# [SRC] 03.1. Discover spark.ml and spark.mllib - And other Libraries

import org.apache.spark.mllib.util.MLUtils

val data = MLUtils.loadLibSVMFile(sc, "/dataset/spark/sample\_libsvm\_data.txt")

// Split data into training (60%) and test (40%).

val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)

val training = splits(0).cache()

val test = splits(1)

import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}

// Run training algorithm to build the model.

val numIterations = 100

val model = SVMWithSGD.train(training, numIterations)

model.clearThreshold()

val scoreAndLabels = test.map(point => (model.predict(point.features), point.label))

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

val metrics = new BinaryClassificationMetrics(scoreAndLabels)

println("Area under ROC = " + metrics.areaUnderROC())

val modelPath = "/tmp/ml-svm-model-" + scala.util.Random.nextInt

model.save(sc, modelPath)

val sameModel = SVMModel.load(sc, modelPath)

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

// Load training data.

val training = spark.read.format("libsvm").load("/dataset/spark/sample\_linear\_regression\_data.txt")

val lr = new LinearRegression()

.setMaxIter(10)

.setRegParam(0.3)

.setElasticNetParam(0.8)

// Fit the model.

val lrModel = lr.fit(training)

// 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.

val trainingSummary = lrModel.summary

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

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

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

println(s"r2: ${trainingSummary.r2}")

trainingSummary.residuals.show()

# [SRC] 03.2. Wrap-up Basic Statistics and Linear Algebra

import org.apache.spark.mllib.random.RandomRDDs.\_

val n = normalRDD(sc, 1000000, 10)

val h = n.histogram(100)

sc.parallelize(h.\_2).zipWithIndex.toDF.createOrReplaceTempView("h")

val v = n.map(x => 1.0 + 2.0 \* x)

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

val df = sc.parallelize(0 to 10).toDF.withColumn("uniform", rand(seed=10)).withColumn("normal", randn(seed=27))

df.describe().show

df.select(mean("uniform"), min("uniform"), max("uniform")).show()

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}

val observations = sc.parallelize(

Seq(

Vectors.dense(1.0, 10.0, 100.0),

Vectors.dense(2.0, 20.0, 200.0),

Vectors.dense(3.0, 30.0, 300.0)

)

)

val summary = Statistics.colStats(observations)

println(s"Mean: ${summary.mean}") // a dense vector containing the mean value for each column

println(s"Variance: ${summary.variance}") // column-wise variance

println(s"Num Non Zeros: ${summary.numNonzeros}") // number of nonzeros in each column

import org.apache.spark.mllib.linalg.\_

import org.apache.spark.mllib.stat.Statistics

import org.apache.spark.rdd.RDD

val seriesX: RDD[Double] = sc.parallelize(Array(1, 2, 3, 3, 5))

val seriesY: RDD[Double] = sc.parallelize(Array(11, 22, 33, 33, 555))

val correlation = Statistics.corr(seriesX, seriesY, "pearson")

println(s"Correlation is: $correlation")

import org.apache.spark.mllib.linalg.\_

import org.apache.spark.rdd.RDD

import org.apache.spark.mllib.stat.Statistics

val data: RDD[Vector] = sc.parallelize(

Seq(

Vectors.dense(1.0, 10.0, 100.0),

Vectors.dense(2.0, 20.0, 200.0),

Vectors.dense(5.0, 33.0, 366.0))

)

val correlMatrix = Statistics.corr(data, "pearson")

println(correlMatrix.toString)

case class Point(id: Int, v1: Double, v2: Double, v3: Double)

val t = sc.parallelize(Seq(

Point(1, 56, 87, 223),

Point(2, 56, 91, 23),

Point(3, 65, 85, 92),

Point(4, 65, 91, 45),

Point(5, 50, 75, 231),

Point(6, 25, 28, 42),

Point(7, 87, 122, 9),

Point(8, 44, 66, 23),

Point(9, 35, 58, 23))).toDF

t.createOrReplaceTempView("t")

val s1 = t.map(p => (p.getAs[Double]("v1")))

val s2 = t.map(p => (p.getAs[Double]("v2")))

val s3 = t.map(p => (p.getAs[Double]("v3")))

import org.apache.spark.mllib.stat.Statistics

sc.parallelize(Seq(

("s1\_s2", Statistics.corr(s1.rdd, s2.rdd, "pearson")),

("s1\_s3", Statistics.corr(s1.rdd, s3.rdd, "pearson")),

("s2\_s3", Statistics.corr(s2.rdd, s3.rdd, "pearson"))

