How to Easily Replace NaN Values with the Median in Pandas

Authored by

stats writer

December 1, 2025

Handling missing data is one of the most critical steps in the data preprocessing pipeline. In the Pandas library for Python, missing values, often represented as NaN (Not a Number), must be addressed before effective analysis or modeling can take place. A robust and frequently utilized method for dealing with these gaps is imputation: replacing the missing values with a calculated statistic from the existing data.

Specifically, replacing NaN values with the column's median is a standard practice, particularly favored when the dataset is susceptible to extreme values or possesses a skewed distribution. The median, defined as the middle value in a sorted dataset, offers a measure of central tendency that is inherently less sensitive to outliers compared to the arithmetic mean. This resilience ensures that the imputed values do not unduly influence the overall statistical properties of the distribution.

The core mechanism for this operation in Pandas relies on the powerful `.fillna()` function, which takes the calculated median of the relevant column(s) as its argument. By executing this simple yet effective procedure, data scientists can restore data integrity and prepare the DataFrame for downstream tasks, minimizing potential biases introduced by incomplete observations.

Understanding the fillna() Function for Imputation

The primary tool for managing missing data imputation within a Pandas DataFrame is the **fillna()** function. This method is highly versatile, allowing users to replace **NaN** values with either a specific scalar value, a dictionary of replacement values keyed by column, or, most commonly in statistical imputation, a calculated series (like the mean or median) derived from the existing data.

When we use **fillna()** in conjunction with the `.median()` method, we are essentially performing two steps simultaneously: first, calculating the robust central measure for the specified column(s), and second, passing that result directly into the imputation function. This technique ensures that every missing entry is replaced by a statistically sound estimate derived from its own population subset, thus preserving the relative order and distributional characteristics of the data as closely as possible.

The use cases for this approach are diverse. Whether you need to fix sparse data in a single feature column, synchronize imputation across a small group of related variables, or apply a generalized imputation strategy across the entire dataset, **fillna()** provides the flexibility required. Below, we detail three distinct structural approaches utilizing this powerful function to handle missing numerical data efficiently.

Three Strategies for Median Imputation

Effective data cleaning often requires targeted strategies based on the scope of the missingness. Depending on whether you have sporadic missing values across many columns or concentrated

missingness in just a few, the syntax of **fillna()** needs adjustment. We categorize the methods based on their application scope--single column, multiple specified columns, or all numerical columns in the DataFrame.

Here are three common ways to use this function for median imputation:

Method 1: Fill NaN Values in One Column with Median

This approach is used for precise handling of missingness in a single, isolated feature. It is the safest method when imputation logic differs across columns.

df = df.fillna(df.median())

Method 2: Fill NaN Values in Multiple Columns with Median

When several columns exhibit similar data characteristics (e.g., all are scores or all are counts) and require the same imputation strategy, this vectorized approach is more efficient.

df] = df].fillna(df].median())

Method 3: Fill NaN Values in All Columns with Median

This is the broadest application, suitable when all numerical features in the DataFrame are appropriate for median imputation. Pandas handles the calculation and mapping automatically.

df = df.fillna(df.median())

Setting Up the Sample Data Scenario

To demonstrate the practical application of these three methods, we will establish a sample DataFrame containing several numerical features and strategically placed NaN values. This mock dataset represents common scenarios encountered in empirical data analysis, such as athlete performance statistics, where some metrics might be missing due to incomplete reporting or recording errors.

We rely on the **NumPy** library to represent missing data points explicitly as `np.nan`, and **Pandas** to manage the data structure. Observe the initial state of the dataset; notice that the `rating`, `points`, and `assists` columns all contain at least one missing observation that requires imputation.

The following Python script initializes the necessary libraries and constructs the base DataFrame

used throughout the subsequent examples:

import numpy as np
import pandas as pd

#create DataFrame with some NaN values
df = pd.DataFrame({'rating': ,
'points': ,
'assists': ,
'rebounds': })

#view DataFrame
df

rating points assists rebounds
0 NaN 25.0 5.0 11
1 85.0 NaN 7.0 8
2 NaN 14.0 7.0 10
3 88.0 16.0 NaN 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Example 1: Targeted Imputation for a Single Column (Rating)

When dealing with large datasets, it is often necessary to apply imputation only to specific features identified as having crucial but missing data points. In this example, we focus solely on the `rating` column. This column has two **NaN** entries that must be imputed using the median of the existing eight observed ratings.

The process involves selecting the column, calculating its median using `df.median()`, and then feeding that scalar result directly into the `.fillna()` method applied back to the same column. This ensures that the replacement value is contextually accurate for the feature being modified, rather than an arbitrary static number.

