Assignment 2 - CS110

Section 1 - Overview

Introduction

A Bloom filter is a probabilistic data structure that is used to test whether an element is a member of a set.

A set is a collection of distinct objects in which order has no significance. Elements are defined as members of a set. Sets are typically collections of numbers, though a set may contain any type of data (including other sets).

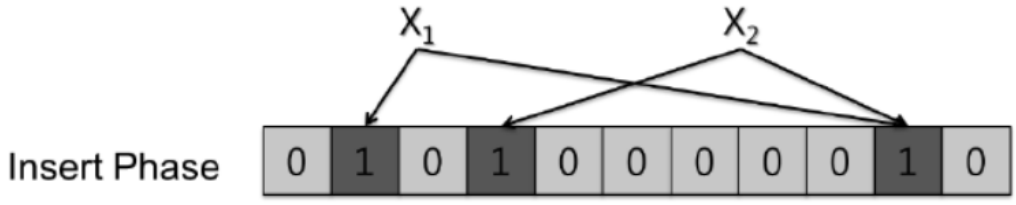
False positive matches are possible, but false negatives are not. In other words, a Bloom filter tells us with absolute certainty that an element isn't in a data set, however there is a probability of error if the filter tells us that an element *is* in a dataset. Later on, we will learn about how we can manipulate the error rate.

Algorithm

The Bloom filter structure consists of a bit array of size *n*, initially all set to 0. *k* independent hash functions are randomly defined to map a set element into one of the n array positions, generating a uniform random distribution (for more details, check Theoretical Analysis). Typically, *k* is much smaller than n, which is proportional to the number of elements to be added to the filter (m) . An empty Bloom filter has all its bit sets to 0.

The operations supported by the original Bloom filter are as follows:

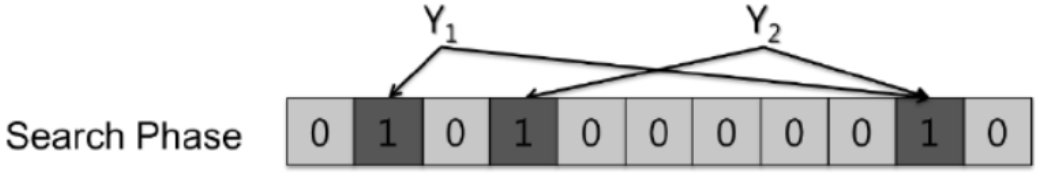
- *Insertion*: The filter applies the k hash functions to an incoming elements and sets the bit in the positions returned by the hash functions to 1. Consider the following diagram, where x_1 and x_2 are incoming elements and there are two hash functions on an array of size 11.



Out[ ]= **Figure 1**. Inserting elements into a Bloom filter.

- *Query*: Querying for an elements involves applying the same *k* hash functions and testing whether all the bits in the positions returned by these functions are set to 1. Consider the diagram below, where

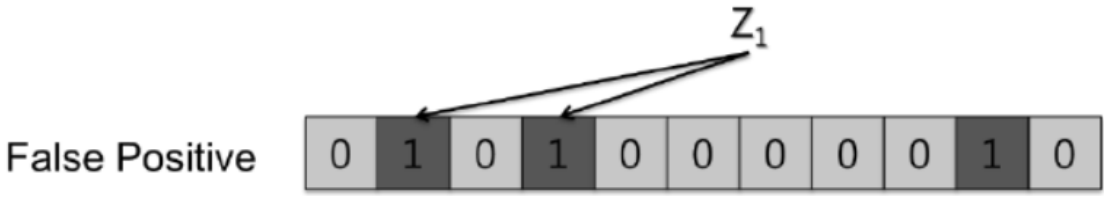
we are querying whether y_1 and y_2 exist in the filter.



Out[ ]= **Figure 2**. Querying elements in a Bloom filter.

If any of the bits at these positions is 0, the element is definitely not in the set. In other words, false negatives are impossible in the Bloom filter.

If all the positions have a bit value of 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. For instance, in the diagram below, the positions corresponding to z_1 are filled, but this is because the bit value was set during the initial insert phase of x_1 and x_2. z_1 yields a positive answer even though it doesn't exist in the Bloom filter.



Out[ ]= **Figure 3**. False positives in a Bloom filter.

It is important to note that as the number of elements in the set grow, the probability of false positives increase. When the false positive rate gets too high, the filter can be regenerated but this should be a relatively rare event.

Deletion operations are not not supposed by the original bloom filter. This is because more than one element can set a bit in the table to 1, as we saw in Figure 1. Therefore, if we switch a bit to 0, we risk deleting a number of other elements as well.

A summary of the disadvantages of using Bloom filters are:

- They are appropriate for fast inserts and queries. However, we can't store an associated object.
- Deletions are not supported.

- There is a small false positive probability.

Theoretical Analysis

Probability of False Positives

Consider m as the number of items we will insert in total, n as the size of the bloom filter and k as the number of hash functions. For theoretical analysis, we make the assumption that for each choice of hash function h_i and possible object x, $h_i(x)$ maps each x to a random number with equal probability over the range [0,n-1] and the output is independent from all other hash functions.

The probability that a given bit of the Bloom filter is 0 i.e. has not been set to 1 after the data set n has been inserted is $\left(1 - \frac{1}{n}\right)^{k \times m}$. A dart game is a useful analogy to explain the intuition behind this probability.

On an insertion, k darts, uniformly random and independent of each other, are thrown at the Bloom filter. Any position that a dart hits is set to 1. For the bit to remain 0, it has to be missed by all k darts thrown at the dart table.

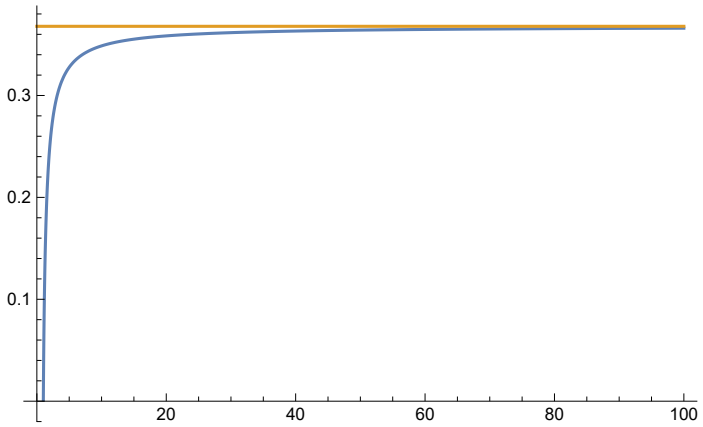
A given bit flip is uniformly likely to happen to any of the n bits, therefore it's associated with a probability of $\frac{1}{n}$, where n is the size of the hash table. The probability that the bit flip will not happen for a particular position is then $1 - \frac{1}{n}$. This is the chance of surviving a single dart (analogous to one hash function). In the case of k hash functions over m items inserted into the Bloom filter, the probability that a bit has not been set to 1 is $\left(1 - \frac{1}{n}\right)^{k \times m}$. Therefore, the probability that a given bit of the Bloom filter *has* been set to 1 after the m items are inserted is $1 - \left(1 - \frac{1}{n}\right)^{k \times m}$.

Since $\left(1-\frac{1}{n}\right)^{n} \approx 1/e = e^{-1}$ (see graph below), we can approximate $\left(1-\frac{1}{n}\right)^{km}$ by:

$$\Rightarrow \left(1-\frac{1}{n}\right)^{k\times m} = \left(\left(1-\frac{1}{n}\right)^{n}\right)^{\frac{km}{n}} \approx \left(e^{-1}\right)^{\frac{km}{n}} = e^{\frac{-km}{n}}$$

In[236]:=
```
Plot[{(1-1/n)^n, E^-1}, {n, 0, 100}, PlotRange→All]
```

Out[236]=



Therefore, the probability that one given bit of the Bloom filter *has* been set to 1 after the m items are inserted can be expressed as $1 - e^{\frac{-km}{n}}$.

The probability of a false positive (ϵ) i.e. all k bits of a new element are already set is the probability that a given bit of the Bloom filter has been set to 1 exponentiated by the number of hash functions in total, indicating that the bit has to be set to 1 k different times: $\in = \left(1 - e^{\frac{-km}{n}}\right)^{k}$.

Optimal k values

When picking the optimal number of hash functions, we have to make a tradeoff: the more hash functions we have, we are more likely to find a 0-bit for an element that is not a member of the set but the slower the Bloom filter implementation and the quicker it fills up (sacrificing space). However, if we have too few hash functions, we will reuse bits for more and more different objects and false positives will be more likely (sacrificing correctness).

Our motivation is to optimize the number of hash functions for a given m and n such that we minimize the false positive probability rate ϵ. We computed in the previous section that $\in = \left(1 - e^{\frac{-km}{n}}\right)^{k}$. The strategy for this problem is to take the derivative of ϵ with respect to k and find the point at which the derivative is a global minimum. The derivative is 0 when k = $\ln(2) \times \frac{n}{m}$. Although we will not derive this result ourselves, the main intuition is that the optimal

value of k grows linearly with the number of bits per element in the Bloom filter, since $\frac{n}{m}$= $\frac{\text{Size of bloom filter}}{\text{Number of items inserted}}$is the number of bits per element (b) in the Bloom filter.

History

Burton H Bloom conceived the Bloom filter data structure while working with databases at the Computer Usage Company in Massachusetts (Bloom, 1970). His general objective was to find efficient methods and algorithms to (quickly) deal with vast amounts of hash-coded data considering the following computational factors:

- *Space*: Size of the hash area.
- *Reject time*: Time required to identify a nonmember of a set.
- *Allowable error frequency*

Applications of Bloom Filters

Any application that requires fast lookups, has an allowable false positive rate and where space is a consideration is suitable for Bloom filters.

In example, Google Chrome previously used Bloom filters to identify malicious URLs. Any requested URL would be first checked against a local Bloom filter. If the query result was positive, then a full check was conducted against the real blacklist. This saved a lot of time since the latter check is a very costly operation.

Another use case is the ever-so-popular online dating app, Tinder. Tinder populates each user's application with a list of women/men near current location. A Bloom filter is used to record right swipes i.e. when a user expresses interest in another user, and any repeats are removed. Each time more users are fetched from the server, the Bloom filter is applied and any ID's that have been previously swiped right are filtered out. Is there a chance that some users are filtered out even before we have a chance to see them? The answer to this questions truly depends on the false positive rate!

References

Bloom, B. H. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun. ACM, 13(7), 422–426. https://doi.org/10.1145/362686.362692