6 Algorithms for Massive Data Problems: Streaming, Sketching, and Sampling

6.1 Introduction

This chapter deals with massive data problems where the input data is too large to be stored in random access memory. One model for such problems is the streaming model, where n data items $a_1, a_2, \ldots, a_n$ arrive one at a time. For example, the a_i might be IP addresses being observed by a router on the internet. The goal is for our algorithm to compute some statistics, property, or summary of these data items without using too much memory, much less than n. More specifically, we assume each a_i itself is a b-bit quantity where b is not too large. For example, each a_i might be an integer in $\{1, \ldots, m\}$ where $m = 2^b$. Our goal will be to produce some desired output using space polynomial in b and $\log n$; see Figure 6.1.

For example, a very easy problem to solve in the streaming model is to compute the sum of all the a_i. If each a_i is an integer between 1 and $m = 2^b$, then the sum of all the a_i is an integer between 1 and mn and so the number of bits of memory needed to maintain the sum is $O(b + \log n)$. A harder problem, which we discuss shortly, is computing the number of distinct numbers in the input sequence.

One natural approach for tackling a range of problems in the streaming model is to perform random sampling of the input "on the fly". To introduce the basic flavor of sampling on the fly, consider a stream $a_1, a_2, \ldots, a_n$ from which we are to select an index i with probability proportional to the value of a_i. When we see an element, we do not know the probability with which to select it since the normalizing constant depends on all of the elements including those we have not yet seen. However, the following method works. Let s be the sum of the a_i's seen so far. Maintain s and an index i selected with probability $\frac{a_i}{s}$. Initially $i = 1$ and $s = a_1$. Having seen symbols $a_1, a_2, \ldots, a_j$, s will equal $a_1 + a_2 + \cdots + a_j$ and for each i in $\{1, \ldots, j\}$, the selected index will be i with probability $\frac{a_i}{s}$. On seeing a_{j+1}, change the selected index to $j + 1$ with probability $\frac{a_{j+1}}{s+a_{j+1}}$ and otherwise keep the same index as before with probability $1 - \frac{a_{j+1}}{s+a_{j+1}}$. If we change the index to $j + 1$, clearly it was selected with the correct probability. If we keep i as our selection, then it will have been selected with probability

$$\left(1 - \frac{a_{j+1}}{s + a_{j+1}}\right) \frac{a_i}{s} = \frac{s}{s + a_{j+1}} \frac{a_i}{s} = \frac{a_i}{s + a_{j+1}}$$

which is the correct probability for selecting index i. Finally s is updated by adding a_{j+1} to s. This problem comes up in many areas such as sleeping experts where there is a sequence of weights and we want to pick an expert with probability proportional to its weight. The a_i's are the weights and the subscript i denotes the expert.

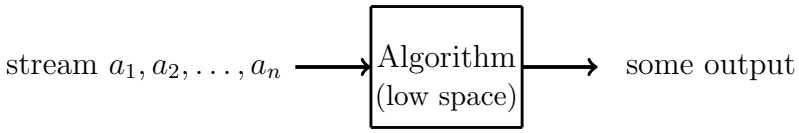

Figure 6.1: High-level representation of the streaming model

6.2 Frequency Moments of Data Streams

An important class of problems concerns the frequency moments of data streams. As mentioned above, 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 using the convention $0^0 = 0$. 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, i.e., the average squared difference from the average frequency.

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

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. First a note on the motivation for these problems. The identity and frequency of the the most frequent item, or more generally, items whose frequency exceeds a given fraction of n, is clearly important in many applications. If the items are packets on a network with source and/or destination addresses, the high frequency items identify the heavy bandwidth users. If the data consists of 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 the source address, destination address, and the number of packets for all the messages passing through them. 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 one natural measure.

6.2.1 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. Note that this is easy to do in $O(m)$ space by just storing a bit-vector that records which elements have been seen so far and which have not. It is also easy to do in $O(n \log m)$ space by storing a list of all distinct elements that have been seen. However, our goal is to use space logarithmic in m and n. We first show that this is impossible using an exact deterministic algorithm. Any deterministic algorithm that determines the number of distinct elements exactly must use at least m bits of memory on some input sequence of length $O(m)$. We then will show how to get around this problem using randomization and approximation.

Lower bound on memory for exact deterministic algorithm

We show that any exact deterministic algorithm must use at least m bits of memory on some sequence of length $m+1$. Suppose we have seen $a_1, \ldots, a_m$, and suppose for sake of contradiction that our algorithm uses less than m bits of memory on all such sequences. There are $2^m - 1$ possible subsets of $\{1, 2, \ldots, m\}$ that the sequence could contain and yet only 2^{m-1} possible states of our algorithm's memory. Therefore there must be two different subsets S_1 and S_2 that lead to the same memory state. If S_1 and S_2 are of different sizes, then clearly this implies an error for one of the input sequences. On the other hand, if they are the same size, then if the next element is in S_1 but not S_2, the algorithm will give the same answer in both cases and therefore must give an incorrect answer on at least one of them.

Algorithm for the Number of distinct elements

To beat the above lower bound, consider approximating the number of distinct elements. Our algorithm will produce a number that is within a constant factor of the correct answer using randomization and thus a small probability of failure. First, the idea: suppose the set S of distinct elements was itself chosen uniformly at random from $\{1, \ldots, m\}$. Let min denote the minimum element in S. What is the expected value of min? If there was one distinct element, then its expected value would be roughly $\frac{m}{2}$. If there were two distinct elements, the expected value of the minimum would be roughly $\frac{m}{3}$. More generally, for a random set S, the expected value of the minimum is approximately $\frac{m}{|S|+1}$. See Figure 6.2. Solving $min = \frac{m}{|S|+1}$ yields $|S| = \frac{m}{min} - 1$. This suggests keeping track of the minimum element in $O(\log m)$ space and using this equation to give

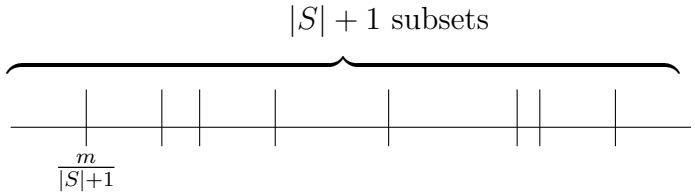

Figure 6.2: Estimating the size of S from the minimum element in S which has value approximately $\frac{m}{|S|+1}$. The elements of S partition the set $\{1, 2, \ldots, m\}$ into $|S|+1$ subsets each of size approximately $\frac{m}{|S|+1}$.

an estimate of $|S|$.

