Conocimientos spark y scala

Para ejecutarlo:

spark-shell

:load programa.scala

Cargar datos, ver esquema e imprimir:

// Cargar el conjunto de datos

val titanicDF = spark.read.option("header", "true").csv("titanic.csv")

// Imprimir el esquema

println("Schema of the Titanic dataset:")

titanicDF.printSchema()

// Imprimir los datos

println("Sample data from the Titanic dataset:")

titanicDF.show()

Tipo de una variable:

// tipo de variable

println("Type of the Titanic dataset:")

println(titanicDF.getClass)

Seleccionar columnas

// datos de dos columnas

println("Data from 'Cabin' and 'Embarked' columns:")

titanicDF.select("Cabin", "Embarked").show()

Tipos de las columnas:

// tipos de las columnas

println("Data types of columns:")

titanicDF.dtypes.foreach(println)

Resumen de los datos:

// resumen de los datos

println("Summary of the Titanic dataset:")

titanicDF.describe().show()

Crear columna:

// crear columnas

val titanicDFWithNewColumn = titanicDF.withColumn("Fare\_Multiplied", col("Fare") \* 0.21)

Renombrar columna:

// renombrar columna

val titanicDFWithRenamedColumn = titanicDFWithNewColumn.withColumnRenamed("Fare\_Multiplied", "New\_Fare")

// mostrar datos

println("Data with the new column:")

titanicDFWithRenamedColumn.show()

Eliminar columna:

// Eliminar columna

val titanicDFWithoutNewColumn = titanicDFWithRenamedColumn.drop("New\_Fare")

// Mostrar otra vez

println("Data without the new column:")

titanicDFWithoutNewColumn.show()

Detener sesion:

// Detener la sesión de Spark

spark.stop()

Cargar datos con esquema personalizado:

// Importar las bibliotecas necesarias

import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.types.\_

// Definir el esquema personalizado

val titanicSchema = StructType(Array(

StructField("PassengerId", IntegerType, nullable = true),

StructField("Survived", IntegerType, nullable = true),

StructField("Pclass", IntegerType, nullable = true),

StructField("Name", StringType, nullable = true),

StructField("Sex", StringType, nullable = true),

StructField("Age", IntegerType, nullable = true),

StructField("SibSp", IntegerType, nullable = true),

StructField("Parch", IntegerType, nullable = true),

StructField("Ticket", StringType, nullable = true),

StructField("Fare", StringType, nullable = true),

StructField("Cabin", StringType, nullable = true),

StructField("Embarked", StringType, nullable = true)

))

// Cargar el datos con el esquema personalizado

val titanicDF = spark.read.schema(titanicSchema).csv("titanic.csv")

Convertir columna a double:

// Convertir columna a double

val titanicDFConverted = titanicDF.withColumn("Fare", titanicDF("Fare").cast(DoubleType))

Filtrar filas:

// Filtrar filas con mayor a 20 de edad

val filteredAgeDF = titanicDFConverted.filter(titanicDFConverted("Age") > 20)

// Mostrar resultados

println("Filtered Age DataFrame:")

filteredAgeDF.show()

// Filtrar filas con mas de 25 de fare y embarked s

val filteredFareEmbarkedDF = titanicDFConverted.filter(titanicDFConverted("Fare") > 25 && titanicDFConverted("Embarked") === "S")

println("Filtered Fare and Embarked DataFrame:")

filteredFareEmbarkedDF.show()

Calcular suma agrupando:

// suma de fare agrupando por pclass

val fareSumDF = titanicDFConverted.groupBy("Pclass").sum("Fare").withColumnRenamed("sum(Fare)", "Total\_Fare")

println("Total Fare Sum by Pclass:")

fareSumDF.show()

Calcular media agrupando:

// media de edad agrupando por embarked

val ageMeanDF = titanicDFConverted.groupBy("Embarked").avg("Age").withColumnRenamed("avg(Age)", "Average\_Age")

println("Average Age by Embarked:")

ageMeanDF.show()

Agrupan y calcular diferentes medidas:

// agrupar por sexo y media de los que sobrevivien y suma de las fare

val groupedSurvivedFareDF = titanicDFConverted.groupBy("Sex").agg(

avg(when($"Survived" === 1, $"Age")).as("Avg\_Age\_Survived"),

sum($"Fare").as("Total\_Fare")

)

println("Grouped Data by Sex - Average Age of Survived and Total Fare:")

groupedSurvivedFareDF.show()

Numero filas y columas e imprimir con formato:

// Mostrar el número de filas y columnas

val numRows = titanicDF.count()

val numCols = titanicDF.columns.length

println(s"Number of rows in the DataFrame: $numRows")

println(s"Number of columns in the DataFrame: $numCols")

Numero filas con nulos:

// cuantas filas con nulos

val nullRowsCount = titanicDF.filter(row => row.anyNull).count()

println(s"Number of rows with at least one null value: $nullRowsCount")

Nulos en cada fila:

// nulos por filas

println("Number of nulls per row:")

titanicDF.select(titanicDF.columns.map(c => sum(when(col(c).isNull, 1).otherwise(0)).alias(c)): \_\*).show()

Eliminar filas con nulos en un atributo:

// Eliminar las filas con nulo en atributos

val cleanedDF = titanicDF.na.drop(Seq("Embarked", "Fare"))

Media de columna:

// media de la columna age

val meanAge = cleanedDF.select(avg("Age")).head().getDouble(0)

Completar nulos con media:

// Completar los nulos de age con la media

val filledDF = cleanedDF.na.fill(meanAge, Seq("Age"))

Agrupar y contar:

// Agrupar por embarked y contar cuantos embarcaron por puerto

val embarkedCount = filledDF.groupBy("Embarked").count().orderBy("Embarked")

println("Number of passengers embarked at each port:")

embarkedCount.show()

Contar por filtros:

// Contar hombres y sobrevivieron

val maleSurvivedCount = filledDF.filter($"Sex" === "male" && $"Survived" === 1).count()

println(s"Number of male passengers who survived: $maleSurvivedCount")

// Valores diferentes columna

println("Different values that the 'Embarked' column can take:")

filledDF.select("Embarked").distinct().show()

Detectar y reemplazar outliers

// detectar y reemplazar outliers en el atributo 'Age'

def replaceOutliers(df: DataFrame, column: String): DataFrame = {

val quantiles = df.stat.approxQuantile(column, Array(0.25, 0.75), 0.05)

val q1 = quantiles(0)

val q3 = quantiles(1)

val iqr = q3 - q1

val lowerRange = q1 - 1.5 \* iqr

val upperRange = q3 + 1.5 \* iqr

df.withColumn(column, when(col(column) < lowerRange || col(column) > upperRange, null).otherwise(col(column)))

}

// Reemplazar outliers

val titanicDFWithoutOutliers = replaceOutliers(titanicDF, "Age").na.drop("any", Seq("Age"))

Nueva columna marcando intervalo y tabla de frecuencia:

// nueva columna en base al atributo 'Age' marcando en qué intervalo de edad

val titanicDFWithAgeInterval = titanicDFWithoutOutliers.withColumn("Age\_Interval", floor(col("Age") / 5) \* 5)

val ageFreq = titanicDFWithAgeInterval.groupBy("Age\_Interval").count()

ageFreq.show()

Agrupar por varios y ordenar:

// Agrupar por Embarked, Pclass y Sex

val groupedDF = titanicDFWithAgeInterval.groupBy("Embarked", "Pclass", "Sex").count().orderBy("Embarked", "Pclass", "Sex")

println("Grouped by Embarked, Pclass, and Sex:")

groupedDF.show()

