How to Perform Stratified Sampling in Pandas (With Examples)

Authored by

stats writer

December 22, 2025

Stratified sampling is a highly effective statistical technique used in data science and research. It involves segmenting the entire target population into mutually exclusive subgroups, known as strata, based on shared characteristics relevant to the study. This methodology ensures that the final sample accurately reflects the heterogeneity present in the population. When working with large datasets, the Pandas library in Python offers a robust and straightforward method for executing complex stratification logic. This guide will walk you through the process, providing detailed code examples to help you master stratified sampling within the Pandas environment, covering both fixed count and proportional methods.

In the realm of statistical analysis, researchers frequently draw samples from a larger data set, or population, to generate reliable inferences and conclusions about the whole. The goal is always to achieve a sample that is representative, minimizing bias and maximizing the generalizability of the findings.

One of the most powerful techniques for achieving this representativeness is **stratified random sampling**. This sophisticated approach systematically divides the population into distinct groups (strata) before randomly selecting observations from each group. This guarantees that critical subgroups are adequately represented in the final sample, which is especially important when dealing with imbalanced or complex data distributions.

This comprehensive tutorial details two essential methods for conducting stratified random sampling efficiently using the capabilities of the **Pandas** library in the Python ecosystem.

Understanding the Importance of Stratification

Why is simply taking a simple random sample not always sufficient? Simple random sampling runs the risk of underrepresenting smaller but statistically significant subgroups, particularly if the population structure is highly skewed or if the characteristic being studied varies widely across different segments. **Stratified sampling** mitigates this risk by treating these subgroups separately.

The core benefit of using stratified sampling lies in its ability to produce more precise estimates. By ensuring that the variability within each stratum is captured, the overall sampling error is often reduced compared to other methods like simple random sampling or cluster sampling. The subgroups (strata) are defined by the user based on domain knowledge--common stratifying variables include geographic location, age bracket, gender, or, as shown in our examples, team affiliation.

Working with the Pandas DataFrame structure makes this process intuitive. Pandas provides the necessary tools, such as the `groupby()` and `sample()` functions, which can be combined elegantly to implement stratification logic effectively across various use cases in data analysis.

Example 1: Stratified Sampling Using Fixed Counts

The first common scenario involves selecting an equal or fixed number of observations from each defined stratum, regardless of the stratum's size relative to the overall population. This method is crucial when you need sufficient data points from every subgroup for detailed comparative analysis.

To demonstrate this, let us create a sample **Pandas DataFrame** representing data for eight fictional basketball players, categorized by their respective teams and positions.

import pandas as pd

#create DataFrame with balanced teams (4 players each)
df = pd.DataFrame({'team': ,
'position': ,
'assists': ,
'rebounds': })

#view DataFrame structure
df

team position assists rebounds
0 A G 5 11
1 A G 7 8
2 A F 7 10
3 A G 8 6
4 B F 5 6
5 B F 7 9
6 B C 6 6
7 B C 9 10

In this initial dataset, we have two distinct strata defined by the 'team' column: Team A and Team B, each containing four players. Our objective is to perform a stratified random selection by pulling exactly two players from each of these teams to form a representative sample of size four. This guarantees a balanced comparison between the two teams.

Executing Count-Based Stratification in Pandas

The core mechanism in Pandas for achieving stratification is the combined use of the `groupby()` function followed by the `apply()` method, which lets us execute a function on each group independently. Within the `apply()` method, we use a simple lambda function incorporating

`sample()`. The `sample()` function is where the random selection occurs, taking the specified count (in this case, 2) as its argument.

We set `group_keys=False` in the `groupby()` call to prevent the grouping key (the 'team' column) from being set as the index of the resulting DataFrame. This ensures the output maintains a clean, standard index structure for the final sample, making it easier to work with downstream statistical analysis tools.

df.groupby('team', group_keys=False).apply(lambda x: x.sample(2))

team position assists rebounds
0 A G 5 11
3 A G 8 6
6 B C 6 6
5 B F 7 9

Upon reviewing the result, it is clear that the code successfully executed the stratified sampling rule: two players were randomly drawn from Team A (indices 0 and 3 in this specific run) and two players were randomly drawn from Team B (indices 6 and 5). This fixed-count approach ensures that both strata are equally weighted in the resulting sample, which is highly beneficial for balanced comparative studies, irrespective of the original team sizes.

Example 2: Stratified Sampling Using Proportions

While fixed-count sampling is useful for balanced analysis, often researchers require the sample to accurately mirror the proportions found in the original population. This is known as **proportional allocation**. If 75% of the population belongs to Stratum A, then 75% of the sample must also belong to Stratum A.

For this example, we modify our initial **DataFrame** to create an imbalanced population distribution, making the need for proportional sampling evident. This scenario is far more common in real-world data science problems where groups are rarely of equal size.

import pandas as pd

#create DataFrame with imbalanced teams (6 players on B, 2 players on A)
df = pd.DataFrame({'team': ,
'position': ,
'assists': ,
'rebounds': })

#view DataFrame
df

team position assists rebounds
0 A G 5 11
1 A G 7 8
2 B F 7 10
3 B G 8 6
4 B F 5 6
5 B F 7 9
6 B C 6 6
7 B C 9 10

Observe the new distribution: the DataFrame now contains 8 players in total. Team A has 2 players (25% of the total), while Team B has 6 players (75% of the total). If we intend to draw a total sample size of $N=4$, we must ensure the sample maintains this 1:3 ratio. This requires us to aim for 1 player from Team A and 3 players from Team B.

Executing Proportional Stratification in Python

Achieving proportional allocation requires incorporating a calculation within the `apply()` function to determine the specific sample size needed for each group dynamically. Since we need to perform mathematical rounding to ensure integer sample sizes, we must introduce the powerful numerical library, NumPy.

The core of the proportional sampling logic is the calculation performed within the lambda function: `int(np.rint(N * len(x) / len(df)))`. Here, `N` is the desired total sample size (set to 4). `len(x)` represents the size of the current stratum being processed by `groupby()`, and `len(df)` is the total size of the original population. This formula calculates the exact proportion of the total sample size (N) that should be allocated to the current stratum (x). We use `np.rint` for rounding to the nearest whole integer, which is essential because the `sample()` function requires an integer input for the number of items to select.

We also chain two important functions at the end of the code block: `sample(frac=1)` randomly shuffles the rows of the resulting sample to randomize the order, and `reset_index(drop=True)` cleans up the index, removing the old indices inherited from the original DataFrame, resulting in a clean, production-ready sample.

import numpy as np

#define total sample size desired

```
N = 4

#perform stratified random sampling
df.groupby('team', group_keys=False).apply(lambda x:
x.sample(int(np.rint(N*len(x)/len(df))))).sample(frac=1).reset_index(drop=True)
```

team position assists rebounds
0 B F 7 9
1 B G 8 6
2 B C 6 6
3 A G 7 8

Verifying Proportional Accuracy and Inference

By inspecting the resulting sample **DataFrame**, we can confirm that the proportional allocation worked perfectly. The total sample size is 4. Team A is represented by 1 player (25% of the sample), and Team B is represented by 3 players (75% of the sample). This distribution exactly matches the original population ratios (25% Team A, 75% Team B).

This proportional method is critical when the primary goal is to maintain the true distribution characteristics of the source data, ensuring that statistical inferences drawn from the sample are directly applicable back to the entire, potentially imbalanced, population without the need for complex weighting adjustments later on. The seamless integration of Pandas grouping functions with NumPy mathematical precision makes this a robust solution in Python for advanced sampling needs.

Both the fixed-count and proportional methods showcase the flexibility of using `groupby()` and `sample()` together. The choice between these two methods depends entirely on the research objective: fixed counts for balanced comparison studies, and proportional allocation for maximizing representativeness of the underlying population structure.

Further Sampling Methods in Pandas

The ability to manipulate data structures efficiently is why DataFrames are the backbone of modern data science workflows. Beyond stratification, Pandas supports various other advanced sampling techniques useful for data exploration and model training.

For instance, if your data is naturally clustered--perhaps geographically or within defined time periods--you might find **cluster sampling** more appropriate. In cluster sampling, entire groups, rather than individual observations, are randomly selected. Understanding these different sampling strategies is key to deriving accurate and reliable insights from complex datasets and ensuring that

your chosen method aligns with your research design.

For those seeking to explore alternative approaches, the following resources provide additional tutorials on related sampling methodologies in Pandas:

How to Perform Cluster Sampling in Pandas