Learning Guide: Handling Missing Data in PySpark with Mean Imputation

Authored by

Mohammed looti

November 11, 2025

The Critical Necessity of Handling Missing Data in PySpark Workflows

Data preparation constitutes the foundational stage of any robust machine learning or statistical analysis project. In real-world scenarios, datasets are rarely pristine; they are frequently plagued by missing data, commonly represented as null values. These gaps are not merely inconveniences; they can catastrophically compromise the integrity of analytical outcomes, significantly bias statistical results, and severely degrade the performance and reliability of predictive models. For data professionals working with massive, distributed datasets--the exact scenario where the speed and scalability of PySpark become essential--ignoring missing data is not an option.

PySpark, as the powerful Python API for Apache Spark, is designed for large-scale data processing. It provides the necessary distributed capabilities to handle complex cleaning tasks efficiently. Among the various strategies available for addressing numerical missing values, statistical imputation stands out as one of the most statistically sound and widely adopted techniques. Specifically, utilizing the arithmetic mean for replacement is often the first choice when the feature's distribution is relatively symmetric and the percentage of missing values is low.

Mean imputation operates on a simple principle: calculating the average value of a non-missing feature and substituting every null entry within that column with this calculated average. While straightforward conceptually, implementing this method efficiently across numerous columns within a distributed environment like Spark demands a high-performance, optimized approach. This guide is dedicated to providing a clear, reusable Python function that performs mean imputation across selected columns in a PySpark DataFrame, ensuring scalability and speed, even when dealing with petabytes of information.

The Optimized Syntax for Distributed Mean Imputation

Achieving high performance when filling null values in PySpark requires minimizing the number of data scans. If we were to calculate the mean for each column individually, Spark would execute multiple distributed jobs, leading to significant overhead. The professional solution involves calculating all required column statistics in a single, efficient pass using the powerful `agg()` method. This approach guarantees that the imputation process is scalable and resource-friendly.

The following defined Python function, `fillna_mean`, encapsulates this optimized logic. It is designed to accept a target DataFrame and a list of columns for processing. The function first leverages `agg()` combined with the `mean()` function from `pyspark.sql.functions` to compute the averages of all specified columns simultaneously. The result is a small, single-row DataFrame containing all the means, retrieved in one distributed operation.

from pyspark.sql.functions import mean

```
#define function to fill null values with column mean
def fillna_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, )
```

The critical step following the aggregation is the conversion of the calculated means into a format usable by Spark's built-in imputation tool. We use `means.first().asDict()` to extract the single row of aggregated means and transform it into a standard Python dictionary. This dictionary maps column names (e.g., 'points') directly to their calculated mean value. The PySpark DataFrame's `fillna()` method is expertly engineered to accept this dictionary, intelligently filling null values only in the specified columns with the corresponding statistical average, thereby executing a clean and highly targeted imputation process.

Step-by-Step Implementation Example with Sample Data

To demonstrate this function in action, we will construct a small, representative PySpark DataFrame simulating player statistics. This dataset is intentionally engineered to contain several null entries, accurately reflecting the challenges encountered in real-world data collection where sensor failures, transcription errors, or data corruption lead to data gaps. By setting up this clear scenario, we can precisely illustrate how the `fillna_mean` function restores the completeness of the data structure.

Our first step involves initializing a Spark session, which is mandatory for all PySpark operations. We then define the raw data, including player metrics such as `team`, `conference`, `points`, and `assists`. Crucially, we introduce missing information by setting specific entries to `None`: player 'B' is missing data for `points`, and player 'C' is missing data for `assists`. This setup ensures that we have clearly identified gaps that require imputation.

The following code block executes the data creation process, transforming the raw Python list into a structured PySpark DataFrame. We then use `df.show()` to visualize the initial state of the data, which explicitly highlights the existing nulls before any data cleaning or imputation is attempted.

