Learning How to Draw Random Samples in R for Statistical Analysis

Authored by

Mohammed looti

November 7, 2025

In the realm of statistical analysis and large-scale data simulation, the practice of drawing a **random sample** is indispensable. When utilizing the powerful R programming environment, this procedure allows researchers to work efficiently with massive datasets while ensuring that the selected subset--the sample--is representative of the entire population. The principle is simple yet critical: every observation must have an equal probability of selection, thereby effectively mitigating selection bias, which is essential for achieving a valid simple random sample.

The central tool for performing this crucial task in R is the built-in **sample()** function. This function is exceptionally flexible, enabling users to perform sophisticated random selections from various data structures, including standard vectors and sequences of integers, all controlled by precise functional parameters.

Understanding the sample() Function in R

The **sample()** function offers a concise and robust method for executing random selection operations within R. Its syntax is designed for clarity, allowing the user complete control over critical factors such as the output size, the allowance of duplicates (replacement), and even the application of custom probability distributions to weight the selection process.

The general structure, which defines its key operational arguments, is presented below:

sample(x, size, replace = FALSE, prob = NULL)

A deep understanding of each argument is vital, as they collectively determine the characteristics and validity of the resulting random sample:

x: This argument represents the **population** from which elements are drawn. It can be a vector of elements or a single positive integer, in which case R automatically samples from the sequence ranging from `1` to the value of `x`.
size: An integer specifying the required **sample size**. This determines the exact number of elements that will be selected and returned from the population `x`.
replace: A crucial logical parameter (**TRUE** or **FALSE**) that dictates whether sampling is done **with replacement** or **without replacement**. The default value is **FALSE**, ensuring that elements, once selected, are removed from the pool and cannot be chosen again.
prob: An optional numeric vector containing probability weights. If this argument is supplied, the selection process will be weighted accordingly, meaning certain elements are more likely to be chosen than others. If omitted, the selection operates on a uniform chance basis (default is **NULL**).

This tutorial will guide you through the effective application of **sample()**, illustrating detailed examples for both one-dimensional vectors and multi-dimensional data frames, covering the most common scenarios encountered in data manipulation.

Statistical Foundations of Sampling Methods

Before moving to code implementation, it is imperative to solidify the statistical distinction between the two fundamental modes of random selection: sampling **with replacement** and sampling **without replacement**. These two mechanisms fundamentally influence how the source data pool is managed during selection and subsequently impact the statistical properties, such as the variance, of the final sample.

Sampling **without replacement** is the default mechanism for the **sample()** function (`replace = FALSE`). Under this condition, once an item is chosen, it is permanently excluded from the pool of available data. This method is mathematically appropriate when analyzing populations that are considered finite, or when the physical selection of the same unit multiple times is nonsensical or redundant. It guarantees that all elements in the resulting sample are unique.

Conversely, sampling **with replacement** (achieved by setting `replace = TRUE`) permits elements to be selected repeatedly. This technique is extensively used in advanced statistical methodologies like the bootstrap method, where resampling is central to estimating population parameters. It is also necessary when simulating processes where the selection of an item does not reduce the overall supply or probability distribution for subsequent draws. Understanding when to apply `replace = TRUE` is critical for accurate simulation and statistical inference.

Practical Application 1: Sampling from a Vector

The most straightforward use case for the **sample()** function involves drawing observations from a one-dimensional structure, known in R as a **vector**. We will now demonstrate both selection modalities--with and without replacement--using a simple, numerically defined vector.

Our initial example illustrates how to extract a sample of five elements from our source vector **without replacement**. Because we intentionally omit the `replace` argument, R automatically uses the default value of `FALSE`. This guarantees that the five selected values are distinct and unique, reflecting standard finite population sampling techniques.

#create vector of data
data <- c(1, 3, 5, 6, 7, 8, 10, 11, 12, 14)

#select random sample of 5 elements without replacement
sample(x=data, size=5)

10 12 5 14 7

It is important to remember that due to the intrinsic nature of the pseudo-random number

generator, the exact sequence of numbers shown in the output above will vary each time the code is executed. This variability is the hallmark of true random selection.

To shift to sampling **with replacement**, we must explicitly set the `replace` parameter to `TRUE` within the function call. This technical modification allows for the possibility of duplicate values appearing in the final result, a feature essential for many resampling and simulation studies.

