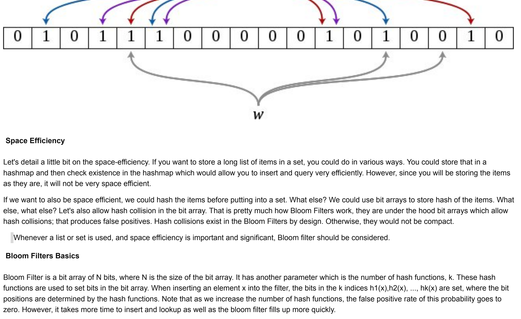
Bugra Akyildiz, Hinge App.

Bloom Filter

Bloom filters are probabilistic space-efficient data structures. They are very similar to hashtables; they are used exclusively membership existence in a set. However, they have a very powerful property which allows to make trade-off between space and false-positive rate when it comes to membership existence. Since it can make a tradeoff between space and false positive rate, it is called probabilistic data structure.



Space Efficiency

Let's detail a little bit on the space-efficiency. If you want to store a long list of items in a set, you could do in various ways. You could store that in a hashmap and then check existence in the hashmap which would allow you to insert and query very efficiently. However, since you will be storing the items as they are, it will not be very space efficient.

If we want to also be space efficient, we could hash the items before putting into a set. What else? We could use bit arrays to store hash of the items. What else, what else? Let's also allow hash collision in the bit array. That is pretty much how Bloom Filters work, they are under the hood bit arrays which allow hash collisions; that produces false positives. Hash collisions exist in the Bloom Filters by design. Otherwise, they would not be compact.

> Whenever a list or set is used, and space effciency is important and significant, Bloom filter should be considered.

Bloom Filters Basics

Bloom Filter is a bit array of N bits, where N is the size of the bit array. It has another parameter which is the number of hash functions, k. These hash functions are used to set bits in the bit array. When inserting an element x into the filter, the bits in the k indices h1(x),h2(x), ... hk(x) are set, where the bit positions are determined by the hash functions. Note that as we increase the number of hash functions, the false positive rate of this probability goes to zero. However, it takes more time to insert and lookup as well as the bloom filter fills up more quickly.

In order to to membership existence in the Bloom Filter, we need to chekck if all of the bits are set; very similar to how we insert item into a bloom filter. If all of the bits are set, then it means that that item is probably in the bloom filter, where if anot all of the bits are set, then it means that the item is not in the Bloom Filter.

Basic Python Implementation

If we want to implement a basic bloom filter, we could easily do so.

```
# 3rd party
import mmh3

class BloomFilter(set):

def __init__(self, size, hash_count):
super(BloomFilter, self).__init__()
self.bit_array = bitarray(size)
self.bit_array.setall(0)
self.size = size
self.hash_count = hash_count

def __len__(self):
return self.size

def __iter__(self):
return iter(self.bit_array)

def add(self, item):
for ii in range(self.hash_count):
index = mmh3.hash(item, ii) % self.size
self.bit_array[index] = 1

return self

def __contains__(self, item):
out = True
for ii in range(self.hash_count):
index = mmh3.hash(item, ii) % self.size
if self.bit_array[index] == 0:
out = False

return out

def main():
bloom = BloomFilter(100, 10)
animals = ['dog', 'cat', 'giraffe', 'fly', 'mosquito', 'horse', 'eagle',
'bird', 'bison', 'boar', 'butterfly', 'ant', 'anaconda', 'bear',
'chicken', 'dolphin', 'donkey', 'crow', 'crocodile']
# First insertion of animals into the bloom filter
for animal in animals:
bloom.add(animal)

# Membership existence for already inserted animals
# There should not be any false negatives
for animal in animals:
if animal in bloom:
print('{} is in bloom filter as expected'.format(animal))
else:
print('Something is terribly went wrong for {}'.format(animal))
print('FALSE NEGATIVE!')

# Membership existence for not inserted animals
# There could be false positives
other_animals = ['badger', 'cow', 'pig', 'sheep', 'bee', 'wolf', 'fox',
'whale', 'shark', 'fish', 'turkey', 'duck', 'dove',
'deer', 'elephant', 'frog', 'falcon', 'goat', 'gorilla',
'hawk' ]
for other_animal in other_animals:
if other_animal in bloom:
print('{} is not in the bloom, but a false positive'.format(other_animal))
else:
print('{} is not in the bloom filter as expected'.format(other_animal))

if __name__ == '__main__':
main()
```

Output is in the following:

```
dog is in bloom filter as expected
cat is in bloom filter as expected
giraffe is in bloom filter as expected
fly is in bloom filter as expected
mosquito is in bloom filter as expected
horse is in bloom filter as expected
eagle is in bloom filter as expected
bird is in bloom filter as expected
bison is in bloom filter as expected
boar is in bloom filter as expected
butterfly is in bloom filter as expected
ant is in bloom filter as expected
anaconda is in bloom filter as expected
bear is in bloom filter as expected
chicken is in bloom filter as expected
dolphin is in bloom filter as expected
donkey is in bloom filter as expected
crow is in bloom filter as expected
crocodile is in bloom filter as expected

badger is not in the bloom filter as expected
cow is not in the bloom filter as expected
pig is not in the bloom filter as expected
sheep is not in the bloom, but a false positive
bee is not in the bloom filter as expected
wolf is not in the bloom filter as expected
fox is not in the bloom filter as expected
whale is not in the bloom filter as expected
shark is not in the bloom, but a false positive
fish is not in the bloom, but a false positive
turkey is not in the bloom filter as expected
duck is not in the bloom filter as expected
dove is not in the bloom filter as expected
deer is not in the bloom filter as expected
elephant is not in the bloom, but a false positive
frog is not in the bloom filter as expected
falcon is not in the bloom filter as expected
goat is not in the bloom filter as expected
gorilla is not in the bloom filter as expected
hawk is not in the bloom filter as expected
```

As you could see the output from above, there are false positives, but there were not any false negatives as expected.

Unlike this implementation of the Bloom Filter, most of the implementations that are available in various languages do not provide a hash function argument, though. This is because false-positive rate is more important than the hash function in terms of application and depending on the false positive rate, you could always adjust the number of hash functions that are going to be used. Generally, the size and *error_rate* which is actually the false positive rate of the Bloom Filter. If you decrease the *error_rate* when you initialize the bloom filter, they would adjust the number of hash functions under the hood.

False Positives

While Bloom Filters can say "definitely not in" with confidence, they will also say possibly infor some number of items. Depending on the application, this could be a huge downside or it could be relatively okay. If it is okay to introduce false positives every now and then, you should definitely consider using Bloom Filters for membership existence for set operations.

Also note that if you are decreasing the false positive rate arbitrarily, you would increase the number of hash functions which would add latency to both insertion and membership existence. One more thing in this section is that, if the hash functions are independent each other and distribute the input space pretty uniformly, then the theoretic false positive rate can be satisfied. Otherwise, the false positive rate of the bloom filter will be worse than the theoretic false positive rate as hash functions correlate each other and hash collisions would occur more often than desired.

> When using a Bloom filter, consider the potential effects of false positives.

Deterministic

If you are using the same size and same number hash functions as well as the hash function, bloom filter is deterministic on which items it gives positive response and which items it gives negative response. For an item x, if it gives it is probably in to that particular item, it will give the same response as 5 minutes later, 1 hour later, 1 day later and 1 week later. I was a little surprised when I found this. It was "probabilistic" so the response of the bloom filter should be somehow random, right? Not really. It is probabilistic in the sense that you cannot know which item it will say it is probably in.

> Otherwise, when it says that it is probably in, it keeps saying the same thing.

Disadvantages

Not everything is so great about Bloom Filters.

The size of the Bloom Filter

The size of the Bloom Filters need to be known a priori based on the number of items that you are going to insert. This is not so great if you do not know or cannot approximate the number of items. You could put an arbitrarily large size, but that would be a waste in terms of space which we are trying to optimize in the very first place and the reason why we adopt to choose Bloom Filter. This could be fixed to create a bloom filter dynamic to the list of items that you want to fit, but depending on the application, this may not be always possible. There is a variant called Scalable Bloom Filter which dynamically adjusts its size for different number of items. This could mitigate some of its shortcomings.

Constructing and Membership Existence in Bloom Filter

While using the Bloom Filters, you not only accept false positive rates, but also you are willing to have a little bit overhead in terms of speed. Comparing to an hashmap, there is definitely an overhead in terms of hashing the items as well as constructing the bloom filter.

Cannot give the items that you inserted

Bloom Filter cannot produce a list of items that are inserted, you could only check if an item is in it, but never get the full item list because of hash collisions and hash functions. This is due to arguably the most significant advantage over other data structures; its space efficiency which comes with this disadvantage.

Removing an element

Removing an element from the Bloom Filter is not possible, you cannot undo an insertion operation as hash results for different items can be indexed in the same position. If you want to do undo inserts, either you need to count the inserts for each index in the BloomFilter or you need to construct the BloomFilter from the start excluding a single item. Both methods involve an overhead and not straightforward. Depending on the application, one might want to try to reconstruct the bloom filter from the start instead of removing or deleting items from the Bloom Filter.

Implementations in Different Languages

In production, you do not want to roll out your own bloom filter implementation. There are two reasons; one of them choosing and implementing good hash functions is crucially important to distribute the error rate for any number of inputs. Second of them, it needs to be battle-tested and should not be error prone both in terms of error rate and its size. There are open source implementations for every language, but the following for node.js and Python are pretty good in my experience:

- Node
- Python

There is also very fast implementation(10x faster than the above Python library both in terms of membership existence and adding the item into the bloom filter), pybloomfilter, but this runs on Pypy and does not support Python 3.

Bio: Bugra Akyildiz is a Senior Machine Learning Engineer at Hinge App. You can find him on Twitter **@bugraa**.

Original. Reposted with permission.

Related:

- Would You Survive the Titanic? A Guide to Machine Learning in Python Part 1
- Getting Started with Data Science – Python
- America's Next Topic Model