# Técnicas escalables de análisis de datos en entornos Big Data: Clasificadores

Métricas de evaluación de clasificadores binarios

# Contenido

1. ML `evaluators`.
2. Métricas para la evaluación de modelos en ML.
3. ML `BinaryClassificationEvaluator`: parámetros.
4. Evaluación filtro anti Spam con ML.
5. Métricas para la evaluación de modelos en Mllib.
6. MLbib `BinaryClassificationMetrics`: argumentos y métodos.
7. Evaluación filtro anti Spam con Mllib.
8. Construcción de curvas ROC para clasificación binaria con MLlib .
9. Referencias.

# 1. *ML* Evaluators

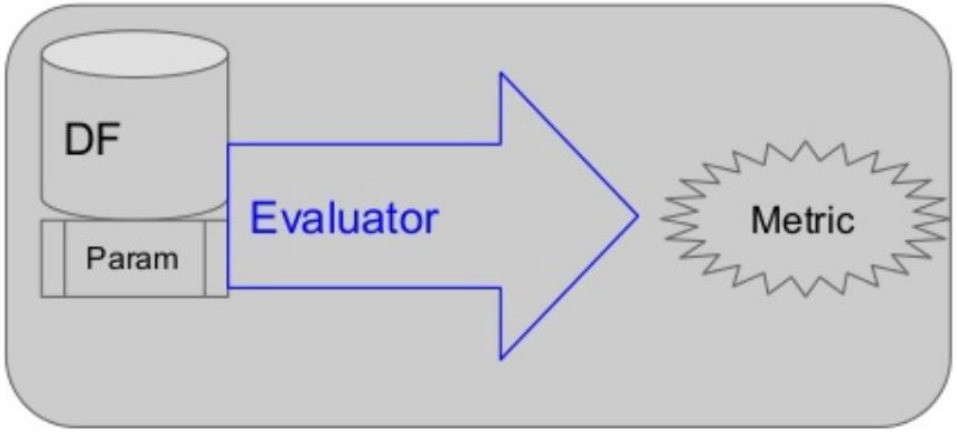
- ***Estimators***: generan *Transformers* ajustando modelos a conjuntos de datos.
- Cualquier tipo de modelo que se pueda/necesite ajustar/entrenar con datos.
- Con el método `fit()`.

- ***Transformers***: generan nuevos *DataFrames* a partir de *DataFrames* existentes procesándolos con modelos .
- Un *transformer* generaliza el concepto de Modelo que procesa datos
  - Modelos ya ajustados/entrenados con datos.
  - Modelos que no necesitan ajustarse (por ejemplo, `HashingTF`).
- Obteniendo las predicciones del modelo.
- Con el método `transform()`.

- ***Evaluators***: evalúan el comportamiento de los métodos de aprendizaje utilizando una métrica.
- En el caso supervisado requieren la predicción y la clase.
- Con el método `evaluate()`.

# Evaluator

- Un `Evaluator` toma un `DataFrame` con datos etiquetados y predicciones y un modelo entrenado (`transformer`).
- Evalúa una métrica para dicho modelo.
- Cuyo valor depende indirectamente de los parámetros con los que se creó el modelo
  - Porque determinan qué modelo se obtiene.
  - Aunque no son parte de la entrada al `Evaluador` (a pesar de la figura).




Ram Kuppsuwamy. Machine learning pipeline with spark ml

# 2. Métricas para la evaluación de modelos en ML

- *ML* proporciona las clases
  - `BinaryClassificationEvaluator`: por defecto AUC (*Area Under Curve*).
  - `MulticlassClassificationEvaluator`.
  - `RegressionEvaluator`.
- En paquete `org.apache.spark.ml.evaluation`.

- Se crea una instancia de métrica

```
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
val bceval = new BinaryClassificationEvaluator()
```

- Se aplica con método `evaluate`: requiere *DataFrame* (o *Dataset*, no un *RDD*) con columnas de predicciones y etiquetas

```
bceval.evaluate(PredictionsandLabelsDF)
```

# 3. ML `BinaryClassificationEvaluator`: parámetros

- `labelCol`: columna con las clases reales
  - Tipo `Double`, por defecto `"label"`.
- `metricName`: métrica a calcular, `areaUnderROC` o `areaUnderPR`
  - Tipo `String`, por defecto `"areaUnderROC"`.
- `numBins`: bins para calcular las curvas (y auc)
  - Tipo `Int`, por defecto `1000`.
- `rawPredictionCol`: scores predichos
  - Tipo `Vector`, por defecto `"rawPredictions"`.

# ML `BinaryClassificationEvaluator`: parámetros para curvas ROC

- `numBins`: `bins` para la construcción de curvas (y `auc`)
  - 0: tantos puntos como instancias con diferente `score`.
  - > 0: submuestrea a `numBins` puntos +2.
  - Por defecto: 1000.

- `rawPredictionCol`: scores predichos
  - "confianza" del clasificador en las predicciones, en intervalo [0.0, 1.0].
  - NO necesariamente probabilidades bien calibradas.
  - Tipo `Vector`.
    - Primer elemento: confianza en la predicción para la clase 0.0.
    - Segundo elemento: confianza en la predicción para la clase 1.0.
    - Ambos con valores en intervalo [0.0, 1.0].

# Pero en Spark 3.5.6

- No todos los clasificadores generan una columna `rawPredictionCol` con scores en intervalo [0.0, 1.0].

- En este caso, las métricas con los valores por defecto pueden calcular valores **erróneos**.

- La solución consiste en cambiar el valor por defecto del parámetro `rawPredictionCol` en la métrica
  - Usando directamente la predicción: `"prediction"` (valores 0/1).
  - Usando la estimación de probabilidad: `"probability"` (que son scores válidos).
  - Lo veremos con detalle sobre el ejemplo del filtro anti-spam.

# 4. Evaluación filtro anti Spam con ML

- Recuperamos el ejemplo del filtro antispam con NB (tasa de error 0.059).

```
load "naiveBayesMLSpam.scala"


// Ya hemos predicho la clase de los ejemplos de prueba
// en el DF predictionsAndLabels
println("Data Frame predictionsAndLabels")
predictionsAndLabels.show(5)
```

```
predictionsAndLabels.show(5)
+-----+--------------------+--------------------+--------------------+----------+
|label|            features|       rawPrediction|         probability|prediction|
+-----+--------------------+--------------------+--------------------+----------+
|  1.0|(100,[0,7,13,16,1...|[-121.72145524720...|[0.04995643550982...|       1.0|
|  1.0|(100,[15,16,40,87...|[-25.138905263514...|[0.84626145067355...|       0.0|
|  1.0|(100,[2,5,17,20,2...|[-111.69097046208...|[0.97183332605587...|       0.0|
|  1.0|(100,[13,17,21,25...|[-136.06260695511...|[0.00250757676764...|       1.0|
|  1.0|(100,[1,7,12,15,2...|[-128.44756748959...|[0.32016658432462...|       1.0|
+-----+--------------------+--------------------+--------------------+----------+
only showing top 5 rows
```

# Evaluamos con parámetros por defecto

```
/* Creamos una instancia de BinaryClassificationEvaluator de ML */
import
org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
val ML_binarymetrics = new BinaryClassificationEvaluator()


/* Calculamos Área bajo la curva ROC, auROC      */
/* con los parámetros por defecto de ML          */
val ML_auROC = ML_binarymetrics.evaluate(predictionsAndLabels)
println("AUC de la curva ROC para la clase SPAM")
println(f"con ML, métrica binaria, parámetros por defecto'):
$ML_auROC%1.4f%n")
println("Este resultado es inconsistente con un clasificador
binario mejor que aleatorio.")
```

```
AUC de la curva ROC para la clase SPAM
con ML, métrica binaria, parámetros por defecto: 0,1940

Este resultado es inconsistente con un clasificador binario mejor
que aleatorio.
```

# Naive bayes y `BinaryClassificationEvaluator`

- Parámetros por defecto

`metricName: areaUnderROC.`

`rawPredictionCol: rawPrediction.`

- **Calcula mal `areaUnderROC`.**

- En el ejemplo anterior, AUC=0.194.
- Corresponde a un clasificador mucho peor que aleatorio, incompatible con una tasa de error de 0.059.

- Posiblemente, porque el contenido de `rawPrediction` que genera el clasificador NB de ML no es el que espera el evaluador
  - No son scores (en rango [0, 1]).

# Examinamos Fila de `rawPrediction`

```
// Examinamos primera fila de rawPrediction
val primer_rP =
predictionsAndLabels.select("rawPrediction").first

println(f"Primera fila de rawPrediction: $primer_rP%n")
```

```
Primera fila de rawPrediction: [[-121.72145524720202,-
118.77609874213178]]
```

- No son scores válidos.

# Alternativa: setRawPredictionCol("prediction")

- **Utilizando** las etiquetas predichas como *rawPredictionCol*

```
/* Calculamos Área bajo la curva ROC, auROC      */
/* fijando setRawPredictionCol("prediction")   */
val ML_binarymetrics = new
BinaryClassificationEvaluator().setRawPredictionCol("prediction")

val ML_auROC = ML_binarymetrics.evaluate(predictionsAndLabels)
println("AUC de la curva ROC para la clase SPAM")
println(f"con ML, métrica binaria,
setRawPredictionCol('prediction'): $ML_auROC%1.4f%n")
```

```
AUC de la curva ROC para la clase SPAM
con ML, métrica binaria, setRawPredictionCol('prediction'): 0,8337
```

# AUC con "prediction" y "label"

con ML, métrica binaria, setRawPredictionCol('prediction'):
0,8337

- Este resultado es correcto.
- Pero corresponde a una curva elaborada con pocos puntos (probablemente 4).
  - Con independencia de numBins.
  - En este sentido, se comporta de forma similar a la métrica de MLlib usando "prediction" y "label".
- La métrica binaria de ML no permite construir la curva ROC.

# Mejor: setRawPredictionCol("probability")

- **Utilizando** las estimaciones de probabilidad como rawPredictionCol
  - Son scores válidos.

```
//
//  ahora con setRawPredictionCol("probability")
//
val ML_binarymetrics = new
BinaryClassificationEvaluator().setRawPredictionCol("probability")

val ML_auROC = ML_binarymetrics.evaluate(predictionsAndLabels)
println("AUC de la curva ROC para la clase SPAM")
println(f"con ML, métrica binaria,
setRawPredictionCol('probability'): $ML_auROC%1.4f%n")
```

```
AUC de la curva ROC para la clase SPAM
con ML, métrica binaria, setRawPredictionCol('probability'):
0,9182
```

# AUC con "probability" y "label"

AUC de la curva ROC para la clase SPAM
con ML, métrica binaria, setRawPredictionCol('probability'):
0,9182

- Este resultado es correcto y más preciso, por crear la curva con más puntos.
- Porque utiliza numBins=1000.

- Y la métrica proporciona un mejor valor: 0.9182 frente a 0.837.

# 5. Métricas para la evaluación de modelos en *MLlib*

- *MLlib* proporciona clases para la evaluación de modelos.
- Para clasificadores:

  `BinaryClassificationMetrics`: problemas de clasificación binaria.

  `MulticlassMetrics`: la clase no es binaria, pero cada instancia pertenece a una sola clase.

  `MultilabelMetrics`: la clase puede tomar varios valores.

- En los paquetes

```
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.evaluation.MultilabelMetrics
```

- Nos centramos en `BinaryClassificationMetrics`.

# 6. MLbib `BinaryClassificationMetrics`: argumentos

- Creamos una nueva instancia **para cada** conjunto de predicciones a evaluar

```
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
val bcmetric = new
BinaryClassificationMetrics(predictionsAndLabelsRDD, numBins)
```

- `predictionsAndLabelsRDD:` RDDs[(double, double)] con
  - (clase predicha, clase real).
  - (`scores` predichos, clase real ).
  - Dependiendo de si el modelo proporciona *scores* o solo la clase.
- `numBins: Int:` puntos a utilizar para la construcción de curvas
  - De interés si `scores.`
  - 0: tantos como instancias con diferente `score.`
  - > 0: submuestrea a `numBins` puntos.
  - Por defecto: 0.

