Contents

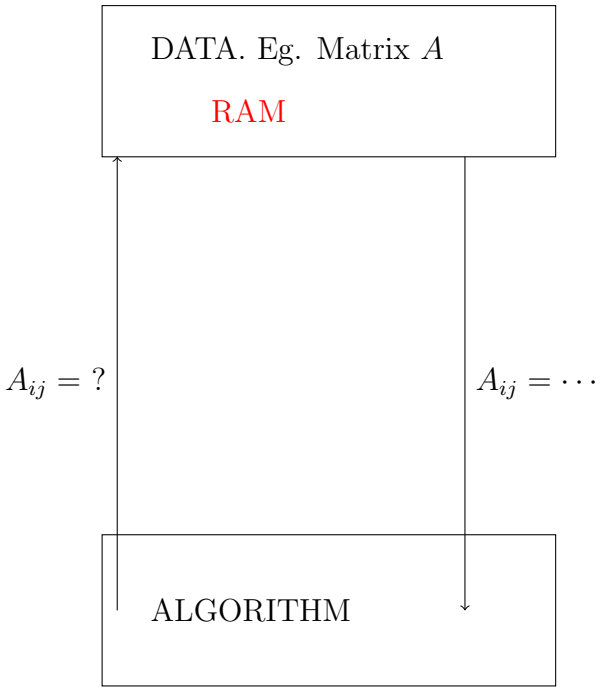

Figure 1.1: Traditional Algorithm Model

1 Algorithms for Massive Data Problems

Massive Data, Sampling

This chapter deals with massive data problems where the input data (a graph, a matrix or some other object) is too large to be stored in random access memory. One model for such problems is the streaming model, where the data can be seen only once. In the streaming model, the natural technique to deal with the massive data is sampling. Sampling is done "on the fly". As each piece of data is seen, based on a coin toss, one decides whether to include the data in the sample. Typically, the probability of including the data point in the sample may depend on its value. Models allowing multiple passes through the data are also useful; but the number of passes needs to be small. We always assume that random access memory (RAM) is limited, so the entire data cannot be stored in RAM.

To introduce the basic flavor of sampling on the fly, consider the following simple primitive. We have a stream of n elements, where, n is not known at the start. We wish to sample uniformly at random one of the n elements. If we have to pick a sample and then never change it, we are out of luck. [You can convince yourself of this.] So we have to allow ourselves to "change our mind": After perhaps drawing one element as a sample, we must allow ourselves to reject it and instead draw the current element as the new sample. With this hint, the reader may want to think of the algorithm. Instead of describing the

algorithm for this problem, we will solve a more general problem:

From a stream of n positive real numbers $a_1, a_2, \ldots, a_n$, draw a sample element a_i so that the probability of picking an element is proportional to its value.

It is easy to see that the following sampling method works. Upon seeing $a_1, a_2, \ldots, a_i$, keep track of the sum $a = a_1 + a_2 + \cdots + a_i$ and a sample a_j, $j \leq i$, drawn with probability proportional to its value. On seeing a_{i+1}, replace the current sample by a_{i+1} with probability $\frac{a_{i+1}}{a+a_{i+1}}$ and update a.

We can prove by induction that this algorithm does in fact sample element a_j with probability $a_j/(a_1 + a_2 + \cdots + a_n)$: Suppose after we have read $a_1, a_2, \ldots, a_i$, we have picked a sample with probability of a_j being the sample equal to $a_j/(a_1 + a_2 + \cdots + a_i)$. The probability that we will keep this a_j after reading a_{i+1} is

$$1 - \frac{a_{i+1}}{a_1 + a_2 + \cdots + a_{i+1}} = \frac{a_1 + a_2 + \cdots + a_i}{a_1 + a_2 + \cdots + a_{i+1}}.$$

Combining the two, we see that the probability that a_j is the sample after reading a_{i+1} is precisely $a_j/(a_1 + a_2 + \cdots + a_{i+1})$ completing the inductive proof.

1.1 Frequency Moments of Data Streams

An important class of problems concerns the frequency moments of data streams. Here a data stream $a_1, a_2, \ldots, a_n$ of length n consists of symbols a_i from an alphabet of m possible symbols which for convenience we denote as $\{1, 2, \ldots, m\}$. Throughout this section, n, m, and a_i will have these meanings and s (for symbol) will denote a generic element of $\{1, 2, \ldots, m\}$. The frequency f_s of the symbol s is the number of occurrences of s in the stream. For a nonnegative integer p, the p^{th} frequency moment of the stream is

$$\sum_{s=1}^{m} f_s^p.$$

Note that the $p = 0$ frequency moment corresponds to the number of distinct symbols occurring in the stream. The first frequency moment is just n, the length of the string. The second frequency moment, $\sum_s f_s^2$, is useful in computing the variance of the stream.

$$\frac{1}{m}\sum_{s=1}^{m}\left(f_s - \frac{n}{m}\right)^2 = \frac{1}{m}\sum_{s=1}^{m}\left(f_s^2 - 2\frac{n}{m}f_s + \left(\frac{n}{m}\right)^2\right) = \frac{1}{m}\sum_{s=1}^{m} f_s^2 - \frac{n^2}{m^2}$$

In the limit as p becomes large, $\left(\sum_{s=1}^{m} f_s^p\right)^{1/p}$ is the frequency of the most frequent element(s).

We will describe sampling based algorithms to compute these quantities for streaming data shortly. But first a note on the motivation for these various problems. The identity and frequency of the the most frequent item or more generally, items whose frequency

exceeds a fraction of n, is clearly important in many applications. If the items are packets on a network with source destination addresses, the high frequency items identify the heavy bandwidth users. If the data is purchase records in a supermarket, the high frequency items are the best-selling items. Determining the number of distinct symbols is the abstract version of determining such things as the number of accounts, web users, or credit card holders. The second moment and variance are useful in networking as well as in database and other applications. Large amounts of network log data are generated by routers that can record for all the messages passing through them, the source address, destination address, and the number of packets. This massive data cannot be easily sorted or aggregated into totals for each source/destination. But it is important to know if some popular source-destination pairs have a lot of traffic for which the variance is the natural measure.

1.1.1 Number of Distinct Elements in a Data Stream

Consider a sequence $a_1, a_2, \ldots, a_n$ of n elements, each a_i an integer in the range 1 to m where n and m are very large. Suppose we wish to determine the number of distinct a_i in the sequence. Each a_i might represent a credit card number extracted from a sequence of credit card transactions and we wish to determine how many distinct credit card accounts there are. The model is a data stream where symbols are seen one at a time. We first show that any deterministic algorithm that determines the number of distinct elements exactly must use at least m bits of memory.

Lower bound on memory for exact deterministic algorithm

Suppose we have seen the first k symbols of the stream and $k > m$. The set of distinct symbols seen so far could be any of the 2^m subsets of $\{1, 2, \ldots, m\}$. Each subset must result in a different state for our algorithm and hence m bits of memory are required. To see this, suppose first that two different size subsets of distinct symbols lead to the same internal state. Then our algorithm would produce the same count of distinct symbols for both inputs, clearly an error for one of the input sequences. If two sequences with the same number of distinct elements but different subsets lead to the same state, then on next seeing a symbol that appeared in one sequence but not the other, we would make an error on at least one of them .

Algorithm for the Number of distinct elements

Let $a_1, a_2, \ldots, a_n$ be a sequence of elements where each $a_i \in \{1, 2, \ldots, m\}$. The number of distinct elements can be estimated with $O(\log m)$ space. Let $S \subseteq \{1, 2, \ldots, m\}$ be the set of elements that appear in the sequence. Suppose that the elements of S were selected uniformly at random from $\{1, 2, \ldots, m\}$. Let *min* denote the minimum element of S. Knowing the minimum element of S allows us to estimate the size of S. The elements of S partition the set $\{1, 2, \ldots, m\}$ into $|S| + 1$ subsets each of size approximately $\frac{m}{|S|+1}$. See

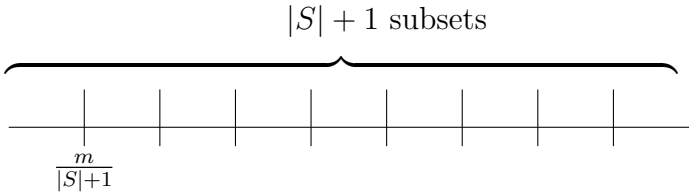

Figure 1.2: Estimating the size of S from the minimum element in S which has value approximately $\frac{m}{|S|+1}$. The elements of S partition the set $\{1, 2, \ldots, m\}$ into $|S|+1$ subsets each of size approximately $\frac{m}{|S|+1}$.

Figure 1.2. Thus, the minimum element of S should have value close to $\frac{m}{|S|+1}$. Solving $min = \frac{m}{|S|+1}$ yields $|S| = \frac{m}{min} - 1$. Since we can determine min, this gives us an estimate of $|S|$.

