How can I perform linear regression in PySpark with an example?

Authored by

stats writer

February 4, 2026

Linear regression is a foundational statistical and machine learning technique utilized to model and understand the relationship between a target variable, often referred to as the dependent variable, and one or more explanatory or independent variables. Due to its simplicity and interpretability, it remains a cornerstone of predictive analytics across various industries. When dealing with massive datasets, employing distributed computing frameworks becomes essential, and this is where **PySpark** excels, providing scalable tools for fitting complex models efficiently.

In the PySpark ecosystem, linear regression is implemented via the LinearRegression class found within the **pyspark.ml** library. This powerful module is designed for creating, configuring, and training a linear regression model using data stored in a distributed Spark dataframe. The ability to handle large volumes of data makes PySpark the preferred choice for enterprises requiring fast model training and deployment across massive clusters.

This comprehensive guide outlines the step-by-step process required to implement a multiple linear regression model in PySpark. We will cover everything from initializing the Spark session and preparing the raw data to fitting the model and interpreting its performance metrics. Following these steps will provide you with a robust, trained model capable of generating accurate predictions on new, unseen data, showcasing the true power of distributed machine learning.

Perform Linear Regression in PySpark (With Example)

Linear regression is a highly effective method we can use to quantify the relationship between one or more predictor variables and a single response variable. The goal is to establish a linear equation that best describes this relationship, allowing for prediction and statistical inference regarding the impact of the predictors.

The following detailed, step-by-step example demonstrates how to properly set up and fit a multiple linear regression model to a structured dataset using the powerful capabilities of PySpark's machine learning library, specifically focusing on predicting student scores.

Step 1: Defining and Creating the Dataset

The crucial first step in any machine learning workflow is preparing the input data. For our example, we will simulate a dataset representing student performance, aiming to predict the final exam score based on the time spent studying and the number of preparatory exams taken. This scenario allows us to implement multiple linear regression, where we have two independent variables influencing one dependent variable.

To handle this data within the distributed framework, we must first initialize a **SparkSession**, which serves as the essential entry point to using all Spark functionality, including interacting with

dataframes and executing ML pipelines. We then define our data rows and column schema before converting this raw Python structure into a distributed Spark dataframe using the `createDataFrame` method. This DataFrame structure is necessary for PySpark's ML routines to operate efficiently.

The following code block demonstrates the creation of the DataFrame, which contains information on **hours spent studying**, the **number of prep exams taken**, and the resulting **final exam score** for various university students. We cap this setup by displaying the first five rows to confirm the data structure and schema are correct before proceeding with feature engineering.

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

We are aiming to fit the following mathematical representation of a multiple linear regression model to this created dataset. This equation establishes the relationship between the final exam score (Y) and the two predictor variables (hours and prep exams), where $beta_0$ is the intercept and $beta_1$ and $beta_2$ are the coefficients associated with the respective predictors:

Exam Score = β0 + β1(hours) + β2(prep exams)

Step 2: Preparing Data for PySpark ML using VectorAssembler

Before any machine learning model in PySpark's MLlib can be trained, the feature columns--the independent variables--must be combined into a single vector column. This crucial preprocessing step is mandatory because the ML algorithms, including LinearRegression, are designed to accept all input features grouped within one designated column of the vector type, simplifying the internal matrix calculations.

We achieve this necessary transformation using the VectorAssembler utility, which is a key transformer found in the `pyspark.ml.feature` library. The VectorAssembler takes an array of input columns (our numerical predictor variables: 'hours' and 'prep_exams') and aggregates them into a new, single output vector column, which we explicitly name 'features'.

After the feature columns are successfully assembled, we finalize the preparatory stage by selecting only the two required columns for the model training: the newly created 'features' vector column and the target variable, 'score'. This streamlined approach ensures that the data fed into the model is in the precise binary format required by Spark dataframe ML routines, significantly improving the efficiency of the subsequent training process.

from pyspark.ml.feature import VectorAssembler

```
#specify predictor variables
assembler = VectorAssembler(inputCols=, outputCol='features')

#format DataFrame for linear regression
data = assembler.transform(df)
data = data.select('features', 'score')
```

It is important to note the clear distinction made here: we designated **hours** and **prep_exams** as the input columns (predictor variables) that form the 'features' vector, while **score** serves as the label column (the dependent variable or response) that the model is trained to predict and explain based on the variations in the features.

