Lecture 1

Streaming Algorithms and Concentration Inequalities

In this lecture we will cover some basic concentration inequalities: given a random variable Z, these inequalities let us bound the probability that Z deviates significantly from its mean. If we don't know *anything* about Z then there is not much that can be said: think of $Z = \alpha$ with probability $1/2$, and $-\alpha$ with probability $1/2$, for an arbitrary real α. In this case $\mathbf{E}[Z] = 0$ but Z can take values arbitrarily far from its mean, with large probability. The more we assume about the structure of Z the better bounds we'll be able to get, with the strongest bounds kicking in when $Z = X_1 + \cdots X_N$ for independent, or partially independent, X_i.

Motivation: estimating the number of distinct elements in a data stream

The streaming model of computation is as follows. There is a "stream" of values $\sigma = (a_1, a_2, \ldots)$ arriving, where $a_i \in \{1, 2, \ldots, n\}$ for all i. The goal is to compute certain properties of the stream using the least possible space. For example, how many distinct elements are there? This is called the 0th moment of the stream, m_0.

Let's first try to compute m_0 using a deterministic procedure. How much space do you need? The most naïve thing is to store one copy of each new element we see. In the worst case all elements are distinct, and this will require space $n \log n$. Can you do better?

Suppose you had a deterministic algorithm that used only space s. For $x \in \{0,1\}^n$ consider the stream $((1, x_1), \ldots, (n, x_n))$. This stream has n distinct elements and we can run our algorithm on it to obtain a memory $m(x) \in \{0,1\}^s$. Now, given $m(x)$, initialize the algorithm with memory $m(x)$ and give it a single element $(i, 0)$. Check if the number of distinct elements changes: if it increases by 1 this means $x_i = 1$, and if it doesn't change it menas $x_i = 0$. So we can recover x from $m(x)$, hence $m(x)$ must have size n.

This argument only applies to deterministic algorithms that are exact. But what about an algorithm that approximates m_0, say up to 10% error? In your homework you will see that even in this case one needs space $\Omega(n)$.

Now we're going to design an algorithm using much less space, but the algorithm won't be deterministic — it'll be randomized. What can we achieve using randomness? For instance, here is a simple trick. Suppose we want that at any time t our memory contains an element $b \in \{1, \ldots, n\}$ that is uniformly distributed over the set of elements seen so far. How do we do this?

The solution is really simple: at each time step $t, t \geq 1$, we replace our current element with the latest item a_t with probability $1/t$. The analysis that follows is also simple: What is the probability that $b = a_i$ at some time $t \geq i$?

$$\begin{aligned}
\mathbf{Pr}[b = a_i] &= \frac{1}{i}\Big(1 - \frac{1}{i+1}\Big) \cdots \Big(1 - \frac{1}{t}\Big) \\
&= \frac{1}{i}\Big(\frac{i}{i+1} \cdots \frac{t-1}{t}\Big) \\
&= \frac{1}{t},
\end{aligned}$$

as desired. Note that if instead we can store k items, we just run k of the above algorithms in parallel. Using the birthday paradox, if we keep about $O(\sqrt{n})$ elements in memory, we will see a collision if and only if the number of distinct elements is n. So we already got a quadratic improvement in space. But we can do much better!

Let f be a random function $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and consider the following procedure:

1. **Initialization:** Set $z = 0$.
2. **Process a_j:** Let ℓ be the largest power of 2 that divides $f(a_j)$. If $\ell > z$ then $z \leftarrow \ell$.
3. **Output:** $2^{z+\frac{1}{2}}$.

Why does this work? Well, if there are d distinct elements, then there is a good chance that one of the $f(d)$ will be divisible by $\log d$ (note that ℓ is simply the number of trailing zeros in the binary representation of $f(d)$). On the other hand it is pretty unlikely that there is an item that has much more than $\log d$ zeros in a row.

What is the space requirement? Storing z only requires about $\log n$ bits. Note there is a problem though: we also need to store f, and if f is a random function we need to store its full table, which again requires $n \log n$ bits! This is an issue, but we'll see how to get around it by using a partially random f — an f that is chosen randomly from a much smaller family of functions than all functions.

How do we analyze this? What is the probability that the algorithm returns an estimate that is correct within relative error 10%? And if it is not high enough, how do we improve the estimate? Before we see how to do all this let's go back to the basics and study some concentration inequalities.

1.1 Markov's Inequality

Suppose our only assumption is $Z \geq 0$. Then Markov's inequality already lets us say something nontrivial:

Theorem 1.1 (Markov)**.** *For a random variable $Z \geq 0$ and any $k > 0$,*

$$\mathbf{Pr}(Z \geq k) \leq \frac{\mathbb{E}[Z]}{k}$$

By choosing $k = \alpha\,\mathbf{E}[Z]$ for $\alpha \geq 1$ we can reformulate the bound as $\mathbf{Pr}(Z \geq \alpha\,\mathbf{E}[Z]) \leq \alpha^{-1}$

Proof. $\mathbf{E}[Z] \geq k\,\mathbf{Pr}(Z \geq k) + 0\,\mathbf{Pr}(Z < k)$. □

We can also do a "proof by picture". Consider the function $g(Z) = \frac{Z}{k}$ and let $f(Z)$ be an indicator function which is 1 if $Z \geq k$ and 0 otherwise. Then $\mathbf{Pr}(Z \geq k) = \mathbf{E}[f(Z)] \leq \mathbf{E}[g(Z)] = \frac{\mathbf{E}[Z]}{k}$.

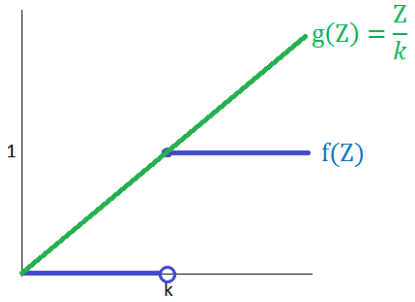

Figure 1.1: $g(Z) \geq f(Z)$ everywhere

Exercise 1. For every $k \geq \mathbf{E}[Z]$, give an example of a random variable for which Markov's inequality is tight for this choice of k.

1.2 Chebyshev's Inequality

Now suppose that we have control over the variance $\mathbb{E}[(Z-\mu)^2]$, where $\mu = E[Z]$.

Theorem 1.2 (Chebyshev)**.** *For a random variable $Z \geq 0$ with mean $\mathbf{E}[Z] = \mu$ and any $t \geq 0$:*

$$\mathbf{Pr}\left(|Z-\mu| \geq t\right) \leq \frac{\mathbf{E}[(Z-\mu)^2]}{t^2}.$$

Proof. Note that the events $|Z - \mu| \geq t$ and $(Z - \mu)^2 \geq t^2$ are equivalent. We can apply Markov's inequality:

$$\begin{aligned}\mathbf{Pr}\left(|Z - \mu| \geq t\right) &= \mathbf{Pr}\left(Z - \mu\right)^2 \geq t^2\right) \\ &\leq \frac{\mathbf{E}[(Z - \mu)^2]}{t^2}.\end{aligned}$$

□

Here again we can give a proof by picture with $g(Z) = \frac{(Z-\mu)^2}{t^2}$ and $f(Z)$ an indicator function which is 1 if $|Z - \mu| \geq t$ and 0 otherwise:

$$\begin{aligned}\mathbf{Pr}\left(|Z - \mu| \geq t\right) &= \mathbf{E}[f(Z)] \\ &\leq \mathbf{E}[g(Z)] \\ &= \mathbf{E}\left[\frac{(Z - \mu)^2}{t^2}\right] \\ &= \frac{\mathbf{E}[(Z - \mu)^2]}{t^2}.\end{aligned}$$

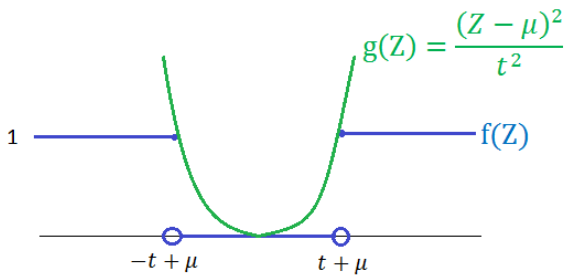

Figure 1.2: $g(Z) \geq f(Z)$ everywhere

From these pictorial proofs one can see that Chebyshev's inequality is just Markov's inequality with a different definition of $g(Z)$. Any function g which is larger than the indicator function will provide a different bound, and depending on the context it might be possible to get a better bound than Markov's or Chebyshev's. We will see one more example of this later on.

Exercise 2. For every expectation $\mu \geq 0$, variance $\sigma^2 \geq 0$ and $t \geq 0$ give a random variable for which Chebyshev's inequality is tight.

1.2.1 Pairwise independence

Chebyshev's inequality is useful when we have good control over the variance $\mathbf{E}[(Z - \mu)]$ of Z. A typical situation where this is the case is if Z can be written as the average of

independent random variables, $Z = (1/N)(X_1 + \cdots + X_N)$. Then

$$\mathbf{E}[Z] = \frac{1}{N}\big(\mathbf{E}[X_1] + \cdots + \mathbf{E}[X_N]\big) = \frac{1}{N}\big(\mu_1 + \cdots + \mu_N\big),$$

so

$$\begin{aligned}\mathbf{Var}(Z) &= \mathbf{E}\left[\left(\frac{1}{N}\big(X_1 + \cdots X_N\big) - \frac{1}{N}\big(\mu_1 + \cdots + \mu_N\big)\right)^2\right] \\ &= \frac{1}{N^2}\sum_i \mathbf{E}\left[\big(X_i - \mu_i\big)^2\right] - \frac{2}{N^2}\sum_{i,j}\mathbf{E}\left[(X_i - \mu_i)(X_j - \mu_j)\right] \\ &= \frac{1}{N^2}\sum_i \mathbf{Var}(X_i),\end{aligned}$$

where in the last line we used that X_i and X_j are independent for $i \neq j$ to write $\mathbf{E}[X_iX_j] = \mathbf{E}[X_i]\,\mathbf{E}[X_j]$. But note the property we are using here is weaker than full independence: it is called *pairwise independence*.

Definition 1.3. A family of random variables $(X_1, \ldots, X_N)$ is called pairwise independent if each X_i is uniformly distributed, and for all $i \neq j$ the random variables (X_i, X_j) are independent.

So Chebyshev's inequality gives us a good concentration bound for pairwise independent random variables. We'll see many uses for this, in particular to the problem of derandomization later on in the course. Today we'll see an application to the problem of estimating frequency moments in streaming.

1.3 Application: estimating frequency moments in the streaming model

1.3.1 Streaming algorithms

A typical goal in streaming would be to estimate the frequency

$$f_i = \frac{|\{1 \leq t \leq T : a_t = i\}|}{T}$$

of element $i \in \{1, \ldots, n\}$. We already saw the 0th moment, which counts the number of distinct elements. The first moment is simply the total number of elements in the stream. The second moment $m_2 = \sum_i f_i^2$? gives a measure of "discrepancy" within the stream: the higher m_2 the less "uneven" the frequencies are. We're going to see small-space algorithms for computing each of these.

We already saw an algorithm for estimating m_0. Before we can analyze it we need to deal with the random function f: instead of a fully random function, we're going to chose a partially random one from a restricted family — a family of hash functions.

1.3.2 Hash Functions

Definition 1.4. A family $\mathcal{H}$ of functions $h : A \mapsto B$ is called k-wise independent if for any distinct points $x_1, ..., x_k \in A$ and $i_1, ..., i_k \in B$,

$$\Pr_{h \in \mathcal{H}} \big(h(x_1) = i_1, \ldots, h(x_k) = i_k\big) = \frac{1}{|B|^k}.$$

So a 1-wise independent family of hash functions is just a family such that any element x in the domain is mapped to a random element in the range, when the function is chosen at random.

Example. For $A = B = \{0, 1, ..., p-1\}$ where p is a prime number, consider the following family of functions:

$$\mathcal{H}_2 = \big\{ f_{a,b} : x \mapsto ax + b \mod p,\ (a, b) \in \{0, \ldots, p-1\}^2 \big\}.$$

Then $\mathcal{H}_2$ is a family of 2-wise independent hash functions. To check this we only need to evaluate for any $x_1 \neq x_2$, i_1 and i_2,

$$\begin{aligned} \Pr_{a,b}(ax_1 + b = i_1 \wedge ax_2 + b = i_2) &= \Pr_{a,b}\left(a = \frac{i_2 - i_1}{x_2 - x_1} \wedge b = i_2 - x_2 a\right) \\ &= \frac{1}{p}\frac{1}{p} \end{aligned}$$

since a and b are chosen independently and uniformly at random.