The above analysis required that the elements of S were picked uniformly at random from $\{1, 2, \ldots, m\}$. This is generally not the case when we have a sequence $a_1, a_2, \ldots, a_n$ of elements from $\{1, 2, \ldots, m\}$. Clearly if the elements of S were obtained by selecting the $|S|$ smallest elements of $\{1, 2, \ldots, m\}$, the above technique would give the wrong answer. If the elements are not picked uniformly at random, can we estimate the number of distinct elements? The way to solve this problem is to use a hash function h where

$$h : \{1, 2, \ldots, m\} \to \{0, 1, 2, \ldots, M-1\}$$

To count the number of distinct elements in the input, count the number of elements in the mapped set $\{h(a_1), h(a_2), \ldots\}$. The point being that $\{h(a_1), h(a_2), \ldots\}$ behaves like a random subset and so the above heuristic argument using the minimum to estimate the number of elements may apply. If we needed $h(1), h(2), \ldots$ to be completely independent, the space needed to store the hash function would too high. Fortunately, only 2-way independence is needed. We recall the formal definition of 2-way independence below. But first recall that a hash function is always chosen at random from a family of hash functions and phrases like "probability of collision" refer to the probability over the choice of hash function.

Universal Hash Functions

The set of hash functions

$$H = \{h \mid h : \{1, 2, \ldots, m\} \to \{0, 1, 2, \ldots, M-1\}\}$$

is *2-universal* if for all x and y in $\{1, 2, \ldots, m\}$, $x \neq y$, and for all z and w in $\{0, 1, 2, \ldots, M-1\}$

$$\text{Prob}[h(x) = z \text{ and } h(y) = w] = \frac{1}{M^2}$$

for a randomly chosen h. The concept of a 2-universal family of hash functions is that given x, $h(x)$ is equally likely to be any element of $\{0, 1, 2, \ldots, M-1\}$ (the reader should prove this from the definition of 2-universal) and for $x \neq y$, $h(x)$ and $h(y)$ are independent.

We now give an example of a 2-universal family of hash functions. For simplicity let M be a prime, with $M > m$. For each pair of integers a and b in the range [0,M-1], define a hash function

$$h_{ab}(x) = ax + b \mod (M)$$

To store the hash function h_{ab}, store the two integers a and b. This requires only $O(\log M)$ space. To see that the family is 2-universal note that $h(x) = z$ and $h(y) = w$ if and only if

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} z \\ w \end{pmatrix} \mod (M)$$

If $x \neq y$, the matrix $\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$ is invertible modulo M and there is only one solution for a and b. Thus, for a and b chosen uniformly at random, the probability of the equation holding is exactly $\frac{1}{M^2}$. [We assumed $M > m$. What goes wrong if this did not hold?]

Analysis of distinct element counting algorithm

Let $b_1, b_2, \ldots, b_d$ be the distinct values that appear in the input. Then $S = \{h(b_1), h(b_2), \ldots h(b_d)\}$ is a set of d random and 2-way independent values from the set $\{0, 1, 2, \ldots, M-1\}$. We now show that $\frac{M}{\min}$ is a good estimate for d, the number of distinct elements in the input, where min=min(S).

Lemma 1.1 *Assume $M > 100d$. With probability at least $\frac{2}{3}$, $\frac{d}{6} \leq \frac{M}{\min} \leq 6d$, where min is the smallest element of S.*

Proof: First, we show that Prob $\left[\frac{M}{\min} > 6d\right] < \frac{1}{6}$.

$$\text{Prob}\left[\frac{M}{\min} > 6d\right] = \text{Prob}\left[\min < \frac{M}{6d}\right] = \text{Prob}\left[\exists k \; h(b_k) < \frac{M}{6d}\right]$$

For $i = 1, 2, \ldots, d$ define the indicator variable

$$z_i = \begin{cases} 1 & \text{if h}(b_i) < \frac{M}{6d} \\ 0 & \text{otherwise} \end{cases}$$

and let $z = \sum_{i=1}^{d} z_i$. If $h(b_i)$ is chosen randomly from $\{0, 1, 2, \ldots, M-1\}$, then Prob $[z_i = 1] =$

$\frac{1}{6d}$. Thus, $E(z_i) = \frac{1}{6d}$ and $E(z) = \frac{1}{6}$. Now

$$\begin{aligned}\text{Prob}\left[\frac{M}{\min} > 6d\right] &= \text{Prob}\left[min < \frac{M}{6d}\right] \\ &= \text{Prob}\left[\exists k\ h(b_k) < \frac{M}{6d}\right] \\ &= \text{Prob}(z \geq 1) = \text{Prob}[z \geq 6E(z)].\end{aligned}$$

By Markov's inequality $\text{Prob}[z \geq 6E(z)] \leq \frac{1}{6}$.

Finally, we show that $\text{Prob}\left[\frac{M}{\min} < \frac{d}{6}\right] < \frac{1}{6}$.

$$\text{Prob}\left[\frac{M}{\min} < \frac{d}{6}\right] = \text{Prob}\left[\min > \frac{6M}{d}\right] = \text{Prob}\left[\forall k\ h(b_k) > \frac{6M}{d}\right]$$

For $i = 1, 2, \ldots, d$ define the indicator variable

$$y_i = \begin{cases} 0 & \text{if h}(b_i) > \frac{6M}{d} \\ 1 & \text{otherwise} \end{cases}$$

and let $y = \sum_{i=1}^{d} y_i$. Now $\text{Prob}(y_i = 1) = \frac{6}{d}$, $E(y_i) = \frac{6}{d}$, and $E(y) = 6$. For 2-way independent random variables, the variance of their sum is the sum of their variances. So $Var(y) = dVar(y_1)$. Further, it is easy to see since y_1 is 0 or 1 that

$$Var(y_1) = E\left[(y_1 - E(y_1))^2\right] = E(y_1^2) - E^2(y_1) = E(y_1) - E^2(y_1) \leq E(y_1).$$

Thus $Var(y) \leq E(y)$. Now by the Chebychev inequality,

$$\begin{aligned}\text{Prob}\left[\frac{M}{\min} < \frac{d}{6}\right] &= \text{Prob}\left[\min > \frac{6M}{d}\right] = \text{Prob}\left[\forall k\ h(b_i) > \frac{6M}{d}\right] \\ &= \text{Prob}(y = 0) \leq \text{Prob}[|y - E(y)| \geq E(y)] \\ &\leq \frac{\text{Var}(y)}{E^2(y)} \leq \frac{1}{E(y)} \leq \frac{1}{6}\end{aligned}$$

Since $\frac{M}{\min} > 6d$ with probability at most $\frac{1}{6}$ and $\frac{M}{\min} < \frac{d}{6}$ with probability at most $\frac{1}{6}$, $\frac{d}{6} \leq \frac{M}{\min} \leq 6d$ with probability at least $\frac{2}{3}$. ∎

1.1.2 Counting the Number of Occurrences of a Given Element.

To count the number of occurrences of an element in a stream requires at most $\log n$ space where n is the length of the stream. Clearly, for any length stream that occurs in practice, we can afford $\log n$ space. For this reason, the following material may never be used in practice, but the technique is interesting and may give insight into how to solve

some other problem.

Consider a string of 0's and 1's of length n in which we wish to count the number of occurrences of 1's. Clearly if we had $\log n$ bits of memory we could keep track of the exact number of 1's. However, we can approximate the number with only $\log \log m$ bits.

Let m be the number of 1's that occur in the sequence. Keep a value k such that 2^k is approximately the number of occurrences m. Storing k requires only $\log \log n$ bits of memory. The algorithm works as follows. Start with k=0. For each occurrence of a 1, add one to k with probability $1/2^k$. At the end of the string, the quantity $2^k - 1$ is the estimate of m. To obtain a coin that comes down heads with probability $1/2^k$, flip a fair coin, one that comes down heads with probability $1/2$, k times and report heads if the fair coin comes down heads in all k flips.

Given k, on average it will take 2^k ones before k is incremented. Thus, the expected number of 1's to produce the current value of k is $1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$.

1.1.3 Counting Frequent Elements

The Majority and Frequent Algorithms

First consider the very simple problem of n people voting. There are m candidates, $\{1, 2, \ldots, m\}$. We want to determine if one candidate gets a majority vote and if so who. Formally, we are given a stream of integers $a_1, a_2, \ldots, a_n$, each a_i belonging to $\{1, 2, \ldots, m\}$, and want to determine whether there is some $s \in \{1, 2, \ldots, m\}$ which occurs more than $n/2$ times and if so which s. It is easy to see that to solve the problem exactly on read only once streaming data with a deterministic algorithm, requires $\Omega(n)$ space. Suppose n is even and the first $n/2$ items are all distinct and the last $n/2$ items are identical. After reading the first $n/2$ items, we need to remember exactly which elements of $\{1, 2, \ldots, m\}$ have occurred. If for two different sets of elements occurring in the first half of the stream, the contents of the memory are the same, then a mistake would occur if the second half of the stream consists solely of an element in one set, but not the other. Thus, $\log_2 \binom{m}{n/2}$ bits of memory, which if $m > n$ is $\Omega(n)$, are needed.

Now lets allow the algorithm a random number generator. The reader may want to think about simple sampling schemes first - like picking an element as a sample and then checking how many times it occurs and modifications of this. The following is a simple low-space algorithm that always finds the majority vote if there is one. If there is no majority vote, the output may be arbitrary. That is, there may be "false positives", but no "false negatives".

Majority Algorithm

Store a_1 and initialize a counter to one. For each subsequent a_i, if a_i is the same as the currently stored item, increment the counter by one. If it differs, decrement the counter by one provided the counter is nonzero. If the counter is zero, then store a_i and set the counter to one.

To analyze the algorithm, it is convenient to view the decrement counter step as "eliminating" two items, the new one and the one that caused the last increment in the counter. It is easy to see that if there is a majority element s, it must be stored at the end. If not, each occurrence of s was eliminated; but each such elimination also causes another item to be eliminated and so for a majority item not to be stored at the end, we must have eliminated more than n items, a contradiction.

Next we modify the above algorithm so that not just the majority, but also items with frequency above some threshold are detected. We will also ensure (approximately) that there are no false positives as well as no false negatives. Indeed the algorithm below will find the frequency (number of occurrences) of each element of $\{1, 2, \ldots, m\}$ to within an additive term of $\frac{n}{k+1}$ using $O(k \log n)$ space by keeping k counters instead of just one counter.

Algorithm Frequent

Maintain a list of items being counted. Initially the list is empty. For each item, if it is the same as some item on the list, increment its counter by one. If it differs from all the items on the list, then if there are less than k items on the list, add the item to the list with its counter set to one. If there are already k items on the list decrement each of the current counters by one deleting an element from the list if its count becomes zero.

Theorem 1.2 *At the end of Algorithm Frequent, for each $s \in \{1, 2, \ldots, m\}$, its counter on the list is at least the number of occurrences of s in the stream minus n/(k+1). In particular, if some s does not occur on the list, its counter is zero and the theorem asserts that it occurs fewer than n/(k+1) times in the stream.*

Proof: View each decrement counter step as eliminating some items. An item is eliminated if it is the current a_i being read and there are already k symbols different from it on the list in which case it and k other items are simultaneously eliminated. Thus, the elimination of each occurrence of an $s \in \{1, 2, \ldots, m\}$ is really the elimination of $k + 1$ items. Thus, no more than n/(k+1) occurrences of any symbol can be eliminated. Now, it is clear that if an item is not eliminated, then it must still be on the list at the end. This proves the theorem. ∎

1.1.4 The Second Moment

This section focuses on computing the second moment of a stream with symbols from $\{1, 2, \ldots, m\}$. Let f_s denote the number of occurrences of symbol s in the stream. The

second moment of the stream is given by $\sum_{s=1}^{m} f_s^2$. To calculate the second moment, for each symbol s, $1 \leq s \leq m$, independently set a random variable x_s to ± 1 with probability $1/2$. Maintain a sum by adding x_s to the sum each time the symbol s occurs in the stream. At the end of the stream, the sum will equal $\sum_{s=1}^{m} x_s f_s$. The expected value of the sum will be zero where the expectation is over the choice of the ± 1 value for the x_s.

$$E\left(\sum_{s=1}^{m} x_s f_s\right) = 0.$$

Although the expected value of the sum is zero, its actual value is a random variable and the expected value of the square of the sum is given by

$$E\left(\sum_{s=1}^{m} x_s f_s\right)^2 = E\left(\sum_{s=1}^{m} x_s^2 f_s^2\right) + 2E\left(\sum_{s \neq t} x_s x_t f_s f_t\right) = \sum_{s=1}^{m} f_s^2,$$

The last equality follows since $E(x_s x_t) = 0$ for $s \neq t$. Thus

$$a = \left(\sum_{s=1}^{m} x_s f_s\right)^2$$

is an estimator of $\sum_{s=1}^{m} f_s^2$. One difficulty which we will come back to is that to store the x_i requires space m and we want to do the calculation in $\log m$ space.

A second issue is that this estimator is depends on its variance which we now compute.

$$Var(a) \leq E\left(\sum_{s=1}^{m} x_s f_s\right)^4 = E\left(\sum_{1 \leq s,t,u,v \leq m} x_s x_t x_u x_v f_s f_t f_u f_v\right)$$

The first inequality is because the variance is at most the second moment and the second equality is by expansion. In the second sum, since the x_s are independent, if any one of s, u, t, or v is distinct from the others, then the expectation of the whole term is zero. Thus, we need to deal only with terms of the form $x_s^2 x_t^2$ for $t \neq s$ and terms of the form x_s^4. Note that this does not need the full power of mutual independence of all the x_s, it only needs 4-way independence, that any four of the x_s's are mutually independent. In the above sum, there are four indices s, t, u, v and there are $\binom{4}{2}$ ways of choosing two of

them that have the same x value. Thus,

$$\begin{aligned} Var\,(a) &\leq \binom{4}{2} E\left(\sum_{s=1}^{m}\sum_{t=s+1}^{m} x_s^2 x_t^2 f_s^2 f_t^2\right) + E\left(\sum_{s=1}^{m} x_s^4 f_s^4\right) \\ &= 6\sum_{s=1}^{m}\sum_{t=s+1}^{m} f_s^2 f_t^2 + \sum_{s=1}^{m} f_s^4 \\ &\leq 3\left(\sum_{s=1}^{m} f_s^2\right)^2 + \left(\sum_{s=1}^{m} f_s^2\right)^2 = 4E^2\,(a)\,. \end{aligned}$$

The variance can be reduced by a factor of r by taking the average of r independent trials. With r independent trials the variance would be at most $\frac{4}{r}E^2\,(a)$, so to achieve relative error ε in the estimate of $\sum_{s=1}^{m} f_s^2$, we need $O(1/\varepsilon^2)$ independent trials.

We will briefly discuss the independent trials here, so as to understand exactly the amount of independence needed. Instead of computing a using the running sum $\sum_{s=1}^{m} x_s f_s$ for one random vector $\mathbf{x}$, we independently generate r m-vectors $(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_r})$ at the outset and compute r running sums

$$\sum_{s=1}^{m} x_{1s} f_s\,,\ \sum_{s=1}^{m} x_{2s} f_s\,,\ \ldots,\ \sum_{s=1}^{m} x_{rs} f_s.$$

Let $a_1 = \left(\sum_{s=1}^{m} x_{1s} f_s\right)^2$, $a_2 = \left(\sum_{s=1}^{m} x_{2s} f_s\right)^2, \ldots,$ $a_r = \left(\sum_{s=1}^{m} x_{rs} f_s\right)^2$. Our estimate is $\frac{1}{r}(a_1 + a_2 + \cdots + a_r)$. The variance of this estimator is

$$\mathrm{Var}\left[\tfrac{1}{r}\,(a_1 + a_2 + \cdots + a_r)\right] = \tfrac{1}{r^2}\left[\mathrm{Var}\,(\mathrm{a}_1) + \mathrm{Var}\,(\mathrm{a}_2) + \cdots + \mathrm{Var}\,(\mathrm{a}_r)\right] = \tfrac{1}{r}\mathrm{Var}(a_1),$$

where we have assumed that the $a_1, a_2, \ldots, a_r$ are mutually independent. Now we compute the variance of a_1 as we have done for the variance of a. Note that this calculation assumes only 4-way independence between the coordinates of x_1. We summarize the assumptions here for future reference:

To get an estimate of $\sum_{s=1}^{m} f_s^2$ within relative error ε with probability close to one, say at least 0.9999, it suffices to have $r = O(1/\varepsilon^2)$ vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_r}$, each with m coordinates of ± 1 with

1. $E\,(\mathbf{x_s}) = 0$ for all s.
2. $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_r}$ are mutually independent. That is for any r vectors $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_r}$ with ± 1 coordinates, $\mathrm{Prob}(\mathbf{x_1} = \mathbf{v_1},\ \mathbf{x_2} = \mathbf{v_2},\ \ldots,\ \mathbf{x_r} = \mathbf{v_r}) = \mathrm{Prob}(\mathbf{x_1} = \mathbf{v_1})\mathrm{Prob}(\mathbf{x_2} = \mathbf{v_2}) \cdots \mathrm{Prob}(\mathbf{x_r} = \mathbf{v_r})$.

