Randomized Matrix Decompositions Using R

N. Benjamin Erichson
University of St Andrews

Sergey Voronin
Tufts University

Steven L. Brunton
University of Washington

J. Nathan Kutz
University of Washington

Abstract

Matrix decompositions are fundamental tools in the area of applied mathematics, statistical computing, and machine learning. In particular, low-rank matrix decompositions are vital, and widely used for data analysis, dimensionality reduction, and data compression. Massive datasets, however, pose a computational challenge for traditional algorithms, placing significant constraints on both memory and processing power. Recently, the powerful concept of randomness has been introduced as a strategy to ease the computational load. The essential idea of probabilistic algorithms is to employ some amount of randomness in order to derive a smaller matrix from a high-dimensional data matrix. The smaller matrix is then used to compute the desired low-rank approximation. Such algorithms are shown to be computationally efficient for approximating matrices with low-rank structure. We present the R package **rsvd**, and provide a tutorial introduction to randomized matrix decompositions. Specifically, randomized routines for the singular value decomposition, (robust) principal component analysis, interpolative decomposition, and CUR decomposition are discussed. Several examples demonstrate the routines, and show the computational advantage over other methods implemented in R.

Keywords: dimension reduction, randomized algorithm, low-rank approximations, singular value decomposition, principal component analysis, CUR decomposition, R.

1. Introduction

In the era of "big data", vast amounts of data are being collected and curated in the form of arrays across the social, physical, engineering, biological, and ecological sciences. Analysis of the data relies on a variety of matrix decomposition methods which seek to exploit low-rank features exhibited by the high-dimensional data. Indeed, matrix decompositions are often the workhorse algorithms for scientific computing applications in the areas of applied mathematics, statistical computing, and machine learning. Despite our ever-increasing computational power, the emergence of large-scale datasets has severely challenged our ability to analyze data using traditional matrix algorithms. Moreover, the growth of data collection is far outstripping computational performance gains. The computationally expensive singular value decomposition (SVD) is the most ubiquitous method for dimensionality reduction, data processing and compression. The concept of randomness has recently been demonstrated as an effective strategy to easing the computational demands of low-rank approximations from matrix decompositions such as the SVD, thus allowing for a scalable architecture for modern "big data" applications. Throughout this paper, we make the following assumption: the data matrix to be approximated has low-rank structure, i.e., the rank is smaller than the ambient dimension of the measurement space.

1.1. Randomness as a computational strategy

Randomness is a fascinating and powerful concept in science and nature. Probabilistic concepts can be used as an effective strategy for designing better algorithms. By the deliberate introduction of randomness into computations (Motwani and Raghavan 1995), randomized algorithms have not only been shown to outperform some of the best deterministic methods, but they have also enabled the computation of previously infeasible problems. The Monte Carlo method, invented by Stan Ulam, Nick Metropolis and John von Neumann, is among the most prominent randomized methods in computational statistics as well as one of the "best" algorithms of the 20th century (Cipra 2000).

Over the past two decades, probabilistic algorithms have been established to compute matrix approximations, forming the field of randomized numerical linear algebra (Drineas and Mahoney 2016). While randomness is quite controversial, and is often seen as an obstacle and a nuisance, modern randomized matrix algorithms are reliable and numerically stable. The basic idea of probabilistic matrix algorithms is to employ a degree of randomness in order to derive a smaller matrix from a high-dimensional matrix, which captures the essential information. Thus, none of the "randomness" should obscure the dominant spectral information of the data as long as the input matrix features some low-rank structure. Then, a deterministic matrix factorization algorithm is applied to the smaller matrix to compute a near-optimal low-rank approximation. The principal concept is sketched in Figure 1.

Several probabilistic strategies have been proposed to find a "good" smaller matrix, and we refer the reader to the surveys by Mahoney (2011), Liberty (2013), and Halko, Martinsson, and Tropp (2011b) for an in-depth discussion, and theoretical results. In addition to computing the singular value decomposition (Sarlos 2006; Martinsson, Rokhlin, and Tygert 2011) and principal component analysis (Rokhlin, Szlam, and Tygert 2009; Halko, Martinsson, Shkolnisky, and Tygert 2011a), it has been demonstrated that this probabilistic framework can also be

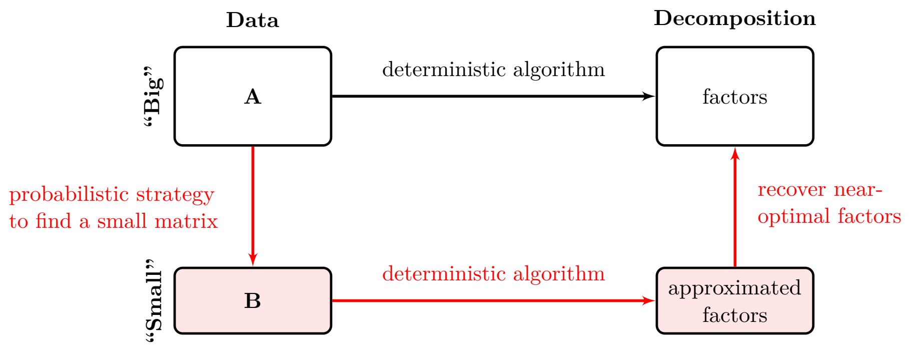


Figure 1: First, randomness is used as a computational strategy to derive a smaller matrix **B** from **A**. Then, the low-dimensional matrix is used to compute an approximate matrix decomposition. Finally, the near-optimal (high-dimensional) factors may be reconstructed.

used to compute the pivoted QR decomposition (Duersch and Gu 2017), the pivoted LU decomposition (Shabat, Shmueli, Aizenbud, and Averbuch 2016), and the dynamic mode decomposition (Erichson, Brunton, and Kutz 2017).

1.2. The rsvd package: Motivation and contributions

The computational costs of applying deterministic matrix algorithms to massive data matrices can render the problem intractable. Randomized matrix algorithms are becoming increasingly popular as an alternative, and implementations are available in a variety of programming languages and machine learning libraries. For instance, Voronin and Martinsson (2015) provide high performance, multi-core and GPU accelerated randomized routines in C.

The **rsvd** package aims to fill the gap in R, providing the following randomized routines:

- Randomized singular value decomposition: `rsvd()`.
- Randomized principal component analysis: `rpca()`.
- Randomized robust principal component analysis: `rrpca()`.
- Randomized interpolative decomposition: `rid()`.
- Randomized CUR decomposition: `rcur()`.

The routines are, in particular, efficient for matrices with rapidly decaying singular values. Figure 2 compares the computational performance of the `rsvd()` function to other existing SVD routines in R, which are discussed in more detail in Section 3.5. Specifically, for computing the dominant k singular values and vectors, the randomized singular value decomposition function `rsvd()` results in significant speedups over other existing SVD routines in R. See Section 3.8 for a more detailed performance evaluation.

The computational benefits of the randomized SVD translates directly to principal component analysis (PCA), since both methods are closely related. Further, the randomized SVD can be used to accelerate the computation of robust principal component analysis (RPCA). More generally, the concept of randomness allows also one to efficiently compute modern matrix

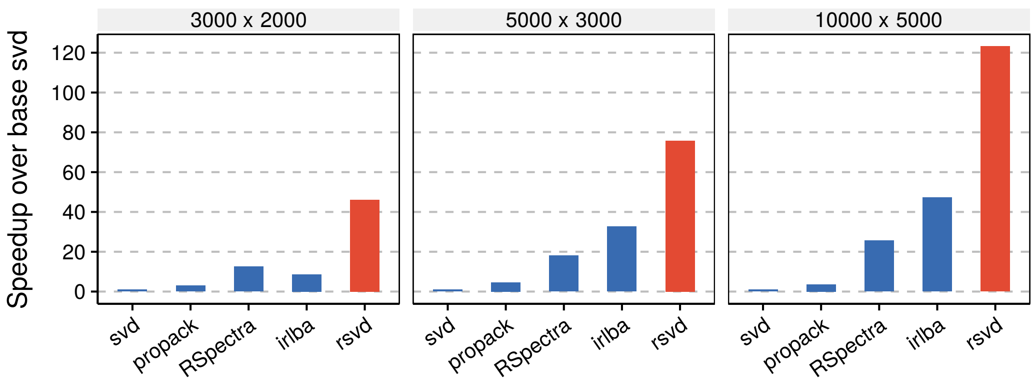


Figure 2: Runtime speedups (relative performance) of fast SVD algorithms compared to the base `svd()` routine in R. Here, the dominant $k = 20$ singular values and vectors are computed for random low-rank matrices with varying dimension $m \times n$. Note, here we are using Microsoft Open 3.5.1 which provides the Intel MKL for parallel mathematical computing (using 4 cores).

decompositions such as the interpolative decomposition (ID) and CUR decomposition. While the performance of the randomized algorithms depends on the actual shape of the matrix, we can state (as a rule of thumb) that significant computational speedups are achieved if the target rank k is about 3-6 times smaller than the smallest dimension $\min\{m, n\}$ of the matrix. The speedup for tall and thin matrices is in general less impressive than for "big" fat matrices.

The **rsvd** package is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=rsvd`. Thus, to install and load within R simply use:

```
R> install.packages("rsvd")
R> library("rsvd")
```

Alternatively, the package can be obtained via github: `https://github.com/erichson/rSVD`.

1.3. Organization

The remainder of this paper is organized as follows. First, Section 2 outlines the advocated probabilistic framework for low-rank matrix approximations. Section 3 briefly reviews the singular value decomposition and the randomized SVD algorithm. Then, the `rsvd()` function and its computational performance are demonstrated. Section 4 first describes the principal component analysis. Then, the randomized PCA algorithm is outlined, followed by the demonstration of the corresponding `rpca()` function. Section 5 outlines robust principal component analysis, and describes the randomized robust PCA algorithm as well as the `rrpca()` function. Section 6 gives a high-level overview of the interpolative and CUR decomposition. Finally, concluding remarks and a roadmap for future developments are presented in Section 7.

1.4. Notation

In the following we give a brief overview of some notation used throughout this manuscript.

Scalars are denoted by lower case letters x, and vectors both in $\mathbb{R}^n$ and $\mathbb{C}^n$ are denoted as bold lower case letters $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$. Matrices are denoted by bold capital letters $\mathbf{A}$ and the entry at row i and column j is denoted as $\mathbf{A}(i, j)$. This notation is convenient for matrix slicing, for instance, $\mathbf{A}(1 : i, :)$ extracts the first $1, 2, \ldots, i$ rows, and $\mathbf{A}(:, 1 : j)$ extracts the first $1, 2, \ldots, j$ columns. The transpose of a real matrix is denoted as $\mathbf{A}^\top$, and without loss of generality, we restrict most of the discussion in the following to real matrices. Further, the column space (range) of $\mathbf{A}$ is denoted as $\mathrm{col}(\mathbf{A})$, and the row space as $\mathrm{row}(\mathbf{A})$.

The spectral or operator norm of a matrix is defined as the largest singular value $\sigma_{\max}$ of $\mathbf{A}$, i.e., the square root of the largest eigenvalue $\lambda_{\max}$ of the positive-semidefinite matrix $\mathbf{A}^\top\mathbf{A}$:

$$\|\mathbf{A}\|_2 = \sqrt{\lambda_{\max}(\mathbf{A}^\top\mathbf{A})} = \sigma_{\max}(\mathbf{A}) = \max_{x \neq 0} \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2}.$$

The Frobenius norm is defined as the square root of the sum of the absolute squares of its elements, which is equal to the square root of the matrix trace of $\mathbf{A}^\top\mathbf{A}$:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |\mathbf{A}(i,j)|^2} = \sqrt{\mathrm{trace}(\mathbf{A}^\top\mathbf{A})}.$$

2. Probabilistic framework for low-rank approximations

Assume that a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ has rank r, where $r \leq \min\{m, n\}$. Then, in general, the objective of a low-rank matrix approximation is to find two smaller matrices such that:

$$\underset{m\times n}{\mathbf{A}} \approx \underset{m\times r}{\mathbf{E}} \; \underset{r\times n}{\mathbf{F}}, \tag{1}$$

where the columns of the matrix $\mathbf{E} \in \mathbb{R}^{m\times r}$ span the column space of $\mathbf{A}$, and the rows of the matrix $\mathbf{F} \in \mathbb{R}^{r\times n}$ span the row space of $\mathbf{A}$. The factors $\mathbf{E}$ and $\mathbf{F}$ can then be used to summarize or to reveal some interesting structure in the data. Further, the factors can be used to efficiently store the large data matrix $\mathbf{A}$. Specifically, while $\mathbf{A}$ requires mn words of storage, $\mathbf{E}$ and $\mathbf{F}$ require only $mr + nr$ words of storage.

In practice, most data matrices do not feature a precise rank r. Rather we are commonly interested in finding a rank-k matrix $\mathbf{A}_k$, which is as close as possible to an arbitrary input matrix $\mathbf{A}$ in the least-square sense. We refer to k as the target rank in the following.

In particular, modern data analysis and scientific computing largely rely on low-rank approximations, since low-rank matrices are ubiquitous throughout the sciences. However, in the era of "big data", the emergence of massive data poses a significant computational challenge for traditional deterministic algorithms.

In the following, we advocate the probabilistic framework, formulated by Halko *et al.* (2011b), to compute a near-optimal low-rank approximation. Conceptually, this framework splits the computational task into two logical stages:

- **Stage A:** Construct a low dimensional subspace that approximates the column space of $\mathbf{A}$. This means, the aim is to find a matrix $\mathbf{Q} \in \mathbb{R}^{m\times k}$ with orthonormal columns such that $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top\mathbf{A}$ is satisfied.
- **Stage B:** Form a smaller matrix $\mathbf{B} := \mathbf{Q}^\top\mathbf{A} \in \mathbb{R}^{k\times n}$, i.e., restrict the high-dimensional input matrix to the low-dimensional space spanned by the near-optimal basis $\mathbf{Q}$. The smaller matrix $\mathbf{B}$ can then be used to compute a desired low-rank approximation.

The first computational stage is where randomness comes into the play, while the second stage is purely deterministic. In the following, the two stages are described in detail.

2.1. The generic randomized algorithm

Stage A: Computing the near-optimal basis

First, we aim to find a near-optimal basis $\mathbf{Q}$ for the matrix $\mathbf{A}$ such that

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top\mathbf{A} \tag{2}$$

is satisfied. The desired target rank k is assumed to be $k \ll \min\{m, n\}$. Specifically, $\mathbf{P} := \mathbf{Q}\mathbf{Q}^\top$ is a linear orthogonal projector. A projection operator corresponds to a linear subspace, and transforms any vector to its orthogonal projection on the subspace. This is illustrated in Figure 3, where a vector $\mathbf{x}$ is confined to the column space $\text{col}(\mathbf{A})$.

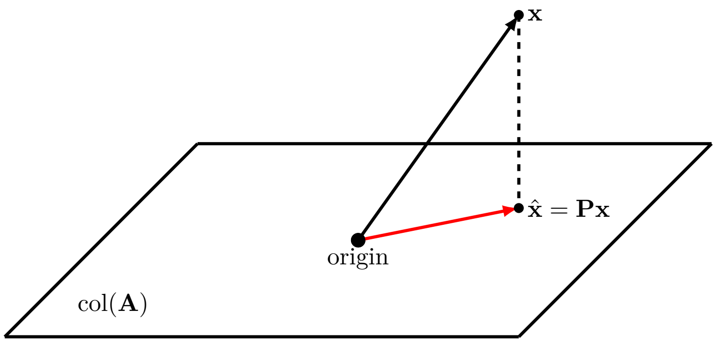


Figure 3: Geometric illustration of the orthogonal projection operator $\mathbf{P}$. A vector $\mathbf{x} \in \mathbb{R}^m$ is restricted to the column space of $\mathbf{A}$, where $\mathbf{Px} \in \mathrm{col}(\mathbf{A})$.

The concept of random projections can be used to sample the range (column space) of the input matrix $\mathbf{A}$ in order to efficiently construct such a orthogonal projector. Random projections are data agnostic, and constructed by first drawing a set of k random vectors $\{\boldsymbol{\omega}_i\}_{i=1}^k$, for instance, from the standard normal distribution. Probability theory guarantees that random vectors are linearly independent with high probability. Then, a set of random projections $\{\mathbf{y}_i\}_{i=1}^k$ is computed by mapping $\mathbf{A}$ to low-dimensional space:

$$\mathbf{y}_i := \mathbf{A}\boldsymbol{\omega}_i \quad \text{for } i = 1, 2, \ldots, k. \tag{3}$$

In other words, this process forms a set of independent randomly weighted linear combinations of the columns of $\mathbf{A}$, and reduces the number of columns from n to k. While the input matrix is compressed, the Euclidean distances between the original data points are approximately preserved. Random projections are also well known as the Johnson-Lindenstrauss (JL) transform (Johnson and Lindenstrauss 1984), and we refer to Ahfock, Astle, and Richardson (2017) for a recent statistical perspective.

Equation 3 can be efficiently executed in parallel. Therefore, let us define the random test matrix $\boldsymbol{\Omega} \in \mathbb{R}^{n\times k}$, which is again drawn from the standard normal distribution, and the columns of which are given by the vectors $\{\boldsymbol{\omega}_i\}$. The samples matrix $\mathbf{Y} \in \mathbb{R}^{m\times k}$, also denoted as sketch, is then obtained by post-multiplying the input matrix by the random test matrix

$$\mathbf{Y} := \mathbf{A}\boldsymbol{\Omega}. \tag{4}$$

Once $\mathbf{Y}$ is obtained, it only remains to orthonormalize the columns in order to form a natural basis $\mathbf{Q} \in \mathbb{R}^{m\times k}$. This can be efficiently achieved using the QR-decomposition $\mathbf{Y} =: \mathbf{QR}$, and it follows that Equation 2 is satisfied.

Stage B: Compute the smaller matrix

Now, given the near-optimal basis $\mathbf{Q}$, we aim to find a smaller matrix $\mathbf{B} \in \mathbb{R}^{k\times n}$. Therefore, we project the high-dimensional input matrix $\mathbf{A}$ to low-dimensional space

$$\mathbf{B} := \mathbf{Q}^\top\mathbf{A}. \tag{5}$$

Geometrically, this is a projection (i.e., a linear transformation) which takes points in a high-dimensional space into corresponding points in a low-dimensional space, illustrated in

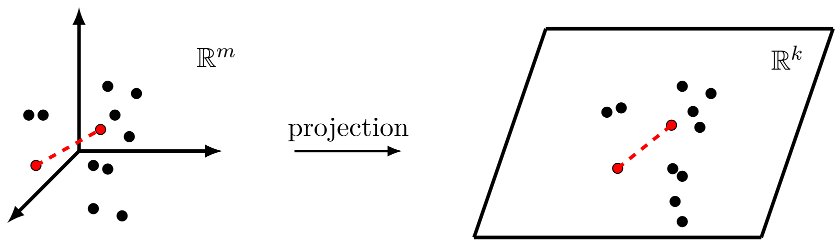


Figure 4: Points in a high-dimensional space are projected into low-dimensional space, while the geometric structure is preserved in an Euclidean sense.

