Tema 3. Clasificadores escalables básicos.

Los contenidos de este tema son:

3.1 Creación de modelos con ML: Transformación a DF “features” y “labels”.
3.2 Árboles de decisión en ML.
3.3 Naive Bayes en ML: Ejemplo de filtro anti-spam.

El tema incluye una actividad de **autoevaluación** con dos grupos de ejercicios. El primero está asociada a la creación de árboles de decisión. El segundo consiste en realizar los ejercicios que se proponen en el ejemplo del filtro antispam.

El tema finaliza con la actividad de **evaluación**: “Limpieza, transformación y creación del primer modelo aplicado al proyecto software.”

El primer apartado, Creación de modelos con ML: Transformación a DF “features” y “labels”, explica cómo transformar los conjuntos de datos al formato que esperan los algoritmos de aprendizaje. E ilustra cómo se crea un modelo con una evaluación básica del mismo en base a la tasa de error. Es muy importante, se verá en el siguiente tema, hacer una partición de los datos disponibles en Entrenamiento y Prueba. El modelo se crea con Entrenamiento y se evalúa con Prueba.

El apartado “Árboles de decisión con ML” explica con detalle los parámetros más importantes del algoritmo de inducción de árboles de ML. La construcción de los árboles se propone como ejercicio de autoevaluación, pues ya se ha hecho en el tema anterior. **Atención**: los árboles en ML son siempre binarios (cada nodo tiene dos hijos en vez de uno por valor, de aquí la importancia de indicar qué atributos son categóricos) y no se podan a posteriori: dos limitaciones importantes en aras de la escalabilidad del algoritmo.

El último apartado “Naive Bayes en ML: Ejemplo de filtro anti-spam.”, empieza describiendo los parámetros del algoritmo Naive Bayes y los tipos de algoritmos. Es importante saber que **los algoritmos NB para texto no pueden utilizarse en otro tipo de datos** y que requieren determinados modelos de texto. Los modelos básicos de texto se describen brevemente en el material complementario. A continuación, se ilustra un ejemplo básico de filtro antispam. Los pasos para crear el filtro se explican, pero la creación se deja como ejercicio de autoevaluación. El tema no profundiza en el procesamiento de texto, pero proporciona un enlace para que el estudiante interesado pueda hacerlo.

Asumimos en este tema que el alumno conoce los algoritmos básicos de Inducción de árboles de decisión y Naive Bayes. Si no es así, os recomendamos la lectura del material complementario.

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

Transformación a Data Frame “features” y “labels”

Creación y evaluación de un modelo de clasificación con *ML*

Contenido

1. Planteamiento.
2. Lectura sobre *DataFrame* con `StructType`.
3. Transformación de datos para la creación del modelo .
4. Transformación de atributos nominales (categóricos).
5. Creación de columnas `feature` y `label`.
6. Entrenamiento y predicción.
7. Evaluación.
8. Ejercicio.
9. Referencias.

1. Planteamiento

- Continuamos la introducción a *ML* describiendo
  - Cómo transformar un conjunto de datos al formato que esperan los modelos de clasificación que proporciona *ML.*
  - Cómo crear el modelo y una primera evaluación del mismo.
- En resumen, necesitamos un `DataFrame` con dos columnas: `features` y `label`
  - `features` contiene el vector de características.
  - `label` la información de clase.
  - Son el equivalente a los `LabeledPoint` de Mllib.
- Examinaremos las principales utilidades que ofrece la biblioteca ML para realizar esta transformación: `StringIndexer`, `OneHotEncoder, VectorIndexer y VectorAssembler.`
- Usaremos el conjunto de datos car.data, disponible en https://archive.ics.uci.edu/ml/datasets/Car+Evaluation

Conjunto de datos

- https://archive.ics.uci.edu/ml/datasets/Car+Evaluation

- Tarea de clasificación.
- Clase: evaluación de un coche.
- Posibles valores: unacc, acc, good, vgood.

- Atributos: 6, categóricos.
- coste:_compra: vhigh, high, med, low.
  coste:_mantenimiento: vhigh, high, med, low.
  puerta: 2, 3, 4, 5more.
  personas: 2, 4, more.
  maletero: small, med, big.
  seguridad: low, med, high.

Archivo car.data

- 1728 instancias, 6 atributos + clase.

```
~/Datos$ vi car.data
vhigh,vhigh,2,2,small,low,unacc
vhigh,vhigh,2,2,small,med,unacc
vhigh,vhigh,2,2,small,high,unacc
vhigh,vhigh,2,2,med,low,unacc
vhigh,vhigh,2,2,med,med,unacc
vhigh,vhigh,2,2,med,high,unacc
vhigh,vhigh,2,2,big,low,unacc
```

- Atributos y clase, categóricos.
- Valores ausentes: ninguno.

Distribución de clases

- Class Distribution (number of instances per class)

Class	N	N[%]
Unacc	1210	(70.023 %)
acc	384	(22.222 %)
good	69	(3.993 %)
v-good	65	(3.762 %)

Fuente: CreacionModelosClasificacionML.scala

```
/*                                                    */
/*     Creación de un modelo con ML                   */
/*                                                    */
/*     Ejemplo: Árbol de decisión                     */
/*                                                    */
/*     Lectura en un DF, con StructType               */
/*     Transformación de datos en un DataFrame        */
/*      para crear DataFrame Features, Labels         */
/*     Inducción de un árbol de decisión              */
/*     Evaluación básica con ML                       */
/*                                                    */
/*     Dpto. Informática, EII, UVa, Septiembre 2025*/
```

2. Lectura sobre *DataFrame:* definir estructura

```
/* Directorio con los datos y conjunto de datos */
val PATH="/home/calonso/Datos/"
val DATA="car.data"

// Creamos  el schema con StrucType, de acuerdo a la estructura de las filas del fichero de datos
import org.apache.spark.sql.types.{StructType,StructField,StringType,DoubleType}
val carSchema = StructType(Array(
    StructField("coste_compra",StringType,true),
    StructField("coste_mantenimiento",StringType,true),
    StructField("puertas",StringType,true),
    StructField("personas",StringType,true),
    StructField("maletero",StringType,true),
    StructField("seguridad",StringType,true),
    StructField("clase",StringType,true)
))
```

Lectura sobre *DataFrame*

```
// Leemos el DataFrame con la estructura definida
import org.apache.spark.sql.Row
val rawcarDF =
spark.read.format("csv").schema(carSchema).load(PATH + DATA)

// Eliminamos posibles líneas vacías con la función Na drop,
y lo mostramos
val carDF= rawcarDF.na.drop("all")
carDF.show(5)
```

```
+------------+------------------+------+--------+--------+---------+-----+
|coste_compra|coste_mantenimiento|puertas|personas|maletero|seguridad|clase|
+------------+------------------+------+--------+--------+---------+-----+
|       vhigh|             vhigh|     2|       2|   small|      low|unacc|
|       vhigh|             vhigh|     2|       2|   small|      med|unacc|
|       vhigh|             vhigh|     2|       2|   small|     high|unacc|
|       vhigh|             vhigh|     2|       2|     med|      low|unacc|
|       vhigh|             vhigh|     2|       2|     med|      med|unacc|
+------------+------------------+------+--------+--------+---------+-----+
only showing top 5 rows
```

3. Transformación de datos para la creación del modelo: resumen de utilidades

- Los algoritmos de aprendizaje de ML esperan un `DataFrame` con al menos dos columnas: `features` y `label`.

- `features` es un vector de `doubles`.
- `label` es un entero índice al valor de la clase, también `double`.

- Si creamos un `DataFrame` a partir de un RDD que contiene `LabeledPoint` con `toDF`, el `DataFrame` resultante contiene las columnas `features` y `label`, creadas a partir de los `LabeledPoint`.

- Pero es mucho mas eficiente transformar los datos en el `DataFrame` frente a hacerlo en un RDD (y luego transformarlo a `DataFrame`)
  - Porque los `DataFrames` se pueden procesar por columnas con facilidad.

Para crear la columna `features`

- ML proporciona `VectorAssembler.`
- Crea una columna `features` con las columnas que deseamos.
- Sus elementos son doubles
  - También incluye metadatos sobre los contenidos de la columna.
  - Especialmente para informar a los algoritmos de qué atributos son categóricos.

- Pero antes suele ser preciso preprocesar los atributos de los datos .

- Especialmente si son nominales (categóricos).

- Para ello, ML proporciona `StringIndexer, VectorIndexer` y OneHotEncoder
  - Que también proporcionan los metadatos para `VectorAssembler.`

Para crear la columna `label`

- La columna `label` describe los valores de la clase.
- Mediante un índice: entero (aunque se almacene como `double`).

- Para ello ML proporciona `StringIndexer`.

4. Transformación de atributos nominales (categóricos)

- ML proporciona `StringIndexer y OneHotEncoder.`
- `StringIndexer` permite convertir `Strings` con valores categóricos a índices enteros a dichos valores
  - Informa a los algoritmos de aprendizaje de que es un atributo categórico.
- Para utilizar la codificación 1 de k: `OneHotEncoder.`

StringIndexer

- Es un *estimator.*
- Creamos una instancia indicándo la(s) columna(s) de entrada y la(s) de salida.
- Se ajusta con `fit` y se aplica con `transform`.

