Bloom Filter

Hitarth Panchal (1641034)
School of Engineering and Applied Sciences, Ahmedabad University
Email: hitarth.p.btechi16@ahduni.edu.in

***Abstract*—This report presents the implementation of bloom filter given the input set size and false positive error probability in the input. It has a great use many applications like cache filtering, data synchronization, chemical structure searching etc. This paper also explains how to implement bloom filter in python. Also there many graphs are presented, which shows trade offs between different parameters of bloom filter.**

I. Introduction

A Bloom filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set. The price paid for this efficiency is that a Bloom filter is a probabilistic data structure: it tells us that the element either definitely is not in the set or may be in the set. Conceived by Burton Howard Bloom in 1970. False positive matches are possible, but false negatives are not in other words, a query returns either ”possibly in set” or ”definitely not in set”. Elements can be added to the set, but not removed.

II. Implementation Design

This algorithm takes size of input set (parameter n) and false positive error probability (parameter p). Given these 2 it calculates parameter k and m from formulas given below. Basically parameter m is size of bit vector and parameter k is number of hash functions to use.

$$m = \frac{n \log_e p}{(\log_e 2)^2} \tag{1}$$

$$k = \frac{m}{n} \log_e 2 \tag{2}$$

These k hash functions has to be independent and also one hash function should map the input elements to the values which all have the same probability of occurrence.

Let’s say to prove that one hash function should generated the values of uniform distributions given the different inputs, a hash function is given 10^6 random strings and a vector is generated of size 10^6. These values are between 0 to 10000, so histograms contains unique values (between 0 to 10000) on X-axis and frequencies on y-axis. Histogram is shown in figure 1. In figure, we can see that all the values has more or less same frequency which means its following uniform distribution.

Now to prove that, these k hash functions are independent a correlation matrix is shown in figure 2. Total 50 hash functions are created and are fed same random strings. Total 10^6 random strings are fed to these 50 hash functions. And a correlation between these 50 vectors of size 10^6 is computed. In figure 2, we can infer that correlation between these vectors is zero.

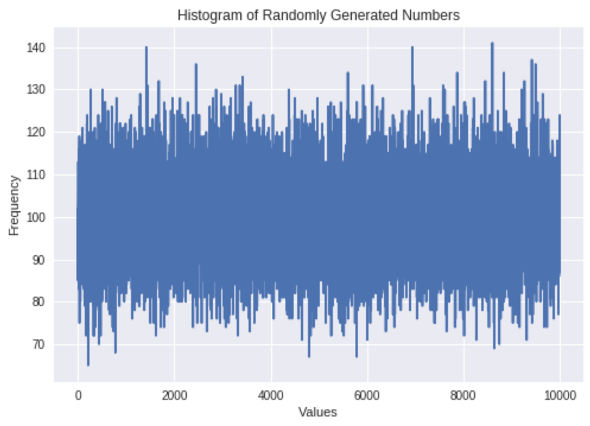

Fig. 1. Histogram of generated values by a hash function.

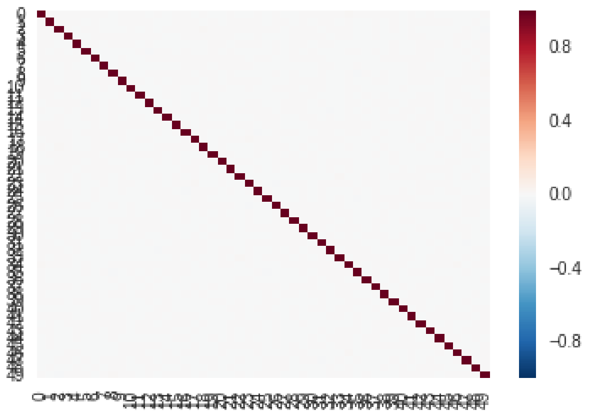

Fig. 2. Correlation between the hash functions

Which means the 50 hash functions are uncorrelated hence independent.

First of all values of *n* and *p* is inputted, from that value of *m* and *k* is computed using equation (1) and (2). A bit vector of size *m* is initialized with all zeros. After that first randomly generated string is given to *k* independent hash functions each of which generates a value between 0 to *m-1*. At those k positions, values in bit vector is set. This hash function are generated with the help of *mmh3* library in python. Given 2 same strings passed to these *k* hash functions, same values (total k values because we have k hash functions) will be generated. Then second string is processed same way as first and values at particular indexes are set in bit vector according

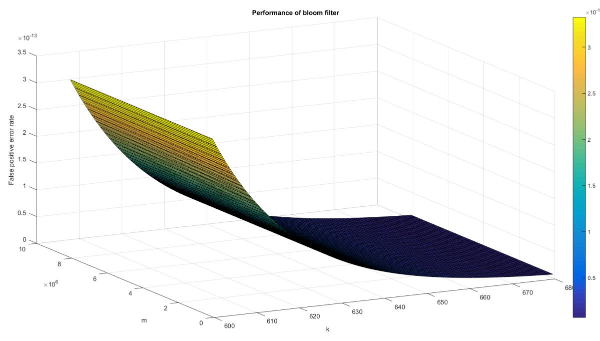

Fig. 3. n = 100

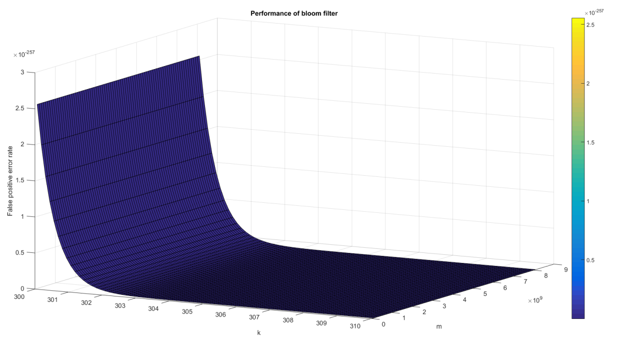

Fig. 4. n = 10000

to hash outputs. If a same index is set already then it will be kept as set. This way all n strings are processed.