Tabla de frecuencias:

// tabla de frecuencias del atributo 'Fare'

val fareFreq = titanicDFWithAgeInterval.groupBy("Fare").count().orderBy("Fare")

println("Fare frequency table:")

fareFreq.show()

Crear rdd

// Crear un RDD

val rdd = spark.sparkContext.parallelize(Seq(

(1, "apple", 5),

(2, "banana", 15),

(3, "orange", 10),

(4, "grape", 20),

(5, "orange", 20)

))

Numero de filas rdd:

// Obtener el número de filas

val numRows = rdd.count()

println(s"Number of rows in the RDD: $numRows")

Imprimir rdd:

// Imprimir

println("All elements in the RDD:")

rdd.collect().foreach(println)

Crear columna con conversion:

// columna que convierta a mayúsculas otra columna

val uppercaseRDD = rdd.map { case (id, fruit, value) => (id, fruit.toUpperCase, value) }

Filtrar filas:

// Filtrar las filas superior a 10

val filteredRDD = uppercaseRDD.filter { case (\_, \_, value) => value > 10 }

Sumar valores reduciendo por key:

// Sumar los valores reduciendo por columna

val sumByFruitRDD = filteredRDD.map { case (\_, fruit, value) => (fruit, value) }

.reduceByKey(\_ + \_)

// Imprimir el resultado

println("Sum of values by fruit:")

sumByFruitRDD.collect().foreach(println)

Crear tabla temporal:

// Crear un DataFrame

val df = spark.createDataFrame(Seq(

(1, "apple", 5),

(2, "banana", 15),

(3, "orange", 10),

(4, "grape", 20)

)).toDF("id", "fruit", "value")

// Registrar el DataFrame como tabla temporal

df.createOrReplaceTempView("fruits")

Consulta sql con condicion

// consulta SQL con condición

val result = spark.sql("SELECT \* FROM fruits WHERE value > 10")

// Mostrar el resultado

result.show()

Valor medio con agrupacion:

// consulta SQL para calcular el valor medio con agrupación

val avgValueByFruit = spark.sql("SELECT fruit, AVG(value) AS avg\_value FROM fruits GROUP BY fruit")

// Mostrar el resultado

avgValueByFruit.show()

Consulta con in:

// consulta SQL con la cláusula IN

val result = spark.sql("SELECT \* FROM fruits WHERE fruit IN ('apple', 'banana')")

// Mostrar el resultado

result.show()

Con ventana:

// Importar las funciones de ventana

import org.apache.spark.sql.expressions.Window

import org.apache.spark.sql.functions.\_

// Definir la ventana

val windowSpec = Window.partitionBy("fruit").orderBy("value")

// consulta SQL utilizando una ventana

val result = spark.sql("SELECT \*, ROW\_NUMBER() OVER (PARTITION BY fruit ORDER BY value) AS row\_number FROM fruits")

// Mostrar el resultado

result.show()

Importar librería para operaciones:

// Importar las bibliotecas necesarias

import org.apache.spark.sql.functions.\_

Porcentajes según atributos:

// Calcular el porcentaje de hombres y mujeres que sobrevivieron y murieron

val genderSurvivalPercentage = titanicDF.groupBy("Sex", "Survived")

.agg(count("Survived").alias("Count"))

.withColumn("Total", sum("Count").over())

.withColumn("Percentage", col("Count") / col("Total") \* 100)

.drop("Count", "Total")

.orderBy("Sex", "Survived")

.show()

Porcentajes de combinaciones de atributos:

// Calcular el porcentaje de personas sobre el total según el puerto de embarque, sexo y si sobrevivieron

val portGenderSurvivalPercentage = titanicDF.groupBy("Embarked", "Sex", "Survived")

.agg(count("Survived").alias("Count"))

.withColumn("Total", sum("Count").over())

.withColumn("Percentage", col("Count") / col("Total") \* 100)

.drop("Count", "Total")

.orderBy("Embarked", "Sex", "Survived").show()

Tabla de frecuencia:

// Construir una tabla de frecuencia por la edad por intervalos de 5 años

val ageFrequencyTable = titanicDF.select("Age")

.withColumn("AgeGroup", floor(col("Age") / 5) \* 5)

.groupBy("AgeGroup")

.agg(count("AgeGroup").alias("Frequency"))

.orderBy("AgeGroup").show()

Encontrar elemento:

// Encontrar el puerto donde la media de edad de los pasajeros que embarcaron fue más alta

val portWithHighestAvgAge = titanicDF.groupBy("Embarked")

.agg(avg("Age").alias("AvgAge"))

.orderBy(desc("AvgAge")).show()

Porcentaje con condicion:

// Calcular el porcentaje de pasajeros que sobrevivieron si tenían hijos

val survivedWithChildrenPercentage = titanicDF.filter("Parch > 0")

.groupBy()

.agg((sum(when(col("Survived") === 1, 1).otherwise(0)) / count("\*") \* 100).alias("Percentage")).show()

Valor medio segun atributos:

// Calcular la tarifa media por clase y puerto de embarque

val avgFareByClassAndPort = titanicDF.groupBy("Pclass", "Embarked")

.agg(avg("Fare").alias("AvgFare"))

.orderBy("Pclass", "Embarked").show()

Matriz de correlacion:

// Crear una matriz de correlación entre el sexo, embarked y la clase

val correlationMatrix = titanicDF.stat.crosstab("Sex", "Embarked")

.join(titanicDF.stat.crosstab("Sex", "Pclass"))

.drop("Sex\_Embarked", "Sex\_Pclass").show()

Distribucion de atributos:

// distribución de pasajeros según la edad y la clase por intervalos

val ageClassDistribution = titanicDF.select("Age", "Pclass")

.na.drop(Seq("Age"))

.withColumn("AgeGroup", floor(col("Age") / 10) \* 10)

.groupBy("AgeGroup", "Pclass")

.count()

.orderBy("AgeGroup", "Pclass").show()

Elemento con condiciones:

// Encontrar la clase en la que más hombres sobrevivieron de todos los hombres que embarcaron en el puerto S

val classWithMostSurvivedMen = titanicDF.filter(col("Sex") === "male" && col("Embarked") === "S")

.groupBy("Pclass")

.agg(sum(when(col("Survived") === 1, 1).otherwise(0)).alias("SurvivedMen"))

.orderBy(desc("SurvivedMen"))

.select("Pclass")

.show()

String indexer:

//eliminamos filas con nulos en estos atributos

val cleanedDF = titanicDF.na.drop(Seq("Embarked", "Sex"))

// String indexer

val stringIndexer = new StringIndexer()

.setInputCols(Array("Sex", "Embarked"))

.setOutputCols(Array("SexIndex", "EmbarkedIndex"))

val titanicIndexed = stringIndexer.fit(cleanedDF).transform(cleanedDF)

Convertir columna a numeros:

import org.apache.spark.sql.types.IntegerType

// Con la función cast()

val titanicDFWithIntPclass = titanicIndexed.withColumn("Pclass", col("Pclass").cast(IntegerType))

Correlaccion entre atributos:

