Reservoir sampling

Contents [hide]
(Top)
Motivation
Simple: Algorithm R
Optimal: Algorithm L
With random sort
Weighted random sampling
Algorithm A-Res
Algorithm A-ExpJ
Algorithm A-Chao
Algorithm A-Chao with Jumps
Relation to Fisher–Yates shuffle
Statistical properties
Limitations
References

Reservoir sampling is a family of randomized algorithms for choosing a simple random sample, without replacement, of k items from a population of unknown size in a single pass over the items. The size of the population is not known to the algorithm and is typically too large for all of it to fit into main memory. The population is revealed to the algorithm over time, and the algorithm cannot look back at previous items. At any point, the current state of the algorithm must permit extraction of a simple random sample without replacement of size k over the part of the population seen so far.

Motivation [edit]

Suppose we see a sequence of items, one at a time. We want to keep ten items in memory, and we want them to be selected at random from the sequence. If we know the total number of items n and can access the items arbitrarily, then the solution is easy: select 10 distinct indices j between 1 and n with equal probability, and keep the i-th elements. The problem is that we do not always know the exact n in advance.

Simple: Algorithm R [edit]

A simple and popular but slow algorithm, Algorithm R, was created by Jeffrey Vitter.[1]

> Initialize an array R indexed from 1 to k, containing the first k items of the input $x_1, \ldots, x_k$. This is the reservoir:
>
> For each new input x_i, generate a random number j uniformly in $\{1, \ldots, i\}$. If $j \in \{1, \ldots, k\}$, then set $R[j] := x_i$; otherwise, discard x_i.
>
> Return R after all inputs are processed.

This algorithm works by induction on $i \geq k$:

> ***Proof***
>
> When $i = k$, Algorithm R returns all inputs, thus providing the basis for a proof by mathematical induction.
>
> Here, the induction hypothesis is that the probability that a particular input is included in the reservoir just before the $(i+1)$-th input is processed is $\frac{k}{i}$ and we must show that the probability that a particular input is included in the reservoir is $\frac{k}{i+1}$ just after the $(i+1)$-th input is processed. Apply Algorithm R to the $(i+1)$-th input. Input x_{i+1} is included with probability $\frac{k}{i+1}$ by definition of the algorithm. For any other $j \in \{k+1, \ldots, i+1\}$, by the induction hypothesis, the probability that the input $x_j \in \{x_1, \ldots, x_i\}$ is included in the reservoir just before the $(i+1)$-th input is processed is $\frac{k}{i}$. The probability that it is still included in the reservoir after x_{i+1} is processed (in other words, that x_j is not replaced by x_{i+1}) is $\frac{k}{i} \times \left(1 - \frac{1}{i+1}\right) = \frac{k}{i} \times \frac{i}{i+1} = \frac{k}{i+1}$. The latter result follows from the assumption that the integer j is generated uniformly at random; once it becomes clear that a replacement will in fact occur, the probability that x_i in particular is replaced by x_{i+1} is $\frac{k}{i+1} = \frac{1}{i+1}$. We have shown that the probability that a new input enters the reservoir is equal to the probability that an existing input in the reservoir is retained. Therefore, we conclude by the principle of mathematical induction that Algorithm R does indeed produce a random sample of the inputs.

While conceptually simple and easy to understand, this algorithm needs to generate a random number for each item of the input, including the items that are discarded. The algorithm's asymptotic running time is thus $O(n)$. Generating this amount of randomness and the linear run time causes the algorithm to be unnecessarily slow if the input population is large.

This is Algorithm R, implemented as follows:

```
(* S has items to sample, R will contain the result *)
ReservoirSample(S[1..n], R[1..k])
  // fill the reservoir array
  for i := 1 to k
      R[i] := S[i]

  // replace elements with gradually decreasing probability
  for i := k+1 to n
    (* randomInteger(a, b) generates a uniform integer from the inclusive range {a, ..., b} *)
    j := randomInteger(1, i)
    if j <= k
        R[j] := S[i]
```

Optimal: Algorithm L [edit]

If we generate its random numbers $u_1, \ldots, u_n \sim U[0,1]$ independently, then the indices of the smallest k of them is a uniform sample of the k-subsets of $\{1, \ldots, n\}$.

The process can be done without knowing n:

> Keep the smallest k of $u_1, \ldots, u_i$ that has been seen so far, as well as u_k, the index of the largest among them. For each new u_{k+1}, compare it with u_k. If $u_{k+1} < u_k$, then discard u_k, store u_{k+1}, and set u_{k+1} to be the index of the largest among them. Otherwise, discard u_{k+1}, and set $u_{k+1} := u_k$.

Now couple this with the stream of inputs $x_1, \ldots, x_n$. Every time some u_i is accepted, store the corresponding x_i. Every time some u_i is discarded, discard the corresponding x_i.

This algorithm still needs $O(n)$ random numbers, thus taking $O(n)$ time. But it can be simplified.

First simplification: it is unnecessary to test new $u_{k+1}, u_{k+2}, \ldots$ one by one, since the probability that the next acceptance happens at u_{k+i} is $(1-u_k)^{i-1} \cdot u_k = (1-u_k)^i$, that is, the interval L of acceptance follows a geometric distribution.

Second simplification: it's unnecessary to remember the entire array of the smallest k of $u_1, \ldots, u_k$ that has been seen so far, but merely u_k the largest among them. This is based on three observations:

- Every time some new x_{k+1} is selected to be entered into storage, a uniformly random entry in storage is discarded.
- $u_{k+1} \sim U[0, u]$
- u_{k+1} has the same distribution as $\max\{u_1, \ldots, u_k\}$, where all $u_1, \ldots, u_k \sim U[0, u]$ independently. This can be sampled by first sampling $u \sim U[0,1]$, then taking $u^{\frac{1}{k}}$.

This is Algorithm L,[2] which is implemented as follows:

```
(* S has items to sample, R will contain the result *)
ReservoirSample(S[1..n], R[1..k])
  // fill the reservoir array
  for i := 1 to k
      R[i] := S[i]

  (* random() generates a uniform (0,1) random number *)
  W := exp(log(random())/k)

  while i <= n
      i := i + floor(log(random())/log(1-W)) + 1
      if i <= n
        (* replace a random item of the reservoir with item i *)
        R[randomInteger(1,k)] := S[i]  // random index between 1 and k, inclusive
        W := W * exp(log(random())/k)
```

This algorithm computes three random numbers for each item that becomes part of the reservoir, and does not spend any time on items that do not. Its expected running time is thus $O(k(1 + \log(n/k)))$[2] which is optimal.[2] At the same time, it is simple to implement efficiently and does not depend on random deviates from exotic or hard-to-compute distributions.

With random sort [edit]

If we associate with each item of the input a uniformly generated random number, the k items with the largest (or, equivalently, smallest) associated values form a simple random sample.[3] A simple reservoir-sampling thus maintains the k items with the currently largest associated values in a priority queue.

```
(*
  S is a stream of items to sample
  S.Current returns current item in stream
  S.Next advances stream to next position
  min-priority queue supports:
    Count --> number of items in priority queue
    Minimum --> returns minimum key value of all items
    Extract-Min() --> Remove the item with minimum key
    Insert(key, Item) --> Adds item with specified key
*)
ReservoirSample(S[1..?])
  H := new min-priority-queue
  while S has data
    r := random()  // uniformly random between 0 and 1, exclusive
    if H.Count < k
      H.Insert(r, S.Current)
    else
      // keep k items with largest associated keys
      if r > H.Minimum
        H.Extract-Min()
        H.Insert(r, S.Current)
  S.Next
  return items in H
```

The expected running time of this algorithm is $O(n + k \log k \log(n/k))$ and it is relevant mainly because it can easily be extended to items with weights.

Weighted random sampling [edit]

This method, also called sequential sampling, is incorrect in the sense that it does not allow to obtain a priori fixed inclusion probabilities. Some applications require items' sampling probabilities to be according to weights associated with each item. For example, it might be required to sample queries in a search engine with weight as number of times they were performed so that the sample can be analyzed for overall impact on user experience. Let the weight of item i be w_i, and the sum of all weights be W. There are two ways to interpret weights assigned to each item in the set:[4]

1. In each round, the probability of every unselected item to be selected in that round is proportional to its weight relative to the weights of all unselected items. If X is the current sample, then the probability of an item $i \notin X$ to be selected in the current round is $w_i / (W - \sum_{j \in X} w_j)$.
2. The probability of each item to be included in the random sample is not proportional to its relative weight, i.e., w_i / W. A simple counter-example is the case, e.g., $k = n$.

