# [SRC] 08.1. Build the Training and Test Datasets

%sh head /dataset/sentiment/thinknook/sentiment-analysis-dataset.csv

val file = sc.textFile("/dataset/sentiment/thinknook/sentiment-analysis-dataset.csv")

val dataset = file.filter(l => l.charAt(0).isDigit)

// val splits = dataset.randomSplit(Array(0.08, 0.02, 0,9), seed = 11L)

val splits = dataset.randomSplit(Array(0.8, 0.2), seed = 11L)

case class SentimentText(id: Int, label: Double, sentiment: String, text: String)

def parse(line: String): SentimentText = {

val tokens = line.split(",")

tokens(1) match {

case "0" => SentimentText(tokens(0).toInt, 0.0, "NEGATIVE", tokens(3).trim)

case "1" => SentimentText(tokens(0).toInt, 1.0, "POSITIVE", tokens(3).trim)

}

}

val training = splits(0).map(parse).persist

training.count

training.toDF.show

val test = splits(1).map(parse).persist

test.count

test.toDF.show

# [SRC] 08.2. Train a Logistic Regression Model

import org.apache.spark.sql.Dataset

import org.apache.spark.ml.feature.Tokenizer

import org.apache.spark.ml.feature.StopWordsRemover

import org.apache.spark.ml.feature.HashingTF

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val remover = new StopWordsRemover()

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("noStopWords")

val hashingTF = new HashingTF()

.setInputCol(remover.getOutputCol)

.setOutputCol("features")

.setNumFeatures(100)

/\*

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

val word2Vec = new Word2Vec()

.setInputCol("noStopWords")

.setOutputCol("features")

.setVectorSize(100)

.setMinCount(0)

\*/

def featurize(dataset: Dataset[SentimentText]) = {

val words = tokenizer.transform(dataset)

val noStopWords = remover.transform(words)

hashingTF.transform(noStopWords)

/\*

val model = word2Vec.fit(noStopWords)

model.transform(noStopWords)

\*/

}

import org.apache.spark.ml.linalg.Vector

val mlf = featurize(training.toDS)

mlf.show

val mltest = featurize(test.toDS)

import org.apache.spark.mllib.feature.{HashingTF => MLLibHashingTF}

val mllibHashingTF = new MLLibHashingTF(100)

def mllibFeaturize(s: String) = {

mllibHashingTF.transform(s.sliding(2).toSeq)

}

import org.apache.spark.mllib.linalg.{Vector => MLLibVector}

val mllibf = training.map(

st => (st.id, st.label, st.sentiment, st.text, mllibFeaturize(st.text)))

.toDF("id", "label", "sentiment", "text", "features")

mllibf.show

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

val lr = new LogisticRegression()

.setMaxIter(10)

// .setRegParam(0.3)

// .setElasticNetParam(0.8)

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

val model = lr.fit(MLUtils.convertVectorColumnsToML(mlf))

println(s"Coefficients: ${model.coefficients}")

println(s"Intercept: ${model.intercept}")

val trainingSummary = model.summary

val objectiveHistory = trainingSummary.objectiveHistory

objectiveHistory.foreach(loss => println(loss))

# [SRC] 08.3. Evaluate your Classifier

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

val predictions = model.transform(MLUtils.convertVectorColumnsToML(mltest))

predictions.show

val predictions = model.transform(MLUtils.convertVectorColumnsToML(mltest))

val predictionsAndLabels = predictions.map { r =>

(r.getAs[Double]("prediction"), r.getAs[Double]("label"))

}

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

val evaluator = new BinaryClassificationEvaluator

val pl = predictionsAndLabels.toDF("rawPrediction", "label")

evaluator.setMetricName("areaUnderROC")

println(s"areaUnderROC: ${evaluator.evaluate(pl)}")

evaluator.setMetricName("areaUnderPR")

println(s"areaUnderPR: ${evaluator.evaluate(pl)}")

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

// Instantiate metrics object.

val metrics = new BinaryClassificationMetrics(predictionsAndLabels.rdd)

// ROC Curve.

val roc = metrics.roc.toDF

roc.createOrReplaceTempView("roc")

// Precision by threshold.

