Learning PySpark: A Step-by-Step Guide to Imputing Missing Values Using the Median

Authored by

Mohammed looti

November 11, 2025

Understanding Null Values and Data Imputation

When navigating the complexities of large datasets, particularly within a powerful PySpark environment, encountering missing data--typically represented as null values--is an inevitable reality. These gaps, if left unaddressed, can severely undermine the reliability of statistical analysis and lead to catastrophic failures in crucial downstream processes, such as training sophisticated machine learning models. Therefore, the meticulous handling of missing values constitutes a foundational and critical step in any robust data cleaning pipeline. Failing to properly manage these data deficiencies can result in statistically biased outcomes, significantly diminished model performance, and ultimately, inaccurate conclusions drawn from the underlying dataset.

The standardized methodology employed to mitigate the detrimental effects of missing data is formally known as Data Imputation. This systematic procedure involves replacing the identified null values with strategically substituted estimates derived from existing, non-missing data points within the same column or related records. While simpler data scenarios might permit the straightforward deletion of rows containing nulls (listwise deletion), this approach is often highly impractical and detrimental in big data environments. Deletion results in substantial information loss, which is particularly problematic if the missingness pattern is widespread or non-random. Effective imputation strategies, conversely, allow data scientists to retain valuable observations while ensuring the numerical completeness required for complex, distributed calculations.

The selection of the most appropriate Data Imputation technique must be guided by a thorough understanding of the data's inherent nature and its statistical distribution. For numerical attributes, common replacement methods leverage measures of central tendency, including the mean (average), the mode (most frequent value), or the Median. When applying these sophisticated strategies to massive, distributed datasets, computational efficiency is paramount. PySpark is specifically engineered to provide specialized functions that enable the efficient calculation of these aggregate statistics across an entire distributed DataFrame before the replacement logic is applied, ensuring scalability and performance.

Why Choose the Median for Imputation?

The decision to utilize the Median in preference to other central tendency measures, such as the arithmetic mean, is typically a deliberate choice rooted in principles of statistical robustness. The mean is calculated by summing all data points and dividing by the total count, making its magnitude highly susceptible to the influence of extreme values, commonly known as **outliers**. If a dataset exhibits a heavily skewed distribution--for instance, if a few isolated data points are dramatically larger or smaller than the majority--the calculated mean will be disproportionately pulled towards these extremes. This results in a biased imputation value that fails to accurately represent the typical, central data point for that feature.

In sharp contrast, the Median is defined as the value that occupies the 50th percentile of the data; it is the physical center point that separates the upper half from the lower half of an ordered sample. Because the Median relies solely on the rank ordering of values rather than their absolute magnitude, it maintains remarkable stability even when faced with the presence of severe outliers. Replacing a missing null value with the column Median guarantees that the imputed value introduces minimal distortion into the overall statistical distribution of that specific feature. This makes the median a safer, more statistically sound, and generally robust choice for Data Imputation, especially when the underlying distribution of the numerical data is either unknown or suspected to be non-normal.

Successfully implementing median imputation within the PySpark environment necessitates a specific, optimized workflow dictated by the distributed architecture of the DataFrame. It is computationally inefficient, if not impossible, to calculate the median on a row-by-row basis. Instead, we must first execute a large-scale aggregation across the entire dataset to compute the median value for each target column simultaneously. While this initial aggregation step demands significant computational resources, it is essential. Once these medians are computed, they are typically collected into a local dictionary structure and then efficiently passed to PySpark's highly optimized built-in `fillna()` method. This method effectively broadcasts the constant replacement values across all distributed DataFrame partitions, completing the imputation process with high performance.

Defining the PySpark Median Imputation Function

To efficiently manage the systematic replacement of nulls with column medians across multiple numerical columns within a DataFrame, adopting a dedicated function is considered best practice. This functional encapsulation elegantly packages the complex logic required to calculate all necessary medians in a single, highly efficient aggregation step. This crucial approach minimizes expensive data shuffling operations, which are the primary performance bottlenecks in distributed computing. By ensuring that the medians are calculated only once and subsequently utilized for the imputation across all null fields, we guarantee optimal performance and reusability.

The technological cornerstone of this solution lies in the robust aggregation methods provided by the `pyspark.sql.functions` module, specifically leveraging the powerful `median()` function. The functional definition provided below is designed to streamline this entire process: it accepts the input DataFrame and a list of target columns; it calculates the medians for these specified columns using the efficient `df.agg()` method; it converts the resulting single-row DataFrame containing the medians into a standard Python dictionary; and finally, it applies these calculated values using the native `fillna()` method. This methodology is characterized by its clarity, reusability, and optimization for high-volume, large-scale data processing in Spark.

The following standard Python syntax provides the robust, reusable function necessary for calculating and applying the column median to replace null values in any specified numerical columns within your PySpark environment. This code block should be defined once and imported for use across multiple data preprocessing scripts, ensuring consistency and efficiency.

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,)

Within this implementation, the `medians` variable is instantiated as a new, temporary, single-row DataFrame where every column holds the calculated median statistic for its corresponding original column. We then proceed to extract these critical median values into a standard Python dictionary structure by calling `medians.first().asDict()`. This resulting dictionary generates a precise mapping of column names directly to their calculated median values, which is the exact input format required by the PySpark `fillna()` method. This targeted approach allows for efficient, simultaneous imputation across the entire distributed DataFrame. For instance, the operation demonstrated in the code above specifically targets and imputes the missing values within the **points** and **assists** columns.

Practical Example: Implementing Median Imputation

To fully appreciate the practical utility and inherent effectiveness of the median imputation function, let us apply it to a tangible, real-world data scenario. Imagine we are tasked with analyzing a dataset comprising various performance statistics for professional basketball players, all structured within a PySpark DataFrame. Predictably, this dataset contains several missing entries in core performance metrics, which must be rigorously addressed before any meaningful comparative analysis or predictive modeling can commence. Our first steps involve establishing an active Spark session and meticulously defining the initial data structure that contains these nulls.

The sample data presented below serves to illustrate a small, yet representative, DataFrame where null values are present in the key numerical features: **points** and **assists**. This setup accurately mimics common data quality issues encountered in real-world scenarios, often

stemming from data collection errors or incomplete records that result in gaps in numerical features. It is important to observe the explicit use of `None` within the data definition, which the PySpark framework correctly interprets as a structural null value upon the construction of the DataFrame.

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
+----+----------+------+-------+

As clearly illustrated in the output display, the row associated with Team B in the West conference exhibits a significant missing value in the **points** column, and similarly, the record for Team C in the East conference is missing a value in the **assists** column. The presence of these nulls renders standard mathematical or statistical operations on these columns impossible or prone to error. We must now apply our previously defined, optimized `fillna_median` function to perform robust Data

Imputation using the statistically reliable Median value derived from the non-missing entries.

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 of Median Imputation

Upon the successful execution of our custom imputation function, the underlying PySpark framework first performs the necessary cluster-wide calculation to determine the median value exclusively from the non-null records in the designated target columns. For the **points** column, the observed, valid values are . When these values are sorted in ascending order, the sequence becomes . Since we have five total observations (an odd count), the median is unambiguously defined as the middle value, which is **8**. Following the same logic for the **assists** column, the valid values are . When sorted, the sequence is . Consequently, the calculated median for the **assists** column is precisely **4**.

The final output DataFrame definitively confirms that the original null values have been accurately and successfully replaced by these pre-calculated medians. Specifically, the missing null value in

the **points** column (associated with Team B, West) has been substituted with 8, and the null value in the **assists** column (for Team C, East) has been replaced by 4. This robust process guarantees that the statistical integrity of the feature distributions is minimally impacted while effectively resolving the critical issue of data missingness. Crucially, the records now contain valid numerical values, allowing them to be fully included in all subsequent data aggregation, visualization, and machine learning model training steps without generating errors.

This specialized technique, which relies on the strategic pre-calculation and application of medians, is designed to be exceptionally efficient within a distributed computing environment. It successfully bypasses the significant performance degradation associated with traditional, iterative row-by-row processing methods. By fully leveraging optimized DataFrame operations, particularly the use of the `agg()` method for cluster-wide statistical computation followed by the efficient `fillna()` broadcast, we effectively harness the full parallel processing power of the Spark cluster to execute scalable and robust data preprocessing tasks.

Conclusion: Robust Data Preparation in PySpark

Addressing missing data is an absolutely fundamental requisite of high-quality data preparation, and median imputation stands out as a highly robust and reliable methodology suitable for numerical data, especially in scenarios where the potential influence of statistical outliers or skewed distributions is a major analytical concern. The specific methodology meticulously demonstrated throughout this article--defining a reusable function to centrally compute medians across specified columns using `df.agg()` and subsequently applying these constants via `fillna()`--represents the most idiomatic, efficient, and performant way to accomplish this essential task using PySpark. This best-practice approach ensures that your critical data preparation workflows scale seamlessly and effectively, capable of handling petabytes of data without compromising analytical accuracy or quality.

While the Median is strongly recommended for its intrinsic resistance to highly skewed data distributions, it is vital for practitioners to recognize that the optimal strategy for Data Imputation is always context-dependent. Depending on the specific characteristics of your dataset and your ultimate modeling objectives, alternative strategies may prove necessary. For handling missing values in categorical data, for instance, imputation using the mode (most frequent category) is the accepted standard. For sequential data, such as time series, sophisticated techniques like forward fill or backward fill are often more appropriate. Furthermore, for advanced predictive modeling, more computationally demanding approaches like K-Nearest Neighbors (KNN) imputation or complex model-based imputation might be considered, although these require significantly greater computational overhead compared to simple, statistical imputation methods.

Successfully mastering these foundational data cleaning and manipulation techniques within the

PySpark ecosystem is absolutely essential for any data engineer or data scientist operating in the big data domain. The proven ability to efficiently transform, cleanse, and structure distributed data structures directly impacts the overall speed, stability, and predictive accuracy of subsequent analytical pipelines. For comprehensive and deeper exploration of PySpark's extensive data manipulation capabilities, including the full technical documentation for the `fillna()` function and various other optimized statistical methods, practitioners are strongly encouraged to consult the official Apache Spark documentation portal.

Additional Resources

To further enhance your mastery of big data preprocessing using PySpark, the following resources and tutorials explain how to efficiently perform other common, complex tasks: