Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

Gene H. Golub and Charles F. Van Loan. *Matrix Computations (the 3rd version)*. JHU Press, 1996.

N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

Rasmus Munk Larsen. Propack-software for large and sparse svd calculations. *Available online. URL http://sun.stanford.edu/rmunk/PROPACK*, 2004.

Richard Lehoucq, Daniel Sorensen, and Chao Yang. *ARPACK User's Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM Press, 1998.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

H. Li, G. C. Linderman, A. Szlam, K. P. Stanton, Y. Kluger, and M. Tygert. Algorithm 971: An implementation of a randomized algorithm for principal component analysis. *ACM Transactions on Mathematical Software*, 43(3):1–14, 2017.

Yingyu Liang, Maria-Florina F Balcan, Vandana Kanchanapally, and David Woodruff. Improved distributed principal component analysis. In *Advances in Neural Information Processing Systems*, pages 3113–3121, 2014.

Cameron Musco and Christopher Musco. Randomized block Krylov methods for stronger and faster approximate singular value decomposition. In *Advances in Neural Information Processing Systems*, pages 1396–1404, 2015.

Sergey Voronin and Per-Gunnar Martinsson. RSVDPACK: An implementation of randomized algorithms for computing the singular value, interpolative, and cur decompositions of matrices on multi-core and gpu architectures. *arXiv preprint arXiv:1502.05366*, 2015.

Wenjian Yu, Yu Gu, Jian Li, Shenghua Liu, and Yaohang Li. Single-pass PCA of large high-dimensional data. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3350–3356, Aug. 2017.

Wenjian Yu, Yu Gu, and Yaohang Li. Efficient randomized algorithms for the fixed-precision low-rank matrix approximation. *SIAM Journal on Matrix Analysis and Applications*, 39(3):1339–1359, 2018.