Pandas Tutorial: Handling Missing Data by Imputing NaN Values with the Mean

Authored by

Mohammed looti

October 31, 2025

Introduction: Mastering Missing Data Imputation with Pandas

In the critical stages of data analysis and data science workflows, encountering missing values is nearly unavoidable. These gaps in data, frequently denoted as **NaN** (Not a Number), pose a significant threat to the validity and trustworthiness of subsequent modeling and analysis if left unaddressed. The **Pandas** library, a fundamental tool in the Python data ecosystem, provides powerful, streamlined functions designed specifically for managing and resolving such data deficiencies.

One of the most widely accepted and simple methods for treating missing numerical data is **data imputation**, which involves substituting the missing entries with a statistically derived estimate. Among the various imputation techniques, replacing **NaN** values with the **mean** (average) of the column is a common, effective choice. This strategy is particularly suitable when the data distribution is relatively symmetric and helps maintain the overall central tendency of the feature while ensuring data completeness.

This comprehensive article serves as a practical guide to utilizing the robust **`fillna()`** function within **Pandas** to systematically replace **NaN** entries using the column **mean**. We will progress through three distinct, practical scenarios, illustrating how to apply this technique with precision: first to a single column, then to a specific subset of columns, and finally, globally across all relevant columns within a **DataFrame**. Each method is accompanied by clear, executable code examples.

Understanding the Pandas `fillna()` Function for Statistical Imputation

The **`fillna()`** function stands as an essential component in the **Pandas** toolkit for effectively handling **NaN** values. Its primary purpose is to replace missing entries with a value or method explicitly defined by the user. When employing mean imputation, we harness **`fillna()`** by providing it with the calculated **mean** of the respective **Series** (column) or the means calculated across the entire **DataFrame**.

The core syntax of **`fillna()`** is highly adaptable. It accepts various inputs for replacement, including a simple scalar value, a dictionary mapping specific columns to specific replacement values, or, crucial for our purpose, a **Series** or **DataFrame** of calculated means. By computing the **mean** of the relevant column(s) using the **`.mean()`** method and supplying that result to **`fillna()`**, we ensure that every **NaN** is replaced by a statistically grounded value relevant to its specific feature.

A key aspect of this process is understanding the context in which **`fillna()`** operates. When applied to a single **Series** (e.g., `df`), it uses a scalar mean value for replacement across that column alone. Conversely, when applied to a full **DataFrame**, **`fillna()`** intelligently uses the column-wise averages generated by **`.mean()`** (axis=0 by default), ensuring that the missing value

in 'Column A' is replaced by the mean of 'Column A', and so on for all relevant columns.

Setting Up Our Example DataFrame with Missing Data

To effectively demonstrate the varied applications of the **`fillna()`** function, we must first establish a representative sample **DataFrame**. This structure is designed to mimic real-world datasets that contain inherent missing entries, providing a practical testing ground for our imputation techniques. We will rely on the **NumPy** library to generate the necessary **NaN** indicators within our data structure.

Our hypothetical **DataFrame** captures performance statistics, featuring numerical attributes such as **rating**, **points**, **assists**, and **rebounds**. By intentionally inserting **NaN** values into certain records, we simulate common data collection failures or incomplete observations. Our fundamental objective throughout the following examples will be to accurately fill these missing spots using the calculated **mean** of their respective feature columns.

The Python code provided below initializes this sample **DataFrame**. Following the creation snippet, we display the resulting data structure, clearly highlighting the locations of the **NaN** values. This foundational setup will be crucial for understanding how each subsequent imputation method modifies the data.

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 87.00 14.0 9.0 10
9 86.00 19.0 5.0 7

Example 1: Targeted Imputation of NaN Values in a Single Column

Our first practical demonstration focuses on applying mean imputation to a single, chosen column. This highly specific requirement is frequent in data cleaning when the integrity of other features must be strictly preserved. For this example, we will concentrate solely on the **'rating'** column, which presently contains two **NaN** entries that require careful handling.

The methodology involves two clear steps. First, we calculate the **mean** of the **'rating'** column using the expression `df.mean()`. It is important to note that the **Pandas** `.mean()` method automatically excludes existing **NaN** values from the calculation, providing a true average of the available data points. Second, we pass this computed scalar **mean** into the `fillna()` function applied directly to the **'rating' Series**. This operation ensures that all missing values in that specific column are replaced by the calculated average.

The following code block executes this targeted imputation. Upon viewing the updated **DataFrame**, you will observe that the **NaN** values located at index 0 and 2 in the **'rating'** column are successfully replaced by the calculated mean of **85.125**. This demonstrates a precise and effective way to fill missing data when column isolation is necessary for data integrity.

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

#view updated DataFrame
df

rating points assists rebounds
0 85.125 25.0 5.0 11
1 85.000 NaN 7.0 8
2 85.125 14.0 7.0 10
3 88.000 16.0 NaN 6
4 94.000 27.0 5.0 6
5 90.000 20.0 7.0 9
6 76.000 12.0 6.0 6
7 75.000 15.0 9.0 10
8 87.000 14.0 9.0 10
9 86.000 19.0 5.0 7

As confirmed by the output, the two initial **NaN** values within the **'rating'** column have been accurately substituted with **85.125**, the calculated **mean** of the non-missing entries for that feature. This successful implementation confirms that only the intended column was modified, paving the way for more complex imputation scenarios.