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

Applying the Imputation Function and Verifying Expectations

As the initial output confirms, both the **`points`** and **`assists`** columns contain a single `null` entry each. Our immediate objective is to invoke the `fillna_mean` function, instructing it to calculate the averages of the non-null values in these specific columns and subsequently replace the missing entries with those calculated averages. This ensures the data set achieves completeness, which is necessary for downstream analytical models that typically require fully populated numerical features.

Before executing the function, it is beneficial to manually calculate the expected statistical averages that Spark will determine in its distributed operation. This step provides a crucial check for verifying the function's accuracy:

For the **`points`** column, the existing non-null values are . The total sum is 40. With 5 valid

observations, the calculated mean is 40 / 5 = 8.

For the **`assists`** column, the existing non-null values are . The total sum is 30. With 5 valid observations, the calculated mean is 30 / 5 = 6.

Based on these calculations, we anticipate that the null in the `points` column will be replaced by 8, and the null in the `assists` column will be replaced by 6. The following code block executes the imputation function by passing the DataFrame `df` and the list of target columns . The resulting, cleaned PySpark DataFrame is then displayed.

from pyspark.sql.functions import mean

#define function to fill null values with column mean
def fillna_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
+----+----------+------+-------+

Analyzing the Success of the Imputation Process

A careful review of the final DataFrame output confirms that the imputation was executed successfully and accurately aligned with our calculated statistical expectations. For player 'B', the missing entry in the **`points`** column has been correctly replaced by the value **`8`**. This confirms the

function calculated the mean from the existing data and applied it precisely where the null was found.

Correspondingly, the `null` entry in the **`assists`** column for player 'C' has been replaced by **`6`**. This outcome powerfully illustrates the core strength of the custom PySpark function: its ability to handle multiple columns concurrently while ensuring that the imputation value used for each column is derived exclusively from the non-null population within that specific feature. This method preserves data integrity and prevents cross-contamination of statistical measures between features.

This technique--using the `fillna()` method paired with a dictionary derived from a single, aggregated calculation--represents a best practice pattern in PySpark. By calculating all required means in one `agg()` operation, we drastically reduce the overhead associated with distributed data processing. This is a far more efficient and scalable solution than using iterative loops or performing sequential aggregations, making this function ideal for production-level data cleansing tasks on large Spark clusters.

Conclusion and Advanced Considerations for Null Value Handling

Mastery of efficient missing data handling, particularly mean imputation for numerical data, is a foundational requirement for any data scientist utilizing PySpark DataFrames. The provided `fillna_mean` function offers a production-ready template that effectively leverages Spark's distributed capabilities to perform statistical calculations quickly and reliably. However, it is essential to contextualize this technique within broader data preparation strategies. Mean imputation assumes the data is missing completely at random (MCAR) and that replacing the nulls with the average will not significantly distort the feature's underlying distribution.

Data professionals must always select their imputation strategy based on a deep understanding of the data's nature and the percentage of missing observations. For instance, if a feature's distribution is highly skewed (e.g., income data), the mean can be heavily influenced by outliers, making the median a more robust and statistically sound choice for imputation. Similarly, for categorical features, mode imputation (replacing nulls with the most frequent category) is the standard method. PySpark readily supports these alternatives, often requiring only minor adjustments to the aggregation expression within the function.

To further enhance proficiency in managing missing data within the Spark ecosystem, data engineers and scientists should explore the full documentation of the **`fillna()`** function, as well as related tools like **`drop()`** for removing incomplete records, and the comprehensive preprocessing transformers available in the Spark ML libraries. These tools collectively form the bedrock for creating clean, robust, and scalable data preparation workflows.

Additional Resources for PySpark Proficiency

To continue building expertise in data manipulation and preparation using the distributed power of PySpark, consider exploring tutorials on the following related topics:

Implementing Mode Imputation for Categorical Features.

Calculating Windowed Averages and Rolling Statistics.

Efficiently Joining Large PySpark DataFrames.

Using the Spark ML library for Feature Scaling and Transformation.