Notes on Randomized Algorithms

William Cohen

November 27, 2017

1 Bloom filters: the basic idea

These notes generally follow the Daniely et al paper from ICLR 2017.[1].

Let $\mathcal{X}_{V,k}$ be the set of all sets X such that X is a subset of V of size at most k. A typical use of X would be to encode the set of words in a document, or a sparse binary feature vectore. Let $d = |V|$ be the vocabulary size (or alternatively the dimension of the feature space).

Let h be a hash function mapping V to a range of integers $\{1, \ldots, m\}$, where $m < d$. The sketch $S_h(X)$ for a set $X \in \mathcal{X}_{V,k}$ is an length-m bit vector, defined as follows:

$$S_h(X) \equiv b_1 \ldots b_m \ : \ b_j = \bigvee_{x \in X} (h(x) = j)$$

Or if you prefer, $S_h(X)$ is computed with the following procedure: start with $\mathbf{s} = 0^m$. Then to construct $S_h(X)$, we just hash each $x \in X$ to a position in $\mathbf{s}$, and then set the bit at that position, ignoring collisions, as shown in the code below:

1. Let $\mathbf{s} = 0^m$ be an binary vector of length m with all the bits cleared.
2. For word $x \in X$ set $\mathbf{s}[h(x)] = 1$.
3. Return $\mathbf{s}$ as $S_h(X)$, the sketch of X.

Since $m < d$ the sketch clearly loses information, relative to X: in particular we can't determine which elements x were in X from S_h, because typically many x's will hash to the same position $h(x)$ in S_h. It turns out that you lose much less information about X if you construct multiple sketches for X. Let's assume we have a set of t hash functions, $h_1, \ldots, h_t$, each of which hash V into $\{1, \ldots, m\}$. From these t hash functions we can create t sketches of X, $S_{h_1}(X)$, $S_{h_2}(X)$, $\ldots$, $S_{h_t}(X)$. Following Daniely et al, we will denote this set of sketches $S_{1:t}(X)$. Sometimes it will be convenient to think of this as a matrix where row j is $S_j(X)$.

Figure 1 gives some examples, where $m = 3$ and (coincidentally) $t = 3$. The t sketches are called a *Bloom filter* for X.

[1] https://openreview.net/pdf?id=r1br_2Kge

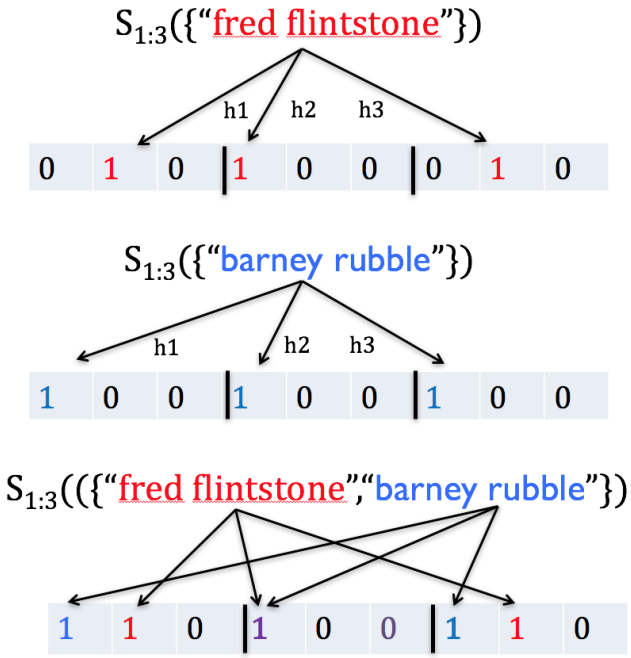

Figure 1: Example of Bloom filters constructed for three small sets.

