EECS 16B Designing Information Devices and Systems II
Fall 2021

Note 13B: Speeding up OMP

1 Setup for OMP

In 16A, you were (usually) introduced to orthogonal matching pursuit (OMP), an algorithm that can find sparse approximate solutions to systems of equations.

In 16A, the context was radio transmissions from the Internet Of Things. We have a huge number n of devices that could potentially be transmitting a signal, but we know that at most k are actually on. The m-long signal $\vec{y}$ we receive is the sum of the signals from each active device. Each device has a particular song that it sings if it is transmitting, with the amplitude of that song depending on its distance from the receiver, etc.
Specifically, let $A \in \mathbb{R}^{m \times n}$ be a matrix with $n > m$ (a wide matrix), where the kth column represents the song that could be sent by device k if it was indeed transmitting:

$$A = \begin{bmatrix} | & | & & | \\ \vec{S}_1 & \vec{S}_2 & \cdots & \vec{S}_n \\ | & | & & | \end{bmatrix}$$

Then, let $\vec{x} \in \mathbb{R}^n$ be a vector where the kth entry represents the "volume" of the kth device. Our received signal is

$$\vec{y} = A\vec{x} = \begin{bmatrix} | & | & & | \\ \vec{S}_1 & \vec{S}_2 & \cdots & \vec{S}_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ \vdots \\ x[n] \end{bmatrix} = \sum_{k=1}^{n} x[k]\vec{S}_k$$

This is the "noise-free" case. More realistically, $\vec{y}$ is only approximately given by the above, but there is another (presumably small) noise component as well. This noise is not out to get us in any way and is unrelated to any of the specific songs.

From $\vec{y}$, we want to infer $\vec{x}$, given that at most $k < m < n$ entries of $\vec{x}$ are non-zero. OMP gives us a way to find which k entries are "on." Then, we can form a matrix A_k with just the columns for the "on" devices, and we can find an estimate for $\vec{x}$ by using standard least-squares.

At a high level, in each iteration of the algorithm, we track the residual $\vec{r}$: the part of $\vec{y}$ still unexplained by the best least-squares fit to our current list of hypothesized "on" devices. In the beginning, when we don't have any hypothesized "on" devices yet, the residual $\vec{r}$ is just $\vec{y}$ itself. We find one more hypothesized "on" device by correlating the residual $\vec{r}$ with each column of A and taking the one whose correlation $\vec{S}_\ell^\top \vec{r}$ has the maximum absolute value. Having added this new device ℓ to our list of hypothesized "on" devices, we recompute the least-square estimate and update the residual. Then we repeat until done. Eventually, we find all k "on" devices or decide that the residual is already small enough with fewer than k, allowing us to solve for $\vec{x}$.

Explicitly, let $A_k = [\vec{S}_{i[1]}, \vec{S}_{i[2]}, \cdots, \vec{S}_{i[k]}]$ denote the matrix slice whose columns correspond to the k hypothesized"on" devices at iteration k. Here $i[1]$ is the index of the device that was hypothesized in the first iteration, $i[2]$ for the winner at the second iteration, and so on through $i[k]$. To find the residual $\vec{r}$ after the k-th iteration, we need to project $\vec{y}$ onto the columns of A_k and then subtract this projection from $\vec{y}$ (recall the projection formula):

$$\vec{r} = \vec{y} - A_k(A_k^\top A_k)^{-1}A_k^\top \vec{y}.$$

However, matrix inversion is computationally expensive — for an $n \times n$ matrix, inversion is O(n^3). We have to do inversion at every time step, making our algorithm slow when we are finding lots of columns.

Is there a way to avoid doing such computations every iteration?

The general secret to making an algorithm run faster is to find a way to avoid duplicating work. Is there some way we can instead better reuse the work we did in earlier iterations?

Yes! All we are doing is projecting $\vec{y}$ onto the subspace spanned by the k vectors $\{\vec{S}_{i[1]}, \vec{S}_{i[2]}, \ldots, \vec{S}_{i[k]}\}$. For general $\vec{S}_k$, we need to compute the projection matrix $A_k(A_k^\top A_k)^{-1}A_k^\top$. But if the $\vec{S}_i$ were all orthogonal, meaning that $\vec{S}_i^T\vec{S}_j = 0$ for $i \neq j$, then we have another faster way of computing the projection without having to invert a matrix.