Converting the intuition into an algorithm via hashing

In general, the set S might not have been chosen uniformly at random. If the elements of S were obtained by selecting the $|S|$ smallest elements of $\{1, 2, \ldots, m\}$, the above technique would give a very bad answer. However, we can convert our intuition into an algorithm that works well with high probability on every sequence via hashing. Specifically, we will use a hash function h where

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

and then instead of keeping track of the minimum element $a_i \in S$, we will keep track of the minimum *hash value*. The question now is: what properties of a hash function do we need? Since we need to store h, we cannot use a totally random mapping since that would take too many bits. Luckily, a pairwise independent hash function, which can be stored compactly is sufficient.

We recall the formal definition of pairwise 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 in the choice of hash function.

2-Universal (Pairwise Independent) Hash Functions

A set of hash functions

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

is *2-universal* or *pairwise independent* if for all x and y in $\{1, 2, \ldots, m\}$ with $x \neq y$, $h(x)$ and $h(y)$ are each equally likely to be any element of $\{0, 1, 2, \ldots, M-1\}$ and are statistically independent. It follows that a set of hash functions H is 2-universal if and only if for all x and y in $\{1, 2, \ldots, m\}$, $x \neq y$, $h(x)$ and $h(y)$ are each equally likely to be any element of $\{0, 1, 2, \ldots, M-1\}$, and for all w, z we have:

$$\operatorname*{Prob}_{h \sim H}\big(h(x) = w \text{ and } h(y) = z\big) = \tfrac{1}{M^2}.$$

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

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

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 $h(x) = w$ and $h(y) = z$ if and only if

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

If $x \neq y$, the matrix $\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$ is invertible modulo M.[28] Thus

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}^{-1} \begin{pmatrix} w \\ z \end{pmatrix} \pmod{M}$$

and for each $\binom{w}{z}$ there is a unique $\binom{a}{b}$. Hence

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

and H is 2-universal.

Analysis of distinct element counting algorithm

Let $b_1, b_2, \ldots, b_d$ be the distinct values that appear in the input. Select h from the 2-universal family of hash functions H. Then the set $S = \{h(b_1), h(b_2), \ldots, h(b_d)\}$ is a set of d random and pairwise 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 6.1 *With probability at least $\frac{2}{3} - \frac{d}{M}$, we have $\frac{d}{6} \leq \frac{M}{\min} \leq 6d$, where min is the smallest element of S.*

Proof: First, we show that $\text{Prob}\left(\frac{M}{\min} > 6d\right) < \frac{1}{6} + \frac{d}{M}$. This part does not require pairwise independence.

$$\begin{aligned} \text{Prob}\left(\frac{M}{\min} > 6d\right) &= \text{Prob}\left(\min < \frac{M}{6d}\right) = \text{Prob}\left(\exists k,\ h(b_k) < \frac{M}{6d}\right) \\ &\leq \sum_{i=1}^{d} \text{Prob}\left(h(b_i) < \frac{M}{6d}\right) \leq d\left(\frac{\lceil \frac{M}{6d} \rceil}{M}\right) \leq d\left(\frac{1}{6d} + \frac{1}{M}\right) \leq \frac{1}{6} + \frac{d}{M}. \end{aligned}$$

[28] The primality of M ensures that inverses of elements exist in Z_M^* and $M > m$ ensures that if $x \neq y$, then x and y are not equal mod M.

t

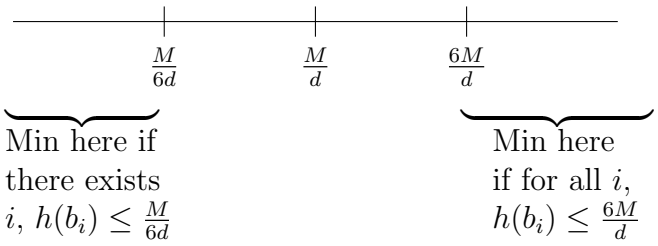

Figure 6.3: Location of the minimum in the distinct counting algorithm.

Next, we show that Prob $\left(\frac{M}{\min} < \frac{d}{6}\right) < \frac{1}{6}$. This part will use pairwise independence. First, Prob $\left(\frac{M}{\min} < \frac{d}{6}\right)$ = Prob $\left(\min > \frac{6M}{d}\right)$ = Prob $\left(\forall k,\ h\left(b_k\right) > \frac{6M}{d}\right)$. For $i = 1, 2, \ldots, d$, define the indicator variable

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

and let

$$y = \sum_{i=1}^{d} y_i.$$

We want to show that with good probability, we will see a hash value in $[0, \frac{6M}{d}]$, i.e., that Prob$(y = 0)$ is small. Now Prob $\left(y_i = 1\right) \geq \frac{6}{d}$, $E\left(y_i\right) \geq \frac{6}{d}$, and $E\left(y\right) \geq 6$. For 2-way independent random variables, the variance of their sum is the sum of their variances. So $\mathrm{Var}\left(y\right) = d\mathrm{Var}\left(y_1\right)$. Further, since y_1 is 0 or 1, $\mathrm{Var}(y_1) = E\left[\left(y_1 - E(y_1)\right)^2\right] = E(y_1^2) - E^2(y_1) = E(y_1) - E^2(y_1) \leq E\left(y_1\right)$. Thus $\mathrm{Var}(y) \leq E\left(y\right)$. By the Chebyshev inequality,

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

Since $\frac{M}{\min} > 6d$ with probability at most $\frac{1}{6} + \frac{d}{M}$ 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 $\frac{2}{3} - \frac{d}{M}$. ∎

6.2.2 Number of Occurrences of a Given Element.

To count the number of occurrences of a given 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, one 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 with $\log n$ bits of memory we could keep track of the exact number of 1's. However, the number can be approximated with only $\log \log n$ 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 m of occurrences. 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 $1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$.

