PySpark is a Python interface for Apache Spark that allows you to write Spark applications using Python APIs. It also provides a PySpark shell for interactively analyzing your data in a distributed environment. PySpark supports most of Spark's features, including Spark SQL, DataFrames, Streaming, Machine Learning (MLlib), and Spark Core. ###ARTICLEThe provided code is used to manipulate a DataFrame in Apache Spark, which appears to be containing information about various cars. Here's a paraphrased version of the article: In this example, we start by filtering out the top 5 rows from the DataFrame and then displaying them using `slice1.show()` and `slice2.show()`. The rest of the data is assigned to the variables `part2` and displayed separately. Next, a new column named "Car New" is added to the original DataFrame. This column is created by multiplying the 'Present_Price' column with 2, resulting in a new value for each row. The newly created column is then dropped from the DataFrame, leaving us with the original data without this additional column. This manipulation can be useful when you want to perform specific operations on subsets of your data while still keeping track of the rest of it. In this example, we will go over data pre-processing using PySpark and how to use the VectorAssembler for training a machine learning model. We will begin by converting our dataframe to a pandas dataframe. ###CONVERTING TO PANDAS DATAFRAMEIt seems like you've provided a long output of a machine learning model in PySpark, including various metrics and evaluation results. I'll try to help you make sense of it and provide a concise summary. **Summary of the Model Performance** The model is a linear regression model, and the output provides various metrics to evaluate its performance. Here are the key takeaways: 1. **MAE (Mean Absolute Error)**: The MAE for the train set is 1.25, and for the test set, it's 1.35. This indicates that, on average, the model's predictions are off by about 1.25-1.35 units. 2. **MSE (Mean Squared Error)**: The MSE for the train set is 3.88, and for the test set, it's 4.08. This suggests that the model is doing reasonably well, but there's still some room for improvement. 3. **RMSE (Root Mean Squared Error)**: The RMSE for the train set is 1.97, and for the test set, it's 2.02. This is similar to the MAE, indicating that the model's predictions are off by about 2 units on average. 4. **R2 (Coefficient of Determination)**: The R2 value for the train set is 0.85, and for the test set, it's 0.83. This indicates that the model is able to explain about 85% of the variability in the data, which is a good fit. **Overall Assessment** Based on these metrics, it seems that the linear regression model is doing reasonably well in predicting the target variable. The MAE, MSE, and RMSE values are all relatively low, and the R2 value is high, indicating a good fit. However, there is always room for improvement. You may consider trying other models, such as decision trees, random forests, or gradient boosting machines, to see if they perform better on this dataset. **Example Use Cases** 1. **Predicting Continuous Outcomes**: This model can be used to predict continuous outcomes, such as stock prices, temperatures, or energy consumption. 2. **Understanding Relationships**: The model can help you understand the relationships between the independent variables and the target variable, which can inform business decisions or policy interventions. **Next Steps** 1. **Feature Engineering**: You may want to try feature engineering techniques, such as polynomial transformations or interaction terms, to see if they improve the model's performance. 2. **Hyperparameter Tuning**: You can try tuning the model's hyperparameters, such as the regularization strength or the number of iterations, to see if it improves the model's performance. 3. **Model Comparison**: You can compare the performance of this model with other models, such as machine learning algorithms or statistical models, to see which one performs best on this dataset.from pyspark.ml.feature import VectorAssembler from pyspark.ml.regression import LinearRegression from pyspark.sql import SQLContext from pyspark.ml.evaluation import RegressionEvaluator from sklearn.datasets import load_boston import pandas as pd import matplotlib.pyplot as plt # Load Boston Housing Price dataset boston = load_boston() df_boston = pd.DataFrame(boston.data,columns=boston.feature_names) df_boston['target'] = pd.Series(boston.target) sc = SparkContext().getOrCreate() sqlContext = SQLContext(sc) data = sqlContext.createDataFrame(df_boston) # Define SqlContext and create data frame by using df_boston data features = boston.feature_names.tolist() va = VectorAssembler(inputCols=features, outputCol='features') va_df = va.transform(data) va_df = va_df.select(['features', 'target']) va_df.show(3) # Split data into the train and test parts. (train, test) = va_df.randomSplit([0.8, 0.2]) # Define linear regression model to use lin_reg = LinearRegression(featuresCol='features', labelCol='target', predictionCol='pred_target') # Fit linear regression model to data fit = lin_reg.fit(va_df) # View model summary print(fit.summary.r2) import pyspark.ml.regression from pyspark.sql import SQLContext, DataFrame from pyspark.ml.feature import VectorAssembler from pyspark.ml.evaluation import RegressionEvaluator from sklearn.datasets import load_boston import pandas as pd import matplotlib.pyplot as plt boston = load_boston() df_boston = pd.DataFrame(boston.data,columns=boston.feature_names) df_boston['target'] = pd.Series(boston.target) sc = SparkContext().getOrCreate() sqlContext = SQLContext(sc) data = sqlContext.createDataFrame(df_boston) features = boston.feature_names.tolist() va = VectorAssembler(inputCols=features, outputCol='features') va_df = va.transform(data) va_df = va_df.select(['features', 'target']) va_df.show(3) (train, test) = va_df.randomSplit([0.8, 0.2]) glr=GeneralizedLinearRegression(labelCol="target",family="poisson",maxIter=10,regParam=0.3) model = glr.fit(train) print("Coefficients: ", model.coefficients) print("Intercept: ", model.intercept) print(str(model.summary)) tdata = model.transform(test) tdata.show(3) rmse = RegressionEvaluator(labelCol="target", predictionCol="prediction", metricName="rmse") rmse = rmse.evaluate(tdata) mae = RegressionEvaluator(labelCol="target", predictionCol="prediction", metricName="mae") mae = mae.evaluate(tdata) r2 = RegressionEvaluator(labelCol="target", predictionCol="prediction", metricName="r2") r2 = r2.evaluate(tdata) print("RMSE: ", rmse) print("MAE: ", mae) print("R-squared: ", r2) x_ax = range(0, tdata.count()) y_pred=tdata.select("prediction").collect() y_orig=tdata.select("target").collect() plt.plot(x_ax, y_orig, label="original") plt.plot(x_ax, y_pred, label="predicted") plt.title("Boston test and predicted data") plt.xlabel('X-axis') plt.ylabel('Y-axis') plt.legend(loc='best',fancybox=True, shadow=True) plt.grid(True) plt.show() sc.stop() ###ARTICLEmeanAbsoluteError# Returns