Algorithm A-Res [edit]

The following algorithm was given by Efraimidis and Spirakis that uses interpretation 1:[5]

```
(*
  S is a stream of items to sample
  S.Current returns current item in stream
  S.Weight returns weight of current item in stream
  S.Next advances stream to next position
  The power operator is represented by '^'
  min-priority-queue supports:
    Count --> number of items in the priority queue
    Minimum --> returns minimum key of any item in the priority queue
    Extract-Min() --> Remove the item with minimum key
    Insert(key, Item) --> Adds item with specified key
*)
ReservoirSample(S[1..?])
  H := new min-priority-queue
  while S has data
    r := random() ^ (1/S.Weight)   // random() produces a uniformly random number in (0,1)
    if H.Count < k
      H.Insert(r, S.Current)
    else
      // keep k items with largest associated keys
      if r > H.Minimum
        H.Extract-Min()
        H.Insert(r, S.Current)
  S.Next
  return items in H
```

This algorithm is identical to the algorithm given in Reservoir Sampling with Random Sort except for the generation of the items' keys. The algorithm is equivalent to assigning each item a key r^{1/w_i} where r is the random number and then selecting the k items with the largest keys. Equivalently, a more numerically stable formulation of this algorithm computes the keys as $-\ln(r)/w_i$ and select the k items with the smallest keys.[6] [dubious? verification]

Algorithm A-ExpJ [edit]

The following algorithm is a more efficient version of A-Res, also given by Efraimidis and Spirakis.[5]

```
(*
  S is a stream of items to sample
  S.Current returns current item in stream
  S.Weight returns weight of current item in stream
  S.Next advances stream to next position
  The power operator is represented by '^'
  min-priority-queue supports:
    Count --> number of items in the priority queue
    Minimum --> minimum key of any item in the priority queue
    Extract-Min() --> Remove the item with minimum key
    Insert(key, Item) --> Adds item with specified key
*)
ReservoirSampleWithJumps(S[1..?])
  H := new min-priority-queue
  while S has data and H.Count < k
    r := random() ^ (1/S.Weight)   // random() produces a uniformly random number in (0,1)
    H.Insert(r, S.Current)
    S.Next
  X := log(random()) / log(H.Minimum)  // this is the amount of weight that needs to be jumped over
  while S has data
    X := X - S.Weight
    if X <= 0
      t := H.Minimum ^ S.Weight
      r := random(t, 1) ^ (1/S.Weight) // random(x, y) produces a uniformly random number in (x, y)

      H.Extract-Min()
      H.Insert(r, S.Current)

      X := log(random()) / log(H.Minimum)
    S.Next
  return items in H
```

This algorithm follows the same mathematical properties that are used in A-Res, but instead of calculating the key for each item and checking whether that item should be inserted or not, it calculates an exponential jump to the next item which will be inserted. This avoids having to create random variates for each item, which may be expensive. The number of random variates required is reduced from $O(n)$ to $O(k \log(n/k))$ in expectation, where k is the reservoir size, and n is the number of items in the stream.[5]

Algorithm A-Chao [edit]

Following algorithm was given by M. T. Chao uses interpretation 2:[7]

```
(*
  S has items to sample, R will contain the result
  S[i].Weight contains weight for each item
*)
WeightedReservoir-Chao(S[1..n], R[1..k])
  WSum := 0
  // fill the reservoir array
  for i := 1 to k
      R[i] := S[i]
      WSum := WSum + S[i].Weight
  for i := k+1 to n
      WSum := WSum + S[i].Weight
      p := S[i].Weight / WSum // probability for this item
      j := random()           // uniformly random between 0 and 1
      if j <= p               // select item according to probability
          R[randomInteger(1,k)] := S[i]  //uniform selection in reservoir for replacement
```

For each item, its relative weight is calculated and used to randomly decide if the item will be added into the reservoir. If the item is selected, then one of the existing items of the reservoir is uniformly selected and replaced with the new item. The trick here is that, if the probabilities of all items in the reservoir are already proportional to their weights, then by selecting uniformly which item to replace, the probabilities of all items remain proportional to their weight after the replacement.

Note that Chao doesn't specify how to sample the first k elements. He simple assumes we have some other way of picking them in proportion to their weight. Chao: "Assume that we have a sampling plan of fixed size with respect to X, a set time A, such that its first-order inclusion probability of X_i is $\eta k \cdot g_i$".

Algorithm A-Chao with Jumps [edit]

Similar to the other algorithms, it is possible to compute a random weight. j and subtract items' probability mass values, skipping them while $j > 0$, reducing the number of random numbers that have to be generated.[7]

```
(*
  S has items to sample, R will contain the result
  S[i].Weight contains weight for each item
*)
WeightedReservoir-Chao(S[1..n], R[1..k])
  WSum := 0
  // fill the reservoir array
  for i := 1 to k
      R[i] := S[i]
      WSum := WSum + S[i].Weight
  j := randomExp() // uniformly random between 0 and 1
  for i := k+1 to n
      WSum := WSum + S[i].Weight
      p := S[i].Weight / WSum // probability for this item
      j := j - p
      if j <= 0
          R[randomInteger(1,k)] := S[i]  //uniform selection in reservoir for replacement
          j := randomExp() * WSum
          pNone := 1
      pNone := pNone * (1 - p)
      j := j - pNone
```

Relation to Fisher–Yates shuffle [edit]

Suppose one wanted to draw k random cards from a deck of cards. 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:[8]

```
(* S has the input, R will contain the output permutation *)
Shuffle(S[1..n], R[1..n])
  R[1] := S[1]
  for i from 2 to n do
      j := randomInteger(1, i)  // inclusive range
      R[i] := R[j]
      R[j] := S[i]
```

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

```
(* S has items to sample, R will contain the result *)
ReservoirSample(S[1..n], R[1..k])
  R[1] := S[1]
  for i from 2 to k do
      j := randomInteger(1, i)  // inclusive range
      R[i] := R[j]
      R[j] := S[i]
  for i from k + 1 to n do
      j := randomInteger(1, i)  // inclusive range
      if j <= k
          R[j] := S[i]
```

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

Statistical properties [edit]

Probabilities of selection of the reservoir methods are discussed in Chao (1982)[7] and Tillé (2006).[9] While the first-order selection probabilities are equal to k/N for all cases, 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 Tillé (2004).[10]

Limitations [edit]

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.[11]

References [edit]

1. ^ ^ Vitter, Jeffrey S. (1 March 1985). "Random sampling with a reservoir" (PDF). *ACM Transactions on Mathematical Software*. **11** (1): 37–57. CiteSeerX 10.1.1.138.784 ; doi:10.1145/3147.3165 ; S2CID 1198767 .
2. ^ ^ Li, Kim-Hung (4 December 1994). "Reservoir-Sampling Algorithms of Time Complexity $O(n(1+\log(N)))$". *ACM Transactions on Mathematical Software*. **20** (4): 481–493. doi:10.1145/198429.198435 ; S2CID 10370342 .
3. ^ Fan, C.; Muller, M.E.; Rezucha, I. (1962). "Development of sampling plans by using sequential (item by item) selection techniques and digital computers". *Journal of the American Statistical Association*. **57** (298): 387–402. doi:10.1080/01621459.1962.10480647 ; JSTOR 2282647 .
4. ^ ^ Efraimidis, Pavlos S. (2015). "Weighted Random Sampling over Data Streams". *Algorithms, Probability, Networks, and Games*. Lecture Notes in Computer Science. **9295**: 183–195. arXiv:1012.0256 ; doi:10.1007/978-3-319-24024-4_12 ; ISBN 978-3-319-24023-7 ; S2CID 2508731 .
5. ^ ^ Efraimidis, Pavlos S.; Spirakis, Paul G. (2006-03-16). "Weighted random sampling with a reservoir". *Information Processing Letters*. **97** (5): 181–185. doi:10.1016/j.ipl.2005.11.003 .
6. ^ Arratia, Richard (2002). Ilwis Bolotova (ed.). "On the amount of dependence in the prime factorization of a uniform random integer". *Contemporary Combinatorics*. **10**: 29–91. CiteSeerX 10.1.1.140.3070 ; ISBN 978-3-540-42760-0 .
7. ^ ^ Chao, M. T. (1982). "A general purpose unequal probability sampling plan". *Biometrika*. **69** (3): 653–656. doi:10.1093/biomet/69.3.653 .
8. ^ National Research Council (2013). *Frontiers in Massive Data Analysis*. The National Academies Press. p. 121. ISBN 978-0-309-28781-8 .
9. ^ Tillé, Yves (2006). *Sampling Algorithms*. Springer. ISBN 978-0-387-30814-2 .
10. ^ Deville, Jean-Claude; Tillé, Yves (2004). "Efficient balanced sampling: The cube method" (PDF). *Biometrika*. **91** (4): 893–912. doi:10.1093/biomet/91.4.893 .
11. ^ Reservoir Sampling in MapReduce

Categories: Algorithms | Analysis of algorithms | Randomized algorithms

This page was last edited on 29 December 2022, at 10:10 (UTC).
Text is available under the Creative Commons Attribution-ShareAlike License 3.0; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
Privacy policy About Wikipedia Disclaimers Contact Wikipedia Mobile view Developers Statistics Cookie statement
Wikimedia Foundation Powered by MediaWiki