Similarity Filters
A Survey

Marcus Gregersen
mabg@itu.dk

Martin Faartoft
mlfa@itu.dk

Rick Marker
rdam@itu.dk

May 21st 2014

Abstract

The abstract goes here

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 XX and YY and suggest improvements.

Indsæt XX og YY

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 given 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

Ordinary hashing tries to minimize the probability of a collision, while Locality-Sensitive Hashing (LSH) attempts to maximize the collision probability for sufficiently similar elements.

Not all similarity-measures admits LSH, but the Hamming distance metric on bit-vectors admits a particular simple one. Simply sample a number of bit positions from the input at random.

K-closeness

- If a vector y in S differs from x in at most k bits, it is k-close to x. The data structure must return "close".

- If every vector y in S differs from x in at least $2k$ bits, there is no vector that is k-close to x. We would like the data structure to return “not close”.
- In all other cases, i.e. when the distance to the nearest vector is between $k+1$ and $2k-1$, 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 / $n * d^k$ inserts ELLER 1 insert, $n * d^k$ queries)

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.

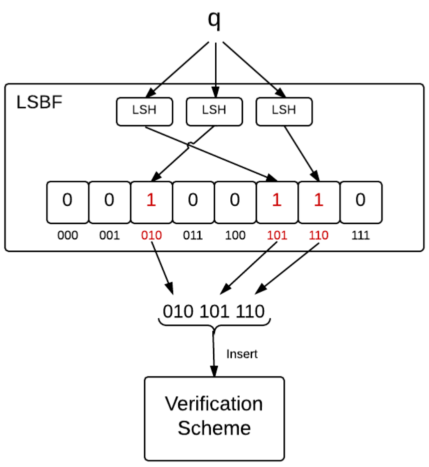

Figure 1: Maintaining the Verification Scheme, when adding elements to an LSBF

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.

3 Theory

4 Experiments

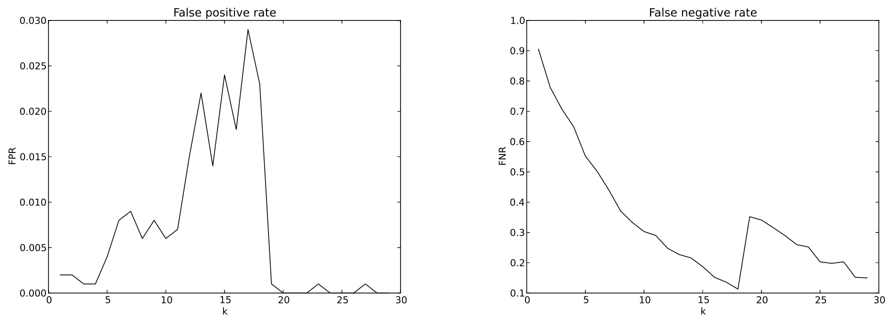

Figure 2: The observed false positive and false negative rates for $n = 1000$, $l = 65536$, $\epsilon = 0.1$, and $\sigma = 0.4$

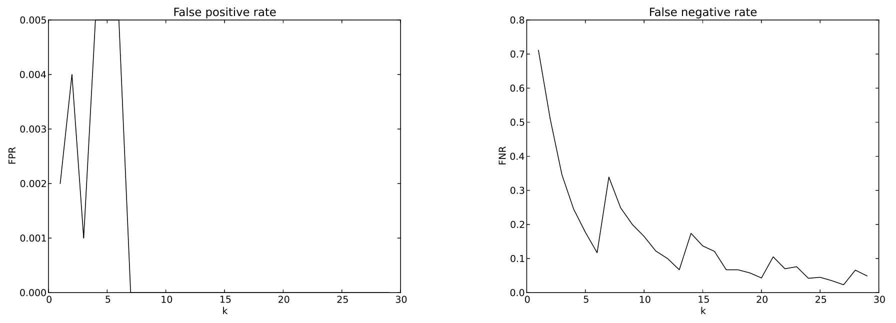

Figure 3: The observed false positive and false negative rates for $n = 10000$, $l = 65536$, $\epsilon = 0.05$, and $\sigma = 0.4$

5 Suggested Improvements

5.1 Balanced LSH Bit Sampling

The default way to initialize the LSH functions used in the Distance-Sensitive Bloom filter (DSBF), is for each of the k LSH functions, to choose l' bit positions to sample, uniformly at random. On average this should spread the sampled bits out nicely across the k hash-functions, but in some cases this will result in a skewed distribution, meaning that some bits are oversampled, while others may not be sampled at all. This skew will degrade the performance of the DSBF, resulting in more false answers.
This can be remedied, by initializing the LSHs such that all bit positions

are sampled equally often. In practice we do this by first preparing a vector of length $k \cdot l^{'}$ by repeating the the integers from 0 to l as many times as required. We then shuffle the vector, and assign each of the k functions a sub-vector of length $l^{'}$, making sure that every index in the sub-vector is unique. This preprocessing step takes $O(l)$ time, and protects the DSBF from poor performance due to skewed sampling.

Figure TODO shows an experiment run with Balanced bit sampling enabled (TODO some color) and disabled (TODO some other color), and shows that this optimization in some cases improves the performance of the DSBF, but never degrades it.

produce and insert graphs here

5.2 Eliminating False Negatives

Building on the previous idea from 5.1, we suggest a way to eliminate false negatives from the DSBF data-structure.

False Negatives happen when a query element is within distance ϵ of an element in S, but is reported as being 'not close'. This happens when too many of the k LSH functions report false, driving the number of trues below the threshold t. We know that each of the LSHs that report false, will sample at least one bit in which q and every element $s \in S$ differ. Since q, by definition, is 'close' to some element $s \in S$, the maximum number of differences between q and the closest element in S is: $\epsilon_{abs} = \lceil \epsilon \cdot l \rceil$, where l is the element length. Assuming that each is sampled an equal number of times (ensured by 5.1)): $l_n = \lceil \frac{k \cdot l'}{l} \rceil$, the maximum number of hash-functions that can report false is: $n_{false} = e_{abs} \cdot l_n$.

Providing a guarantee against false negatives can be achieved by lowering the DSBF threshold appropriately. Armed with the knowledge of n_{false} from 5.2, we can set $t = k - n_{false}$, and thereby eliminate the possibility of false negatives completely.

Lowering t comes at a price, the lower the value of t, the higher the probability of a false positive. In the edge case, where $t = 0$, the DSBF becomes useless, answering 'close' to every query. Taking this into account, we have to constrain the value of n_{false} such that $t > 0$.

$$t > 0 \leftrightarrow k > n_{false} \leftrightarrow k > \epsilon_{abs} \cdot l_n$$

This turns out to be a big deal, k is usually below 100, l_n can be kept small, but not smaller than 1. Those two bound ϵ_{abs}, to definitely by below 100, which in turn makes it impossible for ϵ to be a constant fraction of l, for increasing l values.

6 Conclusion

References

[1] Adam Kirsch, Michael Mitzenmacher Distance-Sensitive Bloom Filters

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

Appendix