

Notes for Ma221 Lecture 8, Mar 1, 2022

Dealing with (nearly) low-rank matrices

Motivation: Real data is often low-rank, so we want to both

- (1) take precautions to avoid getting inaccurate LS solutions,
- (2) take advantage of it to compress the data, go faster, both deterministically and using randomization

I will use (1) as an application of compression, but compressing large data sets comes up in other applications too.

Solving a Least Squares Problem when A is rank deficient

Thm: Let A be $m \times n$ with $m \geq n$, but rank $r < n$.

$$\text{Let } A = U * \text{Sigma} * V^T = [U_1, U_2, U_3] * \begin{bmatrix} \text{Sigma}_1 & 0 \\ 0 & \text{Sigma}_2 \\ 0 & 0 \end{bmatrix} * [V_1, V_2]^T$$

be the SVD of A where

U is $m \times m$, U_1 is $m \times r$, U_2 is $m \times n-r$, U_3 is $m \times m-n$

Sigma is $m \times n$, Sigma_1 is $r \times r$, Sigma_2 is $n-r \times n-r$ and exactly 0 (later: tiny)

V is $n \times n$, V_1 is $n \times r$, V_2 is $n \times n-r$

Then the set of vectors x minimizing $\|A*x-b\|_2$ can be written as

$$\{x = V_1 * \text{Sigma}_1^{-1} * U_1^T * b + V_2 * y_2, y_2 \text{ any vector of dimension } n-r\}$$

The unique vector minimizing $\|A*x-b\|_2$ and $\|x\|_2$ is

$$x = V_1 * \text{Sigma}_1^{-1} * U_1^T * b, \text{ i.e. the choice } y_2 = 0$$

Def: $A^+ = V_1 * \text{Sigma}_1^{-1} * U_1^T$ is the Moore-Penrose pseudo-inverse of the rank-r matrix A (includes the full rank case $n=r$). (In practice, Sigma_2 are all singular values less than some use-determined threshold tol.)

So square or not, full rank or not, "best" solution is $\text{argmin}_x \|A*x-b\|_2 = A^+ * b$

Proof: $\|A*x-b\|_2$

$$= \|U * \text{Sigma} * V^T * x - b\|_2$$

$$= \|\text{Sigma} * V^T * x - U^T * b\|_2 \text{ since U is orthogonal}$$

$$= \|\text{Sigma} * y - U^T * b\|_2 \text{ where } y = V^T * x \text{ and } x \text{ have the same 2-norm, since V is orthogonal, so finding the LS solution minimizing } \|y\|_2 \text{ also minimizes } \|x\|_2$$

$$= \left\| \begin{bmatrix} \text{Sigma}_1 * y_1 - U_1^T * b \\ - U_2^T * b \\ - U_3^T * b \end{bmatrix} \right\|_2$$

$$\left\| \begin{bmatrix} - U_2^T * b \\ - U_3^T * b \end{bmatrix} \right\|_2$$

where $y = [y_1; y_2]$, y_1 is $r \times 1$, and y_2 is $n-r \times 1$

This is obviously minimized by choosing $y_1 = \text{Sigma}_1^{-1} * U_1^T * b$.

And $\|y\|_2^2 = \|y_1\|_2^2 + \|y_2\|_2^2$ is minimized by $y_2 = 0$.

So $x = V * y = [V_1, V_2] * [y_1; y_2] = V_1 * y_1 + V_2 * y_2 = V_1 * \text{Sigma}_1^{-1} * U_1^T * b$.

(We pause the recorded lecture here.)

Solving a Least Squares Problem when A is (nearly) rank deficient,
with the truncated SVD

We have defined the condition number of a matrix as $\sigma_{\max} / \sigma_{\min}$, where $\sigma_{\min} = 0$ for a rank deficient matrix, so the condition number is formally infinite. To illustrate this, compare the (minimum norm) least square solutions of

$$\operatorname{argmin} \left\| \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\|_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and

$$\operatorname{argmin} \left\| \begin{bmatrix} 1 & 0 \\ 0 & e \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\|_2 = \begin{bmatrix} 1 \\ 1/e \end{bmatrix}$$

which are arbitrarily different as e approaches 0. Given this sensitivity, it is natural to ask what sense it can make to solve a rank deficient least squares problem, when the solution can change discontinuously? In particular, we usually only know A to within some tolerance tol , i.e. the true A' could satisfy $\|A - A'\| \leq \text{tol}$. When A is nearly singular, what should we do?

Def: Given A and some tolerance tol bounding the uncertainty in A , the "truncated SVD" of A is defined as $A(\text{tol}) = U * \text{Sigma}(\text{tol}) * V^T$, where $A = U * \text{Sigma} * V^T$ is the usual SVD, and $\text{Sigma}(\text{tol}) = \text{diag}(\sigma_i(\text{tol}))$, where

$$\sigma_i(\text{tol}) = \begin{cases} \sigma_i & \text{if } \sigma_i \geq \text{tol} \\ 0 & \text{if } \sigma_i < \text{tol} \end{cases}$$

In other words, we "truncate" all the singular values smaller than tol to zero. Or, we replace A by the lowest rank matrix $A(\text{tol})$ within a distance tol (measured by the 2-norm) of A .

Using the truncated SVD effectively replaces the usual condition number $\sigma_{\max}(A)/\sigma_{\min}(A)$ by $\sigma_{\max}(A)/\text{tol}$, which can be much smaller, depending on the choice of tol . In other words, tol is a "knob" one can turn to trade off conditioning with how closely $A*x$ can approximate b (raising tol , and so lowering the rank of A , decreases the dimension of the space that can be approximated by $A*x$). Replacing A by an "easier" matrix is also called "regularization" (using the truncated SVD is just one of several mechanisms). The following Lemma illustrates this in a special case, where just vector b changes. The general case depends on whether one picks tol in a "gap" between singular values, since otherwise a small perturbation could change the number of singular values $> \text{tol}$.

Lemma: The difference between

$x_1 = \operatorname{argmin}_x \|A(\operatorname{tol})x - b_1\|_2$ and $x_2 = \operatorname{argmin}_x \|A(\operatorname{tol})x - b_2\|_2$
 (where we choose the solution of smallest norm in both cases)
 is bounded by $\|b_1 - b_2\|_2 / \operatorname{tol}$.

So choosing a larger tol limits the sensitivity of the solution.

Proof: Let $A = U \Sigma V^T$ and $A(\operatorname{tol}) = U \Sigma(\operatorname{tol}) V^T$ as above. Then

$$\begin{aligned} \|x_1 - x_2\|_2 &= \|(A(\operatorname{tol})^{-1}(b_1 - b_2))\|_2 = \|V^*(\Sigma(\operatorname{tol})^{-1})^* U^T (b_1 - b_2)\|_2 \\ &= \|\operatorname{diag}(1/\sigma(1), 1/\sigma(2), \dots, 1/\sigma(k), 0 \dots 0) U^T (b_1 - b_2)\|_2 \\ &\quad \text{where } \sigma(k) \geq \operatorname{tol} \\ &\leq (1/\sigma(k)) * \|U^T (b_1 - b_2)\|_2 \\ &\leq (1/\operatorname{tol}) * \|b_1 - b_2\|_2 \end{aligned}$$

Setting small singular values to zero also compresses the matrix, since it costs just $m \cdot k + k + k \cdot n$ numbers to store, as opposed to $m \cdot n$. So tol is also a "knob" that trades off compression with approximation accuracy.

The SVD is the most precise and most accurate way to approximate a matrix by one of lower rank, costing $O(mn^2)$, with a big constant. Now we explore other cheaper ones, both deterministic and randomized.

Solving a Least Squares Problem when A is (nearly) rank deficient, with (Tikhonov) regularization, aka ridge regression

Another common way to deal with the drawback of the solution becoming unbounded as the smallest singular values of A approach 0, is regularization, or computing

$$\begin{aligned} x &= \operatorname{argmin}_x \operatorname{norm}(A^*x - b, 2)^2 + \lambda^2 \operatorname{norm}(x, 2)^2 \\ &= \operatorname{argmin}_x \operatorname{norm}([A; \lambda I]^* x - [b; 0], 2)^2 \end{aligned}$$

where I is an n -by- n identity matrix, and where λ is a "tuning parameter" to be chosen by the user. The larger λ is chosen, the more weight is placed on $\operatorname{norm}(x, 2)$, so the more important it is to minimize relative to $\operatorname{norm}(A^*x - b, 2)$.

From the second line above, we can easily write down the normal equations:

$$\begin{aligned} x &= ([A; \lambda I]^T)^{-1} [A; \lambda I]^T [b; 0] \\ &= (A^T A + \lambda^2 I)^{-1} A^T b \end{aligned}$$

Adding λ^2 to the diagonal of $A^T A$ before Cholesky just increases all the eigenvalues of $A^T A$ by λ^2 , pushing it farther away from being singular.

If $A = U \Sigma V^T$ is the thin SVD of A , then substituting this for A yields

$$\begin{aligned} x &= V * (\Sigma * \operatorname{inv}(\Sigma^2 + \lambda^2 I)) * U^T * b \\ &= V * \operatorname{diag}(\sigma_i / (\sigma_i^2 + \lambda^2)) * U^T * b \end{aligned}$$

which reduces to the usual solution $x = V * \operatorname{inv}(\Sigma) * U^T * b$ when $\lambda = 0$. Note that when $\sigma_i \gg \lambda$, the above expression is close to $1 / \sigma_i$ as expected, and when $\sigma_i < \lambda$, it is bounded by σ_i / λ^2 , so it can't get larger than $1/\lambda$. Thus λ and tol in $A(\operatorname{tol})$ play similar roles.

(We pause the recorded lecture here.)

Solving a Least Squares Problem when A is (nearly) rank deficient, with QR

Our next alternative to the truncated SVD is QR with column pivoting:

Suppose we did $A = QR$ exactly, with A of rank $r < n$; what would R look like?

If the leading r columns of A were full rank (true for "most" such A), then

$R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$ with R11 $r \times r$ and full rank, so $R22 = 0$.

$$\begin{bmatrix} 0 & R22 \end{bmatrix}$$

If A is nearly low-rank, we can hope that $\|R22\| < \text{tol}$, and set R22 to zero.

Assuming that this works for a moment, write

$A = QR = [Q, Q'] \begin{bmatrix} R \\ 0 \end{bmatrix}$ with $[Q, Q'] = [Q1, Q2, Q']$ square and orthogonal as before,

$$\begin{bmatrix} 0 \end{bmatrix}$$

with Q1 $m \times r$, Q2 $m \times (n-r)$ and Q' $m \times (m-n)$. Thus

$$\text{argmin}_x \|Ax - b\|_2 = \text{argmin}_x \|[Q1, Q2, Q'] \begin{bmatrix} R \\ 0 \end{bmatrix} x - b\|_2$$

$$\begin{bmatrix} 0 \end{bmatrix}$$

$$= \text{argmin}_x \|[R11 \ R12] \begin{bmatrix} x1 \\ x2 \end{bmatrix} - [Q1^T b]\|$$

$$\|[0 \ 0] \begin{bmatrix} x1 \\ x2 \end{bmatrix} - [Q2^T b]\|$$

$$\|[0 \ 0] \begin{bmatrix} x1 \\ x2 \end{bmatrix} - [Q'^T b]\|_2$$

$$= \text{argmin}_x \|[R11 \cdot x1 + R12 \cdot x2 - Q1^T b]\|$$

$$\|[- Q2^T b]\|$$

$$\|[- Q'^T b]\|_2$$

with solution $x1 = \text{inv}(R11) \cdot Q1^T b - \text{inv}(R11) \cdot R12 \cdot x2$ for any $x2$.

But how do we pick $x2$ to minimize $\|x\|_2$?

Ex: $A = \begin{bmatrix} e & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$, $b = \begin{bmatrix} b1 \\ b2 \end{bmatrix}$, so we get $x = \begin{bmatrix} (b1 - x2)/e \\ x2 \end{bmatrix}$

$$\begin{bmatrix} 0 & 0 \\ 0 & R22 \end{bmatrix} \begin{bmatrix} b1 \\ b2 \end{bmatrix} \quad \begin{bmatrix} x2 \end{bmatrix}$$

and so if e is tiny, we better pick $x2$ carefully (close to $b1$) to keep $\text{norm}(x)$ small.

But if we permute the columns of A to A^*P and minimize $\|A^*P \cdot \hat{x} - b\|_2$ we get

$A^*P = \begin{bmatrix} 1 & e \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$ so we get $\hat{x} = \begin{bmatrix} b1 - e \cdot \hat{x}2 \\ \hat{x}2 \end{bmatrix}$

$$\begin{bmatrix} 0 & 0 \\ 0 & R22 \end{bmatrix} \begin{bmatrix} b1 - e \cdot \hat{x}2 \\ \hat{x}2 \end{bmatrix}$$

so $\text{norm}(x)$ is much less sensitive to the choice of $\hat{x}2$.

What would a "perfect" R factor look like? We know the SVD gives us the best possible

answer, so comparing $\begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$ to $\begin{bmatrix} \text{Sigma}_1 & 0 \\ 0 & \text{Sigma}_2 \end{bmatrix}$ makes sense.

These observations motivate the following (informal) definition:

(Informal) Def: A Rank Revealing QR Factorization (RRQR) of A is $A^*P = Q^*R$ where

P is a permutation, Q orthogonal, $R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$ with R11 $k \times k$ where

$$\begin{bmatrix} 0 & R22 \end{bmatrix}$$

(1) R22 is "small", ideally $\text{sigma_max}(R22) = O(\text{sigma}_{(k+1)}(A))$, i.e. R22 "contains" the $n-k$ smallest singular values of A

(2) R11 is "large", ideally $\text{sigma_min}(R11)$ not much smaller than $\text{sigma}_k(A)$

If in addition to (1) and (2) we have

(3) $\text{norm}(\text{inv}(R11) \cdot R12)$ is "not too large"

then we call $A^*P = Q^*R$ a "strong rank revealing QR"

(Informal) Thm: If (1), (2) and (3) hold, then

$$\begin{aligned} \sigma_i(A) &\geq \sigma_i(R11) \geq \sigma_i(A)/\sqrt{1+\text{norm}(\text{inv}(R11)*R12)^2} \text{ for } i=1:k \\ \sigma_i(A) &\leq \sigma_{\max}(R22)*\sqrt{1+\text{norm}(\text{inv}(R11)*R12)^2} \text{ for } i=k+1:n \end{aligned}$$

In other words, the singular values of R11 are good approximations of the largest k singular values of A, and the smallest n-k singular values of A are roughly bounded by norm(R22). Or, the leading k columns of A*P contain most of the information of the range space of A:

$$\begin{aligned} A*P=[A1,A2]=Q*R=[Q1,Q2]*[R11 \ R12] &\sim [Q1,Q2]*[R11 \ R12] = Q1*R11*[I,\text{inv}(R11)*R12] \\ &\quad [0 \ R22] \quad [0 \ 0] \\ &= A1*[I,\text{inv}(R11)*R12] \end{aligned}$$

But how do we compute the permutation P?

(We pause the recorded lecture here.)

Simplest: QR with column pivoting (QRCP). This is the oldest and simplest way, and often works, but like LU with partial pivoting, there are some rare matrices where it fails by a factor near 2^n . And it maximizes communication.

At the first step, the algorithm chooses the column of A of largest norm; at each following step, the algorithm picks the column with the biggest component orthogonal to all the columns already chosen.

Intuitively, this "greedy algorithm" picks the column with the biggest projection in a direction not spanned by previously chosen columns.

The algorithm is simply

```
for i=1:min(m-1,n) or until the trailing submatrix (R22) is small enough
    Choose largest column in trailing matrix, i.e. argmax_{j>=i} norm(A(i:m,j))
    if i neq j, swap columns i and j
    Multiply A(i:m,i:n) by a Householder reflection to zero out A(i+1:m,i)
endfor
```

If the algorithm stops after k steps, because the trailing matrix $A(k+1:m,k+1:n)$ has no column larger than some tolerance, the cost is about $4mnk$, versus $O(mn^2)$ for the SVD, so much cheaper if $k \sim \text{rank}(A) \ll n$.

To understand why this works: the first multiplication by a Householder reflection decomposes trailing columns into a part parallel to first column (now just parallel to e_1) and an orthogonal part (in rows 2:m). Choosing the column of biggest norm in rows 2:m chooses the column with the biggest component orthogonal to the first column. At each step we similarly choose the trailing column with the biggest component orthogonal to the previously chosen columns. The arithmetic cost is low, since we can just update $\text{norm}(A(i:m,j))$ from iteration to iteration for $O(m*n)$ cost, rather than recomputing them for $O(mn^2)$ cost.

But this simple algorithm "maximizes communication", since we read and write the entire trailing matrix $A(i:m,i:n)$ at each step. This is available in LAPACK's `geqpf` and Matlab's `[Q,R,P]=qr(A)`.

Here is a typical example, which shows how well each $R(i,i)$ approximates $\sigma(i)$:

```
A = randn(50)*diag(.5^(1:50))*randn(50);[Q,R,P]=qr(A);r=abs(diag(R));s=svd(A);
figure(1),semilogy(1:50,r,'r',1:50,s,'k'),title('red = diag(r), black = svd')
figure(2),plot(r./s),title('diag(r)/svd'), figure(3), spy(P)
```

As a (rare) example where QRCP fails to pivot well, consider $A = S^*C^*D$ where

```
cs = 1/sqrt(2); sn = 1/sqrt(2); n=30;
S = diag(sn.^(0:n-1)); C = eye(n)-cs*triu(ones(n,n),1); D = diag(.9999.^(0:n-1));
A = S*C*D; [Q,R,P]=qr(A);figure(2),spy(P);pause, r=abs(diag(R));s=svd(A);
figure(1),semilogy(1:n,r,'r+-',1:n,s,'k+-'),title('red = diag(R), black = svd')
figure(3),semilogy(1:n,r./s,'k+-'), title('diag(r)/svd')
```

i.e. C has ones on the diagonal and $-cs$ above the diagonal. sn and cs satisfy $sn^2+cs^2=1$. D is not necessary in exact arithmetic, but avoids roundoff problems.

A is upper triangular, and QRCP does not permute any columns, so $A = Q^*R = I^*A$.

and letting $k=n-1$ yields $R_{22} = sn^{(n-1)}$ but

$$\begin{aligned}\sigma_n(A) &= 1/\text{norm}(\text{inv}(A)) \sim 1/\text{norm}(\text{inv}(C)*\text{inv}(S)) \leq 1/(\text{norm}(\text{inv}(C)(:,n))*sn^{(1-n)}) \\ &= 1/(\text{norm}([cs*(1+cs)^{(n-2)}, cs*(1+cs)^{(n-3)}, \dots])*sn^{(1-n)}) \\ &< sn^{(n-1)}/[cs*(1+cs)^{(n-2)}]\end{aligned}$$

so $\sigma_n(A)$ can be smaller than R_{22} by an exponentially large factor

$cs*(1+cs)^{(n-2)}$, which can grow as fast as $2^{(n-2)}$ when $cs \sim 1$.

So QRCP has two weaknesses: (rare) failure to pivot well, and high communication cost. We address these in turn.

(We pause the recorded lecture here.)

Gu/Eisentat Strong RRQR Algorithm. This algorithm deals with the rare failure to pivot correctly. It uses a more complicated pivoting scheme, that depends on the norms of columns of R_{22} , rows of $\text{inv}(R_{11})$, and entries of $\text{inv}(R_{11})*R_{12}$, and guarantees a strong RRQR. It does so by cheaply exchanging a column in R_{11} and another column not in R_{11} if that increases $\det(R_{11})$ by a sufficient factor. It still costs only $4mnk$ (plus lower order terms), about the same as QRCP when $m \gg n$.