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

val si = new
StringIndexer().setInputCol("category").setOutputCol("
categoryIndex")
val sm:StringIndexerModel = si.fit(miDataFrame)
newDataFrame =
sm.transform(miDataFrame).drop("category")
```

Ejemplo de `StringIndexer`

- Suponer que `miDataFrame` tiene las columnas `"id"` y `"category"`:

```
id  | category
----|----------
  0 | a
  1 | b
  2 | c
  3 | a
  4 | a
  5 | c
```

- `"category"` es una columna de strings con tres etiquetas: “a”, “b”, y “c”. Si aplicamos `StringIndexer` con `"category"` como columna de entrada y `"categoryIndex"` como columna de salida, eliminando la columna `"category"`, (ejemplo de la página anterior) obtenemos el siguiente `DataFrame:`

```
id |  categoryIndex
---|---------------
 0 | 0.0
 1 | 2.0
 2 | 1.0
 3 | 0.0
 4 | 0.0
 5 | 1.0
```

Adaptado de: https://spark.apache.org/docs/3.5.6/ml-features.html#stringindexer

IndexToString

- Permite recuperar las etiquetas originales que se han codificado con `StringIndexer.`
- Suponer el siguente `DataFrame` creado con `StringIndexer:`

```
id  |  categoryIndex
----|--------------------
  0 | 0.0
  1 | 2.0
  2 | 1.0
  3 | 0.0
  4 | 0.0
  5 | 1.0
```

- Aplicando IndexToString con "categoryIndex" como columna de entrada y "originalCategory" como columna de salida, recuperamos las etiquetas originales (se infieren a partir de los metadatos de las columnas):

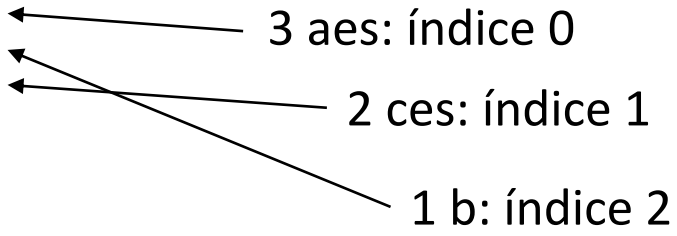
```
id | categoryIndex   | originalcategory
---|-----------------|-----------------------
 0 | 0.0             | a
 1 | 2.0             | b
 2 | 1.0             | c
 3 | 0.0             | a
 4 | 0.0             | a
 5 | 1.0             | c
```

Adaptado de:
https://spark.apache.org/docs/3.5.6/ml-features.html#stringindexer

Asignación de índices con `StringIndexer`

- Por defecto: "`frequencyDesc`":

id	categoryIndex	originalcategory
0	0.0	a
1	2.0	b
2	1.0	c
3	0.0	a
4	0.0	a
5	1.0	c



- El criterio se puede modificar con `setStringOrderType("alphabetDesc")`:

id	categoryIndex	originalcategory
0	0.0	a
1	1.0	b
2	2.0	c
3	0.0	a
4	0.0	a
5	2.0	c

- **Obligatorio** en las **entregas de software**: garantiza que la codificación del modelo es la misma que la que se obtiene con cualquier conjunto de datos (donde ocurran todas las categorías).

Transformación de múltiples columnas con `StringIndexer`: nombres de las columnas

```
/*      Transformamos a DF con índices  numéricos Double      */
/*                                                            */
/*      con StringIndexer                                     */
import org.apache.spark.sql.DataFrame
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.StringIndexerModel

// Obtenemos el nombre de las columnas de carDF, salvo la clase
val attributeColumns = carDF.columns.toSeq.filter(_ !=
"clase").toArray

// Generamos los nombres de las nuevas columnas
 val outputColumns = attributeColumns.map(_ + "-num").toArray
```

Transformación de múltiples columnas con `StringIndexer`: parametrizar StringIndexer

```
// Creamos StringIndexer para transformar todas las  columnas de los atributos
// Asignando indices por orden alfabético descendiente
// para que la asignación no dependa del conjunto de entrenamiento/prueba

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

Crear `StringIndexerModel` y aplicar transformación

```
// Creamos el StringIndexerModel
val simColumns = siColumns.fit(carDF)

// Creamos el DF carDFnumeric transformando todos sus atributos
(sin la clase)
// Eliminando las columnas originales
val carDFnumeric =
simColumns.transform(carDF).drop(attributeColumns:_*)
```

Examinamos carDFnumeric

```
carDFnumeric.show(10)
+-----+----------------------+----------+-----------+----------------+-----------+-------------+
|clase|coste_mantenimiento-num|puertas-num|maletero-num|coste_compra-num|personas-num|seguridad-num|
+-----+----------------------+----------+-----------+----------------+-----------+-------------+
|unacc|                   0.0|       3.0|        0.0|             0.0|        2.0|          1.0|
|unacc|                   0.0|       3.0|        0.0|             0.0|        2.0|          0.0|
|unacc|                   0.0|       3.0|        0.0|             0.0|        2.0|          2.0|
|unacc|                   0.0|       3.0|        1.0|             0.0|        2.0|          1.0|
|unacc|                   0.0|       3.0|        1.0|             0.0|        2.0|          0.0|
|unacc|                   0.0|       3.0|        1.0|             0.0|        2.0|          2.0|
|unacc|                   0.0|       3.0|        2.0|             0.0|        2.0|          1.0|
|unacc|                   0.0|       3.0|        2.0|             0.0|        2.0|          0.0|
|unacc|                   0.0|       3.0|        2.0|             0.0|        2.0|          2.0|
|unacc|                   0.0|       3.0|        0.0|             0.0|        1.0|          1.0|
+-----+----------------------+----------+-----------+----------------+-----------+-------------+
only showing top 10 rows
```

Alternativa: `VectorIndexer`

- Por defecto, todas las características del vector de características de un *DataFrame* son consideradas continuas por los algoritmos de aprendizaje de *ML*
  - Salvo que hayamos utilizado `StringIndexer o VectorIndexer`
    - Pero se siguen representando como *`Double`*.

- Para informar a los algoritmos de qué características son categóricas ML proporciona `StringIndexer` y `VectorIndexer.`

- Esta información se añade como metadatos que utilizan los algoritmos.

- Importante en los algoritmos que pueden explotar esta información (por ejemplo, los árboles de decisión)

VectorIndexer

- `VectorIndexer` toma como entrada una columna de tipo `Vector` (de ML) de un `DataFrame`.
- Todos los elementos con menos valores distintos que `MaxCategories` se consideran categóricos.
- Permite procesar simultáneamente todas las características nominales (que antes se habrán fusionado con `VectorAssembler`)
  - Asumiendo que típicamente las características continuas tienen muchos más atributos que `MaxCategories` (cuidado con esta suposición).
- Es un *Estimator.*

Ejemplo de VectorIndexer

```
/*       Añadimos información (metadatos) para que en la columna features,*/
/*       un vector de doubles, las características con                    */
/*       menos de 5 valores sean consideradas categóricas                 */

//import org.apache.spark.ml.feature.VectorIndexer

//val vi=new
VectorIndexer().setInputCol("features").setOutputCol("features_cateroric"
).setMaxCategories(5)

//val viModel=vi.fit(miDataFrame)

//val nuevoDataFrame
=viModel.transform(miDataFrame).drop("features").withColumnRenamed
("features_categoric“, "features").
```

- **NO LO UTILIZAMOS EN ESTE EJEMPLO**, porque todos los atributos originales son categóricos.

Codificación 1 de k: `OneHotEncoderEstimator`

- Otra forma de transformar atributos categóricos a numéricos es la codificación 1_de_k.
- Se emplea para evitar que los algoritmos utilicen el orden de los índices (0<1<2<…) cuando los valores nominales no están ordenados.
- Recordar:

Marital status
Separated
Divorced
Widowed
Married
Widowed
Separated
Never married
Married

→

Married	Divorced	Separated	Widowed	Never Married
0	0	1	0	0
0	1	0	0	0
0	0	0	1	0
1	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
1	0	0	0	0

Figure 8.3 One-hot encoding of the `marital status` column. It gets expanded into five new columns, each containing only ones and zeros. Each row contains only one `1` in the column corresponding to the original column value.

Tomado de Spark in Action

OneHotEncoder

- Es un *estimator.*
- Creamos una instancia indicándole las columnas de entrada y las de salida.
- Se ajusta con `fit` y se aplica con `transform.`
- Añade columnas codificando las columnas de entrada como vectores de `double` dispersos.

```
import org.apache.spark.ml.feature.OneHotEncoder
import org.apache.spark.ml.feature.OneHotEncoderModel

val df = spark.createDataFrame(Seq(
  (0.0, 1.0), (1.0, 0.0), (2.0, 1.0),
  (0.0, 2.0), (0.0, 1.0),(2.0, 0.0)
 )).toDF("categoryIndex1", "categoryIndex2")
```

Tomado https://spark.apache.org/docs/3.5.6/ml-features.html#onehotencoder

OneHotEncoder

- Es un *estimator.*
- Creamos una instancia indicándole las columnas de entrada y las de salida.
- Se ajusta con `fit` y se aplica con `transform.`
- Añade columnas codificando las columnas de entrada como vectores de `double` dispersos.

```
val hot = new OneHotEncoder()
.setInputCols(Array("categoryIndex1", "categoryIndex2"))
.setOutputCols(Array("categoryVec1", "categoryVec2"))
val hotm = encoder.fit(elDataFrame)
val nuevoDataFrame = hotm.transform(elDataFrame)
```

Tomado de https://spark.apache.org/docs/3.5.6/ml-features.html#onehotencoder

Transformación de múltiples columnas con `OneHotEncoder`: nombres de las columnas

```
/*      Tansformamos a DF con codificación 1 de K en atributos */
/*                                                             */
/*      con OneHotEncoder                                      */
/*                                                             */

import org.apache.spark.ml.feature.OneHotEncoder
import org.apache.spark.ml.feature.OneHotEncoderModel


// Generamos los nombres de las nuevas columnas
val inputColumns = outputColumns
val outputColumns = attributeColumns.map(_ + "-hot").toArray
```