2 Using Gram Schmidt to speed up OMP

Now, we would like to use Gram Schmidt to speed up OMP. Recall that in each iteration of OMP, we add one more device to our list of "on" devices, appending that song to a matrix of selected songs: $A_k = [A_{k-1} \mid \vec{S}_{i[k]}]$. Then, we use least squares to get the updated residual: $\vec{r}_k = \vec{y} - A_k(A_k^T A_k)^{-1}A_k^T\vec{y}$. This step is the one that most slows us down because we need to perform an expensive inversion each time. It also tells us that while the algorithm is running, the purpose of A_k is just to act as a representative for the subspace spanned by the selected columns of A.

The intuition is that instead of just appending the song $\vec{S}_{i[k]}$ to the matrix A_{k-1}, we can use Gram-Schmidt as we go to make sure that the relevant matrix Q_k is orthonormal. Let Q_k be the orthonormal matrix that represents the same subspace as A_k. How do we maintain Q_k?

First, we initialize $Q_0 = [\ ]$ just as we had initialized A_0. Then, every time we find a new song, we perform Gram-Schmidt orthonormalization before adding it to Q_k, so Q_k remains orthonormal at every iteration.

Recall that we want to solve for $\vec{x}$ in the following equation, where $\vec{x}$ has (at most) k nonzero entries:

$$A\vec{x} = \begin{bmatrix} | & | & & | \\ \vec{S}_1 & \vec{S}_2 & \cdots & \vec{S}_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ \vdots \\ x[n] \end{bmatrix} = \sum_{k=1}^{n} x[k]\vec{S}_k \approx \vec{y}.$$

Following through on our idea above, we get this procedure:

Procedure:

- At time $j = 0$, we have no selected columns, so our residual is all of $\vec{y}$. So we initialize our variables as follows: $\vec{r}_0 = \vec{y}$, $Q_0 = [\ ]$. We will also use a list (thought of as a vector, so that we can index into it naturally) $\vec{i}$ to hold the indices of the columns we have selected so far, initialized to an empty list [].

- Repeat for $j = 1, \ldots, k$: (or stop when the residual is too small, etc.)

  (a) Correlate $\vec{r}_{j-1}$ with all of the columns of A — i.e. compute $A^\top \vec{r}_{j-1}$. Find the song with the highest absolute correlation $|\vec{S}_\ell^\top \vec{r}_{j-1}|$. Record the maximizer's index[1] at the end of $\vec{i}$. In mathematical language:

$$i[j] = \arg\max_\ell |\vec{S}_\ell^\top \vec{r}_{j-1}|. \tag{1}$$

  (b) Orthonormalize $\vec{S}_{i[j]}$ relative to Q_{j-1} following the spirit of Gram-Schmidt:

    i. Find $\vec{e}_j$, or the part of $\vec{S}_{i[j]}$ that is orthogonal to the subspace spanned by Q_{j-1}:

$$\begin{aligned} \vec{e}_k &= \vec{S}_{i[j]} - Q_{j-1} Q_{j-1}^\top \vec{S}_{i[j]} \\ &= \vec{S}_{i[j]} - \sum_{\ell=1}^{j-1} \left(\vec{q}_\ell^\top \vec{S}_{i[j]} \right) \vec{q}_\ell. \end{aligned} \tag{2}$$

    This step does require a number of operations that is growing with the iteration count j, but the number of operations grows linearly in j instead of cubicly the way that inverting a generic $j \times j$ matrix from scratch by Gaussian elimination would cost.

    ii. Normalize to find $\vec{q}_j$ itself: $\vec{q}_j = \frac{\vec{e}_j}{\|\vec{e}_j\|}$.

    iii. Column concatenate[2] the matrix Q_{j-1} with $\vec{q}_j$ to update: $Q_j \leftarrow [Q_{j-1} \mid \vec{q}_j]$.

  (c) To find the new residual, project $\vec{y}$ onto Q_j:

$$\begin{aligned} \vec{r}_k &= \vec{y} - Q_j Q_j^\top \vec{y} \\ &= \vec{y} - \sum_{\ell=1}^{j} \left(\vec{q}_\ell^\top \vec{y} \right) \vec{q}_\ell. \end{aligned}$$

  We can speed this step up by noticing that in the previous iteration, we had:

$$\vec{r}_{j-1} = \vec{y} - \sum_{\ell=1}^{j-1} \left(\vec{q}_\ell^\top \vec{y} \right) \vec{q}_\ell.$$

  Almost all the terms are the same in the sum, except the last one. This means we don't have to redo all that work. We can compute the new residual by updating the previous one. We just need to subtracting the projection of $\vec{y}$ onto $\vec{q}_j$:

$$\vec{r}_j \leftarrow \vec{r}_{j-1} - (\vec{q}_j^\top \vec{y}) \vec{q}_j. \tag{3}$$

  This is a lot faster than having to recompute everything.

