Streaming Algorithms

Data Sampling

Debugging a big data program is usually tedious, so instead of storing the streams, data is sampled.

1. Store 1/10 of the stream, by generating a random integer from 0 to 9 and store when 0 is encountered
    a. Causes bias
    b. Causes problems when examples have been repeated
2. Sample 1/10th of the users, not the transactions
    a. Each time a search query arrives, see if user exists in sample
    b. If so, add query to sample
    c. If not, generate a random number and pick the user if the number is 0
    d. To avoid storing a list of users, we can hash the userid from 0-9 and select if the user hash is 0

The general algorithm is to identify the key components of the query, hash them in some range 0 to b. Get a sample size a/b by selecting a query if its hash is lesser than a.

Data Filtering

We take a general example of a spam filter. Storing all spam email addresses in memory is not convenient, as disk access is slow and disk storage is limited. We use a Bloom Filter where

1. Hash the non spam email IDs to 0 - 8 billion and set the corresponding bit to 1
2. Hash the incoming mail ID
3. If the corresponding bit is 0, reject
4. If bit is 1, maybe spam, may not be spam, so call it not spam

Cascaded Bloom Filters can be used together to make a series of hashes for efficient spam detection as well.

A bloom filter in general contains

1. Array of n bits
2. A collection of k hash functions
3. A set S of keys with m elements
4. Given a key a, determine if it is in S

Initialisation is done by computing the k hash functions and setting the corresponding bits to 1.

Usage is hash(a) and then observing if the corresponding k bits are 1.

The probability of a false positive is given by $(1 - e^{-km/n})^k$.

Counting Distinct Elements

The problem to solve here is for example, how many different users visit each webpage of a website, given User x Webpage combinations.

For this, the Flajolet-Martin algorithm is used by

1. Picking a hash function bigger than the set to be hashed.
2. Hash element in the stream
3. Let R be the number of trailing 0s in the hash or the tail length.
4. 2^R is approximately the number of distinct elements seen

This basic property works because

P(h(a) ends in at least r 0s) = 2^{-r}.

This can be seen by

1. Suppose the hash is $h_1h_2....h_n$.
2. Probability that a bit is a 0 is 0.5.
3. Probability that h_n is 0 is 2^{-1}.
4. Probability that the last 2 bits are 0 is $2^{-1}x2^{-1} = 2^{-2}$
5. Probability that last r bits are 0 is hence 2^{-r}.

Hence, for m distinct elements, this probability that no element has tail length r is $(1 - 2^{-r})^m$.

This probability $(1 - 2^{-r})^m$ can be generalised to e^{-mx} where $x = 2^{-r}$.

P(At least one element has tail r) = $1 - e^{-mx}$.

When m >> 2^r, e^{-mx} approaches 0, which means $1 - e^{-mx}$ is almost 0, which says we’re most likely to find an element with a tail length r.

When $m \sim 2^r$, there is some probability of finding tail lengths of r.

When $m << 2^r$ it is highly unlikely that we are going to find elements with tail length r.

We can implement this filter in two ways

1. Simple approach
    a. If we have only one hash, m will always be a power of 2.
    b. We can pick k hash functions and estimate $m = 2^R$ for each and take average or median
    c. Better estimate
    d. But average might be pulled towards max (outliers)
    e. Median will give the estimate as a power of 2 which is undesirable
2. Combined approach
    a. Divide k hashes into groups
    b. Compute average of each group
    c. Take median of averages