Lab 2 - Bloom Filters

Due: Friday February 24th 11:59PM in PDF form via Websubmit

1 What to submit (MANDATORY)

1. PDF file ((lab2_firstname_lastname.pdf)) including plots and a snapshot of the running code used to answer the questions.
   - Names of the collaborators.
   - Number of late days for this assignment.
   - Number of late days used so far.
   - References used
2. Python script (lab2_firstname_lastname.py) with the code used – i.e. rename the edited Bloom-Filter.py skeleton code used.
3. Also, include the unedited bloomFilterHash.py code.

Failing to meet any of the above requirements will cause a decrease of your grade.

2 Background

A Bloom Filter is one of many different probabilistic data Structures.

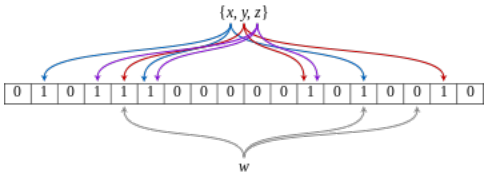

Figure 1: An example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit-array position containing 0. For this figure, m=18 and k=3. Source: http://en.wikipedia.org/wiki/Bloom_filter

Bloom Filters take advantage of hashing to keep records of data with a very low false positive rate. By combining hashing and a bit-array, Bloom Filters can store a lot of data in a very small space. However, as with most randomness, Bloom Filters have an inherent tradeoff, high space efficiency vs a small false positive probability.

A Bloom Filter consists of:

1. An n bit array, A, initialized to 0.
2. k-independent, random hash functions (this is important so keep it in mind). We assume the hash functions, $h_1, .., h_k$, map uniformly and randomly to the range 0,...,$n-1$. A hash function h maps strings of length p to shorter strings of length q. That is:

$$h : \{0,1\}^p \longrightarrow \{0,1\}^q$$

   In the case of a Bloom Filter, p is the length of the input and q is simply $\log(n)$. Explain why $q = \log(n)$.

Bloom Filters have two basic operations: insertion and lookup.

Insertion: for an element s_j the bits A$[h_i(s_j)]$ are set to 1. This is to say, run each of the k hash functions on s_j - remember that the hash function will output a number 0,...,$n-1$ - and set A[index specified by the hash] to 1. If $A[h_i(s_j)] = 1$, don't change it.

Lookup: to check if s_i is in the Bloom Filter check if every bit that it hashes to is set to 1. If this is not the case, you definitely have not inserted it to the filter, for while the hash functions randomly map elements to indices remember that hash functions are always consistent. If all the bits are one, then we are in one of two cases:

1. True positive: s_i is in the Bloom Filter
2. False positive: s_i is not in the Bloom Filter. Because of previous insertions we performed, these bits have happened to be set to 1, so it looks like s_i is in the Bloom Filter, even though it really isn't.

We have run in to the fundamental trade off of Bloom Filters - we have no idea whether we have a false positive or a true positive when we run a lookup.

For this reason, our goal will be to *design the Bloom Filter* in a way that minimizes the false positive probability, so that most of the time our lookups will return true positives. (Even though we can't tell the difference between true and false positives.) The point behind this lab is to understand how to design a bloom filter in a way that minimizes the false positive probability.

3 The Math

1. On a given insertion, what is the probability that a specific bit stays 0? Keep in mind we have k hash functions, and an n-bit array. Explain your reasoning. Hint: consider the probability of one bit being set to 1. Now remember the hash functions are independent - what can we do with independent probabilities?
2. Suppose we have inserted m elements in to the Bloom Filter, what is the probability that any specific bit is still 0? And the probability the bit has been flipped to 1? Explain your reasoning.
3. Now we have all the pieces to determine the **false positive probability** with k hash functions, into a Bloom Filter of size n, after m insertions. Imagine you're looking up an element s in this Bloom Filter. Further, you know (based on previous knowledge) that you did not insert this element to the Bloom Filter. What is the probability that the lookup operation returns true (that s has been inserted to the Bloom Filter)? Assume that the probability of each bit being set is independent. Hint: recall that $1 - x \approx e^{-x}$
4. Using calculus, we can minimize a function, by setting the derivative to 0. Assuming that m and n are fixed, what is the value for k that minimizes the probability of false positives?
5. Notice that $\frac{n}{m}$ is the 'blowup' in the storage size of the Bloom Filter. What does this tell us about the number of hash functions we need to keep a constant false positive probability?
6. Explain why our current implementation of the Bloom filter does not allow us to delete items.

   Bonus: Can you think of a simple modification to the Bloom Filter to support deletion?
7. Imagine you want to store 100000 integers with $1 * 10^7$ digits. How many bits do you need to store all of these integers directly?
8. Using the math above, how many bits do we need to store 100000 items in a Bloom Filter with false positive probability $< .1\%$, using 5 hash functions? How much storage space do you save by switching to a Bloom Filter?

4 Python and Statistics

Now that we've worked through the math, lets work on building our own Bloom Filter in Python.

We have provided some starter code for you so open up bloomFilterHash.py and BloomFilter.py Take a look inside bloomFilterHash.py. Note: you will only need to modify BloomFilter.py.

bloomFilterHash is implemented as a Python class - you can read about Python Classes in the tutorial (http://docs.python.org/tutorial/classes.html). You'll only need some basic class functionality for this lab.

- __init__ method is a constructor like in Java.
- All other functions are user defined. Every class function takes at least one parameter, self, which you do not need to pass when calling the function. It is simply the way Python gains access to other elements of the class.
- Class variables can be defined right above the __init__ method and accessed using the self keyword (i.e. self.variable = 1) from anywhere in the class definition.

The only functions in bloomFilterHash you will need to use are hash_i() and pairwiseHash_i().

hash_i() simply takes your key (s_j in the Bloom Filter definition), and hashes it to a number in the range [0, numBits-1], using hash function i. We're using a hash function from Python's hashlib which gives us hexadecimal values. **It only works if you use the array sizes we specify.**

Now take a look inside Bloom-Filter.py. It's a skeleton for the Bloom filter class you are going to build. We have already imported bloomFilterHash and created an instance of it in the constructor for your Bloom-Filter. Use this object, called hash, to gain access to the hash_i() and pairwiseHash_i() functions. For example, if you want to hash the string "thisismykey" and the 4th hash function, use the following line of code.

```
hashValue = hash.hash_i("thisismykey", 4)
```

Note: Remember, anywhere you want to use a class variable, access it with the self keyword. See bloomFilterHash for an example

9. Complete the other two functions in BloomFilter.py:
   - insert: pass it a key and insert into the BF; does not need a return value
   - lookup: pass it a key and test for membership in the BF; return a boolean value

   Submit the code with your assignment.

10. Add a function to your Bloom Filter rand_inserts() that inserts randomly generated data into your Bloom Filter. Generate the data using numpy.random functions and take it from the the range $[0, 1 * 10^8]$.

11. Create a Bloom Filter with 600 (randomly generated) insertions, using 10 hash functions, and 4095 bits. What is the theoretical false positive probability (use the formula from question 3)?

    Hint: Use the str() function and numpy.random functions to generate random strings

12. Now experiment with randomly generated data using 10 hash functions, 4095 bits, and 600 insertions. Using your code, compute the false positive probability. In order to compute the false positive probability, *lookup()* at least 10000 randomly generated items and keep track of the number of positive results.

    Repeat the experiment 1,000 times. What is the average false positive probability?

Note: The random data you generate to lookup could overlap with the random data you generate while inserting data into the Bloom Filter. We assume that with high probability it will never overlap which means that every positive lookup() is a false positive.

13. Now let's vary the number of hash functions. Again, using 4095 bits, and 600 insertions, generate a plot with number of hash functions on the x-axis, and average false positive probability on the y-axis with 10,000 reps of the experiment. Use values of [2,3,4,...30] for the x-axis. What does this plot tell us about the number of hash functions we need? Explain why.

5 Independence and Hash Functions

In Section 1 we mentioned the requirements for Bloom Filters to obtain their theoretical bound, an n-bit array, and k-independent, random hash functions.

