Learning PySpark: Imputing Missing Values with fillna() in Specific Columns

Authored by

Mohammed looti

November 10, 2025

Handling missing data is a **critical prerequisite** in virtually all large-scale data processing workflows, particularly within distributed computing environments like PySpark. When manipulating a DataFrame, encountering incomplete data is inevitable; often, specific fields will contain null values, which can severely compromise subsequent analysis, introduce statistical biases, or even halt production pipelines. Fortunately, PySpark offers specialized, high-performance tools to manage this challenge. The primary tool is the powerful `DataFrame` method, fillna(). This function facilitates targeted data Imputation, allowing users to precisely replace nulls with specified constant values. Crucially, the flexibility of fillna() allows the operation to be restricted to a designated subset of columns, ensuring that unrelated features remain unmodified and data integrity is maintained.

This comprehensive guide provides the exact methodology required to harness the **fillna()** function effectively when the goal is to replace null values solely in selected columns within a PySpark DataFrame. We will systematically explore the two core use cases: targeting a single column and efficiently targeting a list of multiple columns. Mastering these techniques is fundamental for any data professional seeking to perform accurate and non-destructive data cleaning across massive datasets.

The Essential Role of Imputation in PySpark Workflows

In the demanding environment of distributed computing and large-scale data analysis using PySpark, null values signify the absolute absence of data, making them distinctly different from numerical zero or an empty string. Failing to address these nulls proactively can result in substantial analytical errors. For example, if one attempts to calculate the average of a metric column containing nulls, the aggregation functions might either exclude those rows entirely, skewing the resulting average, or potentially lead to downstream errors if the nulls are propagated through complex calculations. Therefore, a strategic and well-considered approach to missing data Imputation is not optional--it is a necessity.

The strategy chosen for filling these gaps--whether using a constant like zero, a statistical measure like the mean or median, or a specific categorical placeholder--must be dictated by the data's domain and the analysis objective. For numerical features where a missing value reasonably implies a count of zero (such as 'missed free throws' or 'zero sales'), replacing nulls with 0 is often defensible. However, for columns representing continuous measurements like 'age' or 'revenue,' replacing nulls with the column's mean or median is typically a safer statistical practice, although the native fillna() method primarily supports constant or dictionary-based replacements.

The true advantage of fillna() within the Apache Spark ecosystem is its inherent efficiency across distributed clusters. When confronting wide datasets with potentially hundreds of features, applying a uniform, blanket imputation across all columns is highly risky and inefficient. For instance, if a numerical zero is inadvertently applied to a string column, it can cause type errors or corrupt the data structure. This critical requirement for precision mandates the use of the `subset` parameter, allowing the user to precisely control which columns are modified during the cleaning process, thereby preserving the integrity of all unrelated data features.

Targeted Imputation: Controlling Column Selection with the `subset` Parameter

The fundamental mechanism for executing precise null handling within a PySpark DataFrame is the `.fillna()` method. For targeted operations, this method requires two core arguments: the replacement value and the `subset` parameter, which specifies the columns receiving the replacement. By supplying a list of column names to `subset`, we explicitly instruct Spark to localize the Imputation process strictly to those specified fields.

The syntax for implementing this targeted replacement is intuitive, yet attention must be paid to the data type expected by the `subset` argument. When the requirement is to target only one specific column, the `subset` parameter expects the column name as a simple Python string.

Method 1: Imputing Nulls in a Single Column

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

Conversely, in scenarios where several distinct columns must have their null values replaced by the identical constant (e.g., 0, 'Unknown', or -1), the `subset` parameter must be provided with a Python list. This list must contain the string names of all target columns. This multi-column application is extremely common in data preparation, especially when standardizing the initial state of multiple numerical features that suffered from intermittent data collection failures.

Method 2: Imputing Nulls Across Multiple Columns

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

Preparing the PySpark Demonstration DataFrame

To effectively illustrate the precise functionality of targeted fillna(), we first need to construct a robust sample PySpark DataFrame. This sample dataset is intentionally designed to contain diverse missing entries across both numerical (integer) and categorical (string) columns. This heterogeneity is essential for demonstrating the isolated impact of the `subset` parameter on specific features.

Our demonstration DataFrame simulates sports statistics, incorporating four columns: `team` (string), `conference` (string, containing one null entry), `points` (integer, with two null entries), and `assists` (integer, featuring one null entry). This mixed missingness pattern enables us to rigorously test the ability of the `fillna()` function to selectively target the numerical columns for replacement while leaving the categorical fields untouched, a common requirement in real-world data cleaning.

The following Python script initializes the Spark session, defines the input data containing the intentional nulls, specifies the column schema, and creates the resulting DataFrame. Reviewing the output display is crucial for understanding the initial state of the missing data before any imputation takes place.