)).toDF.createOrReplaceTempView("correlations")

import breeze.linalg.{DenseVector => BDV, SparseVector => BSV, Vector => BV, DenseMatrix => BDM}

import breeze.stats.distributions.Poisson

val poi = new Poisson(3.0);

val s = poi.sample(5);

val x = BDV.zeros[Double](5)

x(1) = 2

println(x)

val m = BDM.zeros[Int](5,5)

m(4,::) := BDV(1,2,3,4,5).t

m\*10

println(breeze.linalg.sum(m))

import org.apache.spark.mllib.linalg.{Vector, Vectors}

val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)

val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))

val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))

import breeze.linalg.{DenseVector => BDV, SparseVector => BSV, Vector => BV}

import org.apache.spark.mllib.linalg.{Vector, DenseVector, SparseVector}

def toBreeze(v: Vector): BV[Double] = v match {

case DenseVector(values) => new BDV[Double](values)

case SparseVector(size, indices, values) => {

new BSV[Double](indices, values, size)

}

}

def toSpark(v: BV[Double]) = v match {

case v: BDV[Double] => new DenseVector(v.toArray)

case v: BSV[Double] => new SparseVector(v.length, v.index, v.data)

}

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.regression.LabeledPoint

// Create a labeled point with a positive label and a dense feature vector.

val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))

// Create a labeled point with a negative label and a sparse feature vector.

val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))

import org.apache.spark.mllib.linalg.{Matrix, Matrices}

// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))

val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))

// Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))

val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))

import org.apache.spark.mllib.linalg.{Vectors, Matrices}

import org.apache.spark.mllib.linalg.distributed.{IndexedRowMatrix, IndexedRow}

val rows =

sc.parallelize(Seq(

(0L, Array(1.0, 0.0, 0.0)),

(0L, Array(0.0, 1.0, 0.0)),

(0L, Array(0.0, 0.0, 1.0))

))

.map {

case (i, xs) => IndexedRow(i, Vectors.dense(xs))

}

val indexedRowMatrix = new IndexedRowMatrix(rows)

import org.apache.spark.mllib.linalg.distributed.RowMatrix

val rowMatrix = new RowMatrix(rows.map(\_.vector))

import org.apache.spark.mllib.linalg.{Vectors, Matrices}

import org.apache.spark.mllib.linalg.distributed.{BlockMatrix}

val eye = Matrices.sparse(

3,

3,

Array(0, 1, 2, 3),

Array(0, 1, 2),

Array(1, 1, 1)

)

val blocks = sc.parallelize(

Seq(

((0, 0), eye),

((1, 1), eye),

((2, 2), eye)))

val blockMatrix = new BlockMatrix(blocks, 3, 3, 9, 9)

import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}

val entries = sc.parallelize(

Seq(

(0, 0, 3.0), (2, 0, -5.0), (3, 2, 1.0),

(4, 1, 6.0), (6, 2, 2.0), (8, 1, 4.0))

)

.map{

case (i, j, v) => MatrixEntry(i, j, v)

}

val coordinateMatrix = new CoordinateMatrix(entries, 9, 3)

val localMatrix = Matrices.dense(3, 2, Array(1.0, 2.0, 3.0, 4.0, 5.0, 6.0))

indexedRowMatrix.multiply(localMatrix).rows.collect

// Array(IndexedRow(0,[1.0,4.0]), IndexedRow(0,[2.0,5.0]), IndexedRow(0,[3.0,6.0]))

blockMatrix.multiply(coordinateMatrix.toBlockMatrix(3, 3))

# [SRC] 03.3. Cleanse Data and Engineer Features

import org.apache.spark.ml.feature.Word2Vec

val documentDF = spark.createDataFrame(Seq(

"Hello Packt".split(" "),

"I wish Data Science with Spark is easy".split(" "),

"And now I have a course for this".split(" ")

).map(Tuple1.apply)).toDF("text")

val word2Vec = new Word2Vec()

.setInputCol("text")

.setOutputCol("result")

.setVectorSize(3)

.setMinCount(0)

val model = word2Vec.fit(documentDF)

val result = model.transform(documentDF)

result.select("result").take(3).foreach(println)

import org.apache.spark.ml.feature.StandardScaler

val df = spark.read.format("libsvm").load("/dataset/spark/sample\_libsvm\_data.txt")

val scaler = new StandardScaler()

.setInputCol("features")

.setOutputCol("scaledFeatures")

.setWithStd(true)

.setWithMean(false)

// Compute summary statistics by fitting the StandardScaler.

val scalerModel = scaler.fit(df)

// Normalize each feature to have unit standard deviation.

val scaledData = scalerModel.transform(df)

scaledData.show()

import org.apache.spark.ml.feature.{RegexTokenizer, Tokenizer}

val sentenceDataFrame = spark.createDataFrame(Seq(

(0, "Hi I heard about Spark"),

(1, "I wish Java could use case classes"),

(2, "Logistic,regression,models,are,neat")

)).toDF("label", "sentence")

val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")

val regexTokenizer = new RegexTokenizer()

.setInputCol("sentence")

.setOutputCol("words")

.setPattern("\\W") // alternatively .setPattern("\\w+").setGaps(false)

val tokenized = tokenizer.transform(sentenceDataFrame)

tokenized.select("words", "label").take(3).foreach(println)

val regexTokenized = regexTokenizer.transform(sentenceDataFrame)

regexTokenized.select("words", "label").take(3).foreach(println)

import java.util.Arrays

import org.apache.spark.ml.attribute.{Attribute, AttributeGroup, NumericAttribute}

import org.apache.spark.ml.feature.VectorSlicer

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

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.StructType

val data = Arrays.asList(Row(Vectors.dense(-2.0, 2.3, 0.0)))

val defaultAttr = NumericAttribute.defaultAttr

val attrs = Array("f1", "f2", "f3").map(defaultAttr.withName)

val attrGroup = new AttributeGroup("userFeatures", attrs.asInstanceOf[Array[Attribute]])

val dataset = spark.createDataFrame(data, StructType(Array(attrGroup.toStructField())))

val slicer = new VectorSlicer().setInputCol("userFeatures").setOutputCol("features")

slicer.setIndices(Array(1)).setNames(Array("f3"))

// or slicer.setIndices(Array(1, 2))

// or slicer.setNames(Array("f2", "f3"))

val output = slicer.transform(dataset)

println(output.select("userFeatures", "features").first())

# [SRC] 03.4. Reduce the Dimensionality

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.linalg.distributed.RowMatrix

val filename = "/dataset/wine/wine.data"

val data = sc.textFile(filename).map {line =>

val fields = line.split(",");

Vectors.dense( { fields.drop(1).map(\_.toDouble) } )

}

val dataMatrix = new RowMatrix(data)

println(s"Rows \* Columns: ${dataMatrix.numRows} \* ${dataMatrix.numCols}")

val pca = dataMatrix.computePrincipalComponents(3)

println(pca)

val reducedData = dataMatrix.multiply(pca).rows

val reducedMatrix = new RowMatrix(reducedData)

println(s"Rows \* Columns: ${reducedMatrix.numRows} \* ${reducedMatrix.numCols}")

val svd = dataMatrix.computeSVD(4)

println(s"Left Singular Vectors: ${svd.U}")

println(s"Singular Values: ${svd.s}")

println(s"Right Singular Vectors: ${svd.V}")

val reducedData = dataMatrix.multiply(svd.V).rows

val reducedMatrix = new RowMatrix(reducedData)

println(s"Rows \* Columns: ${reducedMatrix.numRows} \* ${reducedMatrix.numCols}")

import org.apache.spark.ml.feature.PCA

import org.apache.spark.ml.linalg.{Vectors => MlVectors}

val data = Array(

MlVectors.sparse(5, Seq((1, 1.0), (3, 7.0))),

MlVectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),

MlVectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)

)

val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val pca = new PCA()

.setInputCol("features")

.setOutputCol("pcaFeatures")

.setK(3)

.fit(df)

val pcaDF = pca.transform(df)

val pcaFeatures = pcaDF.select("pcaFeatures")

pcaFeatures.show

println(pca.pc)

println(pca.explainedVariance)

# [SRC] 03.5. Pipeline for a Life

import org.apache.spark.ml.Pipeline

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

import org.apache.spark.ml.feature.{HashingTF, Tokenizer}

case class LabeledDocument(id: Long, text: String, label: Double)

case class Document(id: Long, text: String)

val training = sc.parallelize(Seq(

LabeledDocument(0L, "a b c d e spark", 1.0),

LabeledDocument(1L, "b d", 0.0),

LabeledDocument(2L, "spark f g h", 1.0),

LabeledDocument(3L, "hadoop mapreduce", 0.0))).toDF

val tokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")

val hashingTF = new HashingTF().setNumFeatures(1000).setInputCol(tokenizer.getOutputCol).setOutputCol("features")

val lr = new LogisticRegression().setMaxIter(10).setRegParam(0.01)

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

val model = pipeline.fit(training)

val test = sc.parallelize(Seq(

Document(4L, "spark i j k"),

Document(5L, "l m n"),

Document(6L, "mapreduce spark"),

Document(7L, "apache hadoop"))).toDF

model.transform(test).select('id, 'text, 'prediction).collect().foreach(println)

model.write.overwrite().save("/tmp/pipeline")