Figure 4. This process preserves the geometric structure of the data in an Euclidean sense, i.e., the length of the projected vectors as well as the angles between the projected vectors are preserved. This is, due to the invariance of inner products (Trefethen and Bau 1997). Substituting Equation 5 into 2 yields then the following low-rank approximation

$$\underset{m \times n}{\mathbf{A}} \approx \underset{m \times k}{\mathbf{Q}} \; \underset{k \times n}{\mathbf{B}}.$$

This decomposition is referred to as the QB decomposition. Subsequently, the smaller matrix $\mathbf{B}$ can be used to compute a matrix decomposition using a traditional algorithm.

2.2. Improved randomized algorithm

The basis matrix $\mathbf{Q}$ often fails to provide a good approximation for the column space of the input matrix. This is because most real-world data matrices do not feature a precise rank r, and instead exhibit a gradually decaying singular value spectrum. The performance can be considerably improved using the concept of oversampling and the power iteration scheme.

Oversampling

Most data matrices do not feature an exact rank, which means that the singular values $\{\sigma_i\}_{i=k+1}^{n}$ of the input matrix $\mathbf{A}$ are non-zero. As a consequence, the sketch $\mathbf{Y}$ does not exactly span the column space of the input matrix. Oversampling can be used to overcome this issue by using $l := k + p$ random projections to form the sketch, instead of just k. Here, p denotes the number of additional projections, and a small number $p = \{5, 10\}$ is often sufficient to obtain a good basis that is comparable to the best possible basis (Martinsson 2016).

The intuition behind the oversampling scheme is the following. The sketch $\mathbf{Y}$ is a random variable, as it depends on the drawing of a random test matrix $\mathbf{\Omega}$. Increasing the number of additional random projections allows one to decrease the variation in the singular value spectrum of the random test matrix, which subsequently improves the quality of the sketch.

Power iteration scheme

The second method for improving the quality of the basis $\mathbf{Q}$ involves the concept of power sampling iterations (Rokhlin *et al.* 2009; Halko *et al.* 2011b; Gu 2015). Instead of obtaining the sketch $\mathbf{Y}$ directly, the data matrix $\mathbf{A}$ is first preprocessed as

$$\mathbf{A}^{(q)} := (\mathbf{A}\mathbf{A}^\top)^q \mathbf{A}, \tag{6}$$

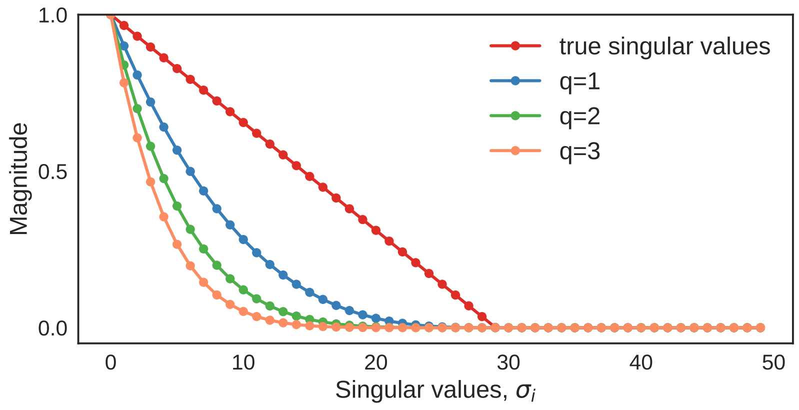


Figure 5: Singular value spectrum of a low-rank ($r = 30$) matrix before and after preprocessing. The computation of power iterations enforce a more rapid decay of singular values.

where q is an integer specifying the number of power iterations. This process enforces a more rapid decay of the singular values. Thus, we enable the algorithm to sample the relevant information related to the dominant singular values, while unwanted information is suppressed. Let $\mathbf{A} = \mathbf{U\Sigma V}^\top$ be the singular value decomposition. It is simple to show that $\mathbf{A}^{(q)} := (\mathbf{AA}^\top)^q\mathbf{A} = \mathbf{U\Sigma}^{2q+1}\mathbf{V}^\top$. Here, $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices whose columns are the left and right singular vectors of $\mathbf{A}$, and $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values in descending order. Hence, for $q > 0$, the modified matrix $\mathbf{A}^{(q)}$ has a relatively fast decay of singular values even when the decay in $\mathbf{A}$ is modest. This is illustrated in Figure 5, showing the singular values of a 50×50 low-rank matrix before (red) and after computing $q = \{1, 2, 3\}$ power iterations. Thus, substituting Equation 6 into 4 yields an improved sketch

$$\mathbf{Y} := \mathbf{A}^{(q)}\mathbf{\Omega}.$$

When the singular values of the data matrix decay slowly, as few as $q = \{1, 2, 3\}$ power iterations can considerably improve the accuracy of the approximation. The drawback of the power scheme is that q additional passes over the input matrix are required.

Algorithm 1 shows a direct implementation of the power iteration scheme. Due to potential round-off errors, however, this algorithm is not recommended in practice.

The numerical stability can be improved by orthogonalizing the sketch between each iteration. This scheme is shown in Algorithm 2, and denoted as subspace iteration (Halko *et al.* 2011b; Gu 2015). The pivoted LU decomposition can be used as an intermediate step instead of the QR decomposition as proposed by Rokhlin *et al.* (2009). Algorithm 3 is computationally more efficient, while slightly less accurate.

2.3. Random test matrices

The probabilistic framework above essentially depends on the random test matrix $\mathbf{\Omega}$ used for constructing the sketch $\mathbf{Y}$. Specifically, we seek a matrix with independent identically distributed (i.i.d.) entries from some distribution, which ensures that its columns are linearly independent with high probability. Some popular choices for constructing the random test matrix are:

```
Input: Input matrix $\mathbf{A}$, the sketch $\mathbf{Y}$, and parameter $q$.
function power_iterations($\mathbf{A}, \mathbf{Y}, q$)
(1) for $j = 1, \dots, q$          perform q power iterations
(2)     $\mathbf{Y} = \mathbf{A}^\top \mathbf{Y}$
(3)     $\mathbf{Y} = \mathbf{A}\mathbf{Y}$
Return: $\mathbf{Y} \in \mathbb{R}^{m \times k}$
```

Algorithm 1: Direct implementation of the power scheme.

```
Input: Input matrix $\mathbf{A}$, the sketch $\mathbf{Y}$, and $q$.
function sub_iterations($\mathbf{A}, \mathbf{Y}, q$)
(1) for $j = 1, \dots, q$                      perform q iterations
(2)     $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y})$              economic QR
(3)     $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{A}^\top \mathbf{Q})$        economic QR
(4)     $\mathbf{Y} = \mathbf{A}\mathbf{Q}$
Return: $\mathbf{Y} \in \mathbb{R}^{m \times k}$
```

Algorithm 2: Subspace iterations.

```
Input: Input matrix $\mathbf{A}$, the sketch $\mathbf{Y}$, and $q$.
function norm_iterations($\mathbf{A}, \mathbf{Y}, q$)
(1) for $j = 1, \dots, q$                      perform q iterations
(2)     $[\mathbf{L}, \sim] = \texttt{lu}(\mathbf{Y})$              pivoted LU
(3)     $[\mathbf{L}, \sim] = \texttt{lu}(\mathbf{A}^\top \mathbf{L})$        pivoted LU
(4)     $\mathbf{Y} = \mathbf{A}\mathbf{L}$
Return: $\mathbf{Y} \in \mathbb{R}^{m \times k}$
```

Algorithm 3: Normalized power iterations.

- **Gaussian.** The default choice to construct a random test matrix is to draw entries from the standard normal distribution, $\mathcal{N}(0, 1)$. The normal distribution is known to have excellent performance for sketching in practice. Further, the theoretical properties of the normal distribution enable the derivation of accurate error bounds (Halko *et al.* 2011b).

- **Uniform.** A simple alternative is to draw entries from the uniform distribution, $\mathcal{U}(-1, 1)$. While the behavior is similar in practice, the generation of uniform random samples is computationally more efficient.

- **Rademacher.** Yet another approach to construct the random test matrix is to draw independent Rademacher entries. The Rademacher distribution is a discrete probability distribution, where the random variates take the values $+1$ and -1 with equal probability. Rademacher entries are simple to generate, and they are cheaper to store than Gaussian and uniform random test matrices (Tropp, Yurtsever, Udell, and Cevher 2016).

Currently, the **rsvd** package only supports standard dense random test matrices; however, dense matrix operations can become very expensive for large-scale applications. This is because it takes $O(mnk)$ time to apply an $n \times k$ dense random test matrix to any $m \times n$ dense input matrix. Structured random test matrices provide a computationally more efficient alternative (Woolfe, Liberty, Rokhlin, and Tygert 2008), reducing the costs to $O(mn \log(k))$. Very sparse random test matrices are even simpler to construct (Li, Hastie, and Church 2006), yet slightly less accurate. These can be applied to any $m \times n$ dense matrix in $O(m \,\mathrm{nnz}(\mathbf{\Omega}))$ using sparse matrix multiplication routines, where nnz() denotes the non-zero entries in the sparse random test matrix.

3. Randomized singular value decompositions

The SVD provides a numerically stable matrix decomposition that can be used to obtain low-rank approximations, to compute the pseudo-inverses of non-square matrices, and to find the least-squares and minimum norm solutions of a linear model. Further, the SVD is the workhorse algorithm behind many machine learning concepts, for instance, matrix completion, sparse coding, dictionary learning, PCA and robust PCA. For a comprehensive technical overview of the SVD we refer to Golub and Van Loan (1996), and Trefethen and Bau (1997).

3.1. Brief historical overview

While the origins of the SVD can be traced back to the late 19th century, the field of randomized matrix algorithms is relatively young. Figure 6 shows a short time-line of some major developments of the singular value decomposition. Stewart (1993) gives an excellent historical review of the five mathematicians who developed the fundamentals of the SVD, namely Eugenio Beltrami (1835–1899), Camille Jordan (1838–1921), James Joseph Sylvester (1814–1897), Erhard Schmidt (1876–1959) and Hermann Weyl (1885–1955). The development and fundamentals of modern high-performance algorithms to compute the SVD are related to the seminal work of Golub and Kahan (1965) and Golub and Reinsch (1970), forming the basis for the **EISPACK**, and **LAPACK** SVD routines.

Modern partial singular value decomposition algorithms are largely based on Krylov methods, such as the Lanczos algorithm (Calvetti, Reichel, and Sorensen 1994; Larsen 1998; Lehoucq, Sorensen, and Yang 1998; Baglama and Reichel 2005). These methods are accurate and are particularly powerful for approximating structured, and sparse matrices.

Randomized matrix algorithms for computing low-rank matrix approximations have gained prominence over the past two decades. Frieze, Kannan, and Vempala (2004) introduced the "Monte Carlo" SVD, a rigorous approach to efficiently compute the approximate low-rank SVD based on non-uniform row and column sampling. Sarlos (2006), Liberty, Woolfe, Martinsson, Rokhlin, and Tygert (2007) and Martinsson *et al.* (2011) introduced a more robust approach based on random projections. Specifically, the properties of random vectors are exploited to efficiently build a subspace that captures the column space of a matrix. Woolfe *et al.* (2008) further improved the computational performance by leveraging the properties of highly structured matrices, which enable fast matrix multiplications. Eventually, the seminal work by Halko *et al.* (2011b) unified and expanded previous work on the randomized singular value decomposition and introduced state-of-the-art prototype algorithms to compute the near-optimal low-rank singular value decomposition.

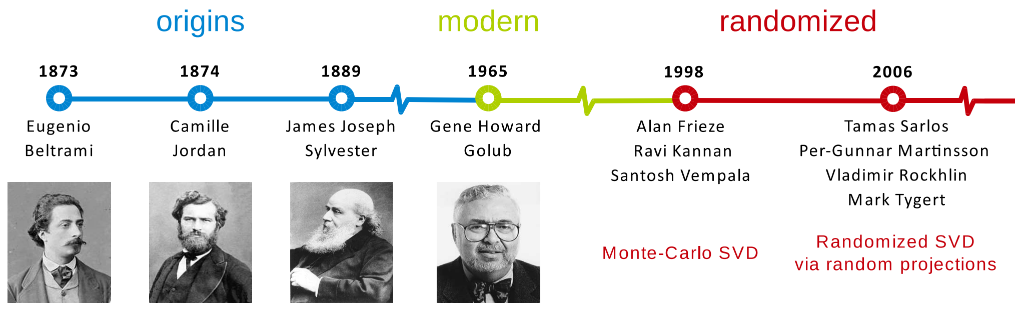


Figure 6: A timeline of major singular value decomposition developments.

3.2. Conceptual overview

Given a real matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$, the singular value decomposition takes the form

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top.$$

The matrices $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_m] \in \mathbb{R}^{m\times m}$ and $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_n] \in \mathbb{R}^{n\times n}$ are orthonormal so that $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$. The left singular vectors in $\mathbf{U}$ provide a basis for the range (column space), and the right singular vectors in $\mathbf{V}$ provide a basis for the domain (row space) of the matrix $\mathbf{A}$. The rectangular diagonal matrix $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ contains the corresponding non-negative singular values $\sigma_1 \geq \ldots \geq \sigma_n \geq 0$, describing the spectrum of the data.

The so called "economy" or "thin" SVD computes only the left singular vectors and singular values corresponding to the number (i.e., n) of right singular vectors

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V} = [\mathbf{u}_1, \ldots, \mathbf{u}_n]\mathrm{diag}(\sigma_1, \ldots, \sigma_n)[\mathbf{v}_1, \ldots, \mathbf{v}_n]^\top.$$

If the number of right singular vectors is small (i.e., $n \ll m$), this is a more compact factorization than the full SVD. The "economy" SVD is the default form of the base `svd()` function in R.

Low-rank matrices feature a rank that is smaller than the dimension of the ambient measurement space, i.e., r is smaller than the number of columns and rows. Hence, the singular values $\{\sigma_i : i \geq r+1\}$ are zero, and the corresponding singular vectors span the left and right null spaces. The concept of the "economy" SVD of a low-rank matrix is illustrated in Figure 7.

In practical applications matrices are often contaminated by errors, and the effective rank of a matrix can be smaller than its exact rank r. In this case, the matrix can be well approximated by including only those singular vectors which correspond to singular values of a significant magnitude. Hence, it is often desirable to compute a reduced version of the SVD

$$\mathbf{A}_k := \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k = [\mathbf{u}_1, \ldots, \mathbf{u}_k]\mathrm{diag}(\sigma_1, \ldots, \sigma_k)[\mathbf{v}_1, \ldots, \mathbf{v}_k]^\top,$$

where k denotes the desired target rank of the approximation. In other words, this reduced form of the SVD allows one to express $\mathbf{A}$ approximately by the sum of k rank-one matrices

$$\mathbf{A}_k \approx \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top.$$

Choosing an optimal target rank k is highly dependent on the task. One can either be interested in a highly accurate reconstruction of the original data, or in a very low dimensional

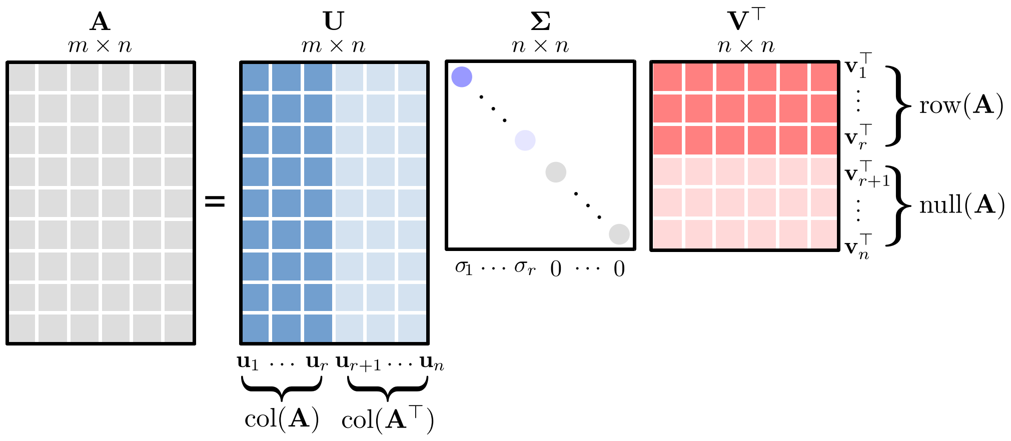


Figure 7: Schematic of the "economy" SVD for a rank-r matrix, where $m \geq n$.

representation of dominant features in the data. In the former case k should be chosen close to the effective rank, while in the latter case k might be chosen to be much smaller.

Truncating small singular values in the deterministic SVD gives an optimal approximation of the corresponding target rank k. Specifically, the Eckart-Young theorem (Eckart and Young 1936) states that the low-rank SVD provides the optimal rank-k reconstruction of a matrix in the least-square sense

$$\mathbf{A}_k := \underset{\text{rank}(\mathbf{A}'_k)=k}{\text{argmin}} \|\mathbf{A} - \mathbf{A}'_k\|.$$

The reconstruction error in both the spectral and Frobenius norms is given by

$$\|\mathbf{A} - \mathbf{A}_k\|_2 = \sigma_{k+1}(\mathbf{A}) \quad \text{and} \quad \|\mathbf{A} - \mathbf{A}_k\|_F = \sqrt{\sum_{j=k+1}^{\min(m,n)} \sigma_j^2(\mathbf{A})}.$$

For massive datasets, however, the truncated SVD is costly to compute. The cost to compute the full SVD of an $m \times n$ matrix is of the order $O(mn^2)$, from which the first k components can then be extracted to form $\mathbf{A}_k$.

3.3. Randomized algorithm

Randomized algorithms have been recently popularized, in large part due to their "surprising" reliability and computational efficiency (Gu 2015). These techniques can be used to obtain an approximate rank-k singular value decomposition at a cost of $O(mnk)$. When the dimensions of $\mathbf{A}$ are large, this is substantially more efficient than truncating the full SVD.

We present details of the randomized low-rank SVD algorithm, which comes with favorable error bounds relative to the optimal truncated SVD, as presented in the seminal paper by Halko *et al.* (2011b), and further analyzed and implemented in Voronin and Martinsson (2015).