the mean absolute error, which is a risk function corresponding to the expected value of the absolute error loss or l1-norm loss. Notes This ignores instance weights (setting all to 1.0) from LinearRegression.weightCol. This will change in later Spark versions. meanSquaredError# Returns the mean squared error, which is a risk function corresponding to the expected value of the squared error loss or quadratic loss. Notes This ignores instance weights (setting all to 1.0) from LinearRegression.weightCol. This will change in later Spark versions. numInstances# Number of instances in DataFrame predictions objectiveHistory# Objective function (scaled loss + regularization) at each iteration. This value is only available when using the "l-bfgs" solver. See also LinearRegression.solver pValues# Two-sided p-value of estimated coefficients and intercept. This value is only available when using the "normal" solver. If LinearRegression.fitIntercept is set to True, then the last element returned corresponds to the intercept. See also LinearRegression.solver predictionCol# Field in "predictions" which gives the predicted value of the label at each instance. predictions# Dataframe outputted by the model's transform method. r2# Returns R^2, the coefficient of determination. Notes This ignores instance weights (setting all to 1.0) from LinearRegression.weightCol. This will change in later Spark versions. See also Wikipedia coefficient of determination r2adj# Returns Adjusted R^2, the adjusted coefficient of determination. Notes This ignores instance weights (setting all to 1.0) from LinearRegression.weightCol. This will change in later Spark versions. Wikipedia coefficient of determination, Adjusted R^2 residuals# Residuals (label - predicted value) rootMeanSquaredError# Returns the root mean squared error, which is defined as the square root of the mean squared error. Notes This ignores instance weights (setting all to 1.0) from LinearRegression.weightCol. This will change in later Spark versions. tValues# T-statistic of estimated coefficients and intercept. This value is only available when using the "normal" solver. If LinearRegression.fitIntercept is set to True, then the last element returned corresponds to the intercept. See also LinearRegression.solver totalIterations# Number of training iterations until termination. This value is only available when using the "l-bfgs" solver. See also LinearRegression.solver Regression is a method for modeling the relationship between data points \(\mathbf{x}\) and corresponding real-valued targets \(y\). In natural sciences and social sciences, regression is used to characterize the relationship between inputs and outputs. Machine learning is concerned with prediction, and regression problems occur whenever we want to predict a numerical value. Examples of regression include predicting prices (e.g., home prices, stock prices), predicting length of stay (for patients in hospital), demand forecasting (for retail sales), etc. Linear regression is the simplest and most popular method among standard tools for regression. It was first proposed in the 19th century and assumes that the relationship between features \(\mathbf{x}\) and targets \(y\) is linear, i.e., that \(y\) can be expressed as a weighted sum of the inputs \(\textbf{x}\), with some noise on the observations. To fit a model for predicting house prices based on area and age, we need to collect data on sales, areas, and ages. The dataset is called a training set or training data, and each row represents one sale. The target (price) is called a label or target, and the variables (age and area) are features or covariates. The linearity assumption says that the target can be expressed as a weighted sum of the features: price = \(w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b\), where \(w_{\mathrm{area}}\) and \(w_{\mathrm{age}}\) are weights, and \(b\) is a bias or intercept. Given a dataset, our goal is to choose the weights and bias so that the predictions best fit the true prices observed in the data. In high-dimensional datasets, it's more convenient to use linear algebra notation, where the prediction \(\hat{y}\) is expressed as a dot product: \(\hat{y} = w_1 \cdot x_1 + ... + w_d \cdot x_d + b\), with weights in vector \(\mathbf{w}\) and inputs in vector \(\mathbf{x}\).To refer to our entire dataset through the design matrix \(\mathbf{X}\), where \(\mathbf{X}\) contians one row for every example and one colomn for every feature. For a collection of data points \(\mathbf{X}\), the predictions \(\hat{\mathbf{y}}\) can be expressed via the matrix-vector product: \({\hat{\mathbf{y}}} = \mathbf{X} \mathbf{w} + b\). Given a training dataset \(\mathbf{X}\) and corresponding targets \(\mathbf{y}\), the goal of linear regression is to find the weight vector \(w\) and bias term \(b\) that, given a new data point \(\mathbf{x}_i\), sampled from the same distribution as the training data, will predict the target \(y_i\) with the lowest error. Even if we beleev that the best model for predicting \(y\) given \(\mathbf{x}\) is linear, we would not expect to find real-world data where \(y_i\) exactly equals \(\mathbf{w}^\top \mathbf{x}+b\) for all points \((\mathbf{x}, y)\). Thus, even when we are confindt that the underlying relationship is linear, we will incorporate a noise term to account for such errors. The loss function quantifies the distance between the real and predicted value of the target, with smaller values being better and perfect predictions incurring a loss of \(0\). A popular loss function in regression problems is the sum of squared errors: \(l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)}\right)^2\). To measure the quality of a model on the entire dataset, we average the losses on the training set: \(L(\mathbf{w}, b) =\frac{1}{n}\sum_{i=1}^n l^{(i)}(\mathbf{w}, b)\). When training the model, we want to find parameters \((\mathbf{w}^*, b^*)\) that minimize the total loss across all training examples. Linear regression can be solved analytically by applying a simple formula, yielding a global optimum: \(\mathbf{w}^* = (\mathbf X^\top \mathbf X)^{-1}\mathbf X^\top \mathbf{y}\). However, for more complex models, we use gradient descent, which iteratively reduces the error by updating the parameters in the direction that incrementally lowers the loss function. In each iteration, we sample a random minibatch of examples, compute the derivative of the average loss on the mini batch with regard to the model parameters, and update the parameters based on this gradient and a predetermined step size \(\eta > 0\). ###to minimize the loss on training data. The more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called generalization. We return to these topics throughout the book. Given the learned linear regression model $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, we can now estimate the price of a new house (not contained in the training data) given its area x_1 and age (year) x_2. Estimating targets given features is commonly called prediction and inference. We will try to stick with prediction because calling this step inference, despite emerging as standard jargon in deep learning, is somewhat of a misnomer. In statistics, inference more often denotes estimating parameters based on a dataset. This misuse of terminology is a common source of confusion when deep learning practitioners talk to statisticians. When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Java. To illustrate why this matters so much, we can consider two methods for adding vectors. To start we instantiate two 10000-dimensional vectors containing all ones. In one method we will loop over the vectors with a Java for loop. In the other method we will rely on a single call to DJL. We need some utilities such as StopWatch. We can load them using the %load macro. %load ../utils/djl-imports %load ../utils/plot-utils %load ../utils/StopWatch.java import java.util.stream.