Transformación de múltiples columnas con `OneHotEncoder` : parametrizar OneHotEncoder

```
/*      Tansformamos a DF con codificación 1 de K en atributos */
/*                                                             */
/*      con OneHotEncoder                                      */
/*                                                             */
```

```
// Creamos OneHotEncoder para transformar todos los
atributos, salvo la clase
val hotColumns = new
OneHotEncoder().setInputCols(inputColumns).setOutputCols(out
putColumns)
```

Crear `OneHotEncoderModel` y aplicar transformación

```
//Creamos el OneHotEncoderModel
val hotmColumns = hotColumns.fit(carDFnumeric)

// Creamos el DF carDFhot transformando todos sus atributos
(sin la clase)
// Eliminando las columnas originales
val carDFhot =
hotmColumns.transform(carDFnumeric).drop(inputColumns:_*)
```

Examinamos carDFhot

```
carDFhot.show(10)
+-----+-----------+---------------+----------------------+------------+-------------+-------------+
|clase|puertas-hot|coste_compra-hot|coste_mantenimiento-hot|seguridad-hot| personas-hot| maletero-hot|
+-----+-----------+---------------+----------------------+------------+-------------+-------------+
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[1],[1.0])|    (2,[],[])|(2,[0],[1.0])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[0],[1.0])|    (2,[],[])|(2,[0],[1.0])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|    (2,[],[])|    (2,[],[])|(2,[0],[1.0])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[1],[1.0])|    (2,[],[])|(2,[1],[1.0])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[0],[1.0])|    (2,[],[])|(2,[1],[1.0])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|    (2,[],[])|    (2,[],[])|(2,[1],[1.0])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[1],[1.0])|    (2,[],[])|    (2,[],[])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[0],[1.0])|    (2,[],[])|    (2,[],[])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|    (2,[],[])|    (2,[],[])|    (2,[],[])|
|unacc|  (3,[],[])|   (3,[0],[1.0])|         (3,[0],[1.0])|(2,[1],[1.0])|(2,[1],[1.0])|(2,[0],[1.0])|
+-----+-----------+---------------+----------------------+------------+-------------+-------------+
only showing top 10 rows
```

Nota: ML codifica uno de los k valores como un valor por defecto con todas las columnas a 0, ahorrándose una columna. Por ejemplo `puertas-hot` es un vector disperso con dimensión 3, a pesar de que el atributo original tiene 4 valores.

5. Creación de columnas `feature` y `label`

- Los algoritmos de aprendizaje esperan una columna `features` con un único vector de `double` y una columna `label` con un índice a la clase, también `double`.

- `VectorAssembler y StringIndexer.`

VectorAssembler

- El paso final para obtener el vector de características consiste en fusionar todas las columnas transformadas en una única columna, `features`, que es un vector de `double`.
- ML proporciona `VectorAssembler`.
- Es un *transformer.*
- Solo acepta como entrada columnas de tipo `numeric, boolean` y `vector`.
- Conserva la información de los metadatos.

Creación de la columna `features`: parametrizar `VectorAssembler`

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

- Para crear el vector de características en columna `"features"`

```
/  Definimos columna Features con todos los atributos, menos la clase, con VectorAssembler (se muestran dos alternativas)
//val va = new VectorAssembler().setOutputCol("features").setInputCols(carDFhot.columns.diff(Array("clase")))

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

Creación de la columna `features`: aplicar la transformación

```
// Creamos el DataFrame carFeaturesClaseDF con columnas features y clase
val carFeaturesClaseDF = va.transform(carDFhot).select("features", "clase")
```

Lo examinamos

```
carFeaturesClaseDF.show(10)
+--------------------+-----+
|            features|clase|
+--------------------+-----+
|(15,[0,3,11,14],[...|unacc|
|(15,[0,3,11,13],[...|unacc|
|(15,[0,3,11],[1.0...|unacc|
|(15,[0,3,12,14],[...|unacc|
|(15,[0,3,12,13],[...|unacc|
|(15,[0,3,12],[1.0...|unacc|
|(15,[0,3,14],[1.0...|unacc|
|(15,[0,3,13],[1.0...|unacc|
|(15,[0,3],[1.0,1.0])|unacc|
|(15,[0,3,10,11,14...|unacc|
+--------------------+-----+
only showing top 10 rows
```

Creación de la columna `label`

- Es la etiqueta de clase.
- Debe de ser un entero (codificada como `double`).
- Se puede codificar a partir de la clase original con `StringIndexer`.

StringIndexer para la columna label

```
/*      Transformamos la etiqueta de clase a enteros          */
/*      y renombramos la columna "clase" a "label"            */
/*                                                            */
/*      también con StringIndexer                             */

import org.apache.spark.ml.feature.StringIndexer

// creamos el StringIndexer para la clase
val indiceClase= new
StringIndexer().setInputCol("clase").setOutputCol("label").setStr
ingOrderType("alphabetDesc")

// Creamos el DataFrame carFeaturesLabelDF con columnas features
y label
val carFeaturesLabelDF =
indiceClase.fit(carFeaturesClaseDF).transform(carFeaturesClaseDF)
.drop("clase")
```

Lo examinamos

```
carFeaturesLabelDF.show(10)
+--------------------+-----+
|            features|label|
+--------------------+-----+
|(15,[0,3,11,14],[...|  1.0|
|(15,[0,3,11,13],[...|  1.0|
|(15,[0,3,11],[1.0...|  1.0|
|(15,[0,3,12,14],[...|  1.0|
|(15,[0,3,12,13],[...|  1.0|
|(15,[0,3,12],[1.0...|  1.0|
|(15,[0,3,14],[1.0...|  1.0|
|(15,[0,3,13],[1.0...|  1.0|
|(15,[0,3],[1.0,1.0])|  1.0|
|(15,[0,3,10,11,14...|  1.0|
+--------------------+-----+
only showing top 10 rows
```

6. Entrenamiento y predicción

- Crear particiones entrenamiento y prueba.
- Crear instancia del modelo de clasificación.
- Es un *Estimator.*
- Entrenar con `fit.`
- Predecir con `transform.`

Partición aleatoria

```
/* Realizamos una partición aleatoria de los datos          */
/*   66% para entrenamiento, 34% para prueba                */

/* Fijamos seed para usar la misma partición en distintos
ejemplos*/

val dataSplits = carFeaturesLabelDF.randomSplit(Array(0.66,
0.34), seed=0)

val trainCarDF = dataSplits(0)
val testCarDF = dataSplits(1)
```

Creamos una instancia del modelo deseado: *estimator*

```
/* Importamos de ML */
import
org.apache.spark.ml.classification.DecisionTreeClassifier

/* Creamos una instancia de DecisionTreeClassifier          */
val DTcar=new DecisionTreeClassifier()
```

- En *ML* siempre se crean instancias de modelos que luego se entrenan.
- Son *estimators.*

Entrenamos y creamos el modelo: *fit*

```
/* Entrenamos el modelo: Árbol de Decisión con los
parámetros por defecto  */
val DTcarModel=DTcar.fit(trainCarDF)
```

- *ML* siempre utiliza ***fit*** para ajustar los parámetros de un modelo a los datos, lo que incluye crear un modelo utilizando un algoritmo de aprendizaje entrenando sobre un conjunto de datos descrito por un *DataFrame.*

- `DTcarModel` es un *transformer.*

Podemos examinar el árbol:

```
DTcarModel.toDebugString
"DecisionTreeClassificationModel: uid=dtc_6b8e207593f8, depth=5, numNodes=23,
numClasses=4, numFeatures=15
  If (feature 14 in {0.0})
   If (feature 10 in {0.0})
    If (feature 9 in {0.0})
     Predict: 1.0
    Else (feature 9 not in {0.0})
     If (feature 11 in {0.0})
      Predict: 3.0
     Else (feature 11 not in {0.0})
      If (feature 2 in {0.0})
       Predict: 1.0
      Else (feature 2 not in {0.0})
       Predict: 3.0
   Else (feature 10 not in {0.0})
    If (feature 0 in {0.0})
     If (feature 3 in {0.0})
      Predict: 3.0
     Else (feature 3 not in {0.0})
      If (feature 2 in {0.0})
       Predict: 1.0
      Else (feature 2 not in {0.0})
       Predict: 3.0
    Else (feature 0 not in {0.0})
     If (feature 4 in {0.0})
      If...
```

Predecimos sobre el conjunto de prueba: *transform*

```
/* Predecimos la clase de los ejemplos de prueba        */

val predictionsAndLabelsDF =
DTcarModel.transform(testCarDF).select("prediction", "label")
```

- *ML* siempre utiliza ***transform*** para aplicar un modelo sobre un *DataFrame*, generando otro *DataFrame* al que añade una columna con las predicciones.

Examinamos predicciones

```
predictionsAndLabelsDF.show(10)
+----------+-----+
|prediction|label|
+----------+-----+
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
|       1.0|  1.0|
+----------+-----+
only showing top 10 rows
```

7. Evaluación

Evaluamos con *ML*

- *ML* soporta la clase *MulticalssMetrics.*
- Es la única métrica que proporciona la tasa de acierto: “accuracy”.