Let's fix X for now, to simplify the notation, so S_{h_j} can be used instead $S_{h_j}(X)$. We can test if some $x \in X$ using this function:

$$\mathit{contains}(S_{1:t}, x) \equiv S_{h_1}[h_1(x)] \wedge \ldots \wedge S_{h_t}[h_t(x)] = \bigwedge_{i=1}^{t} S_{h_i}[h_i(x)]$$

To understand this function better, try using this function to see which of the sketches in Figure 1 contain "fred flintstone". (The first example tells you that $h_1($ *"fred flintstone"*$) = 2$, $h_2($ *"fred flintstone"*$) = 1$, and etc.) If should be pretty easy to convince yourself of the following:

Proposition 1 *If $x \in X$ then $\mathit{contains}(S_{1:t}(X), x) = 1$*

However, the converse is *not* true: it might be that $\mathit{contains}(S_{1:t}(X), x) = 1$ but $x \notin X$. As an example, consider the first sketch in the figure, $S_{1:t}(\{$ *"fred flintstone"*$\})$. If some $x' \neq$ *"fred flintstone"* happened to hash to the same three indices as "fred flintstone", then $\mathit{contains}(S_{1:t}(X), x) = 1$ would return the wrong answer.

To fix this—or at least, probably fix it—we assume the hash functions are random mappings from V to $\{1, \ldots, m\}$. More precisely, we assume that for all $x \in V$, $i \in \{1, \ldots, t\}$, and all $a \in \{1, \ldots, m\}$,

$$\Pr(h_i(x) = a) = \frac{1}{m} \tag{1}$$

An important point: the probability here, and elsewhere in this document, is taken over our *choice of hash functions*.

We will also assume that the t hash functions are are all drawn *independently* at random. More precisely, for all $x \neq x'$, $x, x' \in V$, all $i, j \in \{1, \ldots, t\}$ and all $a, b \in \{1, \ldots, m\}$,

$$\Pr(h_i(x) = a \wedge h_j(x') = b) = \frac{1}{m^2} \tag{2}$$

It's certainly possible to construct hash functions that satisfy Equations 1 and 2, by fixing V and defining each h_i with random draws. However, it also turns out that you can also use certain deterministic and efficient hash functions[2] in place of random h_i's. In practice, I have found that for Bloom filters, most decent hash functions will provide performance similar to the formal bounds below. (To create a set of t hash functions $h_1, \ldots, h_t$ from a single hash function, one simple trick is to define $h_i(x) = (h(x)\ XOR\ b_i)$ where b_i is a random bit string and XOR is bit-wise exclusive or. Another trick is to let b_i be a random string, and define $h_i(x) = h(b_i\ CONCAT\ x)$ where $CONCAT$ is string concatenation.)

Daniely et al show the following upper bound on the probability of an error using the *contains* routine.

Theorem 1 *If the hash functions $h_1, \ldots, h_t$ satisfy Equations 1 and 2, and if $m = \lceil ek \rceil$ and $X \in \mathcal{X}_{V,k}$, then for any $x' \notin X$,*

$$\Pr(\mathit{contains}(S_{1:t}(X), x') = 1) \ \leq \ \frac{1}{e^t}$$

Proof. For $x' \notin X$, $\mathit{contains}(S_{1:t}(X), x') = 1$ only if for every $i \in \{1, \ldots, t\}$, $S_{h_i}[h_i(x')] = 1$. Let $X = \{x_1, \ldots, x_k\}$, and consider a particular h_i. We claim

$$\begin{aligned}
\Pr(S_{h_i}[h_i(x')] = 1) &= \Pr(h_i(x_1) = h_i(x') \vee \ldots \vee h_i(x_k) = h_i(x')) \\
&\leq \sum_{j=1}^{k} \Pr(h_i(x_j) = h_i(x')) \qquad (3) \\
&= \frac{k}{m} \qquad (4) \\
&= \frac{1}{e}
\end{aligned}$$

Line 3 is because $\Pr(A_1 \vee \ldots \vee A_k) \leq \sum_{i=1}^{k} \Pr(A_i)$, a property sometimes called the "union bound". Line 4 holds because of the random-hash assumption made in Equation 1.

Finally, repeated use of the assumption of Equation 2 produces the overall bound of e^{-t} on an error given in the theorem. □

This is a pretty strong result, as the following corollary emphasizes: if we are willing to accept a small (less than δ) chance of a false positive result on containment queries, we can store a set of elements in less than $3 \ln(1/\delta)$ bits per element, regardless of the size of V.

Corollary 1 *Let $X \in \mathcal{X}_{V,k}$, let $m = \lceil ek \rceil$, and let $t = \lceil \ln \frac{1}{\delta} \rceil$. Then*

- *If $x \in X$ then $\mathit{contains}(S_{1:t}(X), x) = 1$*

[2]For example, see Michael Mitzenmacher and Eli Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, 2005.