Example 2: Filling NaN Values Across Multiple Specific Columns with Respective Means

Data preprocessing often requires addressing missing values distributed across several, but not all, features within the **DataFrame**. Applying a consistent imputation strategy to a select subset of columns, while leaving others untouched, is a powerful technique for maintaining data structure control. This scenario is ideal when you have a predefined list of features that necessitate mean imputation.

To efficiently fill **NaN**s across multiple columns simultaneously, we first select these columns using list notation, creating a sub-**DataFrame** (e.g., `df]`). We then call the `.mean()` method on this subset. Crucially, when `.mean()` is applied to a sub-**DataFrame**, it computes the **mean** for each selected column independently, generating a **Series** of means. This resulting **Series** is passed to the `fillna()` function, ensuring that each missing value is replaced by the specific average of its corresponding column.

In our current example, the **'points'** column still contains a **NaN** value at index 1. The following code demonstrates how to target both the **'rating'** (even though it was previously filled, we can re-run the logic) and **'points'** columns. This process will calculate the mean for **'points'** (which is 18.0) and use it to replace the missing entry, while confirming the replacement logic for **'rating'**.

#fill NaNs with column means in 'rating' and 'points' columns
df] = df].fillna(df].mean())

#view updated DataFrame
df

rating points assists rebounds
0 85.125 25.0 5.0 11
1 85.000 18.0 7.0 8
2 85.125 14.0 7.0 10
3 88.000 16.0 NaN 6
4 94.000 27.0 5.0 6
5 90.000 20.0 7.0 9
6 76.000 12.0 6.0 6
7 75.000 15.0 9.0 10

8 87.000 14.0 9.0 10
9 86.000 19.0 5.0 7

The result confirms the successful imputation: the **NaN** entry in the **'points'** column at index 1 has been replaced by its calculated column **mean** of **18.0**. This technique offers a clean, scalable solution for handling missing data across multiple specific features without relying on manual dictionary creation or iterative looping.

Example 3: Filling NaN Values in All Numeric Columns with Their Respective Means

For large-scale datasets, manually identifying and imputing missing values column by column is inefficient. In situations where every numeric column containing missing data should be imputed using its own **mean**, **Pandas** provides an elegant, one-line solution that automatically handles all necessary computations. This global approach is highly efficient for comprehensive data cleaning.

The simplicity of this method lies in applying the `fillna()` function directly to the entire **DataFrame** and supplying the complete result of `df.mean()` as the replacement value. When executed, **Pandas** calculates the **mean** for every numeric column and uses that specific mean to fill any **NaN**s residing within that column. This process intelligently skips any non-numeric columns, preventing datatype errors and ensuring a streamlined operation.

We now apply this method to our sample **DataFrame** to address the final remaining **NaN** value, which is located in the **'assists'** column (at index 3). The following code executes the global mean imputation, ensuring that all numerical columns are thoroughly cleaned of missing entries.

#fill NaNs with column means in each column
df = df.fillna(df.mean())

#view updated DataFrame
df

rating points assists rebounds
0 85.125 25.0 5.000000 11
1 85.000 18.0 7.000000 8
2 85.125 14.0 7.000000 10
3 88.000 16.0 6.666667 6
4 94.000 27.0 5.000000 6
5 90.000 20.0 7.000000 9
6 76.000 12.0 6.000000 6

7 75.000 15.0 9.000000 10
8 87.000 14.0 9.000000 10
9 86.000 19.0 5.000000 7

Reviewing the fully updated **DataFrame** reveals that the **NaN** value in the **'assists'** column has now been replaced by its calculated column **mean**, which is approximately **6.666667**. This final example demonstrates the most comprehensive and scalable method for ensuring that all numerical features are free of missing values using the statistical average.

Conclusion and Best Practices for NaN Imputation

The management of **NaN** values constitutes an indispensable phase in modern data preprocessing. The **`fillna()`** function in **Pandas** offers exceptional flexibility, enabling data scientists to choose between highly targeted imputation (Example 1), multi-column imputation (Example 2), or a broad, global approach (Example 3) across the entire **DataFrame** structure.

While mean imputation is a straightforward and widely implemented technique, it is vital to acknowledge its statistical limitations. Replacing missing entries with the **mean** is most effective when the underlying data is approximately normally distributed and the missingness is considered random (Missing Completely At Random, or MCAR). Conversely, if the data exhibits significant skewness or is contaminated by extreme outliers, utilizing the **median** should be favored, as this measure of central tendency is inherently more robust against the influence of anomalous values.

A rigorous data workflow always begins with a deep exploration of the data's characteristics and the patterns of missingness. Visualizing the distribution of missing data and understanding the domain context are crucial steps that inform the optimal imputation strategy. Although simple mean imputation serves as an excellent foundational technique, complex datasets often require sophisticated methods such as regression imputation, K-Nearest Neighbors (KNN) imputation, or even specialized machine learning models to accurately estimate missing values, thereby preserving inter-feature relationships and minimizing statistical bias.

For those seeking to explore other functionalities and advanced parameters, the comprehensive online documentation for the **`fillna()`** function is available on the official **Pandas** website, providing an essential reference for all data manipulation tasks.

Additional Resources

To further enhance your proficiency in data manipulation and the critical task of handling missing data within **Pandas**, the following related resources and documentation are highly recommended:

Pandas User Guide: Working with Missing Data

GeeksforGeeks: Pandas DataFrame fillna()

DataCamp: How to Handle Missing Data in Python Pandas

Analytics Vidhya: Data Imputation Techniques for Missing Values in Python