1 Introduction

We will continue talking about algorithms that work in very small space, as is required in the streaming model (and other situations were the data is far larger than memory). We will see two more uses of hash functions in dealing with these kinds of situations. In particular, we will see:

• A space-efficient approach for storing summaries, or sketches, of documents so that similarity between documents can be estimated.
• A space-efficient approach for storing a set of items.

2 Locality Sensitive Hashing

For the first application of hashing, suppose you have a large collection $D$ of documents (i.e. web pages or user profiles), you might want to:

• search($D$, $q$): find the document in $D$ most similar to $q$.
• cluster($D$): partition $D$ so that similar documents are in the same cluster.
• removeDups($D$): remove documents that are near duplicates of other documents.

All of these goals require computing $\text{sim}(x, y)$, the similarity between two documents $x$ and $y$ in $D$ for some measure of similarity.

Problem. $D$ is too large to store, and computing $\text{sim}(x, y)$ directly may be too slow.

Idea: Design a smaller “sketch” of the document collection that can be manipulated directly so that we can compute $\text{sim}(x, y)$ using less space.

Definition 1 (A Locality Sensitive Hash (LSH) Family) A set $F$ of hash functions is a locality sensitive hash family for similarity measure $\text{sim}$ if

$$\Pr_{h \in F}[h(x) = h(y)] = \text{sim}(x, y)$$

(1)

for all $x, y$ in the domain of the functions in $F$.

Suppose such a LSH family existed. How could we use it to estimate the similarity $\text{sim}(x, y)$ between two elements $x$ and $y$? The idea is that we could draw $t$ hash functions from this family and compute:

$$\text{sketch}(x) = (h_1(x), h_2(x), h_3(x), \ldots, h_t(x))$$

(2)

$$\text{sketch}(y) = (h_1(y), h_2(y), h_3(y), \ldots, h_t(y))$$

(3)

Then the key insight is that: $\text{sim}(x, y) \approx \frac{\# \text{ times } h_1(x) = h_1(y)}{t}$. Usually, collisions in hashing are bad, but here it has been repurposed to be something good!
The benefit of this is that instead of storing the collection of documents directly, which could be huge, we store only sketch($x$) for each document $x$, and maybe we can make sketch($x$) small by using a family of hash functions with a small range. In fact, if we’re a little flexible with our similarity measure we can always have the hash functions output a single bit in \{0, 1\}.

**Lemma 2** Given a LSH family $F$ for similarity $\text{sim}(x, y)$, then there is an LSH family $F'$ for similarity $\text{sim}'(x, y) = \frac{1 + \text{sim}(x, y)}{2}$ where each $h' \in F'$ outputs a single bit.

**Proof:** Suppose the hash functions in $F$ go from $D$ to some range $X$. Let $B$ be a set of universal hash functions such that $h \in B$ goes from $X$ to \{0, 1\}. Construct $F' = \{b \circ h \mid$ for all $b \in B$ and $h \in F\}$; that is for $h' \in F'$, we have $h'(x) = b(h(x))$. Then there is a collision between $x$ and $y$ using $h'$ when either there is a collision using $h$ or when there is not a collision using $h$ but there is a collision using $b$:

$$\Pr_{b, h \in F'}[b(h(x)) = b(h(y))] = \text{sim}(x, y) + (1 - \text{sim}(x, y))(1/2) = \frac{1 + \text{sim}(x, y)}{2} \quad (4)$$

\[\square\]

### 2.1 A LSH family for the Jaccard Similarity

We haven’t yet shown that that an LSH family exists for any distance measure that is useful. In fact, we will now see that an LSH family can be constructed for the Jaccard similarity. The Jaccard similarity measure is a widely used similarity measure between two sets of items. If $X$ and $Y$ are sets, then

$$\text{Jaccard}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}. \quad (5)$$

The Jaccard is often used to compare documents of text, for example, where $X$ and $Y$ are taken to be the set of words contained each document and the Jaccard measures what fraction of all the words that appear in either document are shared between the two documents.

How can we construct an LSH family $F_{\text{Jaccard}}$ for Jaccard? We will see one introduced by Broder\(^1\)

For simplicity, let’s assume that our sets (documents $X$ and $Y$) are subsets of $[M] = \{1, \ldots, M\}$ for some $M$. If this is not true, we just map all of our elements to a unique number in $[M]$ (or else we can assume that our real universe of elements in the sets can be totally ordered). To define $F_{\text{Jaccard}}$, we will use the concept of a minwise independent permutation family:

**Definition 3 (Minwise independent permutation family)** A set $P$ of permutations on $[M]$ is **minwise independent** if, for any set $X \subset [M]$ and any $e \in X$, we have:

$$\Pr_{\pi \in P} \left[ \min_{x \in X} \{\pi(x)\} = \pi(e) \right] = \frac{1}{|X|}. \quad (6)$$

This definition says that all elements $e \in X$ have equal chance to become the minimum when shuffling according to a randomly chosen permutation $\pi$ in $P$. The set of all permutations is minwise independent since any element could become the minimum.