[1] This is why the notation $\arg\max_\ell f(\ell)$ is convenient. If $\ell = 5$ is where the function $f(\ell)$ takes on its maximum value, say $f(5) = 24$, then $\arg\max_\ell f(\ell)$ returns 5. Meanwhile $\max_\ell f(\ell)$ returns 24. In your calculus class, most likely the context was used to determine whether you were interested in 5 vs 24 when you were maximizing something. For writing an algorithm out explicitly, we need some notation to express our intentions.

[2] When trying to get an efficient implementation in a computer, it is important to avoid doing redundant copying of memory, especially for big arrays. After all, every actual copy would involve having to put charge onto a set of capacitors, and you will learn in 61C how this would all have to go through a set of wires in series, taking time because of RC time constants.

- When the loop above terminates, the information that we are most interested in is in the list $\vec{i}$ — these are the indices of the selected columns of A. To get an actual sparse solution $\vec{x}$ to our original problem $A\vec{x} \approx \vec{y}$, we need to do one last thing: compute $\vec{x}$.

  We can form the matrix slice $A_{\vec{i}}$ whose columns are the selected columns. (This can be built as we go above.)

$$A_{\vec{i}} = \begin{bmatrix} | & | & & | \\ \vec{S}_{i[1]} & \vec{S}_{i[2]} & \cdots & \vec{S}_{i[k]} \\ | & | & & | \end{bmatrix}$$

  Let $\vec{x}_{\vec{i}}$ be the entries of $\vec{x}$ corresponding to the indices in $\vec{i}$. Then, we have the approximate equation to solve:

$$A_{\vec{i}}\vec{x}_{\vec{i}} = \begin{bmatrix} | & | & & | \\ \vec{S}_{i[1]} & \vec{S}_{i[2]} & \cdots & \vec{S}_{i[k]} \\ | & | & & | \end{bmatrix} \begin{bmatrix} x[i[1]] \\ x[i[2]] \\ \vdots \\ x[i[k]] \end{bmatrix} \approx \vec{y}.$$

  The sliced matrix $A_{\vec{i}}$ has dimensions dimension $m \times k$ where $k < m$, so we can solve for the non-zero entries of $\vec{x}$ using standard least squares: $\vec{x}_{\vec{i}} = (A_{\vec{i}}^\top A_{\vec{i}}) A_{\vec{i}}^\top \vec{y}$. All the other entries of our final solution $\vec{x}$ are zero.

The above algorithm is substantially faster than the naive implementation of OMP.

3 Going even faster

In the above algorithm, there are still a couple of things that feel like they are redoing work that we've already done before. First and foremost, it is the very end. We end up doing a least-squares computation to get the final $\vec{x}$ solution — despite us having done projections already as we were updating the residual. Do we really have to invert a matrix?

If we think about it, along the way, we have effectively already computed the coefficients of the $\vec{q}_k$ that we need to represent the projection of $\vec{y}$ on the subspace spanned by Q_k — this happened when we computed $\vec{q}_k^\top \vec{y}$ in (3). So we know the coefficients in one basis — we just need them in the original coordinates. How would we change coordinates back to get the coefficients for $\vec{x}_{\vec{i}}$?

In general, changing coordinates requires solving a system of equations. A general system of equations with k equations and k unknowns costs something $O(k^3)$ to solve by Gaussian Elimination[3]. But this isn't a general system of equations. Instead, it has a peculiar form. This is because $\vec{S}_{i[1]}$ is just a constant multiple times $\vec{q}_1$, $\vec{S}_{i[2]}$ is just a linear combination of $\vec{q}_1$ and $\vec{q}_2$, and so on with $\vec{S}_{i[j]} = \sum_{\ell=1}(\vec{q}_\ell^\top \vec{S}_{i[j]})\vec{q}_\ell$.

Writing this in matrix form $A_{\vec{i}} = Q_k R$, the matrix R has a shape that resembles the shape that we get at the end of the downward pass of Gaussian Elimination:

$$R = \begin{bmatrix} \vec{q}_1^\top \vec{S}_{i[1]} & \vec{q}_1^\top \vec{S}_{i[2]} & \cdots & \vec{q}_1^\top \vec{S}_{i[k]} \\ 0 & \vec{q}_2^\top \vec{S}_{i[2]} & \cdots & \vec{q}_2^\top \vec{S}_{i[k]} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \vec{q}_k^\top \vec{S}_{i[k]} \end{bmatrix}. \tag{4}$$

[3] Recall that this is because each step downward in Gaussian elimination costs $O(k^2)$ operations since the matrix has size k^2, and there are k such steps.