In this section we are going to slightly relax the requirement for independent hash functions and see whether or not we can get a similar guarantee. In order to do this, we need to introduce a new topic, pairwise independence.

Events $A_1, A_2, .., A_k$ are pairwise independent if for any two events A_i, and A_j, $Pr(A_i \cap A_j) = Pr(A_i)Pr(A_j)$. Note: pairwise independence is a weak form of independence of an entire set: it is not true that all events are independent of each other.

Consider an example illustrating how pairwise independence is a weak form of independence: Suppose X and Y represent independent tosses of a fair coin (heads = true, tails = false), and suppose random variable Z is true if the coin flips are different, and false otherwise.

- $A = Pr(\text{first coin is heads}) = \frac{1}{2}$
- $B = Pr(\text{second coin is heads}) = \frac{1}{2}$
- $C = Pr(\text{first and second coins match}) = \frac{1}{2}$

14. Show that these events are pairwise independent
15. Show that while the events are pairwise independent, the set of events (A,B,C) is not independent.

So, does relaxing the requirement for independent hash functions to pairwise independent hash functions realistically affect us? Lets find out.

16. Modify your Bloom Filter class so its constructor accepts one more boolean parameter, `pairwise`. Change the insert() and lookup() functions to use pairwiseHash_i() instead of hash_i() if the pairwise parameter is true.

17. If we have 6 independent hash functions, and a Bloom Filter with 600 items inserted into it and 4095 bits what is the theoretical false positive probability with **independent** hash functions?

18. Similar to question 12, run 1000 experiments with random input using a 4095 bits, 600 insertions and 6 hash functions. What is the average false positive probability? What is the max?

19. Repeat the experiment in question 18 except this time use pairwise independent hash functions. What is the average false positive probability? And the max?

20. What do your results show us about Pairwise Independent hashing?

6 Security of Hash Functions

In this final section, we are going to demonstrate a very important point about hashing - if an adversary has access to your hash function, its easy for it to choose data that will destroy all the good things that hashing gets us.

In this spirit, add the following two functions to your script (found in the provided adversarial.py). Also, add the class variables above adversarialInserts(). Both of these functions assume knowledge of the pairwiseHash_i() function. So when experimenting below, use pairwise independent hashes.

```
        self.badInserts = []
    self.badLookups = []

    def adversarialInserts(self):
        self.personalBF = [0]*self.numBits
        self.badInserts = []
        count = 0
        for i in range (16384):
            flag = True
            val = [0] * self.numHashFunctions
            for j in range (self.numHashFunctions):
                val[j] = self.hash.pairwiseHash_i(i, j)
                if self.personalBF[val[j]] != 0:
                    flag = False
            if flag:
                self.badInserts.append(val[j])
                for j in range (self.numHashFunctions):
                    count += 1
                    self.personalBF[val[j]] = 1
        for i in range(len(self.badInserts)):
            self.insert(self.badInserts[i])
        return self.badInserts

    def adversarialLookups(self):
        self.badLookups = []
        for i in range (16384):
            if i not in self.badInserts:
                if self.lookup(i):
                    self.badLookups.append(i)
        count = 0
        for i in range (len(self.badLookups)):
            if self.lookup(self.badLookups[i]):
                count += 1
        return count * 1.0 / len(self.badLookups)
```

21. Run the test from question 18 except use adversarialInserts() instead of randInserts(). What average and max false positive probabilities do you notice? Note: adversarialInserts() takes data from a smaller range than randInserts(). This is only because it takes a little while for the function to run - feel free to increase the range. The same holds true for adversarialLookups()

22. Now, run the test from question 18 using adversarialInserts() and adversarialLookups(). Adversarial lookups returns a float representing the false positive probability. What max and average do you observe this time?

7 Summary

In this lab we've covered quite a bit of material. It boils down to a few points.

- Bloom Filters can store a lot of data with very few bits. The trade off is a small false positive probability
- Truly independent hashes are ideal, but pairwise independent hashes can come close.
- Finally, we must keep our hash functions well hidden or else an adversary can easily attack our system.