Probabilistic Data Structures: Bloom Filters

Stefan Gerber
California State Channel Islands

Abstract

Typically within computational sciences, an algorithm is used as a set of operations upon an input space that maps directly to a deterministic output space. However, not always does the algorithm's operation necessarily need to be deterministic. Algorithms of the probabilistic classification have attributes that make their running time or output correctness non deterministic. Monte Carlo algorithms, one such subtype, can be used to provide an output that is probabilistically determined. Bloom Filters are one such example of a data structure which seeks to provide efficient answers to problems that can be difficult or cost ineffective to solve at scale with some room for a known error. These data structures allow for an element's membership within a set to be classified as either a true negative or potential false positive and is present within various technologies.

1 Introduction

The Bloom Filter, named not for some relation to flowers or plants, but for its creator Burton Howard Bloom. In his 1970 paper, *Space/time trade-offs in hash coding with allowable errors*, Bloom discusses a new data structure effective in determining non membership within a set of elements. The bloom filter is able to accurately determine that an element does not belong in a set. This comes with the consequence of being unable to precisely identify an element as being a member of the set, resulting in potential false positives. However, the likelihood of element membership is able to be tuned through changing various parameters of the filter's underlying constructions.

2 Bloom Filter Formal Construction

Assume there are k different hash functions that handle the desired input $x : x \in X$ where X is the subset of the input space W of only expected values. Then $K : X \to \mathbb{N}, |X| = n, K_p(x) = y$ where y is some integer output and $0 \le p \le k-1 : p \in \mathbb{N}$.

It is important to note that for some input x and any two random hash functions K_i and K_j, where $0 \le i, j \le k-1$ and $i \ne j$, $K_i(x)$ may be equal to $K_j(x)$. Additionally for any two inputs x_i and x_j, $K_i(x_i)$ may be equal to $K_i(x_j)$ as well. This is to say that these hash functions may collide across both functions and inputs.

The underlying data structure of the bloom filter is an array A of length m initialized to all 0's. Each hash function K is intended to provide a mapping from X to the subset of the Whole Numbers that represent the indices of A. Therefore the output of each function should be taken as modulo m to remain within the proper bounds of A.

There exists two simple identities relating k the number of hash function, m the size of the array, p false positive rate, and n the number of possible inputs.

$$k = \frac{m}{n} \cdot \ln(2)$$

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

These tuning formulas can be manipulated to optimize for any of the desired qualities, usually number of hash functions and size of array.

2.1 Bloom Filter Insertion

When an element is input into some function K_i it returns some integer y resulting in the y mod m index of A to be populated to a 1 denoting potential membership. This is done for each hash function K. The following pseudo code represents this general insertion algorithm.

```
insert-bloom-filter(x :: Input,
   A :: bloom-filter,
   K :: array-of-hash-functions)
   -> bloom-filter {

m = |A|
For each K, Ki {
```

```
y = Ki(x)
A[y mod m] = 1
}
Return A
```

Clearly this input method operates in $O(k)$ time complexity where k is the number of hash functions used, although this is generally a fixed constant resulting in an effective $O(1)$ constant running time. This straight forward method of inserting may result in collisions which makes it impossible to remove an element from the bloom filter. Additionally, the usage of this type of bit array allows for faster non membership checking operation than a hash table regardless of the limits of complexity both being in constant time. This is due to the underlying data structure to essentially be represented as a set of binary integers with efficient bit flipping as the bitwise primary operation, which is quick on the ALU.

2.2 Bloom Filter Membership Evaluation

The evaluation of membership within a bloom filter is yet another straightforward process. By passing the input through every hash function while noting if any of the corresponding array indices have been altered. If any of these hashed indices is a 0, then x the input is definitely not within the set since that index has not been mapped to yet. Therefore to evaluate bloom filter membership simply check each hash function output's corresponding index within the bloom filter until a 0 is found. If every address has been set to a 1, then the element might be within the set. Definite membership cannot be determined due to collisions in the hash functions resulting in some input's values already being entirely mapped to. The pseudocode below represents this probabilistic checking for membership.

```
is-not-in-bloom-filter(x :: Input,
        A :: bloom-filter,
        K :: array-of-hash-functions)
        -> Boolean {

m = |A|
For each K, Ki {
y = Ki(x)
If (A[y mod m] == 1) {
continue
}
If (A[y mod m] == 0){
Return True
}
}
Return False
}
```

For this function, an output of True denotes that the element is definitely not within the bloom filter, while an output of False denotes that the element might be within the set or it might simply be a false positive. Clearly the running time of this function is the same as before $O(k)$ or $O(1)$ since the number of hash functions remains constant.

2.3 Bloom Filter Applications

The original intention of this data structure was to keep a dictionary and determine quickly if a word does not exist within the set. Any misspelled words are returned as not being within the dictionary, saving resources for a potential lookup. The dictionary might not be populated with words, but instead malicious URLS, easily being able to identify a safe URL as non malicious. This dictionary can also be populated with database keys saving time not fetching keys that might not exist.

3 Conclusion

Bloom Filters are a very powerful example of a probabilistic data structure. The ability to quickly determine an element's non membership within a set has many applications within modern software. Unfortunately the inability to definitely determine an element's membership does reduce the possible use cases for this data structure. Although a bloom filter combined with an inverse bloom filter could potentially be used to provide a more detailed description of the input.

References

1. Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), 422–426. https://doi.org/10.1145/362686.362692

2. Dong, F., Wang, P., Li, R., Cui, X., Zhao, J., Tao, J., Zhang, C., & Guan, X. (2025). Poisoning attacks and defenses to learned bloom filters for malicious URL detection. IEEE Transactions on Dependable and Secure Computing, 22(4), 3275–3288. https://doi.org/10.1109/tdsc.2025.3528993