CS Colloqium

Finding Structure with Randomness — March 28, 2016

Lecturer: Joel A. Tropp — Scribe: Kiran Vodrahalli

Contents

1 Introduction

I'm with the department of computing and mathematics at CalTech: Computing, Mathematics, and Control and Dynamics. See jtropp@cms.caltech.edu, http://users.cms.caltech.edu/ jtropp. You can see this talk in a survey paper from 2011.

I'm going to start out by motivating the topic: Matrix Decompositions. I like to explain this with a notorious list of the top ten algorithms of all time. When you look at this list, you see familiar things: The Simplex Algorithm, the Fast Fourier Transform, Quick Sort. Technically this is a list of algorithms of the second half of the 20th century, and some of these don't really pass that test, like the FFT which was invented by Gauss. The "Decompositional Approach to Matrix Computations" doesn't really look like an algorithm though.

The Decompositional Approach section says, it's not my problem! It's my job to give you tools to solve your problem. The tool is matrix decomposition. Decomposition is useful because they allow you to solve many problems with the same framework. For instance, if you want to solve a linear system, you might take an LU decomposition of the matrix. This lets you solve a lot of linear systems at low cost. You might also use it to compute the determinant. One thing I would like to highlight is these decompositions lead to black-box software, which is a good tool. Once you have it, you can do lots of great things. This approach actually has its roots here with von Neumann. He invented the LU decomposition, performed rounding-error analysis, and did a very complete study of this method. Only several years later did they build a computer to try it out.

What's wrong with classical approaches to matrix decomposition? Nothing, provided the matrices aren't too big (perhaps on the order of 100s of megabytes). But once you get into large examples, things stop working so well, since these things scale cubically. We also have a lot of new computational architecture, which people didn't envision in the 50s-80s. There's a lot more effort in distributed computing. Data transfer dominates the cost of the algorithms, not the arithmetic which is what they were optimized for in the first place. For instance, pivoted QR decomposition requires you to have random access to the columns, which is hopless if the columns are stored in different places.

So I'm going to talk to randomized approximations which are much more scalable. These are simple and robust. Classical algorithms are fidgety and don't work well in all these settings.

The reason these are useful are multiplication rich. The most difficult operation is matrix multiplication, which we are very good at. As a result, you can get near peak throughput on matrix-matrix multiplications. Thus, algorithms built out of these things can use the computational platform a lot better rather than algorithms built out of other primitives.

2 Two-stage Low-rank Approximations

The reason we consider low-rank decompositions is that they're one of the most important ways of approximating a matrix. Given a large matrix $A \in \mathbb{R}^{m\times n}$, let $r << m, n$. This is the rank of the approximation. If we can approximate $A \approx BC$, $\text{rank}(BC) \leq r$. There are huge savings in the number of degrees of freedom. If you succeed in this approximation, you have necessarily identified structure in the matrix. That's the only way you could have reduced the number of degrees of freedom this dramatically. This plays a huge role in a lot of areas. This is a simpler model for truncated SVD and so on. Once you have such a thing, you can do a lot of other approximations which are useful or important. You can compute leading singular vectors, spanning sets of rows and columns, leading eigenvectors, etc.

I will focus on finding $\|A - BC\|_2 \leq \text{tol}$ in the spectral norm (2-operator norm) since it nails down all these properties. Frobenius norm bounds are easier to obtain, but useless in a lot of settings.

Example 2.1. Truncated SVD. We want

$$A = U\Sigma V^*$$

where $U, V*$ are orthonormal, and Σ is $r \times r$ diagonal matrix. If you can approximate A this way, you have necessarily found structure.

Let us construct a two-stage randomized SVD algorithm. The first goal when computing a matrix approximation is to find a matrix Q with orthonormal columns where the number of columns is relatively small. We want this to capture most of the action of the matrix A. We want the low-dimensional range subspace to be well aligned with the subspace where A is busiest (the dominant left-singular vectors of A). So let's say we use a **randomized algorithm** to get $A \approx QQ^*A$ where Q has orthonormal columns. Even if we through away everything in A that's in the kernel of Q, we don't lose much.

We already have a low-rank approximation of A if we solve this problem, since A is fat and Q is tall: $B = Q, C = Q^*A$ (using the notation from before).

Now we will form the decomposition by using basis Q to reduce the problem size, apply classical linear algebra to the reduced problem, and then finally obtain truncated SVD in factorized form.

First, we have to find the range with a target rank. We want to find $m \times r$ matrix Q with orthonormal columns such that

$$\|A - QQ^*A\|_2 \approx \min_{\text{rank}(B)=r_0} \|A - B\|_2$$

We will use a randomized algorithm which is efficient to find Q: One matrix-matrix multiplication ($\mathcal{O}(m \times n \times r)$) and then $\mathcal{O}(r^2 n)$ additional work.

We want to do a randomized range finder. The intuition is to pull out a random vector ω_1, mtuliply that into A: $A\omega_1$ should be oriented . Now we do it again: $A\omega_2$, which is in the range of A, but probably not pointing in the same direction. Do this k times, now we have k vectors in the range of A, which tend to be aligned along the major axes of the ellipsoid. Because they're random, with probability 1 they have no linear dependencies, and their images also have no linear dependencies, so then you get a basis. Now all you have to do is orthgonalize them. Now we can get a basis when A is rank-deficient. If it's not rank deficient, we may have to add some more vectors because of perturbations outside of the low-rank plane. Thus, the algorithm is, given $m \times n$ matrix A and a number r of samples.