6.2.3 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-once streaming data with a deterministic algorithm, requires $\Omega(\min(n, m))$ space. Suppose n is even and the last $n/2$ items are identical. Suppose also that after reading the first $n/2$ items, there are two different sets of elements that result in the same content of our memory. In that case, a mistake would occur if the second half of the stream consists solely of an element that is in one set, but not in the other. If $n/2 \geq m$ then there are at least $2^m - 1$ possible subsets of the first $n/2$ elements. If $n/2 \leq m$ then there are $\sum_{i=1}^{n/2} \binom{m}{i}$ subsets. By the above argument, the number of bits of memory must be at least the base 2 logarithm of the number of subsets, which is $\Omega(\min(m, n))$.

Surprisingly, we can bypass the above lower bound by slightly weakening our goal. Again let's require that if some element appears more than $n/2$ times, then we must output it. But now, let us say that if no element appears more than $n/2$ times, then our algorithm may output whatever it wants, rather than requiring that it output "no". 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. Thus for a majority item not to be stored at the end, more than n items must have eliminated, a contradiction.

Next we modify the above algorithm so that not just the majority, but also items with frequency above some threshold are detected. More specifically, the algorithm below finds the frequency (number of occurrences) of each element of $\{1, 2, \ldots, m\}$ to within an additive term of $\frac{n}{k+1}$. That is, for each symbol s, the algorithm produces a value $\tilde{f}_s$ in $[f_s - \frac{n}{k+1}, f_s]$, where f_s is the true number of occurrences of symbol s in the sequence. It will do so using $O(k \log n + k \log m)$ 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. Delete an element from the list if its count becomes zero.

Theorem 6.2 *At the end of Algorithm Frequent, for each $s \in \{1, 2, \ldots, m\}$, its counter on the list $\tilde{f}_s$ satisfies $\tilde{f}_s \in [f_s - \frac{n}{k+1}, f_s]$. If some s does not occur on the list, its counter is zero and the theorem asserts that $f_s \leq \frac{n}{k+1}$.*

Proof: The fact that $\tilde{f}_s \leq f_s$ is immediate. To show $\tilde{f}_s \geq f_s - \frac{n}{k+1}$, view each decrement counter step as eliminating some items. An item is eliminated if the current a_i being read is not on the list and there are already k symbols different from it on the list; in this case, a_i and k other distinct symbols 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 corresponding to distinct symbols. Thus, no more than $n/(k + 1)$ occurrences of any symbol can be eliminated. 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. ∎

Theorem 6.2 implies that we can compute the true frequency of every $s \in \{1, 2, \ldots, m\}$ to within an additive term of $\frac{n}{k+1}$.

6.2.4 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 the symbol s in the stream, and recall that 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. In particular, think of x_s as the output of a random hash function $h(s)$ whose range is just the two buckets $\{-1, 1\}$. For now, think of h as a fully independent hash function. 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 $E(x_s x_t) = E(x_s)E(x_t) = 0$ for $s \neq t$, using pairwise independence of the random variables. Thus

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

is an unbiased estimator of $\sum_{s=1}^{m} f_s^2$ in that it has the correct expectation. Note that at this point we could use Markov's inequality to state that $\text{Prob}(a \geq 3\sum_{s=1}^{m} f_s^2) \leq 1/3$, but we want to get a tighter guarantee. To do so, consider the second moment of a:

$$E(a^2) = E\left(\sum_{s=1}^{m} x_s f_s\right)^4 = E\left(\sum_{1 \leq s,t,u,v \leq m} x_s x_t x_u x_v f_s f_t f_u f_v\right).$$

The last equality is by expansion. Assume that the random variables x_s are 4-wise independent, or equivalently that they are produced by a 4-wise independent hash function. Then, since the x_s are independent in the last sum, if any one of s, u, t, or v is distinct from the others, then the expectation of the 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.

Each term in the above sum has four indices, s, t, u, v, and there are $\binom{4}{2}$ ways of

choosing two indices that have the same x value. Thus,

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

Therefore, $Var(a) = E(a^2) - E^2(a) \leq 2E^2(a)$.

Since the variance is comparable to the square of the expectation, repeating the process several times and taking the average, gives high accuracy with high probability.

Theorem 6.3 *The average x of $r = \frac{2}{\varepsilon^2\delta}$ estimates $a_1, \ldots, a_r$ using independent sets of 4-way independent random variables is*

$$Prob\left(|x - E(x)| > \varepsilon E(x)\right) < \frac{Var(x)}{\epsilon^2 E^2(x)} \leq \delta.$$

Proof: The proof follows from the fact that taking the average of r independent repetitions reduces variance by a factor of r, so that $Var(x) \leq \delta\varepsilon^2 E^2(x)$, and then applying Chebyshev's inequality. ∎

It remains to show that we can implement the desired 4-way independent random variables using $O(\log m)$ space. We earlier gave a construction for a pairwise-independent set of hash functions; now we need 4-wise independence, though only into a range of $\{-1, 1\}$. Below we present one such construction.

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

Consider the problem of generating a random m-dimensional vector $\mathbf{x}$ of ± 1's so that any four coordinates are mutually independent. Such an m-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. 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. Here, k is the ceiling of $\log_2 m$. A basic fact about polynomial interpolation is 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, so that with computations done

over F, $f(a_i) = b_i$ $1 \leq i \leq 4$.

The definition of the pseudo-random ± 1 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 $f(s)$.[29] Thus, the m-dimensional vector $\mathbf{x}$ requires only $O(k)$ bits where $k = \lceil \log m \rceil$.

Lemma 6.4 *The* $\mathbf{x}$ *defined above has 4-way independence.*

Proof: Assume that the elements of F are represented in binary using ± 1 instead of the traditional 0 and 1. Let s, t, u, and v be any four coordinates of $\mathbf{x}$ and let α, β, γ, and δ have values in ± 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, and 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 $f(s) = b_1$, $f(t) = b_2$, $f(u) = b_3$, and $f(v) = b_4$. The probability that $x_s = \alpha$, $x_t = \beta$, $x_u = \gamma$, and $x_v = \delta$ is precisely

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

Four way independence follows since $\text{Prob}(x_s = \alpha) = \text{Prob}(x_t = \beta) = \text{Prob}(x_u = \gamma) = \text{Prob}(x_v = \delta) = 1/2$ and thus

$$\begin{aligned} &\text{Prob}(x_s = \alpha)\text{Prob}(x_t = \beta)\text{Prob}(x_u = \gamma)\text{Prob}(x_v = \delta) \\ &\quad = \text{Prob}(x_s = \alpha,\ x_t = \beta,\ x_u = \gamma \text{ and } x_s = \delta) \end{aligned}$$

■

Lemma 6.4 describes how to get one vector $\mathbf{x}$ with 4-way independence. However, we need $r = O(1/\varepsilon^2)$ mutually independent vectors. Choose r independent 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 $O(\frac{\log m}{\varepsilon^2})$ bits. When a symbol s in the stream is read, compute each polynomial at s to obtain the value for the corresponding value of the x_s and update the running sums. x_s is just the leading bit of the value of the polynomial evaluated at s. This calculation requires $O(\log m)$ 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. To transmit n bits over a channel which may introduce noise, one can introduce redundancy into the transmission so that some channel errors

[29] Here we have numbered the elements of the field F $s = 1, 2, \ldots, m$.

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.

6.3 Matrix Algorithms using Sampling

We now move from the streaming model to a model where the input is stored in memory, but because the input is so large, one would like to produce a much smaller approximation to it, or perform an approximate computation on it in low space. For instance, the input might be stored in a large slow memory and we would like a small "sketch" that can be stored in smaller fast memory and yet retains the important properties of the original input. In fact, one can view a number of results from the chapter on machine learning in this way: we have a large population, and we want to take a small sample, perform some optimization on the sample, and then argue that the optimum solution on the sample will be approximately optimal over the whole population. In the chapter on machine learning, our sample consisted of independent random draws from the overall population or data distribution. Here we will be looking at matrix algorithms and to achieve errors that are small compared to the Frobenius norm of the matrix rather than compared to the total number of entries, we will perform non-uniform sampling.

Algorithms for matrix problems like matrix multiplication, low-rank approximations, singular value decomposition, compressed representations of matrices, linear regression etc. are widely used but some require $O(n^3)$ time for $n \times n$ matrices.

The natural alternative to working on the whole input matrix is to pick a random sub-matrix and compute with that. Here, we will pick a subset of columns or rows of the input matrix. If the sample size s is the number of columns we are willing to work with, we will do s independent identical trials. In each trial, we select a column of the matrix. All that we have to decide is what the probability of picking each column is. Sampling uniformly at random is one option, but it is not always good if we want our error to be a small fraction of the Frobenius norm of the matrix. For example, suppose the input matrix has all entries in the range $[-1, 1]$ but most columns are close to the zero vector with only a few significant columns. Then, uniformly sampling a small number of columns is unlikely to pick up any of the significant columns and essentially will approximate the original matrix with the all-zeroes matrix.[30]

[30]There are, on the other hand, many positive statements one *can* make about uniform sampling. For example, suppose the columns of A are data points in an m-dimensional space (one dimension per row). Fix any k-dimensional subspace, such as the subspace spanned by the k top singular vectors. If we randomly sample $\tilde{O}(k/\epsilon^2)$ columns uniformly, by the VC-dimension bounds given in Chapter 6, with high probability for every vector $\mathbf{v}$ in the k-dimensional space and every threshold τ, the fraction of the

We will see that the "optimal" probabilities are proportional to the squared length of columns. This is referred to as length squared sampling and since its first discovery in the mid-90's, has been proved to have several desirable properties which we will see. Note that all sampling we will discuss here is done with replacement.

Two general notes on this approach:

(i) We will prove error bounds which hold for all input matrices. Our algorithms are randomized, i.e., use a random number generator, so the error bounds are random variables. The bounds are on the expected error or tail probability bounds on large errors and apply to any matrix. Note that this contrasts with the situation where we have a stochastic model of the input matrix and only assert error bounds for "most" matrices drawn from the probability distribution of the stochastic model. A mnemonic is - our algorithms can toss coins, but our data does not toss coins. A reason for proving error bounds for any matrix is that in real problems, like the analysis of the web hypertext link matrix or the patient-genome expression matrix, it is the one matrix the user is interested in, not a random matrix. In general, we focus on general algorithms and theorems, not specific applications, so the reader need not be aware of what the two matrices above mean.

(ii) There is "no free lunch". Since we only work on a small random sample and not on the whole input matrix, our error bounds will not be good for certain matrices. For example, if the input matrix is the identity, it is intuitively clear that picking a few random columns will miss the other directions.

To the Reader: Why aren't (i) and (ii) mutually contradictory?

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

$$AB = \sum_{k=1}^{n} A(:,k)B(k,:).$$

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 and use 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. We will solve for a good choice of probabilities

sampled columns $\mathbf{a}$ that satisfy $\mathbf{v}^T\mathbf{a} \geq \tau$ will be within $\pm\epsilon$ of the fraction of the columns $\mathbf{a}$ in the overall matrix A satisfying $\mathbf{v}^T\mathbf{a} \geq \tau$.

later, but for now just consider the p_k as nonnegative numbers that sum to one. Define an associated random matrix variable that has value

$$X = \frac{1}{p_k} A(:,k)\, B(k,:) \tag{6.1}$$

with probability p_k. 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 $\frac{1}{p_k}$ in X. In particular, X is a matrix-valued random variable each of whose components is correct in expectation. We will be interested in

$$E\left(||AB - X||_F^2\right).$$

This can be viewed as the variance of X, defined as the sum of the variances of all its entries.

