

**Laboratory Manual**

**Subject: Machine Learning–IV (PCCS7010T) Semester: VII Class: B. Tech Experiment No. : 3**

**Title :** Implement Bloom Filter using MapReduce.

**Pre-requisite:** Basics of Machine Learning

Hardware Requirements (if any): Windows 64 bit processor

Theory/Concept Explanation (Compulsory):

Introduction:

A Bloom filter is a **space-efficient probabilistic** data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. **False positive means**, it might tell that given username is already taken but actually it’s not.

**Interesting Properties of Bloom Filters**:

* Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
* Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
* Bloom filters never generate **false negative** result, i.e., telling you that a username doesn’t exist when it actually exists.
* Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete “geeks” (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting “nerd” also Because bit at index 4 becomes 0 and bloom filter claims that “nerd” is not present.

**Working of Bloom Filter**

A empty bloom filter is a **bit array** of **m** bits, all set to zero, like this –

Lightbox

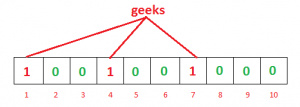
We need **k** number of **hash functions** to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), … hk(x) are set, where indices are calculated using hash functions.   
Example – Suppose we want to enter “geeks” in the filter, we are using 3 hash functions and a bit array of length 10, all set to 0 initially. First we’ll calculate the hashes as follows:

h1(“geeks”) % 10 = 1

h2(“geeks”) % 10 = 4

h3(“geeks”) % 10 = 7

**Note:** These outputs are random for explanation only.   
Now we will set the bits at indices 1, 4 and 7 to 1



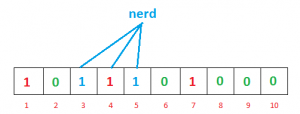
Again we want to enter “nerd”, similarly, we’ll calculate hashes

h1(“nerd”) % 10 = 3

h2(“nerd”) % 10 = 5

h3(“nerd”) % 10 = 4

Set the bits at indices 3, 5 and 4 to 1 



Now if we want to check “geeks” is present in filter or not. We’ll do the same process but this time in reverse order. We calculate respective hashes using h1, h2 and h3 and check if all these indices are set to 1 in the bit array. If all the bits are set then we can say that “geeks” is **probably present**. If any of the bit at these indices are 0 then “geeks” is **definitely not present**.

**False Positive in Bloom Filters**

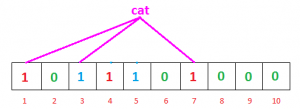
The question is why we said **“probably present”**, why this uncertainty. Let’s understand this with an example. Suppose we want to check whether “cat” is present or not. We’ll calculate hashes using h1, h2 and h3

h1(“cat”) % 10 = 1

h2(“cat”) % 10 = 3

h3(“cat”) % 10 = 7

If we check the bit array, bits at these indices are set to 1 but we know that “cat” was never added to the filter. Bit at index 1 and 7 was set when we added “geeks” and bit 3 was set we added “nerd”.



So, because bits at calculated indices are already set by some other item, bloom filter erroneously claims that “cat” is present and generating a false positive result. Depending on the application, it could be huge downside or relatively okay.  
We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want to decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition to the item and checking membership.   
**Operations that a Bloom Filter supports**

* insert(x) : To insert an element in the Bloom Filter.
* lookup(x) : to check whether an element is already present in Bloom Filter with a positive false probability.

NOTE : We cannot delete an element in Bloom Filter.  
**Probability of False positivity:** Let **m**be the size of bit array, k be the number of hash functions and **n** be the number of expected elements to be inserted in the filter, then the probability of false positive **p** can be calculated as:



**Size of Bit Array:** If expected number of elements **n** is known and desired false positive probability is **p** then the size of bit array **m** can be calculated as:



**Optimum number of hash functions:** The number of hash functions **k** must be a positive integer. If **m** is size of bit array and **n** is number of elements to be inserted, then k can be calculated as:



**Space Efficiency**

If we want to store large list of items in a set for purpose of set membership, we can store it in[hashmap](https://www.geeksforgeeks.org/hashmap-treemap-java/), [tries](https://www.geeksforgeeks.org/trie-insert-and-search/) or simple [array](https://www.geeksforgeeks.org/array/) or [linked list](https://www.geeksforgeeks.org/data-structures/linked-list/). All these methods require storing item itself, which is not very memory efficient. For example, if we want to store “geeks” in hashmap we have to store actual string “ geeks” as a key value pair {some\_key : ”geeks”}.   
Bloom filters do not store the data item at all. As we have seen they use bit array which allow hash collision. Without hash collision, it would not be compact.

**Choice of Hash Function**

The hash function used in bloom filters should be independent and uniformly distributed. They should be fast as possible. Fast simple non cryptographic hashes which are independent enough include [murmur](https://en.wikipedia.org/wiki/MurmurHash), FNV series of hash functions and [Jenkins](https://en.wikipedia.org/wiki/Jenkins_hash_function) hashes.   
Generating hash is major operation in bloom filters. Cryptographic hash functions provide stability and guarantee but are expensive in calculation. With increase in number of hash functions k, bloom filter become slow. All though non-cryptographic hash functions do not provide guarantee but provide major performance improvement.

**Implementation:**

**# Save it as bloomfilter.py**

# Python 3 program to build Bloom Filter

# Install mmh3 and bitarray 3rd party module first

# pip install mmh3

# pip install bitarray

import math

import mmh3

from bitarray import bitarray

class BloomFilter(object):

'''

Class for Bloom filter, using murmur3 hash function

'''

def \_\_init\_\_(self, items\_count, fp\_prob):

'''

items\_count : int

Number of items expected to be stored in bloom filter

fp\_prob : float

False Positive probability in decimal

'''

# False possible probability in decimal

self.fp\_prob = fp\_prob

# Size of bit array to use

self.size = self.get\_size(items\_count, fp\_prob)

# number of hash functions to use

self.hash\_count = self.get\_hash\_count(self.size, items\_count)

# Bit array of given size

self.bit\_array = bitarray(self.size)

# initialize all bits as 0

self.bit\_array.setall(0)

def add(self, item):

'''

Add an item in the filter

'''

digests = []

for i in range(self.hash\_count):

# create digest for given item.

# i work as seed to mmh3.hash() function

# With different seed, digest created is different

digest = mmh3.hash(item, i) % self.size

digests.append(digest)

# set the bit True in bit\_array

self.bit\_array[digest] = True

def check(self, item):

'''

Check for existence of an item in filter

'''

for i in range(self.hash\_count):

digest = mmh3.hash(item, i) % self.size

if self.bit\_array[digest] == False:

# if any of bit is False then,its not present

# in filter

# else there is probability that it exist

return False

return True

@classmethod

def get\_size(self, n, p):

'''

Return the size of bit array(m) to used using

following formula

m = -(n \* lg(p)) / (lg(2)^2)

n : int

number of items expected to be stored in filter

p : float

False Positive probability in decimal

'''

m = -(n \* math.log(p))/(math.log(2)\*\*2)

return int(m)

@classmethod

def get\_hash\_count(self, m, n):

'''

Return the hash function(k) to be used using

following formula

k = (m/n) \* lg(2)

m : int

size of bit array

n : int

number of items expected to be stored in filter

'''

k = (m/n) \* math.log(2)

return int(k)

**# Save it as bloomtest.py**

from bloomfilter import BloomFilter

from random import shuffle

n = 20 #no of items to add

p = 0.05 #false positive probability

bloomf = BloomFilter(n,p)

print("Size of bit array:{}".format(bloomf.size))

print("False positive Probability:{}".format(bloomf.fp\_prob))

print("Number of hash functions:{}".format(bloomf.hash\_count))

# words to be added

word\_present = ['abound','abounds','abundance','abundant','accessible',

'bloom','blossom','bolster','bonny','bonus','bonuses',

'coherent','cohesive','colorful','comely','comfort',

'gems','generosity','generous','generously','genial']

# word not added

word\_absent = ['bluff','cheater','hate','war','humanity',

'racism','hurt','nuke','gloomy','facebook',

'geeksforgeeks','twitter']

for item in word\_present:

bloomf.add(item)

shuffle(word\_present)

shuffle(word\_absent)

test\_words = word\_present[:10] + word\_absent

shuffle(test\_words)

for word in test\_words:

if bloomf.check(word):

if word in word\_absent:

print("'{}' is a false positive!".format(word))

else:

print("'{}' is probably present!".format(word))

else:

print("'{}' is definitely not present!".format(word))

**#Output:**

Size of bit array:124

False positive Probability:0.05

Number of hash functions:4

'war' is definitely not present!

'gloomy' is definitely not present!

'humanity' is definitely not present!

'abundant' is probably present!

'bloom' is probably present!

'coherent' is probably present!

'cohesive' is probably present!

'bluff' is definitely not present!

'bolster' is probably present!

'hate' is definitely not present!

'racism' is definitely not present!

'bonus' is probably present!

'abounds' is probably present!

'genial' is probably present!

'geeksforgeeks' is definitely not present!

'nuke' is definitely not present!

'hurt' is definitely not present!

'twitter' is a false positive!

'cheater' is definitely not present!

'generosity' is probably present!

'facebook' is definitely not present!

'abundance' is probably present!