3. Any four coordinates of $\mathbf{x_1}$ are independent. Same for $\mathbf{x_2}, \mathbf{x_3}, \ldots,$ and $\mathbf{x_r}$.

[Caution: (2) does not assume that $\text{Prob}(\mathbf{x_1} = \mathbf{v_1}) = \frac{1}{2^m}$ for all $\mathbf{v_1}$; such an assumption would mean the coordinates of $\mathbf{x_1}$ are mutually independent.] The only drawback with the algorithm as we have described it so far is that we need to keep the r vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_r}$ in memory so that we can do the running sums. This is too space-expensive. We need to do the problem in space dependent upon the logarithm of the size of the alphabet m, not m itself. If ε is in $\Omega(1)$, then r is in $O(1)$, so it is not the number of trials r which is the problem. It is the m.

In the next section, we will see that the computation can be done in $O(\log m)$ space by using pseudo-random vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_r}$ instead of truly random ones. The pseudo-random vectors will satisfy (i), (ii), and (iii) and so they will suffice. This pseudo-randomness and limited independence has deep connections, so we will go into the connections as well.

Error-Correcting codes, polynomial interpolation and limited-way independence

Consider the problem of generating a random m-vector $\mathbf{x}$ of ± 1's so that any subset of four coordinates is mutually independent, i.e., for any distinct s, t, u, and v in $\{1, 2, \ldots, m\}$ and any a, b, c, and d in {-1, +1},

$$\text{Prob}(x_s = a, \;\; x_t = b, \;\; x_u = c, \;\; x_v = d) = \tfrac{1}{16}.$$

We will see that such an n-dimensional vector may be generated from a truly random "seed" of only $O(\log m)$ mutually independent bits. Thus, we need only store the $O(\log m)$ bits and can generate any of the m coordinates when needed. This allows us to store the 4-way independent random m-vector using only $\log m$ bits. The first fact needed for this is that for any k, there is a finite field F with exactly 2^k elements, each of which can be represented with k bits and arithmetic operations in the field can be carried out in $O(k^2)$ time. [In fact F can be taken to be the set of polynomials of degree $k-1$ with mod 2 coefficients, where, multiplication is done modulo an irreducible polynomial of degree k, again over modulo 2.] Here, k will be the ceiling of $\log_2 m$. We also assume another basic fact about polynomial interpolation which says that a polynomial of degree at most three is uniquely determined by its value over any field F at four points. More precisely, for any four distinct points $a_1, a_2, a_3, a_4 \in F$ and any four possibly not distinct values $b_1, b_2, b_3, b_4 \in F$, there is a unique polynomial $f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3$ of degree at most three that with computations done over F, $f(a_1) = b_1, f(a_2) = b_2, f(a_3) = b_3$, and $f(a_4) = b_4$.

Now our definition of the pseudo-random $\pm$ vector $\mathbf{x}$ with 4-way independence is simple. Choose four elements f_0, f_1, f_2, f_3 at random from F and form the polynomial $f(s) = f_0 + f_1 s + f_2 s^2 + f_3 s^3$. This polynomial represents $\mathbf{x}$ as follows. For $s = 1, 2, \ldots, m$,

x_s is the leading bit of the k-bit representation of $f(s)$. Thus, the m-dimensional vector $\mathbf{x}$ can be computed using only the $4k$ bits in f_0, f_1, f_2, f_3. (Here $k = \lceil \log m \rceil$).

Lemma 1.3 *The x defined above have 4-way independence.*

Proof: Assume that the elements of F are represented in binary using $\pm$ instead of the traditional 0 and 1. Let s, t, u, and v be any four coordinates of $\mathbf{x}$ and let $\alpha, \beta, \gamma, \delta \in \{-1, 1\}$. There are exactly 2^{k-1} elements of F whose leading bit is α and similarly for β, γ, and δ. So, there are exactly $2^{4(k-1)}$ 4-tuples of elements $b_1,\ b_2,\ b_3,\ b_4 \in F$ so that the leading bit of b_1 is α, the leading bit of b_2 is β, the leading bit of b_3 is γ, and the leading bit of b_4 is δ. For each such $b_1,\ b_2,\ b_3$, and b_4, there is precisely one polynomial f so that

$$f(s) = b_1,\ f(t) = b_2,\ f(u) = b_3, \text{ and } f(v) = b_4$$

as we saw above. So, the probability that

$$x_s = \alpha,\ x_t = \beta,\ x_u = \gamma, \text{ and } x_v = \delta$$

is precisely $\frac{2^{4(k-1)}}{\text{total number of } f} = \frac{2^{4(k-1)}}{2^{4k}} = \frac{1}{16}$ as asserted. ∎

The lemma states how to get one vector $\mathbf{x}$ with 4-way independence. However, we need $r = O(1/\varepsilon^2)$ vectors. Also the vectors must be mutually independent. But this is easy, just choose r polynomials at the outset.

To implement the algorithm with low space, store only the polynomials in memory. This requires $4k = O(\log m)$ bits per polynomial for a total of $O(\log m/\varepsilon^2)$ bits. When a symbol s in the stream is read, compute $x_{1s},\ x_{2s}, \ldots, x_{rs}$ and update the running sums. Note that x_{s1} is just the leading bit of the first polynomial evaluated at s; this calculation is in $O(\log m)$ time. Thus, we repeatedly compute the x_s from the "seeds", namely the coefficients of the polynomials.

This idea of polynomial interpolation is also used in other contexts. Error-correcting codes is an important example. Say we wish to transmit n bits over a channel which may introduce noise. One can introduce redundancy into the transmission so that some channel errors can be corrected. A simple way to do this is to view the n bits to be transmitted as coefficients of a polynomial $f(x)$ of degree $n-1$. Now transmit f evaluated at points $1, 2, 3, \ldots, n+m$. At the receiving end, any n correct values will suffice to reconstruct the polynomial and the true message. So up to m errors can be tolerated. But even if the number of errors is at most m, it is not a simple matter to know which values are corrupted. We do not elaborate on this here.

1.2 Sketch of a Large Matrix

We will see how to find a good approximation of an $m \times n$ matrix A, which is read from external memory, where m and n are large. The approximation consists of a sample of s columns and r rows of A along with an $s \times r$ multiplier matrix. The schematic diagram is given in Figure 1.3.

The crucial point is that uniform sampling will not always work as is seen from simple examples. We will see that if we sample rows or columns with probabilities proportional to the squared length of the row or column, then indeed we can get a good approximation. One may recall that the top k singular vectors of the SVD of A, give a similar picture; but the SVD takes more time to compute, requires all of A to be stored in RAM, and does not have the property that the rows and columns are directly from A. However, the SVD does yield the best approximation. Error bounds for our approximation are weaker, though it can be found faster.

We briefly touch upon two motivations for such a sketch. Suppose A is the document-term matrix of a large collection of documents. We are to "read" the collection at the outset and store a sketch so that later, when a query (represented by a vector with one entry per term) arrives, we can find its similarity to each document in the collection. Similarity is defined by the dot product. In Figure 1.3 it is clear that the matrix-vector product of a query with the right hand side can be done in time $O(ns + sr + rm)$ which would be linear in n and m if s and r are $O(1)$. To bound errors for this process, we need to show that the difference between A and the sketch of A has small 2-norm. Recall that the 2-norm $||A||_2$ of a matrix A is $\max_{|\mathbf{x}|=1} \mathbf{x}^T A \mathbf{x}$.

A second motivation comes from recommendation systems. Here A would be a customer-product matrix whose $(i, j)^{th}$ entry is the preference of customer i for product j. The objective is to collect a few sample entries of A and based on them, get an approximation to A so that we can make future recommendations. These results say that a few sampled rows of A (all preferences of a few customers) and a few sampled columns (all customers' preferences for a few products) give a good approximation to A provided that the samples are drawn according to the length-squared distribution.

We first tackle a simpler problem, that of multiplying two matrices. The matrix multiplication also uses length squared sampling and is a building block to the sketch.

1.2.1 Matrix Multiplication Using Sampling

Suppose A is an $m \times n$ matrix and B is an $n \times p$ matrix and the product AB is desired. We show how to use sampling to get an approximate product faster than the traditional multiplication. Let $A(:, k)$ denote the k^{th} column of A. $A(:, k)$ is a $m \times 1$ matrix. Let

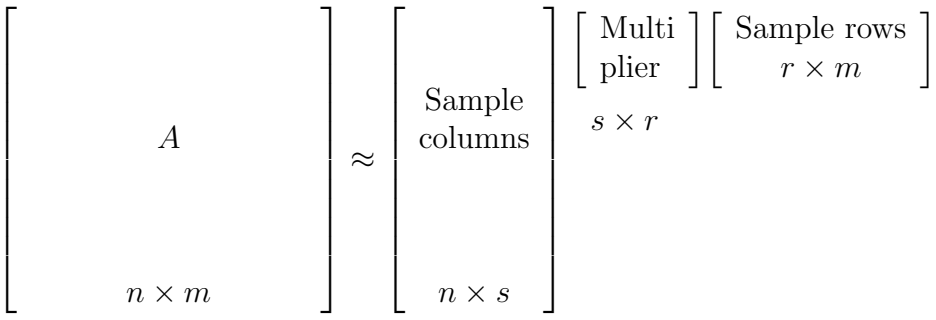

Figure 1.3: Schematic diagram of the approximation of A by a sample of s columns and r rows.