Avoiding communication in QR with column pivoting: Neither of the last two algorithms was designed with minimizing communication in mind, and so both access the entire matrix each time a column is chosen, and so the number of read/writes is also $O(mn^2)$, same as the number of flops, instead of the hoped for factor of $\sqrt{\text{fast_memory_size}}$ smaller.

The first attempt to fix this is in LAPACK's `geqp3.f`. This uses matrix multiply to update the trailing submatrix, as do LU and plain QR, but only reduces the number of reads/writes by 2x compared to the simple routine. Still, it is often faster.

But to approach the lower bound, we seem to need a different pivoting strategy, which can choose multiple pivot columns for each matrix access, not just 1.

The approach is similar to the TSLU algorithm described earlier.

We present the sequential version, which chooses b pivot columns with one pass through the matrix (the parallel version is analogous)

BestColumnsSoFar = (1:b) ... b is a blocksize to be chosen for accuracy/performance
for $k = b+1$ to $n-b+1$ step b ... assume $b \mid n$ for simplicity

form $m \times 2b$ matrix A_{2b} from columns in BestColumnsSoFar and columns $k:k+b-1$

... choose the b "best columns" among the $2b$ columns in A_{2b} , for example by:

factor $A_{2b} = Q^*R$, using TSQR

choose best b columns of R (just $2b \times 2b$), using RRQR or Strong RRQR,

update BestColumnsSoFar based on result

After each outer iteration, BestColumnsSoFar contains the indices of the b best columns found so far among columns 1 to $k+b-1$. The parallel version takes pairs of $m \times b$ submatrices, chooses the best b columns from each set of $2b$, and proceeds to pair these up and choose the best (which is why we call it "tournament pivoting" by analogy to having a "tournament" where at each round we choose the best). However, the flop count roughly doubles compared to QRCP.

(We pause the recorded lecture here.)

So far we have only considered low rank factorizations where (at least) one factor is an orthogonal matrix, say Q in QR . Q can be thought of as linear combinations of columns of A , which approximate the column space of A . But not all data analysis questions can best be answered by such linear combinations. Suppose the rows represent individual people, and the columns represent some of their characteristics, like age, height and income. If one of the columns of Q happens to be

$.2 * \text{age} - .3 * \text{height} + .1 * \text{income} + \dots$

then it can be hard to interpret what this means, as a "predictor" of the other columns/characteristics, like "been treated for disease X". And if there are thousands of columns, it is even harder. Instead, it would be good to be able to approximate the other columns by linear combinations of as few columns as possible, and analogously to approximate the other rows by a subset of the rows.

This leads to the following decomposition:

Def: A CUR decomposition of a matrix A is consists of the matrices

C is a subset of k columns of A

R is a subset of k rows of A

U is a $k \times k$ matrix

where $\text{norm}(A - C*U*R)$ is "small", i.e. close to the lower bound $\sigma_{(k+1)}$, which is attained by the SVD truncated to rank k .

A number of algorithms for computing a CUR decomposition have developed over time, see the class webpage for details. Here we highlight two, because they are easy to implement given the tools we have already presented:

- (1) Perform QR with some kind of column pivoting, to pick the k "most linearly independent" columns of A ; let $J = [j_1, j_2, \dots, j_k]$ be the indices of these columns, and let C consist of these columns of A .
- (2) Perform GEPP, or TSLU, on C , to pick the k most linearly independent rows of C ; let $I = [i_1, i_2, \dots, i_k]$ be the indices of these rows, and let R consist of these rows of A .

Having chosen C and R , we still need to choose U so that $C*U*R$ approximates A . Here are two approaches:

- (1) The best possible U is given by the solution to HW 3.12: the U that minimizes the Frobenius norm of $A - C*U*R$ is $U = \text{pinv}(C)*A*\text{pinv}(R)$, where $\text{pinv}(C)$ is the Moore-Penrose pseudo-inverse of C .
- (2) A cheaper approximation is just to choose U so that $C*U*R$ equals A in columns J and rows I . Since the k -by- k matrices $C(I,1:k) = R(1:k,J) = A(I,J)$ are equal, we just let U be the inverse of this common matrix.

(We pause the recorded lecture here.)

Now we consider randomized algorithms. Related reading is posted on the class webpage. The basic idea for many randomized algorithms is as follows:

Let Q be an $m \times k$ random orthogonal matrix, with $k \ll n$. Then we approximate A by $Q*(Q^T*A)$, the projection of A 's columns onto the space spanned by Q . Since Q^T*A has $k \ll n$ rows, solving the LS problem is much cheaper than with A .

Multiplying Q^T*A costs $2*m*n*k$ flops if done straightforwardly, which is only about 2x cheaper than QRCP above. So there has been significant work on finding structured or sparse Q , and not necessarily orthogonal, to make computing Q^T*A much cheaper. To date the best (and surprising result) says that you can solve a LS problem approximately, where A is sparse, with just $O(\text{nnz}(A))$ flops, where $\text{nnz}(A)$ is the number of nonzeros in A .

We give some motivation for why such a random projection should work by some low-dimensional examples, and then state the Johnson-Lindenstrauss Lemma, a main result in this area.

