UNIVERSIDAD DE JAÉN

Computación Distribuida para la gestión de datos a gran escala

Práctica 5. irisMissing.data

Sergio Perea De La Casa

Máster Universitario en Ingeniería Informática

Enunciado **3**

Partimos del dataset irisMissing.data, es el Iris de UCI (iris.data), al que se le han sustituido valores en algunos atributos por el carácter ?, para simular un dato perdido. Este dataset contiene los valores de las instancias que teníamos en la práctica 2 pero sin cabecera. Vamos a entrenar dos métodos de clasificación, logistic regression y random forests, y a evaluar sus resultados sobre este dataset, obteniendo el % de acierto en clasificación (accuracy). Para ello se va a realizar un preprocesamiento del dataset para adecuarlo a los métodos, se aplican los métodos y se evalúan. 3

Ejercicio 1 **3**

Crea un RDD, a partir del fichero. Comprueba que hay 150 instancias. 3

Ejercicio 2 **4**

Elimina las instancias que contienen valores perdidos y separa los atributos de cada instancia. Comprueba que quedan 144 instancias. 4

Ejercicio 3 **5**

Transformar la clase que es nominal a un valor numérico. En el caso de Iris, como la clase de salida tiene tres valores, éstos se transformarán a 0, 1 ó 2. El procesamiento debe ser genérico y si cambia el número de valores de la clase que se adapte a los valores que haya. 5

Ejercicio 4 **6**

Convierte todos los valores de los atributos de entrada a Double y transforma las instancias al tipo LabeledPoint. 6

Ejercicio 5 **7**

Aplica los métodos y obtén las medidas especificadas. 7

Ejercicio 6 **13**

Construye un RDD en el que los atributos de entrada (las características que no son la etiqueta) estén estandarizados. Vuelve a entrenar un modelo logistic regression y calcula la precisión de este nuevo modelo. 13

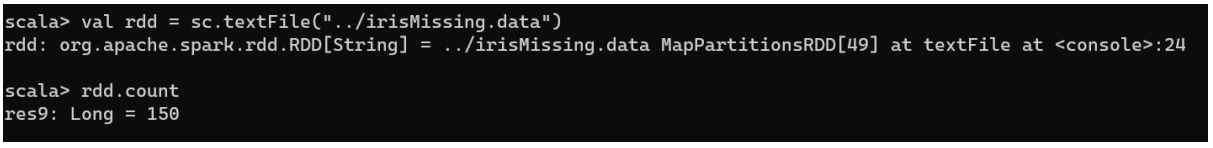
Enunciado

Partimos del dataset irisMissing.data, es el Iris de UCI (iris.data), al que se le han sustituido valores en algunos atributos por el carácter ?, para simular un dato perdido. Este dataset contiene los valores de las instancias que teníamos en la práctica 2 pero sin cabecera. Vamos a entrenar dos métodos de clasificación, logistic regression y random forests, y a evaluar sus resultados sobre este dataset, obteniendo el % de acierto en clasificación (accuracy). Para ello se va a realizar un preprocesamiento del dataset para adecuarlo a los métodos, se aplican los métodos y se evalúan.

Ejercicio 1

Crea un RDD, a partir del fichero. Comprueba que hay 150 instancias.

```
Unset
//Se inicializa el archivo irisMissing.data

scala> val rdd = sc.textFile("../irisMissing.data")
rdd: org.apache.spark.rdd.RDD[String] = ../irisMissing.data
MapPartitionsRDD[49] at textFile at <console>:24

scala> rdd.count
res9: Long = 150
```

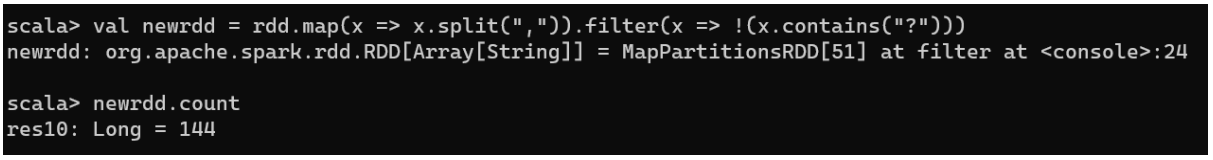


Ejercicio 2

Elimina las instancias que contienen valores perdidos y separa los atributos de cada instancia. Comprueba que quedan 144 instancias.