```
/* Importamos de ML     */
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

/* Creamos una instancia de clasificación multiclass  */
val metrics = new MulticlassClassificationEvaluator()

/* Fijamos como métrica la tasa de acierto: accuracy */
metrics.setMetricName("accuracy")
```

Calculamos su tasa de error

```
/* Calculamos la tasa de acierto        */
val acierto = metrics.evaluate(predictionsAndLabelsDF)

/* Calculamos el error  */
val error = 1 – acierto

// Lo mostramos
println(f"Tasa de error= $error%1.3f")
Tasa de error= 0,190
```

Guardamos el modelo: *save*

```
/* Guardamos el modelo */


// Con opción overwrite
//DTcarModel.write.overwrite().save(PATH + "DTcarModelML")
```

8. Ejercicio de autoevaluación

- Repetir el ejemplo, leyendo los datos directamente a un *DataFrame* y sin utilizar la codificación 1 de K.

- Terminar el ejemplo y denominar al nuevo archivo `arbolesDecisionMLstringIndexer.scala`.

- Comparar los árboles obtenidos con ambos métodos.

9. Referencias (I)

- Car Evaluation Data Set. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Car+Evaluation. Último acceso: septiembre 2025.

- Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

- B. Zupan, M. Bohanec, I. Bratko, J. Demsar, “Machine Learning by Function Decomposition”, Proceedings of the Fourteenth International Conference Machine Learning (ICML'97), pp. 421 – 429, Nashville, Tennessee, July 1997.

- Petar Zečević y Marko Bonaći. Spark in Action. Manning Publications. 2016. ISBN: 9781617292606. https://www.manning.com/books/

Referencias (II)

- Extracting, transforming and selecting features. https://spark.apache.org/docs/3.5.6/ml-features.html. Último acceso: septiembre 2025.
- Extracting, transforming and selecting features: VectorAssembler. https://spark.apache.org/docs/3.5.6/ml-features.html#vectorassembler. Último acceso: septiembre 2025.
- Class Vectors. https://spark.apache.org/docs/3.5.6/api/java/org/apache/spark/ml/linalg/Vectors.html. Último acceso: septiembre 2025.
- Classification and regresion. Decision tree classifier. https://spark.apache.org/docs/3.5.6/ml-classification-regression.html#decision-tree-classifier. Último acceso: septiembre 2025.
- Decision Trees - RDD-based API. https://spark.apache.org/docs/3.5.6/mllib-decision-tree.html . Último acceso: septiembre 2025.

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

Inducción de árboles de decisión en *ML*

Contenido

1. Árboles de decisión en *ML.*
2. Parámetros.
3. Ejemplo: datos car y preparación de datos.
4. Inducción de árboles y ejercicios.
5. OneHotEncoder y árboles de decisión.
6. Referencias.

1. Árboles de decisión en *ML*

- *ML* incluye clases para inducir **árboles de decisión binarios**, con atributos continuos y/o discretos.
- Clasificación binaria o multiclase.
- Regresión.

```
import org.apache.spark.classification.DecisionTreeClasifier
```

- `DecisionTreeClassifier` es un `estimator`.
- Se crea una instancia y se parametriza.
- Se entrena con `fit` para generar un. `DecisionTreeClassifierModel`.

- Se aplica con `transform`.

2. Parámetros de entrada/salida

- Entrada: columnas del Data Frame para entrenar/predecir.
- `labelCol`, por defecto "`label`", tipo `Double`.
- `FeaturesCol`, por defecto "`features`", tipo `Vector`.

- Salida: columnas del nuevo Data Frame al aplicar `transform`.
- `PredictionCol`, por defecto "`prediction`", de tipo `Double`
  - La clase predicha.
- `rawPredictionCol` por defecto "`rawPrediction`", de tipo `Vector`
  - Vector con el número de instancias de entrenamiento de cada clase del nodo hoja que hace la predicción.
- `probabilityCol`, por defecto "`probability`", de tipo `Vector`
  - A partir del vector anterior, normalizando.

- Mejor dejar los nombres por defecto.
- Comunes a la mayoría de los algoritmos.

Parámetros del algoritmo

- `impurity:` Medida de impureza para seleccionar los atributos.
- `maxDepth:` Profundidad máxima del árbol.
- `maxBins:` Número máximo de particiones de atributos continuos.
- `minInstancesPerNode:` número mínimo de instancias en un nodo hijo para realizar una partición.

- Los tres últimos influyen notablemente en el sobreajuste.

Ejemplo

```
/* Creamos una instancia de DecisionTreeClassifier  */
val DTcar=new DecisionTreeClassifier()

/* Elegimos parámetros del modelo        */
val impureza =  "entropy"
val maxProf = 3
val maxBins =5

/* Fijamos parámetros del modelo        */
DTcar.setImpurity(impureza)
DTcar.setMaxDepth(maxProf)
DTcar.setMaxBins(maxBins)
```

- Alternativa:

```
val DTcar=new
DecisionTreeClassifier().setImpurity(impureza).setMaxDepth(max
Prof).setMaxBins(maxBins)
```

impurity

- Los atributos se seleccionan según la **ganancia de información**.
- Clasificación: dos medidas de impureza, gini, entropy.

- Entropy: $H(D)=-\sum_{i=1}^{n} f_i \log_2 f_i$ — ID3, C4.5
- Índice de gini: $I_G(D)= -\sum_{i=1}^{n} f_i \ (1 - f_i)$ — CART

- En principio:
  - "`entropy`" atributos categóricos.
  - "`gini`" numéricos.

- En la práctica, muy poca diferencia.

maxDepth

- Es la profundidad máxima que puede alcanzar el árbol.
- Un árbol de profundidad 0 tiene un único nodo hoja.
- Un árbol de profundidad 1 tiene un nodo interno y dos nodos hojas.
- Valor por defecto: 5

- Es un **parámetro crítico** que hay que ajustar en la etapa de selección de modelos.

- Compromiso:
  - Árbol más profundo: puede sobreajustar.
  - Árbol menos profundo: puede no incluir un atributo necesario.

- Mejor solución: podar después de crear árbol de prof. máxima, pero no escala bien. *ML*, *Mllib* no lo soportan.

maxBins

- Se consideran los valores de los atributos continuos como posibles umbrales para “discretizar” cada atributo.
- Siempre binario: posibles particiones a más profundidad, distintas en cada rama.

- También para los categóricos si tienen más de 2 valores: $(2^{M-1} -1)$ si M valores. Se agrupan 1 frente al resto:
  - azul | rojo, verde azul, rojo | verde rojo| azul, verde

- Valor máximo: 32 (o el número de instancias)
  - Valor por defecto: 32.
  - Valores elevados favorecen el sobreajuste.
  - Inicialmente, mejor probar un valor bajo, 3, 5 o 7, por ejemplo.
- Como mínimo: número máximo de valores de un atributo categórico - 1 (en clasificación binaria).

`minInstancesPerNode`

- Cada atributo seleccionado induce una partición sobre la parte del conjunto de entrenamiento que llega al nodo.
- Detiene la construcción de la rama si ningún atributo genera particiones con menos de `minInstancesPerNode` en cada rama.
- Valor por defecto: 1.
- No suele ser interesante generar particiones con solo dos instancias.
- Valores pequeños favorecen el sobreajuste.

Parada

- Utiliza tres criterios de parada.

- La profundidad del nodo es igual al parámetro `maxDepth (5)`.
- Ningún candidato mejora la ganancia de información sobre el parámetro `minInfoGain (0)`.
- Ningún candidato produce un nodo con al menos `minInstancesPerNode` (1).

3. Ejemplo: datos car y preparación de datos

- https://archive.ics.uci.edu/ml/datasets/Car+Evaluation

- Tarea de clasificación.
- Clase: evaluación de un coche.
- Posibles valores: unacc, acc, good, vgood.
- Atributos: 6, discretos.
- coste:_compra: vhigh, high, med, low.
  coste:_mantenimiento: vhigh, high, med, low.
  puerta: 2, 3, 4, 5more.
  personas: 2, 4, more.
  maletero: small, med, big.
  seguridad: low, med, high.
- Valores ausentes: ninguno.

Distribución de clases

- Class Distribution (number of instances per class)

Class	N	N[%]
Unacc	1210	(70.023 %)
acc	384	(22.222 %)
good	69	(3.993 %)
v-good	65	(3.762 %)

Procesamos conjunto de datos

- Como en script "`creacionModelosClasificacionML.scala`"
- Recordar: Lectura a Data Frame, `StringIndexer`, `OneHotEncoder`, `VectorAssembler`, `StringIndexer` para la clase
- En torno a la línea 145:

```
// Creamos el DataFrame carFeaturesLabelDF con columnas features y label
val carFeaturesLabelDF = indiceClase.fit(carFeaturesClaseDF).transform(carFeaturesClaseDF).drop("clase")
carFeaturesLabelDF.show(10)
```

```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|(15,[0,3,11,14],[...|  1.0|
|(15,[0,3,11,13],[...|  1.0|
|(15,[0,3,11],[1.0...|  1.0|
|(15,[0,3,12,14],[...|  1.0|
|(15,[0,3,12,13],[...|  1.0|
|(15,[0,3,12],[1.0...|  1.0|
```

Partición aleatorio entrenamiento y prueba

```
/* Realizamos una partición aleatoria de los datos                */
/*   66% para entrenamiento, 34% para prueba                      */

/* Fijamos seed para usar la misma partición en distintos
ejemplos*/
val dataSplits = carFeaturesLabelDF.randomSplit(Array(0.66, 0.34),
seed=0)
val trainCarDF = dataSplits(0)
val testCarDF = dataSplits(1)
```

Creamos instancia `DecisionTreeClassfier` y parametrizamos

```
/* Importamos de ML     */
import org.apache.spark.ml.classification.DecisionTreeClassifier

/* Creamos una instancia de DecisionTreeClassifier                */
val Dtcar=new DecisionTreeClassifier()

/* Elegimos parámetros del modelo        */
val impureza =  "entropy"
val maxProf = 3
val maxBins =5

/* Fijamos parámetros del modelo        */
DTcar.setImpurity(impureza)
DTcar.setMaxDepth(maxProf)
DTcar.setMaxBins(maxBins)
```

Ejercicio 1: Entrenamos el clasificador, `DTcarModel_A`

- Entrenamos, examinamos y calculamos tasa de error.

Solución 1: Entrenamos el clasificador, `DTcarModel_A`

Calculamos su tasa de error