---

We now define:

\[ F_{\text{Jaccard}} = \left\{ h_\pi(X) = \min_{x \in X} \{ \pi(x) \} \text{ for all } \pi \text{ on } [M] \right\} \quad (7) \]

That is, each member of our LSH family of hash functions maps a set to its minimum element under a randomly chosen permutation of the elements of our universe.

**Theorem 4** Let \( h \) be a randomly chosen function from \( F_{\text{Jaccard}} \). Then

\[
\Pr[h(X) = h(Y)] = \Pr \left[ \min_{x \in X} \{ \pi(x) \} = \min_{y \in Y} \{ \pi(y) \} \right] = \frac{|X \cap Y|}{|X \cup Y|} = \text{Jaccard}(X, Y). \quad (8)
\]

**Proof:** Let \( \alpha = \min_{a \in X \cup Y} \{ \pi(a) \} \). Then \( \min_{x \in X} \{ \pi(x) \} = \min_{y \in Y} \{ \pi(y) \} = \alpha \) if and only if \( \alpha \in \pi(X \cap Y) \) since each element is assigned a unique position by \( \pi \). In other words,

\[
\Pr[\min_{x \in X} \{ \pi(x) \} = \min_{y \in Y} \{ \pi(y) \}] = \Pr[\alpha \in \pi(X \cap Y)] \quad (9)
\]

\[
= \sum_{e \in X \cap Y} \Pr \left[ \min_{a \in X \cup Y} \{ \pi(a) \} = \pi(e) \right] \quad (10)
\]

\[
= \frac{1}{|X \cup Y|} \quad \text{by minwise independence} \quad (11)
\]

\[
= \frac{|X \cap Y|}{|X \cup Y|} \quad (12)
\]

Summary of the MinHash algorithm Choose some number \( t \) of hash functions \( h_1, \ldots, h_t \) from \( F_{\text{Jaccard}} \). The above shows that each \( h \in F_{\text{Jaccard}} \) has a collision with probability equal to the Jaccard similarity. The sketch(\( X \)) of each document \( X \) is just the vector of the application of these \( t \) hash functions to \( X \).

To estimate \( \text{Jaccard}(X, Y) \), use \( \text{Hamming}(\text{sketch}(X), \text{sketch}(Y))/t \), where the Hamming function is the number of dimensions in which two vectors differ.

**How many samples do we need?** The last question is: how large must \( t \) be for the error to be small. Intuitively, the error decreases rapidly with \( t \) (the estimate when \( t = 1 \) is horrible, but at \( t = 100 \) the number of collisions is probably pretty close to the true probability of collision). To get a better handle on this, we can use Chernoff bounds, which are like Markov bounds but with more power.

**Theorem 5 (Chernoff bounds, simple form)** Let \( B \) be the sum of independent 0/1 random variables \( B_i \). Let \( \mu = \mathbb{E}B \). Then

\[
\Pr[B \geq (1 + \delta)\mu] \leq \exp \left( \frac{-\delta^2 \mu}{3} \right) \quad (13)
\]

\[
\Pr[B \leq (1 - \delta)\mu] \leq \exp \left( \frac{-\delta^2 \mu}{2} \right) \quad (14)
\]

for any \( 0 < \delta < 1 \).
In our case, let
\[ B_i = \begin{cases} 1 & \text{if } h_i(X) = h_i(Y) \\ 0 & \text{otherwise} \end{cases} \] (15)
and let \( B = \sum_i B_i \). Our estimate of the similarity is \( B/t \) and \( \mu = t \cdot \text{Jaccard}(X,Y) \). Substitute these values into (14) and we get:
\[
\Pr \left[ B \leq (1 - \delta) t \cdot \text{Jaccard}(X,Y) \right] \leq \exp \left( -\frac{\delta^2 \text{Jaccard}(X,Y)}{2t} \right) \] (16)
Setting \( t = 2/(\delta^2 \text{Jaccard}(X,Y)) \ln(1/\epsilon) = O(\log \frac{1}{\epsilon}) \) gives the probability of error to be \( \leq \epsilon \). The case of overestimating (13) is almost exactly the same.
(Note that this bound actually depends on the size of the Jaccard distance. This makes sense: very small Jaccard distances are going to need a large number of samples to get within a small multiplicative factor. In practice, one can often assume a reasonable lower-bound on the Jaccard distances you really care about.)