Unset

```
//Se crea un nuevo rdd que recibe un map del antiguo rdd, pero ahora con cada instancia separada por el split de "," y con la eliminación de las instancias que tienen el valor “?”


scala> val newrdd = rdd.map(x => x.split(",")).filter(x => !(x.contains("?")))
newrdd: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[51] at filter at <console>:24


scala> newrdd.count
res10: Long = 144
```



Ejercicio 3

Transformar la clase que es nominal a un valor numérico. En el caso de Iris, como la clase de salida tiene tres valores, éstos se transformarán a 0, 1 ó 2. El procesamiento debe ser genérico y si cambia el número de valores de la clase que se adapte a los valores que haya.

Unset

```
//Se hace un map de pares del rdd anterior, pero ahora con
if-else para verificar si el ultimo elemento del array es
igual a determinada clase y atribuirla el valor 0, 1 o 2,
dependiendo del valor de la clase. El array de pares tiene
como primero elemento el valor de la clase y lo segundo valor
un array con los valores de los atributos.

scala> val rdd = newrdd.map(x => (if(x(4)=="Iris-setosa") 0.0
else if(x(4)=="Iris-versicolor") 1.0 else
2.0,Array(x(0),x(1),x(2),x(3))))
rdd: org.apache.spark.rdd.RDD[(Double, Array[String])] =
MapPartitionsRDD[52] at map at <console>:24
```

```
scala> val rdd = newrdd.map(x => (if(x(4)=="Iris-setosa") 0.0 else if(x(4)=
="Iris-versicolor") 1.0 else 2.0,Array(x(0),x(1),x(2),x(3))))
rdd: org.apache.spark.rdd.RDD[(Double, Array[String])] = MapPartitionsRDD[5
2] at map at <console>:24

scala> rdd.collect()
res14: Array[(Double, Array[String])] = Array((0.0,Array(5.1, 3.5, 1.4, 0.2
)), (0.0,Array(4.9, 3.0, 1.4, 0.2)), (0.0,Array(4.7, 3.2, 1.3, 0.2)), (0.0,
Array(4.6, 3.1, 1.5, 0.2)), (0.0,Array(5.0, 3.6, 1.4, 0.2)), (0.0,Array(5.4
, 3.9, 1.7, 0.4)), (0.0,Array(4.6, 3.4, 1.4, 0.3)), (0.0,Array(5.0, 3.4, 1.
5, 0.2)), (0.0,Array(4.4, 2.9, 1.4, 0.2)), (0.0,Array(4.9, 3.1, 1.5, 0.1)),
 (0.0,Array(5.4, 3.7, 1.5, 0.2)), (0.0,Array(4.8, 3.4, 1.6, 0.2)), (0.0,Arr
ay(4.8, 3.0, 1.4, 0.1)), (0.0,Array(4.3, 3.0, 1.1, 0.1)), (0.0,Array(5.8, 4
.0, 1.2, 0.2)), (0.0,Array(5.7, 4.4, 1.5, 0.4)), (0.0,Array(5.1, 3.5, 1.4,
0.3)), (0.0,Array(5.7, 3.8, 1.7, 0.3)), (0.0,Array(5.1, 3.8, 1.5, 0.3)), (0
.0,Array(5.4, 3.4, 1.7, 0.2)), (0.0,Array(4.6, 3.6, 1.0, 0.2)), (0.0,Array(
5.1, 3.3, 1.7, 0.5)), (0.0,Array(4.8, 3.4, 1.9,...
```

Ejercicio 4

Convierte todos los valores de los atributos de entrada a Double y transforma las instancias al tipo LabeledPoint.