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a low-rank matrix, and without loss of generality $m \geq n$. In the following, we seek the near-optimal low-rank approximation of the form

$$\mathbf{A} \approx \mathbf{U}_k \boldsymbol{\Sigma}_k \mathbf{V}_k^\top,$$

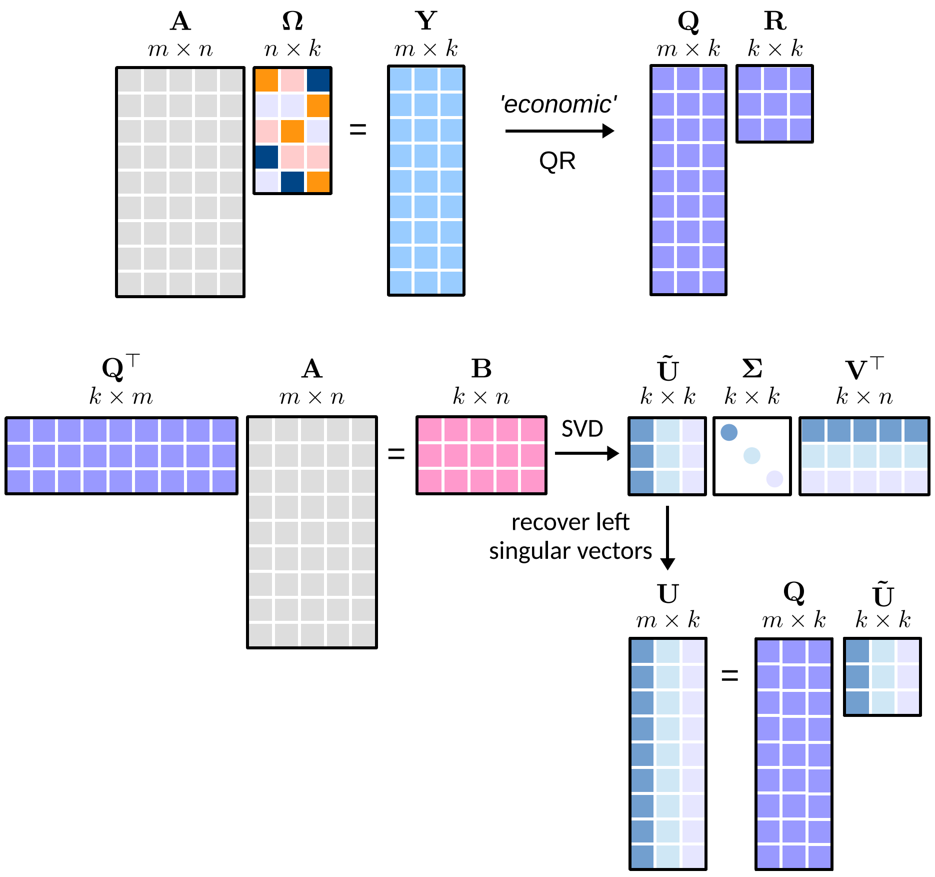

$\mathbf{A}$ $m \times n$
$\mathbf{\Omega}$ $n \times k$
=
$\mathbf{Y}$ $m \times k$
'economic' QR
$\mathbf{Q}$ $m \times k$
$\mathbf{R}$ $k \times k$
$\mathbf{Q}^\top$ $k \times m$
$\mathbf{A}$ $m \times n$
=
$\mathbf{B}$ $k \times n$
SVD
$\tilde{\mathbf{U}}$ $k \times k$
$\mathbf{\Sigma}$ $k \times k$
$\mathbf{V}^\top$ $k \times n$
recover left singular vectors
$\mathbf{U}$ $m \times k$
=
$\mathbf{Q}$ $m \times k$
$\tilde{\mathbf{U}}$ $k \times k$

Figure 8: Conceptual architecture of the randomized singular value decomposition (rSVD). First, a natural basis $\mathbf{Q}$ is computed in order to derive the smaller matrix $\mathbf{B}$. Then, the SVD is efficiently computed using this smaller matrix. Finally, the left singular vectors $\mathbf{U}$ may be reconstructed from the approximate singular vectors $\tilde{\mathbf{U}}$ by the expression in Equation 7.

where k denotes the target rank. Instead of computing the singular value decomposition directly, we embed the SVD into the probabilistic framework presented in Section 2. The principal concept is sketched in Figure 8.

Specifically, we first compute the near-optimal basis $\mathbf{Q} \in \mathbb{R}^{m\times l}$ using the randomized scheme as outlined in detail above. Note that we allow for both oversampling ($l = k + p$), and additional power iterations q, in order to obtain the near-optimal basis matrix. The matrix $\mathbf{B} \in \mathbb{R}^{l\times n}$ is relatively small if $l \ll n$, and it is obtained by projecting the input matrix to low-dimensional space, i.e., $\mathbf{B} := \mathbf{Q}^\top\mathbf{A}$. The full SVD of $\mathbf{B}$ is then computed using a deterministic algorithm

$$\mathbf{B} = \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^\top.$$

Thus, we efficiently obtain the first l right singular vectors $\mathbf{V} \in \mathbb{R}^{n\times l}$ as well as the corresponding singular values $\mathbf{\Sigma} \in \mathbb{R}^{l\times l}$. It remains to recover the left singular vectors $\mathbf{U} \in \mathbb{R}^{m\times l}$ from the approximate left singular vectors $\tilde{\mathbf{U}} \in \mathbb{R}^{l\times l}$ by pre-multiplying by $\mathbf{Q}$

$$\mathbf{U} \approx \mathbf{Q}\tilde{\mathbf{U}}. \tag{7}$$

Input: Input matrix $\mathbf{A}$ with dimensions $m \times n$, and target rank $k < \min\{m, n\}$.
Optional: Parameters p and q to control oversampling, and the power scheme.

```
function rqb(A, k, p, q)
(1)  $l = k + p$                                  slight oversampling
(2)  $\mathbf{\Omega} = \texttt{rnorm}(n, l)$                    generate random test matrix
(3)  $\mathbf{Y} = \mathbf{A}\mathbf{\Omega}$                                compute sketch
(4)  $\mathbf{Y} = \texttt{sub\_iterations}(\mathbf{A}, \mathbf{Y}, q)$    optional: compute power scheme via Algorithm 2
(9)  $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y})$                      form orthonormal basis
(10) $\mathbf{B} = \mathbf{Q}^\top \mathbf{A}$                              project to low-dimensional space
Return: $\mathbf{Q} \in \mathbb{R}^{m \times l}$, $\mathbf{B} \in \mathbb{R}^{l \times n}$
```

Algorithm 4: A randomized QB decomposition algorithm.

The justification for the randomized SVD can be sketched as follows

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top \mathbf{A} = \mathbf{Q}\mathbf{B} = \mathbf{Q}\tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top.$$

Algorithm 5 presents an implementation using the randomized QB decomposition in Algorithm 4. The approximation quality can be controlled via oversampling and additional subspace iterations. Note that if an oversampling parameter $p > 0$ has been specified, the desired rank-k approximation is simply obtained by truncating the left and right singular vectors and the singular values.

The randomized singular value decomposition has several practical advantages:

- **Lower communication costs.** The randomized algorithm presented here requires few (at least two) passes over the input matrix. By passes we refer to the number of sequential reads of the entire input matrix. This aspect is crucial in the area of "big data" where communication costs play a significant role. For instance, the time to transfer the data from the hard-drive into fast memory can be substantially more expensive than the theoretical costs of the algorithm would suggest. Recently, Tropp *et al.* (2016) have introduced an interesting set of new single pass algorithms, reducing the communication costs even further.

- **Highly parallelizable.** Randomized algorithms are highly scalable by design. This is because the computationally expensive steps involve matrix-matrix operations, which are very efficient on parallel architectures. Hence, modern computational architectures such as multi-threading and distributed computing, can be fully exploited.

- **General applicability.** Randomized matrix algorithms work for matrices with arbitrary rates of singular value decay. The approximation for matrices with rapid singular value decay approaches that of the optimal truncated SVD, with high probability (Martinsson 2016; Gu 2015).

```
Input: Input matrix $\mathbf{A}$ with dimensions $m \times n$, and target rank $k < \min\{m, n\}$.
Optional: Parameters $p$ and $q$ to control oversampling, and the power scheme.

function rsvd($\mathbf{A}, k, p, q$)
(1)  $[\mathbf{Q}, \mathbf{B}] = \mathtt{rqb}(\mathbf{A}, k, p, q)$    randomized QB decomposition via Algorithm 4
(2)  $[\tilde{\mathbf{U}}, \mathbf{\Sigma}, \mathbf{V}] = \mathtt{svd}(\mathbf{B})$      compute economic SVD
(3)  $\mathbf{U} = \mathbf{Q}\tilde{\mathbf{U}}$                   recover left singular vectors

Return: $\mathbf{U}(:, 1:k) \in \mathbb{R}^{m\times k}$, $\mathbf{\Sigma}(1:k, 1:k) \in \mathbb{R}^{k\times k}$ and $\mathbf{V}(:, 1:k) \in \mathbb{R}^{n\times k}$
```

Remark 1. In general we achieve a good computational performance, if the target rank is much smaller than the ambient dimensions of the input matrix, e.g., $k < \min\{m, n\}/4$.

Remark 2. As default values for the oversampling and the power iteration scheme we recommend the parameters $p = 10$ and $q = 2$, respectively.

Algorithm 5: A randomized SVD algorithm.

3.4. Theoretical performance

Let us consider the low-rank matrix approximation $\mathbf{QB}$, where $\mathbf{B} := \mathbf{Q}^\top \mathbf{A}$. From the Eckart-Young theorem (Eckart and Young 1936) it follows that the smallest possible error achievable with the best possible basis matrix $\mathbf{Q}$ is

$$\|\mathbf{A} - \mathbf{QB}\|_2 = \sigma_{k+1}(\mathbf{A}),$$

where $\sigma_{k+1}(\mathbf{A})$ denotes the $k + 1$ largest singular value of the matrix $\mathbf{A}$.

In Algorithm 5 we compute the full SVD of $\mathbf{B}$, so it follows that $\|\mathbf{A} - \mathbf{A}_k\|_2 = \|\mathbf{A} - \mathbf{QB}\|_2$, where $\mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$. Following Martinsson (2016), the randomized algorithm for computing the low-rank matrix approximation has the following expected error:

$$\mathsf{E}\|\mathbf{A} - \mathbf{A}_k\|_2 \le \left[1 + \sqrt{\frac{k}{p-1}} + \frac{e\sqrt{k+p}}{p} \cdot \sqrt{\min\{m,n\} - k}\right]^{\frac{1}{2q+1}} \sigma_{k+1}(\mathbf{A}). \tag{8}$$

Here, the operator E denotes the expectation with respect to a Gaussian test matrix $\mathbf{\Omega}$, and Euler's number is denoted as e. Further, it is assumed that the oversampling parameter p is greater or equal to two.

From this error bound it follows that both the oversampling (parameter p) and the power iteration scheme (parameter q) can be used to control the approximation error. With increasing p the second and third term on the right hand side tend towards zero, i.e., the bound approaches the theoretically optimal value of $\sigma_{k+1}(\mathbf{A})$. The parameter q accelerates the rate of decay of the singular values of the sampled matrix, while maintaining the same eigenvectors. This yields better performance for matrices with otherwise modest decay. Equation 8 is a simplified version of one of the key theorems presented by Halko *et al.* (2011b), who provide a detailed error analysis of the outlined probabilistic framework. Further, Witten and Candes (2015) provide sharp error bounds and interesting theoretical insights.

3.5. Existing functionality for SVD in R

The `svd()` function is the default option to compute the SVD in R. This function provides an interface to the underlying **LAPACK** SVD routines (Anderson, Bai, Bischof, Blackford, Dongarra, Du Croz, Greenbaum, Hammarling, McKenney, and Sorensen 1999). These routines are known to be numerical stable and highly accurate, i.e., full double precision.

In many applications the full SVD is not necessary; only the truncated factorization is required. The truncated SVD for an $m \times n$ matrix can be obtained by first computing the full SVD, and then extracting the k dominant components to form $\mathbf{A}_k$. However, the computational time required to approximate large-scale data is tremendous using this approach.

Partial algorithms, largely based on Krylov subspace methods, are an efficient class of approximation methods to compute the dominant singular vectors and singular values. These algorithms are particularly powerful for approximating structured or sparse matrices. This is because Krylov subspace methods only require certain operations defined on the input matrix $\mathbf{A}$ such as matrix-vector multiplication. These basic operations can be computed very efficiently, if the input matrix features some structure like sparsity. The Lanczos algorithm and its variants are the most popular choice to compute the approximate SVD. Specifically, they first find the dominant k eigenvalues and eigenvectors of the symmetric matrix $\mathbf{A}^\top\mathbf{A}$ as

$$\mathbf{A}^\top\mathbf{A}\mathbf{V}_k = \mathbf{V}\boldsymbol{\Sigma}_k^2,$$

where $\mathbf{V}_k$ are the k dominant right singular vectors, and $\boldsymbol{\Sigma}_k^2$ are the corresponding squared singular values. Depending on the dimensions of the input matrix, this operation can also be performed on $\mathbf{A}\mathbf{A}^\top$. See, for instance, Demmel (1997) and Martinsson (2016) for details on how the Lanczos algorithm builds the Krylov subspace, and subsequently approximates the eigenvalues and eigenvectors.

The relationship between the singular value decomposition and eigendecomposition can then be used to approximate the left singular vectors as $\mathbf{U}_k = \mathbf{A}\mathbf{V}_k\boldsymbol{\Sigma}^{-1}$ (see also Section 4.2). However, computing the eigenvalues of the inner product $\mathbf{A}^\top\mathbf{A}$ is not generally a good idea, because this process squares the condition number of $\mathbf{A}$. Further, the computational performance of Krylov methods depends on factors such as the initial guess for the starting vector, and additional steps used to stabilize the algorithm (Gu 2015). While partial SVD algorithms have the same theoretical costs as randomized methods, i.e., they require $O(mnk)$ floating point operations, they have higher communication costs. This is because the matrix-vector operations do not permit data reuse between iterations.

The most competitive partial SVD routines in R are provided by the **svd** (Korobeynikov and Larsen 2016), **RSpectra** (Qiu, Mei, Guennebaud, and Niesen 2016), and the **irlba** (Baglama and Reichel 2005) packages. The **svd** package provides a wrapper for the **PROPACK** SVD algorithm (Larsen 1998). The **RSpectra** package is inspired by the software package **ARPACK** (Lehoucq *et al.* 1998) and provides fast partial SVD and eigendecompositon algorithms. The **irlba** package implements implicitly restarted Lanczos bidiagonalization methods for computing the dominant singular values and vectors (Baglama and Reichel 2005). The advantage of this algorithm is that it avoids the implicit computation of the inner product $\mathbf{A}^\top\mathbf{A}$ or outer product $\mathbf{A}\mathbf{A}^\top$. Thus, this algorithm is more numerically stable if $\mathbf{A}$ is ill-conditioned.

3.6. The `rsvd()` function

The **rsvd** package provides an efficient routine to compute the low-rank SVD using Algorithm 5. The interface of the `rsvd()` function is similar to the base `svd()` function:

```
rsvd(A, k, nu = NULL, nv = NULL, p = 10, q = 2, sdist = "normal")
```

The first mandatory argument `A` passes the $m \times n$ input data matrix. The second mandatory argument `k` defines the target rank, which is assumed to be chosen smaller than the ambient dimensions of the input matrix. The `rsvd()` function achieves significant speedups for target ranks chosen to be $k < \min\{m, n\}/4$. Similar to the `svd()` function, the arguments `nu` and `nv` can be used to specify the number of left and right singular vectors to be returned.

The accuracy of the approximation can be controlled via the two tuning parameters `p` and `q`. The former parameter is used to oversample the basis, and is set by default to `p = 10`. This setting guarantees a good basis with high probability in general. The parameter `q` can be used to compute additional power iterations (subspace iterations). By default this parameter is set to `q = 2`, which yields a good performance in our numerical experiments, i.e., the default values show an optimal trade-off between speed and accuracy in standard situations. If the singular value spectrum of the input matrix decays slowly, more power iterations are desirable.

Further, the `rsvd()` routine allows one to choose between a standard normal, uniform and Rademacher random test matrices. The different options can be selected via the argument `sdist = c("normal", "unif", "rademacher")`.

The resulting model object is itself a list. It contains the following components:

- `d`: k-dimensional vector containing the singular values.
- `u`: $m \times k$ matrix containing the left singular vectors.
- `v`: $n \times k$ matrix containing the right singular vectors. Note that `v` is not returned in its transposed form, as it is often returned in other programing languages.

More details are provided in the corresponding documentation, see `?rsvd`.

3.7. SVD example: Image compression

The singular value decomposition can be used to obtain a low-rank approximation of high-dimensional data. Image compression is a simple, yet illustrative example. The underlying structure of natural images can often be represented by a very sparse model. This means that images can be faithfully recovered from a relatively small set of basis functions. For demonstration, we use the following 1600×1200 grayscale image:

```
R> data("tiger", package = "rsvd")
R> image(tiger, col = gray(0:255 / 255))
```

A grayscale image may be thought of as a real-valued matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, where m and n are the number of pixels in the vertical and horizontal directions, respectively. To compress the image we need to first decompose the matrix $\mathbf{A}$. The singular vectors and values provide a hierarchical representation of the image in terms of a new coordinate system defined by dominant correlations within rows and columns of the image. Thus, the number of singular vectors used for approximation poses a trade-off between the compression rate (i.e., the number of singular vectors to be stored) and the reconstruction fidelity. In the following, we use the arbitrary choice $k = 100$ as target rank. First, the R base `svd()` function is used to compute the truncated singular value decomposition:

```
R> k <- 100
R> tiger.svd <- svd(tiger, nu = k, nv = k)
```

The `svd()` function returns three objects: `u`, `v` and `d`. The first two objects are $m \times k$ and $n \times k$ arrays, namely the truncated left and right singular vectors. The vector `d` is comprised of the $\min\{m, n\}$ singular values in descending order. Now, the dominant $k = 100$ singular values are retained to approximate/reconstruct ($\mathbf{A}_k := \mathbf{U}_k\mathbf{D}_k\mathbf{V}_k^\top$) the original image:

```
R> tiger.re <- tiger.svd$u %*% diag(tiger.svd$d[1:k]) %*% t(tiger.svd$v)
R> image(tiger.re, col = gray(0:255 / 255))
```

The normalized root mean squared error (nrmse) is a common measure for the reconstruction quality of images, computed as:

```
R> nrmse <- sqrt(sum((tiger - tiger.re) ** 2 ) / sum(tiger ** 2))
```

Using only $k = 100$ singular values/vectors, a reconstruction error as low as 12.1% is achieved. This illustrates the general fact that natural images feature a very compact representation. Note, that the singular value decomposition is also a numerically reliable tool for extracting a desired signal from noisy data. The central idea is that the small singular values mainly represent the noise, while the dominant singular values represent the desired signal.

