PDSA

Feb 24, 2020

Contents

Probabilistic data structures is a common name of data structures based on different hashing techniques.

Unlike regular (or deterministic) data structures, they always give you approximated answers and usually provide reliable ways to estimate the error probability.

The potential losses or errors are fully compensated by extremely low memory requirements, constant query time and scaling.

GitHub repository: https://github.com/gakhov/pdsa

CHAPTER 1

Quickstart

```
from pdsa.membership.bloom_filter import BloomFilter

bf = BloomFilter(80000, 4)

print(bf)
print("Bloom filter uses {} bytes in the memory".format(bf.sizeof()))

print("Filter contains approximately {} elements".format(bf.count()))

print("'Lorem' {} in the filter".format(
    "is" if bf.test("Lorem") else "is not"))

words = set(LOREM_IPSUM.split())
for word in words:
    bf.add(word.strip(" .,"))

print("Added {} words, in the filter approximately {} elements".format(
    len(words), bf.count()))

print("'Lorem' {} in the filter".format(
    "is" if bf.test("Lorem") else "is not"))
```

CHAPTER 2

Cardinality

The cardinality is the number of distinct elements in a set.

Calculating the exact cardinality of a multiset requires an amount of memory proportional to the cardinality, which is impractical for very large data sets.

2.1 Linear Counter

A Linear-Time probabilistic counting algorithm, or Linear Counting algorithm, was proposed by Kyu-Young Whang at al. in 1990.

It's a hash-based probabilistic algorithm for counting the number of distinct values in the presence of duplicates.

The algorithm has O(N) time complexity, where N is the total number of elements, including duplicates.

This implementation uses bitvector to store the counter's array.

```
from pdsa.cardinality.linear_counter import LinearCounter

lc = LinearCounter(1000000)
lc.add("hello")
print(lc.count())
```

2.1.1 Build a counter

To build a counter, specify its length.

```
from pdsa.cardinality.linear_counter import LinearCounter

lc = LinearCounter(100000)
```

Note: Memory for the counter is assigned by chunks, therefore the length of the counter can be rounded up to use it in full.

Note: This implementation uses MurmurHash3 family of hash functions which yields a 32-bit hash value that implies the maximal length of the counter.

2.1.2 Index element into the counter

```
lc.add("hello")
```

Note: It is possible to index into the counter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

2.1.3 Size of the counter in bytes

```
print(lc.sizeof())
```

2.1.4 Length of the counter

```
print(len(lc))
```

2.1.5 Count of unique elements in the counter

```
print(lc.count())
```

Warning: It is only an approximation, that is quite good for not huge cardinalities.

2.2 Probabilistic Counter

Probabilistic Counting algorithm with stochastic averaging (Flajolet-Martin algorithm) was proposed by Philippe Flajolet and G. Nigel Martin in 1985.

It's a hash-based probabilistic algorithm for counting the number of distinct values in the presence of duplicates.

This implementation stores number of 32-bit single counters (FM Sketches) consequently in a single bitvector.

```
from pdsa.cardinality.probabilistic_counter import ProbabilisticCounter

pc = ProbabilisticCounter(256)
pc.add("hello")
print(pc.count())
```

2.2.1 Build a counter

To build a counter, specify its length.

```
from pdsa.cardinality.probabilistic_counter import ProbabilisticCounter

pc = ProbabilisticCounter(number_of_counters=256)
```

Note: Memory for the counter is assigned by chunks, therefore the length of the counter can be rounded up to use it in full.

Note: This implementation uses MurmurHash3 family of hash functions which yields a 32-bit hash value that implies the maximal length of the counter.

Note: The Algorithm has been developed for large cardinalities when ratio `card()/num_of_counters > 10-20`, therefore a special correction required if low cardinality cases has to be supported. In this implementation we use correction proposed by Scheuermann and Mauve (2007).

```
from pdsa.cardinality.probabilistic_counter import ProbabilisticCounter

pc = ProbabilisticCounter(
    number_of_counters=256,
    with_small_cardinality_correction=True)
```

2.2.2 Index element into the counter

```
pc.add("hello")
```

Note: It is possible to index into the counter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

2.2.3 Size of the counter in bytes

```
print(pc.sizeof())
```

2.2.4 Length of the counter

```
print(len(pc))
```

2.2.5 Count of unique elements in the counter

```
print(pc.count())
```

Warning: It is only an approximation of the exact cardinality.

2.3 HyperLogLog

HyperLogLog algorithm was proposed by Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier in 2007.

It's a hash-based probabilistic algorithm for counting the number of distinct values in the presence of duplicates.

This implementation uses the classical algorithm with a 32-bit hash function and 4-byte counters.

```
from pdsa.cardinality.hyperloglog import HyperLogLog

hll = HyperLogLog(10)
hll.add("hello")
print(hll.count())
```

2.3.1 Build a counter

To build a counter, specify its precision - the number of bits that should be used to randomly choose the counter (stochastic averaging). The rest of the bits of the 32-bit hash value will be used to index into the selected counter.

```
from pdsa.cardinality.hyperloglog import HyperLogLog

hll = HyperLogLog(precision=10)
```

Note: Precision has to be an integer in range 4 . . . 16.

Note: This implementation uses MurmurHash3 family of hash functions which yields a 32-bit hash value.

2.3.2 Index element into the counter

```
hll.add("hello")
```

Note: It is possible to index into the counter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

2.3.3 Size of the counter in bytes

```
print(hll.sizeof())
```

2.3.4 Length of the counter

```
print(len(hll))
```

2.3.5 Count of unique elements in the counter

```
print(hll.count())
```

Warning: It is only an approximation of the exact cardinality.

CHAPTER 3

Frequency

Many important problems with streaming applications that operate large data streams are related to the estimation of the frequencies of elements, including determining the most frequent element or detecting the trending ones over some period of time.

3.1 Count Sketch

Count Sketch is a simple space-efficient probabilistic data structure that is used to estimate frequencies of elements in data streams and can address the Heavy hitters problem. It was proposed by Moses Charikar, Kevin Chen, and Martin Farach-Colton in 2002.

3.1.1 References

[1] Charikar, M., Chen, K., Farach-Colton, M. Finding Frequent Items in Data Streams Proceedings of the 29th International Colloquium on Automata, Languages and Programming, pp. 693–703, Springer, Heidelberg. https://www.cs.rutgers.edu/~farach/pubs/FrequentStream.pdf

This implementation uses MurmurHash3 family of hash functions which yields a 32-bit hash value. Thus, the length of the counters is expected to be smaller or equal to the (2^{32} - 1), since we cannot access elements with indexes above this value.

```
from pdsa.frequency.count_min_sketch import CountSketch

cs = CountSketch(5, 2000)
cs.add("hello")
cs.frequency("hello")
```

3.1.2 Build a sketch

You can build a new sketch either from specifiyng its dimensions (number of counter arrays and their length), or from the expected overestimation diviation and standard error probability.

Build filter from its dimensions

```
from pdsa.frequency.count_min_sketch import CountSketch

cs = CountSketch(num_of_counters=5, length_of_counter=2000)
```

Build filter from the expected errors

In this case the number of counter arrays and their length will be calculated corresponsing to the expected overestimation and the requested error.

```
from pdsa.frequency.count_min_sketch import CountSketch

cs = CountSketch.create_from_expected_error(deviation=0.000001, error=0.01)
```

Note: The *deviation* is the error ϵ in answering the paricular query. For example, if we expect 10^7 elements and allow the fixed overestimate of 10, the deviation is 10/10^7 = 10^{-6}.

The *error* is the standard error δ (0 < error < 1).

Note: The Count–Min Sketch is approximate and probabilistic at the same time, therefore two parameters, the error ϵ in answering the paricular query and the error probability δ, affect the space and time requirements. In fact, it provides the guarantee that the estimation error for frequencies will not exceed ϵ x n with probability at least 1 – δ.

3.1.3 Index element into the sketch

```
cs.add("hello")
```

Note: It is possible to index into the counter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

3.1.4 Estmiate frequency of the element

```
print(cs.frequency("hello"))
```

Warning: It is only an approximation of the exact frequency.

3.1.5 Size of the sketch in bytes

```
print(cs.sizeof())
```

3.1.6 Length of the sketch

```
print(len(cs))
```

3.2 Count-Min Sketch

Count–Min Sketch is a simple space-efficient probabilistic data structure that is used to estimate frequencies of elements in data streams and can address the Heavy hitters problem. It was presented in 2003 [1] by Graham Cormode and Shan Muthukrishnan and published in 2005 [2].

3.2.1 References

[1] Cormode, G., Muthukrishnan, S. What's hot and what's not: Tracking most frequent items dynamically Proceedings of the 22th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, San Diego, California - June 09-11, 2003, pp. 296–306, ACM New York, NY. http://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/CormodeM-hot.pdf

[2] Cormode, G., Muthukrishnan, S. An Improved Data Stream Summary: The Count–Min Sketch and its Applications Journal of Algorithms, Vol. 55 (1), pp. 58–75. http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf

This implementation uses MurmurHash3 family of hash functions which yields a 32-bit hash value. Thus, the length of the counters is expected to be smaller or equal to the (2^{32} - 1), since we cannot access elements with indexes above this value.

```
from pdsa.frequency.count_min_sketch import CountMinSketch

cms = CountMinSketch(5, 2000)
cms.add("hello")
cms.frequency("hello")
```

3.2.2 Build a sketch

You can build a new sketch either from specifiyng its dimensions (number of counter arrays and their length), or from the expected overestimation diviation and standard error probability.

Build filter from its dimensions

```
from pdsa.frequency.count_min_sketch import CountMinSketch

cms = CountMinSketch(num_of_counters=5, length_of_counter=2000)
```

Build filter from the expected errors

In this case the number of counter arrays and their length will be calculated corresponsing to the expected overestimation and the requested error.

```
from pdsa.frequency.count_min_sketch import CountMinSketch

cms = CountMinSketch.create_from_expected_error(deviation=0.000001, error=0.01)
```

Note: The *deviation* is the error ϵ in answering the paricular query. For example, if we expect 10^7 elements and allow the fixed overestimate of 10, the deviation is 10/10^7 = 10^{-6}.

The *error* is the standard error δ (0 < error < 1).

Note: The Count–Min Sketch is approximate and probabilistic at the same time, therefore two parameters, the error ϵ in answering the paricular query and the error probability δ, affect the space and time requirements. In fact, it provides the guarantee that the estimation error for frequencies will not exceed ϵ x n with probability at least 1 – δ.

3.2.3 Index element into the sketch

```
cms.add("hello")
```

Note: It is possible to index into the counter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

3.2.4 Estmiate frequency of the element

```
print(cms.frequency("hello"))
```

Warning: It is only an approximation of the exact frequency.

3.2.5 Size of the sketch in bytes

```
print(cms.sizeof())
```

3.2.6 Length of the sketch

```
print(len(cms))
```

CHAPTER 4

Membership

A Membership problem for a set is the problem to decide whether some element belongs to the set or not.

In many cases while testing some element for an existance in the set you don't need to know exactly which element from the set has been matched, only the fact of such match matters.

4.1 Classical Bloom Filter

This implementation uses bitvector to store the bloom filter array.

```
from pdsa.membership.bloom_filter import BloomFilter

bf = BloomFilter(1000000, 5)
bf.add("hello")
bf.test("hello")
```

4.1.1 Build a filter

You can build a new filter either from specifiyng its length and number of hash functions, or from the expected capacity and error probability.

Build filter from its length and number of hash function

```
from pdsa.membership.bloom_filter import BloomFilter

bf = BloomFilter(100000, 5)
```

Note: Memory for the filter is assigned by chunks, therefore the length of the filter can be rounded up to use it in full.

Build filter from the expected capacity and error probability

In this case length of the filter and number of hash functions will be calculated to handle the requested number of elements with the requested error.

```
from pdsa.membership.bloom_filter import BloomFilter

bf = BloomFilter.create_from_capacity(10000, 0.02)
```

4.1.2 Add element into the filter

```
bf.add("hello")
```

Note: It is possible to add into the filter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

4.1.3 Test if element is in the filter

```
bf.test("hello") == True

"hello" in bf
```

4.1.4 Size of the filter in bytes

```
print(bf.sizeof())
```

4.1.5 Length of the filter

```
print(len(bf))
```

4.1.6 Count of unique elements in the filter

```
print(bf.count())
```

Warning: It is only an approximation, since there is no reliable way to determine the number of unique elements that are already in the filter.

4.2 Counting Bloom Filter

This implementation uses 4-bit counter implementation to store counts of elements and bitvector to store the bloom filter array.

```
from pdsa.membership.counting_bloom_filter import CountingBloomFilter

bf = CountingBloomFilter(1000000, 5)
bf.add("hello")
bf.test("hello")
bf.remove("hello")
```

4.2.1 Build a filter

You can build a new filter either from specifiyng its length and number of hash functions, or from the expected capacity and error probability.

Build filter from its length and number of hash function

```
from pdsa.membership.counting_bloom_filter import CountingBloomFilter

bf = CountingBloomFilter(100000, 5)
```

Note: Memory for the filter is assigned by chunks, therefore the length of the filter can be rounded up to use it in full.

Build filter from the expected capacity and error probability

In this case length of the filter and number of hash functions will be calculated to handle the requested number of elements with the requested error.

```
from pdsa.membership.counting_bloom_filter import CountingBloomFilter

bf = CountingBloomFilter.create_from_capacity(10000, 0.02)
```

4.2.2 Add element into the filter

```
bf.add("hello")
```

Note: It is possible to add into the filter any elements (internally it uses *repr()* of the python object to calculate hash values for elements that are not integers, strings or bytes.

4.2.3 Test if element is in the filter

```
bf.test("hello") == True

"hello" in bf
```

4.2.4 Delete element from the filter

```
bf.remove("hello")
```

Warning: The implementation uses 4-bit counters that freeze at value 15. So, the deletion, in fact, is a probabilistically correct only.

4.2.5 Size of the filter in bytes

```
print(bf.sizeof())
```

4.2.6 Length of the filter

```
print(len(bf))
```

4.2.7 Count of unique elements in the filter

```
print(bf.count())
```

Warning: It is only an approximation, since there is no reliable way to determine the number of unique elements that are already in the filter.

CHAPTER 5

Rank

The most commonly used rank characteristics are quantiles, and their specific types as percentiles and quartiles.

5.1 Random Sampling

The Random sampling algorithm, often referred to as MRL, was published by Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce Lindsay in 1999 and addressed the problem of the correct sampling and quantile estimation. It consists of the non-uniform sampling technique and deterministic quantile finding algorithm.

This implementation of the simpler version of the MRL algorithm that was proposed by Ge Luo, Lu Wang, Ke Yi, and Graham Cormode in 2013, and denoted in the original articles as Random.

```
from pdsa.rank.random_sampling import RandomSampling

rs = RandomSampling(5, 5, 7)
for i in range(100):
    rs.add(random.randrange(0, 8))
rs.compress()

rs.quantile_query(0.5)
rs.inverse_quantile_query(5)
rs.interval_query(2, 6)
```

5.1.1 Build random sampling data structure

RandomSampling is designed to be built by specifying number of buffers, their capacity and the maximal height (depth) of the data structure.

```
from pdsa.rank.random_sampling import RandomSampling

rs = RandomSampling(5, 5, 7)
```

Build random sampling data structure from the expected error probability

In this case length of the number of buffers, their capacity and the maximal height will be calculated to support the requested error.

```
from pdsa.rank.random_sampling import RandomSampling

rs = RandomSampling.create_from_error(0.01)
```

5.1.2 Add element into RandomSampling

```
rs.add(5)
```

5.1.3 Quantile Query

Given a fraction `q` from [0, 1], the quantile query is about to find the value whose rank in a sorted sequence of the `n` values is `q * n`.

```
rs.quantile_query(0.95)
```

5.1.4 Inverse Quantile Query

Given an element, the inverse quantile query is about to find its rank in sorted sequence of values.

```
rs.inverse_quantile_query(4)
```

5.1.5 Interval (range) Query

Given a value the interval (range) query is about to find the number of elements in the given range in the sequence of elements.

```
rs.interval_query(3, 6)
```

5.1.6 Number of buffers in the data structure

The number of buffers allocated in the data structure.

```
print(len(rs))
```

5.1.7 Size of the data structure in bytes

```
print(rs.sizeof())
```

Warning: Since we do not want to calculate exact size, this function return some estimation.

5.1.8 Number of processed elements

```
print(rs.count())
```

5.2 Quantile Digest (q-digest)

Quantile Digest, or q-digest, is a tree-based stream summary algorithm that was proposed by Nisheeth Shrivastava, Subhash Suri et al. in 2004 in the context of monitoring distributed data from sensors.

```
from pdsa.rank.qdigest import QuantileDigest

qd = QuantileDigest(3, 5)
for i in range(100):
    qd.add(random.randrange(0, 8))
qd.compress()

qd.quantile_query(0.5)
qd.inverse_quantile_query(5)
qd.interval_query(2, 6)
```

5.2.1 Build a q-digest

Quantile Digest is designed to be built on integer numbers from a known range.

The range of the supported integers is defined by the number of bytes in thier maximal representation. Thus, for k-bytes integers, the range will be [0, 2^k - 1].

```
from pdsa.rank.qdigest import QuantileDigest

qd = QuantileDigest(3, 5)
```

Note: The ranges up to 32 bytes only are supported in the current implementation.

5.2.2 Add element into q-digest

```
qd.add(5)
```

5.2.3 Quantile Query

Given a fraction `q` from [0, 1], the quantile query is about to find the value whose rank in a sorted sequence of the `n` values is `q * n`.

```
qd.quantile_query(0.95)
```

5.2.4 Inverse Quantile Query

Given an element, the inverse quantile query is about to find its rank in sorted sequence of values.

```
qd.inverse_quantile_query(4)
```

5.2.5 Interval (range) Query

Given a value the interval (range) query is about to find the number of elements in the given range in the sequence of elements.

```
qd.interval_query(3, 6)
```

5.2.6 Merge q-digests

```
qd1.merge(qd2)
```

Warning: Only q-digets with same compression_factor and range are possible to merge correctly.

5.2.7 Length of the q-digest

Length of the q-digest is the number of buckets (nodes) included into the q-digest.

```
print(len(qd))
```

5.2.8 Size of the q-digest in bytes

```
print(qd.sizeof())
```

Warning: Since we can't calculate exact size of a dict in Cython, this function return some estimation based an ideal size of keys, values of each bucket.

5.2.9 Count of elements in the q-digest

```
print(qd.count())
```

Warning: While we can't say exactly which elements are in the q-digest, (because the compression is a lossy operation), it's still possible to say how many in total elements were added.