Stat260/CS294: Randomized Algorithms for Matrices and Data

Lecture 11 - 10/09/2013

Lecture 11: Randomized Least-squares Approximation in Practice

Lecturer: Michael Mahoney *Scribe: Michael Mahoney*

Warning: these notes are still very rough. They provide more details on what we discussed in class, but there may still be some errors, incomplete/imprecise statements, etc. in them.

11 Randomized Least-squares Approximation in Practice

During this class and the next few classes, we will describe how the RandNLA theory from the last few classes can be implemented to obtain high-quality implementations in practice. Here is the reading for today and the next few classes.

- Avron, Maymounkov, and Toledo, "Blendenpik: Supercharging LAPACK's Least-Squares Solver"
- Avron, Ng, and Toledo, "Using Perturbed QR Factorizations to Solve Linear Least-Squares Problems"

Today, in particular, we will do three things.

- We will provide an overview of some of the implementation challenges.
- We will go into more detail on forward error versus backward error questions.
- We will go into more detail on preconditioning and ϵ-dependence issues.

11.1 Overview of implementation challenges

There are several issues that must be dealt with to implement these RandNLA ideas in practice. By this, we mean that we will want to implement these algorithms as is done in NLA, e.g., in LAPACK. Different issues arise when we implement these algorithms in a data center or a database or a distributed environment or a parallel shared memory environment; and different issues arise when we implement and apply these algorithms in machine learning and data analysis applications. We will touch on the latter two use cases to some extent, but our main focus here will be on issues that arise in providing high-quality implementations on moderately large problems to solve problems to (say) machine precision on a single machine.

Here are the main issues.

- **Forward versus backward error.** TCS worst-case bounds deal with forward error bounds in one step, but in NLA this is done via a two step process, where one considers the posedness of a problem and then the stability of an algorithm for that problem. This two step approach is less general than the usual one step approach in TCS that makes forward error claims on the objective function, but it leads to finer bounds when it works.

- **Condition number issues.** So far, we have said almost nothing about condition number issues, except to note that a condition number factor entered depending on whether we were interested in relative error on the objective versus certificate, but they are very important in finite precision arithmetic and in practical implementations.
- **Dependence on ϵ parameter.** A dependence of $1/\epsilon$ or $1/\epsilon^2$ is natural in TCS and is fine if we view ϵ as fixed and not too large, e.g., 0.1 (and the $1/\epsilon^2$ is a natural bottleneck for Monte Carlo methods due to law of large numbers considerations), but this is actually exponential in the number of accuracy bits (since a number of size n can be represented in roughly $\log(n)$ bits) and it is a serious problem if we want to obtain high precision, e.g., 10^{-10}.

Here are some additional issues, that arise especially when dealing with the third issue above.

- **Hybrid and preconditioned iterative methods.** Here, we go beyond simply doing random sampling and random projection where we call a black box on the subproblem.
- **Failure probability issues.** In some cases, it is preferable to have the failure probability enter into the running time that it takes to get a good solution and not whether or not the algorithm gets a good solution at all. This has been studied in TCS (under the name Las Vegas algorithms); and, fortunately, it meshes well with the condition number and iterative considerations mentioned above.
- **Downsampling more aggressively.** In code, it is difficult to sample $O\left(d\log(d)\right)$ rows, i.e., to loop from 1 to $O\left(d\log(d)\right)$, where the constant in the big-O is unspecified. Instead, one typically wants to loop from 1 to (say) $2d$ or $4d$. At that point, there are worst-case examples where the algorithm might fail, and thus one needs to deal with the situation where, e.g., there are a small number of rows with large leverage that are lost since we downsample more aggressively. Again, we will see that—while a serious problem for worst-case TCS-style analysis—this meshes well with iterative algorithms as used in NLA.

We will deal with each of these issues in turn.

11.2 Forward versus backward error.

Let's start with the following definition.

Definition 1 *A problem P is well-posed if: the solution exists; the solution is unique; and the solution depends continuously on the input data in some reasonable topology.*

We should note that this is sometimes called *well-conditioned* in NLA, but the concept is more general. (For example, P doesn't need to be a LS or low-rank matrix problem—it could be anything such as the MaxCut Problem or the k-Sat Problem or whatever.)

The point here is that, if we work with matrix problems with real-valued continuous variables, even for a well-posed or well-conditioned problem, certain algorithms that solve the problem "exactly," e.g., with infinite precision, perform poorly in the presence of "noise" introduced by truncation and roundoff errors.

This leads to the idea of the numerical stability *of an algorithm.* Let's consider an algorithm as a function f attempting to map input data X to output data Y; but, due to roundoff errors, random sampling, or whatever, the algorithm actually maps input data X to output data Y^*. That is, the algorithm "should" return Y, but it actually returns Y^*. In this case, we have the following.