If the data matrix exhibits low-rank structure, the provided `rsvd()` function can be used as a plug-in function for the base `svd()` function, in order to compute the near-optimal low-rank singular value decomposition:

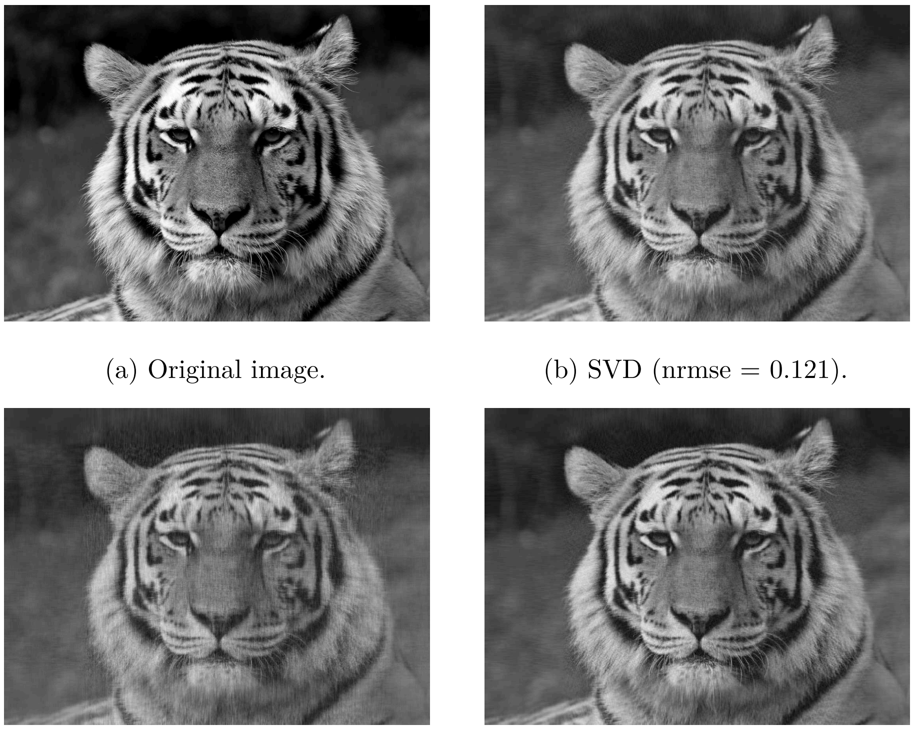
```
R> tiger.rsvd <- rsvd(tiger, k = k)
```

Similar to the base SVD function, the `rsvd()` function returns three objects: `u`, `v` and `d`. Again, `u` and `v` are $m \times k$ and $n \times k$ arrays containing the approximate left and right singular vectors and the vector `d` is comprised of the k singular values in descending order. Optionally, the approximation accuracy of the randomized SVD algorithm can be controlled by the two parameters `p` and `q`, as described in the previous section. Again, the approximated image and the reconstruction error can be computed as:

```
R> tiger.re <- tiger.rsvd$u %*% diag(tiger.rsvd$d) %*% t(tiger.rsvd$v)
R> nrmse <- sqrt(sum((tiger - tiger.re) ** 2) / sum(tiger ** 2))
```

The reconstruction error is about 0.122, i.e., close to the optimal truncated SVD. Figure 9 presents the visual results using both the deterministic and randomized SVD algorithms. By visual inspection, no significant differences can be seen between (b) and (d). However, the quality suffers by omitting subspace iterations in (c).

Table 1 shows the performance for different SVD algorithms in R. The `rsvd()` functions achieves an average speedup of about 4–7 over the `svd()` function. The `svds()` and `irlba()` functions achieve speedups of about 1.5. The computational gain of the randomized algorithm becomes more pronounced with increased matrix dimension, e.g., images with higher resolution.



(a) Original image. (b) SVD (nrmse = 0.121).

(c) rSVD using $q = 0$ (nrmse = 0.165). (d) rSVD using $q = 2$ (nrmse = 0.122).

Figure 9: Subplot (a) shows the original image, and subplots (b), (c) and (d) show the reconstructed images using the dominant $k = 100$ components. The reconstruction quality of randomized SVD with power iterations in (d) is nearly as good as of the deterministic SVD.

Package	Function	Parameters	Time (s)	Speedup	Error
base	`svd()`	`nu = nv = 100`	0.37	*	0.121
svd	`propack.svd()`	`neig = 100`	0.55	0.67	0.121
RSpectra	`svds()`	`k = 100`	0.25	1.48	0.121
irlba	`irlba()`	`nv = 100`	0.24	1.54	0.121
rsvd	`rsvd()`	`k = 100, q = 0`	0.03	12.3	0.165
rsvd	`rsvd()`	`k = 100, q = 1`	0.052	7.11	0.125
rsvd	`rsvd()`	`k = 100, q = 2`	0.075	4.9	0.122
rsvd	`rsvd()`	`k = 100, q = 3`	0.097	3.8	0.121

Table 1: Summary of algorithm runtimes (averaged over 20 runs) and errors. The randomized routines achieve substantial speedups, while attaining similar reconstruction errors with $q \geq 1$.

The trade-off between accuracy and speed of the partial SVD algorithms depends on the precision parameter `tol`, and we set the tolerance parameter for all algorithms to `tol = 1e-5`.

3.8. Computational performance

In the following we evaluate the performance of the randomized SVD routine and compare it to other SVD routines available in R. To fully exploit the power of randomized algorithms we

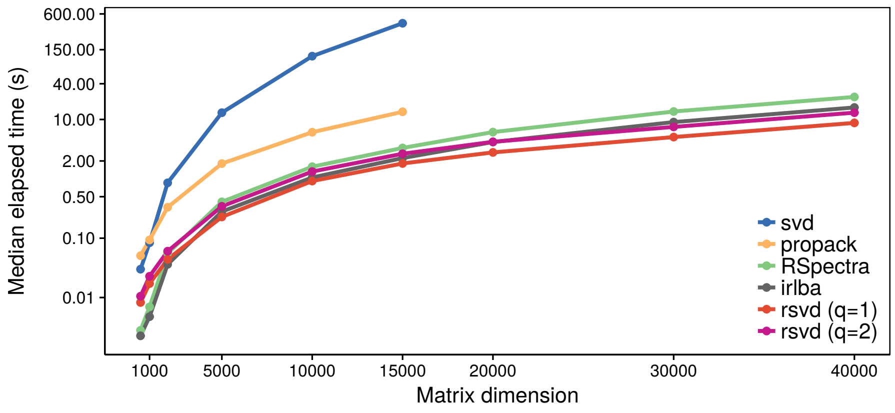


Figure 10: Runtimes for computing rank $k = 20$ approximations for varying matrix dimensions.

use the enhanced R distribution Microsoft R Open 3.4.3. This R distribution is linked with multi-threaded LAPACK libraries, which use all available cores and processors. Compared to the standard CRAN R distribution, which uses only a single thread (processor), the enhanced R distribution shows significant speedups for matrix operations. For benchmark results, see https://mran.microsoft.com/documents/rro/multithread/. However, we see also significant speedups when using the standard CRAN R distribution.

A machine with Intel Core i7-7700K CPU Quad-Core 4.20GHz, 64GB fast memory, and operating-system Ubuntu 17.04 is used for all computations. The **microbenchmark** package is used for accurate timing (Mersmann, Beleites, Hurling, and Friedman 2015).

To compare the computational performance of the SVD algorithms, we consider low-rank matrices with varying dimensions m and n, and intrinsic rank $r = 200$, generated as:

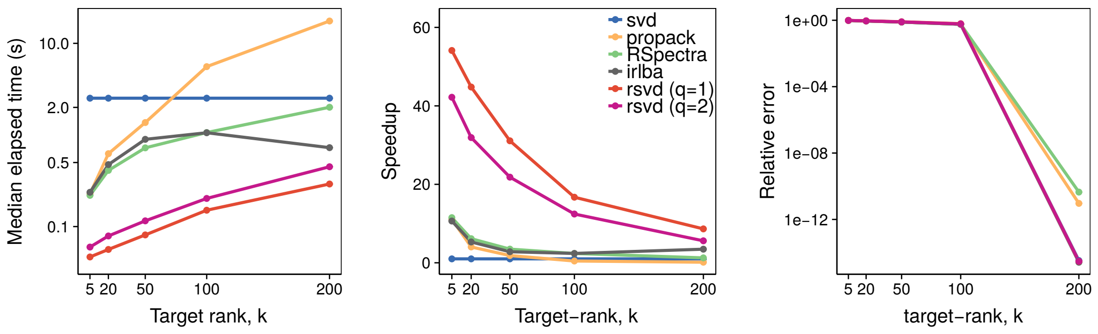
```
R> A <- matrix(rnorm(m * r), m, r) %*% matrix(rnorm(r * n), r, n)
```

Figure 10 shows the runtime for low-rank approximations (target-rank $k = 20$) and varying matrix dimensions $m \times n$, where the second dimension is chosen to be $n := 0.75 \cdot m$. While the routines of the **RSpectra** and **irlba** packages perform best for small dimensions, the computational advantage of the randomized SVD becomes pronounced with increasing dimensions. We now investigate the performance of the routines in more detail. Figures 11, 12, and 13 show the computational performance for varying matrix dimensions and target ranks. The elapsed time is computed as the median over 20 runs. The speedups show the relative performance compared to the base `svd()`, i.e., the average runtime of the `svd()` function is divided by the runtime of the other SVD algorithms. The relative reconstruction error is computed as

$$\frac{\|\mathbf{A} - \mathbf{A}_k\|_F}{\|\mathbf{A}\|_F},$$

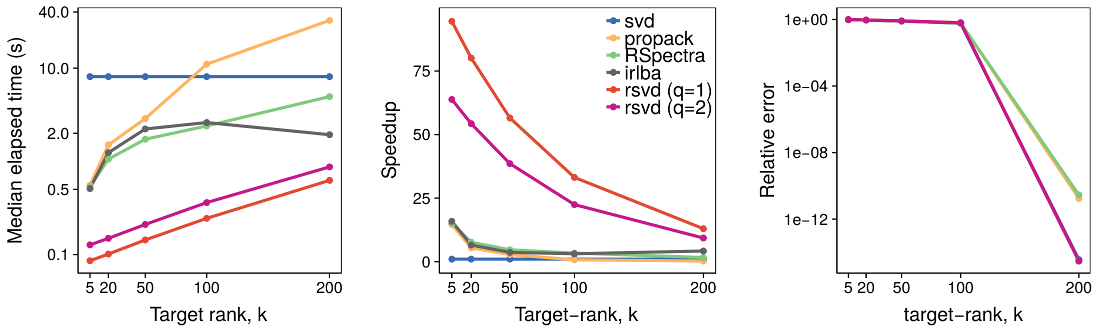
where $\mathbf{A}_k := \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$ is the rank-k matrix approximation.

The `rsvd()` function achieves substantial speedups over the other SVD routines. Here, the oversampling parameter is fixed to $p = 10$, but it can be seen that additional power iterations improve the approximation accuracy. This allows the user to control the trade-off between computational time and accuracy, depending on the application. Note that we have set the precision parameter of the **RSpectra**, **irlba** and **propack** routines to `tol = 1e-5`.



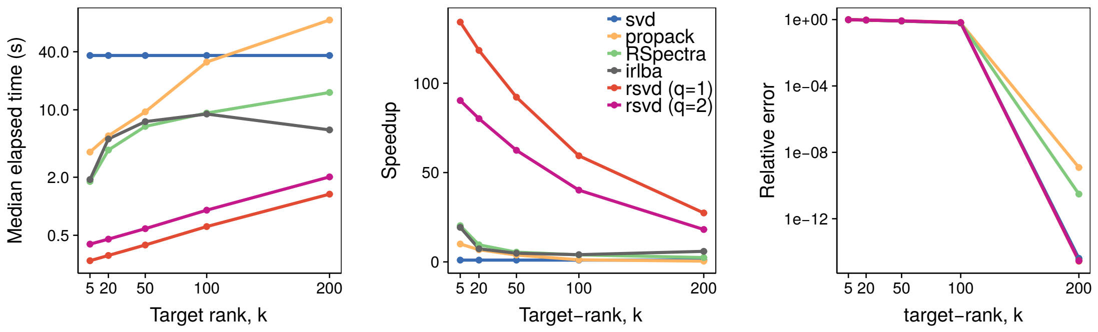
(a) Runtime. (b) Speedups. (c) Relative errors.

Figure 11: Computational performance for a dense 3000 × 2000 low-rank matrix.



(a) Runtime. (b) Speedups. (c) Relative errors.

Figure 12: Computational performance for a dense 5000 × 3000 low-rank matrix.

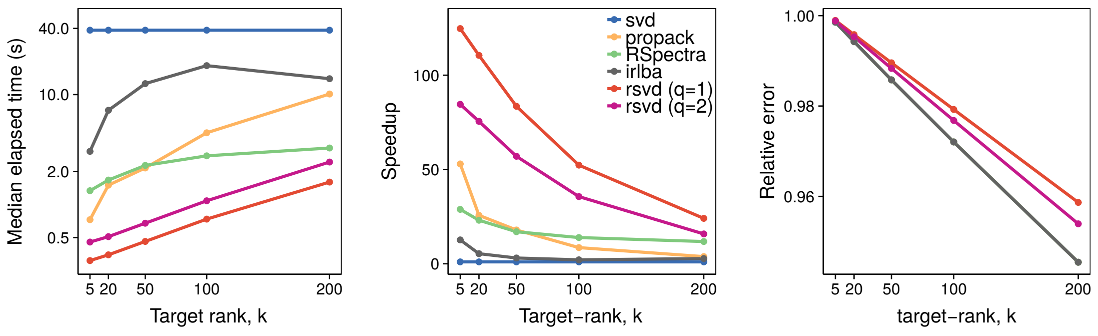


(a) Runtime. (b) Speedups. (c) Relative errors.

Figure 13: Computational performance for a dense 10000 × 5000 low-rank matrix.

Figure 14 show the computational performance for sparse matrices with about 5% non-zero elements. The **RSpectra**, and **propack** routines are specifically designed for sparse and structured matrices, and show considerably better computational performance.

Note that the random sparse matrices do not feature low-rank structure; hence, the large



(a) Runtime. (b) Speedups. (c) Relative errors.

Figure 14: Computational performance for a sparse 10000 × 5000 matrix.

relative error. Still, the randomized SVD shows a good trade-off between speedup and accuracy.

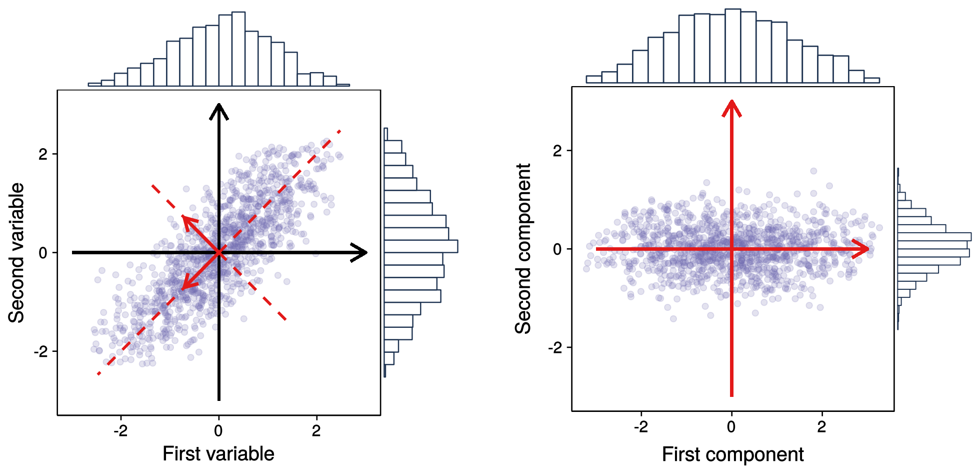
4. Randomized principal component analysis

Dimensionality reduction is a fundamental concept in modern data analysis. The idea is to exploit relationships among points in high-dimensional space in order to construct some low-dimensional summaries. This process aims to eliminate redundancies, while preserving interesting characteristics of the data (Burges 2010). Dimensionality reduction is used to improve the computational tractability, to extract interesting features, and to visualize data which are comprised of many interrelated variables. The most important linear dimension reduction technique is principal component analysis (PCA), originally formulated by Pearson (1901) and Hotelling (1933). PCA plays an important role, in particular, due to its simple geometric interpretation. Jolliffe (2002) provides a comprehensive introduction to PCA.

4.1. Conceptual overview

Principal component analysis aims to find a new set of uncorrelated variables. The so called principal components (PCs) are constructed such that the first PC explains most of the variation in the data; the second PC most of the remaining variation and so on. This property ensures that the PCs sequentially capture most of the total variation (information) present in the data. In practice, we often aim to retain only a few number of PCs which capture a "good" amount of the variation, where "good" depends on the application. The idea is depicted for two correlated variables in Figure 15. Figure 15a illustrates the two principal directions of the data, which span a new coordinate system. The first principal direction is the vector pointing in the direction which accounts for most of the variability in data. The second principal direction is orthogonal (perpendicular) to the first one and captures the remaining variation in the data. Figure 15b shows the original data using the principal directions as a new coordinate system. Compared to the original data, the histograms indicate that most of the variation is now captured by just the first principal component, while less by the second component.

To be more formal, assume a data matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ with m observations and n variables



(a) Original (standardized) data. (b) Rotated data.

Figure 15: PCA seeks to find a new set of uncorrelated variables. Plot (a) shows some two-dimensional data, and its two principal directions. Plot (b) shows the new principal components. Geometrically, the PCs are simply a rotation and reflection of the original data, so that the first component accounts for most of the variation in the data, now.

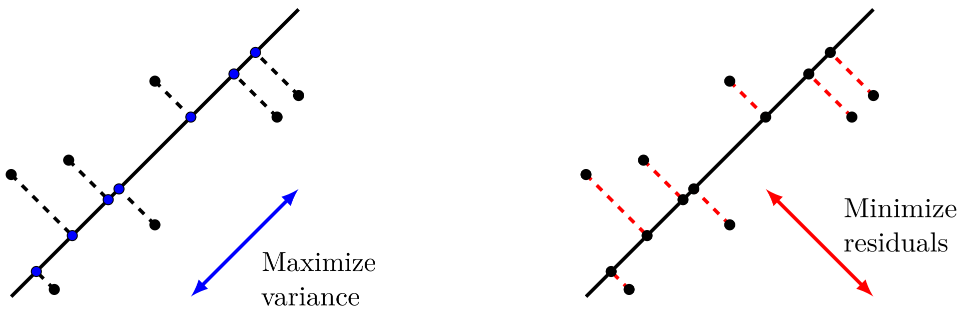


Figure 16: Principal component analysis can be formulated either as a variance maximization or as a least square minimization problem. Both views are equivalent.