```
Tasa de error= 0,237
```

Recordar

- Con los parámetros por defecto

```
val impureza =  "entropy"
val maxProf = 5
val maxBins = 32
```

```
numNodes=23
Tasa de error= 0,190
```

Ejercicio 2: `DTcarModel_B`

- Crea árbol de decisión con

```
impureza: entropy
maxProf:10
maxBins=: 5
```

- Evaluar sobre las mismas particiones.

- Mostrar.

Solución 2: `DTcarModel_B`

Examinamos `DTcarModel_B`

Tasa de error `DTcarModel_B`

`Tasa de error= 0,085`

- Pero profundidad 10 y 211 nodos.

Ejercicio 3: `DTcarModel_C`

- Crea árbol de decisión con

```
impureza: entropy
maxProf:10
maxBins=: 150
```

- Evaluar sobre las mismas particiones.

- Mostrar.

Solución 3: `DTcarModel_C`

Examinamos `DTcarModel_C`

Tasa de error `DTcarModel_C`

`Tasa de error= 0,085`

- El mismo árbol que `DTcarModel_B`, (maxBins=5).

- El resultado era de esperar, pues no hay atributos continuos.
- Y usando `OneHotEncoder` todos los atributos toman valor 0/1.
- Sería suficiente **`maxBins=2.`**

5. OneHotEncoder y árboles de decisión

- Vamos a comparar un árbol utilizando la codificación **1 de K** frente a otro que no la utilice.

- En ambos casos utilizamos `StringIndexer` para obtener los índices a los valores de los atributos categóricos.

Ejercicio 4: Inducir un árbol con los parámetros por efecto sin usar OneHotEncoder, `DTcarModel_D`

- Codificar los atributos categóricos solo con `StringIndexer`

Solución 4: nueva columna `features`

Solución 4: examinamos nuevo *Data Frame*

`carFeaturesClaseDF`

Solución 4: creamos el árbol `DTcarModel_D`

Solución 4: examinamos el árbol

Solución 4: tasa de error sin usar OneHotEncoder

```
Tasa de error= 0,144
```

- Sin `OneHotEncoder`.
- Árbol de profundidad 5, 17 nodos y una tasa de error de 0,144.
- Comprobar que aumentando la profundidad a 10, se obtiene un árbol que tiene 125 nodos y una tasa de error de 0,031.

- Con `OneHotEncoder`
- Árbol de profundidad 5, 23 nodos y una tasa de error de 0,190.
- O árbol de profundidad 10, 211 nodos y una tasa de error de 0,085.

- En este caso, la opción con `StringIndexer` es preferible.
- El resultado depende del conjunto de datos y de la partición aleatoria.

OneHotEncoder y árboles de decisión

- No siempre es preferible la codificación **1 de K** con los árboles de decisión.
- Porque introducen nuevos atributos.
- Que permiten realizar particiones más pequeñas del conjunto de entrenamiento.
- Y seleccionar atributos que generalizan peor.

6. Referencias

- Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

- B. Zupan, M. Bohanec, I. Bratko, J. Demsar, “Machine Learning by Function Decomposition”, Proceedings of the Fourteenth International Conference Machine Learning (ICML'97), pp. 421 – 429, Nashville, Tennessee, July 1997.

- Decision trees. https://spark.apache.org/docs/3.5.6/ml-classification-regression.html#decision-trees. Último acceso: octubre 2025.
- DecisionTreeClassifiers. https://spark.apache.org/docs/3.5.6/ml-classification-regression.html#decision-tree-classifier . Último acceso: octubre 2025.
- Decision Trees - RDD-based API. https://spark.apache.org/docs/3.5.6/mllib-decision-tree.html. Último acceso: octubre 2025.

Tema 4. Metodología experimental de evaluación y selección de modelos

Los contenidos de este tema son:

4.1 Creación y evaluación de clasificadores en base a la tasa de error (revisión).

4.2 Estructura del error, Curvas ROC y otras métricas.

4.3 Métricas para Clasificadores Binarios.

4.4 Métricas Muticlase.

4.5 Selección de modelos con ML.

4.6 Selección de modelos para árboles de decisión: ejercicio de autoevaluación.

El tema incluye un ejercicio de autoevaluación y finaliza con la actividad de evaluación: "Selección y evaluación de modelos para el proyecto software."

Este tema está dedicado a un aspecto esencial en el aprendizaje de modelos: la selección de modelos y la evaluación del modelo seleccionado.

Este proceso forma parte de la **etapa de construcción de un modelo** (recordar las etapas que propone CIRSP-DM) **NO** de la etapa de evaluación. La etapa de evaluación en la metodología CRIPS-DM se refiere a la evaluación desde la perspectiva de su aportación al proceso de negocio. Este aspecto no lo abordamos en esta asignatura.

La evaluación de modelos que aquí nos ocupa se refiere a la evaluación de su **utilidad como clasificador** (o predictor, en general). No es el único aspecto relevante a la hora de juzgar un modelo. El coste de su creación, por ejemplo, también es importante. Pero nos permite centrarnos en un aspecto básico (su calidad como clasificador) que puede abordarse de forma sistemática con métodos estadísticos clásicos.

El primer tema (4.1) es esencial, pues plantea los métodos estándar, reconocidos por la comunidad, utilizados para entrenar y evaluar modelos. El tema se centra en estimar la tasa de error. El error cometido por un clasificador es una medida directa de su calidad como clasificador. No es la única métrica y no es la más informativa, pero es la básica. Además, es un primer paso para estimar cualquier otra métrica, por lo que familiarizare con las metodologías propuestas es importante.

La idea fundamental es que el error verdadero que comete un modelo no puede conocerse, pero puede estimarse con suficiente precisión si disponemos de suficientes datos. Pero para ello se parte de tres hipótesis:

1. Todos los datos disponibles se han seleccionado de forma aleatoria según la distribución de probabilidad subyacente en el problema
2. Con los datos disponibles se han generado, al menos, un conjunto de entrenamiento y otro de prueba de forma totalmente aleatoria.

3. La hipótesis, es decir, el modelo, es independiente de los datos del conjunto de prueba.

La primera hipótesis no siempre es cierta y en cualquier caso no suele depender de los creadores de los modelos.

La segunda hipótesis sí podemos cumplirla, y es nuestra responsabilidad que así sea. Si la segunda hipótesis no se cumple, las estimaciones de las tasas de error, y de cualquier otra métrica, no sirven para nada, generando valores optimistas frente a la realidad. Por ejemplo, la tasa de error estimada sobre el conjunto de entrenamiento es nula en muchos casos. Este es el denominado error de resubstitución. Pero el error verdadero puede tomar cualquier valor mayor que cero (salvo en muy contadas ocasiones)

La tercera hipótesis también podemos satisfacerla si no usamos los datos del conjunto de prueba para crear el modelo. Basta con que los conjuntos de entrenamiento y prueba no se solapen. Esta condición se necesita para poder aplicar los modelos paramétricos de las distribuciones que necesitamos: la distribución binomial y la distribución normal.

El apartado 4.2 profundiza en la evaluación de los clasificadores más allá de la tasa de error. La tasa de error nos resume en un único número el comportamiento de un clasificador. Pero muchas veces este resumen es insuficiente. En general interesa saber cómo se equivoca un clasificador, no solo cuántas veces se equivoca. Imaginar un clasificador que determine si un avión va a tener una avería grave durante un vuelo. Dado que esto es altamente improbable, un clasificador que siempre devuelva "No" tiene una tasa de error muy pequeña (menor que 10^{-6}, quizás). Pero es un clasificador totalmente inútil. Lo que realmente interesa es un clasificador que acierte los poquísimos casos en los que se va a producir una avería. Esto lo mide la "tasa de ciertos positivos" que idealmente vale 1.

Este último análisis requiere conocer la estructura del error, que se refleja en la Matriz de Confusión, a partir de la cual se definen las tasas de ciertos positivos, **tp** (también denominada tasas de detección), y de falsos positivos, **fp** (falsas alarmas). Cuando se dispone de varios clasificadores, su comportamiento como clasificador para la clase de interés está perfectamente descrito por ***tp*** **y** ***fp***: **el mejor clasificador es aquel que teniendo una tasa de falsos positivos asumible, tiene una mayor tasa de ciertos positivos**. Este análisis se puede aplicar a un solo clasificador, mediante las Curvas ROC y al área bajo su curva, AUC.

Finalmente, hay que indicar que muchos dominios de aplicación han desarrollado sus propias métricas. Todas ellas se pueden definir a partir de los elementos de la matriz de confusión. Aunque ninguna tiene las propiedades deseables de las curvas ROC y AUC. Algunas de estas métricas se describen brevemente.

El apartado 4.3 presenta las utilidades que ofrecen ML y MLlib para evaluar métricas binarias: `BinaryClassificationEvaluator` , un *evaluator,* en ML y `BinaryClassificationMetrics` en MLlib. Sólo proporcionan áreas bajo curvas. Para otras métricas, se pueden usar variantes multiclase para problemas binarios.

Comenzamos con ML describiendo los parámetros de la métrica binaria. Nos detenemos más en los parámetros relacionados con el cómputo de AUC ROC, pues su obtención correcta depende del modelo inducido. Se analiza en detalle la problemática usando el ejemplo del filtro anti-spam propuesto en el tema 2. Si queremos obtener las curvas, hay que recurría a la métrica binaria de MLlib, sobre RDDs, para generar sus puntos. La problemática de su construcción es la misma que en ML.

Las métricas multiclase se pueden aplicar a modelos de clasificación en los que la etiqueta de clase tenga dos o más valores. En ML disponemos de `MulticlassClassificationEvaluator`, que proporciona numerosas métricas. Describimos sus parámetros e ilustramos su uso sobre el árbol de decisión inducido en el tema 2. Finalizamos describiendo la clase `MulticlassMetrics` de MLlib, porque es la única que nos permite construir la Matriz de Confusión, importante porque nos proporciona la estructura del error del clasificador.

Un comentario: tened cuidado al utilizar métricas ponderadas, sobre todo en aquellos problemas en los que la clase de interés tiene una frecuencia pequeña. Las métricas ponderadas ocultan el comportamiento sobre las clases con menor frecuencia de aparición.

Finalmente, en el tema 4.5 examinamos las utilidades que proporciona Ml para utilizar las métricas anteriores para la selección de modelos. Esto requiere ajustar los parámetros de los algoritmos e implica construir múltiples clasificadores que se evalúan sobre (parte) del conjunto de entrenamiento. **En el proceso de selección de modelos no se utiliza para nada el conjunto de prueba**. ML permite definir y automatizar todo el proceso.

Todo ello se completa con el ejercicio de autoevaluación que se propone en el tema 4.6. Aquí tenéis que utilizar las utilidades que hemos introducido en el tema anterior para ajustar los parámetros de un árbol de decisión y crear el modelo. Todo ello utilizando solo el conjunto de entrenamiento. El ejercicio propone algunas ideas para reducir el número de modelos a construir.

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

Naive Bayes en *ML*

Aplicación a la clasificación de texto: filtro anti spam

Contenido

1. Naive Bayes en *ML.*
2. Parámetros.
3. Ejemplo: datos SMSSpam.
4. Modelo de texto.
5. Inducción de un filtro anti spam con NB multinomial.
6. Filtro Anti Spam con `Pipeline`.
7. Referencias.

1. Naive Bayes en *ML*

- *ML* proporciona la clase `NaiveBayes` para inducir un clasificador Naive Bayes.
- Clasificación binaria o multiclase.
- Varios tipos: atributos discretos (clasificadores de texto), continuos.