$B(k,:)$ be the k^{th} row of B. $B(k,:)$ is a $1 \times n$ matrix. It is easy to see that

$$AB = \sum_{k=1}^{n} A(:,k)B(k,:) = \sum_{k=1}^{n} \text{ Outer Product of } k \text{ th column of } A \text{ and the } k \text{ th row of } B.$$

[This is easy to verify since, the (i,j) th entry of the outer product of the k th column of A and the k th row of B is just $a_{ik}b_{kj}$.] Note that for each value of k, $A(:,k)B(k,:)$ is an $m \times p$ matrix each element of which is a single product of elements of A and B. An obvious use of sampling suggests itself. Sample some values for k and compute $A(:,k)\,B(k,:)$ for the sampled k's using their suitably scaled sum as the estimate of AB. It turns out that nonuniform sampling probabilities are useful. Define a random variable z that takes on values in $\{1,2,\ldots,n\}$. Let p_k denote the probability that z assumes the value k. The p_k are nonnegative and sum to one. Define an associated random matrix variable that has value

$$X = \frac{1}{p_z} A(:,z)\,B(z,:)$$

which takes on value $\frac{1}{p_k}A(:,k)B(k,:)$ with probability p_k for $k = 1,2,\ldots,n$. Let $E(X)$ denote the entry-wise expectation.

$$E(X) = \sum_{k=1}^{n} \text{Prob}(z=k)\frac{1}{p_k} A(:,k)\,B(k,:) = \sum_{k=1}^{n} A(:,k)B(k,:) = AB.$$

This explains the scaling by $\frac{1}{p_z}$ in X. [In general, if we wish to estimate the sum of n real numbers $a_1, a_2, \ldots, a_n$ by drawing a sample z from $\{1,2,\ldots,n\}$ with probabilities $p_1, p_2, \ldots, p_n$, then the unbiased estimator of the sum $a_1 + a_2 + \cdots + a_n$ is a_z/p_z.]

We wish to bound the error in this estimate for which the usual quantity of interest is the variance. But here each entry x_{ij} of the matrix random variable may have a different variance. So, we define the variance of X as the sum of the variances of all its entries. This

natural device of just taking the sum of the variances greatly simplifies the calculations as we will see.

$$\mathrm{Var}(X) = \sum_{i=1}^{m}\sum_{j=1}^{p}\mathrm{Var}\left(x_{ij}\right) = \sum_{ij} E\left(x_{ij}^2\right) - \sum_{ij}(E(x_{ij})^2) = \sum_{i,j}\sum_{k} p_k \frac{1}{p_k^2} a_{ik}^2 b_{kj}^2 - \sum_{ij}(AB)_{ij}^2.$$

We can simplify by exchanging the order of summations to get

$$\mathrm{Var}(X) = \sum_{k}\frac{1}{p_k}\sum_{i} a_{ik}^2 \sum_{j} b_{kj}^2 - ||AB||_F^2 = \sum_{k}\frac{1}{p_k}|A\left(:,k\right)|^2 |B\left(k,:\right)|^2 - ||AB||_F^2.$$

If p_k is proportional to $|A\left(:,k\right)|^2$, i.e, $p_k = \frac{|A(:,k)|^2}{||A||_F^2}$, we get a simple bound for $\mathrm{Var}(X)$.

$$\mathrm{Var}(X) \leq ||A||_F^2 \sum_{k} |B\left(k,:\right)|^2 = ||A||_F^2 ||B||_F^2.$$

Now, if we do s i.i.d. trials, find s (matrix-valued) random variables $X_1, X_2, \ldots, X_s$ and take their (entry-wise) averages, the variance goes down by a factor of s (as is always the case). It will be useful to give this whole process of estimating the product of two matrices A, B a symbol - $\otimes_s$ which we formally define below.

Algorithm Input A, B $m \times n$ respectively $n \times p$ matrices. Algorithm finds an approximation $A \otimes_s B$ to the matrix product AB.,

- $A \otimes_s B = \frac{1}{s}(X_1 + X_2 + \cdots + X_s)$, where, each X_i is computed independently (of other X_j) by:
  - Draw sample of random variable z from $\{1, 2, \ldots, n\}$ with probabilities proportional to the length squared of the columns of A, i.e.,
    - Prob $(z = k) = |A(:, k)|^2/||A||_F^2$ for $k = 1, 2, \ldots, n$.
  - Set $X_i = \frac{||A||_F^2}{|A(:,z)|^2} A(:, z)B(z, :)$.

Since $\otimes_s$ is a randomized algorithm, it does not give us exactly the same output every time, so $A \otimes_s B$ is not strictly speaking a function of A, B.

Using probabilities that are proportional to length squared of columns of A turns out to be useful in other contexts as well and sampling according to them is called "length-squared sampling".
The Lemma below shows that the error between the correct AB and the estimate $A \otimes_s B$ goes down as s increases. The error is terms of $||A||_F||B||_F$ which is natural.

Lemma 1.4 *Suppose A is an $m \times n$ matrix and B is an $n \times p$ matrix.*

$$E\left(||A \otimes_s B - AB||_F^2\right) \leq \frac{1}{s}||A||_F^2||B||_F^2.$$

Proof: Taking the average of s independent samples divides the variance by s. ∎

The basic step in the algorithm is computing the s outer products, each of which takes time $O(n)$ In addition, we need the time to compute the length-squared of the columns of A (which can be done by reading A once) and drawing the sample.

It is interesting that if A, B come in the correct order, the estimate $A \otimes_s B$ can be found while the data streams.

Lemma 1.5 *If A is in column order and B is in row order, then as the data streams we can draw samples and implement the algorithm $A \otimes_s B$ with $O(s(m + p))$ RAM space in the streaming model.*

Proof: As an extension of our primitive to sample from a stream of positive reals $a_1, a_2, \ldots, a_n$ with probabilities proportional to a_i, we an devise an algorithm to do length-squared sampling. Suppose we have read the first i columns of A and maintain the primitive that we have sampled one of those i columns with probability proportional to length squared. As the $i + 1$ column streams by, we (i) keep it in RAM and (ii) compute its length squared by summing squares of its entries. At the end of reading the column, we can replace the current sampled column with the $i+1$ st column with probability equal to (length squared

of the $i+1$ st column)/(sum of length squared of columns $1, 2, \ldots i+1$), (much as we did earlier with real numbers - details left to the reader), so at the end we draw one length-squared sampled column. We would simultaneously draw s samples. Then we pick the corresponding rows of B (no sampling) and finally carry out the multiplication algorithm $\otimes_s$ with the sample in RAM.

An important special case is the multiplication AA^T where the length-squared distribution can be shown to be the optimal distribution. The variance of X defined above for this case is

$$\sum_{i,j} E(x_{ij}^2) - \sum_{ij} E^2(x_{ij}) = \sum_k \tfrac{1}{p_k} |A(:,k)|^4 - ||AA^T||_F^2.$$

The second term is independent of the p_k. The first term is minimized when the p_k are proportional to $|A(:,k)|^2$, hence length-squared distribution minimizes the variance of this estimator. [Show that for any positive real numbers $a_1, a_2, \ldots, a_n$, the minimum value of $\frac{1}{p_1}a_1^2 + \frac{1}{p_2}a_2^2 + \cdots + \frac{1}{p_n}a_n^2$ over all $p_1, p_2, \ldots, p_n \geq 0$ summing to 1 is attained when $p_k \propto a_k$.]

1.2.2 Approximating a Matrix with a Sample of Rows and Columns

We now return to the problem outlined in the beginning of section (1.2). Suppose an $m \times n$ (m, n are large) matrix A is read from external memory and a sketch of it is to be kept for further computation. Clearly, just a random sample of columns (or rows) will not do, since the sample tells us little about the unsampled columns. A simple sketch idea would be to keep a random sample of rows and a random sample of columns of the matrix. With uniform random sampling, this can fail to be a good sketch. Consider the case where very few rows and columns of A are nonzero. A small sample will miss them, storing only zeros. This toy example can be made more subtle by making a few rows have very high absolute value entries compared to the rest. The sampling probabilities need to depend on the size of the entries. If we do length-squared sampling of both rows and columns, then we can get an approximation to A with a bound on the approximation error. Note that the sampling can be achieved in two passes through the matrix, the first to compute running sums of the length-squared of each row and column and the second pass to draw the sample. Only the sampled rows and columns, which is much smaller than the whole, need to be stored in RAM for further computations. The algorithm starts with:

- Pick s columns of A using length-squared sampling.
- Pick r rows of A using length-squared sampling (here lengths are row lengths).