- *If* $x \notin X$ *then* $\Pr(\mathit{contains}(S_{1:t}(X), x) = 0) \geq 1 - \delta$

Concretely, for $\delta = 0.01$, this works to about $12.5k$ bits to store a set of size k, and for $\delta = 0.001$, this works to about $19k$ bits. If you decide you can afford $32k$ bits, then you will get to δ down to 0.00001. Of course, all we can do with this set is to create it and then check containment.

2 Bloom filters: variants and extensions

2.1 Variants

Bloom filters are more usually defined by letting all the hash functions hash to a single bit vector (which then must be longer). This works fine and leads to essentially the same formulation, but the math is a little messier (IMHO), which is why I'm using the Daniely et al formulation.

If you'd rather have one long bit vector instead of t short ones, you could also modify the hash functions as follows. Recall h_i needs to produce an integer in $\{1, \ldots, m\}$, so its usually defined by hash into $\{1, \ldots, 2^{32} - 1\}$ to get a position p, and then returning $h_i = (p \bmod m)$. But you could also let $h_i = (i-1)t + (p \bmod m)$, so that each h_i returns indices into a different subrange of $\{1, \ldots, tm\}$.

2.2 Incremental changes and combining filters

A more convenient API would allow you to add and delete elements to a Bloom-filter encoded set. It's very easy to a new element y to a Bloom filter $S_{1:t}$: just compute the positions $h_1(y), \ldots, h_t(y)$, and set the corresponding bits of $S_{h_1}, \ldots, S_{h_t}$. More precisely, the algorithm to convert $S_{1:t}(X)$ to $S_{1:t}(X + \{y\})$ is below.

1. Let $\mathbf{s}_1, \ldots, \mathbf{s}_t$ be the bit vectors associated with $S_{1:t}(X)$.
2. For $i = 1, \ldots, t$, set $\mathbf{s}_i[h_i(y)] = 1$

Figure 2 gives code for this sort of "incremental" Bloom filter.

More generally it's easy to compute $S_{1:t}(X_1 + X_2)$: just compute the bit-wise OR of the sketches in the two filters. This is a really nice property: for instance, if you want to encode a large set, you can split it into subsets and encode the subsets in parallel with different filters, and then combine the subset filters by XOR-ing the sketches. Again, Figure 2 gives pseudo-code for this operation.

Unfortunately, removing elements from a sketch is not easy. One sort-of solution is to maintain a filter $S_{1:t}$ for X and another filter for the set of things removed from X. Call the second filter $\overline{S}_{1:t}$. You can then define containment as

$$\mathit{contains'}(x) \equiv \mathit{contains}(S_{1:t}, x) \wedge \neg \mathit{contains}(\overline{S}_{1:t}, x)$$

This only "sort of" works, however, because if you add x to $S_{1:t}$, remove x by adding it to $\overline{S}_{1:t}$, and then add x back again, *contains'* will be incorrect.

Another case in which Bloom filters can be usefully combined is to build counters that count up to some small integer ℓ. For instance, suppose you want to find all x's that occur at least $\ell = 5$ times in a corpus. You can create five Bloom filters $f_1, \ldots, f_5$ where f_j will (approximately) hold the set of all x's that occur at least j times. Then scan through the corpus using this code:

- For each word occurrence x in the corpus:
  - If $f_4.Contains(x)$ then $f_5.Add(x)$
  - If $f_3.Contains(x)$ then $f_4.Add(x)$
  - If $f_2.Contains(x)$ then $f_3.Add(x)$
  - If $f_1.Contains(x)$ then $f_2.Add(x)$
  - $f_1.Add(x)$

(If you are having trouble understanding this, consider the easier case of $\ell = 2$.) Sometimes this approach is combined with ideas from the "Morris counter" literature.[3]. In a Morris counter, transitions are are done probabilistically, for instance you could imagine this sort of strategy:

- For each word occurrence x in the corpus:
  - If $f_4.Contains(x)$ then with probability 1/16, do $f_5.Add(x)$
  - If $f_3.Contains(x)$ then with probability 1/8, do $f_4.Add(x)$
  - If $f_2.Contains(x)$ then with probability 1/4, do $f_3.Add(x)$
  - If $f_1.Contains(x)$ then with probability 1/2, do $f_2.Add(x)$
  - $f_1.Add(x)$

This sort of counter (called a Talbot-Osborne-Bloom counter) lets you count up to 2^ℓ with only $\log \ell$ filters, with the caveat that you only have a rough idea of the counter values: here for instance if x is contained by f_4 but not f_5, it probably appeared between 16 and 32 times.