val precision = metrics.precisionByThreshold.collect

precision.foreach { case (t, p) =>

println(s"Threshold: $t, Precision: $p")

}

// Recall by threshold.

val recall = metrics.recallByThreshold.collect

recall.foreach { case (t, r) =>

println(s"Threshold: $t, Recall: $r")

}

// Precision-Recall Curve.

val PRC = metrics.pr

// F-measure.

val f1Score = metrics.fMeasureByThreshold.collect

f1Score.foreach { case (t, f) =>

println(s"Threshold: $t, F-score: $f, Beta = 1")

}

val beta = 0.5

val fScore = metrics.fMeasureByThreshold(beta)

f1Score.foreach {

case (t, f) => println(s"Threshold: $t, F-score: $f, Beta = 0.5")

}

%sql select \* from roc order by \_1

// AUPRC.

val auPRC = metrics.areaUnderPR

println("Area under precision-recall curve = " + auPRC)

// AUROC

val auROC = metrics.areaUnderROC

println("Area under ROC = " + auROC)

// Compute thresholds used in ROC and PR curves

val thresholds = precision.map(\_.\_1).foreach(println)

# [SRC] 08.4. Select your Model

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

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

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

import org.apache.spark.ml.feature.Tokenizer

import org.apache.spark.ml.feature.StopWordsRemover

import org.apache.spark.ml.feature.HashingTF

import org.apache.spark.ml.linalg.Vector

import org.apache.spark.sql.Row

val tokenizer = new Tokenizer()

.setInputCol("text")

.setOutputCol("words")

val remover = new StopWordsRemover()

.setInputCol(tokenizer.getOutputCol)

.setOutputCol("noStopWords")

val hashingTF = new HashingTF()

.setInputCol(remover.getOutputCol)

.setOutputCol("features")

val lr = new LogisticRegression()

.setMaxIter(10)

val pipeline = new Pipeline()

.setStages(Array(tokenizer, remover, hashingTF, lr))

import org.apache.spark.ml.tuning.ParamGridBuilder

val paramGrid = new ParamGridBuilder()

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

.addGrid(lr.threshold, Array(0.3, 0.5, 0.7))

.addGrid(hashingTF.numFeatures, Array(100, 1000))

.build()

val trainingDS = training.toDS

trainingDS.cache

trainingDS.persist

import org.apache.spark.ml.tuning.CrossValidator

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

val crossValidator = new CrossValidator()

.setEstimator(pipeline)

.setEvaluator(new BinaryClassificationEvaluator)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(4)

val model = crossValidator.fit(trainingDS)

val fitted = model.transform(test.toDS)

fitted.select("id", "label", "text", "probability", "prediction")

.take(10)

.foreach {

case Row(id: Int, label: Double, text: String, prob: Vector, prediction: Double) =>

println(s"($id, $text) --> prob=$prob, prediction=$prediction")

}

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

val evaluator = new BinaryClassificationEvaluator

val pl = fitted.select("prediction", "label").toDF("rawPrediction", "label")

evaluator.setMetricName("areaUnderROC")

println(s"areaUnderROC: ${evaluator.evaluate(pl)}")

evaluator.setMetricName("areaUnderPR")

println(s"areaUnderPR: ${evaluator.evaluate(pl)}")

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

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

import org.apache.spark.ml.tuning.ParamGridBuilder

val lr = new LogisticRegression()

.setMaxIter(10)

val paramGrid = new ParamGridBuilder()

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

.addGrid(lr.fitIntercept)

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

.build()

val trainValidationSplit = new TrainValidationSplit()

.setEstimator(lr)

.setEvaluator(new RegressionEvaluator)

.setEstimatorParamMaps(paramGrid)

// 80% of the data will be used for training and the remaining 20% for validation.

.setTrainRatio(0.1)

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

val evaluator = new BinaryClassificationEvaluator

val pl = fitted.select("prediction", "label").toDF("rawPrediction", "label")

evaluator.setMetricName("areaUnderROC")

println(s"areaUnderROC: ${evaluator.evaluate(pl)}")

evaluator.setMetricName("areaUnderPR")

println(s"areaUnderPR: ${evaluator.evaluate(pl)}")