Reservoir sampling

From Wikipedia, the free encyclopedia

Reservoir sampling is a family of randomized algorithms for randomly choosing a sample of *k* items from a list *S* containing *n* items, where *n* is either a very large or unknown number. Typically *n* is large enough that the list doesn't fit into main memory. The most common example was labelled *Algorithm R* by Jeffrey Vitter in his paper on the subject.[1]

This simple O(*n*) algorithm as described in the *Dictionary of Algorithms and Data Structures*[2] consists of the following steps (assuming that the arrays are one-based, and that the number of items to select, *k*, is smaller than the size of the source array, *S*):

```
array R[k];    // result
integer i, j;

// fill the reservoir array
for each i in 1 to k do
    R[i] := S[i]
done;

// replace elements with gradually decreasing probability
for each i in k+1 to length(S) do
    j := random(1, i);   // important: inclusive range
    if j <= k then
        R[j] := S[i]
    fi
done
```

The algorithm creates a "reservoir" array of size *k* and populates it with the first *k* items of *S*. It then iterates through the remaining elements of *S* until *S* is exhausted. At the *i*th element of *S*, the algorithm generates a random number *j* between 1 and *i*. If *j* is less than or equal to *k*, the *j*th element of the reservoir array is replaced with the *i*th element of *S*. In effect, for all *i*, the *i*th element of *S* is chosen to be included in the reservoir with probability *k/i*. Similarly, at each iteration the *j*th element of the reservoir array is chosen to be replaced with probability *1/k * k/i*, which simplifies to *1/i*. It can be shown that when the algorithm has finished executing, each item in *S* has equal probability (i.e. *k/length(S)*) of being chosen for the reservoir. To see this, consider the following proof by induction. After the (i-1)th round, let us assume, the probability of a number being in the reservoir array is *k/(i-1)*. Since the probability of the number being replaced in the ithround is *1/i*, the probability that it survives the ith round is *(i-1)/i*. Thus, the probability that a given number is in the reservoir after the ith round is the product of these two probabilities, i.e. the probability of being in the reservoir after the (i-1)th round, and surviving replacement in the ith round. This is *(k/(i-1)) * ((i-1)/i)=k/i*. Hence, the result holds for *i*, and is therefore true by induction.

Contents

Relation to Fisher-Yates shuffle

Suppose one wanted to draw *k* random cards from a deck of playing cards (i.e., *n=52*). A natural approach would be to shuffle the deck and then take the top *k* cards. In the general case, the shuffle also needs to work even if the number of cards in the deck is not known in advance, a condition which is satisfied by the inside-out version of the Fisher-Yates shuffle:

```
To initialize an array a of n elements to a randomly shuffled copy of S, both 0-based:
  a[0] ← S[0]
  for i from 1 to n - 1 do
     r ← random (0 .. i)
     a[i] ← a[r]
     a[r] ← S[i]
```

Note that although the rest of the cards are shuffled, only the top *k* are important in the present context. Therefore, the array *a* need only track the cards in the top *k* positions while performing the shuffle, reducing the amount of memory needed. Truncating *a* to length *k*, the algorithm is modified accordingly:

```
To initialize an array a to k random elements of S (which is of length n), both 0-based:
  a[0] ← S[0]
  for i from 1 to k - 1 do
     r ← random (0 .. i)
     a[i] ← a[r]

     a[r] ← S[i]

  for i from k to n - 1 do
     r ← random (0 .. i)
     if (r < k) then a[r] ← S[i]
```

Since the order of the first *k* cards is immaterial, the first loop can be removed and *a* can be initialized to be the first *k* items of *S*. This yields *Algorithm R*.

Example implementation

The following is a simple implementation of the algorithm in Python that samples the set of English Wikipedia page titles:

```python
import random
SAMPLE_COUNT = 10

# Force the value of the seed so the results are repeatable
random.seed(12345)
```

```
sample_titles = []
for index, line in enumerate(open("enwiki-20091103-all-titles-in-ns0")):
        # Generate the reservoir
        if index < SAMPLE_COUNT:
                sample_titles.append(line)
        else:
                # Randomly replace elements in the reservoir
                # with a decreasing probability.
                # Choose an integer between 0 and index (inclusive)
                r = random.randint(0, index)
                if r < SAMPLE_COUNT:
                        sample_titles[r] = line
print sample_titles
```

Statistical properties

Probabilities of selection of the reservoir methods are discussed in Chao (1982)[3] and Tille (2006).[4] While the first-order selection probabilities are equal to k/n (or, in case of Chao's procedure, to an arbitrary set of unequal probabilities), the second order selection probabilities depend on the order in which the records are sorted in the original reservoir. The problem is overcome by the cube sampling method of Deville and Tille (2004).[5]

Limitations

Reservoir sampling makes the assumption that the desired sample fits into main memory, often implying that k is a constant independent of n. In applications where we would like to select a large subset of the input list (say a third, i.e. $k=n/3$), other methods need to be adopted. Distributed implementations for this problem have been proposed.[6]

See also

- Moving average
- Fisher-Yates shuffle

References

1. Vitter, Jeffrey S. (1 March 1985). "Random sampling with a reservoir" (http://www.cs.umd.edu/~samir/498/vitter.pdf). *ACM Transactions on Mathematical Software* **11** (1): 37–57. doi:10.1145/3147.3165 (https://dx.doi.org/10.1145%2F3147.3165).
2. Dictionary of Algorithms and Data Structures (http://xlinux.nist.gov/dads/HTML/reservoirSampling.html)
3. Chao, M.T. (1982) A general purpose unequal probability sampling plan. Biometrika, 69 (3): 653-656. (http://biomet.oxfordjournals.org/content/69/3/653.abstract)
4. Tille, Y. (2006). Sampling Algorithms. Springer (http://www.amazon.com/Sampling-Algorithms-Springer-Series-Statistics/dp/0387308148)
5. Deville, J.-C., and Y. Tille (2004). Efficient balanced sampling: The cube method. Biometrika 91 (4): 893-912. (http://biomet.oxfordjournals.org/content/91/4/893.short)
6. Reservoir Sampling in MapReduce (http://had00b.blogspot.com/2013/07/random-subset-in-mapreduce.html)

Categories: Algorithms | Analysis of algorithms | Probabilistic complexity theory