// Correlación entre el sexo, puerto de embarque y clase

val correlationMatrix = titanicDFWithIntPclass.stat.corr("SexIndex", "EmbarkedIndex") // Correlación entre sexo y puerto de embarque

val correlationMatrix2 = titanicDFWithIntPclass.stat.corr("SexIndex", "Pclass") // Correlación entre sexo y clase

val correlationMatrix3 = titanicDFWithIntPclass.stat.corr("EmbarkedIndex", "Pclass") // Correlación entre puerto de embarque y clase

println(s"Correlación entre sexo y puerto de embarque: $correlationMatrix")

println(s"Correlación entre sexo y clase: $correlationMatrix2")

println(s"Correlación entre puerto de embarque y clase: $correlationMatrix3")

// Crear un DataFrame para mostrar las correlaciones en una tabla

val correlationDF = Seq(

("Sex-Embarked", correlationMatrix),

("Sex-Pclass", correlationMatrix2),

("Embarked-Pclass", correlationMatrix3)

).toDF("Attribute Pair", "Correlation")

correlationDF.show()

groupby de 3:

// Agrupar por sexo, puerto de embarque y clase

val groupedByAttributes = titanicIndexed.groupBy("Sex", "Embarked", "Pclass").count()

groupedByAttributes.show()

Arbol de decision para clasificacion:

import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.types.\_

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.DecisionTreeClassifier

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler}

import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

import org.apache.spark.sql.types.IntegerType

import org.apache.spark.sql.types.DoubleType

import org.apache.spark.sql.types.\_

// Definir el esquema

val schema = StructType(Array(

StructField("PassengerId", DoubleType, true),

StructField("Survived", DoubleType, true),

StructField("Pclass", DoubleType, true),

StructField("Name", StringType, true),

StructField("Sex", StringType, true),

StructField("Age", DoubleType, true),

StructField("SibSp", DoubleType, true),

StructField("Parch", DoubleType, true),

StructField("Ticket", StringType, true),

StructField("Fare", DoubleType, true),

StructField("Cabin", StringType, true),

StructField("Embarked", StringType, true)

))

// Cargar el conjunto de datos del Titanic

val inicDF = spark.read.schema(schema).option("header", "true").csv("titanic.csv")

//eliminar filas con nulos

val titanicDF = inicDF.na.drop()

// Indexar y codificar variables categóricas

val indexer = new StringIndexer()

.setInputCols(Array("Sex", "Embarked"))

.setOutputCols(Array("SexIndex", "EmbarkedIndex"))

val assembler = new VectorAssembler()

.setInputCols(Array("Pclass", "SexIndex", "EmbarkedIndex", "Age", "SibSp", "Parch", "Fare"))

.setOutputCol("features")

// Definir el clasificador

val dt = new DecisionTreeClassifier()

.setLabelCol("Survived")

.setFeaturesCol("features")

// Construir el pipeline

val pipeline = new Pipeline()

.setStages(Array(indexer, assembler, dt))

// Dividir los datos

val Array(trainData, validationData, testData) = titanicDF.randomSplit(Array(0.6, 0.2, 0.2), seed = 1234)

// métrica de evaluación

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("Survived")

.setPredictionCol("prediction")

.setMetricName("accuracy")

// cuadrícula de parámetros para la optimización

val paramGrid = new ParamGridBuilder()

.addGrid(dt.maxDepth, Array(3, 5, 7))

.addGrid(dt.maxBins, Array(24, 32, 40))

.build()

// validador cruzado

val crossValidator = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid.build())

.setNumFolds(5)

// Entrenar el modelo

val cvModel = crossValidator.fit(trainData)

// Evaluar el modelo en el conjunto de validación

val validationPredictions = cvModel.transform(validationData)

val validationAccuracy = evaluator.evaluate(validationPredictions)

println(s"Accuracy on validation set = $validationAccuracy")

// Hacer predicciones en el conjunto de prueba

val testPredictions = cvModel.transform(testData)

val testAccuracy = evaluator.evaluate(testPredictions)

println(s"Accuracy on test set = $testAccuracy")

// Detener la sesión de Spark

spark.stop()

Random forest para regresion:

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.regression.RandomForestRegressor

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, StandardScaler}

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.Pipeline

import org.apache.spark.sql.types.DoubleType

// Cargar el conjunto de datos

val inicDF = spark.read.option("header", "true").csv("HousingData.csv")

// Convertir las columnas a Double

val doubleDF = inicDF.select(inicDF.columns.map(col(\_) cast DoubleType): \_\*)

//eliminar filas con nulos

val housingDF = doubleDF.na.drop()

// Dividir el dataset en conjuntos de entrenamiento, validación y prueba

val Array(trainData, valData, testData) = housingDF.randomSplit(Array(0.6, 0.2, 0.2), seed = 123)

// Definir el modelo Random Forest

val rf = new RandomForestRegressor()

.setLabelCol("MEDV")

.setFeaturesCol("scaledFeatures")

// Construir el ensamblador de características

val assembler = new VectorAssembler()

.setInputCols(housingDF.columns.dropRight(1)) // todas las columnas menos la última (target)

.setOutputCol("features")

// Construir el escalador de características

val scaler = new StandardScaler()

.setInputCol("features")

.setOutputCol("scaledFeatures")

.setWithStd(true)

.setWithMean(true)

// Construir la pipeline

val pipeline = new Pipeline()

.setStages(Array(assembler, scaler, rf))

// Construir la rejilla de parámetros

val paramGrid = new ParamGridBuilder()

.addGrid(rf.numTrees, Array(10, 20, 30))

.addGrid(rf.maxDepth, Array(5, 10, 15))

.build()

// Configurar la evaluación

val evaluator = new RegressionEvaluator()

.setLabelCol("MEDV")

.setPredictionCol("prediction")

.setMetricName("rmse")

// Configurar la validación cruzada para la selección del modelo

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid.build())

.setTrainRatio(0.8)

// Ajustar el modelo utilizando TrainValidationSplit

val model = trainValidationSplit.fit(trainData)

// Evaluar el modelo en el conjunto de validación

val results = model.transform(valData)

val rmse = evaluator.evaluate(results)

println(s"Root Mean Squared Error on validation set = $rmse")

// Detener la sesión de Spark

spark.stop()

Regresion lineal:

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.regression.LinearRegression

import org.apache.spark.ml.evaluation.RegressionEvaluator

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

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.ml.Pipeline

import org.apache.spark.sql.types.DoubleType

// Cargar el conjunto de datos

val inicDF = spark.read.option("header", "true").csv("HousingData.csv")

// Convertir las columnas a Double

val doubleDF = inicDF.select(inicDF.columns.map(col(\_) cast DoubleType): \_\*)

//eliminar filas con nulos

val housingDF = doubleDF.na.drop()

// Dividir el dataset en conjuntos de entrenamiento, validación y prueba

val Array(trainData, valData, testData) = housingDF.randomSplit(Array(0.6, 0.2, 0.2), seed = 123)

// Definir el modelo de regresión lineal

val lr = new LinearRegression()

.setLabelCol("MEDV")

.setFeaturesCol("features")

// Construir el ensamblador de características

val assembler = new VectorAssembler()

.setInputCols(housingDF.columns.dropRight(1)) // todas las columnas menos la última (target)

.setOutputCol("features")

// Construir la pipeline

val pipeline = new Pipeline().setStages(Array(assembler, lr))

// Construir la rejilla de parámetros

val paramGrid = new ParamGridBuilder()

.addGrid(lr.regParam, Array(0.01, 0.1, 1.0))

.addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))

.build()

// Configurar la evaluación

val evaluator = new RegressionEvaluator()

.setLabelCol("MEDV")

.setPredictionCol("prediction")

.setMetricName("rmse")

// Configurar la validación cruzada para la selección del modelo

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid.build())

.setTrainRatio(0.8)

// Ajustar el modelo utilizando TrainValidationSplit

val model = trainValidationSplit.fit(trainData)

// Evaluar el modelo en el conjunto de validación

val results = model.transform(valData)

val rmse = evaluator.evaluate(results)

println(s"Root Mean Squared Error on validation set = $rmse")

// Detener la sesión de Spark

spark.stop()

Regresion logistica:

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer}

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.sql.types.IntegerType

import org.apache.spark.sql.types.DoubleType

import org.apache.spark.sql.types.\_

import org.apache.spark.ml.feature.StringIndexerModel

// Definir el esquema

val schema = StructType(Array(

StructField("PassengerId", DoubleType, true),

StructField("Survived", DoubleType, true),

StructField("Pclass", DoubleType, true),

StructField("Name", StringType, true),

StructField("Sex", StringType, true),

StructField("Age", DoubleType, true),

StructField("SibSp", DoubleType, true),

StructField("Parch", DoubleType, true),

StructField("Ticket", StringType, true),

StructField("Fare", DoubleType, true),

StructField("Cabin", StringType, true),

StructField("Embarked", StringType, true)

))

// Cargar el conjunto

val inicDF = spark.read.schema(schema).option("header", "true").csv("titanic.csv")

//eliminar filas con nulos

val titanicDF = inicDF.na.drop()

// Convertir variables categóricas a numéricas

val genderIndexer = new StringIndexer()

.setInputCol("Sex")

.setOutputCol("SexIndex")

.fit(titanicDF)

val embarkedIndexer = new StringIndexer()

.setInputCol("Embarked")

.setOutputCol("EmbarkedIndex")

.fit(titanicDF)

val assembler = new VectorAssembler()

.setInputCols(Array("Pclass", "SexIndex", "Age", "SibSp", "Parch", "Fare", "EmbarkedIndex"))

.setOutputCol("features")

// Convertir variables categóricas a numéricas

val genderIndexerModel: StringIndexerModel = genderIndexer.fit(titanicDF)

val embarkedIndexerModel: StringIndexerModel = embarkedIndexer.fit(titanicDF)

val transformedDF = assembler.transform(embarkedIndexerModel.transform(genderIndexerModel.transform(titanicDF)))

val labeledDF = transformedDF.withColumnRenamed("Survived", "label")

// Dividir el dataset en conjuntos

val Array(trainData, valData, testData) = labeledDF.randomSplit(Array(0.6, 0.2, 0.2), seed = 123)

// regresión logística

val lr = new LogisticRegression()

// rejilla de parámetros

val paramGrid = new ParamGridBuilder()

.addGrid(lr.regParam, Array(0.01, 0.1, 0.3))

.addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))

.addGrid(lr.maxIter, Array(10, 100))

.build()

// evaluador

val evaluator = new BinaryClassificationEvaluator()

.setMetricName("areaUnderROC")

// validación cruzada para la selección del modelo

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(lr)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid.build())

.setTrainRatio(0.8)

// Ajustar el modelo u

val model = trainValidationSplit.fit(trainData)

// Evaluar el modelo en el conjunto de validación

val results = model.transform(valData)

val areaUnderROC = evaluator.evaluate(results)

println(s"Area under ROC on validation set = $areaUnderROC")

spark.stop()

SVM clasificacion:

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.classification.LinearSVC

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer}

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.sql.types.\_

import org.apache.spark.ml.feature.StringIndexerModel

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// Definir el esquema

val schema = StructType(Array(

StructField("PassengerId", DoubleType, true),

StructField("Survived", DoubleType, true),

StructField("Pclass", DoubleType, true),

StructField("Name", StringType, true),

StructField("Sex", StringType, true),

StructField("Age", DoubleType, true),

StructField("SibSp", DoubleType, true),

StructField("Parch", DoubleType, true),

StructField("Ticket", StringType, true),

StructField("Fare", DoubleType, true),

StructField("Cabin", StringType, true),

StructField("Embarked", StringType, true)

))

// Cargar el conjunto

val inicDF = spark.read.schema(schema).option("header", "true").csv("titanic.csv")

//eliminar filas con nulos

val titanicDF = inicDF.na.drop()

// Convertir variables categóricas a numéricas

val genderIndexer = new StringIndexer()

.setInputCol("Sex")

.setOutputCol("SexIndex")

.fit(titanicDF)

val embarkedIndexer = new StringIndexer()

.setInputCol("Embarked")

.setOutputCol("EmbarkedIndex")

.fit(titanicDF)

val assembler = new VectorAssembler()

.setInputCols(Array("Pclass", "SexIndex", "Age", "SibSp", "Parch", "Fare", "EmbarkedIndex"))

.setOutputCol("features")

// Convertir variables categóricas a numéricas

val genderIndexerModel: StringIndexerModel = genderIndexer.fit(titanicDF)

val embarkedIndexerModel: StringIndexerModel = embarkedIndexer.fit(titanicDF)

val transformedDF = assembler.transform(embarkedIndexerModel.transform(genderIndexerModel.transform(titanicDF)))

val labeledDF = transformedDF.withColumnRenamed("Survived", "label")

// Dividir el dataset en 3 conjuntos

val Array(trainData, valData, testData) = labeledDF.randomSplit(Array(0.6, 0.2, 0.2), seed = 123)

// modelo SVM

val svm = new LinearSVC()

// rejilla de parámetros

val paramGrid = new ParamGridBuilder()

.addGrid(svm.regParam, Array(0.01, 0.1, 1.0))

.addGrid(svm.maxIter, Array(10, 100))

.build()

// Configurar la evaluación

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("accuracy")

// Configurar la validación cruzada

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(svm)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid.build())

.setTrainRatio(0.8)

// Ajustar el modelo

val model = trainValidationSplit.fit(trainData)

// Evaluar el modelo en el conjunto de validación

val results = model.transform(valData)

val accuracy = evaluator.evaluate(results)

println(s"Accuracy on validation set = $accuracy")

spark.stop()

GBT clasificacion:

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.classification.{GBTClassifier, GBTClassificationModel}

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer}

import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

import org.apache.spark.sql.types.\_

import org.apache.spark.ml.feature.StringIndexerModel

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// Definir el esquema

val schema = StructType(Array(

StructField("PassengerId", DoubleType, true),

StructField("Survived", DoubleType, true),

StructField("Pclass", DoubleType, true),

StructField("Name", StringType, true),

StructField("Sex", StringType, true),

StructField("Age", DoubleType, true),

StructField("SibSp", DoubleType, true),

StructField("Parch", DoubleType, true),

StructField("Ticket", StringType, true),

StructField("Fare", DoubleType, true),

StructField("Cabin", StringType, true),

StructField("Embarked", StringType, true)

))

// Cargar el conjunto

val inicDF = spark.read.schema(schema).option("header", "true").csv("titanic.csv")

//eliminar filas con nulos

val titanicDF = inicDF.na.drop()

// Preprocesamiento: Convertir variables categóricas a numéricas

val genderIndexer = new StringIndexer()

.setInputCol("Sex")

.setOutputCol("SexIndex")

.fit(titanicDF)

val embarkedIndexer = new StringIndexer()

.setInputCol("Embarked")

.setOutputCol("EmbarkedIndex")

.fit(titanicDF)

val assembler = new VectorAssembler()

.setInputCols(Array("Pclass", "SexIndex", "Age", "SibSp", "Parch", "Fare", "EmbarkedIndex"))

.setOutputCol("features")

// Convertir variables categóricas a numéricas

val genderIndexerModel: StringIndexerModel = genderIndexer.fit(titanicDF)

val embarkedIndexerModel: StringIndexerModel = embarkedIndexer.fit(titanicDF)

val transformedDF = assembler.transform(embarkedIndexerModel.transform(genderIndexerModel.transform(titanicDF)))

val labeledDF = transformedDF.withColumnRenamed("Survived", "label")

// Dividir el dataset en conjuntos de entrenamiento, validación y prueba

val Array(trainData, valData, testData) = labeledDF.randomSplit(Array(0.6, 0.2, 0.2), seed = 123)

// Definir el modelo GBT

val gbt = new GBTClassifier()

.setLabelCol("label")

.setFeaturesCol("features")

// Construir la pipeline

val pipeline = new Pipeline()

.setStages(Array( gbt))

// Construir la rejilla de parámetros

val paramGrid = new ParamGridBuilder()

.addGrid(gbt.maxDepth, Array(3, 5))

.addGrid(gbt.maxBins, Array(24, 32))

.addGrid(gbt.maxIter, Array(10, 20))

.build()

// Configurar la evaluación

val evaluator = new MulticlassClassificationEvaluator()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("accuracy")

// Configurar la validación cruzada para la selección del modelo

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid.build())

.setTrainRatio(0.8)

// Ajustar el modelo utilizando TrainValidationSplit

val model = trainValidationSplit.fit(trainData)

// Evaluar el modelo en el conjunto de validación

val results = model.transform(valData)

val accuracy = evaluator.evaluate(results)

println(s"Accuracy on validation set = $accuracy")

// Detener la sesión de Spark

spark.stop()

Listas:

//listas con los tipos de atributos

val listaAtributosNumericos = List("age","wage\_per\_hour","capital\_gains","capital\_losses","dividends\_from\_stocks","total\_person\_earnings","num\_persons\_worked\_for\_employer","own\_business\_or\_self\_employed","weeks\_worked\_in\_year","year")

val listaAtributosCategoricos = List("industry\_code","occupation\_code","class\_of\_worker","education","enrolled\_in\_edu\_last\_wk","marital\_status","major\_industry\_code","major\_occupation\_code","member\_of\_labor\_union","race","sex","full\_or\_part\_time\_employment\_status","reason\_for\_unemployment","hispanic\_Origin","tax\_filer\_status","region\_of\_previous\_residence","state\_of\_previous\_residence","detailed\_household\_and\_family\_status","detailed\_household\_summary\_in\_house\_instance\_weight","migration\_code\_change\_in\_msa","migration\_code\_change\_in\_reg","migration\_code\_move\_within\_reg","live\_in\_this\_house\_one\_year\_ago","migration\_prev\_res\_in\_sunbelt","family\_members\_under\_18","country\_of\_birth\_father","country\_of\_birth\_mother","country\_of\_birth\_self","citizenship","fill\_inc\_questionnaire\_for\_veterans\_ad","veterans\_benefits")

Recorrer atributos y mostrar informacion para cada uno:

// recorremos lista atributos numericos

for (nombre\_columna <- listaAtributosNumericos) {

println("Atributo: "+nombre\_columna);

//valores distintos

val valores\_distintos = census\_df.select(nombre\_columna).distinct.count();

println("Valores distintos: "+valores\_distintos);

//resumen del atributo

val describe = census\_df.describe(nombre\_columna).show();

val media = census\_df.describe(nombre\_columna).filter("summary = 'mean'").select(nombre\_columna).first().getString(0)

//valores nulos

val conteoValoresNulos = census\_df.filter(col(nombre\_columna) === "?").count()

println("Nulos: "+conteoValoresNulos);

//distribucion de los valores del atributo

val distribucion = census\_df.groupBy(nombre\_columna).agg(count("\*").alias("cantidad")).orderBy(desc("cantidad"))

distribucion.show()

// guardar en un csv los datos

distribucion.write.mode("overwrite").csv(nombre\_columna)

}

Analizar atributos y guardar informacion en fichero:

//cuantos valores diferentes puede tomar

def numero\_diferentes(nombre\_columna:String): Long = {

val cuenta=census\_df.select(nombre\_columna).distinct().count()

cuenta

}

//extrae cada un de los valores diferentes

def valores\_diferentes(nombre\_columna:String): String = {

val distintos=census\_df.select(nombre\_columna).distinct().collect().map(row => row.getString(0))

val resultado\_linea = distintos.mkString(", ")

resultado\_linea

}

//agrupa y cuenta cuantos valores de cada uno

def numero\_cada\_uno\_diferentes(nombre\_columna:String): DataFrame = {

val numero\_cada\_uno=census\_df.groupBy(nombre\_columna).count().orderBy(desc("count")).withColumnRenamed("count", "cuenta")

numero\_cada\_uno

}

//guardar datos en fichero

def crear\_fichero\_resultados(df:DataFrame,nombre\_columna:String):Unit={

sc.parallelize(df.collect().toSeq,1).saveAsTextFile(nombre\_columna+"\_ordered")

}

var array\_valores\_columnas\_categoricas = Array[String]()

for (nombre\_columna <- listaAtributosCategoricos) {

println("Atributo: "+nombre\_columna);

//diferentes y su distribucion

val numero\_atributo\_diferentes =numero\_diferentes(nombre\_columna)

val valores\_atributo\_diferentes = valores\_diferentes(nombre\_columna)

val numero\_cada\_uno\_diferentes\_atributo = numero\_cada\_uno\_diferentes(nombre\_columna)

numero\_cada\_uno\_diferentes\_atributo.show(numero\_cada\_uno\_diferentes\_atributo.count().toInt, false)

crear\_fichero\_resultados(numero\_cada\_uno\_diferentes\_atributo,nombre\_columna)

//nulos

val conteoValoresNulos = census\_df.filter(col(nombre\_columna) === "?").count()

println("Nulos: "+conteoValoresNulos+"\n\n");

//moda

val moda\_atributo = numero\_cada\_uno\_diferentes\_atributo.first().getString(0)

var escribir=nombre\_columna+": "+valores\_atributo\_diferentes

array\_valores\_columnas\_categoricas=array\_valores\_columnas\_categoricas:+escribir

}

//lo escribe en fichero

sc.parallelize(array\_valores\_columnas\_categoricas.toSeq,1).saveAsTextFile("resumen")

Convertir valores numericos a intervalos en string:

object CleanDataframe {

def cleanDataframe(census\_df: DataFrame): DataFrame = {

//convertimos valores a intervalos en string

val intervalosAge = Array(0, 18, 35, 50, 70, 100)

val labelsAge = Array("0-18", "19-35", "36-50", "51-70", "71-100")

var census\_df\_nuevo = census\_df.withColumn("age-converted",

when(col("age").between(intervalosAge(0), intervalosAge(1)), labelsAge(0))

.when(col("age").between(intervalosAge(1), intervalosAge(2)), labelsAge(1))

.when(col("age").between(intervalosAge(2), intervalosAge(3)), labelsAge(2))

.when(col("age").between(intervalosAge(3), intervalosAge(4)), labelsAge(3))

.when(col("age").between(intervalosAge(4), intervalosAge(5)), labelsAge(4))

.otherwise("Out")

)

}

Correlaccion entre atributos numericos:

//correlaccion atributos numericos---------------------

for (i <- 0 until listaAtributosNumericos.length) {

//busco el primero

val columna\_actual = listaAtributosNumericos(i)

//busco el resto

for (j <- (i + 1) until listaAtributosNumericos.length) {

val siguiente\_columna = listaAtributosNumericos(j)

val assembler = new org.apache.spark.ml.feature.VectorAssembler().setInputCols(Array(columna\_actual, siguiente\_columna)).setOutputCol("features")

val dfVector = assembler.transform(census\_df)

//hago matriz

val correlacionMatrix = Correlation.corr(dfVector, "features").head().getAs[DenseMatrix](0)

//coger valor de la posicion 0,1

val valor = correlacionMatrix.apply(0, 1)

println("Correlaccion entre "+columna\_actual+" con "+siguiente\_columna+" es: "+valor)

}

}

Correlaccion entre atributos categoricos:

//correlaccion categoricos----------------

var correlaccion\_categoricas=Array[String]()

def comprobarValoresMayores(df: DataFrame):Long= {

val columnNames = df.columns

//obtenemos maximo de cada columna

val maxValues = columnNames.map(col => df.agg(max(col)).collect()(0)(0).asInstanceOf[Long])

val maxAmongMaxValues = maxValues.max

maxAmongMaxValues

}

//para cada atributo categorico

for (i <- 0 until listaAtributosCategoricos.length) {

val columna\_actual = listaAtributosCategoricos(i)

//compruebo su correlacion con el resto de atributos

for (j <- (i + 1) until listaAtributosCategoricos.length) {

val siguiente\_columna = listaAtributosCategoricos(j)

//se hace la tabla de contingecnia

var tablaContingencia = census\_df.groupBy(columna\_actual).pivot(siguiente\_columna).count().na.fill(0)

tablaContingencia = tablaContingencia.drop(columna\_actual)

tablaContingencia.show()

tablaContingencia.printSchema()

//se guardan los resultados

val resultado = comprobarValoresMayores(tablaContingencia)

val fila = columna\_actual+";"+siguiente\_columna+";"+resultado

println(fila)

correlaccion\_categoricas = correlaccion\_categoricas :+ fila

val nombre=columna\_actual+"-"+siguiente\_columna

tablaContingencia.write.mode("overwrite").csv(nombre)

}

}

sc.parallelize(correlaccion\_categoricas.toSeq,1).saveAsTextFile("correlacion\_categoricas")

Reemplazo outliers por limite:

//reemplazo de outliers por limite

for (nombre\_columna <- listaAtributosNumericos) {

val percentiles = census\_df.stat.approxQuantile(nombre\_columna, Array(0.05, 0.95), 0.01)

val limiteInferior = percentiles(0)

val limiteSuperior = percentiles(1)

census\_df = census\_df.withColumn(nombre\_columna, when(col(nombre\_columna) < limiteInferior, limiteInferior).otherwise(when(col(nombre\_columna) > limiteSuperior, limiteSuperior).otherwise(col(nombre\_columna))))

}

Reemplazo nulos por la moda:

//reemplazo de nulos por la moda

for (nombre\_columna <- listaAtributosCategoricos) {

val numero\_cada\_uno=census\_df.groupBy(nombre\_columna).count().orderBy(desc("count")).withColumnRenamed("count", "cuenta")

val moda\_atributo = numero\_cada\_uno.first().getString(0)

census\_df = census\_df.withColumn(nombre\_columna, when(col(nombre\_columna) === "?", moda\_atributo).otherwise(col(nombre\_columna)))

}

Invocar funciones de otros archivos:

//cargamos dataset: invocar funciones de otros archivos----------------------------

:load TransformDataframeV2.scala

:load CleanDataframe.scala

import TransformDataframeV2.\_

import CleanDataframe.\_

val census\_df\_limpio=cleanDataframe(census\_df)

val trainCensusDFProcesado = transformDataFrame(census\_df\_limpio)

GBT buscar mejores parametros y guardar modelo:

//validacion cruzada y parametros---------------------------

//gradient boosted tree

val gbt = new GBTClassifier().setLabelCol("label").setFeaturesCol("features")

val paramGrid = new ParamGridBuilder().addGrid(gbt.maxDepth, Array(3)).addGrid(gbt.maxIter, Array(20)).addGrid(gbt.maxBins, Array(50)).build()

val evaluator = new MulticlassClassificationEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("accuracy")

val pipeline = new Pipeline().setStages(Array(gbt))

val cv = new CrossValidator().setEstimator(pipeline).setEvaluator(evaluator).setEstimatorParamMaps(paramGrid).setNumFolds(2)

val cvModel = cv.fit(trainCensusDFProcesado)

val bestPipelineModel = cvModel.bestModel.asInstanceOf[PipelineModel]

val bestGBTModel = bestPipelineModel.stages(0).asInstanceOf[GBTClassificationModel]

println(s"Best max depth: ${bestGBTModel.getMaxDepth}")

println(s"Best max iterations: ${bestGBTModel.getMaxIter}")

println(s"Best max bins: ${bestGBTModel.getMaxBins}")

//utilizamos mejores parametros----------------------

val GBT = new GBTClassifier().setFeaturesCol("features").setLabelCol("label").setMaxIter(bestGBTModel.getMaxIter).

setMaxDepth(bestGBTModel.getMaxDepth).

setMaxBins(bestGBTModel.getMaxBins).

setMinInstancesPerNode(1).

setMinInfoGain(0.0).

setCacheNodeIds(false).

setCheckpointInterval(10)

val GBTModel\_D =GBT.fit(trainCensusDFProcesado)

//guardar modelo

GBTModel\_D.write.overwrite().save(PATH + "modeloGBT")

Random forest buscar mejores parametros y guardar modelo:

//RF

val rf = new RandomForestClassifier().setLabelCol("label").setFeaturesCol("features")

val paramGrid = new ParamGridBuilder().addGrid(rf.maxDepth, Array(10)).addGrid(rf.numTrees, Array(140)).addGrid(rf.maxBins, Array(150)).build()

val evaluator = new MulticlassClassificationEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("accuracy")

val cv = new CrossValidator().setEstimator(rf).setEvaluator(evaluator).setEstimatorParamMaps(paramGrid).setNumFolds(2)

val cvModel = cv.fit(trainCensusDFProcesado)

import org.apache.spark.ml.classification.RandomForestClassificationModel

val bestModel = cvModel.bestModel.asInstanceOf[RandomForestClassificationModel]

println(s"Number of Trees: ${bestModel.getNumTrees}")

println(s"Max Depth: ${bestModel.getMaxDepth}")

println(s"Max bins: ${bestModel.getMaxBins}")

//utilizamos mejores parametros------------------------------

val RF = new RandomForestClassifier().setFeaturesCol("features").

setLabelCol("label").

setNumTrees(bestModel.getNumTrees).

setMaxDepth(bestModel.getMaxDepth).

setMaxBins(bestModel.getMaxBins).

setMinInstancesPerNode(1).

setMinInfoGain(0.0).

setCacheNodeIds(false).

setCheckpointInterval(10)

val RFModel\_D =RF.fit(trainCensusDFProcesado)

RFModel\_D.toDebugString

//guardamos modelo

RFModel\_D.write.overwrite().save(PATH + "modeloRF")

Cargar un modelo:

val loadedGBTcensusModel = GBTClassificationModel.load(PATH + "modeloGBT")

Evaluar metricas sobre el modelo cargado:

//evaluar metricas

val predictionsAndLabelsDF\_GBT = loadedGBTcensusModel.transform(testCensusDF).select("prediction", "label","rawPrediction", "probability")

predictionsAndLabelsDF\_GBT.show()

val rm\_GBT = new RegressionMetrics(predictionsAndLabelsDF\_GBT.rdd.map(x => (x(0).asInstanceOf[Double], x(1).asInstanceOf[Double])))

println("Test metrics:")

println("Test Explained Variance: ")

println(rm\_GBT.explainedVariance)

println("R² Coefficient")

println(rm\_GBT.r2)

