Randomized algorithms for matrix computations and data analysis

Gunnar Martinsson

Website: `http://people.maths.ox.ac.uk/martinsson/Teaching/2018_random_matrix/`

Randomized dimension reduction

Let $\{\mathbf{a}^{(j)}\}_{j=1}^{n}$ be a set of points in $\mathbb{R}^m$, where m is very large. Consider tasks such as:

- Suppose the points almost live on a linear subspace of (small) dimension k.
  Find a basis for the "best" subspace.
- Given k, find the subset of k vectors with maximal spanning volume.
- Suppose the points almost live on a low-dimensional nonlinear manifold.
  Find a parameterization of the manifold.
- Given k, find for each vector $\mathbf{a}^{(j)}$ its k closest neighbors.
- Partition the points into clusters.

(Note: Some problems have well-defined solutions; some do not. The first can be solved with algorithms with moderate complexity; some are combinatorially hard.)

Randomized dimension reduction

Let $\{\mathbf{a}^{(j)}\}_{j=1}^{n}$ be a set of points in $\mathbb{R}^m$, where m is very large. Consider tasks such as:

- Suppose the points almost live on a linear subspace of (small) dimension k.
  Find a basis for the "best" subspace.
- Given k, find the subset of k vectors with maximal spanning volume.
- Suppose the points almost live on a low-dimensional nonlinear manifold.
  Find a parameterization of the manifold.
- Given k, find for each vector $\mathbf{a}^{(j)}$ its k closest neighbors.
- Partition the points into clusters.

(Note: Some problems have well-defined solutions; some do not. The first can be solved with algorithms with moderate complexity; some are combinatorially hard.)

Idea: Find an embedding $f : \mathbb{R}^m \to \mathbb{R}^d$ for $d \ll m$ that is almost isometric in the sense

$$\|f(\mathbf{a}^{(i)}) - f(\mathbf{a}^{(j)})\| \approx \|\mathbf{a}^{(i)} - \mathbf{a}^{(j)}\|, \qquad \forall\, i,j \in \{1,2,3,\ldots,n\}.$$

Then solve the problems for the vectors $\{f(\mathbf{a}^{(j)})\}_{j=1}^{n}$ in $\mathbb{R}^d$.

Randomized dimension reduction

Let $\{\mathbf{a}^{(j)}\}_{j=1}^{n}$ be a set of points in $\mathbb{R}^m$, where m is very large. Consider tasks such as:

- Suppose the points almost live on a linear subspace of (small) dimension k.
  Find a basis for the "best" subspace.
- Given k, find the subset of k vectors with maximal spanning volume.
- Suppose the points almost live on a low-dimensional nonlinear manifold.
  Find a parameterization of the manifold.
- Given k, find for each vector $\mathbf{a}^{(j)}$ its k closest neighbors.
- Partition the points into clusters.

(Note: Some problems have well-defined solutions; some do not. The first can be solved with algorithms with moderate complexity; some are combinatorially hard.)

Idea: Find an embedding $f : \mathbb{R}^m \to \mathbb{R}^d$ for $d \ll m$ that is almost isometric in the sense

$$\|f(\mathbf{a}^{(i)}) - f(\mathbf{a}^{(j)})\| \approx \|\mathbf{a}^{(i)} - \mathbf{a}^{(j)}\|, \qquad \forall\, i,j \in \{1,2,3,\ldots,n\}.$$

Then solve the problems for the vectors $\{f(\mathbf{a}^{(j)})\}_{j=1}^{n}$ in $\mathbb{R}^d$.

Lemma [Johnson-Lindenstrauss]: For $d \sim \log(n)$, there exists a well-behaved embedding f that "approximately" preserves distances.

To be precise, we have:

Lemma [Johnson-Lindenstrauss]: *Let ε be a real number such that $\varepsilon \in (0, 1)$, let n be a positive integer, and let d be an integer such that*

$$d \geq 4 \left(\frac{\varepsilon^2}{2} - \frac{\varepsilon^3}{3} \right)^{-1} \log(n). \tag{1}$$

Then for any set V of n points in $\mathbb{R}^m$, there is a map $f : \mathbb{R}^m \to \mathbb{R}^d$ such that

$$(1 - \varepsilon) \|\mathbf{u} - \mathbf{v}\|^2 \leq \|f(\mathbf{u}) - f(\mathbf{v})\|^2 \leq (1 + \varepsilon) \|\mathbf{u} - \mathbf{v}\|^2, \qquad \forall\, \mathbf{u}, \mathbf{v} \in V. \tag{2}$$

You can prove that if you pick d as specified, and draw a Gaussian random matrix $\mathbf{R}$ of size $d \times m$, then there is a positive likelihood that the map

$$f(\mathbf{x}) = \frac{1}{\sqrt{d}} \mathbf{R}\, \mathbf{x}$$

satisfies the criteria.

Practical problem: You have two very bad choices:

(1) Pick a small ε; then you get small distortions, but a huge d since $d \sim \frac{8}{\varepsilon^2} \log(n)$.

(2) Pick ε that is not close to 0; then distortions are large.

Question: Is it possible to build algorithms that combine the powerful dimension reduction capability of randomized projections with the accuracy and robustness of classical deterministic methods?

Question: Is it possible to build algorithms that combine the powerful dimension reduction capability of randomized projections with the accuracy and robustness of classical deterministic methods?

Putative answer: Yes — use a two-stage approach:

(A) *Randomized sketching:*

In a pre-computation, random projections are used to create low-dimensional sketches of the high-dimensional data. These sketches are somewhat distorted, but approximately preserve key properties to very high probability.

(B) *Deterministic post-processing:*

Once a sketch of the data has been constructed in Stage A, classical deterministic techniques are used to compute desired quantities to very high accuracy, *starting directly from the original high-dimensional data*.

Example 1 of two-stage approach: Nearest neighbor search in $\mathbb{R}^m$

Objective: Suppose you are given n points $\{\mathbf{a}^{(j)}\}_{j=1}^{n}$ in $\mathbb{R}^m$. The coordinate matrix is

$$\mathbf{A} = \left[\mathbf{a}^{(1)}\ \mathbf{a}^{(2)}\ \ldots\ \mathbf{a}^{(n)}\right] \in \mathbb{R}^{m\times n}.$$

How do you find the k nearest neighbors for every point?

If m is "small" (say $m \leq 10$ or so), then you have several options; you can, e.g, sort the points into a tree based on hierarchically partitioning space (a "kd-tree").

Problem: Classical techniques of this type get very expensive as m grows.

Simple idea: Use a random map to project onto low-dimensional space. This "sort of" preserves distances. Execute a fast search there.

Improved idea: The output from a single random projection is unreliable. But, you can repeat the experiment several times, use these to generate a list of *candidates* for the nearest neighbors, and then compute exact distances to find the k closest among the candidates.

Example 1 of two-stage approach: Nearest neighbor search in $\mathbb{R}^m$

Objective: Suppose you are given n points $\{\mathbf{a}^{(j)}\}_{j=1}^{n}$ in $\mathbb{R}^m$. The coordinate matrix is

$$\mathbf{A} = \left[\mathbf{a}^{(1)}\ \mathbf{a}^{(2)}\ \ldots\ \mathbf{a}^{(n)}\right] \in \mathbb{R}^{m\times n}.$$

How do you find the k nearest neighbors for every point?

(A) *Randomized probing of data:* Use a Johnson-Lindenstrauss random projection to map the n-particle problem in $\mathbb{R}^m$ (where m is large) to an n-particle problem in $\mathbb{R}^d$ where $d \sim \log n$. Run a deterministic nearest-neighbor search in $\mathbb{R}^d$ and store a list of the ℓ nearest neighbors for each particle (for simplicity, one can set $\ell = k$). Then repeat the process several times. If for a given particle a previously undetected neighbor is discovered, then simply add it to a list of potential neighbors.

(B) *Deterministic post-processing:* The randomized probing will result in a list of putative neighbors that typically contains more than k elements. But it is now easy to compute the pairwise distances in the original space $\mathbb{R}^m$ to judge which candidates in the list are the k nearest neighbors.

Jones, Osipov, Rokhlin, 2011

Example 2 of two-stage approach: Randomized SVD

Objective: Given an $m \times n$ matrix $\mathbf{A}$, find an approximate rank-k partial SVD:

$$\begin{array}{ccccc} \mathbf{A} & \approx & \mathbf{U} & \mathbf{D} & \mathbf{V}^* \\ m \times n & & m \times k & k \times k & k \times n \end{array}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{D}$ is diagonal. (We assume $k \ll \min(m, n)$.)

(A) *Randomized sketching:*

Use randomized projection methods to form an approximate basis for the range of the matrix.

(B) *Deterministic post-processing:*

Restrict the matrix to the subspace determined in Stage A, and perform expensive but accurate computations on the resulting smaller matrix.

Observe that distortions in the randomized projections are fine, since all we need is a subspace that captures "the essential" part of the range. Pollution from unwanted singular modes is harmless, as long as we capture the dominant ones. The risk of missing the dominant ones is for practical purposes zero.

Example 2 of two-stage approach: Randomized SVD

Objective: Given an $m \times n$ matrix $\mathbf{A}$, find an approximate rank-k partial SVD:

$$\begin{array}{ccccc} \mathbf{A} & \approx & \mathbf{U} & \mathbf{D} & \mathbf{V}^* \\ m \times n & & m \times k & k \times k & k \times n \end{array}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{D}$ is diagonal. (We assume $k \ll \min(m, n)$.)

(A) *Randomized sketching:*

A.1 Draw an $n \times k$ Gaussian random matrix $\mathbf{R}$.

A.2 Form the $m \times k$ sample matrix $\mathbf{Y} = \mathbf{A}\,\mathbf{R}$.

A.3 Form an $m \times k$ orthonormal matrix $\mathbf{Q}$ such that $\mathbf{Y} = \mathbf{Q}\,\mathbf{S}$.

(B) *Deterministic post-processing:*

B.1 Form the $k \times n$ matrix $\mathbf{B} = \mathbf{Q}^*\,\mathbf{A}$.

B.2 Form SVD of the matrix $\mathbf{B}$: $\mathbf{B} = \hat{\mathbf{U}}\,\mathbf{D}\,\mathbf{V}^*$.

B.3 Form the matrix $\mathbf{U} = \mathbf{Q}\,\hat{\mathbf{U}}$.

The class will spend several lectures describing and analyzing this "Randomized SVD (RSVD)" algorithm, and other closely related techniques for factorizating matrices.

Applications of low-rank approximation of a matrix:

- Computational statistics (Principal Component Analysis, regression analysis, etc).
- Data mining, machine learning, analysis of network matrices, imaging, etc. (Algorithms related to PageRank, Latent Semantic Indexing, relaxed versions of k-means clustering, etc.)
- Accelerating standard packages for linear algebra. *Including methods for full factorizations.*
- Diffusion geometry; a technique for constructing parameterizations on large collections of data points organized (modulo noise) along non-linear low-dimensional manifolds. Requires the computations of eigenvectors of *graph Laplace operators*.
- Fast algorithms for elliptic PDEs: more efficient Fast Multipole Methods, fast *direct* solvers, construction of special quadratures for corners and edges, etc.
- "General" pre-conditioners? *Current research area.*
- Etc.

Review of existing methods I

For a dense $n \times n$ matrix that fits in RAM, excellent algorithms are already part of LAPACK (and incorporated into Matlab, Mathematica, *etc*).

- Double precision accuracy.
- Very stable.
- $O(n^3)$ asymptotic complexity. Reasonably small constants.
- Require extensive random access to the matrix.

When the target rank k is much smaller than n, there also exist $O(n^2\,k)$ methods with similar characteristics (the well-known Golub-Businger method, RRQR by Gu and Eisentstat, *etc*).

For small matrices, the state-of-the-art is quite satisfactory.

(By "small," we mean something like $n \leq 10\,000$ on today's computers.)

Review of existing methods I

For a dense $n \times n$ matrix that fits in RAM, excellent algorithms are already part of LAPACK (and incorporated into Matlab, Mathematica, *etc*).

- Double precision accuracy.
- Very stable.
- $O(n^3)$ asymptotic complexity. Reasonably small constants.
- Require extensive random access to the matrix.

When the target rank k is much smaller than n, there also exist $O(n^2\,k)$ methods with similar characteristics (the well-known Golub-Businger method, RRQR by Gu and Eisentstat, *etc*).

For small matrices, the state-of-the-art is quite satisfactory.

(By "small," we mean something like $n \leq 10\,000$ on today's computers.)

For kicks, we will improve on it anyway, but this is not the main point.

Review of existing methods II

If the matrix is large, but can rapidly be applied to a vector (if it is sparse, or sparse in Fourier space, or amenable to the FMM, etc.), so called *Krylov subspace methods* often yield excellent accuracy and speed.

The idea is to pick a starting vector $\mathbf{r}$ (often a random vector), "restrict" the matrix $\mathbf{A}$ to the k-dimensionsal "Krylov subspace"

$$\text{Span}(\mathbf{r},\ \mathbf{A}\,\mathbf{r},\ \mathbf{A}^2\,\mathbf{r},\ \ldots,\ \mathbf{A}^{k-1}\,\mathbf{r})$$

and compute an eigendecomposition of the resulting matrix.

Advantages:

- Very simple access to $\mathbf{A}$.
- Extremely high accuracy possible. (Double precision accuracy for "converged" eigenmodes, etc.)

Drawbacks:

- The matrix is typically revisited $O(k)$ times if a rank-k approximation is sought. (Blocked versions exist, but the convergence analysis is less developed.)
- Numerical stability issues. These are well-studied and can be overcome, but they make software less portable (between applications, hardware platforms, etc.).

"New" challenges in algorithmic design:

The existing state-of-the-art methods of numerical linear algebra that we have very briefly outlined were designed for an environment where the matrix fits in RAM and the key to performance was to minimize the number of *floating point operations* required.

Currently, *communication* is becoming the real bottleneck:

- While clock speed is hardly improving at all anymore, the cost of a flop keeps going down rapidly. (Multi-core processors, GPUs, cloud computing, etc.)

- The cost of slow storage (hard drives, flash memory, etc.) is also going down rapidly.

- Communication costs are decreasing, but *not* rapidly.
  - Moving data from a hard-drive.
  - Moving data between nodes of a parallel machine. (Or cloud computer ...)
  - The amount of fast cache memory close to a processor is not improving much. (In fact, it could be said to be *shrinking* — GPUs, multi-core, etc.)

- "Deluge of data". Driven by ever cheaper storage and acquisition techniques. Web search, data mining in archives of documents or photos, hyper-spectral imagery, social networks, gene arrays, proteomics data, sensor networks, financial transactions, ...

The more powerful computing machinery becomes,
the more important efficient algorithm design becomes.

- Linear scaling (w.r.t. problem size, processors, etc.).
- Minimal data movement.
- In linear algebra, we will see that *blocking* of numerical methods will be essential. "BLAS3 is far faster than BLAS2." Consider two options for computing a product $\mathbf{b} = \mathbf{A}\mathbf{x}$ in Matlab where $\mathbf{A}$ is an $m \times n$ matrix, and $\mathbf{x}$ is a vector.

  Option 1:

  ```
  b = A*x
  ```

  Option 2:

  ```
  b=zeros(m,1); for i=1:n; b=b+A(:,i)*x(i); end;
  ```

Mathematically equivalent. The same number of flops. But Option 1 is far faster.

Exercise: Compare the speeds of column pivoted QR and unpivoted QR factorization.
Almost the same flop count; very different practical speed.

Brief review of some key facts of the SVD:

Let $\mathbf{A}$ be an $m \times n$ matrix. Then $\mathbf{A}$ admits a *singular value decomposition (SVD)*

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^*, \\ m \times n & & m \times r & r \times r & r \times n \end{array}$$

where $r = \min(m, n)$ and where

$\mathbf{U} = [\mathbf{u}_1\ \mathbf{u}_2\ \cdots\ \mathbf{u}_r]$ is a matrix holding the "left singular vectors" $\mathbf{u}_i$,

$\mathbf{V} = [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_r]$ is a matrix holding the "right singular vectors" $\mathbf{v}_i$,

$\mathbf{D} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$ is a diagonal matrix holding the "singular values" σ_i.

For any k such that $1 \leq k \leq \min(m, n)$, we define the *truncated* SVD as

$$\mathbf{A}_k = \mathbf{U}(:, 1:k)\,\mathbf{D}(1:k, 1:k)\,\mathbf{V}(:, 1:k)^* = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^*.$$

The following theorem states that $\mathbf{A}_k$ is the "optimal" rank-k approximation to $\mathbf{A}$:

Theorem (Eckart-Young): Let $\|\cdot\|$ denote either the Frobenius or spectral norm. Then

$$\|\mathbf{A} - \mathbf{A}_k\| = \min\{\|\mathbf{A} - \mathbf{B}\| : \ \mathbf{B} \text{ is of rank } k\}.$$

Moreover,

$\|\mathbf{A} - \mathbf{A}_k\| = \sigma_{k+1}$, when the spectral norm is used,

$\|\mathbf{A} - \mathbf{A}_k\| = \sqrt{\sigma_{k+1}^2 + \sigma_{k+2}^2 + \cdots + \sigma_r^2}$, when the Frobenius norm is used.

Brief review of some key facts of the SVD: Recall the SVD

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^* \\ m \times n & & m \times r & r \times r & r \times n \end{array} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^*.$$

where $r = \min(m, n)$. Some facts:

- The left singular vectors $\{\mathbf{u}_j\}_{j=1}^k$ form an optimal basis for the column space of $\mathbf{A}$ in the sense that

  $\|\mathbf{A} - \mathbf{U}(:, 1:k)\mathbf{U}(:, 1:k)^*\mathbf{A}\| = \inf\{\|\mathbf{A} - \mathbf{P}\mathbf{A}\| : \text{ where } \mathbf{P} \text{ is an ON proj. to a } k-\text{dimensiona}$

- The right singular vectors $\{\mathbf{v}_j\}_{j=1}^k$ form an optimal basis for the row space of $\mathbf{A}$.
- For a *symmetric* matrix, the eigenvalue decomposition (EVD) and the singular value decomposition are in many ways equivalent, and a truncated EVD is also an optimal rank-k approximation.
- The EVD and the SVD are also in many ways equivalent for a *normal* matrix (recall that $\mathbf{A}$ is normal if $\mathbf{A}\mathbf{A}^* = \mathbf{A}^*\mathbf{A}$), but the EVD might be complex even when $\mathbf{A}$ is real.
- For *non-normal* matrices, eigenvectors and eigenvalues are generally not convenient tools for low rank approximation.
- For a general matrix, the SVD provides the EVDs of $\mathbf{A}^*\mathbf{A}$ and $\mathbf{A}\mathbf{A}^*$:

$$\mathbf{A}\mathbf{A}^* = \mathbf{U}\mathbf{D}^2\mathbf{U}^*, \qquad \text{and} \qquad \mathbf{A}^*\mathbf{A} = \mathbf{V}\mathbf{D}^2\mathbf{V}^*.$$

Brief review of some key facts of the SVD: Recall the SVD

$$\begin{array}{cccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^* & = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^*. \\ m \times n & & m \times r & r \times r & r \times n & \end{array}$$

where $r = \min(m, n)$.

How do you compute an SVD?

Brief review of some key facts of the SVD: Recall the SVD

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^* \\ m \times n & & m \times r & r \times r & r \times n \end{array} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^*.$$

where $r = \min(m, n)$.

How do you compute an SVD?

- Any method must be iterative. (Why?)
- In practice, the iterations essentially always converge rapidly, so observed behavior appears deterministic, with complexity $\sim c\, n^3$ for an $n \times n$ matrix.
- Scaling constant is quite large compared to other standard factorizations.
- Generally hard to block, parallelize, etc. (Even the first step — reduction to bidiagonal form — presents challenges.)