Similarity Filters
An Introduction

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Rick Marker
rdam@itu.dk

April 22nd 2014

1 Introduction

In the following we investigate the Similarity Search problem. Different approaches have been explored by earlier work, and in this report we will investigate two of those; 'Distance-Sensitive Bloom Filters'[1] by Kirsch and Mitzenmacher, and 'Locality-Sensitive Bloom Filter for Approximate Membership Query'[3] by Hua et al.

1.1 The Problem

The Similarity Search problem can be stated as follows: Given a set of elements S, determine if there exists an element $s \in S$ that is 'close' to a given query element q. Where 'close' is defined as being within a given distance, d using a certain metric.

In the following we consider the Similarity Search problem for bit-vectors of length l, and Hamming distance as metric. This reduction maintains a high degree of generality, since elements from many domains can be encoded as bit-vectors.

1.2 Definitions

Hamming distance

A distance metric for bit-vectors. The Hamming distance between two vectors v_1, v_2 is defined as the number of positions i where $v_1[i] \neq v_2[i]$.

Bloom Filter

A Bloom filter is an inexact representation of a set that allows for false positives when queried; that is, it can sometimes say that an element is in the set when it is not. In return, a Bloom filter offers very compact storage: less than

10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set.[2]

Locality-Sensitive Hashing

Regular hashing tries to spread out the hash-values of different elements, to minimize the probability of a collision. Locality-Sensitive Hashing (LSH) tries to group similar elements, by maximizing the collision probability for similar elements.

The LSH is closely tied to the distance metric, and many distance metrics have no known LSH. The Hamming distance metric on bit-vectors has a particularly simple LSH: Sample a fixed number of bits from the input vector, uniformly at random.

It is intuitively obvious, that if two elements only differ on the non-sampled bits, then they will hash to the same value, and thus be considered "close" by the LSH.

The more bits two vectors have in common, the higher the probability will be that a random LSH will hash them to the same value.

ϵ-closeness

- If an element $s \in S$ differs from x in at most an ϵ-fraction of the bits, it is said to be ϵ-close to x. The data structure must return "close".
- If every vector $s \in S$ differs from x in at least a δ-fraction of the bits, then the data structure should return "not close".
- In all other cases, i.e. when the distance to the nearest vector is between ϵ and δ, the data structure can give any answer it likes.

1.3 Naïve Approaches

Brute force The most obvious idea for solving the Similarity Search problem, is brute force. Store the elements S in a linked list. When a query is made, simply scan the linked list, and calculate the distance from each element $s \in S$ the query element q. If s satisfies the distance requirement, a match has been found. If the end of the linked list is reached, no match exists.

This will give a correct answer, and will work well for small problem instances, But the linear requirement on time and space in the total number of bits in S, is prohibitively expensive for many real-world applications.

Bloom filters TODO: If we allow false positives, we can change these characteristics to support either constant-time queries, or space-efficiency, but not both. (her skal der stå noget om: 1 query / $n \cdot d^k$ inserts ELLER 1 insert, $n \cdot d^k$ queries)

If we want to be able to tell if there is an element in the bloom filter which is at most d different from the element we query with, we have two options: One sacrificing space and the other sacrificing time. While keeping in mind that standard Bloom filters only answers exact membership, we see that one way of achieving this is by relying on extra insertions. This means that everytime an element is about to be added to the Bloom filter, all elements within distance d is added in addition to the original element. This approach requires a lot more space depending on the [Universe], if we do not want the accuracy of the Bloom filter to be reduced too much. It does however still guarantee a fast response time

Another way to go around this is by sacrificing time. This can be done by still doing normal insertions but instead querying for all elements with distance d of the element we are comparing to. This approach has the benefit of still being space efficient but will cost on the running time.

2 Related work

2.1 Distance-Sensitive Bloom Filters

Kirsch and Mitzenmacher propose in the paper 'Distance-Sensitive Bloom Filters'[1] a novel way of using Bloom filters. The aim is to expand upon classical Bloom filters, enabling them to answer the question: "Does a set contain an element similar to a given element?". They note that such a data structure has a number of practical uses, if it can be made sufficiently effective in both time and space.