*; int n = 10000; NDManager manager = NDManager.newBaseManager(); NDArray a = manager.ones(new Shape(n)); NDArray b = manager.ones(new Shape(n)); Now we can benchmark the workloads. First, we add them, one coordinate at a time, using a for loop. NDArray c = manager.zeros(new Shape(n)); StopWatch stopWatch = new StopWatch(); for (int i = 0; i < n; i++) { c.set(new NDIndex(i), a.getFloat(i) + b.getFloat(i)); } String.format("%.5f sec", stopWatch.stop()); Alternatively, we rely on DJL to compute the elementwise sum: stopWatch.start(); NDArray d = a.add(b); String.format("%.5f sec", stopWatch.stop()); You probably noticed that the second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the math to the library and need not write as many calculations ourselves, reducing the potential for errors. While you can already get your hands dirty using only the information above, in the following section we can more formally motivate the square loss objective via assumptions about the distribution of noise. Recall from the above that the squared loss $l(y, \hat{y}) = \frac{1}{2} (y - \hat{y})^2$ has many convenient properties. These include a simple derivative $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$. As we mentioned earlier, linear regression was invented by Gauss in 1795, who also discovered the normal distribution (also called the Gaussian). It turns out that the connection between the normal distribution and linear regression runs deeper than common parentage. To refresh your memory, the probability density of a normal distribution with mean μ and variance σ^2 is given as follows: $p(z) = \frac{1}{\sqrt{2 \pi \sigma^2}} \exp\left(-\frac{1}{2 \sigma^2} (z - \mu)^2\right).$ Below we define a Java function to compute the normal distribution. ```java public float[] normal(float[] z, float mu, float sigma) { float[] dist = new float[z.length]; for (int i = 0; i < z.length; i++) { float p = 1.0f / (float) Math.sqrt(2 * Math.PI * sigma * sigma); dist[i] = p * (float) Math.pow(Math.E, -0.5 / (sigma * sigma) * (z[i] - mu) * (z[i] - mu)); } return dist; } ``` We can now visualize the normal distributions. ```java int start = -7; int end = 14; float step = 0.01f; int count = (int) (end / step); float[] x = new float[count]; for (int i = 0; i < count; i++) { x[i] = start + i * step; } public float[] main() { float[] y = normal(x, 5.0f, 2.0f); // Do something with y } ``` Note: I've corrected some minor formatting issues and reformatted the Java code to be more consistent with standard Java coding conventions. ###Minimizing the Negative Log-Likelihood to Estimate Linear Regression Parameters We are tasked with finding the values of b and $\mathbf{w}$ that maximize the likelihood of observing our dataset, given by $(3.1.14)$¶\[P(Y\mid X) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}).\] By simplifying to minimize the Negative Log-Likelihood (NLL) $-\log p(\mathbf y|\mathbf X)$, we can transform this into an equivalent but more tractable optimization problem. The log-likelihood expression $(3.1.15)$¶\[-\log p(\mathbf y|\mathbf X) = \sum_{i=1}^n \frac{1}{2} \log(2 \pi \sigma^2) + \frac{1}{2 \sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2.\] with the assumption that σ is a fixed constant. After removing this term, we're left with minimizing squared error, which is equivalent to maximum likelihood estimation of a linear model under additive Gaussian noise. Rewriting the expression in layer notation, our linear model can be viewed as a single artificial neuron comprising multiple inputs and outputs connected by weights and biases. This transformation aligns with the concept of fully-connected or dense layers found in neural networks, laying the groundwork for understanding how linear regression can be framed within the realm of deep learning. By examining the role of dendrites, synapses, and weighted sums in biological neurons, we gain insight into why linear models were a natural starting point for developing artificial neural network architectures. Moving forward from this foundation, our exploration will delve into the intricacies of multilayer perceptrons, exploring how these complex networks can be composed of numerous layers. The journey ahead will shed light on the evolution of artificial intelligence and its reliance on mathematical and computational advancements. Deriving the Closed-Form Solution for Linear Regression Using Normal Distribution. Linear regression with squared error is a fundamental problem in machine learning and statistics. By incorporating the normal distribution, we can derive a closed-form solution to this optimization problem. Omitting the bias b from the problem is possible by adding one column to X consisting of all ones. The optimization problem can be written in matrix and vector notation as follows: ###ARTICLEparaphrased text here The WallStreetMojo Investment Banking course offered by WallStreetMojo was incredibly informative and well-organized. The material was easy to follow, and it covered all the key areas of investment banking including valuation techniques, financial modeling and equity research. 