Creating Train and Test Datasets from Pandas DataFrames for Machine Learning

Authored by

Mohammed looti

October 29, 2025

In the field of machine learning, the journey toward developing robust and accurate predictive models begins long before the training algorithm is executed. A foundational and absolutely critical step is the meticulous preparation of the input dataset. This preparation involves a strategic division of the comprehensive data into distinct, non-overlapping subsets. This process of data splitting is essential not only for preventing common modeling pitfalls but also for ensuring a trustworthy evaluation of the model's true performance capabilities.

Ignoring this crucial step can lead to highly optimistic but ultimately misleading results, as a model evaluated only on the data it has already seen cannot provide a reliable indication of how it will perform in a real-world production environment. Therefore, understanding and correctly implementing the division into training and testing sets is paramount for any data science professional.

The Essential Role of Data Splitting in Model Validation

The core objective of any statistical or machine learning model is not merely to perfectly replicate the outcomes observed in the provided data, but rather to generalize effectively to new, unseen observations. When a model is trained, it learns patterns, relationships, and features within the input data. If the model evaluation is performed on the same data used for training, the performance metrics will invariably be inflated because the model may have simply memorized the specific examples, including any underlying noise or anomalies.

This memorization phenomenon is formally known as overfitting. An overfit model typically exhibits near-perfect performance on the training data but fails dramatically when exposed to novel data points. This lack of robust generalization renders the model useless for real-world prediction tasks. To accurately gauge the model's capacity for generalized prediction--that is, its ability to correctly handle data it has never encountered--we must reserve a portion of the original dataset solely for testing.

By sequestering this testing data, we create an unbiased benchmark. When the fully trained model is finally run against this reserved set, the resulting performance metrics provide an honest assessment of its ability to infer underlying structure rather than just rote memorization. This methodological rigor ensures that we deploy models that are reliable, stable, and capable of delivering value in operational environments.

Defining the Training and Testing Subsets

When preparing data for machine learning implementation, the entire available pool of data is standardly partitioned into two primary components:

Training Set: This substantial portion of the data is the foundation upon which the model is built.

During the training phase, the algorithm iteratively processes these examples, learning the underlying relationships between features and the target variable, and adjusting its internal parameters to minimize prediction errors. Conventionally, this set comprises the majority of the original dataset, usually allocating 70% to 80% of the observations.

Testing Set: This set is critically reserved for the final, independent evaluation of the trained model. It must remain completely untouched and unseen by the model during the learning phase. Using this fresh data, we obtain a precise and unbiased estimate of the model's true predictive power and its capability for generalization, effectively simulating its performance in a live environment. The testing set typically accounts for the remaining 20% to 30% of the original observations.

In the Python ecosystem, especially when managing structured, tabular data using the highly popular Pandas DataFrame (Usage 1/5), several powerful and efficient methods exist to execute this division. The subsequent sections will provide detailed, practical demonstrations of the two most common approaches employed by data scientists today, focusing on both the standard library method and a native Pandas solution.

Setting Up the Example: A Sample Pandas DataFrame

To provide clear, reproducible examples of data splitting, we must first establish a representative data structure. This structure will take the form of a Pandas DataFrame (Usage 2/5), simulating a typical machine learning dataset. For demonstration purposes, our DataFrame will contain 1,000 observations and three columns, representing input features and an output target variable.

We rely on the NumPy library for generating high-quality random data efficiently. A crucial aspect of any reliable data science workflow is reproducibility. To ensure that the exact same split occurs every time the code is run, we must explicitly set the `random_state` (Usage 1/5) parameter. This practice is essential for debugging, peer review, and ensuring consistency across different development environments.

The following Python script initializes our example DataFrame, named `df`, which will serve as the source data for both splitting techniques:

```
import pandas as pd
import numpy as np

# Make this example reproducible by setting a seed for NumPy's random number generator.
np.random.seed(1)

# Create a DataFrame with 1,000 rows and 3 columns.
# 'x1' and 'x2' represent features, and 'y' could be a target variable.
```

df = pd.DataFrame({'x1': np.random.randint(30, size=1000),
'x2': np.random.randint(12, size=1000),
'y': np.random.randint(2, size=1000)})

# View the first few rows of the newly created DataFrame to understand its structure.
df.head()

x1 x2 y
0 5 1 1
1 11 8 0
2 12 4 1
3 8 7 0
4 9 0 0

The resulting DataFrame `df` is now ready for partitioning. Columns `x1` and `x2` represent the independent variables (features) that our future model would use to make predictions, while column `y` represents the dependent variable (target) that the model attempts to predict. The subsequent methods will demonstrate how to separate these 1,000 observations into their respective training and testing components.

Technique 1: Leveraging scikit-learn's `train_test_split()`

For most machine learning (Usage 3/5) tasks in Python, the scikit-learn (Usage 1/5) library is the definitive standard. Its `model_selection` module provides the highly utilized `train_test_split()` function (Usage 1/5), which is the most recommended and feature-rich way to partition data.

The `train_test_split()` function offers flexibility and control over the splitting process. It efficiently shuffles the data and divides it based on specified proportions. Key parameters include `test_size`, which defines the fraction of the data to be held back for testing (e.g., 0.2 for 20%), and crucially, `random_state` (Usage 2/5). Setting the `random_state` (Usage 3/5) to an integer ensures that the random shuffling sequence is identical every time, which is paramount for achieving reliable and consistent development results.