The way they accomplish this is by using locality sensitive hashing algorithms (TODO REF), such that two elements that are sufficiently similar will, with a high probability, hash to the same value. They also use a partitioned Bloom filter, which is a like a standard Bloom filter except each locality sensitive hashing algorithm maps into its own bit array, instead of all the hashing algorithms share the same bit array.

When querying the Bloom filter for an approximate match, the element is hashed with all the hashing algorithms, and the corresponding indiceses are checked. If there are more bits, than a specified threshold (T), that is set to 1, it returns that the data contains an element that is close to the query element. The use of a threshold is the reason why it is possible to have false positives using this data structure.

A disadvantage to this approach, compared to a classical Bloom filter, is that it introduces the possibility of false negatives.

2.2 Kinesere

The paper written by [TODO] takes a different approach than [TODO ref mitz]. They use a standard Bloom Filter data structure, but with LSH in place of ordinary hash functions - they call this a 'Locality Sensitive Bloom Filter'. This 'naive' approach, has a high probability of both False Positives and False Negatives, so they augment this with additional data structures.

Minimizing False Positives Every time an element q is checked for approximate membership, the LSBF is checked. If every bit in the array that q hashes to is set to 1, that could mean one of two things. 1) An element p exists in the LSBF, that is approximately close to q (a true positive) 2) Multiple elements added to the LSBF, together, have conspired to set the all the bit positions corresponding to q to 1, and no single element is approximately close to q. (a false positive).

To minimize the probability of a FP, a Verification Scheme (VS) data structure is added. This is a standard Bloom filter (ordinary hashing, not LSH), and is maintained as follows: Every time an element is added to the LSBF, the bit positions that corresponds to that element, are encoded (see figure 1) and inserted into the VS Bloom filter. Now when the LSBF is queried, it first verifies that all bits corresponding to the query element q are set to 1 in the LSBF, and then encodes the bit positions (figure 1), and queries the VS for an exact match. If an exact match for the encoded version of q is found in VS, then it is highly likely that a single element p is responsible for setting all the bits in the LSBF, and the query will return a 'yes'. If, on the other hand, the VS does not contain an element that matches the encoded q, then multiple elements must be responsible for the bits being set, and the query will return 'no', effectively catching a false positive.

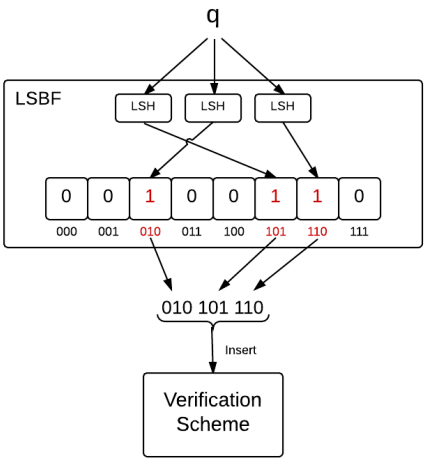

Figure 1: Maintaining the Verification Scheme, when adding elements to an LSBF

Note that the VS is a standard Bloom filter, and therefor allows false positives itself. That means it only serves to minimized false positives, not outright remove them.

Minimizing False Negatives They use a method called Active Overflowed Scheme, which utilizes the property of the locality sensitive hashing algorithms that proximate elements will be stored close to each other. The idea is that a pair of close elements may not be hashed to the same value, but instead be put in adjacent bins in the bit array. So when querying, instead of just looking at the bits which represents the hashed element, you also check the t closest bits to either side. t will depend on the price of false positives compared to false negatives as this approach will introduce more false positives.

References

[1] A. Kirsch and M. Mitzenmacher, "Distance-Sensitive Bloom Filters", Proc. Eighth Workshop Algorithm Eng. and Experiments (ALENEX), 2006.

[2] Bonomi, Flavio and Mitzenmacher, Michael and Panigrahy, Rina and Singh, Sushil and Varghese, George, "An Improved Construction for Counting Bloom Filters", Algorithms – ESA 2006

[3] Yu Hua, Bin Xiao, Bharadwaj Veeravalli, Dan Feng, "Locality-Sensitive Bloom Filter for Approximate Membership Query", IEEE Transactions on Computers, Vol. 61, No. 6, 2012