Split Data into Training & Test Sets in R (3 Methods)

Authored by

Mohammed looti

October 30, 2025

In the realm of machine learning and statistical modeling, a fundamental and mandatory practice for developing robust and reliable predictive models is the partitioning of the original dataset into distinct, non-overlapping subsets. Specifically, we create a training set and a test set. This crucial data segregation allows us to develop and tune the model using one portion of the data, and then rigorously evaluate its true performance and generalizability on entirely unseen data.

The core motivation behind this data splitting procedure is to effectively combat overfitting. Overfitting occurs when a model learns the training data, including its specific noise and idiosyncrasies, too perfectly. Consequently, a model that performs exceptionally well on the training data often performs poorly when presented with new, real-world observations. By reserving a portion of the data as the test set, we secure an unbiased assessment of how well our final model is likely to perform outside of the development environment.

This article provides an expert guide to three common and highly effective methodologies for splitting your data into training and testing partitions within the powerful R programming environment. We will explore techniques ranging from foundational Base R functions, which rely on simple indexing and random sampling, to advanced, specialized packages like caTools and dplyr. Practical, reproducible code examples accompany each approach, ensuring clarity and immediate applicability.

Method 1: Data Partitioning with Base R Functions

The Base R approach offers the most straightforward and transparent method for dataset splitting. It relies exclusively on native R functions to perform random sampling, assigning observations to either the training or test set based on a user-defined proportional split (e.g., 70% for training, 30% for testing). This technique is highly flexible and provides data scientists with a deep, fundamental understanding of the underlying data sampling process, making it an ideal starting point for data partitioning in R.

The methodology centers on generating a logical vector whose length matches the number of rows in the dataset. Each element in this vector is randomly assigned a value of `TRUE` or `FALSE`, which subsequently dictates whether the corresponding row belongs to the training or test set, respectively. A critical component for any statistical analysis is ensuring that the results are reproducible; thus, the `set.seed()` function must always be used prior to any random sampling operation. By setting the seed, we guarantee that the sequence of random numbers generated, and therefore the specific split of the data, remains identical every time the script is executed.

The general template below illustrates how to achieve a standard 70/30 training-test split using Base R. The `sample()` function is instrumental here, generating the necessary random indices. Notice how the `prob` argument controls the exact desired proportions for the split.

#make this example reproducible
set.seed(1)

#use 70% of dataset as training set and 30% as test set
sample <- sample(c(TRUE, FALSE), nrow(df), replace=TRUE, prob=c(0.7,0.3))
train <- df
test <- df

Within this code snippet, `nrow(df)` dynamically calculates the total number of observations in your target dataset. The critical argument `prob=c(0.7, 0.3)` explicitly dictates that 70% of the generated logical values should be `TRUE` (to be allocated to the training set), and the remaining 30% should be `FALSE` (for the test set). The `replace=TRUE` argument ensures that the sampling is conducted with replacement from the boolean vector, thereby guaranteeing that the correct proportions are maintained across the entire dataset when generating the indices.

Method 2: Data Partitioning with the caTools Package

When the distribution of the target variable must be carefully preserved across both subsets, the caTools package, specifically its highly capable `sample.split()` function, provides a superior, specialized approach. This function facilitates stratified sampling, a technique that ensures that the proportions of a specified categorical variable--typically the outcome variable in a classification tasks--are accurately represented in both the training and test sets. This capability is absolutely vital when working with imbalanced datasets, as it prevents the training set from becoming biased toward the majority class.

The `sample.split()` function requires two main inputs: a vector (usually the column representing the target variable from your data frame) and the `SplitRatio`, which defines the desired proportion for the training set. Like the Base R method, it returns a logical vector. However, this vector is intelligently generated to achieve stratification, allowing it to be used directly to subset the main data frame using the `subset()` function.

The following example demonstrates how to perform a 70/30 stratified split using `caTools`. Note that we still employ `set.seed()` for reproducibility. A key difference from the Base R method is that `sample.split()` operates on a specific column (e.g., `df$any_column_name`) to ensure that the stratification logic is correctly applied based on that variable's distribution.

library(caTools)

#make this example reproducible
set.seed(1)

```
#use 70% of dataset as training set and 30% as test set
sample <- sample.split(df$any_column_name, SplitRatio = 0.7)
train <- subset(df, sample == TRUE)
test <- subset(df, sample == FALSE)
```

The primary and most compelling benefit of leveraging `caTools` is the inherent stratification capability. This ensures that the subsets are highly representative of the overall population, particularly regarding key variables. Such balanced partitioning is fundamental for training machine learning models that are robust and generalize well, avoiding bias that might result from disproportionate representation of classes in the training data.

Method 3: Data Partitioning with the dplyr Package

The dplyr package, an essential component of the modern tidyverse, offers an aesthetically pleasing and highly efficient syntax for virtually all data manipulation tasks, including dataset splitting. Its functions are renowned for being intuitive and for their ability to be chained together seamlessly using the pipe operator (`%>%`), which results in code that is both highly readable and exceptionally efficient.

For data partitioning, `dplyr` utilizes the `sample_frac()` function to randomly select a specified fraction of the rows for the training set. Determining the test set then requires a slightly different approach: the remaining observations are precisely identified using the `anti_join()` function. The `anti_join()` function is designed to return all rows from the first data frame that do not have a corresponding match in the second. Crucially, this method requires the prior creation of a unique identifier column, such as an ID, to ensure that the matching and exclusion processes are accurate and unambiguous.

The implementation below details a 70/30 split using the `dplyr` methodology. We begin by adding a temporary unique ID column, which is essential for the subsequent `anti_join()` operation to precisely separate the retained (training) and excluded (testing) rows.

library(dplyr)

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

#create ID column
df$id <- 1:nrow(df)

#use 70% of dataset as training set and 30% as test set
train <- df %>% dplyr::sample_frac(0.70)
```

test <- dplyr::anti_join(df, train, by = 'id')

The `sample_frac(0.70)` command randomly selects exactly 70% of the observations to form the training set. Following this, the `anti_join(df, train, by = 'id')` function operates efficiently to select every row from the original data frame `df` that was *not* included in the `train` set, based on their unique `id`. This guarantees that the test set consists of the precise remaining 30% of the data, providing a clean and elegant separation.

Practical Demonstration: Splitting the Iris Dataset

To provide concrete examples of the three methods in action, we will apply them to R's built-in Iris dataset. This classic dataset contains 150 observations, each describing an iris flower with four key measurements (sepal length, sepal width, petal length, and petal width) and its corresponding species. We will consistently target a 70% training split and a 30% test split.

Example 1: Base R Implementation on Iris

We start by implementing the Base R method, ensuring our results are reproducible by setting the random seed. The goal is a purely random selection of 70% of the rows for training.

#load iris dataset
data(iris)

#make this example reproducible
set.seed(1)

#Use 70% of dataset as training set and remaining 30% as testing set
sample <- sample(c(TRUE, FALSE), nrow(iris), replace=TRUE, prob=c(0.7,0.3))
train <- iris
test <- iris

#view dimensions of training set
dim(train)

106 5

#view dimensions of test set
dim(test)

44 5

The execution reveals that the **training set** contains **106 rows** and the **test set** contains **44 rows**. Since the original dataset had 150 rows, the training set holds 70.6% of the data, which closely aligns with our desired 70% split ratio, validating the effectiveness of the Base R sampling approach. We can inspect the structure by viewing the initial observations:

#view first few rows of training set
head(train)

Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 5.1 3.5 1.4 0.2 setosa
2 4.9 3.0 1.4 0.2 setosa
3 4.7 3.2 1.3 0.2 setosa
5 5.0 3.6 1.4 0.2 setosa
8 5.0 3.4 1.5 0.2 setosa
9 4.4 2.9 1.4 0.2 setosa

Example 2: caTools Implementation on Iris (Stratified Split)

Next, we employ the `caTools` package to perform a stratified split based on the `Species` column. This ensures that the proportions of the three species (setosa, versicolor, virginica) are maintained as closely as possible in both the training and test sets, which is ideal for evaluating a classification model.

library(caTools)

#load iris dataset
data(iris)

#make this example reproducible
set.seed(1)

#Use 70% of dataset as training set and remaining 30% as testing set
sample <- sample.split(iris$Species, SplitRatio = 0.7)
train <- subset(iris, sample == TRUE)
test <- subset(iris, sample == FALSE)

#view dimensions of training set
dim(train)

105 5

#view dimensions of test set

dim(test)

45 5

In this stratified split, the **training set** contains **105 rows** and the **test set** contains **45 rows**. The slight difference in row counts compared to the Base R method (106/44 vs. 105/45) is a direct result of the stratification algorithm, which prioritizes maintaining proportional representation of the target variable (`Species`) over achieving an exact 70.00% row split. The total number of rows (150) remains correct, and the split is statistically more balanced.

Example 3: dplyr Implementation on Iris

Finally, we apply the `dplyr` method, leveraging its expressive syntax and the `anti_join()` function. As required by this method, we first create a temporary unique identifier column before executing the 70% random fractional sampling.

library(dplyr)

#load iris dataset
data(iris)

#make this example reproducible
set.seed(1)

#create ID variable
iris$id <- 1:nrow(iris)

#Use 70% of dataset as training set and remaining 30% as testing set
train <- iris %>% dplyr::sample_frac(0.7)
test <- dplyr::anti_join(iris, train, by = 'id')

#view dimensions of training set
dim(train)

105 6

#view dimensions of test set
dim(test)

45 6

The resulting **training set** contains **105 rows** and the **test set** contains **45 rows**. Crucially, both

resulting data frames now contain 6 columns, reflecting the inclusion of the temporary `id` column we created. While this identifier was essential for the precise execution of the `anti_join()` function, it is generally not a feature required for subsequent machine learning algorithm training. Best practice dictates that this `id` column should be explicitly dropped or excluded from the feature set before proceeding with model development and evaluation.

Conclusion and Best Practices

Mastering data partitioning is a foundational skill for any data scientist operating within R. The three methods presented--Base R, caTools, and dplyr--each offer distinct advantages, allowing users to select the optimal technique based on their specific requirements, the characteristics of the data (e.g., balance of classes), and their preference for coding style. The Base R method is efficient for simple random splits, caTools excels at stratified sampling, and dplyr provides excellent integration into the modern Tidyverse workflow.

Regardless of the chosen method, the consistent application of `set.seed()` remains absolutely paramount. This simple step ensures the reproducibility of your analysis, allowing others (or your future self) to replicate the exact training and test sets, thereby validating the model's performance assessment. Choosing the right partitioning technique is the first critical step toward building robust and reliable predictive models.

For those looking to deepen their understanding of R and explore further data manipulation techniques, the following tutorials explain how to perform other common operations in R: