7. Algorithms for Massive Data Problems

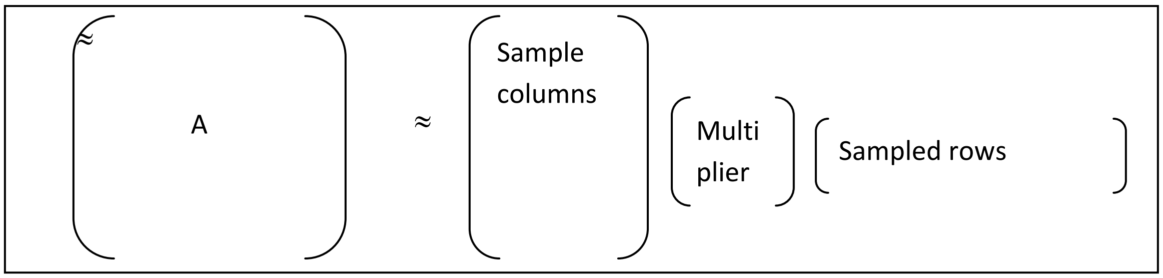
Massive Data, Sampling

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'', i.e., 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 entire data cannot be stored in RAM.

To introduce the basic flavor of sampling on the fly, consider the following primitive. 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 $a = a_1 + a_2 + \ldots + a_i$ and a sample $a_j, j \le i$ drawn with probability proportional to its value. On reading a_{i+1}, replace the current sample by a_{i+1} with probability $\frac{a_{i+1}}{a + a_{i+1}}$ and update a.

7.1 Sketch of a large matrix

We will see how to find a good 2-norm approximation of an $m \times n$ matrix A (m, n large) which is being read from external memory. The approximation consists of a sample of s columns of A, a sample of r rows of A and a $s \times r$ multiplier matrix in between. The schematic diagram is:



The crucial point is that the samples of rows and columns have to drawn according to a special probability distribution proportional to the squared length of the row or column. One may recall that if we do the SVD of A and take only the top k singular vectors, we would get 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 2-norm approximation. We will show error bounds for our approximation, but they will be weaker.

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 as dot product. With the picture above, it is clear that we can do the matrix-vector product of the right hand side with a query in time $O(rn+sr+ms)$ which would be linear if $s,r \in O(1)$. To bound errors for this process, we need to show that the difference between A and the right hand side has small 2-norm. Recall that the 2-norm $\|B\|_2$ of a matrix B is $\max_x\{x^T B\mathbf{x} : |\mathbf{x}|=1\}$.

A second motivation is from Recommendation Systems. Here A would be a customer-product matrix whose $(i,j)^{\text{th}}$ 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. Our results would 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) are enough to give us an approximation to A provided we can hypothesize that the samples were drawn according to length-squared distribution.

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

7.1.1 Matrix multiplication using sampling

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

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

An obvious use of sampling suggests itself. Sample some values for i and compute $A(:,i)B(i,:)$ for the sampled i's and use their sum (scaled suitably) as the estimate of AB. It turns out that non-uniform sampling probabilities will be useful. So, we will define a random variable k that takes on values in $\{1,2,\ldots n\}$. Let p_i denote the probability that k assumes the value i; so the p_i are non-negative and sum to 1. Define an associated random matrix variable

$$X = \frac{1}{p_i} A(:,i)\, B(i,:).$$

Let $E(X)$ denote the entry-wise expectation.

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

This explains the scaling by $\frac{1}{p_i}$ in X. Define the variance of X as the sum of the variances of all its entries.

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

We can simplify the last term by exchanging the order of summations to get

$$\text{Var}(X) \le \sum_k \frac{1}{p_k} \sum_i a_{ik}^2 \sum_j b_{kj}^2 = \sum_k \frac{1}{p_k} | A(:,k) |^2 | B(k,:) |^2.$$

Now, if we make p_i proportional to $| A(:,i) |^2$, in other words $p_i = \frac{| A(:,i) |^2}{\| A \|_F^2}$

we get a simple bound for $\text{Var}(X)$:

$$\text{Var}(X) \le \| A \|_F^2 \sum_k |B(k,:)|^2 = \| A \|_F^2 \| B \|_F^2.$$

Using probabilities 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''.

As usual, to reduce the variance, do s independent trials to pick $i_1, i_2, \ldots, i_s$. Then find the average of the corresponding X 's. This is $\frac{1}{s}\left(\frac{A(:,i_1) B(i_1,:)}{p_{i_1}} + \frac{A(:,i_2) B(i_2,:)}{p_{i_2}} + \ldots + \frac{A(:,i_s) B(i_s,:)}{p_{i_s}} \right)$ which can also be represented compactly as $ADD^T B$, where D is $n \times s$ ``column selector'' matrix with $D_{i_j,j} = \frac{1}{\sqrt{s p_{i_j}}}$ for $j = 1, 2, \ldots s$ and all other entries of D are zero. The i_j diagonal element of DD^T is sp_{i_j}. All other elements of

DD^T are zero. Thus, $ADD^T B$ consists of the weighted sum of the products of column i_j of A with column i_j of B.

Lemma 7.1: Suppose A is an $m \times n$ matrix and B is an $n \times p$ matrix. The product AB can be estimated by $ADD^T B$, where D is $n \times s$ and each column of D has precisely one non-zero entry; D_{ki} is picked to be the non-zero entry in column i with probability p_k proportional to $|A_k|^2$ and if picked, it is set to $1/\sqrt{sp_k}$. The error is bounded by $E\left(\|ADD^T B - AB\|_F^2\right) \le \frac{1}{s}\|A\|_F^2 \|B\|_F^2$.

Proof: Taking the average of s i.i.d. samples divides the variance by s.

■

Finding $ADD^T B$ in the order $\left((AD)(D^T B)\right)$ can be done in $O(s(mn+np+mp))$ time by straightforward multiplication. Note that if $s \in O(1)$ and $m = n = p$, then this time is quadratic in n which is O(number of entries).

An important special case is the multiplication $A^T A$. Then in fact length-squared distribution can be shown to be the optimal distribution. The variance of X defined above for this case is:

$$\sum_{i,j} E(X_{ij})^2 - \sum_{ij} (EX_{ij})^2 = \sum_k \frac{1}{p_k} |A^k|^4 - \|AB\|_F^2 .$$

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. **WHY?**

7.1.2. Approximating a matrix with a sample of rows and columns

Now, consider the following important problem. Suppose a large matrix A is read from external memory and a sketch of it is kept for further computation. A simple-minded sketch idea would be to keep a random sample of rows and a random sample of columns of the matrix. If we do uniform random sampling, this can fail in general. Consider the case where very few rows/columns of A are non-zero. A small sample will miss them and we will only see zeros. This toy example can be made more subtle by making a few rows have very high (absolute value) entries compared to the rest. We need to make the sampling probabilities depend on the size of the entries. If we do length-squared sampling of both columns and rows, then from these, 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:

(i) Pick s columns of A using length-squared sampling. Let C (for columns) be the $m \times s$ matrix of the picked columns scaled by $1/\sqrt{s \text{ prob. of picked column}}$, i.e., $C = AD$ where D is a column selector matrix as in the Lemma 7.1.

(ii) Pick r rows of A using length-squared sampling (here lengths are row lengths). Let R (for rows) be the $r \times n$ matrix of the picked rows scaled by $1/\sqrt{r \text{ prob. of picked row}}$. Write $R = D_1 A$, where D_1 is a $r \times m$ ``row selector matrix''.

We will approximate A by CUR, where U is a $s \times r$ multiplier matrix computed from C and R. Recall the schematic picture at the beginning of Section 7.1. Before stating precisely what U is, we give some intuition. A first idea would be to approximate A by $ADD^T I$, where I is the $n \times n$ identity matrix. By Lemma 7.1, the product $ADD^T I$ approximates $AI = A$. Note that we would not multiply A and I by sampling; the use of Lemma 7.1 is simply a proof device here. Lemma 7.1 tells us that

$$E \| A - ADD^T I \|_F^2 \le \frac{1}{s} \| A \|_F^2 \| I \|_F^2 = \frac{n}{s} \| A \|_F^2,$$

which is not very interesting since we want s<<n. Instead of using the identity matrix which introduced the factor of n, we will use an ``identity-like'' matrix based on R.

THE FOLLOWING NEEDS WORK

Lemma 7.2: If RR^T is invertible, then $R^T(RR^T)^{-1}R$ acts as the identity matrix on the row space of R. I.e., for every vector $\mathbf{x}$ of the form $\mathbf{x} = R^T\mathbf{y}$ (this defines the row space of R), we have $R^T(RR^T)^{-1}R\mathbf{x} = \mathbf{x}$.

Proof: For $\mathbf{x} = R^T\mathbf{y}$ we have $R^T(RR^T)^{-1}R\mathbf{x} = R^T(RR^T)^{-1}RR^T y = R^T y = \mathbf{x}$

■

Now, we are ready to define U.

