Learning to Select Random Rows in R with dplyr

Authored by

Mohammed looti

October 30, 2025

Introduction to Random Sampling in Data Science

Selecting a random subset of observations, often referred to as sampling, is a fundamental operation in data science and statistical analysis. Whether you are creating training and testing sets for machine learning models, performing quick exploratory data analysis on massive datasets, or simply ensuring the results of an operation are not biased by data order, the ability to extract a truly random sample is essential. The R programming environment provides several robust tools for this purpose, but none are as streamlined and intuitive as those found within the dplyr package. This tutorial focuses specifically on two powerful functions from **dplyr** that allow users to select random rows from a data frame efficiently: **`sample_n()`** and **`sample_frac()`**.

Efficiently managing and manipulating data structures is paramount when working with large volumes of information. The standard base R functions are certainly capable of sampling, but the syntax often becomes cumbersome when chaining multiple operations together. The **dplyr** approach simplifies this significantly, integrating seamlessly into the Tidyverse ecosystem. By leveraging the pipe operator (**`%>%`**), analysts can express complex data transformations, filtering, and sampling procedures in a highly readable and sequential manner. Understanding these sampling functions is crucial for anyone working regularly with quantitative data in R, as they form the backbone of many simulation and validation techniques.

The Power of dplyr for Data Manipulation

The **dplyr** package, a core component of the Tidyverse, provides a consistent set of verbs for solving the most common data manipulation challenges. These verbs--including `select`, `filter`, `mutate`, `summarise`, and the ones we discuss here, `sample_n` and `sample_frac`--are designed to be intuitive and fast. They operate primarily on the data frame structure, which is the primary way tabular data is stored and managed in R. When sampling, we need methods that are both statistically sound and computationally swift, especially when dealing with millions of observations.

The key to **dplyr**'s readability is the pipe operator (**`%>%`**), which allows the output of one function to be passed directly as the first argument (usually the data frame) of the next function. This structure eliminates nested function calls, making the code flow linearly from data input to final output. When we use **`sample_n()`** or **`sample_frac()`**, we typically pipe the source data frame into the sampling function, telling **R** precisely how many rows or what proportion of rows to extract randomly. We will examine two distinct methodologies for selecting these subsets based on whether a specific count or a relative proportion is required.

Method 1: Selecting a Fixed Number of Rows with sample_n

The first primary method for extracting random observations relies on specifying an exact count of

rows desired in the final sample. This is achieved using the `sample_n()` function. This function is ideal when the required sample size is fixed, regardless of the overall size of the source data frame. For instance, if a study requires exactly 50 participants for a specific validation test, **`sample_n()`** ensures that the resultant data structure contains precisely 50 randomly selected rows.

The basic syntax for utilizing this function is remarkably simple, integrating perfectly with the piping workflow introduced by **dplyr**. The primary argument required by **`sample_n()`** is `size`, which is an integer specifying the number of rows to return.

We can see the application of this method below:

df %>% sample_n(5)

This command instructs **R** to take the data frame **`df`** and randomly select exactly **5** rows from it. By default, this function performs sampling without replacement, meaning once a row is selected, it cannot be selected again in the same sample. However, **`sample_n()`** also supports an argument, `replace = TRUE`, which allows for sampling with replacement, a technique often utilized in bootstrapping or simulation studies where the probability of selecting any given row remains constant across selections.

Method 2: Selecting a Fractional Sample with sample_frac

While **`sample_n()`** is excellent for fixed-size samples, data analysts often need to extract a sample that represents a specific percentage or fraction of the original data set. For this purpose, **dplyr** provides the `sample_frac()` function. This function is particularly useful when the size of the source data frame is variable or when the goal is to maintain a proportional relationship between the sample and the population.

Instead of passing an integer for the sample size, **`sample_frac()`** requires a fractional value between 0 and 1 (exclusive of 0). This fraction determines the proportion of the total rows to be included in the final random sample. For example, a value of **`.10`** would select 10% of the rows, while **`.5`** would select half. This method ensures scalability; if the source data frame grows, the sample size automatically scales proportionally.

The syntax for **`sample_frac()`** is structurally identical to **`sample_n()`**, requiring the fraction as the primary argument, `size`:

df %>% sample_frac(.25)

In this specific execution, the function randomly selects **25%** of all rows present in the data frame `df`. Similar to its counterpart, `sample_frac()` defaults to sampling without replacement, but it also supports the `replace = TRUE` argument for scenarios requiring sampling with replacement. Choosing between `sample_n()` and `sample_frac()` depends entirely on whether the requirement is for an absolute number or a relative proportion of the data.

Practical Implementation and Data Setup

To demonstrate the practical application of both sampling methods, we will first establish a sample data frame in R. This synthetic dataset contains eight observations, representing hypothetical team performance metrics, which will allow us to clearly visualize the effect of both fixed-count and fractional sampling. Establishing a consistent dataset is essential for repeatable examples.

The following code block shows the creation and viewing of our example data structure. We define three variables: `team` (character), `points` (numeric), and `rebounds` (numeric). This simple eight-row structure provides a small but functional example for demonstrating how **dplyr** interacts with data for random selection.

#create data frame
df <- data.frame(team=c('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'),
points=c(10, 10, 8, 6, 15, 15, 12, 12),
rebounds=c(8, 8, 4, 3, 10, 11, 7, 7))

#view data frame
df

team points rebounds
1 A 10 8
2 B 10 8
3 C 8 4
4 D 6 3
5 E 15 10
6 F 15 11
7 G 12 7
8 H 12 7

This data frame, `df`, serves as our population for the next two examples. It contains eight distinct rows, which makes the calculation for fractional sampling straightforward and easy to verify. Before executing any **dplyr** functions, it is necessary to load the package into the current **R** session using the `library()` command, ensuring that the defined functions are accessible.

Case Study: Demonstrating sample_n (Example 1)

Our first practical demonstration uses `sample_n()` to extract a fixed number of rows from our established data frame. For this example, we have chosen to select **5** rows randomly. This represents a substantial portion of our small population (5 out of 8 rows), which makes the randomization effect highly visible. Remember, because the selection is random, executing this code multiple times will yield different results, though the total number of rows returned will always be five.

We begin by ensuring the **dplyr** library is loaded. We then pipe the data frame `df` into `sample_n()`, specifying the desired count:

library(dplyr)

#randomly select 5 rows from data frame
df %>% sample_n(5)

team points rebounds
1 F 15 11
2 A 10 8
3 D 6 3
4 G 12 7
5 B 10 8

The output clearly shows five distinct rows that were randomly selected from the original eight observations. Crucially, the order of the rows in the output is also randomized, not preserving the original index order. This confirms that `sample_n()` successfully performs both the selection and the shuffling necessary for robust random sampling without replacement. If we ran this script again, we would likely get a different combination of five teams.

Case Study: Demonstrating sample_frac (Example 2)

Our second demonstration applies the proportional sampling method using `sample_frac()`. This is beneficial when the user wants the sample size to be dependent on the overall size of the source data. We aim to select **25%** of all rows available in the `df` data frame. Since our data frame contains 8 total rows, 25% of 8 is calculated precisely as 2 rows.

The implementation follows the same structure as the previous example. We load the **dplyr** library (though once loaded in a session, it is not strictly necessary to reload it) and pipe the data frame into `sample_frac()`, providing the fraction `.25`:

library(dplyr)

#randomly select 25% of all rows from data frame
df %>% sample_frac(.25)

team points rebounds
1 E 15 10
2 G 12 7

The resulting output contains exactly two rows (Team E and Team G in this particular random execution), confirming that the fractional sampling calculation worked as expected: 8 observations multiplied by 0.25 yields 2 observations. This dynamic approach ensures that scripts remain robust even as the underlying data volume changes dramatically over time.

It is important to note that when working with large datasets, the fractional result might not always be a perfect integer. In such cases, **dplyr** internally handles the rounding to ensure an integer number of rows is returned, typically rounding down or selecting based on internal stochastic methods to keep the sample size as close as possible to the specified fraction. For further detailed documentation regarding both the **`sample_n`** and **`sample_frac`** functions, including arguments related to weighting (`weight`) or grouping (`group_by`), analysts should consult the official **dplyr** documentation.

Additional Resources

Mastering random sampling is just one component of effective data wrangling in R. The **dplyr** package offers a comprehensive suite of tools for manipulating and summarizing data structures far beyond simple row selection.

For those looking to expand their skills in data transformation using the Tidyverse, the following tutorials explain how to perform other common and essential operations:

Filtering observations based on specific criteria (using `filter()`).
Selecting or dropping specific variables (columns) from a data frame (using `select()`).
Creating new variables or modifying existing ones (using `mutate()`).
Summarizing data across groups (using `group_by()` and `summarise()`).

Note: You can find the complete documentation for the `sample_n` and `sample_frac` functions in dplyr.