How to Fill Null Values in a PySpark DataFrame with the Mean

Authored by

stats writer

February 5, 2026

The Necessity of Handling Missing Data in PySpark

PySpark is recognized as an indispensable framework for scalable data analysis and complex data manipulation tasks within the Python ecosystem. Leveraging the distributed computing power of Apache Spark, PySpark allows data scientists and engineers to process massive datasets efficiently. However, real-world datasets are inherently imperfect, often containing missing information. These gaps, commonly represented as null values, pose a significant challenge to the reliability and accuracy of any subsequent analysis or machine learning model training. Effective strategies for handling these missing entries are paramount for maintaining the integrity of the data pipeline.

Ignoring or improperly managing null values can lead to biased statistical results, diminished predictive performance, and erroneous conclusions. While there are various techniques for dealing with missing data--such as complete case analysis (dropping rows) or advanced modeling--imputation remains one of the most practical and widely adopted methods. Imputation involves replacing the missing data point with a substitute value derived from the existing data. When dealing with numerical features, substituting the null value with a measure of central tendency, such as the mean, is often the simplest and most effective starting point, particularly when the data distribution is roughly symmetrical.

This approach is particularly streamlined within PySpark, which offers highly optimized functions for aggregation and transformation across distributed data partitions. Although PySpark's native `fillna()` function can handle constant values, replacing nulls with a calculated statistical measure like the column mean requires a slightly more complex, yet elegant, multi-step process involving aggregation and dictionary mapping. This article will detail an expert-level, reusable method to efficiently compute and apply column means to impute missing data in a PySpark DataFrame.

Understanding Null Value Imputation

Imputation by the mean is a statistical technique where any missing entry (a null value) within a specific column is replaced by the arithmetic average of all non-missing values in that same column. While this technique is straightforward, it requires careful execution in a distributed environment like Apache Spark. When working with a PySpark DataFrame, we cannot simply use a single line of code, as Spark operates lazily and requires an explicit aggregation step to calculate the means before they can be applied back to the original structure.

The core challenge is performing two distinct steps efficiently: first, calculating the required statistic (the mean) across potentially vast datasets, and second, broadcasting those calculated values back across the cluster to fill the missing entries. We must leverage functions from the `pyspark.sql.functions` module, specifically `mean`, to achieve this aggregated calculation. Furthermore, because different columns will have different means, we need a mechanism to map

the correct mean value back to the corresponding column during the imputation step, which is handled gracefully by PySpark's native `fillna()` method when provided with a dictionary.

The PySpark Approach to Calculating the Mean

To perform mean imputation correctly in PySpark, the most robust method involves defining a helper function that handles the aggregation dynamically for multiple columns. This function will first calculate the mean of all specified columns in a single, efficient pass over the DataFrame using the `agg()` method. The `agg()` function allows us to apply multiple aggregation operations concurrently, minimizing data shuffling and maximizing performance, which is crucial for big data tasks.

The syntax below illustrates the necessary import and the foundational logic required to calculate and structure the means. We use a generator expression within the `agg()` call to iterate over the list of target columns, calculate the mean for each, and alias the resulting columns appropriately. This results in a small, single-row DataFrame containing all the required mean statistics, ready for the next imputation phase.

Defining a Custom Function for Mean Imputation

The following defined function, `fillna_mean`, encapsulates the logic for both calculating the necessary means and applying them to the target columns. This abstraction makes the process reusable and clean, allowing data practitioners to call the function easily across various stages of their data analysis workflow.

The `means` calculation step returns a single row DataFrame. To utilize this in the `fillna()` function, we must extract these values into a standard Python dictionary format. This is accomplished using `means.first().asDict()`, which retrieves the first (and only) row of the aggregated results and converts it into a key-value mapping where column names are keys and their respective means are the values. This dictionary is precisely what the built-in `fillna()` function requires to perform column-specific imputation.

The complete, self-contained function definition and its subsequent invocation are shown below. Note how the final line demonstrates filling null values specifically in the **points** and **assists** columns of the target DataFrame using their computed means.

from pyspark.sql.functions import mean

#define function to fill null values with column mean
deffillna_mean(df, include=set()):
means = df.agg(*(

```
mean(x).alias(x) for x in df.columns if x in include
))
return df.fillna(means.first().asDict())

#fill null values with mean in specific columns
df = fillna_mean(df, )
```

Practical Example: Setting Up the Sample DataFrame

To demonstrate the functionality of the `fillna_mean` function, we will first create a sample DataFrame representing basketball player statistics. This dataset includes a few intentionally inserted null values in the numerical columns to simulate real-world data imperfections that require imputation.

We initialize a SparkSession, define the sample data rows, and assign clear column headers: `team`, `conference`, `points`, and `assists`. The `points` column contains a null value for player B, and the `assists` column contains a null value for player C. This initial setup is critical for verifying that the imputation process correctly identifies and replaces these missing entries with the calculated column mean.

The code block below shows the initialization process in PySpark and the resulting DataFrame display, clearly highlighting where the missing data points are located.

```
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
+----+----------+------+-------+

Executing the Mean Imputation Process

With the DataFrame containing null values established, we can now apply the previously defined `fillna_mean` function. This function will calculate the mean for the specified columns--`points` and `assists`--and then return a new DataFrame where the null entries have been replaced.

For the `points` column, the non-null values are 11, 8, 10, 6, and 5. The sum is 40, and there are 5 non-null entries. Therefore, the calculated mean is 40 / 5 = 8. For the `assists` column, the non-null values are 4, 9, 3, 12, and 2. The sum is 30, and there are 5 non-null entries. The mean for assists is 30 / 5 = 6.

The execution block below demonstrates running the imputation function and displaying the final, cleaned DataFrame. Observe how the `null` indicators in the output have been replaced by the calculated mean values, 8 and 6, ensuring that the dataset is now ready for robust statistical modeling or further data analysis.

from pyspark.sql.functions import mean

#define function to fill null values with column mean
deffillna_mean(df, include=set()):
means = df.agg(*(
mean(x).alias(x) for x in df.columns if x in include
))
return df.fillna(means.first().asDict())

#fill null values with mean in specific columns
df = fillna_mean(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|      6|
+----+----------+------+-------+
```

Verifying the Imputation Results

A direct comparison between the initial and final DataFrames confirms the success of the imputation strategy. For the row corresponding to player B in the West conference, the original null value in the `points` column has been accurately replaced with **8**. This value is mathematically verified as the arithmetic mean of all other recorded point totals (11, 8, 10, 6, 5).

Similarly, for the row corresponding to player C in the East conference, the missing entry in the `assists` column has been filled with **6**. This demonstrates the efficiency of using the custom function combined with PySpark's distributed aggregation capabilities to ensure data completeness across the entire DataFrame. Utilizing a tailored function like `fillna_mean` is far superior to performing manual calculations or relying on less performant, iterative solutions, particularly when managing large-scale data processing on Apache Spark clusters.

Conclusion and Further Resources

Handling missing data is a fundamental prerequisite for high-quality data analysis. By leveraging the power of PySpark and defining a reusable function that calculates and applies column means across a distributed DataFrame, data professionals can ensure their datasets are robust and ready for modeling. This method provides an efficient, scalable solution for mean imputation, overcoming the common hurdle of integrating aggregation results back into the original dataset structure.

While mean imputation is effective for many numerical features, it is important to remember that it relies on the assumption of missing data being Missing At Random (MAR). For skewed distributions or sensitive applications, alternative imputation methods, such as median imputation or model-based imputation, might be necessary. Nonetheless, the architecture presented here--

using `agg()` to calculate statistics and `fillna()` with a dictionary--forms a fundamental pattern for advanced data preparation tasks in PySpark.

For those interested in exploring alternative data preparation techniques in PySpark, including median calculation or mode imputation, the official documentation provides extensive resources.

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

How to handle categorical data encoding.
Techniques for filtering and selecting data efficiently.
Advanced window functions and group operations.