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|
+----+----------+------+-------+
```

Practical Example: Imputing Missing Values in a Single Column

Our first practical application isolates the use of the fillna() method to the `points` column exclusively. Given that `points` is a numerical performance metric, replacing missing entries with the integer 0 is a statistically sound imputation strategy if the absence of data implies an actual score of zero. The critical component here is the precise specification of the `subset` argument, which must receive the column name, `'points'`, as a single string literal.

By executing the following syntax, we command PySpark to scan only the `points` column for null values and substitute them with 0. It is vital to observe that the `conference` and `assists` columns, despite both containing nulls in the original dataset, will be entirely unaffected by this specific operation. This preservation is essential for allowing us to apply distinct, context-appropriate handling methods later (e.g., filling categorical nulls with 'Unknown' rather than 0).

#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|
+----+----------+------+-------+
```

The output clearly confirms that the two null entries previously found in the `points` column have been successfully converted to zeros. Conversely, the null entry in `conference` (for team 'C') and the null entry in `assists` (also for team 'C') remain exactly as they were, validating the strict, column-specific nature enforced by the `subset` parameter. This isolated application ensures that data cleaning steps are highly controlled and non-interfering, which is paramount in robust data engineering.

Practical Example: Simultaneous Imputation Across Multiple Numerical Columns

A frequent requirement in data preparation involves applying the same constant replacement value to several related numerical features simultaneously. For instance, if both `points` and `assists` are metrics where a null scientifically implies a zero contribution, it is most efficient to apply a single fillna() operation to both columns at once. To execute this, the crucial step is passing a Python list containing the names of all target columns to the `subset` argument.

The following code targets the null values present in both the **points** and **assists** columns, replacing them uniformly with the integer 0. This method significantly streamlines the code, guarantees consistency across these related features, and, importantly, leaves other columns--such as `conference`, which holds categorical null data--completely unaffected, thereby reserving them for specialized, string-based imputation logic.

#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|
+----+----------+------+-------+
```

Reviewing this output confirms that all nulls in the combined set of specified columns--`points` and

`assists`--have been replaced by 0. Crucially, the categorical null in the `conference` column, which was explicitly excluded from the `subset` list, remains intact. This validates the effectiveness of using a list within the `subset` parameter for powerful, simultaneous, yet highly targeted data Imputation across multiple features, ensuring maximum efficiency while adhering to strict data governance standards.

Best Practices and Advanced Alternatives to Constant Imputation

While replacing null values with a constant (like 0) using fillna() is an excellent starting point for basic data cleansing, it is vital to recognize this method as only one component of the comprehensive PySpark data preparation toolkit. The most important decision is the choice of the replacement value itself. We utilized 0 in our examples for simplicity and its appropriateness for count data, but for columns with complex distributions or different data types, more sophisticated methods are often necessary.

For instance, if a numerical feature exhibits a heavily skewed data distribution, replacing nulls with the column's mean can introduce undesirable bias. In these scenarios, the median is generally preferred as a more robust statistical measure for Imputation. PySpark supports these advanced statistical replacements through the `Imputer` class, available in `pyspark.ml.feature`. The `Imputer` can efficiently calculate statistics (mean, median, or mode) across several columns and subsequently use those calculated values to fill nulls, offering a statistically superior alternative to simple constant replacement. Furthermore, for categorical columns (like `conference`), replacing nulls with a descriptive constant string such as "Unknown" or "Missing Category" is far more appropriate than attempting to fill them with a numerical value.

A fundamental concept in PySpark is that the `.fillna()` operation, like nearly all `DataFrame` transformations, is immutable--it returns a new DataFrame rather than modifying the original in place. Consequently, to ensure the persistence of the imputation changes, the result of the `.fillna()` call must always be explicitly assigned back to a variable (e.g., `df = df.fillna(0, subset=)`). Best practice dictates thoroughly analyzing the underlying reasons for the missing data before finalizing an imputation strategy, as alternative actions, such as dropping rows (using `dropna()`), might be the better choice if the null percentage is low and the missingness is confirmed to be completely random.

Note 1: We strategically chose the integer 0 for null replacement in our numerical examples for clarity. However, practitioners possess the full flexibility to utilize any appropriate replacement value (e.g., a specific string, a calculated statistical measure, or a different integer) based on the

domain knowledge and specific data requirements.

Note 2: The complete official documentation for the PySpark **fillna()** function can be found here.

Conclusion: Mastering Precision in PySpark Data Cleansing

Gaining mastery over the targeted application of the `.fillna()` function in PySpark is an indispensable skill for effective large-scale data cleansing operations. By skillfully utilizing the `subset` parameter, data professionals can achieve highly efficient management of missing data across specific columns, guaranteeing that the Imputation process is both exceptionally precise and non-destructive to the integrity of the remaining DataFrame structure. Whether your task involves simple constant replacement for a single numerical feature or simultaneous cleansing across a list of related columns, the methods detailed in this guide provide the necessary control and robustness.

As you advance, we highly recommend exploring more sophisticated imputation techniques, such as conditional replacement (e.g., imputing based on aggregated group statistics) or leveraging machine learning models for predictive Imputation. These advanced methodologies invariably build upon the solid foundational techniques provided by the basic, yet powerful, **fillna()** function.

Additional Resources

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