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

We want to choose p_k to minimize this quantity, and notice that we can ignore the $||AB||_F^2$ term since it doesn't depend on the p_k's at all. We can now simplify by exchanging the order of summations to get

$$\sum_{ij}\sum_{k} p_k \frac{1}{p_k^2} a_{ik}^2 b_{kj}^2 = \sum_{k} \frac{1}{p_k} \left(\sum_i a_{ik}^2\right)\left(\sum_j b_{kj}^2\right) = \sum_k \frac{1}{p_k} \left|A(:,k)\right|^2 \left|B(k,:)\right|^2.$$

What is the best choice of p_k to minimize this sum? It can be seen by calculus[31] that the minimizing p_k are proportional to $|A(:,k)||B(k,:)|$. In the important special case when $B = A^T$, pick columns of A with probabilities proportional to the squared length of the columns. Even in the general case when B is not A^T, doing so simplifies the bounds. This sampling is called "length squared sampling". If p_k is proportional to $|A(:,k)|^2$, i.e, $p_k = \frac{|A(:,k)|^2}{||A||_F^2}$, then

$$E\left(||AB - X||_F^2\right) = \text{Var}(X) \leq ||A||_F^2 \sum_k |B(k,:)|^2 = ||A||_F^2 ||B||_F^2.$$

To reduce the variance, we can do s independent trials. Each trial i, $i = 1, 2, \ldots, s$ yields a matrix X_i as in (6.1). We take $\frac{1}{s}\sum_{i=1}^{s} X_i$ as our estimate of AB. Since the variance of a sum of independent random variables is the sum of variances, the variance

[31] By taking derivatives, for any set of nonnegative numbers c_k, $\sum_k \frac{c_k}{p_k}$ is minimized with p_k proportional to $\sqrt{c_k}$.

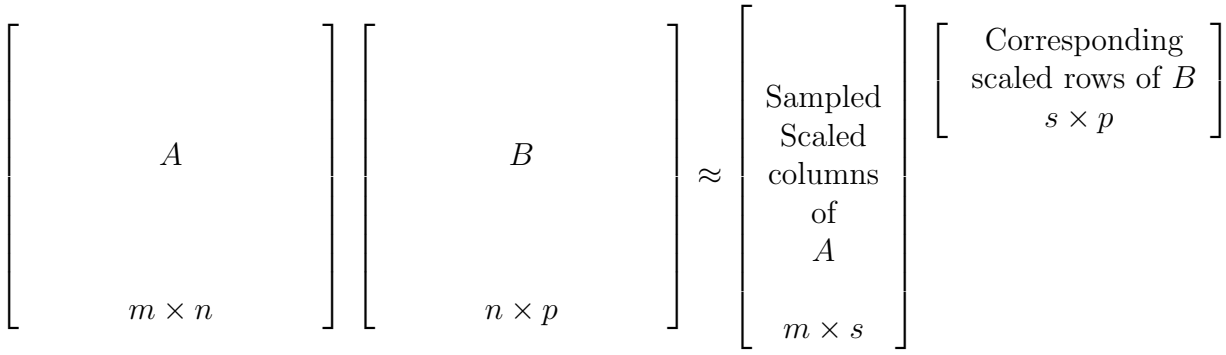

Figure 6.4: Approximate Matrix Multiplication using sampling

of $\frac{1}{s}\sum_{i=1}^{s} X_i$ is $\frac{1}{s}\text{Var}(X)$ and so is at most $\frac{1}{s}||A||_F^2||B||_F^2$. Let $k_1, \ldots, k_s$ be the k's chosen in each trial. Expanding this, gives:

$$\frac{1}{s}\sum_{i=1}^{s} X_i = \frac{1}{s}\left(\frac{A(:,k_1)\,B(k_1,:)}{p_{k_1}} + \frac{A(:,k_2)\,B(k_2,:)}{p_{k_2}} + \cdots + \frac{A(:,k_s)\,B(k_s,:)}{p_{k_s}}\right). \quad (6.2)$$

We will find it convieneint to write this as the product of an $m \times s$ matrix with a $s \times p$ matrix as follows: Let C be the $m \times s$ matrix consisting of the following columns which are scaled versions of the chosen columns of A:

$$\frac{A(:,k_1)}{\sqrt{sp_{k_1}}}, \frac{A(:,k_2)}{\sqrt{sp_{k_2}}}, \ldots \frac{A(:,k_s)}{\sqrt{sp_{k_s}}}.$$

Note that the scaling has a nice property, which the reader can verify:

$$E\left(CC^T\right) = AA^T. \quad (6.3)$$

Define R to be the $s \times p$ matrix with the corresponding rows of B similarly scaled, namely, R has rows

$$\frac{B(k_1,:)}{\sqrt{sp_{k_1}}}, \frac{B(k_2,:)}{\sqrt{sp_{k_2}}}, \ldots \frac{B(k_s,:)}{\sqrt{sp_{k_s}}}.$$

The reader may verify that

$$E\left(R^T R\right) = B^T B. \quad (6.4)$$

From (6.2), we see that $\frac{1}{s}\sum_{i=1}^{s} X_i = CR$. This is represented in Figure 6.4. We summarize our discussion in Theorem 6.3.1.

