COMPSCI 614: Randomized Algorithms with Applications to Data Science

Prof. Cameron Musco
University of Massachusetts Amherst. Spring 2024.
Lecture 8
• Problem Set 2 is due next Wednesday.
• One page project proposal due Tuesday 3/12.
• No quiz this week – focus on the problem set/project proposal.
Summary

Last Time:

- Graph connectivity with low communication
- Approach via Boruvka’s algorithm and sparse recovery/ℓ₀ sampling.

Today:

- Finish up ℓ₀ sampling analysis.
- Other approaches to sparse recovery and applications to data processing in streams.
- The count-sketch algorithm.
Summary

Last Time:
- Graph connectivity with low communication
- Approach via Boruvka’s algorithm and sparse recovery/\ell_0 sampling.

Today:
- Finish up \ell_0 sampling analysis.
- Other approaches to sparse recovery and applications to data processing in streams.
- The count-sketch algorithm.
$l_0$ Sampling and Graph Sketching
A Graph Communication Problem

Consider $n$ nodes, each only knows its own neighborhood. They want to send messages to a central server, who will then determine if the graph is connected.

Saw how this can be accomplished via $\ell_0$ sampling using messages of size just $O(\log^3 n)$. 
Key Ingredient 1: $\ell_0$ Sampling

**Theorem:** There exists a distribution over random matrices $A \in \mathbb{Z}^{O(\log^2 n) \times n}$ such that for any fixed $x \in \mathbb{Z}^n$, with probability at least $1 - 1/n^c$, we can learn $(i, x_i)$ for some $x_i \neq 0$ from $Ax$.

<table>
<thead>
<tr>
<th>Random sketching matrix $A$</th>
<th>$x$</th>
<th>$Ax$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -1 0 0 1 -1 0 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1 0 1 1 0 0 -1 0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1 1 -1 0 -1 -1 0 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 -1 -1 -1 1 1 1 0</td>
<td>-2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Key Property:** Given sketches $Ax_1$ and $Ax_2$, can easily compute $A(x_1 + x_2)$ and recover a nonzero entry from $x_1 + x_2$ with high probability.
Simulating Boruvka’s Algorithm via Sketches

- For independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$, each node computes $A_j v_i$ and sends these sketches to the central server. $O(\log^3 n)$ bits in total.

- The central server uses $A_1 v_1, \ldots, A_1 v_n$ to simulate the first step of Boruvka’s – i.e., to identify one outgoing edge from each node.

- For each subsequent step $j$, let $S_1, S_2, \ldots S_c$ be the current connected components. Observe that $\sum_{i \in S_k} v_i$ has non-zero entries corresponding exactly to the outgoing edges of $S_k$.

- Overall, using the $\log_2 n$ different sketches from each node, the server can simulate the full algorithm and determine with high probability if the graph is connected or not.
Simulating Boruvka’s Algorithm via Sketches

- For independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$, each node computes $A_j v_i$ and sends these sketches to the central server. $O(\log^3 n)$ bits in total.

- The central server uses $A_1 v_1, \ldots, A_1 v_n$ to simulate the first step of Boruvka’s – i.e., to identify one outgoing edge from each node.

- For each subsequent step $j$, let $S_1, S_2, \ldots S_c$ be the current connected components. Observe that $\sum_{i \in S_k} v_i$ has non-zero entries corresponding exactly to the outgoing edges of $S_k$.

- So, from $A_j \sum_{i \in S_k} v_i = \sum_{i \in S_k} A_j v_i$, the server can find an outgoing edge from each connected component $S_k$. Thus, the server can simulate the $j^{th}$ round of Boruvka’s algorithm.

- Overall, using the $\log_2 n$ different sketches from each node, the server can simulate the full algorithm and determine with high probability if the graph is connected or not.
Implementing $\ell_0$ Sampling
Construction:

- Let $S_0, S_1, \ldots, S_{\log_2 n}$ be random subsets of $[n]$. Each element is included in $S_j$ independently with probability $1/2^j$.

- For each $S_j$, compute $a_j = \sum_{i \in S_j} x_i$, $b_j = \sum_{i \in S_j} x_i \cdot i$ and $c_j = \sum_{i \in S_j} x_i \cdot r^i \mod p$, where $r$ is a random value in $[p]$ and $p$ is a prime with $p \geq n^c$ for some large constant $c$.

- Observe: The vector $[a_0, \ldots, a_{\log_2 n}, b_0, \ldots, b_{\log_2 n}, c_0, \ldots, c_{\log_2 n}]$ can be written as $Ax$, where $A \in \mathbb{Z}^{3 \log_2 n \times n}$ is a random matrix.
Construction Intuition

We will recover a nonzero element from a sampling level when there is exactly one nonzero element at that level.

With good probability, there is will exactly one element at some level. Can improve success probability via repetition.
$S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^i$.

$a_j = \sum_{i \in S_j} x_i$, $b_j = \sum_{i \in S_j} x_i \cdot i$ and $c_j = \sum_{i \in S_j} x_i \cdot r^i \mod p$, where $r$ is a random value in $[p]$ and $p = n^c$ for large enough constant $c$.

Claim 1: If there is a unique $i \in S_j$ with $x_i \neq 0$, then $a_j = x_i$ and $b_j = x_i \cdot i$. So, from these quantities we can exactly determine $(i, x_j)$.

Claim 2: $c_j$ lets us test if there is a unique such $i$. In particular, we check that $b_j a_j \in [n]$ and that $c_j = a_j r b_j / a_j \mod p$.

- If there is a unique $i \in S_j$ with $x_i \neq 0$, the test passes.
- If not, it fails with probability at most $\frac{n}{n^c} = \frac{1}{n^{c-1}}$. 

$X \pm X_2$
Recovering Unique Nonzeros

$S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^i$. $a_j = \sum_{i \in S_j} x_i$, $b_j = \sum_{i \in S_j} x_i \cdot i$ and $c_j = \sum_{i \in S_j} x_i \cdot r^i \mod p$, where $r$ is a random value in $[p]$ and $p = n^c$ for large enough constant $c$.

Claim 1: If there is a unique $i \in S_j$ with $x_i \neq 0$, then $a_j = x_i$ and $b_j = x_i \cdot i$. So, from these quantities we can exactly determine $(i, x_j)$.

Claim 2: $c_j$ lets us test if there is a unique such $i$. In particular, we check that $\frac{b_j}{a_j} \in [n]$ and that $c_j = a_j \cdot r^{b_j/a_j} \mod p$.

- If there is a unique $i \in S_j$ with $x_i \neq 0$, the test passes.
- If not, it fails with probability at most $\frac{n}{p} = \frac{1}{n^{c-1}}$. 
Recovering Unique Nonzeros

Claim 2: $c_j$ lets us test if there is a unique such $i$. In particular, we check that $\frac{b_j}{a_j} \in [n]$ and that $c_j = a_j \cdot r^{b_j/a_j} \mod p$.

- If there is a unique $i \in S_j$ with $x_i \neq 0$, the test passes.
- If not, it fails with probability at most $\frac{n}{p} \leq \frac{1}{n^{c-1}}$. 
Claim 2: $c_j$ lets us test if there is a unique such $i$. In particular, we check that $b_j/a_j \in [n]$ and that $c_j = a_j \cdot r^{b_j/a_j} \mod p$.

- If there is a unique $i \in S_j$ with $x_i \neq 0$, the test passes.
- If not, it fails with probability at most $\frac{n}{p} \leq \frac{1}{n^{c-1}}$.

Proof via polynomial identity testing: If $|\{i \in S_j : x_i \neq 0\}| > 1$, then

$$p(r) = c_j - a_j r^{b_j/a_j} \mod p = \sum_{i \in S_j} x_i r^i - a_j r^{b_j/a_j} \mod p$$

is a non-zero polynomial of degree at most $n$ over $\mathbb{Z}_p$. Only if the test passes, $p(r)$ has degree $\leq 1$. 

\[0.5r^3 + 1 \cdot r^4 - 2r^5?\]
Recovering Unique Nonzeros

**Claim 2:** $c_j$ lets us test if there is a unique such $i$. In particular, we check that $\frac{b_j}{a_j} \in [n]$ and that $c_j = a_j \cdot r^{b_j/a_j} \mod p$.

- If there is a unique $i \in S_j$ with $x_i \neq 0$, the test passes.
- If not, it fails with probability at most $\frac{n}{p} \leq \frac{1}{n^{c-1}}$.