Now to check if a string either belongs to a set or not, k values will be generated by the hash functions by inputting the test string. And now if one or more values is/are unset at those k indexes in bit vector then it can be inferred that, inputted string is not in the set with probability of 1. For the other way around if all the values are set in the bit vector then it may be present in the set or may not. It may happen that all the values at those k locations are set but actually the element is not present in the set, which is called the a false positive. And probability of false positive is computed using the formula given below.

$$p = (1 - e^{-\frac{kn}{m}})^k \tag{3}$$

Goal is to make p close to 0. To achieve this goal parameters k and m has to be optimized. In this project, false positive error probability and input size are taken as inputs and m and k are computed using equation (1) and (2) respectively. After that a simulation is done using Monte-Carlo technique. A simulated false positive error rate is computed and analytical false positive error rate is also computed (equation 3). And both the values are coming out to be nearly same. Which infers that the simulation is done correctly. In simulation, total 10^6 unique strings are generated (Using *uuid* library) which is completely different from previously processed strings (Using python's *random* library). Now based on these simulated probability is calculated.

III. Performance

Here false positive error rate is taken as the a performance metric. Since the size of the input is given in the input (n), other two parameters (m and k) are kept variable and then the 3D graph is plotted using MATLAB for better visualization purposes. Shown in figure 3 and figure 4. Which shows the value of probability at different values of parameters k and m. This trade-off can be understood using equation (3).

IV. Conclusion

Given some of the parameters best possible performance can be gained using analytical formulas (eq. 1,2,3). Which narrows down the optimization process. In bloom filter there has to have a method to remove the elements from the set. Also for the steaming purposes parameter n (input size) is not fixed, hence the filter has to adapt according to that to get minimum false positive error rate.

V. CODE

Imports of required libraries.

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from statistics import median
import matplotlib as mpl
import math
import random
import string
import mmh3
import uuid
import time
```

Code for creating the histogram of generated numbers by a hash function.

```
inp = np.array([abs(mmh3.hash(''.join(random.choices(string.ascii_uppercase +
string.digits, k=10)), 1))%10000 for i in range(10**6)])
dist_vals, vals_frq = np.unique(inp, return_counts=True)
dist_vals, vals_frq = np.array(dist_vals), np.array(vals_frq)

mpl.style.use('seaborn')
plt.plot(dist_vals, vals_frq)
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram␣of␣Randomly␣Generated␣Numbers')
plt.savefig("uniform.png")
plt.show()
```

Code for creating correlation matrix of 50 hash functions.

```
mat = np.zeros((50, 10**5))
for j in range(mat.shape[0]):
    random.seed(0)
    for i in range(mat.shape[1]):
        rand_string = ''.join(random.choices(string.ascii_uppercase + string.digits, k=10))
        mat[j, i] = abs(mmh3.hash(rand_string, j))%1000

mat = mat.astype(np.floata64)
sns_plot = sns.heatmap(np.corrcoef(mat), vmin=-1, vmax=1, center=0)
sns_plot.figure.savefig("corr.png")
```

Implementation of bloom filter.

```
def update_bit_vector(list_nums):
    global bit_vector
    for num in list_nums:
        resulted_vector = [0]*params['m']
        resulted_vector[num] = 1
        bit_vector = [(op1|op2) for op1, op2 in
        zip(bit_vector, resulted_vector)]

params = {}
params['n'] = int(input("Enter input size : "))
prob = float(input("Enter error probability : "))
params['m'] = int(-params['n']*np.log(prob)*
math.pow(np.log(2), -2))
params['k'] = int(math.ceil(np.log(2)*
(params['m']/float(params['n']))))

print("k : " + str(params['k']))
print("m : " + str(params['m']))
bit_vector = [0]*params['m']

start = time.time()
for ip in range(params['n']):
    rand_string = ''.join(random.choices(string.ascii_uppercase +
    string.digits, k=10))
    seeds = []
    for i in range(1,params['k']+1):
        seeds.append(abs(mmh3.hash(rand_string, i)) % params['m'])

    update_bit_vector(seeds)

end = time.time()
print("Time to process all strings : " + str(end-start) + "s")


def check_availability(seeds):
    for idx in seeds:
        if bit_vector[idx] != 1:
            return 0

    return 1

def monte_carlo_sim():
    false_positives = 0
    num_examples = 10**6
    for i in range(num_examples):
        rand_string = str(uuid.uuid1())
        seeds = []
        for j in range(1,params['k']+1):
            seeds.append(abs(mmh3.hash(rand_string, j)) %
            params['m'])
        false_positives += check_availability(seeds)

    return false_positives/float(num_examples)


start = time.time()
sim_val = monte_carlo_sim()
end = time.time()
print("Using simulation false positive error probability : " + str(sim_val))
print("Simulation time : " + str(end-start) + "s")
analytic_prob = math.pow(1-math.exp(-(params['n']/float(params['m'])*params['k'])),
params['k'])
```

```
print("Analytically computed false positive error probability : ", str(analytic_prob))
print("False positive error probability given in Input : " + str(prob))
```

MATLAB code for creating 3D graph of performance of bloom filter.

```
x=600;
k=x:x+80;
m = 10^5:10^5:(81)*10^5;

n=100;
[K, M] = meshgrid(k, m);
Z = (1-exp(-K.*n*(1./M))).^K;

surf(K,M,Z)
title('Performance of bloom filter')
xlabel('k')
ylabel('m')
zlabel('False positive error rate')
colorbar
```