PySpark: Use fillna() with Specific Columns

Authored by

stats writer

November 16, 2025

RECOMMENDED CITATION

stats writer (2025). *PySpark: Use fillna() with Specific Columns*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=92458

Introduction to Handling Missing Data in PySpark

In the realm of big data processing, data quality is paramount. Missing values, often represented as null values, are a pervasive issue that can severely impact the reliability and performance of analytical models. When working with large-scale datasets using PySpark, efficiently addressing these gaps is a fundamental requirement for successful data preparation. While generalized imputation methods can fill missing values across the entire dataset, real-world scenarios often demand surgical precision, requiring us to target specific columns for replacement based on contextual knowledge of the data types and distribution. This specialized approach ensures that valuable information in other columns remains untouched, preventing unintended data corruption or skewed analyses.

The PySpark framework provides robust tools for this purpose, chief among them being the powerful fillna() method. Unlike its pandas counterpart, the fillna() function in Spark is optimized for distributed computation and offers flexible parameters, allowing data engineers to specify not just the replacement value, but also the exact subset of columns that require transformation. Mastering the correct use of the `subset` argument is key to performing targeted data cleaning operations efficiently across a large-scale DataFrame.

We will explore the definitive methods for leveraging the `subset` parameter within the fillna() function. These techniques allow you to precisely control which columns receive the replacement value, enabling tailored imputation strategies specific to numeric or categorical fields. By focusing the operation, we ensure computational efficiency and maintain the integrity of columns where null values are either intentional or require alternative cleaning methods.

Core Mechanics: Using fillna() with Column Subsets

The power of the fillna() method lies in its versatility. When used without the `subset` parameter, it attempts to replace null values across all eligible columns (typically numeric columns if a numeric replacement is provided, or string columns if a string replacement is provided). However, when the `subset` parameter is explicitly defined, the operation is strictly confined to the named columns, regardless of the overall data type compatibility across the entire DataFrame. This granular control is essential for complex data cleaning pipelines where different columns might require different default values or no imputation at all.

There are two primary ways to specify the target columns using the `subset` argument, depending on whether you are focusing on a single column or multiple columns simultaneously. Understanding the required data structure for the `subset` argument--a string for a single column and a list of strings for multiple columns--is crucial for syntactical correctness and execution success within the PySpark environment. This precision prevents unintended side effects, such as

replacing missing categorical text with a numeric zero or vice-versa, which could corrupt the data schema.

Method 1: Targeted Replacement in One Column

To replace null values exclusively within a single column, we pass the column name as a string to the `subset` parameter. This is the most straightforward and precise application of the method, typically used when a specific column requires a constant default value, such as 0 for missing scores or 'N/A' for missing categorical entries. This isolation capability is vital when applying highly specific business rules to particular data fields.

df.fillna(0, subset='col1').show()

Method 2: Bulk Replacement Across Multiple Columns

When several columns share the same replacement logic--for example, when multiple metric columns must default to zero--it is inefficient to call fillna() repeatedly. Instead, we can pass a Python list containing the names of all target columns to the `subset` parameter. This executes the imputation process simultaneously across all specified fields, maximizing performance and code cleanliness within the PySpark cluster, especially beneficial when operating on wide datasets.

df.fillna(0, subset=).show()

Prerequisite: Setting Up the Sample PySpark DataFrame

Before diving into the specific examples of targeted imputation, we must first establish a sample DataFrame containing various types of missing data. This dataset will mimic typical real-world data structures, including categorical columns (like 'team' and 'conference') and numeric columns (like 'points' and 'assists'), with strategic placement of null values to demonstrate how the fillna() method interacts selectively with different data types when the `subset` parameter is utilized. The following code initializes a Spark session and constructs our source data structure, which represents basic sports statistics.

The structure of this initial dataset is critical for observing the targeted replacement behavior. Note that the 'points' and 'assists' columns, which are intended to be numeric, contain null values. Crucially, the 'conference' column, which is categorical (string type), also contains a null entry. This diversity allows us to verify that when we specify a numeric replacement (like 0) and target a specific numeric column, the categorical columns are correctly ignored, even if they contain missing data. This adherence to data type consistency and the preservation of non-targeted columns is a key feature of reliable data cleaning in PySpark.

```
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| null| 3|
| B| West| null| 12|
| B| West| 6| 4|
| C| null| 5| null|
+----+----------+------+-------+
```


The resulting DataFrame clearly illustrates the missing data points. Specifically, the 'points' column has two nulls, the 'assists' column has one null, and the 'conference' column also has one null. Our subsequent examples will focus on replacing the numeric nulls in 'points' and 'assists' with the constant value 0, demonstrating how the `subset` argument prevents us from inadvertently affecting the 'conference' column, which might require a different imputation method (like mode replacement or categorical labeling). This setup allows for a clear demonstration of the function's precision.

Example 1: Implementing fillna() on a Single Specific Column

When data scientists determine that missing values in a key metric column should be replaced by

a neutral baseline, such as zero, the single-column target method is employed. In this specific scenario, we aim to fill the missing entries in the **points** column exclusively. By restricting the scope of the fillna() operation using the string `'points'` in the `subset` argument, we ensure that only the **points** column is modified, leaving all other columns--including other numeric columns and the categorical `conference` column--completely untouched.

Executing this operation requires defining the replacement value (in this case, `0`) and then specifying the column name within the `subset` parameter. This demonstrates fine-grained control over data manipulation, a necessity when dealing with diverse datasets where a global replacement strategy would be inappropriate. The resulting DataFrame will show a clean, modified `points` column while retaining the original structure and null values in the other fields.

#fill null values in 'points' column with zeros
df.fillna(0, subset='points').show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 0| 3|
| B| West| 0| 12|
| B| West| 6| 4|
| C| null| 5| null|
+----+----------+------+-------+
```

A meticulous review of the output confirms the successful, targeted imputation. The two rows where `points` was previously null now display the value 0. Crucially, we observe two important verification points: first, the categorical null in the `conference` column remains as `null`; and second, the numeric null in the `assists` column also remains as `null`. This confirms that the `subset='points'` instruction strictly governed the scope of the fillna() function, proving the effectiveness of this precise approach in PySpark data cleaning workflows.

Example 2: Applying fillna() to Several Specific Columns Simultaneously

In many preparatory data cleaning tasks, it is common to apply the same imputation rule (e.g., replacement by zero or median) to a defined set of numeric features. This technique streamlines the process and ensures consistency across related variables. For our example, we want to treat both the **points** and **assists** columns uniformly, replacing their respective null values with zero. To achieve this bulk replacement, we pass a Python list containing to the `subset` parameter of the

fillna() function.

Using a list of strings for the `subset` argument instructs PySpark to iterate the replacement logic across all named columns within a single operation. This approach is highly efficient, particularly when dealing with DataFrames containing dozens or hundreds of columns, and it avoids the overhead associated with chaining multiple single-column replacement calls. The zero replacement value is applied only where nulls exist within the boundary defined by the list, providing a scalable solution for feature engineering.

#fill null values in 'points' and 'assists' column with zeros
df.fillna(0, subset=).show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 0| 3|
| B| West| 0| 12|
| B| West| 6| 4|
| C| null| 5| 0|
+----+----------+------+-------+
```

Upon reviewing the output, we confirm that the imputation has successfully occurred in both targeted columns. The null values previously present in the `points` column (rows 3 and 4) are now 0, and the single null value in the `assists` column (row 6) is also now 0. Crucially, just as in the single-column example, the categorical null in the `conference` column remains unaffected. This confirms that even when targeting multiple columns, the fillna() method respects the boundaries set by the `subset` parameter, ensuring that unrelated columns are protected from unintentional data modification and maintaining the required data integrity for downstream analysis.

Advanced Considerations: Different Replacement Values and Data Types

While using `0` as the replacement value is common for initializing missing counts, the fillna() function is highly flexible regarding the value used for imputation. The replacement value can be any constant that is compatible with the target column's data type. For instance, if the target columns were of type string (like `conference`), we would need to pass a string replacement value, such as `'UNKNOWN'` or `'MISSING'`. Attempting to use a numeric replacement value (like 0) on a string column will generally result in an error or a failure to impute, as PySpark enforces strict type compatibility during the replacement process unless explicit casting is performed.

Furthermore, the DataFrame API also allows the use of dictionaries for replacement, which provides an even finer level of control. If different columns require different replacement constants, you can pass a dictionary where keys are column names and values are the specific replacements. For instance, `df.fillna({'points': 0, 'conference': 'UNKNOWN'})` would handle both numeric and categorical null values simultaneously, without requiring the `subset` parameter (as the dictionary implicitly defines the subset). This method is often preferred when managing heterogeneous imputation rules across a single DataFrame transformation step.

Conclusion and Best Practices for Targeted Imputation

Targeted imputation using the `subset` parameter of fillna() is an indispensable technique for robust data cleaning in PySpark. This methodology provides the necessary precision to manage missing data without risking integrity across unrelated features. By specifying columns either as a single string or a list of strings, data engineers can effectively apply consistent replacement rules, such as substituting null values with zeros in numerical features like scores or counts.

For best practices in production environments, always verify the resulting schema and data distribution after performing imputation. While constant value replacement is simple, more sophisticated data cleaning often involves dynamically calculated values, such as the mean, median, or mode of the column. In PySpark, calculating these aggregate statistics requires an initial pass (e.g., using `df.agg()`) and then passing the calculated value into the fillna() function, maintaining the strict requirement that fillna() accepts only a constant value or a dictionary of constant values for the replacement argument itself. This two-step process ensures both accuracy and efficiency in distributed environments.

Key Takeaways and Documentation Reference

Precision Control: Always use the `subset` parameter when you need to limit the scope of null values replacement to specific columns, protecting other data from accidental modification.

Syntax for Single Column: Pass the column name as a string (e.g., `subset='column_name'`).

Syntax for Multiple Columns: Pass the column names as a Python list of strings (e.g., `subset=`).

Data Type Compatibility: The replacement value must be compatible with the data type of the target column(s). Using a numeric replacement (like 0) only works reliably on numeric columns.

If further detailed information or advanced usage scenarios for handling missing data in PySpark are required, consult the official documentation for the fillna() function.