Step 3: Training the Linear Regression Model

With the data correctly formatted into the required 'features' and 'label' columns, we can proceed to initialize and train the LinearRegression estimator. In PySpark MLlib terminology, we first create an instance of the estimator (the algorithm), configure its crucial parameters, and then use the `.fit()` method to train it on our prepared data. The distributed nature of Spark handles the complexity of processing large datasets across the cluster automatically during this fitting process.

When configuring the LinearRegression object, we must explicitly specify the names of the input columns. We set `featuresCol='features'` to point to the vector column created by the VectorAssembler, and `labelCol='score'` to identify our target variable. We also define a `predictionCol='pred_score'`, which specifies the name of the new column that will be generated in the DataFrame when we use the fitted model to make predictions on test or validation data later.

The actual training process is initiated by calling `lin_reg.fit(data)`. This operation utilizes distributed optimization algorithms to efficiently minimize the sum of squared differences (or errors) between the actual scores and the scores predicted by the linear model equation. The output of the `.fit()` method is the final fitted model object, which encapsulates all the necessary parameters, such as the calculated intercept and feature coefficients, that mathematically define the derived linear relationship.

```
from pyspark.ml.regression import LinearRegression

#specify linear regression model to use
lin_reg = LinearRegression(featuresCol='features',
labelCol='score',
predictionCol='pred_score')
```

#fit linear regression model to data
fit = lin_reg.fit(data)

Step 4: Analyzing the Regression Model Summary

Once the model training is complete, the resulting `fit` object provides a comprehensive summary of the model's performance and coefficient statistics via the `.summary` attribute. This summary is absolutely essential for evaluating the model's overall goodness-of-fit and understanding the magnitude, direction, and statistical significance of our predictor variables.

We begin this analysis by extracting the core parameters of the linear equation: the intercept ($beta_0$) and the coefficients ($beta_1$ and $beta_2$). These values mathematically define the optimal line (or hyperplane in multiple linear regression) that best fits the relationship between our study hours, prep exams, and the final score. The intercept represents the baseline expected score when both predictor variables are zero. The coefficients quantify the change in the expected score for every one-unit increase in the respective predictor variable, assuming all other variables are held constant.

The summary statistics also provide crucial information regarding the statistical relevance of the model terms. By examining the p-values associated with each coefficient, we can determine if that specific predictor contributes significantly to explaining the variance in the outcome. Furthermore, the R-squared metric (coefficient of determination) gives us a robust measure of how well the regression predictions approximate the real data points.

#view model intercept and coefficients
print(fit.intercept, fit.coefficients)

67.67352554133275

#view p-values of intercept and coefficents
print(fit.summary.pValues)

#view RMSE of model
print(fit.summary.r2)

0.7340272170388176

Interpreting the Model Coefficients and Equation

Upon reviewing the raw output from the fitted model, we can extract the refined parameters that define our predictive equation. These coefficients, typically rounded for practical use, allow us to

construct a highly interpretable relationship between the predictors and the outcome variable. Understanding these metrics is critical for translating statistical output into actionable business or educational insights.

The extracted key components defining the linear model are as follows:

Intercept ($beta_0$): **67.674**
Coefficient of Hours ($beta_1$): **5.556**
Coefficient of Prep Exams ($beta_2$): **-0.602**

These values combine to form the finalized regression equation, which is the mathematical core of our trained model:

Exam Score = 67.674 + 5.556(hours) - 0.602(prep_exams)

This equation signifies that for every additional hour studied, the exam score is expected to increase by 5.556 points, assuming the number of prep exams remains constant. The negative coefficient for prep exams (-0.602) suggests that, controlling for hours studied, taking more prep exams slightly decreases the predicted score. While this finding might seem statistically sound within the model constraints, it warrants deeper domain-specific investigation, as it could signal complex interactions or the presence of confounding variables not included in the model.