Theorem 6.5 *Suppose A is an $m \times n$ matrix and B is an $n \times p$ matrix. The product AB can be estimated by CR, where C is an $m \times s$ matrix consisting of s columns of A picked according to length-squared distribution and scaled to satisfy (6.3) and R is the*

$s \times p$ matrix consisting of the corresponding rows of B scaled to satisfy (6.4). The error is bounded by:

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

Thus, to ensure $E\left(||AB - CR||_F^2\right) \leq \varepsilon^2 ||A||_F^2 ||B||_F^2$, it suffices to make s greater than or equal to $1/\varepsilon^2$. If ε is $\Omega(1)$, so $s \in O(1)$, then the multiplication CR can be carried out in time $O(mp)$.

When is this error bound good and when is it not? Let's focus on the case that $B = A^T$ so we have just one matrix to consider. If A is the identity matrix, then the guarantee is not very good. In this case, $||AA^T||_F^2 = n$, but the right-hand-side of the inequality is $\frac{n^2}{s}$. So we would need $s > n$ for the bound to be any better than approximating the product with the zero matrix.

More generally, the trivial estimate of the all zero matrix for AA^T makes an error in Frobenius norm of $||AA^T||_F$. What s do we need to ensure that the error is at most this? If $\sigma_1, \sigma_2, \ldots$ are the singular values of A, then the singular values of AA^T are $\sigma_1^2, \sigma_2^2, \ldots$ and

$$||AA^T||_F^2 = \sum_t \sigma_t^4 \quad \text{and} \quad ||A||_F^2 = \sum_t \sigma_t^2.$$

So from Theorem 6.3.1, $E(||AA^T - CR||_F^2) \leq ||AA^T||_F^2$ provided

$$s \geq \frac{(\sigma_1^2 + \sigma_2^2 + \ldots)^2}{\sigma_1^4 + \sigma_2^4 + \ldots}.$$

If $\text{rank}(A) = r$, then there are r non-zero σ_t and the best general upper bound on the ratio $\frac{(\sigma_1^2+\sigma_2^2+\ldots)^2}{\sigma_1^4+\sigma_2^4+\ldots}$ is r, so in general, s needs to be at least r. If A is full rank, this means sampling will not gain us anything over taking the whole matrix!

However, if there is a constant c and a small integer p such that

$$\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_p^2 \geq c(\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2), \tag{6.5}$$

then,

$$\frac{(\sigma_1^2 + \sigma_2^2 + \ldots)^2}{\sigma_1^4 + \sigma_2^4 + \ldots} \leq c^2 \frac{(\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_p^2)^2}{\sigma_1^4 + \sigma_2^4 + \ldots + \sigma_p^2} \leq c^2 p,$$

and so $s \geq c^2 p$ gives us a better estimate than the zero matrix. Increasing s by a factor decreases the error by the same factor. Condition 6.5 is indeed the hypothesis of the subject of Principal Component Analysis (PCA) and there are many situations when the data matrix does satisfy the condition and sampling algorithms are useful.

6.3.2 Implementing Length Squared Sampling in Two Passes

Traditional matrix algorithms often assume that the input matrix is in random access memory (RAM) and so any particular entry of the matrix can be accessed in unit time. For massive matrices, RAM may be too small to hold the entire matrix, but may be able to hold and compute with the sampled columns and rows.

Consider a high-level model where the input matrix or matrices have to be read from external memory using one pass in which one can read sequentially all entries of the matrix and sample.

It is easy to see that two passes suffice to draw a sample of columns of A according to length squared probabilities, even if the matrix is not in row-order or column-order and entries are presented as a linked list. In the first pass, compute the length squared of each column and store this information in RAM. The lengths squared can be computed as running sums. Then, use a random number generator in RAM to determine according to length squared probability the columns to be sampled. Then, make a second pass picking the columns to be sampled.

If the matrix is already presented in external memory in column-order, then one pass will do. The idea is to use the primitive in Section 6.1: given a read-once stream of positive numbers $a_1, a_2, \ldots, a_n$, at the end have an $i \in \{1, 2, \ldots, n\}$ such that the probability that i was chosen is $\frac{a_i}{\sum_{j=1}^{n} a_j}$. Filling in the specifics is left as an exercise for the reader.

6.3.3 Sketch of a Large Matrix

The main result of this section is that for any matrix, a sample of columns and rows, each picked according to length squared distribution provides a good sketch of the matrix. Let A be an $m \times n$ matrix. Pick s columns of A according to length squared distribution. Let C be the $m \times s$ matrix containing the picked columns scaled so as to satisy (6.3), i.e., if $A(:, k)$ is picked, it is scaled by $1/\sqrt{sp_k}$. Similarly, pick r rows of A according to length squared distribution on the rows of A. Let R be the $r \times n$ matrix of the picked rows, scaled as follows: If row k of A is picked, it is scaled by $1/\sqrt{rp_k}$. We then have $E(R^T R) = A^T A$. From C and R, one can find a matrix U so that $A \approx CUR$. The schematic diagram is given in Figure 6.5.

One may recall that the top k singular vectors of the SVD of A give a similar picture; however, 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. The last property, that the approximation involves actual rows/columns of the matrix rather than linear combinations, is called an *interpolative approximation* and is useful in many contexts. On the other hand, the SVD yields the best 2-norm approximation. Error bounds for the approximation CUR are weaker.

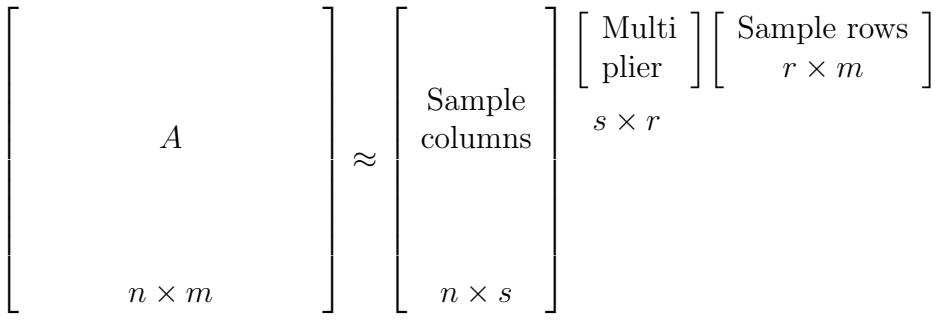

Figure 6.5: Schematic diagram of the approximation of A by a sample of s columns and r rows.

