databrickspractica_a_solucion (1)

(https://databricks.com)

Explicacion del notebook

2

```
%scala
// Lista los archivos y directorios en la ruta /FileStore/tables/
val files = dbutils.fs.ls("/FileStore/tables/")

// Muestra los resultados
files.foreach(file => println(s"Nombre: ${file.name}, Tamaño: ${file.size} bytes, Ruta: ${file.path}"))
```

```
Nombre: CATEGORIES.csv, Tamaño: 151 bytes, Ruta: dbfs:/FileStore/tables/CATEGORIES.csv
Nombre: CUSTOMERS.csv, Tamaño: 2095 bytes, Ruta: dbfs:/FileStore/tables/CUSTOMERS.csv
Nombre: Case/, Tamaño: 0 bytes, Ruta: dbfs:/FileStore/tables/Case/
Nombre: Case.xlsx, Tamaño: 13106 bytes, Ruta: dbfs:/FileStore/tables/Case.xlsx
Nombre: Clean_USA_Housing.csv, Tamaño: 479709 bytes, Ruta: dbfs:/FileStore/tables/Clean_USA_Housing.csv
Nombre: DEALERS.csv, Tamaño: 283 bytes, Ruta: dbfs:/FileStore/tables/DEALERS.csv
Nombre: EMPLOYEES.csv, Tamaño: 432 bytes, Ruta: dbfs:/FileStore/tables/EMPLOYEES.csv
Nombre: Mall_Customers.csv, Tamaño: 3966 bytes, Ruta: dbfs:/FileStore/tables/Mall_Customers.csv
Nombre: ORDERS.csv, Tamaño: 1138 bytes, Ruta: dbfs:/FileStore/tables/ORDERS.csv
Nombre: PRICES.csv, Tamaño: 540 bytes, Ruta: dbfs:/FileStore/tables/PRICES.csv
Nombre: PRODUCTS.csv, Tamaño: 6984 bytes, Ruta: dbfs:/FileStore/tables/PRODUCTS.csv
Nombre: PatientInfo.xlsx, Tamaño: 199778 bytes, Ruta: dbfs:/FileStore/tables/PatientInfo.xlsx
Nombre: RIESGO_CREDITICIO.csv, Tamaño: 1656 bytes, Ruta: dbfs:/FileStore/tables/RIESGO_CREDITICIO.csv
Nombre: SIZES.csv, Tamaño: 73 bytes, Ruta: dbfs:/FileStore/tables/SIZES.csv
Nombre: bot_devices.csv, Tamaño: 736279 bytes, Ruta: dbfs:/FileStore/tables/bot_devices.csv
Nombre: case-1.csv, Tamaño: 12461 bytes, Ruta: dbfs:/FileStore/tables/case-1.csv
Nombre: case.csv, Tamaño: 12461 bytes, Ruta: dbfs:/FileStore/tables/case.csv
Nombre: iot_devices-1.csv, Tamaño: 1773764 bytes, Ruta: dbfs:/FileStore/tables/iot_devices-1.csv
Nombre: iot_devices.csv, Tamaño: 1773764 bytes, Ruta: dbfs:/FileStore/tables/iot_devices.csv
Nombre: patient_info-1.csv, Tamaño: 488857 bytes, Ruta: dbfs:/FileStore/tables/patient_info-1.csv
Nombre: patient_info-2.csv, Tamaño: 488857 bytes, Ruta: dbfs:/FileStore/tables/patient_info-2.csv
```