**Proof via polynomial identity testing:** If $|\{i \in S_j : x_i \neq 0\}| > 1$, then

$$p(r) = c_j - a_j r^{b_j/a_j} \mod p = \sum_{i \in S_j} x_i r^i - a_j r^{b_j/a_j} \mod p$$

is a non-zero polynomial of degree at most $n$ over $\mathbb{Z}_p$.

- This polynomial has $\leq n$ roots, so for a random $r \in [p]$, $\Pr[p(r) = 0] \leq \frac{n}{p}$.
- Thus, $c_j = a_j r^{b_j/a_j}$ with probability $\leq \frac{n}{p} \leq \frac{1}{n^{c-1}}$. 


Completing The Analysis

Recall: $S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^i$.

- If any $S_j$ contains a unique $i$ with $x_i \neq 0$, we will recover it.
- It remains to show that with good probability, at least one $S_j$ contains such an $i$. 

\[
\Pr\left[ \left\{ i \in S_j : x_i \neq 0 \right\} \right] = \frac{1}{8}.
\]

If we repeat the whole process $t = \Theta(\log n)$ times, with probability $\geq 1 - 1/n^c$ we will recover some nonzero element of $x$. In total, $A$ is a random matrix with $t \cdot \log_2 n = \Theta(\log n)$ rows.
Completing The Analysis

Recall: $S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^j$.

- If any $S_j$ contains a unique $i$ with $x_i \neq 0$, we will recover it.
- It remains to show that with good probability, at least one $S_j$ contains such an $i$.

Claim: For $j$ with $2^{j-2} \leq \|x\|_0 \leq 2^{j-1}$, $\Pr[\{|i \in S_j : x_i \neq 0\}| = 1] \geq 1/8$. 

\[ \begin{align*}
\text{nonzero entries in } x
\end{align*} \]
Completing The Analysis

Recall: $S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^j$.

- If any $S_j$ contains a unique $i$ with $x_i \neq 0$, we will recover it.
- It remains to show that with good probability, at least one $S_j$ contains such an $i$.

Claim: For $j$ with $2^{j-2} \leq \|x\|_0 \leq 2^{j-1}$, $\Pr[\{|i \in S_j : x_i \neq 0| = 1\}] \geq 1/8$.

$$\Pr[\{|i \in S_j : x_i \neq 0| = 1\}] = \|x\|_0 \cdot \frac{1}{2^j} \cdot \left(1 - \frac{1}{2^j}\right)^{\|x\|_0 - 1}$$
Completing The Analysis

Recall: $S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^j$.

- If any $S_j$ contains a unique $i$ with $x_i \neq 0$, we will recover it.
- It remains to show that with good probability, at least one $S_j$ contains such an $i$.

Claim: For $j$ with $2^{j-2} \leq \|x\|_0 \leq 2^{j-1}$, $\Pr[|\{i \in S_j : x_i \neq 0\}| = 1] \geq 1/8$.

$$
\Pr[|\{i \in S_j : x_i \neq 0\}| = 1] = \|x\|_0 \cdot \frac{1}{2^j} \cdot \left(1 - \frac{1}{2^j}\right)^{\|x\|_0 - 1}
\geq \frac{\|x\|_0}{2^j} \left(1 - \frac{\|x\|_0}{2^j}\right)
$$

$$
\left(1 - \frac{1}{2^j}\right)^{\|x\|_0 - 1} \geq \left(1 - \frac{1}{2^j}\right)^{\|x\|_0} \geq \left(1 - \frac{\|x\|_0}{2^j}\right)
$$
Completing The Analysis

Recall: $S_0, \ldots, S_{\log_2 n}$ are random subsets of $[n]$, sampled at rates $1/2^j$.

- If any $S_j$ contains a unique $i$ with $x_i \neq 0$, we will recover it.
- It remains to show that with good probability, at least one $S_j$ contains such an $i$.