```
import org.apache.spark.ml.classification.NaiveBayes
```

- `NaiveBayes` es un `estimator.`
- Se crea una instancia y se parametriza.
- Se entrena con `fit` para generar un `NaiveBayesModel.`

- Se aplica con `transform.`

2. Parámetros de entrada/salida

- Entrada: columnas del Data Frame para entrenar/predecir.
- `labelCol`, por defecto "`label`", tipo `Double`.
- `FeaturesCol`, por defecto "`features`", tipo `Vector`.

- Salida: columnas del nuevo Data Frame al aplicar `transform`.
- `PredictionCol`, por defecto "`prediction`", de tipo `Double`
  - La clase predicha.
- `rawPredictionCol` por defecto "`rawPrediction`", de tipo `Vector`
  - No está documentada.
- `probabilityCol`, por defecto "`probability`", de tipo `Vector`
  - Estimación de probabilidades, calibrado según tipo de modelo .

- Mejor dejar los nombres por defecto.
- Comunes a la mayoría de los algoritmos.

Parámetros del algoritmo

- `smoothing` : parámetro de suavizado.
- `modelType` : tipo de algoritmo Naive Bayes.

smoothing

- Parámetro de suavizado
  - aditivo o estimación bayesiana.
- Determina el numero de ejemplos virtuales por cada valor de un atributo dada la clase.
- Por defecto, 1: estimador de Laplace.

- Importante en conjuntos de datos pequeños, especialmente para evitar estimaciones nulas si no hay ocurrencias de un valor de un atributo dada la clase.

- Su efecto disminuye asintóticamente con el tamaño del conjunto de datos.

modelType

- Selecciona el tipo de algoritmo Naive Bayes.
- “multinomial” , “bernoulli”, “complement”, “gausian”.

- “multinomial” , “bernoulli” y “complement” se utilizan para clasificación de texto.

- “gausian” requiere atributos continuos.

- Por defecto, “multinomial”.

Ejemplo

```
/* Creamos una instancia de NaiveBayes  */
val NB=new NaiveBayes()

/* Elegimos parámetros del modelo        */
val modelType = "bernoulli"
val smoothing =1
NB.setModelType(modelType)
NB.setSmoothing(smoothing)

val NBmodel = new NaiveBayes().fit(trainingData)
```

- Siempre que `trainingData` sea un DF con columnas `features` and `labels`
- Y que los valores de los elementos del vector de características sean `0.0/1.0` (modelo `Bernoulli`)

3. Ejemplo: datos SMSSpam

- https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection
- Almeida, Hidalgo . SMS Spam Collection v.1, 2011.
- Tarea de clasificación de documentos.
- Clase: determinar si un mensaje SMS es *spam.*
- Posibles valores: *spam*, *ham.*

- Conjunto de datos: 5574 SMS etiquetados.

- 774 mensajes de *spam*:

"spam Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030"

- 4827 mensajes *ham*

"ham Nah I don't think he goes to usf, he lives around here though"

Distribución de clases

- Distribución de clases

Class	N	N[%]
spam	774	(13,9 %)
ham	4827	(86,6 %)

Leemos los datos: definimos la estructura del Data Frame

```
/* Directorio con los datos y conjunto de datos */
val PATH = "/home/calonso/Datos/smsspamcollection/"
val DATA = "SMSSpamCollection"


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

// Defino el schema del DataFrame: clase y mensaje
val miSchema = StructType(Array(
 StructField("clase", StringType, true),
 StructField("mensaje", StringType, true)))
```

Leemos los datos al Data Frame `mensajesDF`

```
/* Leo los datos: tienen ham/spam \t texto */
/* Lo puedo leer como un csv con "\t" como separador */

val mensajesDF =
spark.read.format("csv").option("delimiter","\t").schema(miS
chema).load(PATH + DATA)
mensajesDF.show(10)
```

```
+-----+--------------------+
|clase|             mensaje|
+-----+--------------------+
|  ham|Go until jurong p...|
|  ham|Ok lar... Joking ...|
| spam|Free entry in 2 a...|
|  ham|U dun say so earl...|
|  ham|Nah I don't think...|
| spam|FreeMsg Hey there...|
|  ham|Even my brother i...|
|  ham|As per your reque...|
| spam|WINNER!! As a val...|
| spam|Had your mobile 1...|
+-----+--------------------+
only showing top 10 rows
```

Eliminamos filas vacías

```
mensajesDF.first
res45: org.apache.spark.sql.Row = [ham,Go until jurong
point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat...]


/* No tiene valores vacíos o nulos, pero por si
acaso... */
val nonEmpty = mensajesDF.na.drop

nonEmpty.first
res46: org.apache.spark.sql.Row = [ham,Go until jurong
point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat...]
```

Creación de una partición estratificada para Entrenamiento/Prueba

- Separar los datos en Spam y Ham, añadiendo la etiqueta de clase.
- Crear particiones de entrenamiento/prueba sobre Spam y Ham, con el mismo porcentaje.
- Unir datos de prueba y entrenamiento de ambas clases.

Separar los datos en Spam y Ham, añadiendo la etiqueta de clase

```
// Separamos los datos en Spam y Ham
//  Asignamos índices de clase (ham --> 0)
//  y renombramos columna a label

val mensajesHam = nonEmpty.filter($"clase" === 
"ham").withColumn("clase", 
lit(0.0)).withColumnRenamed("clase", "label")

val mensajesSpam = nonEmpty.filter($"clase" === 
"spam").withColumn("clase", 
lit(1.0)).withColumnRenamed("clase", "label")



mensajesHam.show(5)
mensajesSpam.show(5)
```

Los examinamos

```
mensajesHam.show(5)
+-----+--------------------+
|label|             mensaje|
+-----+--------------------+
|  0.0|Go until jurong p...|
|  0.0|Ok lar... Joking ...|
|  0.0|U dun say so earl...|
|  0.0|Nah I don't think...|
|  0.0|Even my brother i...|
+-----+--------------------+
only showing top 5 rows
mensajesSpam.show(5)
+-----+--------------------+
|label|             mensaje|
+-----+--------------------+
|  1.0|Free entry in 2 a...|
|  1.0|FreeMsg Hey there...|
|  1.0|WINNER!! As a val...|
|  1.0|Had your mobile 1...|
|  1.0|SIX chances to wi...|
+-----+--------------------+
only showing top 5 rows
```

Ejercicio 1

- Crear particiones de entrenamiento/prueba de Spam y Ham, con el mismo porcentaje.
- Unir datos de prueba y entrenamiento de ambas clases.

Solución 1

- Crear particiones de entrenamiento/prueba de Spam y Ham, con el mismo porcentaje.
- Unir datos de prueba y entrenamiento de ambas clases.

Los examinamos

```
trainingData.show(5)
+-----+--------------------+
|label|             mensaje|
+-----+--------------------+
|  0.0| &lt;#&gt;  in mc...|
|  0.0| &lt;#&gt;  mins ...|
|  0.0| &lt;DECIMAL&gt; ...|
|  0.0| and  picking the...|
|  0.0| gonna let me kno...|
+-----+--------------------+
only showing top 5 rows
testData.show(5)
+-----+--------------------+
|label|             mensaje|
+-----+--------------------+
|  1.0|Free entry in 2 a...|
|  1.0|FreeMsg Hey there...|
|  1.0|WINNER!! As a val...|
|  1.0|Had your mobile 1...|
|  1.0|SIX chances to wi...|
+-----+--------------------+
only showing top 5 rows
```

Procesamos el conjunto de entrenamiento

- Metodológicamente, más correcto
  - Para que los `estimator` no utilicen información del conjunto de prueba.
- Aunque el modelo final se cree con “todos” los datos disponibles.

- Extraer palabras: Tokenizer

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

- Tokenizer es un transformer.
- Al aplicarlo, crea una nueva columna, extrayendo las palabras de un `String,` convirtiendo a minúsculas.

- Crear vector de características: depende del modelo de texto.

Separamos los mensajes en palabras, creando Data Frame `trainingPalabras`