Unset

```
//Ahora se convierte el valor del rdd de pares anterior en
double, haciendo un mapValues del rdd anterior para que
convierta solamente el valor

scala> val rddDouble = rdd.mapValues(x => x.map(_.toDouble))
rddDouble: org.apache.spark.rdd.RDD[(Double, Array[Double])] =
MapPartitionsRDD[53] at mapValues at <console>:24


//Ahora se convierte cada instancia del rdd anterior (double),
a tipo LabeledPoint, convirtiendo su array en un vector denso.

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

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

scala> val result = rddDouble.map(x => LabeledPoint(x._1,
Vectors.dense(x._2)))
result:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[54] at map at <console>:30
```

```
scala> val rddDouble = rdd.mapValues(x => x.map(_.toDouble))
rddDouble: org.apache.spark.rdd.RDD[(Double, Array[Double])] = MapPartitionsRDD[53] at mapValues at <console>:24
```

```
scala> val result = rddDouble.map(x => LabeledPoint(x._1, Vectors.dense(x._2)))
result: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = MapPartitionsRDD[54] at map at <console>:30
```

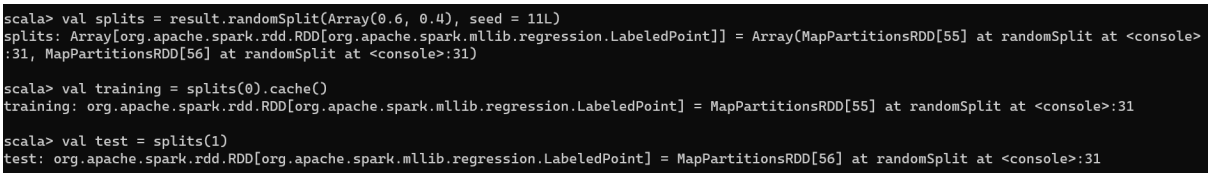
Ejercicio 5

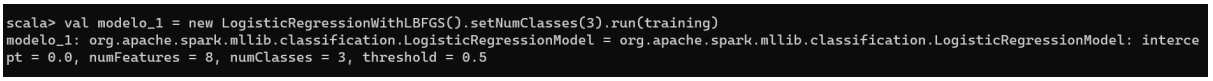
Aplica los métodos y obtén las medidas especificadas.

LOGISTIC REGRESSION

```
Unset
//Se hace los imports de los métodos

scala> import
org.apache.spark.mllib.classification.{LogisticRegressionModel
, LogisticRegressionWithLBFGS}
import
org.apache.spark.mllib.classification.{LogisticRegressionModel
, LogisticRegressionWithLBFGS}



//Se hacen los dataset de entrenamiento y test

scala> val splits = result.randomSplit(Array(0.6, 0.4), seed =
11L)
splits:
Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regressi
on.LabeledPoint]] = Array(MapPartitionsRDD[55] at randomSplit
at <console>:31, MapPartitionsRDD[56] at randomSplit at
<console>:31)



//La parte training recibe 60% de los datos
scala> val training = splits(0).cache()
training:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[55] at randomSplit at
<console>:31


//La parte test recibe 40% de los datos

scala> val test = splits(1)
test:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[56] at randomSplit at
```

```
<console>:31



//Ejecuta el método de LinealRegression con los datos de
training y con el número de clases 3

scala> val modelo_1 = new
LogisticRegressionWithLBFGS().setNumClasses(3).run(training)
modelo_1:
org.apache.spark.mllib.classification.LogisticRegressionModel
=
org.apache.spark.mllib.classification.LogisticRegressionModel:
intercept = 0.0, numFeatures = 8, numClasses = 3, threshold =
0.5
```






```
Unset
//Crea una variable prediction que recibe el método ya
entrenado pero ahora ejecutando con los datos de test

scala> val prediction = test.map {
     | case LabeledPoint(label, features) =>
     | val prediction = modelLR.predict(features)
     | (prediction, label)}
prediction: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[318] at map at <console>:32

scala> import
org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.evaluation.MulticlassMetrics

scala> val metrics = new MulticlassMetrics(prediction)
```

```
metrics: org.apache.spark.mllib.evaluation.MulticlassMetrics =
org.apache.spark.mllib.evaluation.MulticlassMetrics@73314c32

scala> val accuracy = metrics.accuracy
accuracy: Double = 1.0
```

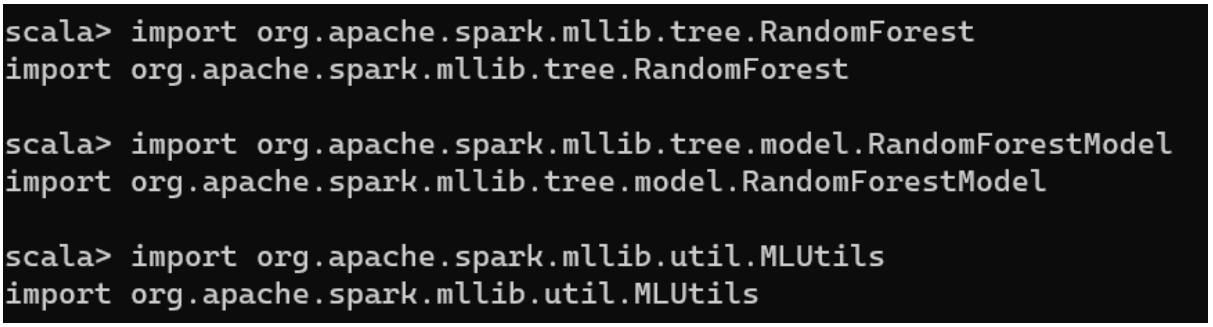
```
scala> val prediction = test.map {
     | case LabeledPoint(label, features) =>
     | val prediction = modelLR.predict(features)
     | (prediction, label)}
prediction: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[318] at map at <console>:32
```

```
scala> import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.evaluation.MulticlassMetrics

scala> val metrics = new MulticlassMetrics(prediction)
metrics: org.apache.spark.mllib.evaluation.MulticlassMetrics = org.apache.spark.mllib.evaluation.MulticlassMetrics@73314c32

scala> val accuracy = metrics.accuracy
accuracy: Double = 1.0
```

RANDOM FOREST

Unset

```
scala> import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.RandomForest

scala> import
org.apache.spark.mllib.tree.model.RandomForestModel
import org.apache.spark.mllib.tree.model.RandomForestModel

scala> import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.util.MLUtils
```



Unset

```
scala> val splitRF = result.randomSplit(Array(0.6, 0.4))
splitRF:
Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regressi
on.LabeledPoint]] = Array(MapPartitionsRDD[321] at randomSplit
at <console>:35, MapPartitionsRDD[322] at randomSplit at
<console>:35)

scala> val (trainingData, testData) = (splits(0), splits(1))
trainingData:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[55] at randomSplit at
<console>:31
testData:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
```

```
eledPoint] = MapPartitionsRDD[56] at randomSplit at
<console>:31

//Se definen las variables y valores que son necesarios para
que el método RF se ejecute:

scala> val num_clases = 3
num_clases: Int = 3

scala> val categoriaFeaturesInfo = Map[Int, Int]()
categoriaFeaturesInfo: scala.collection.immutable.Map[Int,Int]
= Map()

scala> val num_arboles = 5
num_arboles: Int = 5

scala> val featureSubsetStrategy = "auto"
featureSubsetStrategy: String = auto

scala> val maxDepth = 5
maxDepth: Int = 5

scala> val maxBins = 32
maxBins: Int = 32

scala> val impurity = "gini"
impurity: String = gini


scala> val modelo_2 =
RandomForest.trainClassifier(trainingData, num_clases,
categoriaFeaturesInfo, num_arboles, featureSubsetStrategy,
impurity
, maxDepth, maxBins)
modelo_2: org.apache.spark.mllib.tree.model.RandomForestModel
=
TreeEnsembleModel classifier with 5 trees
```

```
scala> val modelo_2 = RandomForest.trainClassifier(trainingData, num_clases, categoriaFeaturesInfo, num_arboles, featureSubsetStrategy, impurity
, maxDepth, maxBins)
modelo_2: org.apache.spark.mllib.tree.model.RandomForestModel =
TreeEnsembleModel classifier with 5 trees
```

Unset

```
// Ejecuta el modelo de entrenamiento con los datos del test
para mirar la precisión y la tasa de error del modelo Random
Forest

scala> val etiquetasYPredicciones = testData.map{
     | point =>
     | val prediction = modelo_2.predict(point.features)
     | (point.label, prediction)}
etiquetasYPredicciones: org.apache.spark.rdd.RDD[(Double,
Double)] = MapPartitionsRDD[347] at map at <console>:37


//La tasa de error del modelo:

scala> val tasa_error = etiquetasYPredicciones.filter(r =>
r._1 != r._2).count.toDouble / testData.count()
tasa_error: Double = 0.0

scala> val metricasRF = new
MulticlassMetrics(etiquetasYPredicciones)
metricasRF:
org.apache.spark.mllib.evaluation.MulticlassMetrics =
org.apache.spark.mllib.evaluation.MulticlassMetrics@2ef85218


scala> val precision_metricasRF = metricasRF.accuracy
precision_metricasRF: Double = 1.0
```

```
scala> val etiquetasYPredicciones = testData.map{
     | point =>
     | val prediction = modelo_2.predict(point.features)
     | (point.label, prediction)}
etiquetasYPredicciones: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[347] at map at <console>:37
```

```
scala> val tasa_error = etiquetasYPredicciones.filter(r => r._1 != r._2).count.toDouble / testData.count()
tasa_error: Double = 0.0
```

```
scala> val metricasRF = new MulticlassMetrics(etiquetasYPredicciones)
metricasRF: org.apache.spark.mllib.evaluation.MulticlassMetrics = org.apache.spark.mllib.evaluation.MulticlassMetrics@2ef85218

scala> metric
metricasRF   metrics

scala> val precision_metricasRF = metricasRF.accuracy
precision_metricasRF: Double = 1.0
```

Ejercicio 6

Construye un RDD en el que los atributos de entrada (las características que no son la etiqueta) estén estandarizados. Vuelve a entrenar un modelo logistic regression y calcula la precisión de este nuevo modelo.

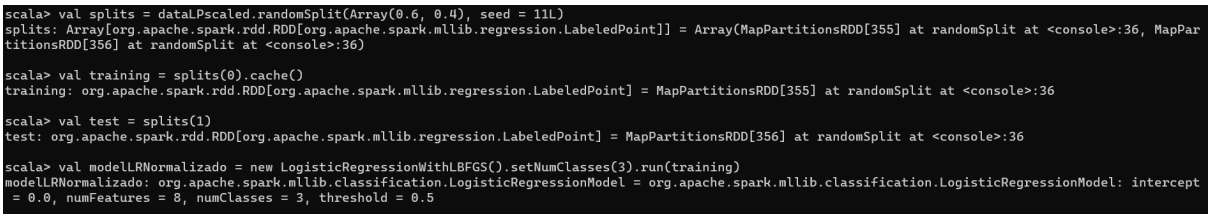
Unset

```
//Se usa el escalado estandar:

scala> import org.apache.spark.mllib.feature.{StandardScaler,
StandardScalerModel}
import org.apache.spark.mllib.feature.{StandardScaler,
StandardScalerModel}


scala> val scaler = new StandardScaler(withMean = true,
withStd = true).fit(result.map(lp => lp.features))
scaler: org.apache.spark.mllib.feature.StandardScalerModel =
org.apache.spark.mllib.feature.StandardScalerModel@6fbc9f12


scala> val dataLPscaled = result.map(lp =>
LabeledPoint(lp.label, scaler.transform(lp.features)))
dataLPscaled:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[354] at map at <console>:37
```