**Claim:** For $j$ with $2^{j-2} \leq \|x\|_0 \leq 2^{j-1}$, $\Pr[\{|i \in S_j : x_i \neq 0\}| = 1] \geq 1/8$.

\[
\Pr[\{|i \in S_j : x_i \neq 0\}| = 1] = \|x\|_0 \cdot \frac{1}{2^j} \cdot \left(1 - \frac{1}{2^j}\right)^{\|x\|_0 - 1}
\]

\[
\geq \frac{\|x\|_0}{2^j} \left(1 - \frac{\|x\|_0}{2^j}\right)
\]

\[
\geq \frac{1}{4} \cdot \left(1 - \frac{1}{2}\right) = \frac{1}{8}.
\]
Completing The Analysis

Recall: \( S_0, \ldots, S_{\log_2 n} \) are random subsets of \([n]\), sampled at rates \(1/2^j\).

- If any \( S_j \) contains a unique \( i \) with \( x_i \neq 0 \), we will recover it.
- It remains to show that with good probability, at least one \( S_j \) contains such an \( i \).

Claim: For \( j \) with \( 2^{j-2} \leq \|x\|_0 \leq 2^{j-1} \), \( \Pr[|\{i \in S_j : x_i \neq 0\}| = 1] \geq 1/8 \).

\[
\Pr[|\{i \in S_j : x_i \neq 0\}| = 1] = \|x\|_0 \cdot \frac{1}{2^j} \cdot \left(1 - \frac{1}{2^j}\right)^{\|x\|_0 - 1}
\]
\[
\geq \frac{\|x\|_0}{2^j} \left(1 - \frac{\|x\|_0}{2^j}\right)
\]
\[
\geq \frac{1}{4} \cdot \left(1 - \frac{1}{2}\right) = \frac{1}{8}.
\]

If we repeat the whole process \( t = O(\log n) \) times, with probability \( \geq 1 - 1/n^c \) we will recover some nonzero element of \( x \). In total, \( A \) is a random matrix with \( t \cdot \log_2 n = O(\log^2 n) \) rows.
Application to Streaming Computation
A Graph Streaming Problem

Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.

![Graph Diagram]

Algorithmic Question:

How much memory must an algorithm use to solve this problem with high probability?

What is the worst-case memory required by a naive deterministic algorithm that just stores the current state of the graph? How can you improve on this when there are no edge deletions?
Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.
A Graph Streaming Problem

Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.
Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.
A Graph Streaming Problem

Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.
Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.
A Graph Streaming Problem

Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.

Algorithmic Question: How much memory must an algorithm use to solve this problem with high probability? $O(n^2)$
A Graph Streaming Problem

Consider a setting where an algorithm must process a stream of edge insertions or deletions, which define a graph. At the end of the stream, the algorithm should output whether that graph is connected or not.

Algorithmic Question: How much memory must an algorithm use to solve this problem with high probability?

What is the worst-case memory required by a naive deterministic algorithm that just stores the current state of the graph? How can you improve on this when there are no edge deletions?
Randomized Solution via $\ell_0$ sampling

- The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_j v_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{(\binom{n}{2})}$ is the incidence vector for node $u$.
- $O(n \log^3 n)$ bits of storage in total.

$A_j v_u = O(\log^2 n)$ space in nodes

$\log n$ random $A_1, \ldots, A_{\log n}

Total space $O(n \log^3 n)$
Randomized Solution via $\ell_0$ sampling

• The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_j v_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{\binom{n}{2}}$ is the incidence vector for node $u$.

• $O(n \log^3 n)$ bits of storage in total.

• **Key Idea: Linear Updates.** When an edge $(u, v)$ is inserted or deleted, one entry is either incremented or decremented in each of $v_u, v_v$. The algorithm can update $A_j v_u$ and $A_j v_v$ in $O(\log^2 n)$ time – simply set $A_j v_u = A_j v_u \pm A_{j,k}$.
Randomized Solution via $\ell_0$ sampling

- The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_jv_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{(n)}_{(n^2)}$ is the incidence vector for node $u$.

- $O(n \log^3 n)$ bits of storage in total.

- **Key Idea: Linear Updates.** When an edge $(u, v)$ is inserted or deleted, one entry is either incremented or decremented in each of $v_u, v_v$. The algorithm can update $A_jv_u$ and $A_jv_v$ in $O(\log^2 n)$ time – simply set $A_jv_u = A_jv_u \pm A_{j,k}$.

\[
\begin{bmatrix}
1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 \\
-1 & 0 & 1 & 1 & 0 & 0 & -1 & 0 \\
1 & 1 & -1 & 0 & -1 & -1 & 0 & 1 \\
0 & -1 & -1 & -1 & 1 & 1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
v_u \\
1 \\
0 \\
0 \\
-1 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
= 
\begin{bmatrix}
A_jv_u \\
1 \\
-2 \\
1 \\
1 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]
Randomized Solution via $\ell_0$ sampling

• The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_jv_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{\binom{n}{2}}$ is the incidence vector for node $u$.