//metricas------------------------------

val predictionsGBT = loadedGBTcensusModel.transform(testCensusDF).select("prediction").rdd.map(\_.getDouble(0))

val labelsGBT = loadedGBTcensusModel.transform(testCensusDF).select("label").rdd.map(\_.getDouble(0))

val metrics = new MulticlassMetrics(predictionsGBT.zip(labelsGBT))

println("Confusion matrix:")

println(metrics.confusionMatrix)

val accuracy = metrics.accuracy

println("Summary Statistics")

println(f"Accuracy = $accuracy%1.4f")

val labels = metrics.labels

labels.foreach {l => val pl = metrics.precision(l)

println(f"PrecisionByLabel($l) = $pl%1.4f")}

labels.foreach {l => val fpl = metrics.falsePositiveRate(l)

println(f"falsePositiveRate($l) = $fpl%1.4f")}

labels.foreach {l => val fpl = metrics.truePositiveRate(l)

println(f"truePositiveRate($l) = $fpl%1.4f")}

Funcion para transformar dataframe:

object TransformDataframeV2 {

def transformDataFrame(census\_df: DataFrame): DataFrame = {

//filtramos todas menos income

val attributeColumns = census\_df.columns.toSeq.filter(\_ != "income").toArray

//añadimos sufijo num

val outputColumns = attributeColumns.map(\_ + "-num").toArray

//aplicamos string indexer

val siColumns= new StringIndexer().setInputCols(attributeColumns).setOutputCols(outputColumns).setStringOrderType("alphabetDesc")

val simColumns = siColumns.fit(census\_df)

val censusDFnumeric = simColumns.transform(census\_df).drop(attributeColumns:\_\*)

//creamos vector

val va = new VectorAssembler().setOutputCol("features").setInputCols(outputColumns)

val censusFeaturesClaseDF = va.transform(censusDFnumeric).select("features", "income")

//convertimos income a valores

val indiceClase= new StringIndexer().setInputCol("income").setOutputCol("label").setStringOrderType("alphabetDesc")

val censusFeaturesLabelDF = indiceClase.fit(censusFeaturesClaseDF).transform(censusFeaturesClaseDF).drop("income")

censusFeaturesLabelDF

}

}

Importacion de diferentes paquetes:

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.ml.classification.{DecisionTreeClassifier,RandomForestClassifier, GBTClassificationModel, GBTClassifier}

import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.{OneHotEncoder, OneHotEncoderModel, StringIndexer, StringIndexerModel, VectorAssembler}

import org.apache.spark.ml.linalg.DenseVector

import org.apache.spark.ml.{Pipeline, PipelineModel}

import org.apache.spark.ml.stat.ChiSquareTest

import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}

import org.apache.spark.mllib.evaluation.{MulticlassMetrics, RegressionMetrics, BinaryClassificationMetrics}

import org.apache.spark.sql.{DataFrame, SparkSession,Row}

import org.apache.spark.sql.types.{IntegerType, StringType, DoubleType, StructField, StructType}

Otra forma cargar datos:

//carga de datos

println("\nCARGA DE LOS DATOS")

val PATH = "/home/usuario/Regresion/MiniProyecto2/"

val ARCHIVO = "hour.csv"

val bikeDF = spark.read.format("csv").

option("inferSchema", true).

option("header", true).

load(PATH + ARCHIVO)

Contar y ordenar:

println("Número de alquileres por época del año:")

bikeDF.

groupBy("season").

count().

orderBy(asc("season")).

withColumnRenamed("count", "cuenta").

show()

Eliminar varias columnas:

// Eliminar atributos no necesarios

val columnasAEliminar = Seq(

"instant",

"dteday",

"atemp",

"windspeed",

"casual",

"registered"

)

val nuevoDF = bikeDF.drop(columnasAEliminar: \_\*)

nuevoDF.count()

Partir los datos y guardarlos:

//particion datos

println("\nPARTICIÓN DE LOS DATOS")

val splitSeed = 123

val Array(trainingData, testData) = nuevoDF.

randomSplit(Array(0.7, 0.3), splitSeed)

testData.write.mode("overwrite").csv(PATH + "testData")

println("Conjunto de pruebas guardado")

testData.show(5)

Pruebas con varios modelos de regresion:

val featureCols = Array(

"holiday",

"workingday",

"temp",

"hum",

"season",

"yr",

"mnth",

"hr",

"weekday",

"weathersit"

)

val assembler = new VectorAssembler().

setInputCols(featureCols).

setOutputCol("features")

// validacion cruzada sobre regresion lineal

println("\nVALIDACIÓN CRUZADA PARA REGRESIÓN LINEAL")

val lr = new LinearRegression().

setLabelCol("cnt").

setFeaturesCol("features")

val pipeline = new Pipeline().

setStages(Array(assembler, lr))

val paramGrid = new ParamGridBuilder().

addGrid(lr.regParam, Array(0.1,0.2,0.3)).

addGrid(lr.elasticNetParam, Array(0.5,0.75,1)).

build()

val evaluator = new RegressionEvaluator()

.setLabelCol("cnt")

.setPredictionCol("prediction")

.setMetricName("rmse")

val cv = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(3)

val cvModel = cv.fit(trainingData)

println("\nPARÁMETROS MEJOR MODELO REGRESIÓN LINEAL")

val bestModel = cvModel.bestModel.asInstanceOf[PipelineModel]

val lrModel = bestModel.stages(1).asInstanceOf[LinearRegressionModel]

println(s"""Parámetros del mejor modelo:

regParam = ${lrModel.getRegParam}, elasticNetParam = ${lrModel.getElasticNetParam}

""")

lrModel.write.overwrite().save(PATH + "best\_LinearRegressionModel")

println("Mejor modelo regresión lineal guardado")

// validacion cruzada sobre gbt regressor

println("\nVALIDACIÓN CRUZADA PARA REGRESOR GBT")

val gbt = new GBTRegressor().

setLabelCol("cnt").

setFeaturesCol("features")

val pipeline1 = new Pipeline().

setStages(Array(assembler, gbt))

val paramGrid1 = new ParamGridBuilder().

addGrid(gbt.maxDepth, Array(5, 8, 11)).

addGrid(gbt.maxIter, Array(10, 15, 20)).

build()

val evaluator1 = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val cv1 = new CrossValidator().

setEstimator(pipeline1).

setEvaluator(evaluator1).

setEstimatorParamMaps(paramGrid1).

setNumFolds(3)

val cvModel1 = cv1.fit(trainingData)

println("\nPARÁMETROS MEJOR MODELO REGRESOR GBT")

val bestModel1 = cvModel1.bestModel.asInstanceOf[PipelineModel]

val gbtModel = bestModel1.stages(1).asInstanceOf[GBTRegressionModel]

println(s"""Parámetros del mejor modelo:

maxDepth = ${gbtModel.getMaxDepth}, maxIter = ${gbtModel.getMaxIter}

""")

gbtModel.write.overwrite().save(PATH + "best\_GBTRegressionModel")

println("Mejor modelo regresor GBT guardado")

//validacion cruzada sobre random forest

println("\nVALIDACIÓN CRUZADA PARA REGRESOR RF")

val rf = new RandomForestRegressor().

setLabelCol("cnt").