Suppose x is a vector in \mathbb{R}^2 , and q is a random unit vector in \mathbb{R}^2 , i.e.

$q = [\sin t; \cos t]$ where t is uniformly distributed on $[0, 2\pi)$. What is the distribution of $|x^T q|^2 = (\text{norm}(x) * |\cos(\text{angle}(x,q))|)^2$? It is easy to see that $\text{angle}(x,q)$ is also uniformly distributed on $[0, 2\pi)$, so the expected value $E(|x^T q|^2) = .5 * \text{norm}(x)^2$, and more importantly, the probability that $|x^T q|^2$ underestimates $\text{norm}(x)^2$ by a tiny factor ϵ , is $\text{Prob}(|\cos(t)|^2 < \epsilon) \sim 2 * \sqrt{\epsilon} / \pi$, so tiny too.

Now suppose x is a vector in \mathbb{R}^3 , and Q represents a random plane, i.e. Q is a random 3×2 orthogonal matrix, and $x^T Q$ is the projection of x onto the plane of Q . We again ask how well the size of the projection $\|x^T Q\|^2$ approximates $\|x\|^2$: Now the probability that $\|x^T Q\|^2 < \epsilon \|x\|^2$ is the same as the probability that x is nearly parallel to the perpendicular to the plane of Q , which can be thought of as a random point on the unit sphere in \mathbb{R}^3 . This probability is $O(\epsilon)$, much tinier, because x needs to be nearly orthogonal to both columns of Q .

Intuitively, as the number of columns of Q increases, the chance that $\|x^T Q\|^2$ greatly underestimates $\|x\|^2$ decreases rapidly. The Johnson-Lindenstrauss Lemma captures this, not just for one vector x , but for many vectors:

J-L Lemma: Let $0 < \epsilon < 1$, and x_1, \dots, x_n be any n vectors in \mathbb{R}^m , and $k \geq 8 \ln(n) / \epsilon^2$. Let F be a random $k \times m$ orthogonal matrix multiplied by $\sqrt{m/k}$. Then with probability at least $1/n$, for all $1 \leq i, j \leq n, i \neq j$,

$$1 - \epsilon \leq \frac{\|F(x_i - x_j)\|^2}{\|x_i - x_j\|^2} \leq 1 + \epsilon$$

The point is that for $F \cdot x$ to be an ϵ approximation of x in norm, the number of rows of F grows "slowly", proportional to $\ln(n) = \ln(\# \text{ vectors})$ and $1/\epsilon^2$. The probability $1/n$ seems small, but being positive it means that F exists (the original goal of Johnson and Lindenstrauss). And it comes from showing that the probability of a large error for any one vector $x_i - x_j$ is tiny, just $2/n^2$. This justifies using a single random F in the algorithms below.

(The proof follows by observing that we can think of F as fixed and each vector $x = x(i) - x(j)$ as random, and simply take F as the first k rows of the $m \times m$ identity, and each entry of x an i.i.d. (independent, identically distributed) Gaussian $N(0,1)$, i.e. with 0 mean and unit standard deviation, reducing the problem to reasoning about sums of squares of i.i.d. $N(0,1)$ variables. See the paper by Dasgupta and Gupta on the class webpage for details.)

(We pause the recorded lecture here.)

There is a range of different F matrices that can be used, which tradeoff cost and statistical guarantees, and are useful in different applications. The presentation below is based on the 2011 and 2020 survey articles by Tropp et al posted on the class web page, and other papers posted there.

One can construct a random orthogonal $m \times k$ matrix Q , with $m \geq k$, simply as follows: Let A be $m \times k$, with each entry i.i.d. $N(0,1)$, and factor $A = QR$ (and there are LAPACK routines for this). Then $F = \sqrt{m/k} \cdot Q^T$ in the J-L Lemma. But this costs $O(mk^2)$ to form F , which is too expensive in general.

In some applications, it is enough to let each entry of F be i.i.d. $N(0,1)$, without doing QR until later in the algorithm, we will see an example of this below. But multiplying F^*x still costs $O(mk)$ flops, when x is dense.

The next alternative is the subsampled randomized Fourier Transform (SRFFT), for which F^*x only costs $O(m \log(m))$ or even $O(m \log(k))$. (We will talk more about the FFT later in the semester.) In this case $F = R^*FFT^*D$ where

D is $m \times m$ diagonal with entries uniformly distributed on the unit circle in the complex plane

FFT is the $m \times m$ Fast Fourier Transform

R is $k \times m$, a random subset of k rows of the $m \times m$ identity

There are also variations in the real case where the FFT is replaced by an even cheaper real (scaled) orthogonal matrix, the Hadamard transform, all of whose entries are ± 1 , and D 's diagonal is also ± 1 ; this is called SRHT. The only randomness is in D and R . The intuition for why this works is that multiplying by D and then the FFT "mixes" the entries of x sufficiently randomly that sampling k of them (multiplying by R) is good enough.

Finally, when x is sparse, we would ideally like an algorithm whose cost only grew proportionally to $\text{nnz}(x)$. In the real case, we can take $F = S^*D$ where

D is $m \times m$ diagonal with entries randomly ± 1 (analogous to above)

S is $k \times m$, where each column is a randomly chosen column of the $k \times k$ identity

Note that S and R above are similar, but not the same. Multiplying $y = S^*D^*x$ can be interpreted as initializing $y=0$ and then adding or subtracting each entry of x to a randomly selected entry of y . If x is sparse, the cost is $\text{nnz}(x)$ additions or subtractions. This is called Randomized Sparse Embedding in the literature.

This F is not as "statistically strong" as the F 's described above, so we may need to choose a larger k to prove any approximation properties.

(We pause the recorded lecture here.)

Given these options for F , we give some examples of their use. We will give more examples in the context of iterative methods (Chap 6) later in the semester.

Consider the dense least squares problem $x_{\text{true}} = \text{argmin}_x ||A^*x - b||_2$ where A is $m \times n$. We approximate this by

$$x_{\text{approx}} = \text{argmin}_x ||F^*(A^*x - b)||_2 = \text{argmin}_x ||(F^*A)^*x - F^*b||_2.$$

Using results based on the J-L Lemma, we can take F to have $k = n \log(n) / \epsilon^2$ rows in order to get $||A^*x_{\text{approx}} - b||_2 \leq (1 + \epsilon) ||A^*x_{\text{true}} - b||_2$, in other words the residual is nearly as small as the true residual with high probability.

But there is no guarantee about how far apart x_{true} and x_{approx} are.

Now we consider the cost. Given a dense F , forming F^*A using dense matmul costs $O(kmn) = O(mn^2 \log(n) / \epsilon^2)$, so more than solving the original problem using QR, $O(mn^2)$. If we use SRFFT or SRHT and A is dense, forming F^*A costs just $O(mn \log(m))$.

Next, F^*A has dimension $k \times n$ and so solving using QR costs $O(kn^2) = O(n^3 \log(n)/\epsilon^2)$, for a total cost of $O(mn \log(m) + n^3 \log(n)/\epsilon^2)$, potentially much less than $O(mn^2)$ when $m \gg n$ and ϵ is not too small. In other words, if high accuracy is needed, a randomized algorithm like this may not help (we discuss randomized iterative algorithms later, which help address this).

Finally, we mention a least squares algorithm that costs just $O(\text{nnz}(A))$, plus lower order terms. For details also see the papers by Clarkson and Woodruff, and by Meng and Mahoney on the class webpage. This uses the F called a Randomized Sparse Embedding above, where we take $k = O((n/\epsilon)^2 \log^6(n/\epsilon))$. Then forming F^*A and F^*b costs $\text{nnz}(A)$ and $\text{nnz}(b)$, resp., much less than SRFFT when A is sparse. But note that k grows proportionally to n^2 , much faster than with the SRFFT, where k grows like n . If we solved $\arg\min_x \| (F^*A)^*x - F^*b \|_2$ using dense QR, that would cost $O(kn^2) = O(n^4 \log^6(n/\epsilon)/\epsilon^2)$, which is much larger than with SRFFT. So instead we use a randomized algorithm again (say SRFFT) to solve $\arg\min_x \| (F^*A)^*x - F^*b \|_2$.

Thm: With probability at least $2/3$,

$$\| A^*x_{\text{approx}} - b \|_2 \leq (1+\epsilon) * \| A^*x_{\text{true}} - b \|_2$$

How can we make the probability of getting a good answer much closer to 1? Just run the algorithm repeatedly: After s iterations, the probability that at least one iteration will have a small error rises to $1 - (1/3)^s$, and of the s answers x_1, \dots, x_s , the one that minimizes $\text{norm}(A^*x_i - b, 2)$ is obviously the best one.

(We pause the recorded lecture here.)

Next we consider the problem of using a randomized algorithm for computing a low rank factorization of the $m \times n$ matrix A , with $m \gg n$.

We assume we know the target rank k , but since we often don't know k accurately in practice, we will often choose a few more columns $k+p$ to see if a little larger (or smaller) k is more accurate.

This algorithm is of most interest when $k \ll n$, i.e. the matrix is low rank.

Note that in the following, the F matrices will be tall-and-skinny, and applied on the right (eg A^*F), so the transpose of the cases so far.

Randomized Low-Rank Factorization:

1. choose a random $n \times (k+p)$ matrix F
2. form $Y = A^*F$, which is $m \times (k+p)$; we expect Y to accurately span the column space of A
3. factor $Y = Q^*R$, so Q also accurately spans the column space of A .
4. form $B = Q^*A$, which is $(k+p) \times n$

We now approximate A by $Q^*B = Q^*Q^*A$, the projection of A onto the column space of Q .

In fact by computing the small SVD $B = U \Sigma V^T$, we can write $Q^*B = (Q^*U)^*\Sigma^*V^T$ as an approximate SVD of A .