#create vector of data
data <- c(1, 3, 5, 6, 7, 8, 10, 11, 12, 14)

#select random sample of 5 elements with replacement
sample(x=data, size=5, replace=TRUE)

12 1 1 6 14

In this second result, the value '1' appears twice. This confirms that after its initial selection, the element was effectively "replaced" back into the data pool, making it available for subsequent selection draws.

Practical Application 2: Sampling Rows from a Data Frame

While sampling single elements from a vector is instructional, most real-world data science tasks involve extracting complete observations (or rows) from a data frame. This capability is paramount for procedures such as partitioning data into training and testing sets for machine learning models, or for conducting cross-validation exercises.

To successfully sample full rows, we ingeniously combine the **sample()** function with standard R subsetting. The **sample()** function is tasked with generating a set of random row indices. We use the `nrow()` function to dynamically determine the total number of observations, ensuring the random index selection aligns perfectly with the size of the data frame.

The code block below demonstrates the creation of a sample data frame and the subsequent random selection of three complete rows:

#create data frame
df <- data.frame(x=c(3, 5, 6, 6, 8, 12, 14),
y=c(12, 6, 4, 23, 25, 8, 9),
z=c(2, 7, 8, 8, 15, 17, 29))

#view data frame
df

x y z
1 3 12 2
2 5 6 7
3 6 4 8
4 6 23 8
5 8 25 15
6 12 8 17
7 14 9 29

#select random sample of three rows from data frame
rand_df <- df

#display randomly selected rows
rand_df

x y z
4 6 23 8
7 14 9 29
1 3 12 2

The mechanism behind this row selection is achieved through three coordinated steps using R's powerful subsetting syntax, `df`:

The standard R subsetting notation, `df`, is initiated, where the first parameter specifies the indices to be selected.

The function `nrow(df)` dynamically calculates the total count of rows in the data frame (7 in this example), establishing the complete range of possible indices (1 through 7).

The **sample()** function then selects three random index values from this range. By intentionally leaving the column parameter blank (the space after the comma), the operation implicitly selects *all* columns corresponding to the randomly chosen row indices, yielding a new subset data frame composed of three complete and unbiased observations.

Ensuring Reproducibility with set.seed()

A core challenge when dealing with functions dependent on pseudo-random number generation, such as **sample()**, is the lack of consistency: running the same code twice typically produces different results. In professional statistical research and data science, reproducibility is not optional; it is a fundamental requirement for validating findings, ensuring transparency, and allowing peers to verify analytical results.

To guarantee that a specific random sample remains identical across multiple executions, regardless of the user or the computing environment, we must initialize the random number generator using the essential **set.seed()** function. The integer passed to this function (e.g., 23 in our example) acts as the starting point, or 'seed,' for the generation sequence.

When **set.seed()** is executed immediately preceding the **sample()** call, the sequence of random numbers generated will be identical every time. This practice effectively "locks in" the sampling process, transforming a volatile random process into a consistent, verifiable one. This step is non-negotiable for any analysis intended for publication or collaborative sharing.

We now apply this vital function to our previous data frame example to demonstrate how the resulting sample is fixed:

```
#make this example reproducible
set.seed(23)

#create data frame
df <- data.frame(x=c(3, 5, 6, 6, 8, 12, 14),
y=c(12, 6, 4, 23, 25, 8, 9),
z=c(2, 7, 8, 8, 15, 17, 29))

#select random sample of three rows from data frame
rand_df <- df

#display randomly selected rows
rand_df
```

```
x y z
5 8 25 15
2 5 6 7
6 12 8 17
```

If you execute this revised code block repeatedly, the output will consistently show rows 5, 2, and 6 from the original data frame, proving that the sample selection is now fully **reproducible**.

Further Exploration of Sampling Techniques

While the **sample()** function facilitates simple random sampling--the foundation of all selection methods--it is important to recognize that complex research often necessitates more nuanced strategies. Depending on the inherent structure of the population and the specific objectives of the study, you may need to employ advanced techniques like stratified, systematic, or cluster

sampling.

These specialized methods are designed to ensure that the resulting sample precisely reflects specific demographic subgroups or spatial distributions within the larger population, thereby enhancing the overall validity and precision of the statistical inference. For those looking to delve deeper into these specialized selection processes within the R environment, the following resources provide comprehensive guidance and examples:

Stratified Sampling in R (With Examples)
Systematic Sampling in R (With Examples)
Cluster Sampling in R (With Examples)