- **Forward error.** This is $\Delta Y = Y - Y^*$, and so this is the difference between the exact/true answer and the answer that was output by the algorithm. (This is typically what we want to bound, although note that one might also be interested in some function of Y, e.g., the objective function value in an optimization, rather than that argmin.)
- **Backward error.** This is the smallest ΔX such that $f(X + \Delta X) = Y^*$, and so this tells us what input data the algorithm that we ran actually solved exactly.

In general, the forward error and backward error are related by a problem-specific complexity measure, often called the condition number as follows:

$$|\text{forward error}| \leq |\text{condition number}| \times |\text{backward error}| .$$

In particular, backward stable algorithms provide accurate solutions to well-conditioned problems.

TCS typically bounds the forward error directly in one step, while NLA bounds the forward error indirectly in two steps, i.e., by considering only well-posed problems and then bounding the backward error for those problems. That typically provides finer bounds, but it is less general, e.g., since it says nothing about ill posed problems.

In light of this discussion, observe that the bounds that we proved the a few classes ago bound the forward error in the TCS style. In particular, recall that the bounds on the objective are of the form

$$\left\|A\tilde{x}_{opt} - b\right\|_2 \leq (1+\epsilon) \left\|Ax_{opt} - b\right\|_2 , \tag{1}$$

and this implies that

$$\begin{aligned} \left\|\tilde{x}_{opt} - x_{opt}\right\|_2 &\leq \sqrt{1-\gamma^2}\kappa(A)\sqrt{\epsilon}\left\|x_{opt}\right\|_2 \\ &= \tan(\theta)\kappa(A)\sqrt{\epsilon}\left\|x_{opt}\right\|_2 , \end{aligned} \tag{2}$$

where $\theta = \cos^{-1}\left(\frac{\|Ax_{opt}\|_2}{\|b\|_2}\right)$ is the angle between the vector b and the column space of A.

This is very different than the usual stability analysis which is done in NLA which is done in terms of backward error as follows. Consider the approximate solution $\tilde{x}_{opt}$ (usually in NLA this is different than the exact solution in exact arithmetic, e.g., due to roundoff errors; but in RandNLA it is different, even in exact arithmetic, since we solve a random subproblem of the original problem), and consider the perturbed problem that it is the exact solution to. That is,

$$\tilde{x}_{opt} = \text{argmin} \left\|(A + \delta A)x + b\right\|_2 , \tag{3}$$

where $\|\delta A\|_\xi \leq \tilde{\epsilon}\|A\|_\xi$. (We could of course include a perturbed version of b in Equation (3).) By standard NLA methods, Equation (3) implies a bound on the forward error

$$\left\|\tilde{x}_{opt} - x_{opt}\right\|_2 \leq \left(\kappa(A) + \frac{\kappa^2(A)\tan(\theta)}{\eta}\right)\tilde{\epsilon}\left\|x_{opt}\right\|_2 , \tag{4}$$

where $\eta = \frac{\|A\|_2 \|x\|_2}{\|Ax\|_2}$. Importantly, Equation (2) and Equation (4), i.e., the two different forward error bounds on the vector or certificate achieving the optimal solution, are not obviously comparable.

There are some but very few results on the numerical stability of RandNLA algorithms, and this is a topic of interest, especially when solving the subproblem to achieve a low-precision solution. If RandNLA methods are used as preconditions for traditional algorithms on the original problem, then this is less of an issue, since they inherit the properties of the traditional iterative algorithms, plus something about the quality of the preconditioning (which is probably less of an issue).

11.3 Preconditioning, the dependence on ϵ, and related issues

There are two broad methodologies for solving sparse and dense linear systems: direct methods and iterative methods. The two classes of methods are complementary, and each comes with pros and cons, some of which are listed here.

- **Direct methods.** These methods typically involve factoring the coefficient matrix into the product of simpler matrices whose inverses are easier to apply. For example, $A = LU$ in general, $A = LL^T$ for SPSD matrices, and $A = QR$ or $A = U\Sigma V^T$ for overdetermined/rectangular problems. These are generic, robust, predictable, and efficient; but they can have limited scalability, they may run out of memory, they may be too slow for large and especially sparse matrices, etc.
- **Iterative methods.** These methods typically involve starting with an approximate solution and iteratively refining it, in the simplest case by doing matrix-vector products. These often scale much better to large and/or sparse problems; but they can be more fragile and can be slower than direct methods for many inputs.

The issue about iterative methods being slower depends on several factors, but there is usually some sort of problem-specific complexity measure, e.g., the condition number, that determines the number of iterations, and this can be large in worst-case.

A partial solution that can over this difficulty is to use something called a preconditioner and then work with preconditioned iterative methods. This comes at the expense of a loss of generality since a given preconditioner in general doesn't apply to every problem instance. Among other things, preconditioning opens the door to hybridization: we can use direct methods to construct a preconditioner for an iterative methods, e.g., use an incomplete decomposition to minimize fill-in and then apply an iterative method. This can lead to improved robustness and efficiency, while not sacrificing too much generality.

