How to Easily Split Your PySpark Data into Training and Test Sets

Authored by

stats writer

January 3, 2026

In the high-stakes environment of distributed machine learning, such as that facilitated by PySpark, correctly segmenting data is a fundamental prerequisite for building robust and reliable predictive models. Splitting the dataset into dedicated training and test sets is a critical step that ensures proper model validation.

This partitioning process guarantees that the predictive model is evaluated on data it has never encountered during the learning phase, which is the gold standard for assessing generalization ability. By doing so, we effectively mitigate the severe risk of overfitting, where a model performs excellently on training data but poorly on novel observations.

The procedure in PySpark is streamlined and highly efficient, relying primarily on the powerful randomSplit() method. This method allows you to partition a massive Spark DataFrame into multiple subsets based on defined weights, ensuring a statistically sound and reproducible separation.

When initiating any supervised learning task, it is standard practice to partition the source dataset into a training set and a test set before model fitting commences.

The randomSplit() Function: Core Mechanism

The most straightforward and efficient mechanism for splitting a dataset into training and test partitions within the PySpark environment is utilizing the randomSplit function, which operates directly on the DataFrame object. This method distributes rows randomly across the specified number of resulting DataFrames based on provided proportionality weights.

The basic syntax demonstrates its simplicity and efficiency. It takes a list of weights representing the proportional distribution and an optional integer seed for ensuring deterministic results across multiple executions:

train_df, test_df = df.randomSplit(weights=, seed=100)

Understanding the parameters is key to mastering this operation. The randomSplit method returns a list of new DataFrames, where the order in the list corresponds directly to the order of the weights provided. For instance, the first weight corresponds to the first resulting DataFrame, typically designated as the training set.

Interpreting Weights and Ensuring Reproducibility

The **weights** argument is a list of floating-point numbers that explicitly define the percentage of observations from the original DataFrame that should be allocated to each resulting subset. These

weights must sum up to a value close to 1.0 (or 100%).

In the canonical example provided above, we designated 0.7 (70%) of the observations to be placed into the training set (assigned to `train_df`) and 0.3 (30%) to be reserved for the test set (assigned to `test_df`). This 70/30 split is a common practice, though ratios like 80/20 or even 60/40 may be employed depending on the dataset size and modeling requirements.

Equally important is the **seed** argument. This integer value serves as the initial state for the random number generator used during the splitting process. Providing a seed ensures that the random assignment of rows to the training or test set remains identical every time the code is executed. This deterministic behavior is crucial for reproducibility, allowing collaborators or future self to replicate the exact model results.

Example: Setting Up the PySpark DataFrame

To illustrate the splitting process, we will begin by creating a typical DataFrame that represents student performance data. This dataset contains numerical variables related to effort and outcome: hours spent studying, the number of preparatory exams taken, and the final score achieved.

The following code snippet initializes a SparkSession and then constructs the sample data, defining the column structure before creating the PySpark 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 first five rows of dataframe
df.limit(5).show()

+-----+----------+-----+
|hours|prep_exams|score|
+-----+----------+-----+
1	1	76
2	3	78
2	3	85
4	5	88
2	2	72
+-----+----------+-----+

Our immediate goal is to prepare this data for a predictive task, such as linear regression, where we intend to use **hours** and **prep_exams** as independent (predictor) variables to forecast the final **score** (the response variable).

Executing the Data Split Operation

Prior to fitting any predictive model, we must proceed with the data partitioning. In this scenario, we adhere to the common practice of allocating 70% of the total observations for model training and reserving the remaining 30% for impartial model testing.

The execution of the randomSplit function is concise, assigning the resulting DataFrames directly to the variables `train_df` and `test_df`. We utilize a fixed seed of 100 to ensure that this specific split configuration is identical upon every run of the code.

#split dataset into training and test sets
train_df, test_df = df.randomSplit(weights=, seed=100)

It is important to note that the weights provided are not strict guarantees of the resulting row counts but rather the expected probabilities of allocation for each row. Due to the nature of random sampling, especially with smaller datasets, the resulting counts may sometimes deviate slightly from the exact mathematical proportion, although they remain statistically representative.

Verifying Dataset Sizes for Validation

After executing the split, the next crucial step is to verify the count of rows in each resulting DataFrame to confirm that the distribution aligns with the intended proportions. This check is performed using the **count()** function, a standard operation available on all PySpark DataFrames.

In our example, the original DataFrame had 20 total rows. Based on the 70/30 split, we expect approximately 14 rows in the training set and 6 rows in the test set. The code below confirms these resulting counts:

#view count of rows in train_df
print(train_df.count())

14

#view count of rows in test_df
print(test_df.count())

6

As shown in the output, the training set (`train_df`) contains 14 rows, which exactly corresponds to 70% of the 20 original rows. The test set (`test_df`) contains 6 rows, accurately representing the remaining 30%. This confirms a successful and proportionate data split.

Inspecting the Split DataFrames

While the row counts confirm the size, it is beneficial to inspect the content of the newly created training and test sets to ensure the integrity of the random partitioning. This involves using the **limit()** and **show()** functions to display the first few records of each subset.

By reviewing the output, we can visually confirm that the rows in the training and testing sets are distinct and represent a random sample of the original student data:

#view first five rows of training set
train_df.limit(5).show()

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

```
|hours|prep_exams|score|
+-----+----------+-----+
| 1| 1| 76|
| 2| 3| 78|
| 2| 3| 85|
| 4| 5| 88|
| 1| 2| 69|
+-----+----------+-----+

#view first five rows of test set
test_df.limit(5).show()

+-----+----------+-----+
|hours|prep_exams|score|
+-----+----------+-----+
| 2| 2| 72|
| 2| 0| 88|
| 4| 1| 94|
| 3| 4| 82|
| 4| 4| 85|
+-----+----------+-----+
```

We have successfully partitioned the original data into a training set and an independent test set, fulfilling the critical data preparation requirement for supervised machine learning.

Conclusion and Next Steps

With the data now correctly segregated, the subsequent steps in the machine learning workflow can commence. The training set (`train_df`) is used exclusively for model fitting and parameter estimation. Once the model is trained, its performance and generalization capability must be assessed using the test set (`test_df`).

By strictly adhering to this methodology--training on one set and testing on a completely separate, unseen set--we ensure a rigorous and unbiased evaluation of the model's effectiveness in a real-world scenario. This practice is foundational to producing reliable predictive models in PySpark.

Note: You can find the complete documentation for the PySpark **randomSplit** function here.

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

How to Select Specific Columns in PySpark
How to Filter Rows in PySpark
How to Perform Joins in PySpark