Big Data Analytics Programming

Exercises

Laurens Devos *laurens.devos@cs.kuleuven.be*

2022-2023

Exercise Session 1: Approximations

1.1 Introduction

In this exercise, you will experiment with three different approximate algorithms: *HyperLogLog*, *Bloom filters*, and *reservoir sampling*.

We will carry out some experiments on the following data set: https://people.cs.kuleuven.be/~laurens.devos/bdap/enwiki-2013-frequencies.txt.gz[1] (also available on Toledo). This file contains the number of outgoing links of 4.2 million Wikipedia pages.

We will be reading directly from the `gzip` file, so you do not need to decompress it. You can explore the dataset using the following command: `gzip -cd path/to/enwiki-2013-frequencies.txt.gz | head`.

The `Makefile` included in the assignment's ZIP file contains targets to run your code: use `make <target>` with `<target>` substituted by `HyperLogLog`, `BloomFilter`, or `ReservoirSampling`.

1.2 Probabilistic counting: *HyperLogLog*

In this exercise, you will implement the *HyperLogLog* algorithm for estimating the number of distinct elements in a multiset. For small multisets, this is a trivial task: we can keep a count for each item in memory. For large datasets, this approach becomes infeasible. Do you know why?

Probabilistic counting algorithms such as HyperLogLog can provide a reasonable estimate of the number of distinct items using only a small amount of memory. A description of the algorithm is given in the course slides, and a more detailed discussion of the algorithm can be found in the paper: http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf (See Fig. 3). Note that the algorithm in the paper assumes array indexes start at 1, whereas in Java, it starts at 0.

Here are some example usages of the HyperLogLog algorithm. An Internet provider might use HyperLogLog to track the number of different IP addresses that are routed through their network. A large website might use it to track the number of unique visitors. In this exercise, we will try to find the number of unique frequency values in the Wikipedia dataset.

You can find a Java file called `HyperLogLog.java` in the assignment ZIP. You should implement the `computeApproximateCardinality` function. Some useful Java commands are:

- `y = x << b`
  This removes the first `b` bits from integer `x`.
  For example `1101001011000... << 4 = 001011000...`

[1] The dataset we use is derived from the *English Wikipedia hyperlink network* dataset, accessible from https://snap.stanford.edu/data/enwiki-2013.html. The original dataset contains information of 101,311,614 links between 4,203,323 pages.

- `y = x >>> (32-b)`
  This retains the first b bits from 32-bit integer `x` (32 bit is default in Java).
  For example `1101001011000. . . >>> (32 - 4) = 1101`
- `Integer.numberOfLeadingZeros(x)`
  This returns the number of leading zeros in `x`.
  For example `Integer.numberOfLeadingZeros(0b001011000. . . ) = 2`
- `hash(some_integer)`
  This is a static function in the HyperLogLog class. Use this to generate a hash from an `int` value. This function uses the MurmurHash, for which an implementation is included in the ZIP file.

The algorithm uses $m = 2^b$ registers. The value b determines how you should split the 32-bit hashed integers into two parts: (1) an index value into one of the registers, and (2) a number of bits for which we determine the position of the most significant non-zero bit. How does varying values of $b \in [4, 16]$ influence the estimate? Do the experiments confirm this?

Although we are working with over 4 million distinct data points, the cardinality of the frequency values of outgoing links in Wikipedia pages is relatively small. The paper proposes a *small range correction*. How does this affect the estimate?

1.3 Answering member-of questions: *Bloom filters*

Bloom filters are an efficient probabilistic data structure that can be used to determine whether an element s is a member of a set S: $s \in S$. A description of a Bloom filter can be found in the course slides. Bloom filters only make one kind of mistake: it might answer that some s is an element of S when it actually is not (false positive), but it never claims that some s' is not in S when it actually is (false negative). Can you explain why this is true?

We will use Bloom filters to get a second estimate on the number of frequency values in the Wikipedia dataset. We have seen in the previous exercise that the cardinality is fairly low. An alternative approach to computing the cardinality is the following: for each value in the dataset, check whether we have seen it before using a Bloom filter. If we have not, increment the cardinality count. Finally, insert the value into the Bloom filter so that we do not count it again the next time we see it.

In the file `BloomFilter.java`, implement the `computeApproximateCardinality` function. A Bloom filter has two main parameters: the size M of the bitset and the number of times b a value is hashed in to the data structure. Use the function `hashIndex(some_int, max_index, seed)` to hash into the bitset data structure. A small implementation of a bitset is already provided. How large does M need to be to get an accurate estimation? How does increasing the number of hash functions b affect the performance when M is relatively low? What if M is relatively large? Does it ever overestimate the cardinality?

1.4 Reservoir sampling

In this exercise, the goal is to implement reservoir sampling, an algorithm that randomly samples k elements from a large dataset or stream. We will use reservoir sampling to estimate the average number of outgoing links on a Wikipedia page. We will do this by randomly selecting k pages from the dataset and computing the average on only those frequency values.

In the file `ReservoirSampling.java`, fill in the missing parts of the function `printApproximateMean`. Answer the following questions:

- How large does your sample need to be to get an approximation that is correct up to 0.1 outgoing links, when averaged over 10 runs?
- Under what properties of a data stream is the reservoir sampling algorithm described in class not good? How could you fix it? Note this is does not refer to the data or any properties of the data.

1.5 Misra Gries Algorithm: Proof of Correctness

The Misra Gries algorithm is an approximate sketching algorithm used to count how often the most frequent elements appear in a stream of elements using only $k << n$ counters, with n the number of distinct elements in the stream.

The algorithm works as follows. Maintain space for at most k counters. Initially, all these counters are unclaimed. Processing an element x in the stream:

- **If** there is a counter for x, increment it.
- **Else if** there is **no** counter for x, but we still have an unclaimed counter lying around, then claim this counter for x and initialize it to 1.
- **Else** decrease all counters by 1, and remove all counters with a count of 0. These counters can be reclaimed later.

The counters are an underestimation of the true number of occurrences of the associated elements. It is possible to keep track of a maximum distance m to the real number of occurrences.

This algorithm can be used in a distributed setting: multiple processes running on different computers can process multiple streams independently. The counters of the individual processes can be merged to produce a global count estimate as follows:

- Phase 1:
  - **If** a counter for x is present in both structures, then sum them up.
  - **Else if** a counter for x is present in one of the two structures, keep the counter.
- Phase 2:
  - Take the $(k+1)$th largest counter.
  - Subtract its value from all counters.
  - Delete all counters that are less than or equal to 0.

We now want to prove that any counters in a merged sketch is smaller than the true number of occurrences by at most:

$$\frac{m - m'}{k+1}, \tag{1}$$

where k is the number of counters, m' is the sum of all counters, and m is the number of elements in the stream (not necessarily distinct). **Prove that this is true.**

Hint: first, prove the property for a single sketch, then prove that the property remains true after applying the merge step.