(iii)Let $U = D^T R^T (RR^T)^{-1}$ (assuming RR^T is invertible.)

(iv)**Lemma 7.3**: Let $U = D^T R^T (RR^T)^{-1}$ (assuming RR^T is invertible). Then

$CUR = ADD^T R^T (RR^T)^{-1} R$ and $E\left(\| A - CUR \|_2\right) \le \frac{2}{r^{1/4}} \| A \|_F$ provided $s \ge r^{3/2}$.

Proof: The first assertion is obvious. Let $R^T(RR^T)^{-1}R = W$. Then $CUR = ADD^TW$.

$E\left(\|A - CUR\|_2\right) \le E\left(\|A - AW\|_2\right) + E\left(\|AW - ADD^TW\|_2\right)$. The second term is bounded by Lemma 7.1 which tells us $E\left(\|AW - ADD^TW\|_2\right) \le E\left(\|AW - ADD^TW\|_F\right) \le \frac{1}{\sqrt{s}}\|A\|_F\|W\|_F$ (since for any random variable Y , $E(Y) \le \sqrt{E(Y^2)}$). From Lemma 7.2, W acts like the identity on a r dimensional space and $W\mathbf{z} = 0$ for any $\mathbf{z}$ perpendicular to the row space of R . So, $\|W\|_F^2$, which we saw (Lemma 7.1) was the sum of its squared singular values, is r and hence $E\left(\|AW - ADD^TW\|_2\right) \le \frac{\sqrt{r}}{\sqrt{s}}\|A\|_F \le \frac{1}{r^{1/4}}\|A\|_F$.

Consider the first term $E\|A - AW\|_2$. We recall $\|A - AW\|_2 = \text{Max}\,|(A - AW)\mathbf{x}|/|\mathbf{x}|$. If $\mathbf{x}$ attains this maximum, $\mathbf{x}$ must be orthogonal to the row space of R , since by Lemma 9.2, for any vector in the row space of R , W acts like the identity. For $\mathbf{x}$ orthogonal to rows of R , we have

$$|(A - AW)\mathbf{x}|^2 = |A\mathbf{x}|^2 = \mathbf{x}^T A^T A\mathbf{x} = \mathbf{x}^T(A^TA - R^TR)\mathbf{x} \le \|A^TA - A^TD_1^TD_1A\|_2 \le \|A^TA - A^TD_1^TD_1A\|_F .$$

We will use Lemma 7.1 again to bound this, since we are picking columns of A^T (same as rows of A) in picking R . Indeed, Lemma 7.1 tells that $E\left(\|A^TA - A^TD_1D_1^TA\|_F\right) \le \frac{1}{\sqrt{r}}\|A\|_F^2$ which implies that $E\left(\|A - AW\|_2 \le \frac{1}{r^{1/4}}\|A\|_F\right)$ proving the Lemma.

■

7.2 Frequency moments of data streams

Introduction

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 non-negative 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

$$\frac{1}{m}\sum_s\left(f_s - \frac{n}{m}\right)^2 = \frac{1}{m}\sum_s f_s^2 - \frac{n^2}{m^2}.$$

In the limit as p becomes large, $\left(\sum_s f_s^p\right)^{1/p}$, 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. The number of distinct symbols could be a first step in the important task of duplicate elimination. The second moment (and variance) are useful in networking as well as database applications and others. 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 see if there is a skew – if some popular source-destination pairs have a lot of traffic for which the variance is the natural measure.

7.2.1 Number of distinct elements in a data stream

Consider a sequence $a_1, a_2, \cdots, a_n$ 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 $k \ge m$ symbols. 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 subsets of distinct symbols of different size lead to the same internal state. Then our algorithm would produce the same count of the number 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 would result in subsets of different size and thus require different states.

Algorithm for the Number of distinct elements

Let $a_1 a_2 \cdots a_n$ be a sequence of elements where each $a_i \in \{1,2,\cdots,m\}$. The number of distinct elements can be estimated with $O(\log m)$ space. Let $S \subseteq \{1,2,\cdots,m\}$ be the set of elements that appear in the sequence. Suppose that the elements of S were selected uniformly at random from $\{1,2,\cdots,m\}$ subject to their total number |*S*|. 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 $\{1,2,\cdots,m\}$ into $|S|+1$ subsets each of size approximately $\frac{m}{|S|+1}$. 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 $\{1,2,\cdots,m\}$. This is generally not the case when we have a sequence $a_1 a_2 \cdots a_n$ of elements from $\{1,2,\cdots,m\}$. Clearly if the elements of *S* were obtained by selecting the $|S|$ smallest elements of $\{1,2,\cdots,m\}$, 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

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

DO WE NEED TWO DIFFERENT SIZE SETS m AND M? IF NOT WE COULD AVOID USING A CAPITAL FOR AN INTEGER.

To count the number of distinct elements in the input, count the number of elements in the mapped set $h(a_1), h(a_2), \cdots$, the point being that $h(a_1), h(a_2), \cdots$ 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 $h(a_1), h(a_2), \cdots$ 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 below. But first recall that a hash function is always chosen from a family of hash functions (at random) and phrases like ``probability of collision'' refer to the probability in this choice.

Universal Hash Functions

The set of hash functions

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

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

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

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 $\{0,1,2,\cdots,M-1\}$ 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. 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}, we need only store the two integers a and b and 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 $\frac{1}{M^2}$.

Analysis of distinct element counting algorithm

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

Lemma 7.3: Assume *M*>100*d*. With probability at least 2/3, $\frac{d}{6} \leq \frac{M}{\min} \leq 6d$, where *min* is the smallest element of S.

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

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

For $i = 1, 2, \cdots, d$ define the indicator variable $z_i = \begin{cases} 1 & \text{if } \mathrm{h}(b_i) < \frac{M}{6d} \\ 0 & \text{otherwise} \end{cases}$

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

$\text{Prob}[z_i = 1] \leq \frac{1}{6d}$. Thus, $E(z_i) = \frac{1}{6d}$ and $E(z) = \frac{1}{6}$. Now

$$\text{Prob}\left[\frac{M}{\min} > 6d\right] = \text{Prob}\left[\exists k \; h(b_k) < \frac{M}{d6}\right] = \text{Prob}(z \geq 1) = \text{Prob}\left[z \geq 6E(z)\right].$$

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

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

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

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 $E(y) \geq 6$. For 2-way independent random variables, the variance of their sum is the sum of their variances. So $Var(y) = dVar(y_i)$. Further, it is easy to see that $Var(y_1) \leq E(y_1)$, so we have $Var(y) \leq E(y)$. Now by the Chebychev inequality, it follows that

$$\text{Prob}\left[\frac{M}{\min}<\frac{d}{6}\right]=\text{Prob}\left[\min>\tfrac{6M}{d}\right]=\text{Prob}\left[\forall k\ h(b_i)>\frac{6M}{d}\right]$$
$$=\text{Prob}(y=0)\le\text{Prob}\left[\left|y-E(y)\right|\ge E(y)\right]\le\frac{\text{Var}(y)}{(E(y))^2}\le\frac{1}{E(y)}\le\frac{1}{6}$$

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

■

7.2.2 Counting the number of occurrences of a given element.

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 $\log\log n$ 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 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 ½, *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 $1+2+4+\cdots+2^{k-1}=2^k-1$.

7.2.3 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 on streaming data (read once only) exactly 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 consists of an element in one set, but not the other. Thus, $\log_2 \binom{m}{n/2}$ bits of memory, which if m>2n is $\Omega(n)$, are needed.

The following is a simple low-space algorithm which always finds the majority vote if there is one. If there is no majority, the output may be arbitrary. That is, there may be ``false positives'', but no ``false negatives''.

Majority Algorithm: Store a_1 and initialized 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, and the counter is zero, then store a_i and set the counter to one; otherwise, decrement the counter by one.

To analyze the algorithm, it is convenient to view the decrement counter step as ``eliminating'' two items, the new one and the one which 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 1.

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. Else, if the list has less than k items, add to the list the new item with its counter set to 0ne. Otherwise, decrement each of the current k counters by one. Delete an element from the list if its count becomes zero.

Theorem 7.1: 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 can be eliminated in one of two ways. Either it is the current a_i being read and there are already k symbols on the list in which case, it and k other items are being eliminated. The second way is when the item had caused an increment in an existing counter earlier and that counter is being decremented now. In this case too, k other items are being 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.

■

Theorem 7.1 implies that we can compute the true relative frequency (number of occurrences / n) of every $s \in \{1, 2, \ldots m\}$ to within an additive term of $\frac{n}{k+1}$. (Think of why these is no contradiction in the fact that we can do it for EVERY element.)