Relation with pairwise independent random variables. Observe that a family of 2-wise independent hash functions lets us define a family of $|A|$ pairwise independent random variables as follows. The underlying probability space Ω is taken to be the family of hash functions $\mathcal{H}$, and for every $x \in A$ we have a random variable X such that $X(h) = h(x)$. When we study derandomization it will be important to construct many pairwise independent random variables using the fewest possible random bits. Here we have $|A| = p$ random variables, but the number of random bits is only what is required to choose a random $h \in \mathcal{H}$, so about $2 \log p$ bits. Compare this to $\log p$ bits needed to choose just *one* random value in $\{0, \ldots, p-1\}$!

1.3.3 Estimating the number of distinct elements

$m_0 = \sum f_i^0$ is the number of distinct elements in the stream. We modify the algorithm we saw earlier: instead of using a completely random function f we use a random function $h : \{1, \ldots, n\} \to \{1, \ldots, n\}$ that is taken from a family $\mathcal{H}$ of pairwise independent hash functions. Here is the algorithm:

1. **Initialization:** Choose a random $h \in \mathcal{H}$. Set $z = 0$.

2. **Process a_j:** Let ℓ be the largest power of 2 that divides $h(a_j)$. If $\ell > z$ then $z \leftarrow \ell$.

3. **Output:** $2^{z+\frac{1}{2}}$.

First note that this time the space is truly $O(\log n)$: we need $\log n$ bits to store z, and $2\log n$ bits to store h using the construction we saw earlier. Now let's analyze the success probability of the algorithm.

For each $j \in \{1, \ldots, n\}$ and $r \geq 0$ let $X_{r,j}$ be the indicator random variable for the event that $h(j)$ is divisible by 2^r, and $Y_r = \sum_{j:f_j>0} X_{r,j}$. Let t be the final value of z at the end of the algorithm. Note that $Y_r = 0$ means no element had r or more zeros, thus $t \leq r-1$. Using that $h(j)$ is uniformly distributed,

$$\mathbf{E}[X_{r,j}] = \mathbf{Pr}\left(2^r \text{ divides } h(j)\right) = \frac{1}{2^r}.$$

By linearity of expectation, $\mathbf{E}[Y_r] = m_0/2^r$, and using pairwise independence

$$\mathbf{Var}[Y_r] = \sum_{j:f_j>0} \mathbf{Var}[X_{r,j}] \leq \sum_{j:f_j>0} \mathbf{E}[X_{r,j}^2] = \sum_{j:f_j>0} \mathbf{E}[X_{r,j}] = \frac{m_0}{2^r}.$$

Using first Markov and then Chebyshev,

$$\mathbf{Pr}\left(Y_r > 0\right) = \mathbf{Pr}(Y_r \geq 1) \leq \frac{\mathbf{E}[Y_r]}{1} = \frac{m_0}{2^r},$$
$$\mathbf{Pr}(Y_r = 0) \leq \mathbf{Pr}(|Y_r - \mathbf{E}[Y_r]| \geq \frac{m_0}{2^r}) \leq \frac{\mathbf{Var}[Y_r]}{(m_0/2^r)^2} \leq \frac{2^r}{m_0}.$$

Let $\hat{m}_0 = 2^{z+\frac{1}{2}}$ be the output of the algorithm, and a and b the smallest and largest integer such that $2^{a+\frac{1}{2}} \geq 3m_0$ and $2^{b+\frac{1}{2}} \leq m_0/3$ respectively. Then

$$\mathbf{Pr}(\hat{m}_0 \geq 3m_0) = \mathbf{Pr}(z \geq a) = \mathbf{Pr}(Y_a > 0) \leq \frac{m_0}{2^a} \leq \frac{\sqrt{2}}{3},$$
$$\mathbf{Pr}(\hat{m}_0 \leq m_0/3) = \mathbf{Pr}(z \leq b) = \mathbf{Pr}(Y_{b+1} = 0) \leq \frac{2^{b+1}}{m_0} \leq \frac{\sqrt{2}}{3}.$$

So we have obtained a factor 3-approximation to m_0 that is correct with probability $1 - 2\sqrt{2}/3$, by the union bound. The space requirement is only what is needed to store the hash function, $2\log n$ bits, and the value of z, an extra $\log n$ bits. But the success probability is pretty low, about 6%! However it is possible to boost this very quickly: for any ε, δ we can reduce the error to $\pm\varepsilon m_0$ and boost the success probability to $1-\delta$ using only $O(\varepsilon^{-2}\log(1/\delta))$ independent repetitions. We will see how to do this in the next lecture.