# MLlib `BinaryClassificationMetrics`: métodos

- Parametrizamos y obtenemos resultados con los métodos:

- `areaUnderPR(): Double`
- `areaUnderROC(): Double`
- `precisionByThreshold(): RDD[(Double, Double)]`
- `recallByThreshold(): RDD[(Double, Double)]`
- `roc(): RDD[(Double, Double)]`
- `pr(): RDD[(Double, Double)]`

- Por ejemplo, el área bajo la curva

```
val auc_ROC= bcmetric.areaUnderROC
```

- O los puntos de la curva ROC

```
val curva_ROC= bcmetric.roc
```

# 7. Evaluación filtro anti Spam con *MLlib*

- Recuperamos el ejemplo del filtro anti spam con NB en ML, con las predicciones de modelo en el Data Frame `predictionsAndLabels`

```
:load "naiveBayesMLSpam.scala" // el nombre de vuestro script
```

- Tenemos que trabajar con RDDs: es Mllib.

- Creamos un RDD con "predictions" and "labels“.

# Creamos predictionsAndLabelsRDD

```
// Ahora convertimos predictionsAndLabels a RDD
// para trabajar con MLlib
// con filas (prediction, label)  (el orden es importante)
val predictionsAndLabelsRDD =
predictionsAndLabels.rdd.map(row => (row.getDouble(4),
row.getDouble(0)))

println("predictionsAndLabelsRDD:")
predictionsAndLabelsRDD.take(5).foreach(x => println(x))
```

```
predictionsAndLabelsRDD:
(1.0,1.0)
(0.0,1.0)
(0.0,1.0)
(1.0,1.0)
(1.0,1.0)
```

# MLlib, AUC ROC con parámetros por efecto

```
/* Creamos una instancia de BinaryClassificationMetrics de Mllib */
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

val MLlib_binarymetrics = new
BinaryClassificationMetrics(predictionsAndLabelsRDD)



/* Calculamos Área bajo la curva ROC, auROC      */
//  AUC=0.83
val MLlib_auROC = MLlib_binarymetrics.areaUnderROC
println(f"%nAUC de la curva ROC para la clase SPAM")
println(f"con MLlib, métrica binaria, parámetros por defecto:
$MLlib_auROC%1.4f%n")
```

```
AUC de la curva ROC para la clase SPAM
con MLlib, métrica binaria, parámetros por defecto: 0,8337
```

# MLlib, curva ROC con parámetros por efecto

```
/* Y los puntos de la curva ROC */
// Curva
val MLlib_curvaROC =MLlib_binarymetrics.roc

//
//  Se puede comprobar que usando prediciones y etiquetas,
//  MLlib construye la curva ROC con solo 4 puntos
//
println("Puntos para construir curva ROC con MLlib
predictionsAndLabelsRDD:")
MLlib_curvaROC.take(10).foreach(x => println(x))

Puntos para construir curva ROC con MLlib predictionsAndLabelsRDD:
(0.0,0.0)
(0.020371479928100 66,0.687747035573122 6)
(1.0,1.0)
(1.0,1.0)
```

# Puntos de la curva

- Si el clasificador proporciona la clase (no los *scores*) la curva se construye con solo 4 puntos:
  - (0,0)
  - Punto correspondiente al umbral clase 0
  - Punto correspondiente al umbral clase 1
  - (1, 1)

```
(0.0,0.0)
(0.02037147992810066,0.6877470355731226)
(1.0,1.0)
(1.0,1.0)
```

En realidad, solo un punto significativo.

# 8. Construcción de curvas ROC para clasificación binaria con MLlib

- Necesitamos que el clasificador proporcione los *scores:*

  `Array[(Double, Double)]` (score, clase).

- La probabilidad es un score válido.

- En nuestro ejemplo, disponemos de la columna `"probability"`.

- Podemos generar un RDD con filas (probabilidad clase, clase).