Such matrices are called *upper-triangular* because all the potentially nonzero entries are in a triangle on top. Notice that all of these computations were essentially done during the computation of (2), with the last one effectively being done during the normalization part. So we can reuse all that work if we saved those computations in such a matrix.

Solving a system of equations involving such a matrix is cheap since we can do it using back-substitution, the same way that we finished the upward pass of Gaussian Elimination. This only costs $O(k^2)$ computations since we only have to flow information upward through the matrix, essentially touching each entry only once. This buys us a speedup at the end.

3.1 Even more speed

Once we have noticed the above pattern, we can wonder if we could save even more duplicated work. Where can we start looking for something to speed up? OMP is making repeated passes through the potential columns as it hunts for ones to select. Each time it picks one, it has to execute (2) which has a cost that is growing linearly with the iteration count k. This is because we need to compute the many $\vec{q}_\ell^\top \vec{S}_{i[k]}$ and use them to compute the orthogonal direction that this new $i[k]$ column brings to the subspace we are projecting onto. Can we speed this up so that it doesn't take an increased amount of time each iteration?

At first glance, there doesn't seem like much that we can do. After all, we need to compute all these inner products to orthonormalize. The only question is whether we could somehow rearrange the computations so that these could have already been computed before.

This is when we realize that in every iteration of OMP, we have to do (1) where we compute all the inner-products between the columns of A and the current residual. But the residuals are related to each other! So what is it that we actually have to compute anew?

Notice that at the start of iteration $k+1$ by (3),

$$\begin{aligned}\vec{S}_\ell^\top \vec{r}_k &= \vec{S}_\ell^\top (\vec{r}_{k-1} - (\vec{q}_k^\top \vec{y})\vec{q}_k) \\ &= \vec{S}_\ell^\top \vec{r}_{k-1} - (\vec{q}_k^\top \vec{y})\vec{S}_\ell^\top \vec{q}_k. \end{aligned} \tag{5}$$

The term $\vec{S}_\ell^\top \vec{r}_{k-1}$ was computed last time, so we can just keep this. The term $\vec{q}_k^\top \vec{y}$ is common to this entire iteration, and so we can just keep this. The new thing we actually need to compute is $\vec{S}_\ell^\top \vec{q}_k = \vec{q}_k^\top \vec{S}_\ell$. Notice that these are exactly the computations that we need to do (2).

This means that we can refactor the computations to compute in the above manner and save some work[4]. It turns out that this perspective of carefully tracking progress also opens the door to efficient implementations of more nuanced algorithms. For example, once OMP decides that it is going to add a column to its list, it uses that column to the maximum extent reasonable. This is why once a column has been selected, it will never get selected again. In principle however, you can imagine that after using a little bit of that column, the residual could change so that the winner of (1) is no longer that particular column. Other related algorithms can be made that try to navigate this differently, but these are not in scope for 16B.

[4] In principle, from the point of view of complexity scaling with problem size, we can make each iteration cost the same amount by actually tracking running updates (2) for each of the columns of A. It costs m operations to compute an inner-product $\vec{q}_k^\top \vec{S}_\ell$ — we can use the same number of operations to update a running version of (2) as well. In practice, we can just compute (2) using the saved values for $\vec{q}_k^\top \vec{S}_\ell$ only for the chosen column at lower actual computational cost, even though the sum would superficially seem to have growing complexity with iteration j.

3.2 Being more aggressive to speed up even more

Up to this point, we have been getting speedups by noticing how we can reuse work. Notice that to do this, we have had to actually change the computations (leveraging our understanding of what the algorithm is actually doing and the relevant math). However, the sped-up algorithms have been functionally identical to the original naive implementation of OMP.

To get further speedups, we have to be willing to potentially change the solutions that are found. The key insight is that in many problems, the various columns of A are approximately orthogonal to each other, of similar sizes, and there are many coefficients in $\vec{x}$ that are of the same relative size. This means that we can be more aggressive about selecting columns — we don't just have to take one maximizer in (1). We can add the column with the highest absolute correlation, as well as other columns whose absolute correlations are not that far away. When k is large, this lets us make many iterations worth of progress at the cost of a single iteration. Principled ways of deciding how many columns to add and to set thresholds require an understanding of probability and so are very much out of scope in 16B. However, you can play around with this to see what happens.

Contributors:

- Anant Sahai.
- Jennifer Shih.
- Rachel Hochman.
- Vasuki Narasimha Swamy.
- Steven Cao.