3

```
%scala
//EJEMPLO 1: MODELO DE CLASIFICACION CON SCALA

// Databricks notebook source
////////////////////////////////////////////
/// SIMPLE VERSION OF LOG REG EXAMPLE //////
////////////////////////////////////////////

// Note that usually all imports would occur at the top and
// most of this would be in an object this layout if for learning purposes only

// Logistic Regression Example
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.sql.SparkSession

// Optional: Use the following code below to set the Error reporting
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

//LogisticRegression: La clase que provee Spark para entrenar modelos de regresión logística
//SparkSession: Punto de entrada principal para trabajar con DataFrames y el stack de SQL en Spark.
//Log4j: Se utiliza para configurar el nivel de logs de Spark (en este caso se setea a ERROR para evitar mensajes
extensos).

// Spark Session
val spark = SparkSession.builder().getOrCreate()

// Use Spark to read in the Titanic csv file.
val data = spark.read.option("header","true").option("inferSchema","true").format("csv").load("/FileStore/tables/t

// Print the Schema of the DataFrame
data.printSchema()

/////////////////////
/// Display Data /////
/////////////////////
val colnames = data.columns
val firstrow = data.head(1)(0)
println("\n")
println("Example Data Row")
for(ind <- Range(1,colnames.length)){
  println(colnames(ind))
  println(firstrow(ind))
  println("\n")
}

//////////////////////////////////////////////////
//// Setting Up DataFrame for Machine Learning ////
//////////////////////////////////////////////////

// Grab only the columns we want
val logregdataall = data.select(data("Survived").as("label"), $"Pclass", $"Sex", $"Age", $"SibSp", $"Parch", $"Far
$"Embarked")
val logregdata = logregdataall.na.drop()

//Se seleccionan las columnas relevantes y se renombra Survived como label (columna que Spark ML interpretará como
clasificación).
//Se eliminan las filas con valores nulos (na.drop()).


// A few things we need to do before Spark can accept the data!
// We need to deal with the Categorical columns

// Import VectorAssembler and Vectors
import org.apache.spark.ml.feature.{VectorAssembler,StringIndexer,VectorIndexer,OneHotEncoder}
import org.apache.spark.ml.linalg.Vectors

// Deal with Categorical Columns

val genderIndexer = new StringIndexer().setInputCol("Sex").setOutputCol("SexIndex")
val embarkIndexer = new StringIndexer().setInputCol("Embarked").setOutputCol("EmbarkIndex")

val genderEncoder = new OneHotEncoder().setInputCol("SexIndex").setOutputCol("SexVec")
```

```
val embarkEncoder = new OneHotEncoder().setInputCol("EmbarkIndex").setOutputCol("EmbarkVec")

// StringIndexer: Convierte valores categóricos (ej. "male", "female") en índices numéricos.
//Sex → SexIndex
//Embarked → EmbarkIndex
//OneHotEncoder: Convierte el índice numérico en un vector binario donde cada posición representa una categoría.
//SexIndex → SexVec
//EmbarkIndex → EmbarkVec

// Assemble everything together to be ("label","features") format
val assembler = (new VectorAssembler()
                 .setInputCols(Array("Pclass", "SexVec", "Age","SibSp","Parch","Fare","EmbarkVec"))
                 .setOutputCol("features") )

//VectorAssembler: Toma columnas numéricas (incluyendo las transformaciones One-Hot) y las combina en un único vec
features. Este vector es lo que Spark ML utiliza para entrenar los modelos.

/////////////////////////////
/// Split the Data ////////
/////////////////////////
val Array(training, test) = logregdata.randomSplit(Array(0.7, 0.3), seed = 12345)

// Se divide el DataFrame en 70% para training y 30% para test. Se fija una semilla (seed = 12345) para mantener l
reproducibilidad de la división.

//////////////////////////////
// Set Up the Pipeline ///////
//////////////////////////////
import org.apache.spark.ml.Pipeline

val lr = new LogisticRegression()

val pipeline = new Pipeline().setStages(Array(genderIndexer,embarkIndexer,genderEncoder,embarkEncoder,assembler, l

// Pipeline: Permite encadenar etapas de transformación y un estimador final (en este caso, la regresión logística

// Fit the pipeline to training documents.
val model = pipeline.fit(training)

// Ajusta (entrena) la pipeline con los datos de entrenamiento.
// Devuelve un PipelineModel que internamente contiene los transformadores ajustados y el modelo de regresión logí
entrenado.

// Get Results on Test Set
val results = model.transform(test)

// Aplica el modelo entrenado sobre el conjunto de prueba.
// Devuelve un DataFrame con nuevas columnas, incluyendo la predicción (prediction) para cada fila.

////////////////////////////////////
//// MODEL EVALUATION /////////////
////////////////////////////////////

// For Metrics and Evaluation
import org.apache.spark.mllib.evaluation.MulticlassMetrics

// Need to convert to RDD to use this
val predictionAndLabels = results.select($"prediction",$"label").as[(Double, Double)].rdd

// Instantiate metrics object
val metrics = new MulticlassMetrics(predictionAndLabels)

// Confusion matrix
println("Confusion matrix:")
println(metrics.confusionMatrix)

// MulticlassMetrics: Clase de Spark para calcular métricas de evaluación (precision, recall, F1, etc.).
// Se imprime la matriz de confusión para observar cuántas instancias han sido clasificadas correctamente o incorr
cada clase.
```