# Obtención RDD probabilitiesAndLabelsRDD (I)

```
/*  Construcción curva ROC clasificación binaria           */
/*  a partir de las probabilidades de la predicción        */
/*  generando RDD con probabilidad de la clase 1 y labels   */

//
//   ATENCIÓN, necesitamos la clase Vector de ML
//

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


//Creamos RDD vectorLabelsAndProbabilitiesRDD
// Y examinamos primer elemento
val vectorLabelsAndProbabilitiesRDD =
predictionsAndLabels.select("label", "probability").rdd.first

vectorLabelsAndProbabilitiesRDD: org.apache.spark.sql.Row =
[1.0,[0.0499564355098263,0.9500435644901774]]
```

# Obtención RDD probabilitiesAndLabelsRDD (II)

```
vectorLabelsAndProbabilitiesRDD: org.apache.spark.sql.Row =
[1.0,[0.04995643550982263,0.9500435644901774]]

// Pero la métrica espera tuplas con (probability, label)
// o (scores, lables) para clasificadores de MLlib que puedan
generar scores
//
// La probabilidad de la clase 1 es el segundo elemento del
vector (índice 1 de r._1(1))
//
val probabilitiesAndLabelsRDD =
predictionsAndLabels.select("label", "probability").rdd.map{row
=> (row.getAs[Vector](1).toArray, row.getDouble(0))}.map{r => (
r._1(1), r._2)}

Paso Intermedio:
predictionsAndLabels.select("label", "probability").rdd.map{row =>
(row.getAs[Vector](1).toArray, row.getDouble(0))}.first
res70: (Array[Double], Double) = (Array(0.04995643550982263,
0.9500435644901774),1.0)
```

# Examinamos RDD probabilitiesAndLabelsRDD (II)

```
println(f"%nRDD probabilitiesAndLabelsRDD:")
probabilitiesAndLabelsRDD.take(5).foreach(x => println(x))
```

```
RDD probabilitiesAndLabelsRDD:
(0.9500435644901774,1.0)
(0.15373854932644798,1.0)
(0.02816667394412429,1.0)
(0.9974924232323557,1.0)
(0.6798334156753798,1.0)
```

# Creamos nueva métrica con 15 bins y calculamos auc

```
// Creamos nueva métrica
// con 15 bins para al curva ROC (17 puntos)
val MLlib_binarymetrics = new
BinaryClassificationMetrics(probabilitiesAndLabelsRDD,15)

/* Calculamos Área bajo la curva ROC, auROC      */
//  AUC=0.9143
val MLlib_auROC = MLlib_binarymetrics.areaUnderROC
println(f"%nAUC de la curva ROC para la clase SPAM")
println(f"con MLlib, métrica binaria,
probabilitiesAndLabelsRDD, 15 bins: $MLlib_auROC%1.4f%n")
```

```
AUC de la curva ROC para la clase SPAM
con MLlib, métrica binaria, probabilitiesAndLabelsRDD, 15
bins: 0,9153
```

- Comparar con ML_auROC: 0,9182
  - Con 1000 bins

# Obtenemos los puntos de la curva ROC, 15 bins

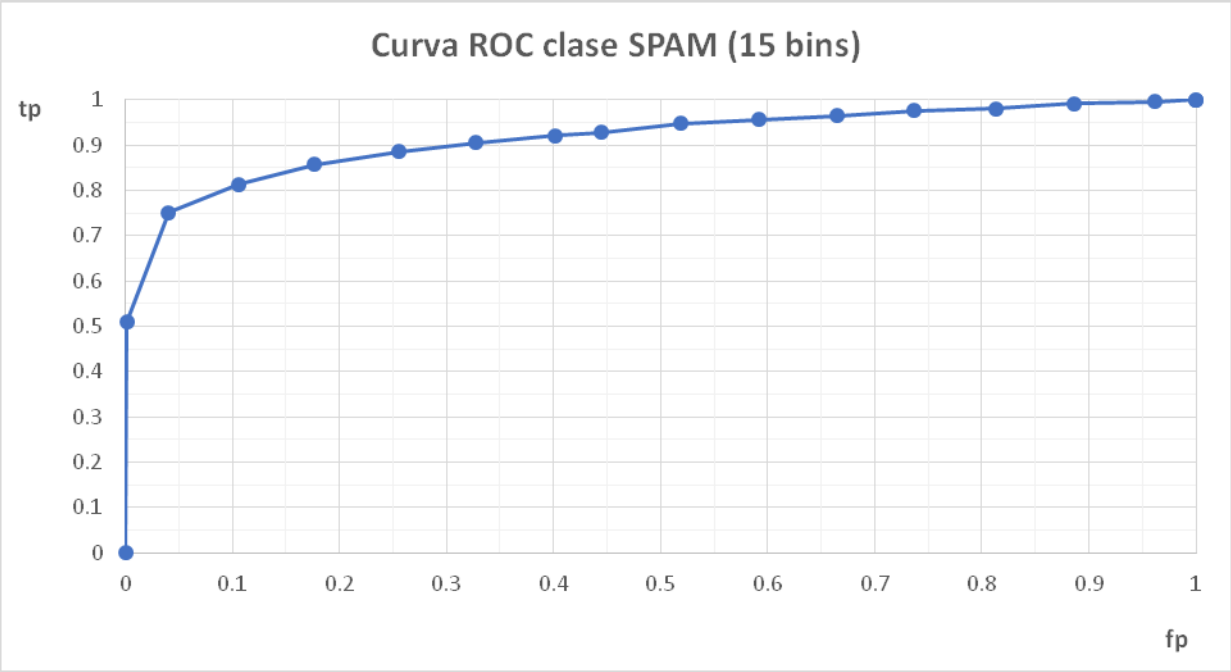
```
/* Y los puntos de la curva ROC */
// Curva
val MLlib_curvaROC =MLlib_binarymetrics.roc

println("Puntos para construir curva ROC con MLlib,
probabilitiesAndLabelsRDD, 15 bins:")
MLlib_curvaROC.take(17).foreach(x => println(x))
```

