Bloom filters

Project Due: 8:00 am October 15, 2018

In this assignment you will implement a Bloom filter and then analyze the false positive rate that your implementation achieves versus the theoretical claim.

Restrictions

- You must complete this assignment on your own; do not share your code with anyone. Questions may be asked, and answered, via Piazza, so long as no code is shared with anyone except instructors.
- You must use **python (2.7.x)** to implement your project. Template code is provided and must be used.
- You may use any built-in python functions, functions provided by libraries already imported in the template code, and any plotting libraries, such as matplotlib. You may not use any other libraries or code from any other source, especially any related to hashing or Bloom filters.
- You must use the "random" function import (numpy.random) in the template code for random number generation. Do not use numpy for any other functionality in your graded code submission.
- You need to turn in a report (see below) to Gradescope, and a single source code file ("bloom.py") to Canvas. Please do not submit an archive or .zip file, but rather individual files.

Template Code

You have been given template code in the form of two .py files to assist you in completing this assignment. You must use this template code for Task 2, which will be checked and graded, and you may also use this template code to perform Tasks 1 and 3, if you wish.

The two template files you have been provided are "bloom.py" and "bfProjectUtils.py". You will modify and turn in only "bloom.py", so any code you add must be in "bloom.py", while "bfProjectUtils.py" holds utilities you may use, but should not modify or turn in. **The only python file you should submit to Canvas is "bloom.py".**

To execute your code, using the following command:

python bloom.py -c <config_file> -i <input_file> -o <output_file> -v <validation_file>

- config_file : (“testConfigHashType#.txt”) File name of configuration file holding constants defining the Bloom filter and the hash function for specified hash type that it should use (either 1 or 2). See file comments for more info.

- input_file : (“testInput.txt”) File name of file holding list of random integers drawn from universe of size N as described in <config_file>, the first *m* of which will be added into the Bloom filter, while all the values will be tested for membership in the Bloom filter.

- output_file : File name of output file, to hold results of Bloom filter test on data in <input_file>. You may name this however you wish.

- validation_file : (“validResHashType#.txt”) File name of known results to test your <output_file> against.

The configuration files contain data describing the format of the Bloom filter to be built, as well as precomputed seeds and coefficients used by the hash function calculations for either type of hash function (as described in Task 1). The template code provided reads this file and constructs a dictionary data structure holding the data that you should then consume as you construct your Bloom filter for Task 2.

“bfProjectUtils.py” contains multiple functions to assist you. Of particular note is **findNextPrime(n)** which will find the next prime number >= n.

When you are finished, before you submit “bloom.py” to T-Square, make sure Task 2 works using the following command line, as this is what will be used to test your program:

python bloom.py -c testConfigHashType2.txt -i testInput.txt -o testOutput.txt -v validResHashType2.txt

For a successful implementation should see displayed in the console :

compareResults : Your bloom filter performs as expected
Task 2 complete

This component is worth half of your project grade. We will also test your program against another (withheld) dataset using different configuration data, to verify the results for grading.

There are function stubs and optional command line arguments included in the sample code to help you perform Tasks 1 and 3, should you wish to use this code for these tasks, but this is not required. These code artifacts are described in comments within “bloom.py”.

Report

You need to turn in a report for this project, and a template file, “BloomFilterReportTemplate.odt” has been provided for you to use. Your report must follow the layout of the template, and you should only

use the space provided for your analysis and plots. Your report should be submitted to Gradescope as a .pdf file.

Your grade will depend partially on the quality of this report. This is a subjective measure – suggestions are given for each task about what to discuss, but the details are up to you. The idea is that we are considering using Bloom filters, let's say for a project at our company, and you must describe what design choices are available, as well as the advantages and disadvantages for each. Describe how Bloom filters work, and how well they perform compared to the theoretical results discussed in class. Provide plots/graphs where appropriate to illustrate your results.

Tasks

Your goal is to implement a Bloom filter that allows us to check efficiently whether an element is part of a given set or not, allowing for some false positives.

1. Design a suitable hash function

Let N denote the size of the universe, and assume that the elements will be non-negative integers from the set $U = \{0, 1, \ldots, N-1\}$ where N is some large positive integer. A Bloom filter internally uses a table or array to mark set membership. Let n be the size of this table, where $n \ll N$.

Your first task is to implement a hash function h that is appropriate for your Bloom filter. It must satisfy the following criteria:

- Maps any number from U to a random number in the range $[0, n-1]$ with uniform probability. The hash output will serve as an index into the Bloom filter's table.
- Different instantiations of the hashing function must produce outputs that are independent. That is, if k is the number of hash functions used and x is any element, the hashed values $h_0(x)$, $h_1(x)$, … $h_{k-1}(x)$, must not be correlated.
- You should be able to later figure out which bucket a given element was hashed to, requiring only a compact representation of the hash function.
- Each run should be random and independent of previous runs (so every time you run your program it should have different results, which are random and independent of previous runs).

Try the following two types of hash function:

1. For each of your *k* hash functions, pick a seed s_i. Let r(x) be the built-in random number generator seeded at x. Then the hash function is:

$$h_i(x) = r(s_i + x)$$

   (i.e. it's the random number generator seeded at $s_i + x$)
2. For each of your *k* hash functions, pick two random numbers a_i, b_i such that a_i is uniform in $\{1,2,\ldots,n-1\}$and b_i is uniform in$\{0, 1, \ldots, n-1\}$. Then the hash function is:

$$h_i(x) = ((a_i * x + b_i) \bmod P) \bmod n$$

Note: for this hash function to have the desired properties, both *P* and n must be prime, and must be P>= N; the template code you have been provided has a function you can use to find a suitable prime, or you can lookup online to get a prime number of the desired order of magnitude.

Test out these 2 hash functions (and, optionally, other choices you might find) with the random data provided. Try to compare these approaches; for example, you might look at a scatter-plot of the values mapped to, or compare the max/min loads (or other statistics) of the bins. Compare how they perform using data drawn randomly from the universe to how well they perform with data having some correlation (for example, data in a sequence, like "first 2m even numbers entered").

***Report**: Did one of the functions work better than the other? Did data choice matter? Why or why not? Justify your answer with relevant plots (on page 2 of the report template) and discussion. More important is how do these hash functions compare when utilized in your implementation of a Bloom filter.*

2. Implement a Bloom filter

Let m be the number of items in the subset *S* that your Bloom filter needs to represent. Let *n = cm* denote the size of your hash table. Let *N* denote the size of your universe (this should be enormous). Finally, let k denote the number of hash functions used.

Now implement the two operations of the Bloom filter:

- add(x): Adds the element x to the set.
- contains(x): Returns true if x is possibly in the set, false if it is definitely not.

For this task only (Task 2) you should not generate random values for the seeds/coefficients for your hash functions, but must instead use the provided values.

For this task you should implement Type 2 hash functions with your Bloom filter:

$$h_i(x) = \big((a_i * x + b_i) \bmod P\big) \bmod n$$

Note that the values for *n* and N (=*P*) provided in the Type 2 configuration file ("testConfigHashType2.txt") are both already prime, and should not be changed. For Type 2 hash functions and the given input file, "testInput.txt", the validation file that you must match is "validResHashType2.txt". The files with "Type1.txt" in their name can be used to test your Type 1 hash function implementation if you wish, but this will not be graded here.

***Report**: Discuss the details of your Bloom filter implementation on report page 1, in the space provided, along with any design choices you made.*

3. Analyze False Positive Rate for different values of k

Given a universe of size N (choose a huge N) and subset S where $|S| = m$, compute the false positive rate for your Bloom filter for different choices of c and k, and compare its performance with theoretically derived values.

To do this you must perform multiple trials with your Bloom filter for each value of c and k (# of hash functions), rebuilding the Bloom filter every time and deriving new values for the hash function coefficients, and take the average of the performance across all the trials.

You may either use the included "input.txt" file as a data source, or generate your own data (m = 100000 is sufficient, set $N = 2^{32} - 1$). Performing 10 trials for each value of c and k should be sufficient to provide reasonable results. Fix c first to be 10, and then 15, and sweep k from .4 * c to 1.0 * c. So, for example, you should perform 10 trials with c = 10, k = 4, regenerating coefficients for each trial, then 10 with c = 10, k = 5, etc. You must perform this with both types of hash function described in Task 1.

You should plot a curve for each value of c and each hash function (so a total of 4 plots), where the x-axis on each plot is k and the y-axis is the average *false positive rate* across all trials for a particular c and k combination. On each plot also include the theoretical curve of false positives vs. k using the equation given in the notes:

$$f(k) = \left(1 - e^{-\left(\frac{k}{c}\right)}\right)^k$$

Lastly, mark where the theoretically predicted optimal k value is and this value's theoretically derived false positive value.

***Report**: Make sure to include the 4 relevant plots in your report on page 3 in the spaces provided. Discuss the results of your experiments on page 1. Briefly describe how you derived your theoretical values. How do your results compare to the theoretical false positive rate which is a function of k and c? Do your experiments agree with the theoretical result for the optimal choice of k? Justify your discussion with the relevant plot/graphs. Any other conclusions, insights, or observations?*