(column-wise, mean-centered). Then, the principal components can be expressed as a weighted linear combination of the original variables

$$\mathbf{z}_i := \mathbf{X}\mathbf{w}_i,$$

where $\mathbf{z}_i \in \mathbb{R}^m$ denotes the ith principal component. The vector $\mathbf{w}_i \in \mathbb{R}^n$ is the ith principal direction, where the elements of $\mathbf{w}_i = [w_1, \ldots, w_n]^\top$ are the principal component coefficients.

The problem is now to find a suitable vector $\mathbf{w}_1$ such that the first principal component $\mathbf{z}_1$ captures most of the variation in the data. Mathematically, this problem can be formulated either as a least square problem or as a variance maximization problem (Cunningham and Ghahramani 2015). The two views are illustrated in Figure 16. This is, because the total variation equals the sum of the explained and unexplained variation (Jolliffe 2002), illustrated in Figure 17.

We follow the latter view, and maximize the variance of the first principal component $\mathbf{z}_1 = \mathbf{X}\mathbf{w}_1$

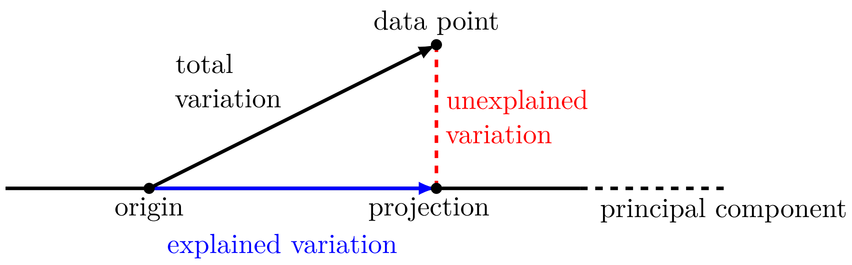


Figure 17: The Pythagorean theorem provides a geometrical explanation for the relationship between the two views: The PCs can be obtained by either maximizing the variance or by minimizing the unexplained variation (squared residuals) of the data.

subject to the normalization constraint $\|\mathbf{w}\|_2^2 = 1$

$$\mathbf{w}_1 := \underset{\|\mathbf{w}\|_2^2=1}{\operatorname{argmax}} \mathsf{VAR}(\mathbf{X}\mathbf{w}), \tag{9}$$

where VAR denotes the variance operator. We can rewrite Equation 9 as

$$\mathbf{w}_1 := \underset{\|\mathbf{w}\|_2^2=1}{\operatorname{argmax}} \frac{1}{m-1}\|\mathbf{X}\mathbf{w}\|_2^2 = \underset{\|\mathbf{w}\|_2^2=1}{\operatorname{argmax}} \mathbf{w}^\top(\frac{1}{m-1}\mathbf{X}^\top\mathbf{X})\mathbf{w}. \tag{10}$$

We note that the scaled inner product $\mathbf{X}^\top\mathbf{X}$ forms the sample covariance matrix

$$\mathbf{C} := \frac{1}{m-1}\mathbf{X}^\top\mathbf{X}.$$

$\mathbf{C}$ corresponds to the sample correlation matrix if the columns of $\mathbf{X}$ are both centered and scaled. We substitute $\mathbf{C}$ into Equation 10

$$\mathbf{w}_1 := \underset{\|\mathbf{w}\|_2^2=1}{\operatorname{argmax}} \mathbf{w}^\top\mathbf{C}\mathbf{w}.$$

Next, the method of Lagrange multipliers is used to solve the problem. First, we formulate the Lagrange function

$$\mathcal{L}(\mathbf{w}_1, \lambda_1) = \mathbf{w}_1^\top\mathbf{C}\mathbf{w}_1 - \lambda_1(\mathbf{w}_1^\top\mathbf{w}_1 - 1).$$

Then, we maximize the Lagrange function by differentiating with respect to $\mathbf{w}_1$

$$\frac{\partial\mathcal{L}(\mathbf{w}_1, \lambda_1)}{\partial\mathbf{w}_1} = \mathbf{C}\mathbf{w}_1 - \lambda_1\mathbf{w}_1,$$

which leads to the well known eigenvalue problem. Thus, the first principal direction for the mean centered matrix $\mathbf{X}$ is given by the dominant eigenvector $\mathbf{w}_1$ of the covariance matrix $\mathbf{C}$. The amount of variation explained by the first principal component is expressed by the corresponding eigenvalue λ_1. More generally, the subsequent principal component directions can be obtained by computing the eigendecompositon of the covariance or correlation matrix

$$\mathbf{C}\mathbf{W} = \mathbf{W}\mathbf{\Lambda}.$$

The columns of $\mathbf{W} \in \mathbb{R}^{n\times n}$ are the eigenvectors (principal directions) which are orthonormal, i.e., $\mathbf{W}^\top\mathbf{W} = \mathbf{W}\mathbf{W}^\top = \mathbf{I}$. The diagonal elements of $\mathbf{\Lambda} \in \mathbb{R}^{n\times n}$ are the corresponding eigenvalues. The matrix $\mathbf{W}$ can also be interpreted as a projection matrix that maps the original observations to new coordinates in eigenspace. Hence, the n principal components $\mathbf{Z} \in \mathbb{R}^{m\times n}$ can be more concisely expressed as

$$\mathbf{Z} := \mathbf{X}\mathbf{W}.$$

Since the eigenvectors have unit norm, the projection should be purely rotational without any scaling; thus, $\mathbf{W}$ is also denoted as rotation matrix.

PCA whitening

In some situations, the scaled eigenvectors

$$\mathbf{L} := \mathbf{W}\mathbf{\Lambda}^{0.5}$$

provide a more insightful interpretation of the data. $\mathbf{L} \in \mathbb{R}^{n\times n}$ is denoted as a loading matrix and provides a factorization of the covariance (correlation) matrix

$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^\top.$$

Thus, the loadings have the following two interesting properties:

- The squared column sums equal the eigenvalues.
- The squared row sums equal the variable's variance.

Further, the loading matrix $\mathbf{L}$ can be used to compute the n whitened principal components

$$\mathbf{Z}_{\text{white}} := \mathbf{X}\mathbf{L}.$$

Essentially, whitening rescales the principal components so that they have unit variance. This process is also called sphering (Kessy, Lewin, and Strimmer 2018). In other words, whitening scales the ith principal component by the corresponding eigenvalue $1/\sqrt{\lambda_i}$ as

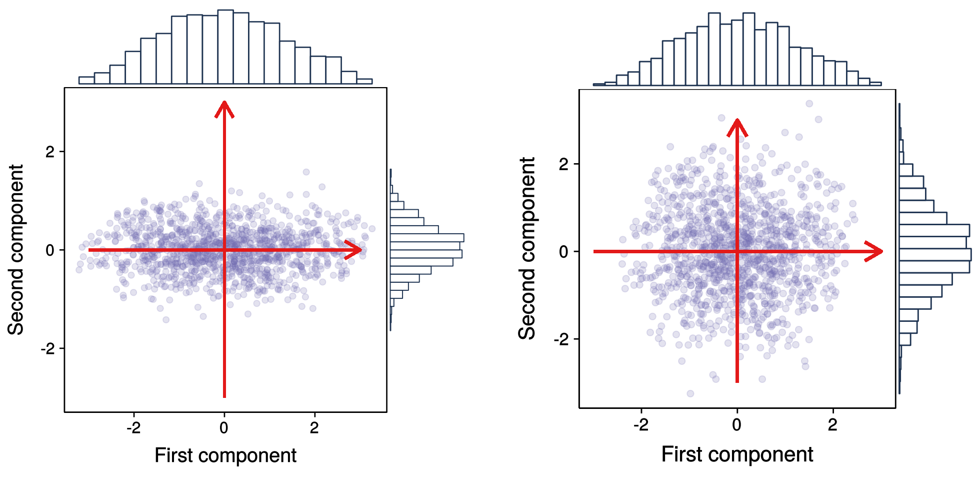
$$\mathbf{z}_{\text{white}} := \frac{\mathbf{z}_i}{\sqrt{\lambda_i}}.$$

This is best illustrated by revisiting the above example shown in Figure 15. In Figure 18 we show both the rotated and the whitened version of the data.

Dimensionality reduction

In practice, we often seek a useful low-dimensional representation to reveal the coherent structure of the data. Choosing a "good" target-rank k, i.e., the number of PCs to retain, is a subtle issue and often domain specific. Little is gained by retaining too many components. Conversely, a bad approximation is produced if the number of retained components is too small. Fortunately, the eigenvalues tell us the amount of variance captured by keeping only k components given by

$$\frac{\sum_{i=1}^{k}\lambda_i}{\sum_{i=1}^{n}\lambda_i}.$$



(a) Rotated data. (b) Rotated and whitened data.

Figure 18: Plot (a) shows the principal components and plot (b) shows the whitened principal components. The whitened components are uncorrelated and have unit variance.

Thus, PCs corresponding to eigenvalues of small magnitude account only for a small amount of information in the data.

Many different heuristics, like the scree plot and Kaiser criterion, have been proposed to identify the optimal number of components (Jolliffe 2002). A computational intensive approach to determine the optimal number of components is via cross-validation approximations (Josse and Husson 2012), while a mathematically refined approach is the optimal hard threshold method for singular values, formulated by Gavish and Donoho (2014). An interesting Bayesian approach to estimate the intrinsic dimensionality of a high-dimensional dataset was recently proposed by Bouveyron, Latouche, and Mattei (2017).

4.2. Randomized algorithm

The singular value decomposition provides a computationally efficient and numerically stable approach for computing the principal components. Specifically, the eigenvalue decomposition of the inner and outer dot product of $\mathbf{X} = \mathbf{U\Sigma V}^\top$ can be related to the SVD as

$$\mathbf{X}^\top\mathbf{X} = (\mathbf{V\Sigma U}^\top)(\mathbf{U\Sigma V}^\top) = \mathbf{V\Sigma}^2\mathbf{V}^\top, \tag{11a}$$

$$\mathbf{X}\mathbf{X}^\top = (\mathbf{U\Sigma V}^\top)(\mathbf{V\Sigma U}^\top) = \mathbf{U\Sigma}^2\mathbf{U}^\top. \tag{11b}$$

It follows that the eigenvalues are equal to the squared singular values. Thus, we recover the eigenvalues of the sample covariance matrix $\mathbf{C} := (m-1)^{-1}\mathbf{X}^\top\mathbf{X}$ as $\mathbf{\Lambda} = (m-1)^{-1}\mathbf{\Sigma}^2$.

The left singular vectors $\mathbf{U}$ correspond to the eigenvectors of the outer product $\mathbf{XX}^\top$, and the right singular vectors $\mathbf{V}$ correspond to the eigenvectors of the inner product $\mathbf{X}^\top\mathbf{X}$. This allows us to define the projection (rotation) matrix $\mathbf{W} := \mathbf{V}$.

Having established the connection between the singular value decomposition and principal component analysis, it is straight forward to show that the principal components can be computed as

$$\mathbf{Z} := \mathbf{XW} = \mathbf{U\Sigma V}^\top\mathbf{W} = \mathbf{U\Sigma}.$$

```
Input: Centered/scaled input matrix $\mathbf{X}$ with dimensions $m \times n$, and target rank $k < \min\{m, n\}$.
Optional: Parameters $p$ and $q$ to control oversampling, and the power scheme.

function rpca($\mathbf{X}, k, p, q$)
(1)   $[\mathbf{U}, \mathbf{\Sigma}, \mathbf{W}] = \texttt{rsvd}(\mathbf{X}, k, p, q)$    randomized SVD (Algorithm 5)
(2)   $\mathbf{\Lambda} = \mathbf{\Sigma}^2/(m-1)$                     recover eigenvalues
(3)   $\mathbf{Z} = \mathbf{U}\mathbf{\Sigma}$                                optional: compute $k$ principal components

Return: $\mathbf{W} \in \mathbb{R}^{n\times k}$, $\mathbf{\Lambda} \in \mathbb{R}^{k\times k}$ and $\mathbf{Z} \in \mathbb{R}^{m\times k}$
```

Algorithm 6: A randomized PCA algorithm.

The randomized singular value decomposition can then be used to efficiently approximate the dominant k principal components. This approach is denoted as randomized principal component analysis, first introduced by Rokhlin *et al.* (2009), and later by Halko *et al.* (2011a). Szlam, Kluger, and Tygert (2014) provide some additional interesting implementation details for large-scale applications. Algorithm 6 presents an implementation of the randomized PCA. The approximation accuracy can be controlled via oversampling and additional power iterations as described in Section 2.2.

4.3. Existing functionality for PCA in R

The `prcomp()` and `princomp()` functions are the default options for performing PCA in R. The `prcomp()` routine uses the singular value decomposition and the `princomp()` function uses the eigenvalue decomposition to compute the principal components (Venables and Ripley 2002). Other options in R are the PCA routines of the **ade4** (Dray and Dufour 2007) and **FactoMineR** (LÃł, Josse, and Husson 2008) packages, which provide extended plot and summary capabilities. All these routines, however, are based on computationally demanding algorithms.

In many applications, only the dominant principal components are required. In this case, partial algorithms are an efficient alternative to constructing low-rank approximations, as discussed in Section 3. For instance, the **irlba** package provides a computationally efficient routine for computing the dominant principal components using the implicitly restarted Lanczos method (Baglama, Reichel, and Lewis 2017).

Another class of methods are incremental PCA algorithms, also denoted as online PCA. These techniques are interesting if the data matrix is not entirely available to start with, i.e., the algorithms allow one to update the decomposition with each new arriving observation in time. Cardot and Degras (2015) give an overview of online PCA algorithms and the corresponding routines are provided via the **onlinePCA** package (Degras and Cardot 2016). Similarly, the **idm** package provides an incremental PCA algorithm (Degras and Cardot 2017).

4.4. The `rpca()` function

The `rpca()` function provides an efficient routine for computing the dominant principal components using Algorithm 6. This routine is in particular relevant if the information in large-scale data matrices can be approximated by the first few principal components.

The interface of the `rpca()` function is similar to the `prcomp()` function:

```
rpca(A, k, center = TRUE, scale = TRUE, retx = TRUE, p = 10, q = 2)
```

The first mandatory argument `A` passes the $m \times n$ input data matrix. Note, that the analysis can be affected if the variables have different units of measurement. In this case, scaling is required to ensure a meaningful interpretation of the components. The `rpca()` function centers and scales the input matrix by default, i.e., the analysis is based on the implicit correlation matrix. However, if all of the variables have same units of measurement, there is the option to work with either the covariance or correlation matrix. In this case, the choice largely depends on the data and the aim of the analysis. The default options can be changed via the arguments `center` and `scale`. The second mandatory argument `k` sets the target rank, and it is assumed that `k` is smaller than the ambient dimensions of the input matrix. The principal components are returned by default; otherwise the argument `retx` can be set to `FALSE` to not return the PCs. The parameters `p` and `q` are described in Section 3.6.

The resulting model object is a list and contains the following components:

- `rotation`: $n \times k$ matrix containing the eigenvectors.
- `eigvals`: k-dimensional vector containing the eigenvalues.
- `sdev`: k-dimensional vector containing the standard deviations of the principal components, i.e., the square root of the eigenvalues.
- `x`: $m \times k$ matrix containing the principal components (rotated variables).
- `center, scale`: the numeric centering and scalings used (if any).

Utility functions

The `rpca()` routine comes with methods that can be used to summarize and display the model information. These are similar to the `prcomp()` function. The `summary()` function provides information about the explained variance, standard deviations, proportion of variance as well as the cumulative proportion of the computed principal components. The `print()` function can be used to print the eigenvectors (principal directions). The `plot()` function can be used to visualize the results, using the **ggplot2** package (Wickham 2009).

4.5. PCA example: Handwritten digits

Handwritten digit recognition is a widely studied problem (Lecun, Bottou, Bengio, and Haffner 1998). In the following, we use a downsampled version of the MNIST (Modified National Institute of Standards and Technology) database of handwritten digits. The data are obtained from `http://yann.lecun.com/exdb/mnist` and can be loaded from the command line:

```
R> data("digits", package = "rsvd")
R> label <- as.factor(digits[, 1])
R> digits <- digits[, 2:785]
```

The data matrix is of dimension 12000×785. Each row corresponds to a digit between 0 and 3. The first column is comprised of the class labels, while the following 784 columns record the pixel intensities for the flattened 28×28 image patches. Figure 21a shows some of the digits. In R the first digit can be displayed as:

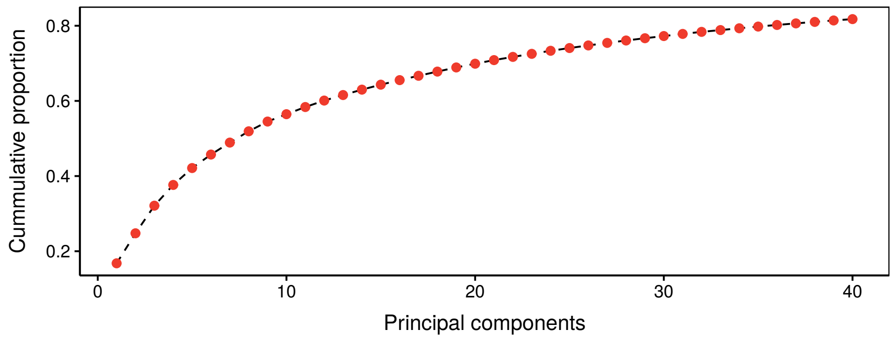


Figure 19: Cumulative proportion of the variance explained by the principal components. The first 40 PCs explain about 82% of the total variation in the data.

```
R> digit <- matrix(digits[1, ], nrow = 28, ncol = 28)
R> image(digit[, 28:1], col = gray(255:0 / 255))
```

The aim of principal component analysis is to find a low-dimension representation which captures most of the variation in the data. PCA helps to understand the sources of variability in the data as well as to understand correlations between variables. The principal components can be used for visualization, or as features to train a classifier. A common choice is to retain the dominant 40 principal components for classifying digits, using the k-nearest neighbor algorithm (Lecun *et al.* 1998). Those can be efficiently approximated using the `rpca()` function:

```
R> digits.rpca <- rpca(digits, k = 40, center = TRUE, scale = FALSE)
```

The target rank is defined via the argument `k`. By default, the data are mean centered and standardized, i.e., the correlation matrix is implicitly computed. Here, we set `scale = FALSE`, since the variables have the same units of measurement, namely pixel intensities. The analysis can be summarized using the `summary()` function. The screeplot function can be used to visualize the cumulative proportion of the variance captured by the principal components:

```
R> ggscreeplot(digits.rpca, type = "cum")
```

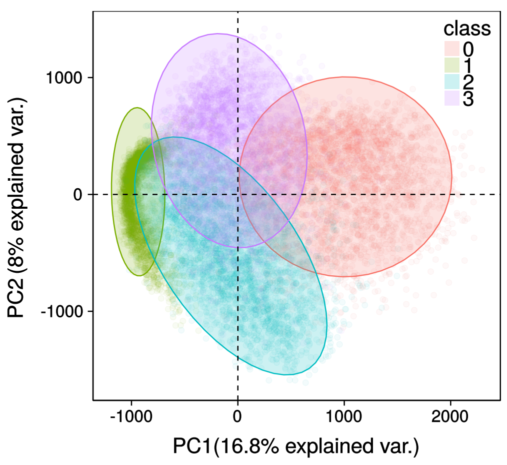
Figure 19 shows the corresponding plot. Next, the PCs can be plotted in order to visualize the data in low-dimensional space:

```
R> ggindplot(digits.rpca, groups = label, ellipse = TRUE, ind_labels = FALSE)
```

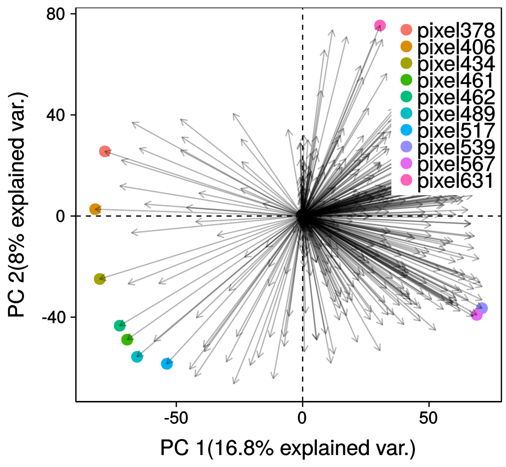
The so-called individual factor map, using the first and second principal component, is shown in Figure 20a. The plot helps to reveal some interesting patterns in the data. For instance, 0's are distinct from 1's, while 3's share commonalities with all the other classes. Further, the correlation between the original variables and the PCs can be visualized:

```
R> ggcorplot(digits.rpca, alpha = 0.3, top.n = 10)
```

The correlation plot, also denoted as variables factor map, is shown in Figure 20b. It shows the projected variables in eigenspace. This representation of the data gives some insights into the structural relationship (correlation) between the variables and the principal components.



(a) Individuals factor map.



(b) Variables factor map.

Figure 20: Plotting functionality to visualize the PCs: (a) shows the individuals factor map, overlaid with ellipses for each class; (b) shows the variables factor map.

In order to quantify the quality of the dimensionality reduction, we can compute the relative error between the low-rank approximation and the original data. Recall, that the dominant k principal component were defined as $\mathbf{Z}_k := \mathbf{X}\mathbf{W}_k$. Hence, we can approximate the input matrix as $\mathbf{X} \approx \mathbf{Z}_k\mathbf{W}_k^\top$:

```
R> digits.re <- digits.rpca$x %*% t(digits.rpca$rotation)
```

Since the procedure has centered the data, we need to add the mean pixel values back:

```
R> digits.re <- sweep(digits.re, 2, digits.rpca$center, FUN = "+")
```

The relative error can then be computed:

```
R> norm(digits - digits.re, "F") / norm(digits, "F")
```

The relative error is approximately 32.8%. Figure 21b and 21c show the samples of the reconstructed digits using both the `prcomp()` and `rpca()` function. By visual inspection, there is virtually no noticeable difference between the deterministic and the randomized approximation.

Runtimes and relative errors for different PCA functions in R are listed in Table 2. The randomized algorithm is much faster than the `prcomp()` function, while attaining near-optimal results. Both the `dudi.pca()` and `PCA()` functions are slower than the base `prcomp()` function. This is because we are using the MKL (math kernel library) accelerated R distribution Microsoft R Open 3.4.1. The timings can vary compared to using the standard R distribution.

Handwritten digit recognition

The principal component scores can be used as features to efficiently train a classifier. This is because PCA assumes that the interesting information in the data are reflected by the dominant principal components. This assumption is not always valid, i.e., in some applications

Package	**Function**	**Parameters**	**Time (s)**	**Speedup**	**Error**
base	`prcomp()`	`rank.  = 40`	0.56	*	0.327
FactoMineR	`PCA()`	`ncp = 40`	0.97	0.57	0.327
ade4	`dudi.pca()`	`nf = 40`	0.91	0.61	0.327
irlba	`prcomp_irlba()`	`n = 40`	0.47	1.2	0.327
rsvd	`rpca()`	`k = 40`	0.37	1.5	0.328

Table 2: Summary of the computational performance of different PCA functions.

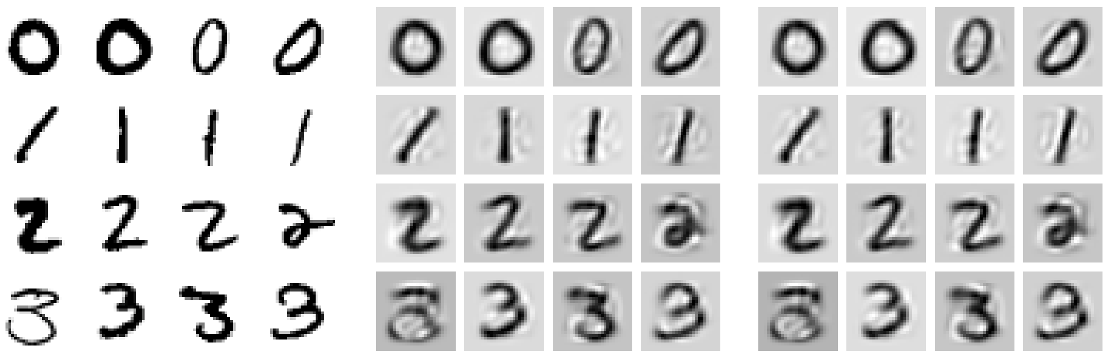

(a) Handwritten digits. (b) Deterministic. (c) Randomized.

Figure 21: Handwritten digits, and its low-rank approximations using $k = 40$ components.

it can be the case that the variance corresponds to noise rather than to the underlying signal. The question is, how good are the randomized principal components suited for this task? In the following, we use a simple k-nearest neighbor (kNN) algorithm to classify handwritten digits in order to compare the performance. The idea of kNN is to find the closest point (or set of points) to a given target point (Hastie, Tibshirani, and Friedman 2009). There are two reasons to use PCA for dimensionality reduction: (a) kNN is known to perform poorly in high-dimensional space, due to the "curse of dimensionality" (Donoho 2000); (b) kNN is computational expensive when high-dimensional data points are used for training.

First, we split the dataset into a training and a test set using the **caret** package (Kuhn 2008). We aim to create a balanced split of the dataset, using about 80% of the data for training:

```
R> library("caret")
R> trainIndex <- createDataPartition(label, p = 0.8, list = FALSE)
```

We then compute the dominant $k = 40$ randomized principal components of the training set:

```
R> train.rpca <- rpca(digits[trainIndex, ], k = 40, scale = FALSE)
```

We can use the `predict()` function to rotate the test set into low-dimensional space:

```
R> test.x <- predict(train.rpca, digits[-trainIndex, ])
```

The base **class** package provides a kNN algorithm, which we use for classification:

```
R> library("class")
R> knn.1 <- knn(train.rpca$x, test.x, label[trainIndex], k = 1)
```

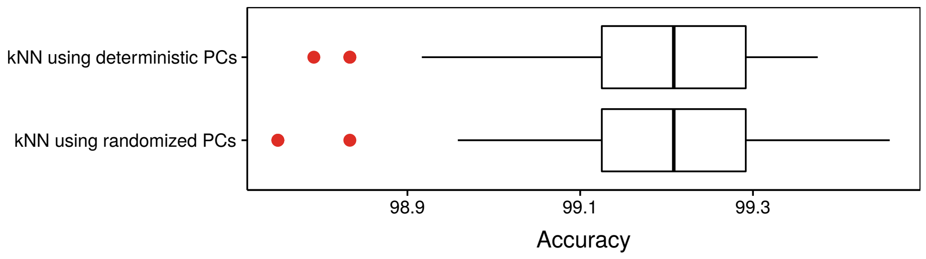


Figure 22: Performance of digits classification over 50 random splits. There is no significant difference in terms of the accuracy between using the randomized and deterministic PCs.

The test images are simply assigned to the class of the single nearest neighbor. The performance can be quantified by computing the accuracy, i.e., the number of correctly classified digits divided by the total number of predictions made:

```
R> 100 * sum(label[-trainIndex] == knn.1) / length(label[-trainIndex])
```

For comparison, the above steps can be repeated with the additional argument `rand = FALSE` or using the `prcomp()` function. Both the randomized PCA and the deterministic PCA algorithms achieve an accuracy of about 99.18%. Figure 22 shows the performance.

5. Randomized robust principal component analysis

Thus far, we have viewed matrix approximations as a factorization of a given matrix into a product of smaller (low-rank) matrices, as formulated in Equation 1. However, there is another interesting class of matrix decompositions, which aim to separate a given matrix into low-rank, sparse and noise components. Such decompositions are motivated by the need for robust methods which can more effectively account for corrupt or missing data. Indeed, outlier rejection is critical in many applications as data is rarely free of corrupt elements. Robustification methods decompose data matrices as follows:

$$\underset{m \times n}{\mathbf{A}} \approx \underset{m \times n}{\mathbf{L}} + \underset{m \times n}{\mathbf{S}} + \underset{m \times n}{\mathbf{E}},$$

where $\mathbf{L} \in \mathbb{R}^{m \times n}$ denotes the low-rank matrix, $\mathbf{S} \in \mathbb{R}^{m \times n}$ the sparse matrix, and $\mathbf{E} \in \mathbb{R}^{m \times n}$ the noise matrix. Note that the sparse matrix $\mathbf{S}$ represents the corrupted entries (outliers) of the data matrix $\mathbf{A}$. In the following we consider only the special case $\mathbf{A} = \mathbf{L} + \mathbf{S}$, i.e., the decomposition of a matrix into its sparse and low-rank components. This form of additive decomposition is also denoted as robust principal component analysis (RPCA), and its remarkable ability to separate high-dimensional matrices into low-rank and sparse component makes RPCA an invaluable tool for data science. The additive decomposition is, however, different from classical robust PCA methods known in the statistical literature. These techniques are concerned with computing robust estimators for the empirical covariance matrix, for instance, see the seminal work by Hubert, Rousseeuw, and Vanden Branden (2005) and Croux and Ruiz-Gazen (2005).

While traditional principal component analysis minimizes the spectral norm of the reconstruction error, robust PCA aims to recover the underlying low-rank matrix of a heavily corrupted input matrix. Candès, Li, Ma, and Wright (2011) proved that it is possible to exactly separate such a data matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ into both its low-rank and sparse components, under rather broad assumptions. This is achieved by solving a convenient convex optimization problem called *principal component pursuit* (PCP). The objective is to minimize a weighted combination of the nuclear norm $\|\cdot\|_* := \sum_i \sigma_i$ and the ℓ_1 norm $\|\cdot\|_1 := \sum_{ij} |m_{ij}|$ as

$$\min_{\mathbf{L},\mathbf{S}} \|\mathbf{L}\|_* + \lambda \|\mathbf{S}\|_1 \text{ subject to } \mathbf{A} - \mathbf{L} - \mathbf{S} = 0,$$

where λ is an arbitrary balance parameter which puts some weight on the sparse error term in the cost function. Typically λ is chosen to be $\lambda = \max\{n, m\}^{-0.5}$. The PCP concept is mathematically sound, and has been applied successfully to many applications like video surveillance and face recognition (Wright, Ganesh, Rao, Peng, and Ma 2009). Robust PCA is particular relevant if the underlying model of the data naturally features a low-rank subspace that is polluted with sparse components. The concept of matrix recovery can also be extended to the important problem of matrix completion.

The biggest challenge for robust PCA is computational efficiency, especially given the iterative nature of the optimization required. Bouwmans, Sobral, Javed, Jung, and Zahzah (2016) have identified more than 30 related algorithms to the original PCP approach, aiming to overcome the computational complexity, and to generalize the original algorithm.

5.1. The inexact augmented Lagrange multiplier method

A popular choice to compute RPCA, due to its favorable computational properties, is the inexact augmented Lagrange multiplier (IALM) method (Lin, Chen, and Ma 2011). This method formulates the following Lagrangian function

$$\mathcal{L}(\mathbf{L}, \mathbf{S}, \mathbf{Z}, \mu) = \|\mathbf{L}\|_* + \lambda \|\mathbf{S}\|_1 + \langle \mathbf{Z}, \mathbf{A} - \mathbf{L} - \mathbf{S} \rangle + \frac{\mu}{2} \|\mathbf{A} - \mathbf{L} - \mathbf{S}\|_F^2, \tag{12}$$

where μ and λ are positive scalars, and $\mathbf{Z}$ the Lagrange multiplier. Further, $\langle \cdot, \cdot \rangle$ is defined as $\langle \mathbf{A}, \mathbf{B} \rangle := \operatorname{trace}(\mathbf{A}^\top \mathbf{B})$. The method of augmented Lagrange multipliers can be used to solve the optimization problem (Bertsekas 1999). Lin *et al.* (2011) have proposed both an exact and inexact algorithm to solve Equation 12. Here, we advocate the latter approach. Specifically, the inexact algorithm avoids solving the problem

$$\mathbf{L}_{i+1}, \mathbf{E}_{i+1} = \operatorname*{argmin}_{\mathbf{L},\mathbf{E}} \mathcal{L}(\mathbf{L}, \mathbf{E}, \mathbf{Z}_i, \mu_k),$$

by alternately solving the following two sub-problems at step i:

$$\mathbf{L}_{i+1} = \operatorname*{argmin}_{\mathbf{L}} \mathcal{L}(\mathbf{L}, \mathbf{E}_i, \mathbf{Z}_i, \mu_k),$$

and

$$\mathbf{E}_{i+1} = \operatorname*{argmin}_{\mathbf{E}} \mathcal{L}(\mathbf{L}_{i+1}, \mathbf{E}, \mathbf{Z}_i, \mu_k).$$

For details, we refer the reader to Lin *et al.* (2011).

5.2. Randomized algorithm

The singular value decomposition is the workhorse algorithm behind the IALM method. Thus, the computational costs can become intractable for "big" datasets. However, randomized SVD

Input: Input matrix $\mathbf{A}$ with dimensions $m \times n$, and λ to put weight on the sparse error term.
Optional: Parameters p and q to control oversampling, and the power scheme.

```
function rrpca($\mathbf{A}, \lambda, p, q$)
(1)  $k = 2$                                                          initialize target rank
(2)  $\mu = 1.25 \cdot \|\mathbf{A}\|_2$                              initialize $\mu$
(3)  $\mathbf{Z} = \mathbf{A} \cdot \texttt{dual\_norm}(\mathbf{A})^{-1}$     initialize Lagrange multiplier
(4)  $\mathbf{S} = \texttt{matrix}(0, m, n)$                          initialize sparse matrix
(5)  repeat
(6)    $[\mathbf{U}, \boldsymbol{\Sigma}, \mathbf{V}] = \texttt{rsvd}(\mathbf{A} - \mathbf{S} + \mathbf{Z} \cdot \mu^{-1}, k, p, q)$     randomized SVD using Algorithm 5
(7)    $k, l = \texttt{predict\_rank}(\boldsymbol{\Sigma}, \mu^{-1})$     predicted rank, and updated target rank
(8)    $\boldsymbol{\Sigma}_l = \texttt{soft\_thres}(\texttt{diag}(\boldsymbol{\Sigma})(1:l), \mu^{-1})$     soft threshold top $l$ singular values
(9)    $\mathbf{L} = \mathbf{U}(:, 1:l)\boldsymbol{\Sigma}_l\mathbf{V}(:, 1:l)^\top$     update low-rank matrix
(10)   $\mathbf{S} = \texttt{soft\_thres}(\mathbf{A} - \mathbf{L} + \mathbf{Z} \cdot \mu^{-1}, \lambda \cdot \mu^{-1})$     update sparse matrix via soft thresholding
(11)   $\mathbf{Z} = \mathbf{Z} + (\mathbf{A} - \mathbf{L} - \mathbf{S}) \cdot \mu$     update Lagrange multiplier
(12)   update $\mu$
(13) until some convergence criterion is reached
```

Return: $\mathbf{L} \in \mathbb{R}^{m \times n}$, and $\mathbf{S} \in \mathbb{R}^{m \times n}$

Remark 3. Lin *et al.* (2011) provide details on how to predict the rank in Step (6).

Remark 4. Our randomized RPCA algorithm automatically switches from the randomized SVD to the deterministic SVD, if the target rank is predicted to be $k > \min\{m, n\}/4$.

Algorithm 7: A randomized robust PCA algorithm.

can be used to substantially ease the computational burden of the IALM method. Algorithm 7 outlines the randomized implementation.

5.3. Existing functionality for robust PCA in R

Only few R packages provide robust PCA routines. For comparison, we consider the **rpca** package (Sykulski 2015). The provided RPCA function implements the algorithm described by Candès *et al.* (2011). This algorithm is highly accurate. However, a large number of iterations is required for convergence.

5.4. The `rrpca()` function

The `rrpca()` function implements the inexact augmented Lagrange multiplier method. The interface of the `rrpca()` function takes the form of:

```
rrpca(A, lambda = NULL, maxiter = 50, tol = 1.0e-5, p = 10, q = 2,
  trace = FALSE, rand = TRUE)
```

The first mandatory argument `A` passes the $m \times n$ input data matrix. The second argument `lambda` is used to put some weight on the sparse error term in the cost function. By default λ

is set to $\lambda = \max\{n, m\}^{-0.5}$. The next two parameters `maxiter`, and `tol` are used to control the stopping criterion of the algorithm. The routine stops either if a specified maximal number of iterations, or if a certain tolerance level is reached. The parameters `p` and `q` are described in Section 2. The argument `rand` can be used to switch between the deterministic and randomized algorithms. By default the randomized algorithm is selected (i.e., the randomized SVD is used). Setting this argument `rand = FALSE` selects the deterministic algorithm (i.e., the deterministic SVD is used). To print out progress information, the argument `trace` can be set `TRUE`.

The resulting model object is a list and contains the following components:

- `L`: $m \times n$ matrix containing the low-rank component.
- `S`: $m \times n$ matrix containing the sparse component.

5.5. Robust PCA example: Grossly corrupted handwritten digits

To demonstrate the randomized robust PCA algorithm, we consider a grossly corrupted subset of the handwritten digits dataset. We first extract a subset comprising only twos:

```
R> data("digits", package = "rsvd")
R> two <- subset(digits[, 2:785], digits[, 1] == 2)
```

Then, we corrupt the data using salt and pepper noise, i.e., we draw i.i.d. uniform entries in the interval $[0, 255]$ and sparsify the matrix so that about 10% nonzero elements are retained:

```
R> m <- nrow(two); n <- ncol(two)
R> S <- matrix(runif(m * n, 0, 255), nrow = m, ncol = n)
R> S <- S * matrix(rbinom(m*n, size = 1, prob = 0.1), nrow = m, ncol = n)
```

The digits are then corrupted as follows:

```
R> two_noisy <- two + S
R> two_noisy <- ifelse(two_noisy > 255, 255, two_noisy)
```

Note, the last line ensures that the pixel intensities remain in the interval $[0, 255]$. Samples of the corrupted digits are shown in Figure 23a. Robust PCA is now used for matrix recovery (denoising) by separating the data into a low-rank and sparse component:

```
R> two.rrpca <- rrpca(two_noisy, trace = TRUE, rand = TRUE)
```

Figure 23c and 23d shows samples of the low-rank component $\mathbf{L}$ and the sparse component $\mathbf{S}$. For comparison, Figure 23b shows samples of the low-rank component which are computed using the deterministic routine (`rand = FALSE`).

Table 3 summarizes the computational results. The randomized routine does not show a computational advantage in this case. The reasons are twofold. First, the dataset requires a relatively large target rank to approximate the data accurately, i.e., in this example the predicted rank-k of the IALM algorithm for the final iteration is 232. Secondly, the data matrix is tall and thin (i.e., the ratio of rows to columns is large). For both of theses reasons, the performance of the deterministic algorithm remains competitive. The advantage of the randomized algorithms becomes pronounced for higher-dimensional problems which feature low-rank structure.

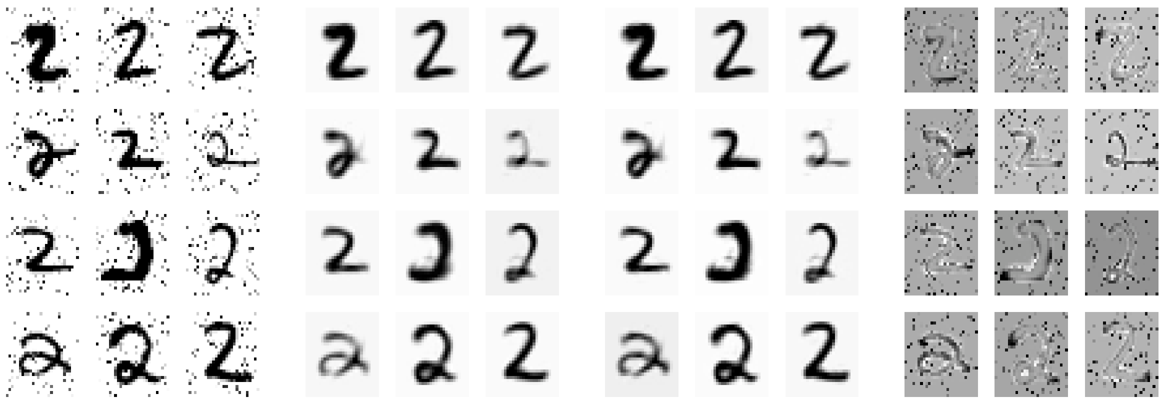

(a) Noisy digits. (b) Deterministic **L**. (c) Randomized **L**. (d) Randomized **S**.

Figure 23: Separation of noisy handwritten digits into a low-rank and a sparse component.

Package	Function	Parameters	Time (s)	Speedup	Error	Iterations
rpca	`rpca()`	`max.iter = 50`	11.88	*	0.265	50
rsvd	`rrpca()`	`rand = FALSE`	6.33	1.8	0.328	27
rsvd	`rrpca()`	`rand = TRUE`	5.56	2.1	0.329	27

Table 3: Summary of the computational performance of different RPCA functions.

It appears that the `rpca()` routine of the **rpca** package is more accurate, while computationally less attractive (the algorithm converges slowly). However, using the relative error for comparing the algorithms might be misleading since we do not know the ground truth. The relative error is computed using the original data as baseline, which contains some perturbations itself. Clearly, the advocated IALM algorithm removes not only the salt and paper noise, but also shadows, specularities, and saturations from the digits. This seems to be favorable, yet it leads to a larger relative error. Thus, to better compare the algorithms we perform a small simulation study using synthetic data. By superimposing a low-rank matrix with a sparse component, the ground truth is known. First, the low-rank component is generated:

```
R> m <- 300; n <- 300; k <- 5
R> L1 <- matrix(rnorm(m * k), nrow = m, ncol = k)
R> L2 <- matrix(rnorm(n * k), nrow = k, ncol = n)
R> L <- L1 %*% L2
```

The sparse component with about 20% nonzero i.i.d. uniform entries in the interval $[-500, 500]$ is generated from:

```
R> S <- matrix(runif(m * n, -500, 500), nrow = m, ncol = n)
R> S <- S * matrix(rbinom(m * n, size = 1, prob = 0.2), nrow = m, ncol = n)
```

The data matrix is then constructed by superimposing the low-rank and sparse components:

```
R> A <- L + S
```

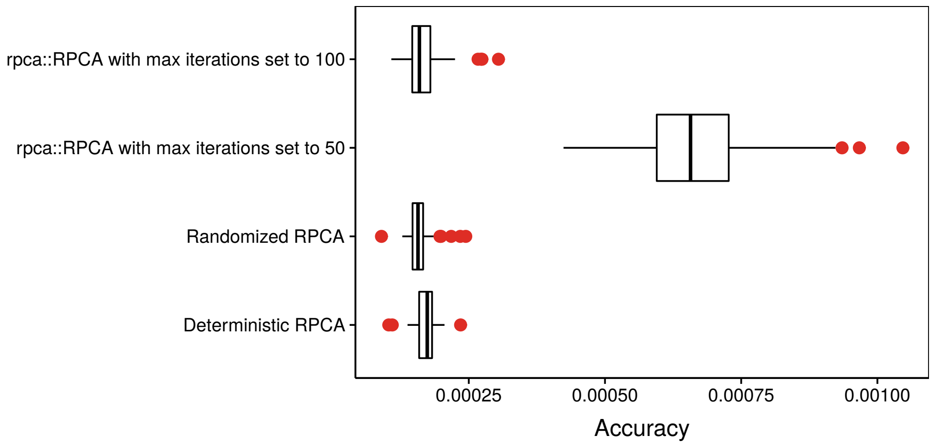


Figure 24: Matrix recovery performance of different RPCA algorithms.

Figure 24 shows the performance of the RPCA algorithms over 50 runs. Both the randomized and deterministic routines provided by the **rsvd** package show a better performance than the **rpca** package, when the maximum number of iterations is set to 50. In addition, we show the performance after 100 iterations for the RPCA algorithms from the **rpca** package.

6. Additional functionality

Principal components analysis seeks a set of new components which are formed as weighted linear combinations of the input data. This approach allows one to efficiently approximate and summarize the data, however, interpretation of the resulting components can be difficult. For instance, in a high-dimensional data setting it is cumbersome to interpret the large number of weights (loadings) required to form the components. While in many applications the eigenvectors have distinct meanings, the orthogonality constraints may not be physical meaningful in other problems. Thus, it is plausible to look for alternative factorizations which may not provide an optimal rank-k approximation, but which may preserve useful properties of the input matrix, such as sparsity and non-negativity as well as allowing for easier interpretation of its components. Such properties may be found in the CUR and the interpolative decompositions (ID), which are both tools for computing low-rank approximations.

6.1. Randomized CUR decomposition

Mahoney and Drineas (2009) introduced the CUR matrix decomposition, as an interesting alternative to traditional approximation techniques such as SVD and PCA. The CUR decomposition admits a factorization of the form

$$\underset{m \times n}{\mathbf{A}} \approx \underset{m \times k}{\mathbf{C}} \quad \underset{k \times k}{\mathbf{U}} \quad \underset{k \times n}{\mathbf{R}},$$

where the components of the matrix $\mathbf{C} \in \mathbb{R}^{m \times k}$ and $\mathbf{R} \in \mathbb{R}^{k \times n}$ are formed by small subsets of actual columns and rows, respectively. The matrix $\mathbf{U} \in \mathbb{R}^{k \times k}$ is formed so that $\|\mathbf{A} - \mathbf{CUR}\|_F$ is small. The CUR factorization is illustrated in Figure 25.

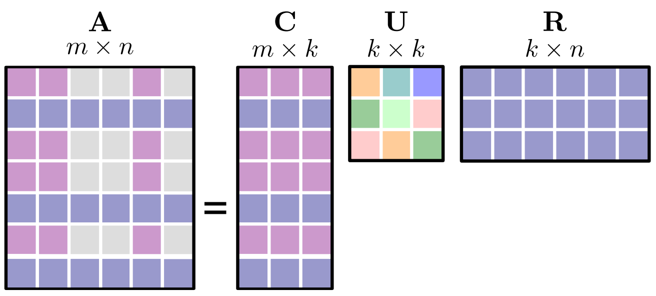


Figure 25: Schematic of the rank-k CUR decomposition of an $m \times n$ matrix. The components are formed by small subsets of actual columns and rows of the input matrix.

The low-rank factor matrices $\mathbf{C}$ and $\mathbf{R}$ are interpretable, since their components maintain the original structure of the data. This allows one to fully leverage any field-specific knowledge about the data, i.e., experts have often a clear understanding about the actual meaning of certain columns and rows. However, the CUR decomposition is not unique, and different computational strategies lead to different subsets of columns and rows, for instance, see Mahoney and Drineas (2009) and Boutsidis and Woodruff (2017). Thus, the practical meaning of the selected rows and columns should always be carefully examined depending on the problem and its objective.

Note, that the rank-k SVD ($\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^\top$) of a general $m \times n$ matrix $\mathbf{A}$ yields an optimal approximation of rank k to $\mathbf{A}$, in the sense that $\|\mathbf{A} - \mathbf{A}_k\| \leq \|\mathbf{A} - \mathbf{M}_k\|$ for any rank k matrix $\mathbf{M}_k$, both in the operator (spectral) and Frobenius norms. However, if $\mathbf{A}$ is a sparse matrix, the $m \times k$ and $n \times k$ factors $\mathbf{U}_k$ and $\mathbf{V}_k$ are typically dense. Even though the low-rank SVD is optimal for a given rank k, the choice of rank may be limited to relatively low values with respect to $\min(m, n)$ for sparse matrices, in order to achieve any useful compression ratios. Of course, the usefulness of the SVD is not limited to compression; but the utility of a low-rank approximation is greatly reduced once the storage size of the factors exceeds that of the original matrix. The CUR decomposition provides an interesting alternative for compression, since its components preserve sparsity.

The **rCUR** package provides an implementation in R (Bodor, Csabai, Mahoney, and Solymosi 2012). The **rsvd** package implements both the deterministic and randomized CUR decomposition, following the work by Voronin and Martinsson (2017). Specifically, the interpolative decomposition is used as an algorithmic tool to form the factor matrices $\mathbf{C}$ and $\mathbf{R}$. Algorithm 8 outlines the computational steps of the `rcur()` routine as implemented in the **rsvd** package.

6.2. The `rcur()` function

The `rcur()` function provides the option to compute both the deterministic and the randomized CUR decomposition via Algorithm 8. The interface of the `rcur()` function is as follows:

```
rcur(A, k, p = 10, q = 0, idx_only = FALSE, rand = TRUE)
```

The first mandatory argument `A` passes the $m \times n$ input data matrix. The second mandatory argument `k` sets the target rank, which is required to be $k < \min\{m, n\}$. The parameters `p` and `q` are described in Section 2. The argument `rand` can be used to switch between the deterministic and the randomized algorithm. The latter is used by default, and is more efficient

```
Input: Input matrix $\mathbf{A}$ with dimensions $m \times n$, and target rank $k < \min\{m, n\}$.
Optional: Parameters $p$ and $q$ to control oversampling, and the power scheme.

  function rcur($\mathbf{A}, k, p, q$)
(1)  $[\mathbf{C}, \mathbf{Z}, J] = \texttt{rid}(\mathbf{A}, k, p, q)$      randomized column ID (Algorithm 10)
(2)  $[\sim, \mathbf{S}, P] = \texttt{qr}(\mathbf{C}^\top)$              pivoted QR decomposition
(3)  $I = P(1:k)$                                 extract top $k$ row indices
(4)  $\mathbf{R} = \mathbf{A}(I, :)$                              extract $k$ rows from input matrix
(5)  $\mathbf{R}^\dagger = \texttt{pinv}(\mathbf{R})$                        compute pseudoinverse
(6)  $\mathbf{U} = \mathbf{Z}\mathbf{R}^\dagger$                               compute well-conditioned matrix
Return: $\mathbf{C} \in \mathbb{R}^{m\times k}$, $\mathbf{U} \in \mathbb{R}^{k\times k}$ and $\mathbf{R} \in \mathbb{R}^{k\times n}$
```

Remark 5. The deterministic rank-k CUR decomposition is computed by replacing the `rid()` function with the deterministic `id()` function, described in Algorithm 9.

Algorithm 8: A randomized CUR decomposition algorithm.

for large-scale matrices. The argument `idx_only` can be set to `TRUE` in order to return only the column and row index sets which is more memory efficient than returning $\mathbf{C}$ and $\mathbf{R}$.

The resulting model object is a list containing the following components:

- `C`: $m \times k$ matrix containing the column skeleton.
- `R`: $k \times n$ matrix containing the row skeleton.
- `U`: $k \times k$ matrix.
- `C.idx`: k-dimensional vector containing the column index set.
- `R.idx`: k-dimensional vector containing the row index set.

6.3. Randomized interpolative decomposition

The interpolative decomposition yields a low-rank factorization of the form

$$\underset{m\times n}{\mathbf{A}} \approx \underset{m\times k}{\mathbf{C}} \; \underset{k\times n}{\mathbf{Z}}.$$

The factor matrix $\mathbf{C} \in \mathbb{R}^{m\times k}$ is formed by a small number of columns, while $\mathbf{Z} \in \mathbb{R}^{k\times n}$ is a well-conditioned matrix containing the identity. $\mathbf{C}$ is also denoted as a skeleton matrix, and $\mathbf{Z}$ as the interpolation matrix.

The question is how to choose "interesting" columns of $\mathbf{A}$ to form $\mathbf{C}$? For certain datasets (such as images), one may choose the k columns corresponding to highest brightness/contrast, or highest amount of variation or detail. As an example, in a photograph of a building structure, the structure portion would be more critical than the ground or sky. Of course, a general method is needed to pick k columns from a matrix. In linear algebra, such a method exists: *pivoting*. Following Halko *et al.* (2011b), we advocate the QR factorization with pivoting

$$\underset{m\times n}{\mathbf{A}} \; \underset{n\times n}{\mathbf{P}} = \underset{m\times r}{\mathbf{Q}} \; \underset{r\times n}{\mathbf{S}},$$

where $r := \min\{m, n\}$. $\mathbf{P}$ is the permutation matrix, which simply dictates the re-arrangement

```
Input: Input matrix $\mathbf{A}$ with dimensions $m \times n$, and target rank $k < \min\{m, n\}$.

function id($\mathbf{A}, k$)

(1)  $[\sim, \mathbf{S}, P] = \texttt{qr}(\mathbf{A})$                        pivoted QR decomposition
(2)  $\mathbf{S}^{\dagger} = \texttt{pinv}(\mathbf{S}(1:k, 1:k))$              compute pseudoinverse
(3)  $\mathbf{T} = \mathbf{S}^{\dagger}\mathbf{S}(1:k, (k+1):n)$               compute expansions coefficients
(4)  $\mathbf{Z} = \texttt{matrix}(0, k, n)$                                   create empty $k \times n$ matrix
(5)  $\mathbf{Z}(:, P) = \texttt{cbind}(\texttt{diag(k)}, \mathbf{T})$         ordered expansions coefficients, using pivots $P$
(6)  $J = P(1:k)$                                                              extract top $k$ column indices from pivots
(7)  $\mathbf{C} = \mathbf{A}(:, J)$                                           extract $k$ columns from input matrix

Return: $\mathbf{C} \in \mathbb{R}^{m \times k}$, $\mathbf{Z} \in \mathbb{R}^{k \times n}$, and $J \in \mathbb{N}^{k}$
```

Remark 6. The QR decomposition returns the permutation matrix $\mathbf{P}$ in form of a vector $P \in \mathbb{R}^n$. This vector contains the indices such that $\mathbf{P} = \mathbf{I}(:, P)$, where $\mathbf{I} \in \mathbb{R}^{n \times n}$ denotes the identity matrix. Thus, J is comprised of the k dominant pivots.

Algorithm 9: An interpolative decomposition algorithm.

of the columns of $\mathbf{A}$. The matrix $\mathbf{Q}$ has orthonormal columns, and $\mathbf{S}$ is upper triangular.[1] Because, the pivoted QR decomposition is an iterative algorithm, it can be stopped after k iterations to obtain only the k dominant pivots. Thus, the column subset used to form $\mathbf{C}$ is simply based on the pivoting strategy used in the QR factorization. The computational steps required to compute the ID are outlined in Algorithm 9.

The procedure can be considerably accelerated by means of randomization. Specifically, we can first compute the randomized QB decomposition via Algorithm 4. Then, the smaller matrix $\mathbf{B}$ is used to compute the ID decomposition. The computational steps are outlined in Algorithm 10. For a detailed discussion, and theoretical results we refer to Voronin and Martinsson (2015), and Voronin and Martinsson (2017). Therein, it is also described how the factor matrix $\mathbf{Z}$ can be efficiently constructed. Note, however, that our algorithm differs from the implementation by Voronin and Martinsson (2015). We compute the ID based on the matrix $\mathbf{B}$ (obtained as described in Section 2). In our experiments, this approach shows to be more accurate, while slightly more computational demanding.