The following code snippet demonstrates the implementation of `train_test_split()` (Usage 2/5) on our example Pandas DataFrame (Usage 3/5), followed by an inspection of the resulting subsets to confirm the correct distribution:

from sklearn.model_selection import train_test_split

Split the original DataFrame into training and testing sets.

```
# 20% of the data will be used for testing (test_size=0.2).
# random_state=0 ensures that the split is reproducible.
train, test = train_test_split(df, test_size=0.2, random_state=0)

# View the first few rows of each set to confirm the split.
print(train.head())
```

```
x1 x2 y
687 16 2 0
500 18 2 1
332 4 10 1
979 2 8 1
817 11 1 0
```

```
print(test.head())
```

```
x1 x2 y
993 22 1 1
859 27 6 0
298 27 8 1
553 20 6 0
672 9 2 1
```

```
# Print the size (number of rows and columns) of each set to verify the split proportions.
print(train.shape, test.shape)
```

```
(800, 3) (200, 3)
```

The output confirms a successful 80/20 split: the training set (Usage 3/5) now contains 800 rows, and the testing set (Usage 4/5) contains 200 rows. A crucial advantage of `train_test_split()` (Usage 3/5) is its direct return of both sets in a single, clean operation, streamlining the data preparation stage.

Technique 2: Simple Splitting using Pandas `sample()` Method

While the scikit-learn approach is the standard for complex modeling pipelines, there are scenarios where a lighter, native Pandas (Usage 4/5) solution is preferable. The `sample()` method (Usage 1/5), native to the Pandas DataFrame (Usage 5/5), provides a highly intuitive and efficient way to perform random sampling directly on the data object itself.

The `sample()` method (Usage 2/5) uses the `frac` parameter to determine the proportion of rows to

select for the sample. To achieve the 80/20 split, we sample 80% of the DataFrame for the training set. The remaining observations, which constitute the testing set, are then isolated by using the `drop()` method on the original DataFrame, removing all indices that were selected for training. As with all random processes in data science, specifying the `random_state` (Usage 4/5) parameter ensures that the sampling process is perfectly replicable.

Here is the practical implementation of the `sample()` approach, demonstrating how to generate both the training and testing sets (Usage 5/5) using only Pandas functionalities:

 # Split the original DataFrame into training and testing sets using the sample() method.
 # frac=0.8 means 80% of the DataFrame will be sampled for the training set.
 # random_state=0 ensures the sampling is reproducible.
train = df.sample(frac=0.8,random_state=0)
 # The test set is created by dropping the indices of the training set from the original DataFrame.
test = df.drop(train.index)

# View the first few rows of each set to confirm the split.
print(train.head())

x1 x2 y
993 22 1 1
859 27 6 0
298 27 8 1
553 20 6 0
672 9 2 1

print(test.head())

x1 x2 y
9 16 5 0
11 12 10 0
19 5 9 0
23 28 1 1
28 18 0 1

# Print the size (number of rows and columns) of each set to verify the split proportions.
print(train.shape, test.shape)

(800, 3) (200, 3)

The resulting shapes confirm that the split was performed correctly, yielding 800 rows for training and 200 for testing. This method is highly effective for simple random splits and leverages the speed and efficiency of native Pandas indexing operations.

Comparative Analysis: Choosing the Right Tool

When faced with the choice between `train_test_split()` (Usage 4/5) from scikit-learn (Usage 2/5) and the `sample()` method (Usage 3/5) in Pandas, the decision often hinges on the complexity of the data and the overall workflow requirements.

The `train_test_split()` (Usage 5/5) function is overwhelmingly the preferred choice in production-level machine learning (Usage 4/5) pipelines. Its advantages include the ability to handle multiple arrays simultaneously (e.g., separating features X and labels y in one call), and more critically, its support for advanced splitting techniques like **stratified sampling**. Stratified sampling ensures that the proportion of different classes in the target variable is maintained identically across both the training and testing sets, which is vital when dealing with imbalanced datasets.

Conversely, the `sample()` method (Usage 4/5) shines in its simplicity and directness. It requires no external dependencies beyond Pandas and is ideal for quick exploratory data analysis (EDA) or when only a simple, non-stratified random split is required. While it requires the extra step of using `.drop(train.index)` to derive the test set, its native integration into the Pandas environment can make the code cleaner for data manipulation tasks that precede the actual modeling phase.

Summary and Best Practices

Effectively splitting a dataset (Usage 5/5) into dedicated training and testing sets remains a cornerstone practice in building reliable machine learning (Usage 5/5) models. This strategic division is the primary defense against overfitting and the only way to obtain an unbiased, reliable estimate of a model's true ability to generalize to real-world scenarios.

Regardless of whether you choose the feature-rich and standardized approach of `train_test_split()` or the straightforward efficiency of the `sample()` method (Usage 5/5), the consistent use of the `random_state` (Usage 5/5) parameter is non-negotiable. This simple addition elevates your data science work from experimental code to reproducible, professional engineering, ensuring that your results are consistent and trustworthy across all development stages.

Additional Resources

To further enhance your Python data science skills, explore these related tutorials that delve into other common tasks:

How to Calculate Balanced Accuracy in Python