3 Countmin sketches

3.1 Countmin sketches: the basic idea

An alternative to counting using multiple Bloom filters is to use a countmin sketch, a construct closely related to a Bloom filter, but is based on real numbers, not bits. While a Bloom filter is a randomized version of a set, a countmin sketch is a randomized version of a vector $\mathbf{a}$—or if you prefer, a dictionary indexed by V that maps an element $x \in V$ to a

[3]For example, see Van Durme, Benjamin, and Ashwin Lall. "Probabilistic Counting with Randomized Storage." IJCAI. 2009.

Bloom filters

Initialize(k, δ)**:**
$m = ek$
$t = \ln(1/\delta)$
// create t bit vectors from $\{0,1\}^m$
$\mathbf{s}_1 = \ldots = \mathbf{s_t} = 0^m$

Add(x)**:**
for $i = 1, \ldots, t$:
$\mathbf{s}_i[h_i(x)] = 1$

AddAll(b)**:**
// b is another Bloom filter
for $i = 1, \ldots, t$:
$\mathbf{s}_i = \mathbf{s}_i \mid b.\mathbf{s}_i$ *// bit-wise OR*

Contains(x)**:**
return $\bigwedge_{i=1}^{t} \mathbf{s}_i[h_i(x)]$

Countmin sketches

Initialize(k, δ)**:**
$m = ek$
$t = \ln(1/\delta)$
// create t real vectors from $\mathcal{R}^m$
$\mathbf{r}_1 = \ldots = \mathbf{r_t} = 0^m$

Increment(x,v)**:**
for $i = 1, \ldots, t$:
$\mathbf{r}_i[h_i(x)] = \mathbf{r}_i[h_i(x)] + v$

IncrementAll(c)**:**
// c is another cm sketch
for $i = 1, \ldots, t$:
$\mathbf{r}_i = \mathbf{r}_i + c.\mathbf{r}_i$

Retrieve(x)**:**
return $min(\mathbf{r}_1[h_1(x)], \ldots, \mathbf{r}_t[h_t(x)])$

Figure 2: Code for incremental versions of a Bloom filter and a countmin sketch

positive real value $\mathbf{a}[x]$. The countmin sketch's main use to is to keep a large number of running sums, one for each element of V.

Figure 2 shows pseudo-code for a countmin sketch. When it is initialized, t real vectors of length m are allocated (instead of bit vectors). The main operations that are supported are to *increment* the value $\mathbf{a}[x]$ associated with x by some positive real quantity v, and to *retrieve* the value associated with x, and they are implemented using analogous methods to their Bloom filter counterparts. To increment $\mathbf{a}[x]$ by v, one finds the t indices associated with x using hash functions $h_1, \ldots, h_t$, and then increments the corresponding sketch elements by v. To retrieve the value for x, one takes the minimum sketch value at these same positions.

It's easy to see that one can implement a Bloom filter using a countmin sketch: you can replace calls to *Add(x)* with *Increment(x,1)* and replace calls to *Contains(x)* with the test (*Retrieve*$(x) > 0$). It's also easy to see that when an incorrect value for x is retrieved, it will always be strictly too large.

Finally, it can be observed that an incorrect value is retrieved only when there has been a collision for every one of the t hash functions—the same circumstance that leads a Bloom filter to return a false positive with a *Contains* call. Put another way:

Corollary 2 *Let c be a countmin sketch. Assume that at most k distinct values of x have appeared in c.Increment*$(j, \cdot)$ *calls. Define* $\mathbf{a}[x]$ *to be the value that should be associated with x, i.e., if the previous calls to c to increment x have been Increment*(x, v_1)*, ...,*

Increment(x, v_J)*, define*

$$\mathbf{a}[x] \equiv \sum_{j=1}^{J} v_j$$

If the hash functions $h_1, \ldots, h_t$ *satisfy Equations 1 and 2, and if* $m = \lceil ek \rceil$ *and* $X \in \mathcal{X}_{V,k}$*, then*

$$\Pr(c.Retrieve(x) \neq \mathbf{a}[x]) \ \leq \ \frac{1}{e^t}$$

Or:

Corollary 3 *Let* c *be a countmin sketch, let* $m = \lceil ek \rceil$*, and let* $t = \lceil \ln \frac{1}{\delta} \rceil$*. Then in the conditions of Corollary 2:*

- *For all* $x \in V$*,* $c.Retrieve(x) \geq \mathbf{a}[x]$
- *For any* $x \in V$*,* $\Pr(c.Retrieve(x) \neq \mathbf{a}[x]) \leq 1 - \delta$

3.2 Countmin sketches: other results

Corollary 2 is a non-standard analysis of countmin sketches. The more usual theorem[4] bases the analysis not on the number of values x that have been incremented in the sketch, but on the sum of all the increments: i.e., thinking of $\mathbf{a}$ as an ordinary vector, what is assumed is not that $|\{x \in V : \mathbf{a}[x] > 0\}| < k$ but that

$$(\sum x \in V \mathbf{a}[x]) < n$$

The quantity $\sum_{x \in V} \mathbf{a}[x]$ is also called the *L1 norm of* $\mathbf{a}$ and is written $\|\mathbf{a}\|_1$. Another bound on error is

Theorem 2 (Cormode and Muthukrishnan) *Let* c *be a countmin sketch, let* $m = \lceil \frac{e}{\epsilon} \rceil$*, and let* $t = \lceil \ln \frac{1}{\delta} \rceil$*. Then in the conditions of Corollary 2:*

- *For all* $x \in V$*,* $c.Retrieve(x) \geq \mathbf{a}[x]$
- *For any* $x \in V$*,* $\Pr(c.Retrieve(x) \leq \mathbf{a}[x] + \epsilon \|\mathbf{a}\|_1) \leq 1 - \delta$

There are number of additional results of this sort. One is for vectors with skewed values. A sequence of values $f_1, f_2, \ldots,$ has a *Zipf distribution* if $f_i \propto i^{-z}$ and $\sum_i f_i = 1$. Famously the frequency of words in English text have a Zipf distribution: the most frequent words account for most word occurrences, and there is a long tail of infrequent words. For $z > 1$, a countmin sketch with width

$$m = \frac{1}{\epsilon^{-z}}$$

[4] Cormode, Graham, and S. Muthukrishnan. "An improved data stream summary: The count-min sketch and its applications." Latin American Symposium on Theoretical Informatics. Springer, Berlin, Heidelberg, 2004.

can approximately encode a Zipf-distributed sequence[5], and similar results hold for some relaxations of the Zipf distribution. The intuition behind these bounds is that most x's will be from the tail of the distribution and will have small values $\mathbf{a}[x]$, and that even t collisions can shown to be inconsequential, if all the collisions involve the "tail" x's: although the value of *Retrieve* will be incorrect the error will be small, since the "tail" x's are associated with small values.

Another interesting result is that one can approximate an inner product, say of two vectors $\mathbf{a}$ and $\mathbf{b}$, using only their countmin sketches. Let $\mathbf{r}_1, \ldots, \mathbf{r}_t$ be the sketch vectors for $\mathbf{a}$ and let $\mathbf{q}_1, \ldots, \mathbf{q}_t$ be the sketch vectors for $\mathbf{b}$. You can approximate

$$\mathbf{a} \cdot \mathbf{b} \equiv \sum_{x \in V} \mathbf{a}[x]\mathbf{b}[x] \approx \min(\mathbf{r_1} \cdot \mathbf{q_1}, \ldots, \mathbf{r_t} \cdot \mathbf{q_t})$$

Finally, like Bloom filters, it is easy to combine two countmin sketches. One can also compute a weighted sum of two sketches: for instance to compute $\alpha\mathbf{a} + \beta\mathbf{b}$ one would create a new sketch with sketch vectors $\mathbf{u}_1, \ldots, \mathbf{u}_t$ where for $i = 1, \ldots, t$,

$$\mathbf{u}_i = \alpha\mathbf{r}_i + \beta\mathbf{q}_i$$

[5]Cormode, Graham, and S. Muthukrishnan. "Summarizing and mining skewed data streams." Proceedings of the 2005 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2005.