CMSC 420: Coding Project 6
Bloom Filters

1 Due Date and Time

Due to Gradescope by Sunday 14 May at 11:59pm. You can submit as many times as you wish before that.

2 Get Your Hands Dirty!

This document is intentionally brief and much of what is written here will be more clear once you start looking at the provided files and submitting.

3 Assignment

We have provided the template `bloom.py` which you will need to complete. This file defines a Bloom filter which in turn manages the recording of passwords which have experienced hacking attempts.

3.1 Tracefile

The idea behind this project is that the tracefile initializes a Bloom filter and then contains a record of usernames which have experienced hacking attempts. After a certain number of hacking attempts the username should be stored in a Bloom filter (so that when the user attempts to log in next, a warning is given).

The first line in the tracefile initializes the Bloom filter and has the form:

`initialize,m,k,threshold,fpmax`

For example:

`initialize,5,4,3,0.20`

In this case:

- `m = 5` is the number of bits in the Bloom filter so the filter itself is the `bitarray` with 5 bits. This also means that each hash function will map from the set of usernames (which are strings) to an integer between 0 and 4.
- `k = 4` is the number of hash functions used. Details are given later as to how these hash functions are to be written.
- `threshold = 3` is the `threshold` such that after a username has undergone 3 hacking attempts it should be added to the Bloom filter.
- `fpmax = 0.20` is the maximum acceptable probability of a false positive above which we must rebuild the Bloom filter. More details on rebuilding are given later.

Each successive line except for the last one has the form:

`hack,username`

This records the fact that `username` was hacked.

The last line in the tracefile is either `dump`, which dumps the Bloom filter as a simple binary string, or `check,username` which checks to see if `username` is in the Bloom Filter.

3.2 Hash Functions

For a given Bloom filter there will be a family of k hash functions indexed by $j = 1, ..., k$. For each j the hash function is implemented as follows:

- Take the `username` (a string) and convert each character to ASCII using `ord`.
- Add the values together.
- Concatenate the number with itself until the result is m or greater.
- Raise it to the j power.
- Extract the same number of leftmost digits as m has.
- Take the result mod m.

For example, suppose we call:

```
hash('justin',5,10000)
```

In this case we implement the $j = 5$ hash function and we need a result between 0 and 9999. We proceed as follows:

- Username: `justin`
- ASCII: `[106, 117, 115, 116, 105, 110]`
- Sum: 669
- Concatenate until 10000 or greater: 669669
- 669669**5 : 134679339003648440230419539349
- Needs 5 digits since $m = 10000$: 13467
- Result mod $m = 10000$: 3467

3.3 False Positive Measurements

The probability of a false positive should be measured using the slightly inaccurate but convenient:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

3.4 Rebuilding

When insertion of a new username causes the probability of a false positive to be larger than the acceptable probability `fpmax` we rebuild the Bloom filter to have the smallest possible m-value such that the new probability of a false positive is less than half the acceptable probability.

3.5 Recording Usernames

In order to count the number of hacking attempts for each username and in order to rebuild the Bloom filter we need to keep a record of the usernames we have entered and how many times they have been hacked. These should be stored in the dictionary `usernamedict` whose keys are usernames and whose values are the number of hacking attempts.

4 Details of the File

There is one function you should complete which is not an instance method. This is specifically done so it can be tested independently:

- `def hash(username:str,j:int,m:int) -> int:`

  Apply the j^{th} hash function to `username` and return an entry between 0 and $m-1$. See above for how this works.

The instance methods should do the following:

- `def hack(self,keyusername: str):`

  Record a hacking attempt on the username `username`. This should store the username in the `usernamedict` and update the number of hacking attempts.

- `def insert(self,keyusername: str):`

  Insert the `username` into the Bloom filter. This should only be called when the number of hacking attempts has reached the `threshold`.

- `def check(self,username:str) -> str:`

  Check if the `username` is in the Bloom filter. If it is, then return:

  ```
  json.dumps({'username':username,'status':'UNSAFE'}
  ```

  If it is not, then return:

  ```
  json.dumps({'username':username,'status':'SAFE'}
  ```

5 Additional Functions

You will probably want some additional functions as well as helper functions to handle the necessary operations.

6 What to Submit

You should only submit your completed `bloom.py` code to Gradescope for grading. We suggest that you begin by uploading it as-is (it will run!), before you make any changes, just to see how the autograder works and what the tests look like. Please submit this file as soon as possible.

7 Testing

The testing can be categorized into three types:

1. First a stress test is performed on the `hash` function so write that code first.
2. After that, tracefiles are contructed of type `dump with no rebuilds` and `hack with no rebuilds`. These are just like the tracefiles as described earlier except they are specifically designed so that no Bloom filter rebuilding will need to occur. This allows you to test all the code except for the rebuilding, since rebuilding will never happen.
3. Finally full tracefiles are constructed of varying size as described earlier.

8 Local Testing

We have provided the testing file `test_bloom.py` which you can use to test your code locally. Simply put the lines from a tracefile (either from the autograder or just make one up) into a file `whatever` and then run:

```
python3 test_bloom.py -tf whatever
```