How to Replace Null Values with the Median in PySpark

Authored by

stats writer

January 2, 2026

The task of handling missing data is a fundamental requirement in almost any data processing pipeline. When working with large datasets, particularly within the distributed computing framework of PySpark, efficient and scalable methods for imputation are essential. One robust statistical technique for dealing with numerical missing data is replacement using the median. This method is particularly favored because, unlike the mean, the median is highly resistant to outliers, providing a more stable estimate for central tendency. In PySpark, achieving this requires a multi-step process: calculating the median across the non-null values for the designated columns and subsequently applying these calculated values using the built-in imputation capabilities.

The standard procedure for filling null values with the median involves leveraging specific functions designed for distributed computation. Specifically, we utilize aggregation functions like `median()` or `approxQuantile()` to determine the central value, followed by the powerful fillna() function to execute the replacement across the DataFrame.

To implement this robust imputation logic efficiently in PySpark, we can define a reusable function. This function abstracts the necessary steps--aggregation and subsequent filling--into a single, callable unit. Below is the structure for defining such a utility, specifically targeting numerical columns within a DataFrame:

from pyspark.sql.functions import median

```
#define function to fill null values with column median
def fillna_median(df, include=set()):
medians = df.agg(*(
median(x).alias(x) for x in df.columns if x in include
))
return df.fillna(medians.first().asDict())

#fill null values with median in specific columns
df = fillna_median(df, )
```

This code snippet showcases how to selectively impute null values. In this particular conceptual example, we are targeting the **points** and **assists** columns, filling their missing entries with their calculated respective column medians. This method ensures that the imputation is performed column-wise, maintaining the statistical integrity of each feature.

Understanding the Necessity of Median Imputation

Before diving into the code specifics, it is essential to understand why the median is often the preferred choice over the mean (average) for imputing missing numerical data. The choice of

imputation strategy directly impacts the statistical characteristics of the resulting dataset.

The primary advantage of the median is its resilience to extreme values or outliers. If a numerical column has a skewed distribution--for instance, a few players with exceptionally high scores in the basketball dataset--the mean would be dragged significantly toward these extremes, leading to a biased imputation value. By contrast, the median, which is the 50th percentile of the data, is less sensitive to these boundary values, offering a more representative central estimate for the missing data point. This makes it a safer and more robust choice when dealing with real-world, often noisy datasets.

In the context of PySpark, where datasets are typically massive and distributed, selecting a robust measure like the median ensures that the imputation step does not introduce unintended statistical artifacts, thereby preserving the quality of the data for subsequent machine learning or analytical tasks. Utilizing the median is a cornerstone of responsible data preparation when distributional assumptions cannot be guaranteed.

Deep Dive into the PySpark Median Calculation

PySpark provides specialized functions optimized for distributed computation. To calculate the median across a DataFrame, we use the built-in median() function, which is part of the `pyspark.sql.functions` module. Unlike calculating the mean, finding the true median in a highly distributed environment like Spark can be computationally expensive as it requires sorting the data. However, Spark's implementation is highly optimized to handle this complexity.

The median calculation typically involves two main steps executed through the custom function defined earlier. First, the `agg()` function is used on the DataFrame to apply the median() function across the specified columns. The Python generator expression `(median(x).alias(x) for x in df.columns if x in include)` efficiently creates the necessary aggregation expressions dynamically for all target columns. This aggregation results in a new, single-row DataFrame containing the median value for each requested column.

Once the aggregated DataFrame of medians is generated, we extract these values into a standard Python dictionary format. This is achieved by calling `medians.first().asDict()`. The `first()` method retrieves the single resulting row, and `asDict()` converts that row into a key-value mapping where the keys are the column names and the values are the calculated medians. This dictionary format is crucial because it is the required input structure for the subsequent imputation step using the fillna() function.

Implementing the Custom Imputation Function: fillna_median()

The custom function, `fillna_median(df, include)`, is designed to streamline the entire

imputation process. It accepts two parameters: the input DataFrame (`df`) and an iterable (such as a list or set) specifying the column names to be imputed (`include`). The body of the function executes the logic described above in a concise and scalable manner.

The core mechanism of the function lies in how it prepares the imputation dictionary and then uses the fillna() function. The `fillna()` method in PySpark is highly flexible; when provided with a dictionary, it automatically replaces null values in the specified columns with the corresponding values provided in the dictionary. For example, if the dictionary is `{'points': 8, 'assists': 4}`, `fillna()` will look for nulls specifically in the 'points' column and replace them with 8, and similarly for 'assists' with 4.

By defining and utilizing this function, we ensure that the logic for calculating the median and applying the fillna() function is encapsulated, promoting clean code and reusability across different data preparation tasks. This approach is superior to manually calculating and applying each column median, especially when dealing with dozens or hundreds of numerical features requiring imputation.

Practical Example: Setting Up the PySpark Environment and Data

To demonstrate the utility of the `fillna_median` function, let us work through a concrete example involving player statistics. We will begin by initializing a Spark session and creating a sample DataFrame that intentionally includes null values in the numerical columns.

The following setup establishes the necessary environment and creates the raw data structure representing basketball players, their teams, and their game statistics. Note the use of `None` in the data list, which PySpark interprets as a null value upon DataFrame creation.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = ,
,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names

```
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| null| 2|
| C| East| 5| null|
+----+----------+------+-------+
```

Upon reviewing the output of `df.show()`, we can clearly observe that the **points** column is missing a value for Team B (West), and the **assists** column is missing a value for Team C (East). Our goal is to calculate the median for the existing non-null values in each respective column and use these medians to fill the corresponding null values, preparing the data for analysis.

Executing the Median Imputation Process

With the sample DataFrame now created, the next step is to apply the custom `fillna_median` function to perform the imputation. This execution phase seamlessly combines the calculation of the median and the replacement of the missing data points in a single, clean operation.

We specifically instruct the function to focus on the `points` and `assists` columns, as these are the only numerical columns containing null values that require treatment. The function first scans the non-null data for these columns to determine their medians. The code below executes the imputation and displays the resulting updated DataFrame:

```
from pyspark.sql.functions import median

#define function to fill null values with column median
def fillna_median(df, include=set()):
medians = df.agg(*(
median(x).alias(x) for x in df.columns if x in include
))
return df.fillna(medians.first().asDict())
```

```
#fill null values with median in specific columns
df = fillna_median(df, )

#view updated DataFrame
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| 8| 2|
| C| East| 5| 4|
+----+----------+------+-------+
```

Analyzing the Results and Verification

The resulting DataFrame clearly confirms the successful imputation. The row originally containing a null value in the **points** column (Team B, West) now shows a value of **8**. This value is precisely the calculated median for all other non-null entries in that column.

Specifically, the null value in the **points** column (non-null values: 11, 8, 10, 6, 5) was replaced with **8**. Similarly, the null value in the **assists** column (non-null values: 4, 9, 3, 12, 2) was replaced with **4**. By using the fillna() function in conjunction with the aggregated medians dictionary, we have successfully completed the missing data treatment in a scalable, distributed manner, preparing the data for robust analysis.

Alternative Imputation Strategies in PySpark

While median imputation is highly effective, especially against outliers, PySpark offers other methods for handling missing data, depending on the data type and the underlying statistical assumptions. Understanding these alternatives provides context for when median imputation is the optimal choice.

Common imputation techniques available in PySpark include:

Mean Imputation: Using the `avg()` function in aggregation, then `fillna()`. This is computationally simpler but susceptible to outliers.
Mode Imputation: Useful for categorical or discrete numerical data, using the `mode()` function or a

combination of `groupBy()` and `count()` to find the most frequent value.
Dropping Records: The simplest approach, using `df.na.drop()`, removes rows containing null values entirely. This is only feasible if the percentage of missing data is very small and dropping rows does not introduce significant bias.

Choosing the median, as demonstrated in this article, is generally the best starting point for numerical feature imputation when the data distribution is unknown or potentially skewed, offering a balance between simplicity and statistical robustness.

Summary and Key Takeaways

Handling null values effectively is a prerequisite for high-quality data science projects. In the PySpark environment, imputing missing numerical entries with the column median provides a reliable and scalable solution that minimizes the impact of potential outliers.

The implementation strategy relies on two crucial steps: using the optimized `df.agg(median(x))` operation to calculate the required central tendencies in a distributed manner, and then utilizing the flexible `df.fillna(dictionary)` method to apply those calculated values precisely to the target columns. By encapsulating this logic within a reusable function like `fillna_median`, data engineers and scientists can maintain clean, efficient, and reproducible data processing pipelines.

The following tutorials explain how to perform other common tasks in PySpark: