Learning Pandas: A Practical Guide to Imputing Missing Values with the Median

Authored by

Mohammed looti

October 31, 2025

Addressing missing data is perhaps the most critical initial phase in the data preprocessing pipeline, essential for any analytical task or machine learning model training. The presence of NaN (Not a Number) values introduces statistical bias, compromises the integrity of results, and can halt model execution. Fortunately, the widely utilized Pandas library in Python provides highly effective tools to manage data imperfections, with the powerful **fillna()** function serving as the primary mechanism for systematic data imputation.

When selecting an imputation strategy, replacing missing values with the **median** stands out as a robust and often preferred choice. Unlike the mean, the median--the middle value of a dataset--is significantly less susceptible to the distorting influence of outliers or observations in highly skewed data distributions. This resilience makes median imputation a statistically sound technique for preserving the underlying shape of the data while ensuring a stable measure of central tendency. This guide provides a comprehensive walkthrough of three practical methods to apply the **fillna()** function using the median across your Pandas DataFrame, moving from targeted column updates to global dataset cleaning.

The Role of fillna() in Robust Data Preprocessing

The **fillna()** function is the cornerstone of missing data handling in Pandas. Its versatility allows developers and analysts to replace missing (NaN) entries using various inputs, including scalar constants, column-specific dictionaries, propagation methods (like forward-fill or backward-fill), or calculated statistical measures. When performing statistical imputation, **fillna()** leverages methods like **.median()**, **.mean()**, or **.mode()** to compute the required statistic exclusively from the available, non-missing values within a specified data series or column, subsequently using that computed value to populate the NaN gaps.

The decision to employ the median over the mean is crucial when dealing with real-world datasets that frequently violate the assumptions of normality. Consider financial or demographic data, where extreme high or low values (outliers) can pull the mean significantly away from the true center of the data. If the mean were used for imputation in such cases, the introduced values would be unrepresentative, potentially leading to misleading analyses. By contrast, the median accurately reflects the 50th percentile, offering a stable central value that remains largely unaffected by these anomalies, thereby making it a more reliable measure for imputation in many scenarios.

The fundamental objective of using **fillna()** paired with column medians is to ensure that the imputed values are statistically conservative. This approach minimizes the risk of injecting unwarranted bias into the dataset. By replacing NaNs with the median of their respective columns, we effectively preserve the original distribution shape of the features, a critical factor when dealing with interval-scaled or ordinal data types that exhibit noticeable skewness.

Preparing the Environment: Creating a Sample DataFrame

To effectively demonstrate the mechanics of median imputation, we must first establish a representative Pandas DataFrame containing strategically placed NaN values. This setup mirrors common challenges encountered during data acquisition where specific data points are simply unavailable or corrupted. Our example will simulate a dataset tracking hypothetical player statistics, allowing us to clearly illustrate how imputation targets and resolves these data deficiencies.

We begin by importing the necessary dependencies: NumPy, which is instrumental for representing missing values using its `np.nan` constant, and Pandas for core DataFrame operations. We will construct a DataFrame featuring numerical columns such as 'rating', 'points', 'assists', and 'rebounds', deliberately embedding NaNs in specific entries to mimic real-world data imperfections.

The initial visualization of this DataFrame is crucial, as it provides a baseline reference, highlighting the exact locations of the missing data before any transformations are applied. Understanding this initial state is key to verifying the success and precision of the subsequent median imputation techniques.

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

Method 1: Precise Imputation for a Single Column

In many data cleaning scenarios, missing values are isolated to one or two key features that require immediate attention. In these instances, the most efficient and least intrusive method is to target a single column for imputation using its self-calculated median. This precise approach ensures that the imputation is entirely localized, preventing accidental modifications to other columns that may either be complete or require a different specialized imputation strategy.

The implementation involves a three-step process: First, isolate the specific column (a Pandas Series) from the DataFrame. Second, compute the median value of that series using the **.median()** method. Third and finally, feed this calculated median value directly into the **fillna()** function applied to the same column. By assigning the result back to the original column, the missing entries are replaced in place.

For our practical example, we will focus exclusively on the 'rating' column, which currently contains two NaN values. We calculate the median of all non-missing ratings and then utilize this single value to fill the gaps. This exercise demonstrates the fundamental concept of column-specific imputation, a highly common and necessary task in data preparation.