We will see that from these sampled columns and rows, we can construct a succinct approximation to A. For this, we first start with what seems to be a strange idea.

Observe that $A = AI$, where I is the $n \times n$ identity matrix. Now, lets approximate AI by $A \otimes_s I$. The error is given by Lemma (1.4):

$$||A - A \otimes_s I||_F^2 \leq \frac{1}{s}||A||_F^2||I||_F^2.$$

Our aim would be to make the error at most $\varepsilon||A||_F^2$ for which we will need to make $s > ||I||_F^2/\varepsilon = n/\varepsilon$ which of course is ridiculous since we want $s << n$. But this was caused by the fact that I had large (n) rank. A smaller rank "identity-like" matrix might work better. For this, we will find the SVD of R, where, R is the $r \times n$ matrix consisting of the sampled rows of A.

$$\text{SVD of } R \ : \quad R = \sum_{i=1}^{t} \sigma_i \mathbf{u_i} \mathbf{v_i}^T,$$

where, t =rank(R), so that all the σ_i above are positive.

Claim 1 *The matrix $W = \sum_{i=1}^{t} \mathbf{v_i}\mathbf{v_i}^T$ acts as the identity when applied to any vector $\mathbf{v}$ in the row space of R (from the left). I.e., for such $\mathbf{v}$, $W\mathbf{v} = \mathbf{v}$.*

Proof: Since $\mathbf{v}$ is in the space of $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_t}$, we can write $\mathbf{v} = \sum_{i=1}^{t} \alpha_i \mathbf{v_i}$ and then we get (using the orthogonality of $\mathbf{v_i}$): $W\mathbf{v} = \sum_i \alpha_i \mathbf{v_i} = \mathbf{v}$ as claimed. ∎

We will approximate A by

$$A \approx AW \approx A \otimes_s W.$$

The following Lemma shows that these approximations are valid;

Lemma 1.6 *1. $E\left(||A - AW||_2\right) \leq \frac{1}{\sqrt{r}}||A||_F$.*

2. $E\left(||AW - A \otimes_s W||_2\right) \leq \frac{\sqrt{r}}{\sqrt{s}}||A||_F$.

3. $A \otimes_s W$ can be found from the sampled rows and columns of A.

Proof: $||A - AW||_2 = |(A - AW)\mathbf{x}|$ for some unit length vector $\mathbf{x}$ by definition of $||\cdot||_2$. Let $\mathbf{y}$ be the component of $\mathbf{x}$ in the space spanned by $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_t}$ and let $\mathbf{v} = \mathbf{x} - \mathbf{y}$ be the component of $\mathbf{x}$ orthogonal to this space. By Claim (1), $AW\mathbf{y} = A\mathbf{y}$, so that $(A - AW)\mathbf{y} = 0$. Hence, without loss of generality, we may assume $\mathbf{x}$ has no component in the space spanned by the $\mathbf{v_i}$. So, $W\mathbf{x} = 0$ and thus $(A - AW)\mathbf{x} = A\mathbf{x}$. Also, $R\mathbf{x} = \mathbf{0}$. Now imagine approximating the matrix multiplication A^TA by $A^T \otimes_r A$. Since the columns of A^T are rows of A, this can be done using the sampled rows of A. So, it follows from $R\mathbf{x} = \mathbf{0}$ that $A^T \otimes_r A\mathbf{x} = \mathbf{0}$. Using this, we get

$$|A\mathbf{x}|^2 = \mathbf{x}^T A^T A\mathbf{x} = \mathbf{x}^T(A^TA - A^T \otimes_r A)\mathbf{x} \leq ||A^TA - A^T \otimes_r A||_2.$$

So, $||AW - A||_2^2 \leq ||A^TA - A^T \otimes_r A||_2$. Thus, we have

$$E(||A - AW||_2) \leq \left(E(||A - AW||_2^2)\right)^{1/2} \leq E||A^TA - A^T \otimes_r A||_2 \leq \frac{1}{r}||A||_F^2,$$

by Lemma (1.4) proving (1).

For (2), Lemma (1.4) implies that $E\left(||AW - A \otimes_s W||_2\right) \leq \frac{1}{\sqrt{r}}||A||_F||W||_F$. Now, $||W||_F^2$ is the sum of the squared lengths of the rows of W which equals trace of (WW^T) and since $WW^T = \sum_{i-1}^{t} \mathbf{v_i}\mathbf{v_i}^T$, we get that trace of WW^T is t which proves (2).

The proof of (3) is left to the reader. Also left to the reader are choices of s, r to make the error bounds small. ∎

1.3 Graph and Matrix Sparsifiers

The problem we consider here is the following: we have a $n \times d$ matrix A, where, $n >> d$. We are to find a subset of some r rows of A and multiply each of these r rows by some scaler to form a $r \times d$ matrix B so that **for every** $d-$vector $\mathbf{v}$, we have

$$|B\mathbf{v}| \approx_\varepsilon |A\mathbf{v}|,$$

where, for two non-negative real numbers a, b, we say $b \approx_\varepsilon a$ iff $b \in [(1-\varepsilon)a, (1+\varepsilon)a]$. [In words, b is an approximation to a with relative error at most ε.] So, we wish the length of $B\mathbf{v}$ to be a relative error approximation to length of $A\mathbf{v}$ for every $\mathbf{v}$. We give two motivations for this problem before solving it by showing that with $r = O(d \log d/\varepsilon^4)$, we can indeed solve the problem. [In words, essentially $O(d \log d)$ rows of A suffice, however large n is.]

Start with the familiar example of a document term matrix. Suppose we have n documents and d terms, where, $n >> d$. Each document is represented as before as a $d-$ vector with one component per term. A very general problem we can ask is for a summary of A so that for every new document (which is itself represented as a $d-$vector $\mathbf{v}$), we should be able to compute an approximation to the vector of similarities of $\mathbf{v}$ to each existing document - i.e., the vector $A\mathbf{v}$ to relative error ε, namely, we want a vector $\mathbf{z}$ approximating $A\mathbf{v}$ in the sense $z_i \approx_\varepsilon (A\mathbf{v})_i$ for every i. We do not know a solution of this problem at the current state of knowledge. A simpler question is to ask that the summary be able to approximate $|A\mathbf{v}|^2$ which is the sum of squared similarities of the new document to every document in the collection and this is the problem we solve here.

A second important example is that of graphs. Suppose we are given a graph $G(V, E)$ with d vertices and n edges. n could be as large as $\binom{d}{2}$ (and no larger). Represent the graph by a signed edge-vertex incidence matrix A. A has one column per vertex and one row per edge. each row has exactly two non-zero entries - a +1 for one end vertex of the edge and a -1 for the other end vertex. [We arbitrarily choose which end vertex gets a 1 and which a -1.] For $S \subseteq V$, the cut $(S, \bar{S})$ is the set of edges between S and $\bar{S}$. It is easy to show that

$$\text{The number of edges in the cut } (S, \bar{S}) \quad = \quad |A\mathbf{1}_S|^2,$$

where, $\mathbf{1}_S$ is the vector with 1 for each $i \in S$ and 0 for each $i \notin S$. [We leave this to the reader.] So the problem here would "sparsify" the graph - i.e., produce a subset of $O(d \log d)$ edges to form a sub-graph H and weight them so that **for every cut**, the weight of the cut in H is a relative error ε approximation to the number of edges in the cut in G.

Theorem 1.7 *Suppose A is any $n \times d$ matrix. We can find (in polynomial time) subset of $r = \Omega(d \log d/\varepsilon^4)$ rows of A and multiply each by a scaler to form a $r \times d$ matrix B so that*

$$|B\mathbf{v}| \quad \approx_\varepsilon \quad |A\mathbf{v}| \quad \forall d - \textit{vectors } \mathbf{v}.$$

Proof: The proof will use length-squared sampling on a matrix A' obtained from A by scaling the rows of A suitably. First, we argue that we cannot do without this scaling. Consider the case of graphs, where, A is the edge-vertex incidence matrix as above. The graph might have one (or more) edge(s) which must be included in the sparsifier. A simple example is the case when G consists of two cliques of size $n/2$ each connected by a single edge. So the cut with one of the clique-vertices forming S has one edge and unless we pick this edge in our subset, we would have 0 weight across the cut which is not correct. But if we just use uniform sampling, we are quite unlikely to pick this edge among our sample of $d \log d$ edges since there are in total $\Omega(d^2)$ edges.