2.2 A LSH family for a cosine-related distance

Often documents are not represented by sets but by vectors. For example, a webpage could be represented by a vector \( v \) that has an entry for each keyword and contains 1 in that entry if the keyword is present and 0 if not. A good measure of the distance (dissimilarity) between two documents \( u, v \) is then the angle \( \theta(u,v) \) between the vectors that represent them, or some function of that angle. This distance is frequently used in information retrieval applications. We will see a LSH family \( F_{\text{AngleSim}} \) for the measure:
\[
\text{AngleSim}(u,v) = 1 - \frac{\theta(u,v)}{\pi} \] (17)
between two vectors \( u \) and \( v \).

The main idea is to choose a random hyperplane \( r \) and define a hash function as:
\[
h_r(x) = \begin{cases} 1 & r \cdot x \geq 0 \\ 0 & r \cdot x < 0 \end{cases} \] (18)
We then take \( F_{\text{AngleSim}} = \{ h_r(x) \} \) for all choices of random hyperplanes \( r \).

**Theorem 6** When \( h \) is drawn randomly from \( F_{\text{AngleSim}} \), we have:
\[
\Pr_{\hat{h}}[h(u) = h(v)] = 1 - \frac{\theta(u,v)}{\pi} = \text{AngleSim}(u,v) \] (19)
for all \( u, v \).

**Proof:** The probability that a random hyperplane splits \( u \) and \( v \) is \( \frac{2\theta(u,v)}{2\pi} \), where the factor of 2 comes from the fact that we can specify the hyperplane is two different ways:

---

2This LSH family was introduced by Charikar, “Similarity Estimation Techniques from Rounding Algorithms” in STOC 2002 pp. 380-388
The arguments about how many samples you need for Jaccard work just as well for AngleSim.

3 Bloom filters

Let’s change gears to a completely different problem, but one where (a) hashing still provides an elegant solution and (b) space is at a premium. Suppose we want to maintain a set $S$ that is a subset of a very large universe $U$ of possible items. We want our storage to be small, and we want to support the following two operations:

1. $\text{add}(S,x)$: put item $x$ into our set $S$
2. $\text{query}(S,x)$: return $\text{true}$ if $x$ has been added to $S$ in the past.

Notice that we do not need to support deletion of elements.

As with our discussion of $\epsilon$-heavy hitters, in this case we will allow for some error. Specifically, if $x \in S$ we must return $\text{true}$ on $\text{query}(S,x)$. However, we allow some false positives: if $x \notin S$ we can wrongly return $\text{true}$ when $S$ is queried for $x$. Of course, we want to make this false positive rate (FPR) as small as possible.

What is an application of the above data structure? Suppose we have a very long list of websites that host malicious code. Whenever a user visits a website we would like to quickly check whether that website is one of the malicious ones. Over time, new malicious sites will be discovered and need to be added to our list. Here, we can take $S$ to be the set of bad websites, and when a new one is discovered we can add it to $S$. At every visit to a site, we use query to check whether the site is malicious. We can tolerate some false positives, since these will just result in a warning to the user that they can ignore (or a more expensive check to see if the site really is malicious). We can’t tolerate false negatives, since saying a website is safe when it is known to not be is very bad.

The Bloom filter is one solution to this problem\(^3\). Bloom filters have been widely used for network applications as well as for problems in genomics. Apparently, the Chrome browser (in some version) used a Bloom filter for exactly the malicious-website-tracking application described above. This allows them to ship a small data structure that stores their long list of bad websites. Bloom filters also have the advantage that it is not possible to easily extract the actual items contained within them — this allows the list of bad websites to be kept somewhat secret.

3.1 How do Bloom filters work?

A Bloom filter is simply an array \( B \) of \( m \) bits, plus \( t \) hash functions \( h_1, \ldots, h_t \) where each \( h_i : U \to \{0, \ldots, m - 1\} \). We’ll see how to choose \( m \) and \( t \) shortly. \( B \) is initially set to all zeros.