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 6.5 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 $\max_{|\mathbf{x}|=1} |A\mathbf{x}|$. The fact that the sketch is an interpolative approximation means that our approximation essentially consists a subset of documents and a subset of terms, which may be thought of as a representative set of documents and terms. Additionally, if A is sparse in its rows and columns, each document contains only a small fraction of the terms and each term is in only a small fraction of the documents, then this sparsity property will be preserved in C and R, unlike with SVD.

A second motivation comes from analyzing gene microarray data. Here, A is a matrix in which each row is a gene and each column is a condition. Entry (i, j) indicates the extent to which gene i is expressed in condition j. In this context, a CUR decomposition provides a sketch of the matrix A in which rows and columns correspond to actual genes and conditions, respectively. This can often be easier for biologists to interpret than a singular value decomposition in which rows and columns would be linear combinations of the genes and conditions.

It remains now to describe how to find U from C and R. There is a $n \times n$ matrix P of the form $P = QR$ that acts as the identity on the space spanned by the rows of R and zeros out all vectors orthogonal to this space. We state this now and postpone the proof.

Lemma 6.6 *If RR^T is invertible, then $P = R^T(RR^T)^{-1}R$ has the following properties:*

(i) *It acts as the identity matrix on the row space of R. I.e., $P\mathbf{x} = \mathbf{x}$ for every vector $\mathbf{x}$ of the form $\mathbf{x} = R^T\mathbf{y}$ (this defines the row space of R). Furthermore,*

(ii) *if $\mathbf{x}$ is orthogonal to the row space of R, then $P\mathbf{x} = \mathbf{0}$.*

If RR^T is not invertible, let rank $(RR^T) = r$ and $RR^T = \sum_{t=1}^{r} \sigma_t \mathbf{u_t}\mathbf{v_t}^T$ be the SVD of RR^T. Then,

$$P = R^T \left(\sum_{t=1}^{r} \frac{1}{\sigma_t^2} \mathbf{u_t}\mathbf{v_t}^T \right) R$$

satisfies (i) and (ii).

We begin with some intuition. In particular, we first present a simpler idea that does not work, but that motivates an idea that does. Write A as AI, where I is the $n \times n$ identity matrix. Approximate the product AI using the algorithm of Theorem 6.3.1, i.e., by sampling s columns of A according to a length-squared distribution. Then, as in the last section, write $AI \approx CW$, where W consists of a scaled version of the s rows of I corresponding to the s columns of A that were picked. Theorem 6.3.1 bounds the error $||A-CW||_F^2$ by $||A||_F^2||I||_F^2/s = \frac{n}{s}||A||_F^2$. But we would like the error to be a small fraction of $||A||_F^2$ which would require $s \geq n$, which clearly is of no use since this would pick as many or more columns than the whole of A.

Let's use the identity-like matrix P instead of I in the above discussion. Using the fact that R is picked according to length squared sampling, we will show the following proposition later.

Proposition 6.7 *$A \approx AP$ and the error $E\left(||A - AP||_2^2\right)$ is at most $\frac{1}{\sqrt{r}}||A||_F^2$.*

We then use Theorem 6.3.1 to argue that instead of doing the multiplication AP, we can use the sampled columns of A and the corresponding rows of P. The s sampled columns of A form C. We have to take the corresponding s rows of $P = R^T(RR^T)^{-1}R$, which is the same as taking the corresponding s rows of R^T, and multiplying this by $(RR^T)^{-1}R$. It is easy to check that this leads to an expression of the form CUR. Further, by Theorem 6.3.1, the error is bounded by

$$E\left(||AP - CUR||_2^2\right) \leq E\left(||AP - CUR||_F^2\right) \leq \frac{||A||_F^2||P||_F^2}{s} \leq \frac{r}{s}||A||_F^2, \qquad (6.6)$$

since we will show later that:

Proposition 6.8 $||P||_F^2 \leq r$.

Putting (6.6) and Proposition 6.7 together, and using the fact that by triangle inequality $||A - CUR||_2 \leq ||A - AP||_2 + ||AP - CUR||_2$, which in turn implies that $||A - CUR||_2^2 \leq 2||A - AP||_2^2 + 2||AP - CUR||_2^2$, the main result below follows.

Theorem 6.9 *Let A be an $m \times n$ matrix and r and s be positive integers. Let C be an $m \times s$ matrix of s columns of A picked according to length squared sampling and let R be a matrix of r rows of A picked according to length squared sampling. Then, we can find from C and R an $s \times r$ matrix U so that*

$$E\left(||A - CUR||_2^2\right) \leq ||A||_F^2 \left(\frac{2}{\sqrt{r}} + \frac{2r}{s}\right).$$

If s is fixed, the error is minimized when $r = s^{2/3}$. Choosing $s = 1/\varepsilon^3$ and $r = 1/\varepsilon^2$, the bound becomes $O(\varepsilon)||A||_F^2$. When is this bound meaningful? We discuss this further after first proving all the claims used in the discussion above.