The best possible approximation for any Q is when Q equals the first $k+p$ left singular vectors of $A = U_A \Sigma_A V_A^T$, in which case $Q^*Q^T A = U_A(1:m, 1:k+p) \Sigma_A(1:k+p, 1:k+p) (V_A(1:n, 1:k+p))^T$, and $\text{norm}(A - Q^*Q^T A, 2) = \text{sigma}(k+p+1)$. But our goal is to only get a good rank k approximation, so we are willing to settle for an error like $\text{sigma}(k+1)$.

Thm: If we choose each $F(i,j)$ to be i.i.d. $N(0,1)$, then the expected value of $\text{norm}(A - Q^*Q^T A, 2)$ is

$$E(\text{norm}(A - Q^*Q^T A, 2)) \leq (1 + 4 \sqrt{k+p}) / ((p-1) \sqrt{\min(m,n)}) * \text{sigma}(k+1)$$

and

$$\text{Prob}(\text{norm}(A - Q^*Q^T A, 2) \leq (1 + 11 \sqrt{k+p}) \sqrt{\min(m,n)} * \text{sigma}(k+1)) \geq 1 - 6/p^p$$

So for example choosing just $p=6$ makes the probability about .9999.

When is a Randomized Low Rank Factorization cheaper than a deterministic algorithm like QRCP, which costs $O(m*n*(k+p))$? When A is sparse, with $\text{nnz}(A)$ nonzero entries, the last 3 steps of the algorithm cost:

$$(2) 2 * \text{nnz}(A) * (k+p)$$

$$(3) 2 * m * (k+p)^2$$

$$(4) 2 * \text{nnz}(A) * (k+p)$$

each of which can be much smaller than $m*n*(k+p)$.

Whether the cost of (3) dominates (2) and (4) depends on the density of A ; if A averages $(k+p)$ or more nonzeros per row, then (2) and (4) dominate (3).

Later in Chap 7 we will consider other, so-called Krylov subspace algorithms that also attempt to compute an approximate SVD of a sparse matrix, whose cost is also dominated by a number of matrix-vector multiplications by A , each one costing $2 * \text{nnz}(A)$ flops. We will see that the ideas of Krylov subspace methods and randomized algorithms can be combined.

When A is dense, we need another approach. As mentioned above, forming an explicit dense F , and multiplying it by a dense A^*F , will cost $2 * m * n * (k+p)$ flops, comparable to QRCP. If we use SRFFT or SRHT for F , the cost of forming $Y = A^*F$ drops to $O(m*n*\log(n))$, potentially much less than QRCP. Factoring $Y = Q^*R$ is still $O(m*(k+p)^2)$, also potentially much less than QRCP when $k+p \ll n$. But the cost of $B = Q^T A$ is still $O(m*n*(k+p))$, like QRCP. So we need another idea:

Randomized Low-Rank Factorization via Row Extraction

1. choose a random $n \times (k+p)$ matrix F

2. form $Y = A^*F$, which is $m \times (k+p)$; we expect Y to accurately span the column space of A

3. factor $Y = Q^*R$, so Q also accurately spans the columns space of A .

4. find the "most linearly independent" $k+p$ rows of Q ; write $P^*Q = [Q_1; Q_2]$ where P is a permutation and Q_1 contains these $k+p$ rows; we can use GEPP on Q or QRCP on Q^T for this
5. let $X = P^*Q^*inv(Q_1) = [I; Q_2^*inv(Q_1)]$; we expect $norm(X)$ to be $O(1)$ (eg if QRCP yields a strong rank revealing decomposition)
6. let $P^*A = [A_1; A_2]$ where A_1 has $k+p$ rows; our low rank factorization of A is $A \sim P^T X^* A_1$

The cost of the algorithm on a dense matrix is

- (2) $O(m^*n^*\log(n))$ or $O(m^*n^*\log(k+p))$ if F is SRFFT or SRHT
- (3) $2^*m^*(k+p)^2$
- (4) $2^*m^*(k+p)^2$
- (5) $O(m^*(k+p)^2)$

which is much better than the previous $O(m^*n^*(k+p))$ cost when the rank $k+p$ is low compared to n .

The next theorem tells us that if QRCP or GEPP works well in step 5,

ie. $norm(X) = O(1)$, then we can't weaken the approximation by a large factor:

Thm: $norm(A - P^T X^* A_1) \leq (1 + norm(X)) * norm(A - Q^* Q^T^* A)$

Proof: suppose for simplicity that $P = I$. Then

$$\begin{aligned}
 norm(A - X^* A_1) &= norm(A - Q^* Q^T^* A + Q^* Q^T^* A - X^* A_1) \\
 &\leq norm(A - Q^* Q^T^* A) + norm(Q^* Q^T^* A - X^* A_1) \\
 &= norm(A - Q^* Q^T^* A) + norm(X^* Q_1^* Q^T^* A - X^* A_1) \\
 &\leq norm(A - Q^* Q^T^* A) + norm(X) * norm(Q_1^* Q^T^* A - A_1) \\
 &\leq norm(A - Q^* Q^T^* A) + norm(X) * norm(Q^* Q^T^* A - A) \\
 &= (1 + norm(X)) * norm(A - Q^* Q^T^* A)
 \end{aligned}$$