# Learning Group Sampling with dplyr in R: A Step-by-Step Guide

Authored by

**Mohammed looti**

November 13, 2025

In modern data science workflows, analysts frequently encounter situations where they must extract representative subsets of data based on specific categories or groups. This essential practice, often referred to as stratified sampling or statistical sampling by group, is vital for tasks ranging from model validation to exploratory data analysis. It ensures that the resulting sample accurately reflects the underlying distribution of the categorized data, preventing systematic bias in analysis. When working within the R programming language, this complex task is handled efficiently and elegantly by specialized packages, most notably the powerful **dplyr** library.

The ability to select unbiased subsets is not merely a convenience; it is a fundamental requirement for robust statistical inference. If a dataset is vast, working with the entire population can be computationally prohibitive. However, simply taking a random slice of the entire dataset can severely misrepresent groups that constitute a smaller portion of the overall population. By utilizing grouped sampling methodologies, we can guarantee proportional or fixed representation from every defined stratum.

## The Crucial Role of Stratified Sampling in Modern Data Analysis

When analyzing large datasets, relying solely on simple random sampling often introduces significant risks, particularly if the dataset exhibits high variance in category sizes. Imagine a scenario involving customer feedback where 90% of customers are from Region A, and 10% are from Region B. A simple random sample of 100 observations might, by chance, select 95 observations from Region A and only 5 from Region B, or potentially exclude Region B entirely. Such an outcome would lead to biased conclusions regarding the customer experience, as the minority group's data would be underrepresented or missing.

This challenge underscores why sampling by group--a form of stratified sampling--becomes indispensable. By implementing this method, the analyst guarantees that a specified number or proportion of observations is drawn from each predefined category, thereby preserving the integrity and structure of the underlying data for subsequent analyses. This procedure ensures a balanced and robust foundation for any subsequent modeling or visualization efforts, mitigating the risk of disproportionate representation.

Fortunately, the dplyr package, a core component of the Tidyverse ecosystem in R, provides powerful and intuitive tools designed specifically for data manipulation tasks like this. The combination of its grouping mechanism, facilitated by the `group_by()` function, and its sampling functions allows for streamlined execution of even complex sampling strategies, transforming otherwise cumbersome tasks into highly readable code sequences.

## Introducing the dplyr Ecosystem and the Tidyverse Philosophy

The **dplyr** package revolutionized data manipulation in R by introducing a consistent set of

functions (or "verbs") that map naturally to common data operations. This focus on clarity and consistency is central to the Tidyverse philosophy, which prioritizes making data science easier, more effective, and more reproducible. The core power of **dplyr** stems from its ability to chain these operations together using the pipe operator (`%>%`), allowing complex transformations to be read sequentially from left to right, significantly improving code readability.

For operations that need to be performed on subsets of data, such as calculating summaries or extracting samples per category, **dplyr** requires the analyst to first explicitly define these subsets. This is where the `group_by()` function plays its critical role. It acts as an internal instruction to R, signaling that all subsequent operations in the current pipeline should be applied independently to the partitioned groups, rather than to the entire data frame globally.

Once the data is conceptually partitioned using `group_by()`, functions like `sample_n()` or `sample_frac()` can be applied. These functions then respect the group boundaries, executing the sampling logic on each distinct subgroup individually. This two-step process--grouping followed by action--forms the backbone of effective grouped data manipulation within the **dplyr** framework.

## Core Mechanics: Combining group_by() and sample_n()

To perform grouped operations, we leverage a robust, sequential approach using the piping sequence (`%>%`). This sequence always begins with the data structure and flows into the operation definitions. First, we define the groups using the `group_by()` function. This function takes the data frame and one or more categorical column names as arguments. Once executed, it effectively partitions the data frame internally, ensuring that subsequent operations are applied independently to each subset defined by the unique combinations of those categorical variables.

Second, we apply a sampling function. For selecting a fixed number of rows from each group, we use `sample_n()`. Crucially, when `sample_n()` is called immediately after `group_by()`, it interprets its required `size` parameter not as the total number of rows to select from the entire dataset, but as the exact count of observations to select from *each* distinct group. This architectural feature is what makes **dplyr** so intuitive and efficient for complex data wrangling tasks requiring stratified selection.

The power of this combination is evident in its conciseness. Instead of writing complex loops or applying subsetting logic repeatedly for every unique category, the analyst can accomplish the task in a single, flowing chain of commands, making the intent of the code transparent and reducing the opportunity for manual errors in subgroup management.

## Syntax Deep Dive: Mastering the Parameters of sample_n()

The primary tool for selecting a random, fixed-size sample of rows is the sample_n() function.

When used in conjunction with `group_by()`, it provides the precise control needed for subgroup sampling. Understanding its parameters is essential for successful implementation. The standard syntax for this function is:

**sample_n(tbl, size, replace = FALSE, weight = NULL, .drop = FALSE, ...)**

While the full function includes several arguments, the key parameters determining the core sampling process are:

**tbl**: This mandatory argument specifies the input data structure, typically a data frame or tibble that has already been piped from the `group_by()` function.
**size:** This required parameter dictates the number of rows to select. As previously noted, when operating on grouped data, this value represents the exact count of observations sampled from *each* group.
**replace**: This logical argument determines whether to sample with replacement (`TRUE`) or without replacement (`FALSE`). Sampling without replacement (the default setting) means that a single row cannot be selected multiple times, ensuring a unique sample set. This is the standard behavior and generally preferred unless dealing with specific simulation techniques.

It is important to emphasize that for most standard exploratory sampling tasks, leaving the value for the **replace** argument set to **FALSE** is the most common and statistically sound practice. Using sampling with replacement (`TRUE`) allows the same observation to appear multiple times in the final sample, which is typically undesirable when the goal is to obtain a unique, representative subset of the population within each category for downstream analysis.