• $O(n \log^3 n)$ bits of storage in total.

• **Key Idea: Linear Updates.** When an edge $(u, v)$ is inserted or deleted, one entry is either incremented or decremented in each of $v_u, v_v$. The algorithm can update $A_jv_u$ and $A_jv_v$ in $O(\log^2 n)$ time – simply set $A_jv_u = A_jv_u \pm A_{j,k}$.

$$
\begin{align*}
&I_0 \text{ sampling matrix } A_j \\
&\begin{pmatrix}
1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 \\
-1 & 0 & 1 & 1 & 0 & 0 & -1 & 0 \\
1 & 1 & -1 & 0 & -1 & -1 & 0 & 1 \\
0 & -1 & -1 & -1 & 1 & 1 & 1 & 0
\end{pmatrix}
\end{align*}
\begin{pmatrix}
v_u & \\
\end{pmatrix}
= 
\begin{pmatrix}
A_jv_u & \\
\end{pmatrix}
\begin{pmatrix}
1 \\
-2 \\
1 \\
1 \\
1 \\
1 \\
0 \\
0 \\
1 \\
0
\end{pmatrix}
$$
Randomized Solution via $\ell_0$ sampling

- The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_j v_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{(n)}$ is the incidence vector for node $u$.

- $O(n \log^3 n)$ bits of storage in total.

- **Key Idea: Linear Updates.** When an edge $(u, v)$ is inserted or deleted, one entry is either incremented or decremented in each of $v_u, v_v$. The algorithm can update $A_j v_u$ and $A_j v_v$ in $O(\log^2 n)$ time – simply set $A_j v_u = A_j v_u \pm A_{j,k}$.

![L_0 sampling matrix $A_j$](image)
Randomized Solution via $\ell_0$ sampling

- The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_jv_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{(n)}$ is the incidence vector for node $u$.
- $O(n \log^3 n)$ bits of storage in total.
- **Key Idea: Linear Updates.** When an edge $(u, v)$ is inserted or deleted, one entry is either incremented or decremented in each of $v_u, v_v$. The algorithm can update $A_jv_u$ and $A_jv_v$ in $O(\log^2 n)$ time – simply set $A_jv_u = A_jv_u \pm A_{j,k}$.
- At the end of the stream (or at any time during it) can use the sketched neighborhoods to simulate Boruvka’s algorithm and determine connectivity with high probability.
Randomized Solution via $\ell_0$ sampling

- The algorithm samples independent $\ell_0$ sampling matrices $A_1, \ldots, A_{\log_2 n}$ and maintains $A_j v_u$ for all $j$ and all $u \in [n]$, where $v_u \in \mathbb{R}^{(n)}$ is the incidence vector for node $u$.
- $O(n \log^3 n)$ bits of storage in total.
- **Key Idea: Linear Updates.** When an edge $(u, v)$ is inserted or deleted, one entry is either incremented or decremented in each of $v_u, v_v$. The algorithm can update $A_j v_u$ and $A_j v_v$ in $O(\log^2 n)$ time – simply set $A_j v_u = A_j v_u \pm A_{j,k}$.
- At the end of the stream (or at any time during it) can use the sketched neighborhoods to simulate Boruvka’s algorithm and determine connectivity with high probability.
- Can think of the algorithm as computing $AB \in \mathbb{R}^{\log^3 n \times n}$ where $A \in \mathbb{R}^{\log^3 n \times \binom{n}{2}}$ is made up of the appended sketching matrices and $B \in \mathbb{R}^{\binom{n}{2} \times n}$ is the vertex-edge-incidence matrix.
Other Applications of Linear Sketching
Linear Sketching

- $\ell_0$ sampling is an example of a linear sketching algorithm. We compress our data via a random linear function (i.e., the random matrix $A$), and prove that we can still recover useful information from the compression.