```
/* Extraemos las palabras con Tokenizer                    */
import org.apache.spark.ml.feature.Tokenizer

val tokenizer = new
Tokenizer().setInputCol("mensaje").setOutputCol("palabras")

val trainingPalabras = tokenizer.transform(trainingData)
trainingPalabras.show(19)
+-----+--------------------+--------------------+
|label|             mensaje|            palabras|
+-----+--------------------+--------------------+
|  0.0| &lt;#&gt;  in mc...|[, &lt;#&gt;, , i...|
|  0.0| &lt;#&gt;  mins ...|[, &lt;#&gt;, , m...|
|  0.0| &lt;DECIMAL&gt; ...|[, &lt;decimal&gt...|
|  0.0| and  picking the...|[, and, , picking...|
|  0.0| gonna let me kno...|[, gonna, let, me...|
|  0.0| said kiss, kiss,...|[, said, kiss,, k...|
|  0.0| says that he's q...|[, says, that, he...|
|  0.0|"Gimme a few" was...|["gimme, a, few",...|
|  0.0|"Happy valentines...|["happy, valentin...|
+-----+--------------------+--------------------+
only showing top 9 rows
```

4. Modelo de texto

- *Bag of words.*
- *Multinomial.*
- Vector de características: número de ocurrencias de cada palabra del vocabulario en el documento.

- Dificultades

1. Hay que definir un vocabulario.
2. Vector de características de elevada dimensionalidad.

- Soluciones

1. Técnicas *feature hashing,* para evitar asociar explícitamente cada palabra con un índice.
2. Vectores *sparse*: pues la mayoría de los documentos incluyen pocas palabras del vocabulario.

Feature Hashing I

- Técnica para manejar datos categóricos y de texto cuando las características pueden tomar muchos valores únicos (miles millones…).

- Hasta ahora, hemos utilizado la codificación mediante StringIndexer (*1 de k* no es viable para tantos valores)
  - Luego habría que hacer un *broadcast* (implícito o explicito) del *mapping* a todos los *workers.*

- Este método no escala bien si las características tienen muchos valores.
- Alternativa: utilizar una función *hash* que a cada valor le asigne un índice.

Feature Hashing II

- Utilizar una función de hash para asociar un índice a cada valor de una característica.
- Particularmente útil en el caso de texto.
- Permite transformar un *String* , o una secuencia de palabras, en un vector con ocurrencias (o cuenta de ocurrencias, según implementación).

Feature Hashing III

- Ventajas:
- No es necesario definir un vocabulario
  - Solo precisamos que a cada palabra le asigne siempre el mismo índice.
- No es necesario construir de forma explícita el *mapping* valores-índices (StringIndexer).
- No es preciso hacer un *broadcast*: cada *worker* lo construye.

- Pero*:*
- No podemos invertir el *mapping* y obtener el valor correspondiente a un índice.
- Colisiones (si bien no parece ser crítico).

HashingTF

- *Mllib y ML* proporciona la clase `HashingTF`.

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

- Su constructor tiene un parámetro: `numFeatures: Int`
  - Por defecto: 2^{20}.

- Creación de una instancia limitando el vocabulario a 100 palabras distintas (palabras adicionales: colisiones):

```
/* Creamos vector de caraterísticas con HashingTF */
import org.apache.spark.ml.feature.HashingTF


val numFeatures = 100
val hashingTF = new
HashingTF().setInputCol("palabras").setOutputCol(“features")
.setNumFeatures(numFeatures)
```

Aplicamos y examinamos

- Al aplicarlo crea una columna con la cuenta de ocurrencias en un vector disperso.

```
val trainingFeaturized = hashingTF.transform(trainingPalabras)
trainingFeaturized.show(6)
```

```
+-----+--------------------+--------------------+--------------------+
|label|             mensaje|            palabras|            features|
+-----+--------------------+--------------------+--------------------+
|  0.0| &lt;#&gt;  in mc...|[, &lt;#&gt;, , i...|(100,[5,36,63,72,...|
|  0.0| &lt;#&gt;  mins ...|[, &lt;#&gt;, , m...|(100,[5,56,69,72,...|
|  0.0| &lt;DECIMAL&gt; ...|[, &lt;decimal&gt...|(100,[3,5,10,14,2...|
|  0.0| and  picking the...|[, and, , picking...|(100,[21,24,39,42...|
|  0.0| gonna let me kno...|[, gonna, let, me...|(100,[2,3,9,21,42...|
|  0.0| said kiss, kiss,...|[, said, kiss,, k...|(100,[5,9,10,13,1...|
+-----+--------------------+--------------------+--------------------+
only showing top 6 rows
```

Finalmente creamos Data Frame `trainingDF` con columnas `"label", "features"`

```
/* Eliminamos columnas "mensaje", "palabras"   */

val trainingDF = trainingFeaturized.drop("mensaje",
"palabras")
trainingDF.show(6)
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(100,[10,13,14,17...|
|  0.0|(100,[27,30,45,70...|
|  1.0|(100,[2,9,12,17,2...|
|  0.0|(100,[1,3,19,45,6...|
|  0.0|(100,[13,15,35,40...|
|  1.0|(100,[0,4,6,11,14...|
+-----+--------------------+
only showing top 6 rows
```

5. Inducción de un filtro anti spam con NB multinomial

- Crearemos un filtro anti spam básico, basado en el modelo de documento "*Bag of words*" , con una representación multinomial, sin ningún preprocesamiento adicional, salvo separar las palabras
  - Ver https://www.kdnuggets.com/2017/02/natural-language-processing-key-terms-explained.html para conocer los elementos de preprocesado de texto más comunes.

- Ya tenemos el conjunto de entrenamiento con la representación deseada: ejemplos etiquetados descritos con el modelo multiniomial de cuenta de palabras, para un vocabulario de 100 palabras.
- En un Data Frame con columnas `"label", "features".`

- A continuación, inducir el clasificador con el conjunto de entrenamiento.
- Procesar el conjunto de prueba.
- Evaluar el clasificador sobre el conjunto de prueba.

Ejercicio 2: inducir un clasificador NB Multinomial

- Entrenar un clasificador Naive Bayes Multinomial utilizando el estimador de Laplace para la estimación de los parámetros del modelo.

Solución 2: inducir un clasificador NB multinomial

- Entrenar un clasificador Naive Bayes Multinomial utilizando el estimador de Laplace para la estimación de los parámetros del modelo.

Ejercicio 3: Procesar los datos de prueba

- Obtener el Data Frame `testDF` a partir del Data Frame `testData.`
- Aplicando las mismas etapas que al conjunto de entrenamiento.

Solución 3: Procesar los datos de prueba

- Obtener el Data Frame `testDF` a partir del Data Frame testData.
- Aplicando las mismas etapas que el conjunto de entrenamiento.

Solución 3: lo examinamos

```
testDF.show(6)
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(100,[1,6,9,10,17...|
|  0.0|(100,[9,19,26,30,...|
|  0.0|(100,[5,9,10,15,1...|
|  0.0|(100,[4,5,6,17,20...|
|  0.0|(100,[3,5,6,8,9,1...|
|  0.0|(100,[12,14,17,20...|
+-----+--------------------+
only showing top 6 rows
```

Ejercicio 4: estimar la tasa de error sin utilizar las clases `metrics`

Solución 4: estimar la tasa de error sin utilizar las clases `metrics`

Solución 4: estimar la tasa de error sin utilizar las clases `metrics`