#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

The resulting DataFrame clearly shows that the median value for 'rating' was computed as **86.5**, successfully replacing the NaNs in rows 0 and 2. Crucially, the missing values in the 'points' and 'assists' columns remain untouched, confirming that the imputation was applied with the intended precision, preserving the data integrity of the other features.

Method 2: Efficient Imputation Across Selected Columns

When multiple columns within a dataset exhibit missing values, and the median is deemed the appropriate imputation statistic for all of them, processing each column individually becomes inefficient. Pandas allows for a highly streamlined operation where NaN values across a specified subset of columns can be imputed simultaneously using their respective medians. This method significantly enhances code readability and operational speed, particularly when dealing with wide datasets.

To execute this, you must first select the desired columns by passing a list of their names to the DataFrame. Applying the **.median()** method to this multi-column selection instructs Pandas to calculate a separate median for each column in the subset, returning a Series where the index is the column name and the value is its median. By passing this Series of medians into the **fillna()** function, each missing value is correctly matched and replaced by the corresponding column's median.

We will now apply this technique to the 'rating' and 'points' columns. This demonstration shows how Pandas intelligently handles the vectorized operation, calculating the median for 'rating' (86.5) and 'points' (16.0) independently and using those distinct values for imputation within their respective columns. Note that since 'rating' was already imputed in Method 1, this step confirms the replacement of the remaining NaN in 'points'.

#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

Inspection of the output confirms that the NaN in the 'points' column (row 1) has been successfully replaced by its calculated median of 16.0. The 'assists' column, which was not included in this selection, retains its original missing value (row 3), further emphasizing the targeted nature of this multi-column imputation.

Method 3: Comprehensive Imputation for the Entire DataFrame

For datasets where missing values are broadly distributed across numerous numerical features, or when a uniform median imputation strategy is acceptable for all applicable columns, applying the process globally provides the highest level of efficiency. This method automatically identifies all numerical columns and fills their missing entries with their respective median values in a single, concise operation.

The execution is elegantly simple: apply the **.median()** method directly to the entire DataFrame. Pandas automatically returns a Series containing the median for every numerical column. By passing this result to the **fillna()** function applied to the DataFrame, every remaining NaN value is replaced by the calculated median specific to the column it belongs to. This holistic approach is incredibly powerful for completing the data cleaning phase.

We conclude our examples by applying this comprehensive method to our current DataFrame. This step will address the single remaining NaN in the 'assists' column, using the median of that column to achieve a fully cleaned dataset, ready for advanced analysis or modeling.

#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

The final DataFrame confirms that all numerical features are now complete. Specifically, the NaN in the 'assists' column (row 3) has been successfully replaced by the column's median value (7.0). This demonstrates the efficiency and thoroughness of applying median imputation globally, resulting in a fully prepared and robust dataset for subsequent analytical tasks.

Conclusion and Best Practices for Median Imputation

Effective handling of missing data is indispensable for reliable data analysis, and the Pandas **fillna()** function provides a powerful and adaptable solution. Replacing NaN values with the column median is a statistically sound strategy, offering superior stability compared to the mean, particularly when datasets contain extreme outliers or exhibit significant skewness.

We have successfully navigated three distinct methods for median imputation: **Method 1** utilized precise, column-specific imputation; **Method 2** efficiently targeted a selected subset of columns; and **Method 3** performed a comprehensive, global replacement across the entire numerical DataFrame. The selection of the appropriate method should always be guided by the specific structure of the missing data and the analytical goals of the project.

While median imputation is robust, analysts should remain mindful of its limitations. It is a univariate technique, meaning it imputes values based solely on the column itself, ignoring potential predictive relationships with other features. For more complex data scenarios or when needing alternatives like forward-fill, backward-fill, or interpolation, consult the **official Pandas documentation** for a complete overview of the **fillna()** function's capabilities.

Additional Resources

To further expand your expertise in data manipulation and cleaning using Pandas, the following official resources provide deeper insights into advanced techniques essential for robust data preprocessing:

Pandas documentation on Handling Missing Data
The Pandas Cookbook featuring various Data Imputation Strategies
Guides for optimizing DataFrame performance and memory usage