\[
\begin{pmatrix}
1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 \\
-1 & 0 & 1 & 1 & 0 & 0 & -1 & 0 \\
1 & 1 & -1 & 0 & -1 & -1 & 0 & 1 \\
0 & -1 & -1 & -1 & 1 & 1 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
x \\
1 \\
0 \\
0 \\
-2 \\
0 \\
0 \\
3 \\
0
\end{pmatrix} =
\begin{pmatrix}
1 \\
-2 \\
1 \\
5
\end{pmatrix}
\]
Linear Sketching

• $\ell_0$ sampling is an example of a linear sketching algorithm. We compress our data via a random linear function (i.e., the random matrix $A$), and prove that we can still recover useful information from the compression.

\[
\begin{bmatrix}
1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 \\
-1 & 0 & 1 & 1 & 0 & 0 & -1 & 0 \\
1 & 1 & -1 & 0 & -1 & -1 & 0 & 1 \\
0 & -1 & -1 & -1 & 1 & 1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
x_8
\end{bmatrix}
= \begin{bmatrix}
1 \\
1 \\
-2 \\
1 \\
5
\end{bmatrix}
\]

• Linearity is useful because it lets us easily aggregate sketches in distributed settings and update sketches in streaming settings.

• Aside from recovering non-zero entries we might want to estimate norms or other aggregate statistics of $x$, find large magnitude entries, sample entries with probabilities according to their magnitudes.
Goal: For a vector $x \in \mathbb{R}^n$ we would like to find all entries of $x$ with magnitude at least $\epsilon \|x\|_2$ or $\epsilon \|x\|_1$. $\epsilon > 0$
**Linear Sketching for Heavy-Hitters Identification**

**Goal:** For a vector $\mathbf{x} \in \mathbb{R}^n$ we would like to find all entries of $\mathbf{x}$ with magnitude at least $\epsilon \|\mathbf{x}\|_2$ or $\epsilon \|\mathbf{x}\|_1$.

**Common Application:**

- $\mathbf{x}$ is a vector of counts (e.g., views of videos, searches for products, visits from IP addresses, etc.) and we would like to identify all items with large counts.

- We often cannot store all of $\mathbf{x}$ in one place but must store a small-space compression of $\mathbf{x}$ as counts are updated over time, or must aggregate information about $\mathbf{x}$ across multiple machines.
**Set up:** We would like to estimate all entries of a vector $\mathbf{x} \in \mathbb{R}^n$ up to error $\epsilon \| \mathbf{x} \|_2$ with probability at least $1 - \delta$, from a small linear sketch, of size $O\left(\frac{\log(1/\delta)}{\epsilon^2}\right)$. 

\[
\begin{bmatrix}
\end{bmatrix} \quad \begin{bmatrix}
\end{bmatrix} = \mathbf{A} \begin{bmatrix}
\end{bmatrix}
\]
Count Sketch

Set up: We would like to estimate all entries of a vector $x \in \mathbb{R}^n$ up to error $\epsilon \|x\|_2$ with probability at least $1 - \delta$, from a small linear sketch, of size $O\left(\frac{\log(1/\delta)}{\epsilon^2}\right)$.

- Let $m = O(1/\epsilon^2)$ and $t = O(\log(1/\delta))$.
- Pick $t$ random pairwise independent hash functions $h_1, \ldots, h_t : [n] \to [m]$.
- Pick $t$ random pairwise independent hash functions $s_1, \ldots, s_t : [n] \to \{-1, 1\}$.
Count Sketch

**Set up:** We would like to estimate all entries of a vector \( \mathbf{x} \in \mathbb{R}^n \) up to error \( \epsilon \| \mathbf{x} \|_2 \) with probability at least \( 1 - \delta \), from a small linear sketch, of size \( O \left( \frac{\log(1/\delta)}{\epsilon^2} \right) \).

- Let \( m = O(1/\epsilon^2) \) and \( t = O(\log(1/\delta)) \).
- Pick \( t \) random **pairwise independent** hash functions \( h_1, \ldots, h_t : [n] \rightarrow [m] \).
- Pick \( t \) random pairwise independent hash functions \( s_1, \ldots, s_t : [n] \rightarrow \{-1, 1\} \).
- Compute \( t \) independent estimates of \( x(i) \) as
  \[
  \tilde{x}_j(i) = s_j(i) \cdot \sum_{k : h_j(k) = h_j(i)} x(k) \cdot s(k).
  \]