Using the Model for Prediction and Inference

The primary, practical purpose of training a linear regression model is to generate reliable predictions for new, unseen data points based on the established linear relationship. By substituting specific values for the predictor variables into our fitted equation, we can estimate the expected outcome, demonstrating the predictive power derived from the mathematical relationship established during the training phase.

Consider, for instance, a hypothetical student who studies for 3 hours and takes 2 preparatory exams. Using our derived equation, we can confidently estimate their final score:

Estimated exam score = 67.674 + 5.556 * (3) - 0.602 * (2)

Estimated exam score = 67.674 + 16.668 - 1.204 = **83.138**

Thus, the model predicts that a student with these specific characteristics is expected to achieve an exam score of approximately 83.1. This capability to quantify expectation based on independent variables is what makes linear regression a powerful and indispensable tool in statistical and predictive modeling, enabling data-driven decision-making.

Evaluating Statistical Significance (P-Values and R-Squared)

Beyond the raw coefficients, the statistical metrics provided in the model summary offer deep, critical insights into the reliability and explanatory power of the model. The p-values specifically help us determine whether the observed relationship between each predictor and the response variable is statistically significant, typically assessed against a predefined significance level ($alpha$) of 0.05.

Reviewing the extracted p-values from our model summary:

P-value of Intercept: **<.001** (Highly Significant)
P-value of Hours: **0.519** (Not Significant)
P-value of Prep Exams: **<.0001** (Highly Significant)

Since the p-value for the number of prep exams (<0.0001) is far less than 0.05, we reject the null hypothesis and conclude that the number of prep exams has a statistically significant association with the exam score. However, the p-value for hours studied (0.519) is substantially greater than the 0.05 threshold. This strongly suggests that, in the presence of the 'prep_exams' variable, 'hours' does not independently contribute a statistically significant amount of explanatory power to the current model structure.

This finding often necessitates model refinement. A data scientist might decide to remove the non-significant variable ('hours') and retrain a simpler model using only 'prep_exams' as the sole predictor, or alternatively, investigate why 'hours' is not performing as expected. Such investigation could involve checking for data transformation needs, exploring non-linear relationships, or identifying complex multicollinearity issues not accounted for in the preliminary analysis.

Understanding Model Fit with R-Squared

Lastly, we evaluate the overall performance of the model using the R-squared value, which is one of the most common and robust metrics for assessing the overall goodness-of-fit of a regression model.

R-squared of model: **0.734**

The R-squared value of 0.734, or 73.4%, indicates that **73.4%** of the total variation observed in the final exam scores can be successfully explained by the combined linear relationship established by the number of hours studied and the number of prep exams taken. This level of fit is generally considered quite strong for typical real-world social or educational data, suggesting our model captures substantial underlying patterns.

A high R-squared suggests that the predictors we selected account for a large portion of the

variability in the response variable. However, it is crucial to remember that a high R-squared does not automatically guarantee that the model is perfect, nor does it prove causation. Further diagnostic checks, such as examining residuals for patterns and verifying assumptions like homoscedasticity, should always be performed to ensure the model's reliability and adherence to statistical requirements. For a complete list of available regression metrics and advanced diagnostics, users should consult the official PySpark documentation.

Note: Refer to the PySpark documentation for a complete list of regression summary metrics you can view, including Root Mean Squared Error (RMSE) and Adjusted R-squared, which provide additional context on prediction accuracy and model complexity.

Further Exploration in PySpark ML

This step-by-step guide has provided a robust framework for implementing and interpreting linear regression using the scalable infrastructure of PySpark. By leveraging sequential components like the **SparkSession** initializer, the **VectorAssembler** transformer, and the **LinearRegression** estimator, data scientists can efficiently build predictive models capable of handling vast datasets without compromising performance. The methodology demonstrated here serves as a foundational blueprint for executing more complex machine learning tasks within the expansive Spark environment.

Now that you have mastered the basics of fitting a linear model, you may wish to explore other common machine learning tasks and more advanced algorithms supported by the PySpark ML library, such as classification models or clustering techniques.

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