How to Split Data into Training and Test Sets in PySpark

Authored by

stats writer

February 3, 2026

The process of partitioning raw data into distinct subsets--specifically training and test sets--is foundational to responsible and effective data science. In the environment of big data processing using PySpark, achieving this split efficiently is paramount for developing robust predictive models. This critical division ensures that the model is trained on one portion of the data and subsequently evaluated on unseen data, providing an unbiased estimate of its generalization capability.

For data manipulation within the PySpark ecosystem, the preferred and most straightforward method for splitting a large DataFrame is through the utilization of the **randomSplit** function, which is a member of the `pyspark.sql.DataFrame` class. This function allows developers and analysts to define specific proportional weights for the resulting partitions, thereby controlling the exact size of the training and test samples. Common industry standards often dictate an 80/20 or 70/30 ratio, reserving the smaller portion for rigorous testing.

Implementing a proper split in PySpark ensures that the resulting machine learning model, once trained, can be reliably assessed. The performance metrics derived from the test set reflect how well the model would perform in a real-world production environment when faced with new, previously unobserved data points. Failure to split the data correctly often leads to issues like overfitting, where a model performs exceptionally well on the training data but poorly on any new data.

Executing the Data Split using randomSplit

The **randomSplit** function provides a powerful and simple interface for dividing a PySpark DataFrame based on specified probabilities. This function returns multiple new DataFrames, corresponding exactly to the number of weight proportions provided. The underlying mechanism performs a randomized, proportional sampling of the original data, distributing the rows across the new sets.

To initiate the split, the function is called directly on the source DataFrame. A critical aspect of its usage involves passing a list of floating-point numbers--the weights--which must sum up to 1 (or very close to 1, accounting for minor floating-point errors). For instance, if we aim for a 70% training set and a 30% test set, the weight list would be . The function guarantees that the order of the resulting DataFrames corresponds to the order of the weights defined.

The basic syntax for performing this operation is straightforward, yielding two separate DataFrames that can be immediately utilized for subsequent modeling tasks. The following code snippet illustrates the fundamental application of the function:

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

In this structure, `df` represents the original PySpark DataFrame. The resulting objects, `train_df`

and `test_df`, are the new DataFrames containing the training and test observations, respectively. Understanding the parameters of this function is key to controlling the outcome of the randomization process.

Defining Proportions using the Weights Argument

The primary argument controlling the division of the data is `weights`. This argument accepts a list of decimal proportions that dictate the percentage of rows allocated to each resulting DataFrame. It is essential that these weights are correctly defined, as they directly impact the size and representation of the data used for both learning and validation phases.

For high-quality model training, it is generally recommended to allocate the majority of the data to the training set--typically between 70% and 90%. If the dataset is extremely large, even a smaller percentage reserved for testing (e.g., 10%) might suffice to capture the underlying data distribution and provide robust evaluation metrics. The remaining observations are then used as the unseen data for performance evaluation.

In the common scenario where a 70/30 split is desired, as demonstrated in our example, we explicitly choose to place 70% (0.7) of the total observations into the training set (`train_df`) and the remaining 30% (0.3) into the test set (`test_df`). This distribution ensures a substantial dataset for the model to learn complex patterns while reserving a sufficient portion to rigorously assess its predictive capabilities.

Ensuring Reproducibility with the Seed Parameter

A crucial consideration when performing any randomized operation in data science is reproducibility. Since **randomSplit** relies on a random number generator to assign rows to different partitions, running the code multiple times without control could yield different splits each time. This instability makes debugging and comparison of model performance challenging.

To address this, the optional but highly recommended argument `seed` is utilized. The `seed` argument accepts an integer value that initializes the internal random number generator. By specifying the same seed value--such as `seed=100`--we guarantee that the random split will generate exactly the same subsets of data every time the code is executed, regardless of the cluster configuration or execution environment.

Using a fixed seed is a standard practice in statistical computing. It allows collaborators to replicate results precisely and ensures that any improvements or degradations in model performance across different runs are attributable to changes in the model architecture or hyperparameters, rather than variations in the data partitioning itself. Always use a seed when employing the **randomSplit** function.

Practical Implementation: Setting up the PySpark Environment

To illustrate the practical application of data splitting, we must first establish a working PySpark session and create a sample DataFrame. This setup requires importing necessary modules and initializing the `SparkSession`, which serves as the entry point for all Spark functionality.

We will create a DataFrame containing synthetic student performance data, including variables for study time, number of preparatory exams taken, and the final score achieved. This dataset will serve as a representative example of the kind of transactional or observational data commonly processed in PySpark.

The following Python code defines the data structure, column names, creates the DataFrame, and then displays the initial rows to confirm successful creation:

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

This initial DataFrame, `df`, contains 20 observations, which will now be the subject of our partitioning demonstration. We intend to use the variables `hours` and `prep_exams` as predictor variables to forecast the student's `score`, a classic application suitable for a regression model.

Applying the 70/30 Split to the DataFrame

Once the DataFrame is created, we can proceed directly to the splitting operation. Our goal is to allocate approximately 70% of the rows (14 rows) to the training set and 30% (6 rows) to the test set, ensuring that the model has sufficient data for robust training while retaining a genuine test set.

We apply the **randomSplit** function, passing the desired weights and utilizing the seed value `100` to guarantee consistent results across different executions. The function returns two separate DataFrames simultaneously, minimizing computational overhead.

The code below executes the split and assigns the resulting DataFrames to their respective variables:

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

This single, concise line of code effectively handles the necessary randomization and proportional separation required for high-volume data analysis in PySpark. The resulting DataFrames, `train_df` and `test_df`, are now isolated and ready for the next stages of the machine learning

workflow.

Verifying the Count of Partitioned DataFrames

After performing the split, it is vital to verify that the resulting DataFrames contain the expected number of rows, ensuring that the proportions defined by the `weights` argument were correctly enforced. This verification step confirms the integrity of the data partitioning process.

We achieve this verification by calling the **count()** action on both the training and test DataFrames. This action triggers a Spark job to calculate the total number of records within each partition.

The output confirms the proportional split: the original 20 rows were successfully divided, with 14 rows allocated to training (70%) and 6 rows allocated to testing (30%).

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

14

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

6

The results demonstrate that 14 of the 20 total observations were successfully reserved for the training set, adhering precisely to the specified 70% allocation. The remaining 6 rows form the test set, ready for final model evaluation.

Inspecting the Split Data Subsets

Finally, to confirm the contents and structure of the newly created subsets, we can inspect the first few rows of both the training and test DataFrames. This visual check ensures that both sets retain the original column structure and contain distinct data points, confirming the randomized separation.

Using the **limit(5)** and **show()** methods, we can display the first five records from each set, confirming their independent nature:

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

With the successful execution of the **randomSplit** function, the original dataset has been effectively and reproducibly partitioned into the required training and test sets. This foundational step concludes the data preparation phase and allows for the commencement of model building and performance testing.

We can now proceed to fit the desired regression model to `train_df` and subsequently evaluate its predictive accuracy using the unseen data contained within `test_df`. This methodology ensures valid and reliable model assessment.

Further Resources on PySpark Data Manipulation

For users seeking deeper understanding or alternative partitioning methods, the official documentation for the PySpark **randomSplit** function provides comprehensive details on edge cases and advanced applications.

For those interested in exploring related tasks within the PySpark ecosystem, here is a list of common data manipulation tutorials:

Handling Missing Values in PySpark DataFrames.

Performing Joins and Merges on Large Datasets.
Implementing Feature Engineering Techniques using PySpark SQL functions.