Now to the proof of the theorem: First observe that $|A\mathbf{v}|^2 = \mathbf{v}^T(A^TA)\mathbf{v}$ and similarly for B. Also observe that for any two non-negative real numbers a, b, $b^2 \approx_\varepsilon a^2$ implies that $b \approx_\varepsilon a$. So intuitively, it suffices to approximate A^TA. For this, $A^T \otimes_r A$ suggests itself as a sampling based method, since, as we saw earlier, $A^T \otimes A$ may be written as B^TB where B consists of some rows of A scaled.

We start with an important fact about length-squared sampling which is stated below. [The proof is beyond the scope of the book and so we assume the theorem without proof.]

Theorem 1.8 *For any $n \times d$ matrix A, if $r \in \Omega(d \log d/\varepsilon^4)$, then*

$$||A^T \otimes_r A - A^TA||_2 \leq \varepsilon ||A||_2^2.$$

In other words, $O(d \log d)$ sampled rows of A are sufficient to approximate A^TA in spectral norm to error $||A||_2$. Recalling that $A^T \otimes A$ can be written as B^TB where, B is an $r \times d$ matrix consisting of the r sampled rows, scaled, (see section (1.2.1)), we get $\mathbf{v}^T(B^TB)\mathbf{v} = |B\mathbf{v}|^2$ is within $\varepsilon||A||_2^2$ of $|A\mathbf{v}|^2$. But for relative error, we need the error to be at most $\varepsilon|A\mathbf{v}|^2$ and if $||A||_2 > |A\mathbf{v}|^2$ which is the case in general, we do not get a relative error result directly.

To achieve relative error, more work is needed. Find the SVD of A and suppose it is

$$A = \sum_{i=1}^{t} \sigma_i \mathbf{u_i} \mathbf{v_i}^T, \quad t = \text{Rank}(A).$$

Let $P = \sum_{i=1}^{t} \mathbf{u_i}\mathbf{v_i}$. P has all singular values equal to 1, so that $|P\mathbf{v}| = |\mathbf{v}|$ for all vectors $\mathbf{v}$ in the row space of P.

Apply the theorem now to P to get that for every non-zero vector $\mathbf{v}$ in the row space of P:

$$||P^T \otimes_r P - P^T P||_2 \leq \varepsilon ||P||_2 = \varepsilon$$
$$\Longrightarrow \mathbf{v}^T P^T \otimes_r P\mathbf{v} \approx_\varepsilon \mathbf{v}^T P^T P\mathbf{v}.$$

But $P = AD$, where, D the "psuedo-inverse" of A is $D = \sum_{i=1}^{t} \frac{1}{\sigma_i} \mathbf{v_i} \mathbf{v_i}^T$. But then, $\mathbf{v}^T P^T \otimes_r P\mathbf{v} \approx_\varepsilon \mathbf{v}^T P^T P\mathbf{v} \forall \mathbf{v}$ means $\mathbf{v}^T D^T A^T \otimes_r AD\mathbf{v} \approx_\varepsilon \mathbf{v}^T D^T A^T AD\mathbf{v} \forall \mathbf{v}$. But then D is non-singular. Suppose $\mathbf{w}$ is any vector in the span of $\mathbf{v_i}$. $\mathbf{w}$ can be written as $D\mathbf{v}$ with $\mathbf{v} = D^{-1}\mathbf{w}$ and from this it follows that

$$\mathbf{w}^T A^T \otimes_r A\mathbf{w} \approx_\varepsilon \mathbf{w}^T A^T A\mathbf{w}$$

and the theorem follows. ∎

1.4 Sketches of Documents

Suppose one wished to store all the web pages from the WWW. Since there are billions of web pages, one might store just a sketch of each page where a sketch is a few hundred bits that capture sufficient information to do whatever task one had in mind. A web page or a document is a sequence. We begin this section by showing how to sample a set and then how to convert the problem of sampling a sequence into a problem of sampling a set.

Consider subsets of size 1000 of the integers from 1 to 10^6. Suppose one wished to compute the resemblance of two subsets A and B by the formula

$$\text{resemblance}\,(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Suppose that instead of using the sets A and B, one sampled the sets and compared random subsets of size ten. How accurate would the estimate be? One way to sample would be to select ten elements uniformly at random from A and B. However, this method is unlikely to produce overlapping samples. Another way would be to select the ten smallest elements from each of A and B. If the sets A and B overlapped significantly one might expect the sets of ten smallest elements from each of A and B to also overlap. One difficulty that might arise is that the small integers might be used for some special purpose and appear in essentially all sets and thus distort the results. To overcome this potential problem, rename all elements using a random permutation.

Suppose two subsets of size 1000 overlapped by 900 elements. What would the overlap be of the 10 smallest elements from each subset? One would expect the nine smallest elements from the 900 common elements to be in each of the two subsets for an overlap of 90%. The resemblance(A, B) for the size ten sample would be 9/11=0.81.

Another method would be to select the elements equal to zero mod m for some integer m. If one samples mod m the size of the sample becomes a function of n. Sampling mod m allows us to also handle containment.

In another version of the problem one has a sequence rather than a set. Here one converts the sequence into a set by replacing the sequence by the set of all short subsequences of some length k. Corresponding to each sequence is a set of length k subsequences. If k is sufficiently large, then two sequences are highly unlikely to give rise to the same set of subsequences. Thus, we have converted the problem of sampling a sequence to that of sampling a set. Instead of storing all the subsequences, we need only store a small subset of the set of length k subsequences.

Suppose you wish to be able to determine if two web pages are minor modifications of one another or to determine if one is a fragment of the other. Extract the sequence of words occurring on the page. Then define the set of subsequences of k consecutive words from the sequence. Let $S(D)$ be the set of all subsequences of length k occurring in document D. Define resemblance of A and B by

$$\text{resemblance}\,(A,B) = \frac{|S(A)\cap S(B)|}{|S(A)\cup S(B)|}$$

And define containment as

$$\text{containment}\,(A,B) = \frac{|S(A)\cap S(B)|}{|S(A)|}$$

Let W be a set of subsequences. Define $\min(W)$ to be the s smallest elements in W and define $\text{mod}\,(W)$ as the set of elements of w that are zero mod m.

Let π be a random permutation of all length k subsequences. Define $F(A)$ to be the s smallest elements of A and $V(A)$ to be the set mod m in the ordering defined by the permutation.

Then

$$\frac{F(A)\cap F(B)}{F(A)\cup F(B)}$$

and

$$\frac{|V(A)\cap V(B)|}{|V(A)\cup V(B)|}$$

are unbiased estimates of the resemblance of A and B. The value

$$\frac{|V(A)\cap V(B)|}{|V(A)|}$$

is an unbiased estimate of the containment of A in B.

1.5 Exercises

Algorithms for Massive Data Problems

Exercise 1.1 *Given a stream of n positive real numbers $a_1, a_2, \ldots, a_n$, upon seeing $a_1, a_2, \ldots, a_i$ keep track of the sum $a = a_1 + a_2 + \cdots + a_i$ and a sample a_j, $j \leq i$ drawn with probability proportional to its value. On reading a_{i+1}, with probability $\frac{a_{i+1}}{a+a_{i+1}}$ replace the current sample with a_{i+1} and update a. Prove that the algorithm selects an a_i from the stream with the probability of picking a_i being proportional to its value.*

Exercise 1.2 *Given a stream of symbols $a_1, a_2, \ldots, a_n$, give an algorithm that will select one symbol uniformly at random from the stream. How much memory does your algorithm require?*

Exercise 1.3 *Give an algorithm to select an a_i from a stream of symbols $a_1, a_2, \ldots, a_n$ with probability proportional to a_i^2.*

Exercise 1.4 *How would one pick a random word from a very large book where the probability of picking a word is proportional to the number of occurrences of the word in the book?*

Solution: Put your finger on a random word. The probabilities will be automatically proportional to the number of occurrences, since each occurrence is equally likely to be picked. ∎

Exercise 1.5 *For the streaming model give an algorithm to draw s independent samples each with the probability proportional to its value. Justify that your algorithm works correctly.*

Frequency Moments of Data Streams
Number of Distinct Elements in a Data Stream
Lower bound on memory for exact deterministic algorithm
Algorithm for the Number of distinct elements
Universal Hash Functions

Exercise 1.6 *Show that for a 2-universal hash family $Prob\left(h(x) = z\right) = \frac{1}{M+1}$ for all $x \in \{1, 2, \ldots, m\}$ and $z \in \{0, 1, 2, \ldots, M\}$.*

Exercise 1.7 *Let p be a prime. A set of hash functions*

$$H = \{h | \{0, 1, \ldots, p-1\} \rightarrow \{0, 1, \ldots, p-1\}\}$$

is 3-universal if for all u,v,w,x,y, and z in $\{0, 1, \ldots, p-1\}$

$$Prob\left(h\left(x\right) = u,\ h\left(y\right) = v,\ h\left(z\right) = w\right) = \frac{1}{p^3}.$$

(a) *Is the set* $\{h_{ab}(x) = ax + b \mod p | 0 \leq a, b < p\}$ *of hash functions 3-universal?*

(b) *Give a 3-universal set of hash functions.*

Solution: No. $h(x) = u$ and $h(y) = v$ determine a and b and hence $h(z)$. ∎

Exercise 1.8 *Give an example of a set of hash functions that is not 2-universal.*

Solution: Consider a hash function h_i that maps all x's to the integer i.

$$H = \{h_i(x) = i | 1 \leq i \leq m\}$$

is a set of hash functions where every x is equally likely to be mapped to any i in the range 1 to m by a hash function chosen at random. However, $h(x)$ and $h(y)$ are clearly not independent. ∎

Analysis of distinct element counting algorithm
Counting the Number of Occurrences of a Given Element.

