Lab 9: Flajolet-Martin

May 28th, 2014

Objectives

In this lab, we will get some hands-on experience with the Flajolet-Martin algorithm. This algorithm computes an approximation of the number N of distinct elements in a large stream of data using only $O(\log N)$ bits of memory. This is quite remarkable: given that $\lceil \log N \rceil$ is the number of bits it takes just to describe N, Flajolet-Martin is as efficient as can be. Over the course of the lab, you will

- implement key parts of the algorithm,
- understand issues related to combining several estimates to improve the approximation, and
- apply the algorithm to a large dataset on which an exact count would be difficult.

Deliverables

A `pdf` file containing answers to questions **Q1** to **Q5** in the lab. Exceptionally, we will not have mini-interviews on this lab and your grade will be determined exclusively based on your handin.

1 Implementing the Algorithm

There is a variety of different implementations of the Flajolet-Martin algorithm. Here, we quickly describe the particular setup used in this lab. We recall that the goal of the algorithm is to recover the number of distinct elements N in a stream; for more information about the algorithm in general, refer to the course notes.

Let h be a hash function that maps a string x to C bits. For example, for $C = 8$ and $x =$ "anaytics", the hash could be

$$h(\text{“analytics”}) = \underbrace{00101101}_{\text{8 bits}}.$$

The requirement we place on the hash functions is that they map strings more or less uniformly on all the 2^C possible outputs. Futhermore, instead of considering the hash itself directly, we actually only look at its number of leading zero bits (in the example above: 2.) As we iterate over a stream of strings, we store the highest number of leading zero bits we have seen so far. The intution is that if the hash function maps strings to the output space uniformly,

- 1/2 of the strings have 0 as their first bit,
- 1/4 of the strings have 00 as their two first bits,
- 1/8 of the strings have 000 as their three first bits, etc.

By keeping around the rarest occurence encountered so far, we can estimate how many distinct strings we have seen. Let R be the maximum number of leading zero bits, our estimate for N is therefore

$$\hat{N} = 2^R.$$

To improve our estimate, we can run multiple instances in parallel. This is done by defining L different hash functions $h_1, \dots, h_L$, which give us independent estimates $R_1, \dots, R_L$.

Our implementation of Flajolet-Martin (`FlajoletMartin.java`) takes C and L as parameters (`nbBits`, respectively `nbHashes`.)

> **Note**: as we want to be able to deal with large values of N, prefer `long` to `int` everywhere you deal with count estimates.

- Start by familiarizing yourself with the code. Look at how `HashFunction` is implemented, and how a `FlajoletMartin` instance is initialized.
- Complete the function `processWord()`. Once you're done, check your code with the provided test.
- When the stream has been entirely processed, we need to convert $R_1, \dots, R_L$ into the estimates $2^{R_1}, \dots, 2^{R_L}$. Write the necessary logic in `countUniqueWords()`, and make it return the *average* of the estimates.
- You would like to compute the richness of the vocabulary (i.e. the number of distinct words) of *Moby-Dick*, the famous book by American author Herman Melville. Run `FlajoletMartin` on `data/moby-dick.txt` (pass the file path as first argument to the program.) To compare your estimate $\hat{N}$ against the true number of unique elements N, use the class `ExactCount`.
- As an alternative to returning the average of the estimates as our final estimates, we can return the *median*. Complete the function `median()`, test your implementation using the unit test, and use it in `countUniqueWords()`. Run Flajolet-Martin on *Moby-Dick* again a few times.

- **Q1** Compare the estimates you get using the average and the median. Both have advantages and disadvantages—describe what you observe, and explain.

As both average and median have undesirable properties, we will combine $R_1, ..., R_L$ as follows, in order to obtain our final estimate.

1. Group the estimates by buckets of 3, and compute their *average.*
2. Output the *median* of the bucket averages as the final estimator.

Implement this idea in `countUniqueWords()`. **Q2** Include your final version of this function in the handin.

Increase `nbHashes` in `main()` to 100, and run the algorithm again on *Moby-Dick*. You should now get a fairly consistent estimate[1].

2 Large Streams

In this section, we will move away from *Moby-Dick* and look at a substantially larger dataset, one that makes it very difficult to run an algorithm that keeps all distinct items concurrently in memory. This file is available on HDFS, under `/ix/secret.txt`. It is incovenient to copy it anywhere, therefore you will directly stream it (line by line) using

```
hadoop fs -cat /ix/secret.txt
```

There is also a sample available under `/ix/secret-sample.txt`, in case you'd like to test your program more quickly at first.

Start by exporting a runnable JAR containing your program into a file named `ix-lab09.jar`, and copy it over to one of the cluster access servers, for example via `myfiles`.

Exporting a Runnable JAR: in Eclipse, right click on your project, and choose *Export...*. Then, select *Java* → *Runnable JAR* and click on *Next*. Select any launch configuration belonging to the current project (e.g. *FlajoletMartin*) and make sure *Extract required libraries into generated JAR* is checked. Set *Export destination* to your favorite folder, e.g. `myfiles`.

When there is no file passed as first argument, the program `FlajoletMartin` will read from the standard input stream[2]. This means that you can run the following command to run the algorithm on the dataset:

```
hadoop fs -cat /ix/secret.txt | java -cp ix-lab09.jar ix.lab09.streaming.FlajoletMartin
```

The pipe symbol "|" redirects the output of the first command into the input of the second command. Note that processing the whole dataset can take some time—about 20 minutes.

- **Q3** What estimate $\hat{N}$ do you get? Look at the value of `nbBits` in the function `main()`, and explain what problem might arise in this case.
- Increase `nbBits` to 40, and export a JAR with the updated version. Run Flajolet-Martin on `secret.txt` again. **Q4** What estimate $\hat{N}$ do you get this time? Based on this estimate, what is the smallest value that `nbBits` could take for this particular instance of dataset?
- **Q5** How much memory (in bytes) would you roughly need if you if you were to run `ExactCount` on the dataset? Justify your answer.

[1]You might observe that the output has a tendency to overestimate the true count—do not worry too much about it, this is mostly due to the small scale of the numbers involved

[2]See: `http://en.wikipedia.org/wiki/Standard_streams` for more explanations.