## Example: How to Sample by Group Using dplyr

### Practical Demonstration: Sampling Basketball Data by Team

To solidify the theoretical mechanics of grouped sampling, let us construct a simple, practical dataset. This dataset represents fictional basketball player statistics and includes critical information such as the team a player belongs to, their points scored, assists, and rebounds. This structure presents the perfect, simplified scenario for demonstrating how to extract a fair, random sample of players based on their respective teams (our categorical groups).

The goal is to move from a complete, unbalanced list of player statistics to a balanced subset where representation is guaranteed across all teams. Suppose we create the following data frame in the R programming language that contains information about various basketball players:

**#create data frame**
**df <- data.frame(team=c('A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'),**

**points=c(99, 68, 86, 88, 95, 74, 78, 93),**
**assists=c(22, 28, 45, 35, 34, 45, 28, 31),**
**rebounds=c(30, 28, 24, 24, 30, 36, 30, 29))**

#view data frame
df

team points assists rebounds
1 A 99 22 30
2 A 68 28 28
3 A 86 45 24
4 A 88 35 24
5 B 95 34 30
6 B 74 45 36
7 B 78 28 30
8 B 93 31 29

Upon inspection of the resulting data frame, we clearly observe two distinct groups: **Team A** and **Team B**, each containing exactly four player entries. Our objective is now to extract an equal, random sample of players from both groups simultaneously. This balanced approach is crucial for any subsequent comparative analysis of the teams.

## Implementing Fixed-Count Grouped Sampling in R

The most common and illustrative use case for the **dplyr** framework is selecting an identical, fixed count of observations from every subgroup using a chained operation. For instance, if our analytical requirement is to select three random basketball players from each of the two teams (A and B), we can chain the `group_by()` and sample_n() functions together using the pipe operator (`%>%`).

The code sequence below first loads the **dplyr** library, a necessary prerequisite for accessing these functions. It then uses `group_by(team)` to define the scope of the operation, ensuring the separation of Team A and Team B data. Finally, it applies `sample_n(size=3)`. Because the data is grouped, the `size=3` argument is interpreted three rows per team. The resulting output will contain six rows in total, three randomly selected from Team A and three randomly selected from Team B:

**library(dplyr)**

#select three random players from each team
df %>%
group_by(team) %>%

```
sample_n(size=3)

# A tibble: 6 x 4
# Groups: team
team points assists rebounds

1 A 86 45 24
2 A 99 22 30
3 A 88 35 24
4 B 78 28 30
5 B 93 31 29
6 B 95 34 30
```

As clearly demonstrated by the output, we successfully returned three randomly selected players from each team, achieving a perfectly balanced, stratified sample. The inherent flexibility of the `sample_n()` function allows us to easily adjust the sample size simply by modifying the `size` argument. If, for instance, we only needed two random players per team for a smaller experimental setup, the syntax remains virtually identical, only changing the value passed to the `size` parameter:

```
library(dplyr)

#select two random players from each team
df %>%
group_by(team) %>%
sample_n(size=2)

# A tibble: 4 x 4
# Groups: team
team points assists rebounds

1 A 88 35 24
2 A 99 22 30
3 B 78 28 30
4 B 93 31 29
```

This result confirms that exactly two random players were selected from each team, matching the specified criteria. It is critical to remember that because the `sample_n()` function inherently relies on internal random number generation, the specific rows selected for each group will likely change every time this code chunk is executed. This natural randomness is expected in statistical

sampling, but it often necessitates methods for control when sharing or reproducing results, which leads us to the topic of reproducibility.

## Ensuring Robustness and Reproducibility with set.seed()

In many analytical contexts, especially when preparing reports, submitting research for peer review, or debugging complex pipelines, it is absolutely essential that the exact same random sample is generated every single time the script is run. This non-negotiable requirement for consistency is met by utilizing the **set.seed()** function in R.

By calling **set.seed()** with a specific integer value (the "seed") immediately before the sampling operation, we initialize R's pseudo-random number generator to a fixed starting point. This guarantees that the sequence of "random" numbers generated by functions like `sample_n()` will be identical across subsequent runs, thereby making the results fully reproducible across different environments and times.

The specific integer value chosen for the seed (e.g., 1, 42, 100) is arbitrary, but its consistent application is paramount. If the seed is changed, a different random sample will be obtained, but that new sample will also be consistently replicated on subsequent runs using the same new seed. If **set.seed()** is omitted, R defaults to using elements like the current system time to generate a seed, leading to different sampling results each time the script is executed. For example, we could use the following code to ensure our selection of two random players per team is consistent, enabling perfect replication:

```
#make this example reproducible
set.seed(1)

library(dplyr)

#select two random players from each team
df %>%
group_by(team) %>%
sample_n(size=2)
```

By adding **set.seed(1)**, we guarantee that the specific rows selected for Team A and Team B in this example will remain constant regardless of how many times the script is run or who runs it. This capability is fundamental to creating reliable and auditable data analysis pipelines.

## Conclusion and Further Resources

The combination of `group_by()` and `sample_n()` within the **dplyr** package offers a robust,

efficient, and highly readable methodology for performing stratified random sampling in R. This technique is fundamental for creating unbiased, representative subsets of data, especially when dealing with heterogeneous populations structured by categorical variables. Mastering this approach significantly enhances the reliability and efficiency of nearly any data analysis pipeline, ensuring that conclusions drawn from sampled data are statistically sound and accurately reflect the underlying population structure.

The following tutorials explain how to perform other common tasks in R:

<!--

## Featured Posts

-->