5/5Kaveri Choudhury The WallStreetMojo investment banking course has received overwhelmingly positive reviews from its participants. Many praised the course's well-structured format, which breaks down complex topics into simple and easy-to-understand concepts. The content was found to be insightful, informative, and highly relevant to the industry. WallStreetMojo offers an impressive array of courses on financial analysis that cater to a wide range of professionals in the Asset Management Industry. Antoine Bietrix (United Kingdom)Associate Portfolio Manager has rated the platform highly for its comprehensive coverage of investment banking concepts, despite them being simplified. Recently, Sola Samuel Olabokunde (Nigeria)Credit Risk Specialist completed WallStreetMojo's "Introduction to VBA" course and found it to be incredibly valuable. Wiza (Zambia)Business Manager appreciates that the platform helps him stay refreshed and relevant in his roles as a business manager and tutor. Galen Cawley (United States)Independent Trader and Consultant praises Wall Street Mojo for its clear and most comprehensive video courses on financial analysis, which make it easy to understand. Amanda Lewis (United States)Junior Technical Analyst also appreciates the bundle format, making it simple to build knowledge step-by-step. For those interested in investment banking, Ruicheng ZhangUCSB Economics Freshman found WallStreetMojo's "Investment Banking Fundamentals Course" to be an excellent resource. Shelina LoeStudent Researcher completed the free Investment Banking Fundamentals course by Wall Street Mojo and was thoroughly impressed with its engaging content and assessments. Absekha MuruganandamB. Com Graduate appreciates the comprehensive curriculum and practical applications of WallStreetMojo's courses, which significantly enhanced his skills and confidence in finance. Meanwhile, Jesseon Babu (India)Senior Laboratory Manager Operations praises the resource for finance and accounting concepts alongside examples to gain a clearer understanding. Other users have praised the platform for its outstanding value, including Vivek Kumar SharmaFounder - OptiReach Strategies who found the financial modeling and valuation course delivered by Wallstreetmojo to be a great refresher. Abdelmutaal OsmanFreelance Business Consultant also appreciated the bundle's relevance to the present world job market. The WallStreetMojo platform provides a high-quality learning experience for students, offering well-organized study materials, interactive exercises, and video lectures to reinforce key concepts. I recently completed WallStreetMojo's *Intro to Investment Banking* course, and it was an excellent learning experience. The course covers key topics like financial modeling, valuation techniques, and M&A. I recently completed this course and it was a highly valuable experience. The material was well-organized, covering a range of essential topics in a clear and engaging manner. The Investment Banking course exceeded my expectations with its well-structured curriculum and practical approach. Designed for both beginners and those with some financial knowledge. I recently completed the Investment Banking course from WallStreetMojo, and I found it to be an excellent learning resource for anyone looking to gain a solid foundation in this field. This beginner course of learn financial model in excel is one of the best courses that can be done freely to start your career in finance industry. I recently completed the Investment Banking Course by Wall Street Mojo, and it exceeded my expectations! The course is well-structured, covering everything from financial modeling and valuation techniques to M&A and IPO processes. A Comprehensive Review of Investment Banking Online Training," gaining insights into financial models and valuation techniques. I recently completed the What is Investment Banking (FC) course from Wall Street Mojo, and it was an excellent learning experience. The course is well-structured, breaking down complex topics into simple, easy-to-understand concepts. Having completed the investment banking course offered by WallStreetMojo, I have appreciated the quality of the video content. In this course, key investment banking concepts like financial modeling, valuation, and M&A. One of the best tutorials available to learn and gain insights into Financial Modelling. I liked the pace of the course and the fact that assessments were an integral part of it. I recently completed the WallStreetMojo free course, and I couldn't be more impressed! The course is well-organized and covers complex financial concepts in a clear, easy-to-understand way. I recently completed the WallStreetMojo free course, and it was absolutely fantastic! The course is well-structured and provides clear, in-depth explanations of complex financial concepts. During my MBA, I took Wall Street Mojo's Financial Planning and Analysis course, and it was a great supplement to my studies. The content was clear, practical, and focused on real-world applications like budgeting, forecasting, and financial modeling. I had an incredible experience with WallStreetMojo's Investment Banking course. The course was well-structured, insightful, and highly informative. The free online session on investment banking was insightful and well-structured, covering topics like investment banking, retail banking, research departments, and mergers and acquisitions etc. I recently completed the Corporate Finance course on WallStreetMojo, and I'm really impressed with the quality and depth of the content. The course provided in-depth insights into financial modeling, valuation techniques, M&A, and IPO processes, making complex topics easy to understand. I recently completed the Banking Investment Course, and I must say it was a transformative experience that has significantly boosted my understanding of financial markets and investment strategies. The WallStreetMojo Investment Banking Free Course is a solid crash course for beginners looking to understand the fundamentals of investment banking. I recently attended online session on investment banking that was a great introduction to the field. The speaker covered essential topics clearly, using real-world examples to make complex concepts accessible Finance for non finance course was a game-changer! As someone with no background in finance, I found the content incredibly clear and easy to follow. WallStreetMojo's Investment Banking program was a fantastic educational experience. The material was organized effectively and was simple to comprehend. Difficult topics were presented in a straightforward and clear way, making them easy to grasp. The Investment Banking course by WallStreetMojo exceeded my expectations. It provides a thorough understanding of key concepts such as financial modeling, valuation techniques, and M&A analysis. I'm a B.Com student and a fresher. I did the Basic Excel Course from Wall Street Mojo and found it really helpful. The videos were beginner-friendly and easy to follow. I learned important Excel tools, formulas, and charts,Tables I recently completed my investment banking course and I am so happy that I completed it from wall mojo it helped me understand what exactly investment banking is they explained it in such a easy and simple way that I understood it in one go Just completed their investment banking course and i really loved it. It amazes me to see how they are providing such quality content for free with a verified certificate!!! The course is so nice and explained with live examplesRunning Linear Regression in Excel: Uncovering the Basics and Beyond Keerthi Korepu The tutorial explains the basics of regression analysis and demonstrates a few different ways to perform linear regression in Excel. To predict next