The following code shows how to fill the NaN values in the **rating** column with the median value of the **rating** column, updating the DataFrame in place:

#fill NaNs with column median in 'rating' column
df = df.fillna(df.median())

#view updated DataFrame
df

rating points assists rebounds
0 86.5 25.0 5.0 11
1 85.0 NaN 7.0 8
2 86.5 14.0 7.0 10
3 88.0 16.0 NaN 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Upon reviewing the updated DataFrame, we can confirm that the median value calculated for the **rating** column (which was **86.5**) has successfully replaced the two **NaN** entries at index 0 and 2. Crucially, notice that the missing values in the `points` and `assists` columns remain untouched, as this method was scoped strictly to the `rating` feature.

Example 2: Efficient Imputation Across Multiple Columns (Rating and Points)

In scenarios where multiple numerical features share a similar underlying distribution and statistical properties, applying median imputation simultaneously across these columns simplifies the code and improves execution efficiency. This approach utilizes Pandas' capability to apply calculations and operations to a subset of columns defined by a list of column names.

For this demonstration, we will address the remaining missing values in the **points** column, alongside the **rating** column (restarting the operation on the base data to illustrate the method cleanly). We calculate the median for both columns simultaneously, resulting in a Pandas Series containing two median values, one for `rating` and one for `points`. This Series is then passed to `.fillna()`, which intelligently maps each median to the corresponding column's **NaN** entries.

The following code shows how to fill the NaN values in both the **rating** and **points** columns with their respective column medians:

#fill NaNs with column medians in 'rating' and 'points' columns

df] = df].fillna(df].median())

#view updated DataFrame
df

rating points assists rebounds
0 86.5 25.0 5.0 11
1 85.0 16.0 7.0 8
2 86.5 14.0 7.0 10
3 88.0 16.0 NaN 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

In the resulting table, the `rating` column has its **NaNs** replaced by 86.5 (the median rating), and the `points` column has its **NaN** replaced by 16.0 (the median points value). This simultaneous imputation is effective because `df].median()` returns a Pandas Series indexed by the column names, allowing **fillna()** to apply the correct median to the correct column automatically. Note that the missing value in the `assists` column still persists, as it was not included in our list of target columns.

Example 3: Broad Imputation Across the Entire DataFrame

For datasets where median imputation is deemed appropriate across all numerical features--perhaps after a thorough exploratory data analysis confirming the presence of outliers or skewed distributions in every column--the most concise method is to apply `.median()` directly to the entire DataFrame. When `df.median()` is called without specifying an axis, it calculates the median for every single numerical column (`axis=0` by default).

This results in a Series containing the median for every eligible column (excluding non-numerical types like strings or categorical data). When this Series is passed to `df.fillna()`, Pandas handles the mapping based on column names, ensuring that each column's **NaN** values are filled with its own respective median, a process often termed "column-wise imputation."

The following code shows how to fill the NaN values in each column with their column median, ensuring that the remaining gap in the `assists` column is also addressed:

```
#fill NaNs with column medians in each column
df = df.fillna(df.median())

#view updated DataFrame
df
```

```
rating points assists rebounds
0 86.5 25.0 5.0 11
1 85.0 16.0 7.0 8
2 86.5 14.0 7.0 10
3 88.0 16.0 7.0 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7
```

Notice that the **NaN** values in every column--including the previously unaddressed `assists` column--were filled with their column median. The median for the `assists` column, which had a missing value at index 3, was calculated to be **7.0** (based on the original, non-imputed data). This method is highly efficient for comprehensive data cleaning but requires careful validation to ensure that the median is indeed the most appropriate central measure for every single feature involved.

Why Choose Median Imputation Over Mean Imputation?

While mean imputation is another popular choice for handling missing numerical data, the statistical robustness of the median often makes it the superior choice. The key difference lies in sensitivity to extreme values. The arithmetic mean is heavily influenced by outliers; a single extraordinarily high or low value can significantly shift the mean, causing the imputed value to misrepresent the typical center of the data distribution.

The median, conversely, relies solely on the positional order of the data points and is unaffected by the magnitude of the extreme values. For instance, if a dataset of salaries includes a few CEOs with multi-million dollar incomes, the mean salary will be artificially inflated, whereas the median remains a realistic representation of the typical worker's salary. Therefore, if your exploratory data analysis reveals that your numerical columns are skewed or contain significant outliers, median imputation is the statistically sounder choice to maintain the integrity of the data's central tendency.

Choosing the correct imputation strategy is crucial for minimizing bias and variance in subsequent

machine learning models or statistical tests. By leveraging the **fillna()** function with `.median()`, Pandas allows data practitioners to implement this robust strategy with high efficiency and minimal effort.

You can find the complete online documentation for the fillna() function on the official Pandas documentation website for further reading and advanced usage scenarios.