6.4. The `rid()` function

The `rid()` function provides the option to compute both the deterministic and the randomized interpolative decomposition via Algorithms 9 and 10. The interface of the `rid()` function takes the following functional form:

```
rid(A, k, mode = "col", p = 10, q = 0, idx_only = FALSE, rand = TRUE)
```

The first mandatory argument `A` passes the $m \times n$ input data matrix. The second mandatory argument `k` sets the desired target rank, which is required to be $k < \min\{m, n\}$. The argument `mode = c("col", "row")` determines whether the column or row ID should be computed.

[1]Here, we denote the upper triangular matrix as $\mathbf{S}$, since $\mathbf{R}$ is occupied by the CUR decomposition.

```
Input: Input matrix $\mathbf{A}$ with dimensions $m \times n$, and target rank $k < \min\{m, n\}$.
Optional: Parameters $p$ and $q$ to control oversampling and the power scheme.

  function rid($\mathbf{A}, k, p, q$)
(1)   $[\sim, \mathbf{B}] = \texttt{rqb}(\mathbf{A}, k, q, p)$      randomized QB decomposition via Algorithm 4
(2)   $[\sim, \mathbf{Z}, J] = \texttt{id}(\mathbf{B}, k)$        column ID via Algorithm 9
(3)   $\mathbf{C} = \mathbf{A}(:, J)$                  extract $k$ columns from input matrix

Return: $\mathbf{C} \in \mathbb{R}^{m\times k}$, $\mathbf{V} \in \mathbb{R}^{k\times n}$, and $J \in \mathbb{N}^k$
```

Remark 7. The row ID can be computed by transposing the input matrix, i.e., $\mathbf{A}^\top$.

Algorithm 10: A randomized interpolative decomposition algorithm.

The parameters `p` and `q` are described in Section 2. The argument `rand` can be used to switch between the deterministic and randomized algorithms. By default the randomized algorithm is selected, and setting this argument `rand = FALSE` selects the deterministic algorithm.

The resulting model object is a list and contains the following components:

- `C`: $m \times k$ matrix containing the column skeleton, if `mode = "col"`.
- `R`: $k \times n$ matrix containing the row skeleton, if `mode = "row"`.
- `Z`: $k \times n$ or $m \times k$ matrix (depending on `mode`), which is well-conditioned.
- `idx`: k-dimensional vector containing the column or row index set.

7. Conclusion

Dimensionality reduction and the related concept of low-rank matrix approximations are fundamental algorithmic tools in machine learning and computational statistics. However, high-dimensional data pose a growing computational challenge for traditional matrix algorithms. In fact, the exponential growth rate of data is far outstripping advances in computational power, even of modern computational architectures. Thus, in the era of “big data”, the modern computational paradigm of randomized methods for linear algebra provides an attractive method for scalable, tractable computations. The price to pay is the trade-off between the approximation accuracy and computational costs. In fact, randomized methods are highly scalable, and can be used to tackle problems which are infeasible otherwise. The different flavors of both deterministic and randomized methods are illustrated in Figure 26. Thus, randomized algorithms should be the default choice for applications which involve low-rank matrices and do not require approximations with full double precision.

Certainly, the randomized singular value decomposition is the most prominent and ubiquitous randomized algorithm. This algorithm comes with strong theoretical error bounds, and the approximation quality can be controlled via oversampling, and power iterations. The computational advantage can be substantial compared to other SVD routines in R, provided the target rank k is relatively small. While the performance of randomized methods depends on the actual shape of the matrix, we can state (as a rule of thumb) that significant computational speedups are achieved if the target rank k is at least 3–6 times smaller than the ambient dimensions of the measurement space. The speedup for tall and thin matrices is in general

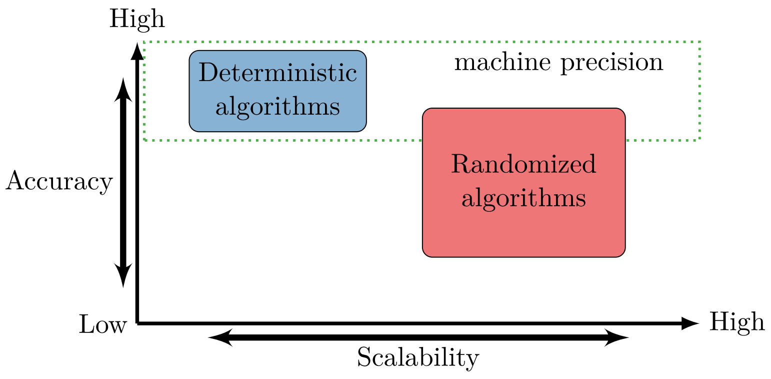


Figure 26: Trade-off between accuracy and scalability. Randomized methods for linear algebra allowing for a scalable architecture for modern "big data" applications.

less impressive than for fat matrices. In addition, the R package **rsvd** provides several other randomized matrix decomposition routines, which are all designed for mid-sized problems, i.e., the input matrix is assumed to fit into fast memory. To fully exploit the power of randomized methods, we recommend to use the enhanced R distribution Microsoft R Open which allows one to use all of the computational resources available.

Future developments of the **rsvd** package will use randomized methods to compute linear discriminant analysis, principal component regression, canonical correlation analysis, and matrix completion problems. Another important direction is to better integrate the **Matrix** package, for instance, to provide efficient routines which allow to deal better with large-scale sparse matrices (Bates and Maechler 2016).

Acknowledgments

We would like to express our gratitude to Edzer Pebesma and Julie Josse, and particularly to the two anonymous reviewers. Their helpful feedback allowed us to greatly improve the manuscript. In particular, we would like to thank Gunnar Martinsson for insightful conversations around randomized methods. It is also a pleasure to thank Bryan W. Lewis for illuminating discussions. Further, we thank Thierry Bouwmans, David Harris-Birtill, Aleksandr Aravkin and Lionel Mathelin for many useful comments. NBE acknowledges support from the UK Engineering and Physical Sciences Research Council. JNK acknowledges support from Air Force Office of Scientific Research (AFOSR) grant FA9550-15-1-0385. SLB acknowledges support from the Army Research Office (ARO) young investigator award grant W911NF-17-1-0422, and from the Air Force Research Laboratory (AFRL) grant FA8651-16-1-0003.

References

Ahfock D, Astle WJ, Richardson S (2017). "Statistical Properties of Sketching Algorithms." *arXiv preprint arXiv:1706.03665*. `https://arxiv.org/abs/1706.03665`.

Anderson E, Bai Z, Bischof C, Blackford S, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999). ***LAPACK** Users' Guide*. 3rd edition. SIAM.

Baglama J, Reichel L (2005). "Augmented Implicitly Restarted Lanczos Bidiagonalization Methods." *SIAM Journal on Scientific Computing*, **27**(1), 19–42.

Baglama J, Reichel L, Lewis BW (2017). ***irlba**: Fast Truncated Singular Value Decomposition and Principal Components Analysis for Large Dense and Sparse Matrices.* R package version 2.2.1, URL https://CRAN.R-project.org/package=irlba.

Bates D, Maechler M (2016). ***Matrix**: Sparse and Dense Matrix Classes and Methods.* R package version 1.2-6, URL https://CRAN.R-project.org/package=Matrix.

Bertsekas DP (1999). *Nonlinear programming*. Athena scientific Belmont.

Bodor A, Csabai I, Mahoney MW, Solymosi N (2012). "**rCUR**: An R Package for CUR Matrix Decomposition." *BMC Bioinformatics*, **13**(1), 103.

Boutsidis C, Woodruff DP (2017). "Optimal CUR Matrix Decompositions." *SIAM Journal on Computing*, **46**(2), 543–589.

Bouveyron C, Latouche P, Mattei PA (2017). "Exact Dimensionality Selection for Bayesian PCA." *arXiv preprint arXiv:1703.02834*. https://arxiv.org/abs/1703.02834.

Bouwmans T, Sobral A, Javed S, Jung SK, Zahzah EH (2016). "Decomposition into Low-Rank Plus Additive Matrices for Background/Foreground Separation: A Review for a Comparative Evaluation with a Large-Scale Dataset." *Computer Science Review*, pp. 1–71. doi:10.1016/j.cosrev.2016.11.001.

Burges CJ (2010). "Dimension Reduction: A Guided Tour." *Foundations and Trends in Machine Learning*, **2**(4), 275–365.

Calvetti D, Reichel L, Sorensen DC (1994). "An Implicitly Restarted Lanczos Method for Large Symmetric Eigenvalue Problems." *Electronic Transactions on Numerical Analysis*, **2**(1), 21.

Candès EJ, Li X, Ma Y, Wright J (2011). "Robust Principal Component Analysis?" *Journal of the ACM*, **58**(3), 1–37. doi:10.1145/1970392.1970395.

Cardot H, Degras D (2015). "Online Principal Component Analysis in High Dimension: Which Algorithm to Choose?" *arXiv preprint arXiv:1511.03688*.

Cipra BA (2000). "The Best of the 20th Century: Editors Name Top 10 Algorithms." *SIAM news*, **33**(4), 1–2.

Croux C, Ruiz-Gazen A (2005). "High Breakdown Estimators for Principal Components: The Projection-Pursuit Approach Revisited." *Journal of Multivariate Analysis*, **95**(1), 206–226.

Cunningham JP, Ghahramani Z (2015). "Linear Dimensionality Reduction: Survey, Insights, and Generalizations." *Journal of Machine Learning Research*, **16**, 2859–2900.

Degras D, Cardot H (2016). ***onlinePCA**: Online Principal Component Analysis.* R package version 1.3.1, URL https://cran.r-project.org/package=onlinePCA.

Degras D, Cardot H (2017). ***idm**: Incremental Decomposition Methods.* R package version 1.8.1, URL https://cran.r-project.org/package=idm.

Demmel J (1997). *Applied Numerical Linear Algebra.* SIAM.

Donoho DL (2000). "High-Dimensional Data Analysis: The Curses and Blessings of Dimensionality." *AMS Math Challenges Lecture*, pp. 1–32.

Dray S, Dufour A (2007). "The **ade4** Package: Implementing the Duality Diagram for Ecologists." *Journal of Statistical Software*, **22**(4), 1–20.

Drineas P, Mahoney MW (2016). "RandNLA: Randomized Numerical Linear Algebra." *Communications of the ACM*, **59**, 80–90.

Duersch JA, Gu M (2017). "Randomized QR with Column Pivoting." *SIAM Journal on Scientific Computing*, **39**(4), C263–C291. doi:10.1137/15M1044680.

Eckart C, Young G (1936). "The Approximation of one Matrix by Another of Lower Rank." *Psychometrika*, **1**, 211–218.

Erichson NB, Brunton SL, Kutz JN (2017). "Randomized Dynamic Mode Decomposition." *arXiv preprint arXiv:1702.02912.*

Frieze A, Kannan R, Vempala S (2004). "Fast Monte-Carlo Algorithms for Finding Low-Rank Approximations." *Journal of the ACM*, **51**(6), 1025–1041.

Gavish M, Donoho DL (2014). "The Optimal Hard Threshold for Singular Values is $4/\sqrt{3}$." *IEEE Transactions on Information Theory*, **60**(8), 5040–5053.

Golub G, Kahan W (1965). "Calculating the Singular Values and Pseudo-Inverse of a Matrix." *SIAM, Series B: Numerical Analysis*, **2**(2), 205–224.

Golub GH, Reinsch C (1970). "Singular Value Decomposition and Least Squares Solutions." *Numerische Mathematik*, **14**, 403–420.

Golub GH, Van Loan CF (1996). *Matrix Computations.* 3 edition. Johns Hopkins University Press.

Gu M (2015). "Subspace Iteration Randomization and Singular Value Problems." *SIAM Journal on Scientific Computing*, **37**(3), 1139–1173. doi:10.1137/130938700.

Halko N, Martinsson PG, Shkolnisky Y, Tygert M (2011a). "An Algorithm for the Principal Component Analysis of Large Data Sets." *SIAM Journal on Scientific Computing*, **33**, 2580–2594.

Halko N, Martinsson PG, Tropp JA (2011b). "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions." *SIAM Review*, **53**(2), 217–288. doi:10.1137/090771806.

Hastie T, Tibshirani R, Friedman J (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Series in Statistics, 2nd edition. Springer-Verlag.

Hotelling H (1933). "Analysis of a Complex of Statistical Variables into Principal Components." *Journal of Educational Psychology*, **24**(6), 417.

Hubert M, Rousseeuw PJ, Vanden Branden K (2005). "ROBPCA: A New Approach to Robust Principal Component Analysis." *Technometrics*, **47**(1), 64–79.

Johnson WB, Lindenstrauss J (1984). "Extensions of Lipschitz Mappings into a Hilbert Space." *Contemporary mathematics*, **26**(189-206), 1.

Jolliffe I (2002). *Principal Component Analysis*. Springer Series in Statistics, 2nd edition. Springer-Verlag.

Josse J, Husson F (2012). "Selecting the Number of Components in Principal Component Analysis using Cross-Validation Approximations." *Computational Statistics & Data Analysis*, **56**(6), 1869–1879.

Kessy A, Lewin A, Strimmer K (2018). "Optimal Whitening and Decorrelation." *The American Statistician*, **0**(0), 1–6. doi:10.1080/00031305.2016.1277159.

Korobeynikov A, Larsen RM (2016). ***svd**: Interfaces to Various State-of-Art SVD and Eigensolvers*. R package version 0.4, URL https://CRAN.R-project.org/package=svd.

Kuhn M (2008). "Caret package." *Journal of Statistical Software*, **28**(5), 1–26.

Larsen RM (1998). "Lanczos Bidiagonalization with Partial Reorthogonalization." *DAIMI Report Series*, **27**, 1–101.

Lecun Y, Bottou L, Bengio Y, Haffner P (1998). "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE*, **86**(11), 2278–2324. doi:10.1109/5.726791.

Lehoucq R, Sorensen D, Yang C (1998). ***ARPACK** Users' Guide*. SIAM.

Li P, Hastie TJ, Church KW (2006). "Very sparse random projections." In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 287–296. ACM.

Liberty E (2013). "Simple and Deterministic Matrix Sketching." In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 581–588. ACM.

Liberty E, Woolfe F, Martinsson PG, Rokhlin V, Tygert M (2007). "Randomized Algorithms for the Low-Rank Approximation of Matrices." *Proceedings of the National Academy of Sciences*, **104**, 20167–20172.

Lin Z, Chen M, Ma Y (2011). "The augmented Lagrange Multiplier Method for Exact Recovery of Corrupted Low-Rank Matrices." *arXiv preprint arXiv:1009.5055*.

LÃł S, Josse J, Husson F (2008). "FactoMineR: An R Package for Multivariate Analysis." *Journal of Statistical Software*, **25**(1), 1–18. doi:10.18637/jss.v025.i01.

Mahoney MW (2011). "Randomized Algorithms for Matrices and Data." *Foundations and Trends in Machine Learning*, **3**, 123–224.

Mahoney MW, Drineas P (2009). "CUR Matrix Decompositions for Improved Data Analysis." *Proceedings of the National Academy of Sciences*, **106**(3), 697–702.

Martinsson PG (2016). "Randomized Methods for Matrix Computations and Analysis of High Dimensional Data." *arXiv preprint arXiv:1607.01649*. `https://arxiv.org/abs/1607.01649`.

Martinsson PG, Rokhlin V, Tygert M (2011). "A Randomized Algorithm for the Decomposition of Matrices." *Applied and Computational Harmonic Analysis*, **30**, 47–68.

Mersmann O, Beleites C, Hurling R, Friedman A (2015). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4-2.1, URL `http://CRAN.R-project.org/package=microbenchmark`.

Motwani R, Raghavan P (1995). *Randomized Algorithms*. Cambridge University Press.

Pearson K (1901). "On Lines and Planes of Closest Fit to Systems of Points in Space." *Philosophical Magazine Series 6*, **2**, 559–572.

Qiu Y, Mei J, Guennebaud G, Niesen J (2016). ***RSpectra**: Solvers for Large Scale Eigenvalue and SVD Problems*. R package version 0.12-0, URL `https://CRAN.R-project.org/package=RSpectra`.

Rokhlin V, Szlam A, Tygert M (2009). "A Randomized Algorithm for Principal Component Analysis." *SIAM Journal on Matrix Analysis and Applications*, **31**, 1100–1124.

Sarlos T (2006). "Improved Approximation Algorithms for Large Matrices via Random Projections." In *Foundations of Computer Science. 47th Annual IEEE Symposium on*, pp. 143–152.

Shabat G, Shmueli Y, Aizenbud Y, Averbuch A (2016). "Randomized LU decomposition." *Applied and Computational Harmonic Analysis*.

Stewart GW (1993). "On the Early History of the Singular Value Decomposition." *SIAM Review*, **35**, 551–566.

Sykulski M (2015). ***rpca**: RobustPCA: Decompose a Matrix into Low-Rank and Sparse Components*. R package version 0.2.3, URL `http://CRAN.R-project.org/package=rpca`.

Szlam A, Kluger Y, Tygert M (2014). "An Implementation of a Randomized Algorithm for Principal Component Analysis." *arXiv preprint arXiv:1412.3510*.

Trefethen LN, Bau D (1997). *Numerical Linear Algebra*. SIAM.

Tropp JA, Yurtsever A, Udell M, Cevher V (2016). "Randomized Single-View Algorithms for Low-Rank Matrix Approximation." *arXiv preprint arXiv:1609.00048*.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S-PLUS*. Statistics and Computing. Springer-Verlag.

Voronin S, Martinsson PG (2015). "RSVDPACK: Subroutines for Computing Partial Singular Value Decompositions via Randomized Sampling on Single Core, Multi Core, and GPU Architectures." *arXiv preprint arXiv:1502.05366*. `http://arxiv.org/abs/1502.05366`.

Voronin S, Martinsson PG (2017). "Efficient Algorithms for CUR and Interpolative Matrix Decompositions." *Advances in Computational Mathematics*, **43**(3), 495–516.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis.* Springer-Verlag. URL http://had.co.nz/ggplot2/book.

Witten R, Candes E (2015). "Randomized algorithms for low-rank matrix factorizations: sharp performance bounds." *Algorithmica*, **72**, 264–281.

Woolfe F, Liberty E, Rokhlin V, Tygert M (2008). "A Fast Randomized Algorithm for the Approximation of Matrices." *Journal of Applied and Computational Harmonic Analysis*, **25**(3), 335–366.

Wright J, Ganesh A, Rao S, Peng Y, Ma Y (2009). "Robust Principal Component Analysis: Exact Recovery of Corrupted Low-Rank Matrices via Convex Optimization." In *Advances in Neural Information Processing Systems*, pp. 2080–2088.

Affiliation:

N. Benjamin Erichson
School of Mathematics and Statistics
University of St Andrews
St Andrews, UK
E-mail: erichson@uw.edu