year's sales numbers, imagine being provided with numerous data points that may impact the outcome. By using regression analysis, you can identify which factors are crucial and how certain you can be about your predictions. In statistical modeling, regression analysis is used to estimate relationships between two or more variables. Dependent variable (dependent or predicted) is the main factor being understood and predicted. Independent variables (explanatory predictors) are the factors that may influence the dependent variable. Regression helps understand how changes in independent variables affect the dependent variable, allowing for mathematical determination of which factors have an impact. ###ARTICLER2 value is calculated from the total sum of squares. It shows how many points fall on the regression line. The R Squared of 0.91 means that 91% of our values fit the regression analysis model. In other words, 91% of the dependent variables (y-values) are explained by the independent variables (x-values). Generally, R Squared of 95% or more is considered a good fit. Adjusted R Square is the R square adjusted for the number of independent variable in the model. You will want to use this value instead of R square for multiple regression analysis. Standard Error shows the precision of your regression analysis - the smaller the number, the more certain you can be about your regression equation. It is very useful for measuring the prediction. Observations is simply the number of observations in your model. The second part of the output is Analysis of Variance (ANOVA). Basically, it splits the sum of squares into individual components that give information about the levels of variability within your regression model: df is the number of the degrees of freedom associated with the sources of variance. SS is the sum of squares. The smaller the Residual SS compared with the Total SS, the better your model fits the data. MS is the mean square. F is the F statistic, or F-test for the null hypothesis. It is used to test the overall significance of the model. Significance F is the P-value of F. The ANOVA part is rarely used for a simple linear regression analysis in Excel, but you should definitely have a close look at the last component. Significance F value gives an idea of how reliable (statistically significant) your results are. If Significance F is less than 0.05 (5%), your model is OK. regression formulas in excel can be accessed through the following functions: =SLOPE(C2:C25, B2:B25) and =CORREL(B2:B25,C2:C25). The correlation coefficient indicates the strength of the relationship between two variables. Additionally, the LINEST function can be used to get more statistics for regression analysis by setting the stats parameter to TRUE. However, Microsoft Excel is not a statistical program and may require professional software like XLSTAT or RegressIt for advanced needs. ###ARTICLEGiven this tool is a lifesaver for me and my team when it comes to efficiently parsing data from different sources. The ability to easily merge tables with the Merge Wizard has been a game-changer, saving us so much time on our daily reports. I've already recommended Ablebits to everyone on my team and we can't imagine going back to doing things the old way. The customer service is top-notch too - they responded to my issue in under 6 hours and provided clear instructions on how to track similar errors in the future. It's worth every cent, trust me! Ablebits has been a major time-saver for me since I first started using it in 2020. Working at a global company with multiple databases would have driven me crazy if I didn't have this tool to compile data from different sources. I love how user-friendly the interfaces are - they're way better than MS, and the functionality is so much more advanced. Plus, the customer service is always super responsive and communicative until my issue is fully resolved. Weighted residuals and metrics in Linear Regression ###ENDARTICLEfrom pyspark.sql.functions import pandas_udf, PandasUDFType from pyspark.sql.types import DoubleType import numpy as np # Define the mean and variance functions using Pandas UDAFs @pandas_udaf(returnType=DoubleType(), type_modifier=PandasUDFMarker.AGGREGATION,) def calculate_mean(pandasSer: pd.Series) -> np.float64: return pandasSer.mean() @pandas_udaf(returnType=DoubleType(), type_modifier=PandasUDFMarker.AGGREGATION,) def calculate_variance(pandasSer: pd.Series) -> np.float64: return pandasSer.var() # Create a Spark DataFrame data = [[1, 2], [3, 4], [5, 6]] columns = ['A', 'B'] df = spark.createDataFrame(data, columns) # Register the Pandas UDAFs as SQL functions spark.udf.register("mean", calculate_mean) spark.udf.register("variance", calculate_variance) # Use the Pandas UDAFs in a Spark DataFrame operation result_df = df.groupBy().agg(mean('A').alias('mean_A'), variance('B').alias('var_B')) The provided text describes how to view and interpret the output of a regression model using Excel. It explains how to interpret the coefficients, p-values, and R-squared value of the model, as well as how to use this information to write a fitted regression equation. The text also discusses the different algorithms used for classification and regression in Spark MLlib, including logistic regression with elastic net regularization. import org.apache.spark.ml.classification.LogisticRegressionModel; import org.apache.spark.sql.Dataset; import org.apache.spark.sql.Row; import org.apache.spark.sql.SparkSession; // Load training data Dataset training = spark.read().format("libsvm") .load("data/mllib/sample_libsvm_data.txt"); LogisticRegression lr = new LogisticRegression() .setMaxIter(10) .setRegParam(0.3) .setElasticNetParam(0.8); // Fit the model LogisticRegressionModel lrModel = lr.fit(training); // Print the coefficients and intercept for logistic regression System.out.println("Coefficients: " + lrModel.coefficients() + " Intercept: " + lrModel.intercept()); // We can also use the multinomial family for binary classification LogisticRegression mlr = new LogisticRegression() .setMaxIter(10) .setRegParam(0.3) .setElasticNetParam(0.8) .setFamily("multinomial"); // Fit the model LogisticRegressionModel mlrModel = mlr.fit(training); // Print the coefficients and intercepts for logistic regression with multinomial family System.out.println("Multinomial coefficients: " + lrModel.coefficientMatrix() + "Multinomial intercepts: " + mlrModel.interceptVector()); ###ARTICLEThe Spark repo provides an example of how to train a multiclass logistic regression model with elastic net regularization using the LogisticRegression class. The model is trained on the sample_multiclass_classification_data.txt dataset. For multinomial logistic regression, the algorithm produces K sets of coefficients, or a matrix of dimension K x J where K is the number of outcome classes and J is the number of features. If the algorithm is fit with an intercept term then a length K vector of intercepts is