▸ data: org.apache.spark.sql.DataFrame = [PassengerId: integer, Survived: integer ... 10 more fields]
▸ logregdata: org.apache.spark.sql.DataFrame = [label: integer, Pclass: integer ... 6 more fields]
▸ logregdataall: org.apache.spark.sql.DataFrame = [label: integer, Pclass: integer ... 6 more fields]

▸ results: org.apache.spark.sql.DataFrame
▸ test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label: integer, Pclass: integer ... 6 more fields]
▸ training: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label: integer, Pclass: integer ... 6 more fields]

```
root
 |-- PassengerId: integer (nullable = true)
 |-- Survived: integer (nullable = true)
 |-- Pclass: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Age: double (nullable = true)
 |-- SibSp: integer (nullable = true)
 |-- Parch: integer (nullable = true)
 |-- Ticket: string (nullable = true)
 |-- Fare: double (nullable = true)
 |-- Cabin: string (nullable = true)
 |-- Embarked: string (nullable = true)



Example Data Row
Survived
0
```

4

```
%scala
dbutils.fs.ls("/FileStore/tables/").foreach(file => println(file.name))
```

```
CATEGORIES.csv
CUSTOMERS.csv
Case/
Case.xlsx
Clean_USA_Housing.csv
DEALERS.csv
EMPLOYEES.csv
Mall_Customers.csv
ORDERS.csv
PRICES.csv
PRODUCTS.csv
PatientInfo.xlsx
RIESGO_CREDITICIO.csv
SIZES.csv
bot_devices.csv
case-1.csv
case.csv
iot_devices-1.csv
iot_devices.csv
patient_info-1.csv
patient_info-2.csv
```

5

```
%scala
//EJEMPLO 2: MODELO DE REGRESION CON SCALA

// Databricks notebook source
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

// To see less warnings
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

// Start a simple Spark Session
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()

// RegressionEvaluator: Permite evaluar el rendimiento de modelos de regresión.
// LinearRegression: Algoritmo de regresión lineal de Spark MLlib.
// ParamGridBuilder y TrainValidationSplit: Herramientas para la validación y búsqueda de hiperparámetros
(aunque no se usan directamente en este ejemplo, se importan).
// Logger y Level de org.apache.log4j para configurar el nivel de log y así reducir la verbosidad de las
salidas.
//      SparkSession: Punto de entrada principal de Spark.
//      VectorAssembler: Permite combinar múltiples columnas en un único vector de características
//      Vectors: Provee estructuras de datos para representar vectores en Spark.

// Prepare training and test data.
val data =
spark.read.option("header","true").option("inferSchema","true").format("csv").load("/FileStore/tables/Clean_US
A_Housing.csv")

// Check out the Data
data.printSchema()

// See an example of what the data looks like
// by printing out a Row
val colnames = data.columns
val firstrow = data.head(1)(0)
println("\n")
println("Example Data Row")
for(ind <- Range(1,colnames.length)){
  println(colnames(ind))
  println(firstrow(ind))
  println("\n")
}

////////////////////////////////////////////////
//// Setting Up DataFrame for Machine Learning ////
////////////////////////////////////////////////

// A few things we need to do before Spark can accept the data!
// It needs to be in the form of two columns
// ("label","features")

// This will allow us to join multiple feature columns
// into a single column of an array of feautre values
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

// Rename Price to label column for naming convention.
// Grab only numerical columns from the data
val df = data.select(data("Price").as("label"),$"Avg Area Income",$"Avg Area House Age",$"Avg Area Number of
Rooms",$"Area Population")

// An assembler converts the input values to a vector
// A vector is what the ML algorithm reads to train a model

// Set the input columns from which we are supposed to read the values
// Set the name of the column where the vector will be stored
val assembler = new VectorAssembler().setInputCols(Array("Avg Area Income","Avg Area House Age","Avg Area
Number of Rooms","Area Population")).setOutputCol("features")

// Use the assembler to transform our DataFrame to the two columns
val output = assembler.transform(df).select($"label",$"features")
```

```
// Create a Linear Regression Model object
val lr = new LinearRegression()

// Fit the model to the data

// Note: Later we will see why we should split
// the data first, but for now we will fit to all the data.
val lrModel = lr.fit(output)

// Print the coefficients and intercept for linear regression
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

// Summarize the model over the training set and print out some metrics!
// Explore this in the spark-shell for more methods to call
val trainingSummary = lrModel.summary

println(s"numIterations: ${trainingSummary.totalIterations}")
println(s"objectiveHistory: ${trainingSummary.objectiveHistory.toList}")

trainingSummary.residuals.show()

println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
println(s"MSE: ${trainingSummary.meanSquaredError}")
println(s"r2: ${trainingSummary.r2}")

// trainingSummary es un LinearRegressionTrainingSummary que contiene información sobre el proceso de
entrenamiento.
//      numIterations: número de iteraciones realizadas por el algoritmo de optimización.
//      objectiveHistory: histórico de valores de la función objetivo durante las iteraciones.
//      residuals: diferencia entre los valores predichos y los valores reales (etiquetados).
//      RMSE (Root Mean Squared Error): error cuadrático medio de la raíz, métrica muy común en regresión.
//      MSE (Mean Squared Error): error cuadrático medio.
//      r2 (R-squared): coeficiente de determinación, indica la proporción de la varianza explicada por el
modelo.
```

▸ data: org.apache.spark.sql.DataFrame = [Avg Area Income: double, Avg Area House Age: double ... 4 more fields]
▸ df: org.apache.spark.sql.DataFrame = [label: double, Avg Area Income: double ... 3 more fields]
▸ output: org.apache.spark.sql.DataFrame

```
root
 |-- Avg Area Income: double (nullable = true)
 |-- Avg Area House Age: double (nullable = true)
 |-- Avg Area Number of Rooms: double (nullable = true)
 |-- Avg Area Number of Bedrooms: double (nullable = true)
 |-- Area Population: double (nullable = true)
 |-- Price: double (nullable = true)



Example Data Row
Avg Area House Age
5.682861321615587


Avg Area Number of Rooms
7.009188142792237


Avg Area Number of Bedrooms
4.09
```

6

```
%scala
//EJEMPLO 3: GRIDSEARCH CON SCALA

// Databricks notebook source
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)


// RegressionEvaluator: Clase para evaluar el rendimiento de un modelo de regresión, normalmente usando
métricas como RMSE, MSE, R-squared, etc.
//      LinearRegression: Algoritmo de regresión lineal provisto por Spark MLlib.
//      ParamGridBuilder y TrainValidationSplit: Herramientas para la búsqueda sistemática de hiperparámetros
y la validación del modelo.
//      Logger.getLogger("org").setLevel(Level.ERROR): Ajusta el nivel de logging para reducir la cantidad de
mensajes que aparecen en consola, facilitando la lectura de resultados.


// Start a simple Spark Session
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()

// Prepare training and test data.
val data =
spark.read.option("header","true").option("inferSchema","true").format("csv").load("/FileStore/tables/Clean_US
A_Housing.csv")
data.printSchema()

// See an example of what the data looks like
val colnames = data.columns
val firstrow = data.head(1)(0)
println("\n")
println("Example Data Row")
for(ind <- Range(1,colnames.length)){
  println(colnames(ind))
  println(firstrow(ind))
  println("\n")
}

////////////////////////////////////////////////
//// Setting Up DataFrame for Machine Learning ////
////////////////////////////////////////////////

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

// Rename label column
// Grab only numerical columns
val df = data.select(data("Price").as("label"),$"Avg Area Income",$"Avg Area House Age",$"Avg Area Number of
Rooms",$"Area Population")

// An assembler converts the input values to a vector
// A vector is what the ML algorithm reads to train a model

// Set the input columns from which we are supposed to read the values
// Set the name of the column where the vector will be stored
val assembler = new VectorAssembler().setInputCols(Array("Avg Area Income","Avg Area House Age","Avg Area
Number of Rooms","Area Population")).setOutputCol("features")

// Transform the DataFrame
val output = assembler.transform(df).select($"label",$"features")

// Create an array of the training and test data
val Array(training, test) = output.select("label","features").randomSplit(Array(0.7, 0.3), seed = 12345)

////////////////////////////////////
//////// LINEAR REGRESSION //////////
////////////////////////////////////
val lr = new LinearRegression()

////////////////////////////////////
```

```
/// PARAMETER GRID BUILDER /////////
///////////////////////////////////
val paramGrid = new ParamGridBuilder().addGrid(lr.regParam,Array(1000,0.001)).build()
//model.validationMetrics

// ParamGridBuilder permite construir una cuadrícula de parámetros.
//      Aquí se está explorando el parámetro regParam (regularización) con dos valores: 1000 y 0.001.
//      build() finaliza la construcción de la cuadrícula de parámetros.

//////////////////////
// TRAIN TEST SPLIT //
//////////////////////

// In this case the estimator is simply the linear regression.
// A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
// 80% of the data will be used for training and the remaining 20% for validation.
val trainValidationSplit = (new TrainValidationSplit()
                            .setEstimator(lr)
                            .setEvaluator(new RegressionEvaluator)
                            .setEstimatorParamMaps(paramGrid)
                            .setTrainRatio(0.8) )


//      TrainValidationSplit:
//      setEstimator: El modelo a entrenar, en este caso lr.
//      setEvaluator: Evalúa el rendimiento del modelo, aquí con RegressionEvaluator por defecto (métricas
como RMSE).
//      setEstimatorParamMaps: La rejilla de hiperparámetros generada por ParamGridBuilder.