While perhaps not obvious from the discussion so far, randomization as used in RandNA can be combined with ideas like hybridization to obtain practical iterative LS solvers. Basically, this is since low-precision solvers, e.g., those obtained when using $\epsilon = 1/2$ (or worse), provide a pretty good solution that can be computed pretty quickly. In particular, this involves using a randomized algorithm to construct a preconditioned for use in a deterministic iterative solver.

Let's say a few words about why this is the case. Recall that Krylov subspace iterative methods for solving large systems of linear equations treat matrices as black boxes and only perform matrix-vector multiplications. Using this basic operation, they find an approximate solution inside the

Krylov subspace:

$$K_n(A,b) = \{b, Ab, A^2b, \ldots, A^{n-1}b\}.$$

For example: (1) CG, for SPSD matrices; (2) LSQR, for LS problems, which is like CG on the normal equations; and (3) GMRES, for general matrices.

While it's not the most common perspective, think of a preconditioned as a way to move between the extremes of direct and iterative solvers, taking advantage of the strengths of each. For example, if we have an SPSD matrix A and we want to do CG and use a preconditioner M, then instead of solving the original problem

$$Ax = b,$$

we might instead solve the preconditioned problem

$$M^{-1}Ax = M^{-1}b.$$

(Alternatively, for the LS problem, instead of solving the original problem

$$\min_x \|Ax - b\|_2,$$

we might instead solve the preconditioned problem

$$\min_y \left\|AR^{-1}y - b\right\|_2,$$

where $Rx = y$.)

For simplicity, let's go back to the CG situation (although the basic ideas extend to overdetermined LS problems, which will be our main focus, as well as to more general problems, which we won't discuss). There are two straw men to consider in preconditioning.

- If $M = A$, then then the solver is basically a direct solver. In particular, constructing the preconditioner requires solving the original problem exactly, and since this provides the solution there is no need to do any iteration.
- If $M = I$, then the solver is basically an unpreconditioned iterative solver. The "preconditioning phase" takes no time, but the iterative phase is no faster than without preconditioning.

The goal, then, is to find a "sweet spot" where M is not too expensive to compute and where it is not too hard to form its inverse. In particular, we want $\kappa\left(M^{-1}A\right)$ to be small and also that M^{-1} is easy to compute and apply, since then we have a good preconditioner. (The extension of this to the overdetermined LS problem is that AR^{-1} is quick to compute and that $\kappa\left(AR^{-1}\right)$ is small, where R is a matrix not-necessarily from QR.)

The way we will use this is that we will use the output of a low-precision random projection or random sampling process to construct a preconditioned. (In particular, we will compute a QR decomposition of ΠA, where Π is a random projection or random sampling matrix.) That is, rather than solving the subproblem on the projection/sampling sketch, we will use it to construct a preconditioner. If we obtain a very good sketch, e.g., one that provides a $1 \pm \epsilon$ relative error subspace embedding, then it will also provide a very good $1 \pm \epsilon$ preconditioner. That works; but, importantly, it is often overkill, since we can get away with lower quality preconditioners.

To understand how this works, we need to get into a few details about how to analyze the quality of preconditioners. The simplest story is that if the eigenvalue ratio, i.e., the condition number, of the problem is small, then we have a good preconditioner. And if we sample/project onto $O\left(d\log(d)\right)$ dimensions, then we will satisfy this. But we can also get good quality preconditioners with many fewer samples. To do this, we need to know a little more than just the eigenvalue ratio, since controlling that is sufficient but not quite necessary to have a good preconditioner, so let's get into that.

Again, for simplicity, let's consider the case for SPSD matrices. (It is simpler, and via LSQR, which is basically CG on the normal equations, most of ideas go through to the LS problem. We will point out the differences at the appropriate points.) Analyzing preconditioners for SPSD matrices is usually done in terms of generalized eigenvalues and generalized condition numbers. Here is the definition.

Definition 2 *Let $A, B \in \mathbb{R}^{n\times n}$, then $\lambda = \lambda\left(A,B\right)$ is a* finite generalized eigenvalue *of the matrix pencil/pair if there exists a vector $v \neq 0$ such that* $\begin{cases} Av = \lambda Bv \\ Bv \neq 0 \end{cases}$.

Given this generalized notion of eigenvalue, we can define the following generalized notion of condition number.

Definition 3 *Let $A, B \in \mathbb{R}^{n\times n}$ be two matrices with the same null space. Then the* generalized condition number *is*

$$\kappa\left(A,B\right) = \frac{\lambda_{max}\left(A,B\right)}{\lambda_{min}\left(A,B\right)}.$$

These notions are important since the behavior of preconditioned iterative methods is determined by the clustering of the generalized eigenvalues, and the number of iterations is proportional to the condition number.

- For CG, the convergence is in $O\left(\sqrt{\kappa\left(A,M\right)}\right)$ iterations.
- For LSQR, if A is preconditioned by a matrix R, then convergence is in $O\left(\sqrt{\kappa\left(A^TA, R^TR\right)}\right)$ iterations.

Next time, we'll discuss how these ideas can be coupled with the RandNLA algorithms we have been discussing.