# Puntos de la curva ROC, 15 bins

```
Puntos para construir curva ROC con MLlib,
probabilitiesAndLabelsRDD, 15 bins:
(0.0,0.0)
(0.001198322348711803 5,0.509881422924901)
(0.040143798681845415,0.750988142292490 1)
(0.1060515278609946,0.814229249011857 7)
(0.1767525464349910 3,0.857707509881423)
(0.2552426602756141 5,0.885375494071146 2)
(0.3277411623726782 5,0.905138339920948 6)
(0.4020371479928100 4,0.920948616600790 5)
(0.4176153385260635 3,0.924901185770751)
(0.4925104853205512 5,0.936758893280632 4)
(0.5656081485919713,0.952569169960474 4)
(0.6393049730377471,0.960474308300395 3)
(0.7118034751348112,0.972332015810276 7)
(0.786698621929299,0.980237154150197 6)
(0.8615937687237867,0.984189723320158 1)
(0.9352905931695626,0.992094861660079 1)
(1.0,1.0)
```

# Usamos otra herramienta para obtener la gráfica




- Buena curva ROC, con auc 0.9153 (MLlib)
- Podemos detectar el 81% del SPAM con solo un 11% de falsos positivos
- O el 51%  casi sin falsos positivos, 0.1%
- O el 76% con 4% falsos positivos

# 9. Referencias (I)


- Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
- Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011
- Tiago A. Almeida and José María Gómez Hidalgo. SMS Spam Collection v.1, 2011. Retrieved from https://www.researchgate.net/publication/258050002_SMS_Spam_Collection_v1 . Último acceso: octubre 2025.
- Venu Kanaparthy. Spam classification with naive bayes using Spark Mllib. Retrieved from https://venukanaparthy.wordpress.com/2015/07/04/spam-classification-with-naive-bayes-using-spark-mllib/ . Último acceso: octubre 2025.
- Nick Pentreath. Machine Learning with Spark. Packt Publishing. 2015. ISBN: 9781783288519. http://www.packtpub.com/

# Referencias (II)


- BinaryClassificationEvaluator. https://spark.apache.org/docs/3.5.6/api/scala/org/apache/spark/ml/evaluation/BinaryClassificationEvaluator.html. Último acceso: octubre 2025.
- Evaluation Metrics - RDD-based API: Binary classification. https://spark.apache.org/docs/3.5.6/mllib-evaluation-metrics.html#binary-classification. Último acceso: octubre 2025.
- BinaryClassificationMetrics. https://spark.apache.org/docs/3.5.6/api/scala/org/apache/spark/mllib/evaluation/BinaryClassificationMetrics.html. Último acceso: : octubre 2025.
- Naive Bayes. https://spark.apache.org/docs/3.5.6/ml-classification-regression.html#naive-bayes. Último acceso: : octubre 2025.
- Naive Bayes - RDD-based API. https://spark.apache.org/docs/3.5.6/mllib-naive-bayes.html. Último acceso: : octubre 2025.