1. \texttt{add}(S, x) — set all bits given by the hash functions to 1:
   
   \[
   \text{for } i = 1 \text{ to } t \\
   \quad B[ h_i(x) ] = 1
   \]

2. \texttt{query}(S, x) — return \texttt{true} if all the bits given by the hash functions are 1:
   
   \[
   \text{for } i = 1 \text{ to } t \\
   \quad \text{if } B[ h_i(x) ] == 0 \\
   \quad \quad \text{return false} \\
   \quad \text{return true}
   \]

3.2 How big of a bitvector and how many hash functions do we need?

It’s clear that if we’ve added \( x \) to \( S \), then \texttt{query}(S, x) will always be true, so we have no false negatives. But it might be the case that, due to collisions, all the bits for \( x \) were set by other elements even though \( x \) was never added. This will lead to a false positive. So the question is: how often does that happen? That depends on \( m \) and \( t \), so we now see how to choose these parameters to minimize the error rate.\(^4\)

**Theorem 7** The probability of a false positive in a Bloom filter is minimized by choosing \( t = \left( \frac{m}{n} \right) \ln 2 \), where \( n \) is the number of items added to the filter.

**Proof:** After a single item is added, the probability that a particular bit is 0 is \((1 - \frac{1}{m})^t\). Since the hash functions can be viewed as “randomly” flipping bits, we can assume that the bits being flipped are independent. (This is not strictly true, and a more sophisticated analysis can remove the assumption.) So after \( n \) items have been added, the probability that a bit is set to 1 is:

\[
1 - \left( 1 - \frac{1}{m} \right)^{tn}
\]  \hfill (20)

We get a false positive when all \( t \) bits are set for a given element \( x \). The probability of this is:

\[
\text{Pr}[\text{FP}] = \left( 1 - \left( 1 - \frac{1}{m} \right)^{tn} \right)^t
\]  \hfill (21)

\[
= \left( 1 - \left( 1 - \frac{1}{m} \right)^{m \frac{tn}{m}} \right)^t
\]  \hfill (22)

\[
\approx \left( 1 - e^{-\frac{tn}{m}} \right)^t
\]  \hfill (23)

where we’ve used the fact that \((1 - 1/m)^m\) is about \( e^{-1} \). The above expression is minimized by choosing \( t = \frac{m}{n} \ln 2 \).

Theorem 8 Let $\rho$ be a given false positive probability. A Bloom filter with FPR $\rho$ containing $n$ items can be achieved with $O(n \log \frac{1}{\rho})$ bits.

**Proof:** We will use the optimal number of hash functions from the previous theorem. Substituting that into (23), we have:

\[
\rho \approx \left(1 - e^{-((m/n) \ln 2)n/m}\right)^{(m/n)\ln 2}
\]

(24)

\[
= \left(1 - e^{-\ln 2}\right)^{(m/n)\ln 2}
\]

(25)

\[
= \left(\frac{1}{2}\right)^{(m/n)\ln 2}
\]

(26)

Solving for $\rho$:

\[
\ln \rho = \ln 2^{-{(n/m)\ln 2}} = -(m/n)(\ln 2)^2
\]

(27)

Therefore, a given $\rho$ error probability can be achieved by setting $m = \frac{n \ln(1/\rho)}{(\ln 2)^2}$. ■

Thus, we need only $O(\log \frac{1}{\rho})$ bits on average if we are willing to tolerate $\rho$ chance of a false positive.

3.3 Notes and Extensions to Bloom Filters

**Where are the items?** Note that a Bloom filter stores a set of items without ever storing the items. The access time is always $O(tH)$, where $H$ is the time to compute the hash value, independent of the number of items in the filter.

**Computing the union of two sets.** Suppose we have Bloom filters $B_1$ and $B_2$ representing two sets. We can assume that $B_1$ and $B_2$ use the same set of hash functions. How can we compute $B_1 \cup B_2$? This is very easy: just compute the bitwise OR of $B_1$ and $B_2$. The resulting vector has 1 bits everywhere that would have been set if the union of the items were directly inserted into it.

**Deletion.** It’s not possible to delete an element from a Bloom filter without possibly introducing false negatives (missed elements). This is because if two elements $x$ and $y$ both set bit $i$ to 1, setting it to 0 when deleting $x$ will also delete $y$. One partial solution to this is to maintain two Bloom filters: the regular filter that contains all the items ever added to $S$ and a deletion filter $D$ that contains all the items ever deleted from $S$. Then $\text{query}(S, x)$ returns true if $x$ is in $B$ but not in $D$, but now false positives in $D$ correspond to false negatives.

**Changing the size of a Bloom filter.** In general, it is not possible to resize a Bloom filter after items have been added to it since this would move the targets of the hash functions around. However, if the filter length is a power of 2, it’s size can be (repeatedly) reduced by a factor of 2. To do this, we “fold” the left half of the filter onto the right half, “OR”ing them together. We then modify our hash functions to throw away their high-order bit, also reducing their range by a factor of 2. So, if it is difficult to guess how many items will be in the filter, one can start out with a large filter and halve it repeatedly as it becomes clear that fewer items will be stored. (Unfortunately, *doubling* a filter is not as easy.)