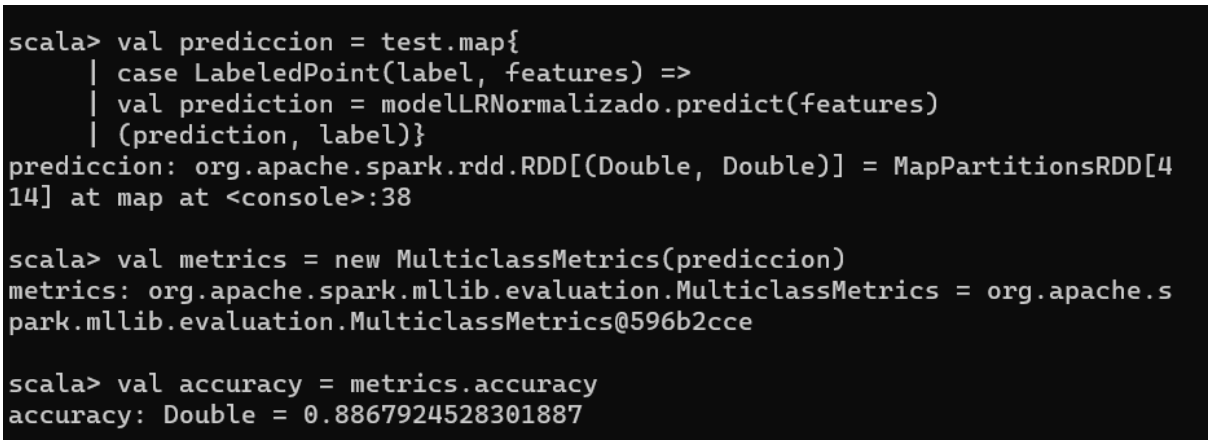
```
scala> val scaler = new StandardScaler(withMean = true, withStd = true).fit(result.map(lp => lp.features))
scaler: org.apache.spark.mllib.feature.StandardScalerModel = org.apache.spark.mllib.feature.StandardScalerModel@6fbc9f12

scala> val dataLPscaled = result.map(lp => LabeledPoint(lp.label, scaler.transform(lp.features)))
dataLPscaled: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = MapPartitionsRDD[354] at map at <console>:37
```

Unset

```
//Una vez normalizado, se aplica el mismo proceso de RL pero
con los datos normalizados y vemos su resultado:

scala> val splits = dataLPscaled.randomSplit(Array(0.6, 0.4),
seed = 11L)
splits:
Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regressi
on.LabeledPoint]] = Array(MapPartitionsRDD[355] at randomSplit
at <console>:36, MapPartitionsRDD[356] at randomSplit at
<console>:36)

scala> val training = splits(0).cache()
training:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[355] at randomSplit at
<console>:36

scala> val test = splits(1)
test:
org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.Lab
eledPoint] = MapPartitionsRDD[356] at randomSplit at
<console>:36

scala> val modelLRNormalizado = new
LogisticRegressionWithLBFGS().setNumClasses(3).run(training)
modelLRNormalizado:
org.apache.spark.mllib.classification.LogisticRegressionModel
=
org.apache.spark.mllib.classification.LogisticRegressionModel:
intercept = 0.0, numFeatures = 8, numClasses = 3, threshold =
0.5
```



```
Unset
scala> val prediccion = test.map{
     | case LabeledPoint(label, features) =>
     | val prediction = modelLRNormalizado.predict(features)
     | (prediction, label)}
prediccion: org.apache.spark.rdd.RDD[(Double, Double)] =
MapPartitionsRDD[414] at map at <console>:38


scala> val metrics = new MulticlassMetrics(prediccion)
metrics: org.apache.spark.mllib.evaluation.MulticlassMetrics =
org.apache.spark.mllib.evaluation.MulticlassMetrics@596b2cce


scala> val accuracy = metrics.accuracy
accuracy: Double = 0.8867924528301887
```



Como podemos observar, la precisión de unos datos normalizados para este modelo es mucho más baja que en el anterior. Por lo que forzar a que los datos se mantengan entre un rango concreto normalizado, perjudica en su precisión en este caso concreto.