setFeaturesCol("features")

val pipeline3 = new Pipeline().

setStages(Array(assembler, rf))

val paramGrid3 = new ParamGridBuilder().

addGrid(rf.maxDepth, Array(5,8,11)).

addGrid(rf.numTrees, Array(5,10,15)).

build()

val evaluator3 = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val cv3 = new CrossValidator().

setEstimator(pipeline3).

setEvaluator(evaluator3).

setEstimatorParamMaps(paramGrid3).

setNumFolds(3)

val cvModel3 = cv3.fit(trainingData)

println("\nPARÁMETROS MEJOR MODELO REGRESOR RF")

val bestModel3 = cvModel3.bestModel.asInstanceOf[PipelineModel]

val rfModel = bestModel3.stages(1).asInstanceOf[RandomForestRegressionModel]

println(s"""Parámetros del mejor modelo:

maxDepth = ${rfModel.getMaxDepth}, numTrees = ${rfModel.getNumTrees}

""")

rfModel.write.overwrite().save(PATH + "best\_RandomForestRegressionModel")

println("Mejor modelo regresor RF guardado")

//validacion cruzada sobre decissiontree

println("\nVALIDACIÓN CRUZADA PARA REGRESOR DT")

val dt = new DecisionTreeRegressor().

setLabelCol("cnt").

setFeaturesCol("features")

val pipeline2 = new Pipeline().

setStages(Array(assembler, dt))

val paramGrid2 = new ParamGridBuilder().

addGrid(dt.maxDepth, Array(5,10,15)).

addGrid(dt.maxBins, Array(16, 32, 64)).

build()

val evaluator2 = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val cv2 = new CrossValidator().

setEstimator(pipeline2).

setEvaluator(evaluator2).

setEstimatorParamMaps(paramGrid2).

setNumFolds(3)

val cvModel2 = cv2.fit(trainingData)

println("\nPARÁMETROS MEJOR MODELO REGRESOR DT")

val bestModel2 = cvModel2.bestModel.asInstanceOf[PipelineModel]

val dtModel = bestModel2.stages(1).asInstanceOf[DecisionTreeRegressionModel]

println(s"""Parámetros del mejor modelo:

maxDepth = ${dtModel.getMaxDepth}, maxBins = ${dtModel.getMaxBins}

""")

dtModel.write.overwrite().save(PATH + "best\_DecisionTreeRegressionModel")

println("Mejor modelo regresor DT guardado")

Evaluar los diferentes modelos:

//carga de datos

val PATH = "/home/usuario/Regresion/MiniProyecto2/"

val ARCHIVO\_TEST = "testData"

val testRaw = spark.read.format("csv").

option("inferSchema", true).

load(PATH + ARCHIVO\_TEST).

toDF(

"season",

"yr",

"mnth",

"hr",

"holiday",

"weekday",

"workingday",

"weathersit",

"temp",

"hum",

"cnt"

)

val featureCols = Array(

"holiday",

"workingday",

"temp",

"hum",

"season",

"yr",

"mnth",

"hr",

"weekday",

"weathersit"

)

val assembler = new VectorAssembler().

setInputCols(featureCols).

setOutputCol("features")

val testData = assembler.transform(testRaw)

testData.show(5)

//evaluar modelo linear regression

println("\nEVALUACIÓN DE MODELO LinearRegressionModel")

val lrModel = LinearRegressionModel.load(PATH + "best\_LinearRegressionModel")

val evaluator = new RegressionEvaluator()

.setLabelCol("cnt")

.setPredictionCol("prediction")

.setMetricName("rmse")

val predictions = lrModel.transform(testData)

val rmse = evaluator.evaluate(predictions)

val metrics = evaluator.getMetrics(predictions)

println(s"MSE: ${metrics.meanSquaredError}")

println(s"R²: ${metrics.r2}")

println(s"root MSE: ${metrics.rootMeanSquaredError}")

println(s"Mean Absolute Error: ${metrics.meanAbsoluteError}")

//evaluar modelo gbt regression

println("\nEVALUACIÓN DE MODELO GBTRegressionModel")

val gbtModel = GBTRegressionModel.load(PATH + "best\_GBTRegressionModel")

val evaluator1 = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val predictions1 = gbtModel.transform(testData)

val rmse1 = evaluator1.evaluate(predictions1)

val metrics1 = evaluator1.getMetrics(predictions1)

println(s"MSE: ${metrics1.meanSquaredError}")

println(s"R²: ${metrics1.r2}")

println(s"root MSE: ${metrics1.rootMeanSquaredError}")

println(s"Mean Absolute Error: ${metrics1.meanAbsoluteError}")

//evaluar modelo randomforest

println("\nEVALUACIÓN DE MODELO RandomForestRegressionModel")

val rfModel = RandomForestRegressionModel.

load(PATH + "best\_RandomForestRegressionModel")

val evaluator3 = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val predictions3 = rfModel.transform(testData)

val rmse3 = evaluator3.evaluate(predictions3)

val metrics3 = evaluator3.getMetrics(predictions3)

println(s"MSE: ${metrics3.meanSquaredError}")

println(s"R²: ${metrics3.r2}")

println(s"root MSE: ${metrics3.rootMeanSquaredError}")

println(s"Mean Absolute Error: ${metrics3.meanAbsoluteError}")

//evaluar modelo decission tree regresor

println("\nEVALUACIÓN DE MODELO DecisionTreeRegressor")

val dtModel = DecisionTreeRegressionModel.

load(PATH + "best\_DecisionTreeRegressionModel")

val evaluator2 = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val predictions2 = dtModel.transform(testData)

val rmse2 = evaluator2.evaluate(predictions2)

val metrics2 = evaluator2.getMetrics(predictions2)

println(s"MSE: ${metrics2.meanSquaredError}")

println(s"R²: ${metrics2.r2}")

println(s"root MSE: ${metrics2.rootMeanSquaredError}")

println(s"Mean Absolute Error: ${metrics2.meanAbsoluteError}")

//seleccion del mejor modelo

println("\nSELECCIÓN DEL MEJOR MODELO")

println(s"RMSE en el conjunto de test para mejor modelo de LinearRegression: ${metrics.rootMeanSquaredError}")

println(s"RMSE en el conjunto de test para mejor modelo de GBTRegressor: ${metrics1.rootMeanSquaredError}")

println(s"RMSE en el conjunto de test para mejor modelo de DecisionTreeRegressor: ${metrics2.rootMeanSquaredError}")

println(s"RMSE en el conjunto de test para mejor modelo de RandomForestRegressor: ${metrics3.rootMeanSquaredError}")

println("\nGUARDADO DEL MEJOR MODELO: GBTRegressor")

gbtModel.write.overwrite().save(PATH + "modelo")

//evaluacion del mejor modelo

println("\nEVALUACIÓN DEL MEJOR MODELO (GBTRegressionModel)")

val bestModel = GBTRegressionModel.load(PATH + "modelo")

val bestEvaluator = new RegressionEvaluator().

setLabelCol("cnt").

setPredictionCol("prediction").

setMetricName("rmse")

val bestPredictions = bestModel.transform(testData)

val bestRmse = bestEvaluator.evaluate(bestPredictions)

val bestMetrics = bestEvaluator.getMetrics(bestPredictions)

println(s"MSE: ${bestMetrics.meanSquaredError}")

println(s"R²: ${bestMetrics.r2}")

println(s"root MSE: ${bestMetrics.rootMeanSquaredError}")

println(s"Mean Absolute Error: ${bestMetrics.meanAbsoluteError}")