Sampling with a reservoir

*paper by J.S. Vitter*

Presented by

Shantanu Joshi
What is the problem?

• Sampling from a population whose size (N) is unknown.
  
  e.g sampling from a tape of indeterminate length
  or sampling the results of a sequence of relational operators.

• Cannot apply conventional sampling methods since N is unknown.

• Moreover doing 1 pass on the file just to find out the size of the population may be too expensive!

One solution – Reservoir Sampling
What is a reservoir?

• It is a storage space where we maintain a random sample of the desired size and keep modifying the sample as we see more and more records of the population.

• The reservoir would be smaller than the population size (N).
Algorithm R

- Insert the first n records of the population in the reservoir.
- Every subsequent record, say the (t+1)\textsuperscript{st} record has the probability \( \frac{n}{t+1} \) of being included in the reservoir.
- If (included) {
  Choose a record to evict, randomly
  Replace that record with the current record
}
- Else {
  Skip over the current record and move on to the next record.
}
Algorithm R (contd.)

- If sample size is bigger than memory, store them on disk and store an array of pointers to these records in memory.

- Apply the same algorithm with the modification that the records are copied on disk and the in-memory pointers are modified.

- *Execution Time*
  
  Algorithm R runs in time $O(N)$

*Reason* - It has to look at every record of the population.
Scope for improvement

• A lower bound for reservoir algorithms is $O(n + \ln(N/n))$
  ⇒ There is scope for improvement of algorithm R.

*Idea:*

Algorithm R makes a call to the `random()` function for each record it comes across in the population till it reaches the end of the file.

Can reduce the number of calls to `random()`, we can get a better time complexity by skipping over some records between two successive calls to `random()`
General Framework

• Need an efficient way to calculate the number of records to be skipped before we call random( ) again.

• This depends on the desired sample size as well as the number of records we’ve seen so far.

Can define $S(n, t)$ as a discrete random variable to denote this.
General Framework

• Number of calls to random( ) reduced, but increases the complexity of generating the DRV, S.

Can draw an analogy between S and coin flipping as foll:

Event of getting a tail(T) – we skip the current record
Event of getting a head(H) – we call random( ) and to include the current record in the reservoir or not.

\[ P(H) = \frac{n}{t+1} \quad \text{and} \quad P(T) = 1 - P(H) \]
\[ = \frac{(t+1-n)}{(t+1)} \]

For \( P[S=s] \), we should have s tails followed by 1 head.

\[ P[H] = \frac{n}{t+s+1} \] since we would have seen \((t+s+1)\) records at that time.
Probability and Distribution functions

The formula for the probability function works out to be

\[ f(s) = \frac{n}{t + s + 1} \frac{t^n}{(t + s)^n} = \frac{n}{t - n} \frac{(t - n)^{s + 1}}{(t + 1)^{s + 1}} \]

The probability distribution function, \( F(s) = P[S \leq s] \) can be expressed as

\[ F(s) = \sum_{x \leq s} [f(x)] = 1 - \frac{t^n}{(t + s + 1)^n} = 1 - \frac{(t + 1 - n)^{s + 1}}{(t + 1)^{s + 1}} \]

Expected value of \( S \), \( E[S] = \frac{t - n + 1}{n - 1} \)
• The value of $S(n, t)$ can be found for all values of $t$ from the distribution function, $F(s)$ and a uniform random variable $U$.

• Pick a random value between 0 and 1 and find the minimum value, $s_{\text{min}}$ such that $F(s_{\text{min}})$ is equal to the randomly generated value.

Algorithm X suggests using a sequential scanning method for finding the value of $s_{\text{min}}$. This gives an algorithm with a running time of $O(N)$, but with much fewer calls to random().
Algorithm Y

• Algorithm Y improves upon X by providing a faster way of searching for the min value of s such that F(s) is satisfied.

Newton’s interpolation method is used to find the min value of s

\[
\frac{(t + 1 - n)^{s+1}}{(t + 1)^{s+1}} \leq V \quad \& \quad H(s) = V \text{ (approx.)}
\]

\[
\Delta H(s) = H(s+1) = H(s) = -f(s+1)
\]

This method converges to give a running time value of

\[
O \left( n^2 \left( 1 + \log \frac{N}{n} \log \log \frac{N}{n} \right) \right)
\]
Newton’s method
Algorithm Z

- **Idea** - Attempt to generate $S(n, t)$ in constant time

- **Approach** – Generate a fast approximation of $S$ and then correct it so that its distribution is the same as the distribution, $F(s)$.

- **How does it work?**
  Assume a fast approximate distribution function, $G(x)$ and sample from it using
  
  $$G(x) - U = 0$$
  
  and then correct that sample.

  Need a distribution function whose inverse is easy to calculate
Preliminaries

• Preliminaries for the algorithm
  – Choose a CRV, X with p.d.f. g(x) such that
    • It can be generated quickly
    • Its distribution function approximates F(s) well

  – Choose a constant c such that
    • f(x) ≤ c.g(x) ; x≥0, c≥ 1
Algorithm Z

1. Algorithm Z{
2. Generate a random sample in the unit interval, say U
3. Quickly generate an independent RV, say X with p.d.f g(x)
4. \( \frac{f(x)}{c \cdot g(x)} \)
5. Reject X
6. Goto 2
9. }else {
10. Accept (X)
11. S = X
12. }

It can be proven that the S generated by the above method has the same distribution as F(s).
Cost considerations

• Costly to compute \( \frac{f(x)}{c \cdot g(x)} \) since \( f(x) \) is actually the probability density function of \( S \).

• Can avoid computation of \( f(x) \) most of the times by rather generating an approximate function, \( h(s) \) which can be generated quickly

\[
h(s) \leq f(s); \quad s \geq 0
\]

• \( f(s) \) would need to be computed only if \( U > \frac{h(x)}{c \cdot g(x)} \) and this occurs with low probability.

• Need to consider a constant \( T \) such that present algorithm is slower than Algorithm X for \( t \leq T.n \).
Actual algorithm

if (t <= T.n)
{
    Use Algorithm X
}
else
{
    Use Algorithm Z
}
Choice of approximation parameters

\[ h(s) = \frac{n}{t + 1} \left( \frac{t - n + 1}{t + s - n + 1} \right)^{n+1}, \quad x \geq 0 \]

\[ c = \frac{t + 1}{t - n + 1} \]

\[ g(x) = \frac{n}{t + x} \left( \frac{t}{t + x} \right)^n \]

\[ G(x) = \text{Prob}[\mathcal{X} \leq x] = \int_0^x g(x) \, dx = 1 - \left( \frac{t}{t + x} \right)^n. \]
Why does it work?

• Need a way to solve the equation
  \[ G(x) - U = 0 \]

Inverse of the distribution function, \( G(x) \) can be calculated easily as

\[
G^{-1}(y) = t((1 - y)^{-1/n} - t).
\]

Moreover functions like \( h(s) \) and the constant \( c \), can also be computed quickly.
Time and Cost

• Number of calls to \( \text{RANDOM}() \)

\[
\text{RAND}(n, t, N) \leq n \left( \frac{2(n + 1)}{t - n - 1} + 3(H_N - H_t) \right) \quad \text{... by induction}
\]

• Time taken by the algorithm

\[
\text{TIME}(n, t, N) = O\left(n \left( 1 + \log \frac{N}{t} \right) \right)
\]

This algorithm hits the lower bound for reservoir algorithms.