Exercise 1.9

(a) *What is the variance of the method in Section 1.1.2 of counting the number of occurrences of a 1 with* $\log \log n$ *memory?*

(b) *Can the algorithm be iterated to use only* $\log \log \log n$ *memory? What happens to the variance?*

Exercise 1.10 *Consider a coin that comes down heads with probability p. Prove that the expected number of flips before a head occurs is* $1/p$.

Solution: Let f be the number of flips before a heads occurs. Then

$$\begin{aligned} E[f] &= a + 2(1-a)a + 3(1-a)^2 a + \cdots \\ &= \tfrac{a}{1-a}\left[(1-a) + 2(1-a)^2 + 3(1-a)^3\right] \\ &= \tfrac{a}{1-a}\tfrac{1-a}{a^2} = \tfrac{1}{a} \end{aligned}$$

∎

Exercise 1.11 *Randomly generate a string* $x_1 x_2 \cdots x_n$ *of* 10^6 *0's and 1's with probability* $1/2$ *of* x_i *being a 1. Count the number of ones in the string and also estimate the number of ones by the approximate counting algorithm. Repeat the process for p=1/4, 1/8, and 1/16. How close is the approximation?*

Counting Frequent Elements
The Majority and Frequent Algorithms
The Second Moment

Exercise 1.12 *Construct an example in which the majority algorithm gives a false positive, i.e., stores a nonmajority element at the end.*

Exercise 1.13 *Construct examples where the frequent algorithm in fact does as badly as in the theorem, i.e., it "under counts" some item by n/(k+1).*

Exercise 1.14 *Recall basic statistics on how an average of independent trials cuts down variance and complete the argument for relative error ε estimate of $\sum_{s=1}^{m} f_s^2$.*

Error-Correcting codes, polynomial interpolation and limited-way independence

Exercise 1.15 *Let F be a field. Prove that for any four distinct points a_1, a_2, a_3, and a_4 in F and any four (possibly not distinct) values b_1, b_2, b_3, and b_4 in F, there is a unique polynomial $f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3$ of degree at most three so that $f(a_1) = b_1$, $f(a_2) = b_2$, $f(a_3) = b_3$ $f(a_4) = b_4$ with all computations done over F.*

Solution: Van Der Monde Matrix etc ∎

Sketch of a Large Matrix

Exercise 1.16 *Suppose we want to pick a row of a matrix at random where the probability of picking row i is proportional to the sum of squares of the entries of that row. How would we do this in the streaming model? Do not assume that the elements of the matrix are given in row order.*

(a) *Do the problem when the matrix is given in column order.*

(b) *Do the problem when the matrix is represented in sparse notation: it is just presented as a list of triples (i, j, a_{ij}), in arbitrary order.*

Solution: Pick an element a_{ij} with probability proportional to a_{ij}^2. We have already described the algorithm for this. Then return the i (row number) of the element picked. The probability of picking one particular i is just the sum of probabilities of picking each element of the row which is $\sum_j \frac{a_{ij}^2}{\sum_{k,l} a_{kl}^2}$, exactly as desired. Note that this works even in the sparse representation. ∎

Matrix Multiplication Using Sampling

Exercise 1.17 *Suppose A and B are two matrices. Show that $AB = \sum_{k=1}^{n} A(:,k)B(k,:)$.*

Exercise 1.18 *Generate two 100 by 100 matrices A and B with integer values between 1 and 100. Compute the product AB both directly and by sampling. Plot the difference in L_2 norm between the results as a function of the number of samples. In generating the matrices make sure that they are skewed. One method would be the following. First generate two 100 dimensional vectors a and b with integer values between 1 and 100. Next generate the i^{th} row of A with integer values between 1 and a_i and the i^{th} column of B with integer values between 1 and b_i.*

Approximating a Matrix with a Sample of Rows and Columns

Exercise 1.19 *Show that ADD^TB is exactly*

$$\frac{1}{s}\left(\frac{A(:,k_1)B(k_1,:)}{p_{k_1}} + \frac{A(:,k_2)B(k_2,:)}{p_{k_2}} + \cdots + \frac{A(:,k_s)B(k_s,:)}{p_{k_s}}\right)$$

Exercise 1.20 *Suppose $a_1, a_2, \ldots, a_m$ are nonnegative reals. Show that the minimum of $\sum_{k=1}^{m} \frac{a_k}{x_k}$ subject to the constraints $x_k \geq 0$ and $\sum_k x_k = 1$ is attained when the x_k are proportional to $\sqrt{a_k}$.*

Sketches of Documents

Exercise 1.21 *Consider random sequences of length n composed of the integers 0 through 9. Represent a sequence by its set of length k-subsequences. What is the resemblance of the sets of length k-subsequences from two random sequences of length n for various values of k as n goes to infinity?*

Exercise 1.22 *What if the sequences in the Exercise 1.21 were not random? Suppose the sequences were strings of letters and that there was some nonzero probability of a given letter of the alphabet following another. Would the result get better or worse?*

Exercise 1.23 *Consider a random sequence of length 10,000 over an alphabet of size 100.*

(a) *For $k = 3$ what is probability that two possible successor subsequences for a given subsequence are in the set of subsequences of the sequence?*

(b) *For $k = 5$ what is the probability?*

Solution: 1.23(a) A subsequence has 100 possible successor subsequences. The probability that a given subsequence is in the set of 10^4 subsequences of the sequence is 1/100. Thus the answer is $\left(1 - \frac{1}{100}\right)^{100} = \frac{1}{e} > 0.6$
(b) For $k = 5$ the universe of possible subsequences has grown to 10^{10} and the probability of being a possible successor drops to 10^{-6}. (1-10^{-6})^{100}=0. However, since the sequence is of length 10^4, the probability that some subsequence's successor is in the set is $(1 - 10^{-6})^{10^6} = \frac{1}{e}$.

Exercise 1.24 *How would you go about detecting plagiarism in term papers?*

Exercise 1.25 ***(Finding duplicate web pages)*** *Suppose you had one billion web pages and you wished to remove duplicates. How would you do this?*

Solution: :?? Suppose web pages are in html. Create a window consisting of each sequence of 10 consecutive html commands including text. Hash the contents of each window to an integer and save the lowest 10 integers for each page. To detect exact duplicates, hash the 10 integers to a single integer and look for collisions. If we want almost exact duplicates we might ask for eight of the ten integers to be the same. In this case hash the three lowest integers, the next three lowest, and the last four to integers and look for collisions. If eight of the ten integers agree one of the above three hashes must agree. **NOT TRUE.**

Exercise 1.26 *Construct two sequences of 0's and 1's having the same set of subsequences of width w.*

Solution: 1.26 For $w = 3$, 11101111 and 11110111.

Exercise 1.27 *Consider the following lyrics:*

> *When you walk through the storm hold your head up high and don't be afraid of the dark. At the end of the storm there's a golden sky and the sweet silver song of the lark.*
> *Walk on, through the wind, walk on through the rain though your dreams be tossed and blown. Walk on, walk on, with hope in your heart and you'll never walk alone, you'll never walk alone.*

How large must k be to uniquely recover the lyric from the set of all subsequences of symbols of length k? Treat the blank as a symbol.

Exercise 1.28 ***Blast****: Given a long sequence a, say 10^9 and a shorter sequence b, say 10^5, how do we find a position in a which is the start of a subsequence b' that is close to b? This problem can be solved by dynamic programming but not in reasonable time. Find a time efficient algorithm to solve this problem.*
Hint: ***(Shingling approach)*** *One possible approach would be to fix a small length, say seven, and consider the shingles of a and b of length seven. If a close approximation to b is a substring of a, then a number of shingles of b must be shingles of a. This should allows us to find the approximate location in a of the approximation of b. Some final algorithm should then be able to find the best match.*