available. The conditional probabilities of the outcome classes k ∈ {1, 2, ..., K} are modeled using the softmax function. The model minimizes the weighted negative log-likelihood, using a multinomial response model, with elastic-net penalty to control for overfitting. The objective function used for training is: min_{β, β0} -[Σ(i=1)^L w_i · log P(Y = y_i|X_i)] + λ [(1/2(1-α)||β||_2^2 + α ||β||_1)] where w_i are the weights, β are the coefficients, and λ is the regularization parameter. The model can be trained using the fit method, which returns a LogisticRegressionModel object. This object provides various methods to extract information about the model, such as the coefficient matrix, intercept vector, training summary, objective history, false positive rate by label, true positive rate by label, precision by label, recall by label, and F-measure by label. We have only used logistic regression in our Spark example, but let's explore an alternative approach with decision trees. The Spark MLlib implementation of decision trees can be used for classification and regression tasks. In this example, we will train a decision tree classifier on the LibSVM dataset and evaluate its performance. ###ARTICLE val setLabel = "Chain indexers and tree in a Pipeline." val pipelineModel = new Pipeline() .setStages(Array(setLabel, "featureIndexer", "dt", "labelConverter")) // Train model. val model = pipeline.fit(trainingData) // Make predictions. val predictions = model.transform(testData) // Select example rows to display. predictions.select("predictedLabel", "label", "features").show(5) // Select (prediction, true label) and compute test error. val evaluator = new MulticlassClassificationEvaluator() .setLabelCol("indexedLabel") .setPredictionCol("prediction") .setMetricName("accuracy") val accuracy = evaluator.evaluate(predictions) println(s"Test Error = ${(1.0 - accuracy)}") val treeModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel] println(s"Learned classification tree model: ${treeModel.toDebugString}") Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/DecisionTreeClassificationExample.scala" in the Spark repo. More details on parameters can be found in the Java API documentation. import org.apache.spark.ml.Pipeline; import org.apache.spark.ml.PipelineModel; import org.apache.spark.ml.PipelineStage; import org.apache.spark.ml.classification.RandomForestClassificationModel; import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator; import org.apache.spark.ml.feature.*; import org.apache.spark.sql.Dataset; import org.apache.spark.sql.Row; import org.apache.spark.sql.SparkSession; // Load and parse the data file, converting it to a DataFrame. Dataset data = spark.read().format("libsvm").load("data/mllib/sample_libsvm_data.txt"); // Index labels, adding metadata to the label column. // Fit on whole dataset to include all labels in index. StringIndexerModel labelIndexer = new StringIndexer() .setInputCol("label") .setOutputCol("indexedLabel") .fit(data); // Automatically identify categorical features, and index them. // Set maxCategories so features with > 4 distinct values are treated as continuous. VectorIndexerModel featureIndexer = new VectorIndexer() .setInputCol("features") .setOutputCol("indexedFeatures") .setMaxCategories(4) .fit(data); // Split the data into training and test sets (30% held out for testing) Dataset[] splits = data.randomSplit(new double[] {0.7, 0.3}); Dataset trainingData = splits[0]; Dataset testData = splits[1]; // Train a RandomForest model. RandomForestClassifier rf = new RandomForestClassifier() .setLabelCol("indexedLabel") .setFeaturesCol("indexedFeatures"); // Convert indexed labels back to original labels. IndexToString labelConverter = new IndexToString() .setInputCol("prediction") .setOutputCol("predictedLabel") .setLabels(labelIndexer.labelsArray()[0]); // Chain indexers and forest in a Pipeline Pipeline pipeline = new Pipeline() .setStages(new PipelineStage[] {labelIndexer, featureIndexer, rf, labelConverter}); // Train model. This also runs the indexers. PipelineModel model = pipeline.fit(trainingData); // Make predictions. Dataset predictions = model.transform(testData); // Select example rows to display. predictions.select("predictedLabel", "label", "features").show(5) from pyspark.ml import Pipeline from pyspark.ml.classification import GBTClassifier from pyspark.ml.feature import StringIndexer, VectorIndexer from pyspark.ml.evaluation import MulticlassClassificationEvaluator # ###ARTICLEUsing Gradient Boosted Trees (GBT) and Multilayer Perceptron Classifiers for Supervised Learning in Spark Gradient Boosted Trees (GBT) and multilayer perceptron classifiers are powerful algorithms for supervised learning tasks such as classification. This document provides an overview of how to use these algorithms in Apache Spark. ### Gradient Boosted Trees (GBT) GBT is an ensemble learning method based on decision trees, where multiple weak models are combined to create a strong predictive model. In Spark, the GBT classifier can be used to train a model on labeled data and make predictions on new, unseen data. Here's an example of how to use the GBT classifier in Spark: ```scala // Load training data val data = spark.read.format("libsvm") .load("data/mllib/sample_libsvm_data.txt") // Split the data into train and test sets val splits = data.randomSplit(new double[]{0.6, 0.4}, 1234L) val train = splits(0) val test = splits(1) // Specify layers for the neural network (in this case, a single layer with 4 inputs) val layers = new int[] {4} // Create the trainer and set its parameters val trainer = new MultilayerPerceptronClassifier() .setLayers(layers) .setBlockSize(128) .setSeed(1234L) .setMaxIter(100) // Train the model val model = trainer.fit(train) // Compute accuracy on the test set val result = model.transform(test) val predictionAndLabels = result.select("prediction", "label") val evaluator = new MulticlassClassificationEvaluator() .setMetricName("accuracy") println(s"Test set accuracy = ${evaluator.evaluate(predictionAndLabels)}") ``` ### Multilayer Perceptron Classifier The multilayer perceptron classifier is a type of neural network that consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network, and each node performs a linear combination of the inputs with weights $\w$ and bias $\b$, followed by an activation function. Here's an example of how to use the multilayer perceptron classifier in Spark: ```python from pyspark.ml.classification import MultilayerPerceptronClassifier # Load training data data = spark.read.format("libsvm") .load("data/mllib/sample_multiclass_classification_data.txt") # Split the data into train and test sets splits = data.randomSplit([0.6, 0.4], 1234) train = splits[0] test = splits[1] # Specify layers for the neural network (in this case, a single layer with 4 inputs) layers = [4, 5, 4, 3] # Create the trainer and set its parameters trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, blockSize=128, seed=1234) # Train