//setTrainRatio(0.8): Define el 80% de los datos (del conjunto de entrenamiento) para entrenamiento interno y
el 20% para validación interna.
//En otras palabras, de la parte de training, Spark tomará el 80% para entrenar y el 20% para validar,
eligiendo así la mejor configuración de hiperparámetros.

// You can then treat this object as the new model and use fit on it.
// Run train validation split, and choose the best set of parameters.
val model = trainValidationSplit.fit(training)

//      Ajusta (fit) el modelo seleccionando los mejores hiperparámetros del paramGrid.
// El objeto resultante model es el mejor modelo encontrado (el que obtiene la mejor métrica definida por
RegressionEvaluator).

////////////////////////////////////
// EVALUATION USING THE TEST DATA ///
////////////////////////////////////

// Make predictions on test data. model is the model with combination of parameters
// that performed best.
model.transform(test).select("features", "label", "prediction").show()

//      transform(test) aplica el modelo final sobre los datos de prueba. Se muestran las columnas features,
label (etiqueta real) y prediction (predicción del modelo).
```

```
▸ data: org.apache.spark.sql.DataFrame = [Avg Area Income: double, Avg Area House Age: double ... 4 more fields]
▸ df: org.apache.spark.sql.DataFrame = [label: double, Avg Area Income: double ... 3 more fields]
▸ output: org.apache.spark.sql.DataFrame
▸ test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
▸ training: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
```

```
root
 |-- Avg Area Income: double (nullable = true)
 |-- Avg Area House Age: double (nullable = true)
 |-- Avg Area Number of Rooms: double (nullable = true)
 |-- Avg Area Number of Bedrooms: double (nullable = true)
 |-- Area Population: double (nullable = true)
 |-- Price: double (nullable = true)



Example Data Row
Avg Area House Age
5.682861321615587


Avg Area Number of Rooms
7.009188142792237
```

```
Avg Area Number of Bedrooms
4.09
```

```
7
```

```
%scala
// ************************************************
// EJEMPLO 4: MODELO DE CLUSTERIZACION CON SCALA
// ************************************************

// IMPORTACIONES INICIALES
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.clustering.KMeans

// 1. Lectura y Exploración de Datos
val csv = spark.read
  .option("inferSchema","true")  // Inferir tipos de columna
  .option("header", "true")      // Primera fila como encabezados
  .csv("/FileStore/tables/Mall_Customers.csv")

// Muestra las primeras filas del DataFrame
csv.show()

// Muestra el esquema de las columnas
csv.printSchema()

// Genera descripción estadística de algunas columnas
csv.select("CustomerID", "Gender", "Age", "AnnualIncome","SpendingScore").describe().show()

// Crea vista temporal para consultas SQL
csv.createOrReplaceTempView("MallCustomerData")

// 2. Visualización con SQL (opcional dentro de Databricks)
// %sql
// select * from MallCustomerData

// 3. División de Datos en Entrenamiento y Prueba
val Array(train, test) = csv.randomSplit(Array(0.7, 0.3), seed=1234)
// Se añade un seed para reproducibilidad

// Conteo de filas
val trainRows = train.count()
val testRows = test.count()
println(s"Training Rows: $trainRows, Testing Rows: $testRows")

// 4. Ensamblado de Características
val assembler = new VectorAssembler()
  .setInputCols(Array("AnnualIncome", "SpendingScore")) // Columnas importantes para K-means
  .setOutputCol("features")

// Transformación del conjunto de entrenamiento
val training = assembler.transform(train)
training.show(5) // Muestra las primeras 5 filas transformadas

// 5. Entrenamiento del Modelo K-means
val kmeans = new KMeans()
  .setK(5)                          // Definimos 5 clusters
  .setFeaturesCol("features")       // Vector de características
  .setPredictionCol("prediction")   // Columna de resultado

val kmeansModel = kmeans.fit(training)

// 6. Transformación del Conjunto de Prueba
val testing = assembler.transform(test)
testing.show(5)

// 7. Predicción en el Conjunto de Prueba
val prediction = kmeansModel.transform(testing)
prediction.show(10)

// 8. Análisis de Resultados
prediction.groupBy("prediction").count().show()

// Crear vista temporal de los resultados
prediction.createOrReplaceTempView("CustomerClusterMallData")

// 9. Consulta Final (SQL) de la Asignación de Clusters
// %sql
```

```
// select AnnualIncome, SpendingScore, prediction from CustomerClusterMallData
```

- csv: org.apache.spark.sql.DataFrame = [CustomerID: integer, Gender: string ... 3 more fields]
- prediction: org.apache.spark.sql.DataFrame
- test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [CustomerID: integer, Gender: string ... 3 more fields]
- testing: org.apache.spark.sql.DataFrame
- train: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [CustomerID: integer, Gender: string ... 3 more fields]
- training: org.apache.spark.sql.DataFrame

```
+----------+------+---+------------+-------------+
|CustomerID|Gender|Age|AnnualIncome|SpendingScore|
+----------+------+---+------------+-------------+
|         1|  Male| 19|          15|           39|
|         2|  Male| 21|          15|           81|
|         3|Female| 20|          16|            6|
|         4|Female| 23|          16|           77|
|         5|Female| 31|          17|           40|
|         6|Female| 22|          17|           76|
|         7|Female| 35|          18|            6|
|         8|Female| 23|          18|           94|
|         9|  Male| 64|          19|            3|
|        10|Female| 30|          19|           72|
|        11|  Male| 67|          19|           14|
|        12|Female| 35|          19|           99|
|        13|Female| 58|          20|           15|
|        14|Female| 24|          20|           77|
|        15|  Male| 37|          20|           13|
|        16|  Male| 22|          20|           79|
|        17|Female| 35|          21|           35|
|        18|  Male| 20|          21|           66|
```

8

- df: org.apache.spark.sql.DataFrame
- result: org.apache.spark.sql.DataFrame

```
+---------------------+
|features             |
+---------------------+
|(5,[1,3],[1.0,7.0])  |
|[2.0,0.0,3.0,4.0,5.0]|
|[4.0,0.0,3.0,6.0,7.0]|
+---------------------+

+------------------------------------------------------------+
|pcaFeatures                                                 |
+------------------------------------------------------------+
|[1.5145785486305263,-6.086854838093992,-0.4238123510389752]  |
|[-5.130332362808652,-4.125947145458627,-0.42381235103897485] |
|[-7.162198265877403,-6.9315702405921495,-0.42381235103897485]|
+------------------------------------------------------------+

import org.apache.spark.ml.feature.PCA
import org.apache.spark.ml.linalg.Vectors
data: Array[org.apache.spark.ml.linalg.Vector] = Array((5,[1,3],[1.0,7.0]), [2.0,0.0,3.0,4.0,5.0], [4.0,0.0,3.0,6.0,
7.0])
df: org.apache.spark.sql.DataFrame = [features: vector]
```