In some situations we are given a very long string, say billions of characters long, and we want to find all words that appear more than once. A reasonably efficient way of doing that is to create a large hash table, keyed by words which stores the count for each observed word. This gives us a linear time algorithm in the length of the input.

However, in many cases the number of different words that appear in the input is very large but most of them occur only once (mis-spellings, people’s names etc). This single-occurrence words or singletons place a large demand on the computer memory while containing no useful information.\(^1\)

What we need is a filter. This filter will receive as input the stream of words, one word at a time. For each word it will answer the question “did this word appear earlier in the stream?”. If this is the first time the word appears, then it is filtered out or ignored. If the word has appeared earlier then it is filtered in or passed on to the hash table holding the counters. We would like to find a method which uses much less memory than would be used by the hash table. Bloom filters provide an elegant solution to this problem, but with a slight caveat: while no word that appears more than once will be mistakenly filtered out, the method does allow a small fraction of the singletons to be filtered in.

We now describe Bloom filters. Initially, two integer parameters \(k, m\) are chosen (how to choose it will be described a little later). We then choose and fix \(k\) different hash functions \(h_1, h_2, \ldots, h_k\) that map words to integers in the range \(1, \ldots, l\). We also allocate a bit vector \(B\) of length \(m\) where all bits are initialized to zero.

The filter operates as follows. Given a word \(w\), it computes the \(k\) numbers \(h_1(w), h_2(w), \ldots, h_k(w)\) and uses them as indices into the bit vector \(B\). If all of the \(k\) bits \(B[h_1(w)], B[h_2(w)], \ldots, B[h_k(w)]\) are equal to 1 then we declare that word \(w\) did appear earlier in the stream and therefore \(w\) is filtered in. If any of the \(k\) bits is not 1 then the word \(w\) is filtered out and not counted and the \(k\) bits in \(B\) are set to 1.

We now want to analyze the probability that the Bloom filter makes a mistake. There are two types of mistakes: filtering out a word that appeared previously and filtering in a word that did not appear previously.

We consider each error type in turn:

- **Filtering out a word that appeared previously** (false negative) This can never happen. If the word \(w\) appeared in the past then the bits \(B[h_1(w)], B[h_2(w)], \ldots, B[h_k(w)]\) have been set to 1. As the algorithm never resets bits to zero, these bits must still be all 1 when we encounter \(w\) for the second, third, … time. As a result \(w\) will not be filtered out. The Bloom filter does not make false negative mistakes.

- **Filtering in a word that did not previously appear** (false positive) This can happen. The \(k\) bits that are checked might have been set to 1 as a result of observing other words. However, we will now show that the probability of this event is small (provided \(k\) and \(m\) are set appropriately. Note also that the cost of a false positive mistake is small - it means that the algorithm will unnecessarily store a singleton word in the hash table. The result is a waste of memory space but not an actual error.

\(^1\)Distributions where a significant fraction of the items (words) in a random sample appear only once, are called Zipf distributions. Zipf distributions are prevalent whenever a very large and under-utilized set of labels is used. This includes words, URLs, IP addresses etc. You can think of Zipf distributions as lying in the mid-point between discrete distributions (over a finite set) and density distributions. In the first case we expect all values to appear many times in a large enough sample, while in the second case we don’t expect to see any value more than once.
We now analyze the probability of making a false positive mistake, i.e. incorrectly declaring that a new word appeared earlier in the sequence. Let $n$ be the number of different elements (words) that we inserted into the filter before we test the new word. We assume that each hash function $h_i(w)$ is a number chosen uniformly at random from the range $1 \leq i \leq m$. Consider a particular location $j$ in the bitvector $B$, which is one of the $k$ locations that the new word $w$ is mapped to. We want to compute the probability that this bit is not set to one. The probability that one of the $k$ hash functions, operating on one of the previous $n$ words, does not set the $j$th bit to one is

$$1 - \frac{1}{m}$$

Thus the probability that none of the $k$ hash functions, operating on any of the $n$ words sets the $j$th bit to one is

$$\left(1 - \frac{1}{m}\right)^{kn}$$

Thus the probability that the $j$th bit is set to one is

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Finally, the new word will be identified as new only if all of the $k$ locations in $B$ to which it is hashed have been set to one. As these locations are independent, we get that the probability of making a false positive mistake is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = \left(1 - \left(\left(1 - \frac{1}{m}\right)^{m/n}\right)^{kn/m}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Note that the only way that $m$ and $n$ enter the equation is through the ratio $m/n$. We call the ratio $r = m/n$ the redundancy of the bitmap, because it defines the number of bits that are associated with each word. And we can rewrite the (approximate) probability of a false positives as

$$\left(1 - e^{-k/r}\right)^k$$

The number of hash functions $k$ that approximately minimizes the probability is (remember that $k$ is an integer)

$$k \approx r \ln 2 \approx 0.7r$$  \hspace{1cm} (1.1)

which gives the false positive probability of

$$p = \left(1 - e^{-\ln 2}\right)^k = (1/2)^k \approx (0.6185)^r.$$  

The required redundancy for a desired false positive probability $p$ (assuming the optimal value of $k$ is used) can be computed by taking the ln f the two side in the last expression

$$\ln p = -r(\ln 2)^2.$$  \hspace{1cm} (1.2)

Recalling that $r = m/n$ we get that the length of the bit vector is

$$m = \frac{-n \ln p}{(\ln 2)^2}.$$  

To gain some intuition about these results, let’s compare the performance for the optimal $k \approx m/n$ defined in Equation (1.1) with the performance for $k = 1$. The case $k = 1$ is very intuitive, each word is mapped to
a single bit and if this bit is one, then the algorithm concludes that the word has been seen before. As the length of the bit vector \( B \) is \( m \) and the number of different words already observed is \( n \) then \( n \) of the \( m \) bits in \( B \) are set. The result is that the probability of making a false positive mistake is \( p = n/m \). If we had a perfect hashing function, that maps each word to a different bit, we would be able to use a table with no redundancy, i.e. \( r = 1 \). As we showed above when \( k = 1, p = 1/r \).

Consider now using the optimal setting for \( k \) as defined in Equation (1.1). In this case we have from Equation (1.2) that:

\[
p = \exp \left( -\frac{m}{n} (\ln 2)^2 \right) \leq \exp \left( -0.48 \frac{m}{n} \right) = \exp \left( -0.48r \right)
\]

We find that in both cases the probability of a false positive is a function of the redundancy, however, while for \( k = 1, p \) decreases like \( 1/r \) for the optimal value of \( k \) it decreases much faster, like \( e^{-r} \). Thus with the same size bit vector we get a much smaller probability of error.