Examples:

■

7.2.4 The Second Moment

This section focuses on computing the second moment, $\sum_{s=1}^{m} f_s^2$, of a stream with symbols from $\{1, 2, \ldots m\}$. Here f_s denotes the number of occurrences of symbol s. For $1 \le i \le m$, set each x_i to ± 1 independently 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 the sum will equal $\sum_{s=1}^{m} x_s f_s$. Now

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

Here the expectation is over the choosing of the x_i. Also

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

since the independence of x_s tells us that $E\left(x_s x_t\right) = 0$ for $s \ne t$. Thus

$$A=\left(\sum_s x_s f_s\right)^2$$

is an estimator of $\sum_s f_s^2$. We now compute the variance of this estimator.

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

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. Note that this does not need the full power of mutual independence of all the x_s , it only needs what is called 4-way independence, that any four of the x_s are mutually independent. Continuing the calculation, we need to deal only with terms of the form $x_s^2 x_t^2$ for $t \ne s$ and terms of the form x_s^4 . Thus,

.

$$Var(a)\le \binom{4}{2} E\left(\sum_{s\ne t} x_s^2 x_t^2 f_s^2 f_t^2\right) + E\left(\sum_s x_s^4 f_s^4\right) = 6\sum_{s\ne t} f_s^2 f_t^2 + \sum_s f_s^4$$
$$\le 3\left(\sum_s f_s^2\right)^2 + \left(\sum_s f_s^2\right)^2 = 4\left(E(a)\right)^2$$

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 $\frac{4}{r}\left(E(a)\right)^2$, so to achieve relative error ϵ in the estimate of $\sum_s f_s^2$, we need only $O(1/\epsilon^2)$ independent trials.

We will briefly discuss the independent trials here, so as to understand exactly the amount of independence needed. Instead of computing A as the running sum $\sum_s x_s f_s$ for one random vector $x=(x_1,x_2,\ldots x_m)$, we independently generate *r* *m*-vectors $x^1,x^2,x^3,\ldots,x^r$ at the outset and compute *r* running sums

$$\sum_s x_s^1 f_s,\ \sum_s x_s^2 f_s,\ \ldots\ \sum_s x_s^r f_s .$$

Let $a_1=\left(\sum_s x_s^1 f_s\right)^2, a_2=\left(\sum_s x_s^2 f_s\right)^2,\ldots,a_r=\left(\sum_s x_s^r f_s\right)^2$. Our estimate is $\frac{1}{r}(a_1+a_2+\ldots+a_r)$.

The variance of this estimator is

$$\mathrm{Var}\left(\frac{1}{r}(a_1 + a_2 + \ldots + a_r)\right) = \frac{1}{r^2}\left(\mathrm{Var}(\mathrm{a}_1) + \mathrm{Var}(\mathrm{a}_2) + \ldots\right) = \frac{1}{r}\mathrm{Var}(a_1),$$

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 will summarize the assumptions here for future reference:

To get an estimate of $\sum_s f_s^2$ within relative error ϵ with probability at least 0.9999, it suffices to have $r = O(1/\epsilon^2)$ vectors $x^1, x^2, \cdots, x^r$, each with m coordinates of ± 1 with

(i) $E\left(x_s^t\right) = 0$ for all s and t.

(ii) $x^1, x^2, \cdots, x^r$ are mutually independent. This means that for any r vectors $v^1, v^2 \ldots v^r$ with ± 1 coordinates,

$$\Pr\left(X^1 = v^1; X^2 = v^2; \ldots X^r = v^r\right) = \frac{1}{2^{mr}}.$$

(iii) Any four coordinates of x^1 are independent. Same for $x^2, x^3, \cdots, x^r$.

The only drawback with the algorithm as we have described it so far is that we need to keep the r vectors $x^1, x^2, \cdots, x^r$ in memory so that we can do the running sums. This is too space-expensive. We want to do the problem in space dependent upon the logarithm of m, not m itself. If ϵ is in $\Omega(1)$, then r is in $O(1)$, so it is not the r which is the problem; it is the m.

In the next section, we will see that we can do the computation in $O(\log m)$ space by using pseudo-random vectors $x^1, x^2, \ldots$ 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 have 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 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},

$$\Pr\left(x_s = a; x_t = b; x_u = c; x_v = d\right) = \frac{1}{16}.$$

We will see that such a vector may be generated from a (truly) random ``seed'' of only $O(\log m)$ mutually independent 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 each. We assume this here. 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 4 distinct points $a_1, a_2, a_3, a_4 \in F$ and any 4 (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 all computations done over F, we have: $f(a_1) = b_1;\ f(a_2) = b_2;\ f(a_3) = b_3;\ f(a_4) = b_4$.

Now our definition of the pseudo-random vector X with 4-way independence is simple:

Choose f_0, f_1, f_2, f_3 independently and uniformly at random from F. For $s = 1, 2, \ldots m$, let x_s be the leading bit of the k bit representation of $f(s) = f_0 + f_1 s + f_2 s^2 + f_3 s^3$.

Lemma 7.4: The X defined above has 4-way independence.

Proof: Let s,t,u, and v be any 4 coordinates of 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 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, b_4, there is precisely one polynomial f so that

$$f(s) = b_1;\ f(t) = b_2;\ f(u) = b_3;\ f(v) = b_4$$

as we saw above. So, the probability that we have

$$x_s = \alpha; x_t = \beta; x_u = \gamma; x_v = \delta$$

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

■

The lemma states how to get one vector X with 4-way independence. We needed $r = O(1/\epsilon^2)$ of them. Also they all 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 $O(\log m / \epsilon)$ bits. When a symbol s in the stream is read, compute $x_s^1, x_s^2, \ldots x_s^r$. But note that x_s^1 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 several 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.

Exercises

Exercise 7.1: 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 $a = a_1 + a_2 + \ldots + a_i$ and a sample $a_j, j \le i$ 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 an element being proportional to its value.

■

Exercise 7.2: Given a stream of symbols $s_1, s_2, \ldots, s_n$, give an algorithm that will select one symbol uniformly at random from the stream. How much memory does your algorithm require?

■

Exercise 7.3:. 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.

Exercise 7.4: 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.

■

Exercise 7.5: 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. (i) Do the problem when the matrix is given in column order. (ii) 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.

Exercise 7.6: Suppose A,B are two matrices, then show that $AB = \sum_{i=1}^{n} A(:,i)B(i,:)$.

Exercise 7.7: 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.

Exercise 7.7: Show that $ADD^T B$ is exactly

$$\frac{1}{s}\left(\frac{A(:,i_1)B(i_1,:)}{p_{i_1}} + \frac{A(:,i_2)B(i_2,:)}{p_{i_2}} + \ldots + \frac{A(:,i_s)B(i_s,:)}{p_{i_s}}\right)$$

Exercise 7.8: Suppose $a_1, a_2, \ldots, a_m$ are non-negative reals. Show that the minimum of $\sum_{k=1}^{m} \frac{a_k}{x_k}$ subject to the constraints $x_k \geq 0; \sum_k x_k = 1$ is attained when x_k are proportional to $\sqrt{a_k}$.

Exercise 7.9: Show that for a 2-universal hash family $\Pr\left(h(x) = z\right) = \frac{1}{M+1}$ for all $x \in \{1, 2, \ldots m\}$ and $z \in \{0, 1, 2, \ldots M\}$.

Exercise7.10: Let p be a prime. A set of hash functions $H = \{h \mid \{0,1,\cdots,p-1\} \to \{0,1,\cdots,p-1\}\}$ is 3-universal if for all u,v,w,x,y, and z in $\{0,1,\cdots,p-1\}$ the $\text{Prob}\left(h(x)=u, h(y)=v, h(z)=w\right)$ equals $\frac{1}{p^2}$. Is the set of hash functions H 3-univresal?

Exercise 7.10: Give an example of a set of hash functions that is not 2-universal.

Exercise 7.10: (a) What is the variance of the above method 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 7.11: Consider a coin that comes down heads with probability a. Prove that the expected number of flips before a head occurs is 1/a.

Exercise 7.12: Randomly generate a string $x_1 x_2 \cdots x_n$ of 10^6 0's and 1's with probability ½ 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?

Exercise 7.13: Construct an example in which the majority algorithm gives a false positive, i.e., stores a non-majority element at the end.

Exercise 7.14: Construct examples where the frequent algorithm in fact does as badly as in the theorem, i.e., it ``undercounts'' some item by n/(k+1).

Exercise 7.15: Recall basic Statistics on how an average of independent trials cuts down variance and complete the argument for relative error ϵ estimate of $\sum_s f_s^2$.

Exercise 7.16: Let F be a field. Prove that for any 4 distinct points $a_1, a_2, a_3, a_4 \in F$ and any 4 (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 all computations done over F , we have: $f(a_1) = b_1; f(a_2) = b_2; f(a_3) = b_3; f(a_4) = b_4$.