the model model = trainer.fit(train) # Compute accuracy on the test set result = model.transform(test) predictionAndLabels = result.select("prediction", "label") evaluator = MulticlassClassificationEvaluator(metricName="accuracy") print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels))) ``` ### Conclusion GBT and multilayer perceptron classifiers are powerful algorithms for supervised learning tasks in Spark. By using these algorithms, you can train accurate models on labeled data and make predictions on new, unseen data. The Java MultilayerPerceptronClassifier Example demonstrates how to use Spark MLlib for multiclass classification tasks. It shows a full example code that includes loading training data, specifying layers for the neural network, fitting a multi-layer perceptron model with Spark's mlp function, predicting test data, and printing the coefficients and intercept of the linear SVM model. For binary classification with linear SVM, it uses the LinearSVC class from pyspark.ml.classification. The example shows how to load training data, fit the linear SVM model, print the coefficients and intercept for linear SVC, and compute the classification error on test data using the MulticlassClassificationEvaluator metric. One-vs-Rest classifier (also known as One-vs-All) is another example of machine learning reduction for performing multiclass classification. It takes instances of Classifier and creates a binary classification problem for each of the K classes. The classifier for class i is trained to predict whether the label is i or not, distinguishing class i from all other classes. import org.apache.spark.ml.classification.LogisticRegression; import org.apache.spark.ml.classification.OneVsRest; import org.apache.spark.ml.classification.OneVsRestModel; import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator; import org.apache.spark.sql.Dataset; import org.apache.spark.sql.Row; ###ARTICLEThe code uses the Spark library to train a Factorization Machine (FM) classifier on a dataset in LibSVM format. The FM classifier is particularly useful for large-scale classification tasks where traditional machine learning algorithms can be computationally expensive. from pyspark.ml import Pipeline from pyspark.ml.classification import FMClassifier from pyspark.ml.feature import MinMaxScaler, StringIndexer from pyspark.ml.evaluation import MulticlassClassificationEvaluator data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt") labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data) featureScaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures").fit(data) (trainingData, testData) = data.randomSplit([0.7, 0.3]) fm = FMClassifier(labelCol="indexedLabel", featuresCol="scaledFeatures", stepSize=0.001) pipeline = Pipeline(stages=[labelIndexer, featureScaler, fm]) model = pipeline.fit(trainingData) predictions = model.transform(testData) evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy").evaluate(predictions) print("Test set accuracy = %g" % accuracy) fmModel = model.stages[2] print("Factors: " + str(fmModel.factors)) print("Linear: " + str(fmModel.linear)) print("Intercept: " + str(fmModel.intercept)) The provided text discusses various aspects of using Spark MLlib for classification and regression tasks. the response variable Y_i is related to some distribution from the exponential family of distributions. Spark's GeneralisedLinearRegression interface allows for flexible specification of GLMs which can be used for various types of prediction problems including linear regression, Poisson regression, logistic regression and others. Currently in spark.ml, only a subset of the exponential family distributions are supported and they are listed below. NOTE: Spark currently only supports up to 4096 features through its GeneralisedLinearRegression interface, and will throw an exception if this constraint is exceeded. See the advanced section for more details. Still, for linear and logistic regression models with an increased number of features can be trained using the LinearRegression and LogisticRegression estimators. GLMs require exponential family distributions that can be written in their "canonical" or "natural" form, aka natural exponential family distributions. The form of a natural exponential family distribution is given as: \[f_Y(y|\theta, \tau) = h(y, \tau)\exp{\left(\frac{\theta \cdot y - A(\theta)}{d(\tau)} \right)}\] where θ is the parameter of interest and τ is a dispersion parameter. In a GLM the response variable Y_i is assumed to be drawn from a natural exponential family distribution: \[Y_i \sim f\left(\cdot|\theta_i, \tau \right)\] where the parameter of interest θ_i is related to the expected value of the response variable μ_i by \[\mu_i = A'(\theta_i)\] Here, $A'(\theta_i)$ is defined by the form of the distribution selected. GLMs also allow specification of a link function, which defines the relationship between the expected value of the response variable μ_i and the so called linear predictor η_i: \[g(\mu_i) = \eta_i = \vec{x_i}^T \cdot \vec{\beta}\] Often, the link function is chosen such that $A' = g^{-1}$, which yields a simplified relationship between the parameter of interest θ and the linear predictor η. In this case, the link function $g(\mu)$ is said to be the "canonical" link function. \[\theta_i = A'^{-1}(\mu_i) = g(g^{-1}(\eta_i)) = \eta_i\] A GLM finds the regression coefficients $\vec{\beta}$ which maximize the likelihood function. \[\max_{\vec{\beta}} \mathcal{L}(\vec{\theta}|\vec{y},X) = \prod_{i=1}^{N} h(y_i, \tau) \exp{\left(\frac{y_i\theta_i - A(\theta_i)}{d(\tau)}\right)}\] where the parameter of interest θ_i is related to the regression coefficients $\vec{\beta}$ by \[\theta_i = A'^{-1}(g^{-1}(\vec{x_i} \cdot \vec{\beta}))\] Spark's generalized linear regression interface also provides summary statistics for diagnosing the fit of GLM models, including residuals, p-values, deviances, the Akaike information criterion and others. The following example demonstrates training a GLM with a Gaussian response and identity link function and extracting model summary statistics. ###ARTICLE# Generalized Linear Regression .setFamily("gaussian") .setLink("identity") .setMaxIter(10) .setRegParam(0.3); ###ARTICLEImported Java libraries to load the LIBSVM format dataset into a DataFrame and split it into training and test sets. Selected example rows from the transformed data to display predictions, computed test error using RMSE evaluator, and printed the learned regression tree model. Referenced the Spark repo for full example code at "examples/src/main/scala/org/apache/spark/examples/ml/DecisionTreeRegressionExample.scala" in Scala API docs for more details. Utilized VectorIndexer to index categorical features, added metadata to the DataFrame recognized by tree-based algorithms and trained a DecisionTree model. Utilized a Pipeline to chain indexer and forest, trained on the training data set and evaluated on the held-out test set. Referenced the Python API docs for more details. Used pyspark.ml module to load LIBSVM format dataset into a DataFrame and split it into training and test sets. Utilized VectorIndexer to index categorical features and trained a RandomForest model, chained indexer and forest in a Pipeline and computed RMSE error on test data. Referenced the Scala API docs for more details. Imported necessary libraries to load LIBSVM format dataset into a DataFrame and split it into training and test sets. Used VectorIndexer to index categorical features and trained a RandomForest model using pyspark.ml.regression module. import org.apache.spark.ml.Pipeline; import org.apache.spark.ml.PipelineModel; import org.apache.spark.ml PipelineStage; import org.apache.spark.ml.evaluation.RegressionEvaluator; import org.apache.spark.ml.feature.VectorIndexer; import org.apache.spark.ml.regression.RandomForestRegressionModel; import org.apache.spark.ml.regression.RandomForestRegressor; import org.apache.spark.sql.Dataset; import org.apache.spark.sql.Row; import org.apache.spark.sql.SparkSession; val data = spark.read().format("libsvm").load("data/mllib/sample_libsvm_data.txt") // Automatically identify categorical features, and index them. // Set maxCategories so features with > 4 distinct values are treated as continuous. val featureIndexer = new VectorIndexer() .setInputCol("features") .setOutputCol("indexedFeatures") .setMaxCategories(4) .fit(data) val rf = new RandomForestRegressor() .setLabelCol("label") .setFeaturesCol("indexedFeatures") val pipeline = new Pipeline() .setStages(new Array(PipelineStage) (featureIndexer, rf)) val model = pipeline.fit(data) val predictions = model.transform(testData) val evaluator = new RegressionEvaluator() .setLabelCol("label") .setPredictionCol("prediction") .setMetricName("rmse") val rmse = evaluator.evaluate(predictions) println(s"Root Mean Squared Error (RMSE) on test data = $rmse") val rfModel = model.stages(1).asInstanceOf[RandomForestRegressionModel] println(s"Learned regression forest model: ${rfModel.toDebugString}") val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3)) // Train a GBT model. val gbt = new GBTRegressor() .setLabelCol("label") .setFeaturesCol("indexedFeatures") .setMaxIter(10) // Chain indexer and GBT in a Pipeline. val pipeline = new Pipeline() .setStages(Array(featureIndexer, gbt)) // Train model. This also runs the indexer. val model = pipeline.fit(trainingData) // Make predictions. val predictions = model.transform(testData) // Select example rows to display. predictions.select("prediction", "label", "features").show(5) // Select (prediction, true label) and compute test error. val evaluator = new RegressionEvaluator() .setLabelCol("label") .setPredictionCol("prediction") .setMetricName("rmse") val rmse = evaluator.evaluate(predictions) println(s"Root Mean Squared Error (RMSE) on test data = $rmse") val gbtModel = model.stages(1).asInstanceOf[GBTRegressionModel] println(s"Learned regression GBT model: ${gbtModel.toDebugString}") Okay, the user wants me to paraphrase the given article text using one of the three methods: SE, NNES, or IB. Let me first understand the content. The article discusses the AFTSurvivalRegressionModel in Spark MLlib, comparing its behavior to R's survreg, mentions the L-BFGS optimization, and provides code examples in Python, Scala, Java, and R. There's also a section on isotonic regression with the pool adjacent violators algorithm. I need to randomly select one of the three methods. Let me check the probabilities: 40% SE, 30% NNES, 30% IB. So, I'll pick one. Let's go with IB (Increase Burstiness) since it's 30% and might add variety to the sentences. For IB, I need to vary sentence lengths and structures. Let me look at the original text. The first part talks about the optimization algorithm and the behavior difference. I can split that into shorter sentences. For example, "The optimization algorithm underlying the implementation is L-BFGS." becomes "The implementation utilizes the L-BFGS optimization algorithm." Then, the next sentence about matching results with R's survreg can be rephrased as "It aligns with the outcomes from R's survival function survreg." Next, the part about Spark MLlib outputting zero coefficients for constant columns. I can rephrase that to "In cases where a dataset contains a constant nonzero column, Spark MLlib generates zero coefficients." Then, the code examples in different languages can be structured with bullet points or separated into distinct sections for clarity. I should also check for any technical terms that need to stay accurate, like L-BFGS and isotonic regression. Making sure not to alter any technical details. The isotonic regression section can have its sentences split into more concise parts. For instance, breaking down the mathematical equation into a separate line for clarity. Wait, the user provided a long text, so I need to ensure that the paraphrased version maintains all the information without missing any key points. Let me go through each paragraph and restructure sentences where possible. Also, the code examples should remain intact but perhaps formatted differently if needed. I need to avoid markdown and keep the response in plain text. Let me start paraphrasing each section, ensuring that the burstiness is increased by varying sentence structures and lengths. Let me make sure the technical terms are correct and the explanations are clear. Once done, I'll format the response with the specified delimiters. **$\log \sigma$** The implementation utilizes the L-BFGS optimization algorithm. It aligns with the outcomes from R's survival function survreg. However, when a dataset includes a constant nonzero column, Spark MLlib generates zero coefficients, differing from R's behavior. This discrepancy arises due to the model's handling of redundant features. ### Code Examples **Python:** ```python from pyspark.ml.regression import AFTSurvivalRegression # Create DataFrame data = [(1.218, 1.0, [1.560, -0.605]), ...] schema = StructType([StructField("label", DoubleType(), False), ...]) training = spark.createDataFrame(data, schema) # Train model aft = AFTSurvivalRegression() aft.setQuantileProbabilities([0.3, 0.6]) aft.setQuantilesCol("quantiles") model = aft.fit(training) # Predict model.transform(training).show() ``` **Scala:** ```scala val data = Seq((1.218, 1.0, Vectors.dense(1.560, -0.605)), ...).toDF("label", "censor", "features") val aft = new AFTSurvivalRegression() .setQuantileProbabilities(Array(0.3, 0.6)) .setQuantilesCol("quantiles") val model = aft.fit(data) println(s"Coefficients: ${model.coefficients}") ``` **Java:** ```java Dataset training = spark.createDataFrame(data, schema); AFTSurvivalRegression aft = new AFTSurvivalRegression() .setQuantileProbabilities(new double[]{0.3, 0.6}) .setQuantilesCol("quantiles"); AFTSurvivalRegressionModel model = aft.fit(training); System.out.println("Coefficients: " + model.coefficients()); ``` **R:** ```r library(survival) ovarianDF