1. Draw $n \times r$ a random matrix Ω.

2. Form $Y = A\Omega$ (the range image).

3. Find orthonormal basis Q for the range of Y.

This is the first step of subspace iteration with a random start. What is different is that I'm asserting this is not an iterative algorithm. I'll justify why you can stop after only 3 steps. If you're interested in the history you can check out my survey paper.

For implementation issues, we can pick r samples adaptively with a randomized error estimator. It's hard to figure out what the numerical rank of a matrix is. So how does r compare with target rank (how many extra samples do you need?) You usually only need 5 or 10 extra samples. How do we pick Ω? Standard Gaussian works fine. To compute orthogonal basis, you can use Gram-Schmidt or Householder reflectors.

Theorem 2.2. *Error Bound for Random Range Finder.*
The matrix A is $m \times n$ with $m \geq n$, target rank is r_0, $\sigma_{r_0+1} = \min_{\text{rank}(B)=r_0} \|A - B\|$, and Ω is an $n \times (r_0 + s)$ standard Gaussian. Then, the randomized range finder algorithm yields an $(r_0 + s)$-dimensional orthobasis Q with

$$\boldsymbol{E}[\|A - QQ^*A\|] \leq \left[1 + \frac{4\sqrt{r_0 + s}}{s - 1}\sqrt{n}\right] \sigma_{r_0+1}$$

If σ_{r_0+1} is small, you'll do well. Is $\sqrt{n}$ thought of as small? Yes and No, depending on the application. In more detail:

$$\boldsymbol{E}[\|A - QQ^*A\|] \leq \left[1 + \sqrt{\frac{r_0}{s - 1}}\right] \sigma_{r_0+1} + \frac{e\sqrt{r_0 + s}}{s} \left(\sum_{j>r_0} \sigma_j^2\right)^{1/2}$$

This bound is good only if the spectrum decays somewhat. This bound is also somewhat optimal. It basically depends on what part of the spectrum that you miss. This is not so good for getting minimal errors. Also, this concentrates extremely well; this is typical behavior (it's just messy to display). The probability of a substantially larger error is negligible.

How do we do adaptive error estimation? In practice, you'll estimate the error as you go along, and stop when we reach a target. We form the error estimator:

$$\text{errest} = \max_{1 \leq j \leq 10} \|(I - QQ^*)A\omega_j\|$$

The probability of being worse than this error estimator is

$$\mathbf{P}\{\|(I - QQ^*)A\| \geq 10 \cdot \text{errest}\} \leq 10^{-10}$$

Once we've found Q, we get $B = Q^*A$, which is "fat". Now since B is small, we can use classical linear algebra to compute an SVD of $B = Q^*A$. So now we can write

$$\begin{aligned} A \approx QQ^*A &= Q(Q^*A) \\ &= Q(U\Sigma V^*) \\ &= (QU)\Sigma V^* \end{aligned} \tag{1}$$

Thus we can write $A \approx (QU)\Sigma V^*$ to get an approximate SVD with lower bounds on how good it is. The main cost in the algorithm is $Q^* \times A$. The cost of the SVD is $r^2 n$, which is cheap.

If it's a sparse matrix, we need $m \times n \times 1$ vector multiplies in sequence. Performing multiplies in sequence is onerous, and these methods are fragile, so there are lot of limitations to this approach. There are also classical methods for matrices which require random access, which makes it less easy to implement in the large-scale setting. Even though we haven't saved work, we've organized the work in a much more efficient fashion which eliminates methods of classical algorithms.

3 The Role of Spectral Decay

The random range finder works when the spectrum of A decays quickly. The problem is this behavior is not common in data analysis problems. Examples: Matrix is contaminated with noise: $A = A_0 + N$ with $\|N\| \sim \|A_0\|$. The matrix spectrum also decays slowly: The tail is very big. If so, the randomized range finder gives unreliable results.

The remedy is to use subspace iteration. However we view this as a terminating finite algorithm which ends after a small number of steps. We apply the same algorithm to $(AA^*)^q A$ for a small q, and you just get an extra factor of qr^2n, and $2q+1$ matrix multiplies. So we should be concerned about what q is. We argue that it's very small, i.e. $q = 2, 3$. This is very different from the classical analysis of subspace iteration.

The error bound changes to

$$\mathbf{E}[\|A - QQ^*A\|] \leq \left[1 + \frac{4\sqrt{r_0 + s}}{s-1}\sqrt{n}\right]^{1/(2q+1)} \sigma_{r_0+1}$$

which suggests that even small powers are effective.

Example 3.1. Graph Laplacian.
This is an image processing example. We look at which 9×9 image patches look alike, and there are 9000 patches, which gives us a 9000×9000 sparse symmetric matrix, which is several hundred megabytes. We want to compute eigenvalues and eigenvectors of this moderately large matrix to learn something about the images.

We see that the low-rank approximation does very well, and you only need $q = 3$ to get very close to the optimal, calculated via classical methods.

Example 3.2. Eigenfaces.
Here we have dense 10^5 pixel photos to compute eigenfaces. Unless you're very careful, Matlab will choke or you'll get a bad estimate. Our method works (the power method is necessary since the spectrum doesn't decay naturally). You also get very good estimate for the singular values as well.