Proof of Lemma 6.6: First consider the case that RR^T is invertible. For $\mathbf{x} = R^T\mathbf{y}$, $R^T(RR^T)^{-1}R\mathbf{x} = R^T(RR^T)^{-1}RR^T\mathbf{y} = R^T\mathbf{y} = \mathbf{x}$. If $\mathbf{x}$ is orthogonal to every row of R, then $R\mathbf{x} = \mathbf{0}$, so $P\mathbf{x} = \mathbf{0}$. More generally, if $RR^T = \sum_t \sigma_t \mathbf{u_t}\mathbf{v_t}^T$, then, $R^T \sum_t \frac{1}{\sigma_t^2} R = \sum_t \mathbf{v_t}\mathbf{v_t}^T$ and clearly satisfies (i) and (ii). ∎

Next we prove Proposition 6.7. First, recall that

$$||A - AP||_2^2 = \max_{\{\mathbf{x}:|\mathbf{x}|=1\}} |(A - AP)\mathbf{x}|^2.$$

Now suppose $\mathbf{x}$ is in the row space V of R. From Lemma 6.6, $P\mathbf{x} = \mathbf{x}$, so for $\mathbf{x} \in V$, $(A - AP)\mathbf{x} = \mathbf{0}$. Since every vector can be written as a sum of a vector in V plus a vector orthogonal to V, this implies that the maximum must therefore occur at some $\mathbf{x} \in V^\perp$. For such $\mathbf{x}$, by Lemma 6.6, $(A - AP)\mathbf{x} = A\mathbf{x}$. Thus, the question becomes: for unit-length $\mathbf{x} \in V^\perp$, how large can $|A\mathbf{x}|^2$ be? To analyze this, write:

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

This implies that $||A - AP||_2^2 \leq ||A^T A - R^T R||_2$. So, it suffices to prove that $||A^T A - R^T R||_2^2 \leq ||A||_F^4/r$ which follows directly from Theorem 6.3.1, since we can think of $R^T R$ as a way of estimating $A^T A$ by picking according to length-squared distribution columns of A^T, i.e., rows of A. This proves Proposition 6.7.

Proposition 6.8 is easy to see. By Lemma 6.6, P is the identity on the space V spanned by the rows of R, and $P\mathbf{x} = 0$ for $\mathbf{x}$ perpendicular to the rows of R. Thus $||P||_F^2$ is the sum of its singular values squared which is at most r as claimed.

We now briefly look at the time needed to compute U. The only involved step in computing U is to find $(RR^T)^{-1}$ or do the SVD of RR^T. But note that RR^T is an $r \times r$ matrix and since r is much smaller than n and m, this is fast.

Understanding the bound in Theorem 6.9: To better understand the bound in Theorem 6.9 consider when it is meaningful and when it is not. First, choose parameters $s = \Theta(1/\varepsilon^3)$ and $r = \Theta(1/\varepsilon^2)$ so that the bound becomes $E(||A - CUR||_2^2) \leq \varepsilon ||A||_F^2$. Recall that $||A||_F^2 = \sum_i \sigma_i^2(A)$, i.e., the sum of squares of all the singular values of A. Also, for convenience scale A so that $\sigma_1^2(A) = 1$. Then

$$\sigma_1^2(A) = ||A||_2^2 = 1 \quad \text{and} \quad E(||A - CUR||_2^2) \leq \varepsilon \sum_i \sigma_i^2(A).$$

This, gives an intuitive sense of when the guarantee is good and when it is not. If the top k singular values of A are all $\Omega(1)$ for $k \gg m^{1/3}$, so that $\sum_i \sigma_i^2(A) \gg m^{1/3}$, then the guarantee is only meaningful when $\varepsilon = o(m^{-1/3})$, which is not interesting because it requires $s > m$. On the other hand, if just the first few singular values of A are large and the rest are quite small, e.g, A represents a collection of points that lie very close to a low-dimensional subspace, and in particular if $\sum_i \sigma_i^2(A)$ is a constant, then to be meaningful the bound requires ε to be a small constant. In this case, the guarantee is indeed meaningful because it implies that a constant number of rows and columns provides a good 2-norm approximation to A.

6.4 Sketches of Documents

Suppose one wished to store all the web pages from the World Wide Web. Since there are billions of web pages, one might want to store just a sketch of each page where a sketch is some type of compact description that captures sufficient information to do whatever task one has in mind. For the current discussion, we will think of a web page as a string of characters, and the task at hand will be one of estimating similarities between pairs of web pages.

We begin this section by showing how to estimate similarities between sets via sampling, and then how to convert the problem of estimating similarities between strings into a problem of estimating similarities between sets.

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. Suppose A and B were each of size 1000, over lapped by 500, and both were represented by six samples. Even though half of the six samples of A were in B they would not likely be among the samples representing B. See Figure 6.6. 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

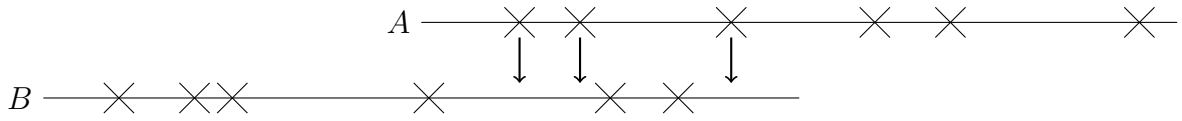

Figure 6.6: Samples of overlapping sets A and B.

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 might one expect the overlap of the 10 smallest elements from each subset to be? One would expect the nine smallest elements from the 900 common elements to be in each of the two sampled subsets for an overlap of 90%. The expected 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 one to handle containment.

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

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, viewing each word as a character. Then define the set of substrings of k consecutive words from the sequence. Let $S(D)$ be the set of all substrings of k consecutive words 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 π be a random permutation of all length k substrings. 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.

6.5 Bibliographic Notes

The hashing-based algorithm for counting the number of distrinct elements in a data stream described in Section 6.2.1 is due to Flajolet and Martin [FM85]. Algorithm Frequent for identifying the most frequent elements is due to Misra and Gries [MG82]. The algorithm for estimating the second moment of a data stream described in Section 6.2.4 is due to Alon, Matias and Szegedy [AMS96], who also gave algorithms and lower bounds for other k^{th} moments. These early algorithms for streaming data significantly influenced further research in the area. Improvements and generalizations of Algorithm Frequent were made in [MM02].

Length-squared sampling was introduced by Frieze, Kannan and Vempala [FKV04]; the algorithms of Section 6.3 are from [DKM06a, DKM06b]. The material in Section 6.4 on sketches of documents is from Broder et al. [BGMZ97].