```
Tasa de error del clasificador = 0,059
```

6. Filtro Anti Spam con `Pipeline`

Ejercicio 5: ahora con un `Pipeline`

- Repetir al ejercicio definiendo las etapas de procesamiento y creación del modelo con un `Pipeline.`
- Aplicar el `pipelineModel` al conjunto de prueba.
- Obtener la tasa de error.
- Atención: Utilizar la misma partición estratificada de los datos para crear los DF `trainingData` y `testData.` No incluir esta etapa en el pipeline.

Solución 5:

7. Referencias

- 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: septiembre 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: septiembre 2025.
- Kdnuggets: Natural Language Processing Key Terms, Explained. https://www.kdnuggets.com/2017/02/natural-language-processing-key-terms-explained.html . Ultimo acceso: septiembre 2025.
- Nick Pentreath. Machine Learning with Spark. Packt Publishing. 2015. ISBN: 9781783288519. http://www.packtpub.com/
- Naive Bayes. https://spark.apache.org/docs/3.5.6/ml-classification-regression.html#naive-bayes. Último acceso: septiembre 2025.
- Extracting, transforming and selecting features. https://spark.apache.org/docs/3.5.6/ml-features.html#extracting-transforming-and-selecting-features. Último acceso: septiembre 2025.

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

Curvas ROC y otras métricas de evaluación de clasificadores

Contenido

1. Motivación: Limitaciones de la tasa de error.
2. Costes de clasificación.
3. Estructura del error: Matriz de confusión.
4. Clasificación con coste.
5. Curvas ROC.
6. Otras métricas.
7. Referencias.

1. Motivación: Limitaciones de la tasa de error

- Hasta ahora, hemos utilizado la tasa de error como criterio para evaluar hipótesis.
- Suposiciones:
  - Distribuciones de clases no muy desequilibradas.
  - Los costes de los errores son los mismos.
- No siempre es así
  - 99,99% de la población no es terrorista.
    - «No terrorista» cierto 99,99%.
  - El 97% de los días del mes las vacas no están en celo
    - «No celo» cierto 97,00%.
- ¿Coste de diagnosticar erróneamente la presencia de una enfermedad frente a no diagnosticar una enfermedad real?

2. Costes de clasificación

- Optimizar la tasa de error sin tener en cuenta el coste de cometer un error de clasificación puede producir resultados no deseados.
- En la práctica, diferentes tipos de error de clasificación suelen tener costes diferentes.
- Interesa conocer la estructura del error para buscar una buena solución de compromiso.
- Otros tipos de costes que no estamos considerando, pero que son importantes en un proyecto de Big Data
  - Coste de obtención datos.
  - Coste aplicación hipótesis.

3. Estructura del error: matriz de confusión

- Problema binario:

		Clase predicha		
		sí	No	
Clase real	sí	Ciertos Positivos (TP)	Falsos Negativos (FN)	P=TP+FN
	no	Falsos Positivos (FP)	Ciertos Negativos (TN)	N=FP+TN

- Tasa de acierto: $[TP + TN]/[P + N]$
- Tasa de ciertos positivos: $tp = \frac{TP}{P} = \frac{TP}{TP+FN}$
- Tasa de falsos positivos: $fp = \frac{FP}{N} = \frac{FP}{FP+TN}$

Matriz de confusión problema multiclase

		Predicted class			
		a	*b*	*c*	*total*
Actual class	*a*	88	10	2	*100*
	b	14	40	6	*60*
	c	18	10	12	*40*
	total	*120*	*60*	*20*	

- Tasas de acierto $= \frac{88+40+12}{88+10+2+\cdots 12} = 0.7$
- Tasa de ciertos positivos por clase
  $TP_a = \frac{88}{88+10+2} = 0{,}88$
- Tasa de falsos positivos por clase
  $FP_a = \frac{14+18}{14+40+6+18+10+12} = 0{,}32$

- Tasas de acierto $= \sum_{i=1}^{k} x_{i,i} \Big/ \sum_{i=1,j=1}^{k} x_{i,j}$
- Tasa de ciertos positivos por clase $TP_i = x_{i,i} / \sum_{j=1}^{k} x_{i,j}$
- Tasa de falsos positivos por clase $FP_i = \frac{\sum_{j=1, j\neq i}^{k} x_{j,i}}{\sum_{j,l, j\neq i}^{k} x_{j,l}}$

- **No utilizar métricas promediadas**: enmascaran el mal comportamiento en las clases de interés (que suelen tener pocos ejemplos).

4. Clasificación con coste

- Si se conoce el coste de los errores, se puede incluir en el proceso de toma de decisión.
- Matriz de coste

		Clase predicha	
		a	b
Clase actual	a	0	1
	b	5	0

		Clase predicha		
		a	b	c
Clase actual	a	0	1	4
	b	1	0	8
	c	8	16	0

- Clasificación: remplazar la tasa de error por el coste promedio por predicción y asignar la clase con menor coste.

- **PERO** los costes de los errores rara vez se conocen con precisión.
- Alternativa: analizar distintos escenarios (curvas ROC, PR, etc…).

5. Curvas ROC

- Análisis ROC: *Receiver Operation Characteristic.*
- Origen en Teoría de la señal, para mostrar el compromiso entre la tasa de detección (tp) y la tasa de falsas alarmas (fp) sobre canales con ruido.
- Muy utilizados en diagnosis médica.
- Especialmente interesante si costes muy diferentes o distribución de clases muy desigual (*skewed*).
- Especialmente útil en problemas binarios.

Espacio ROC

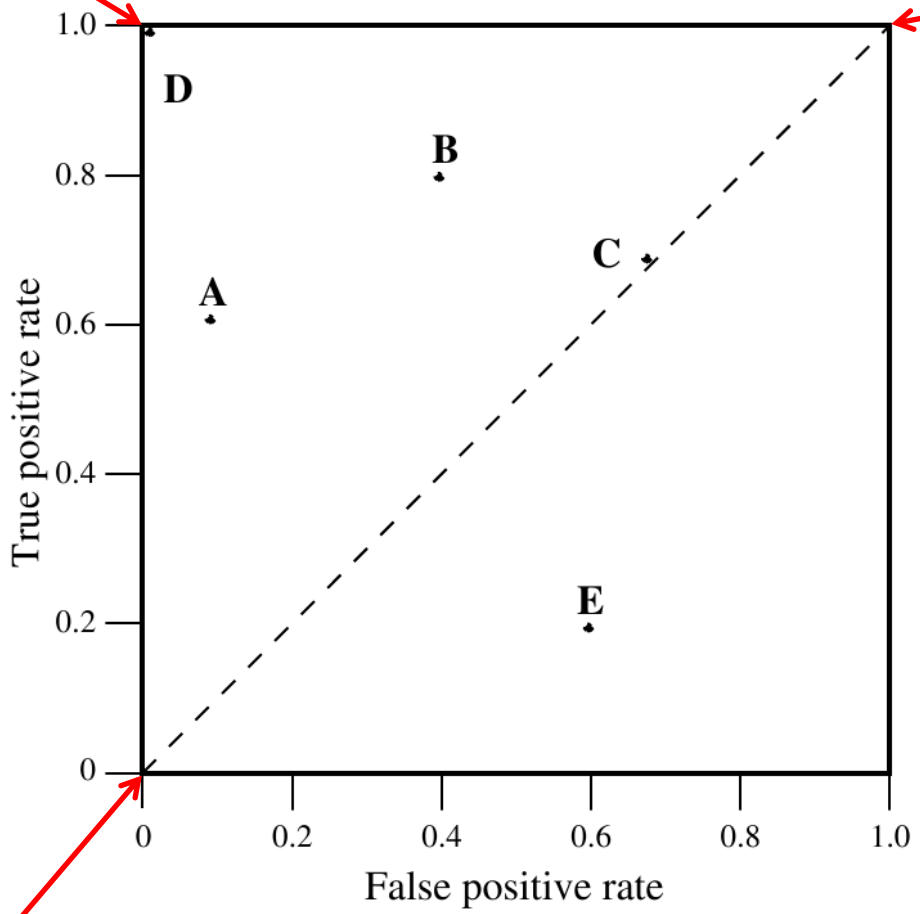
- Gráficos bidimensionales.
- Eje Y: tasa de ciertos positivos.
- Eje X: tasa de falsos positivos.
- Punto en el espacio: pares (fp, tp).
  - Tasa de ciertos positivos: $tp = TP/P$.
  - Tasa de falsos positivos: $fp = FP/N$.
- Relación entre los beneficios (ciertos positivos) y los costes (falsos positivos).

- Cada clasificador binario: un punto en el espacio ROC.
- $(0,0)$ clasificador que siempre predice la clase negativa.
- $(1, 1)$ clasificador que siempre predice la clase positiva.
- Un clasificador es mejor que otro si está encima y a la izquierda.
- Predicción aleatoria: diagonal.

Puntos característicos

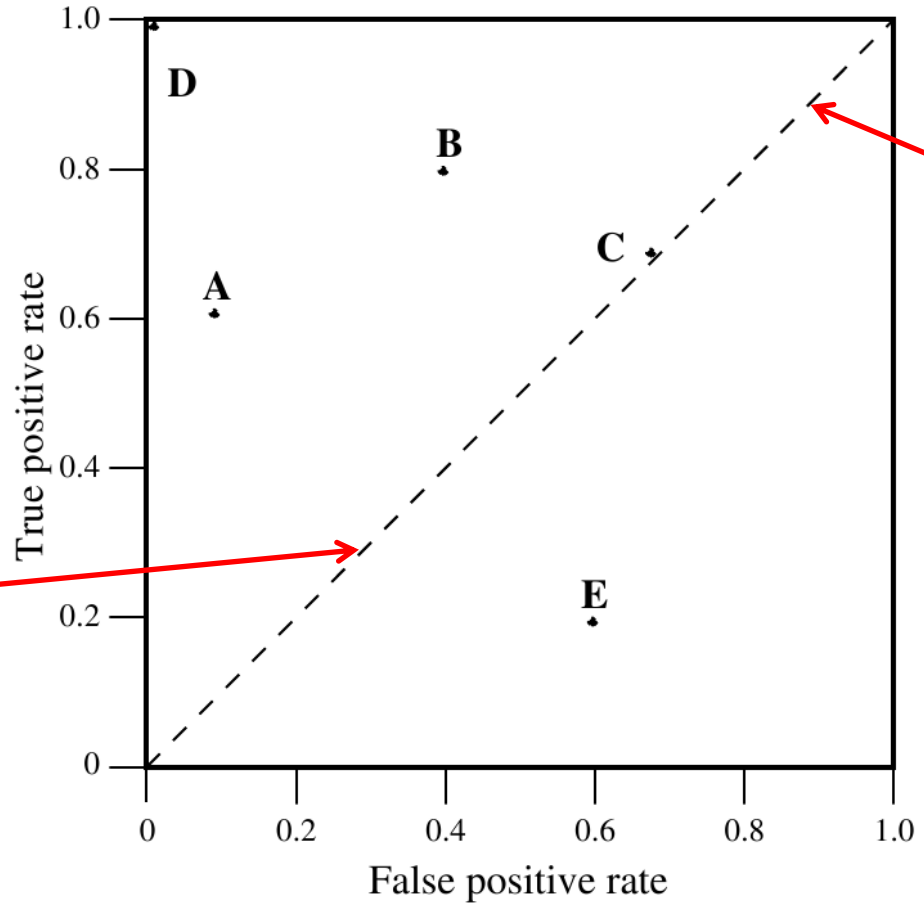
Ideal: $(0, 1)$

Siempre positivo: $(1, 1)$



Siempre negativo: $(0, 0)$

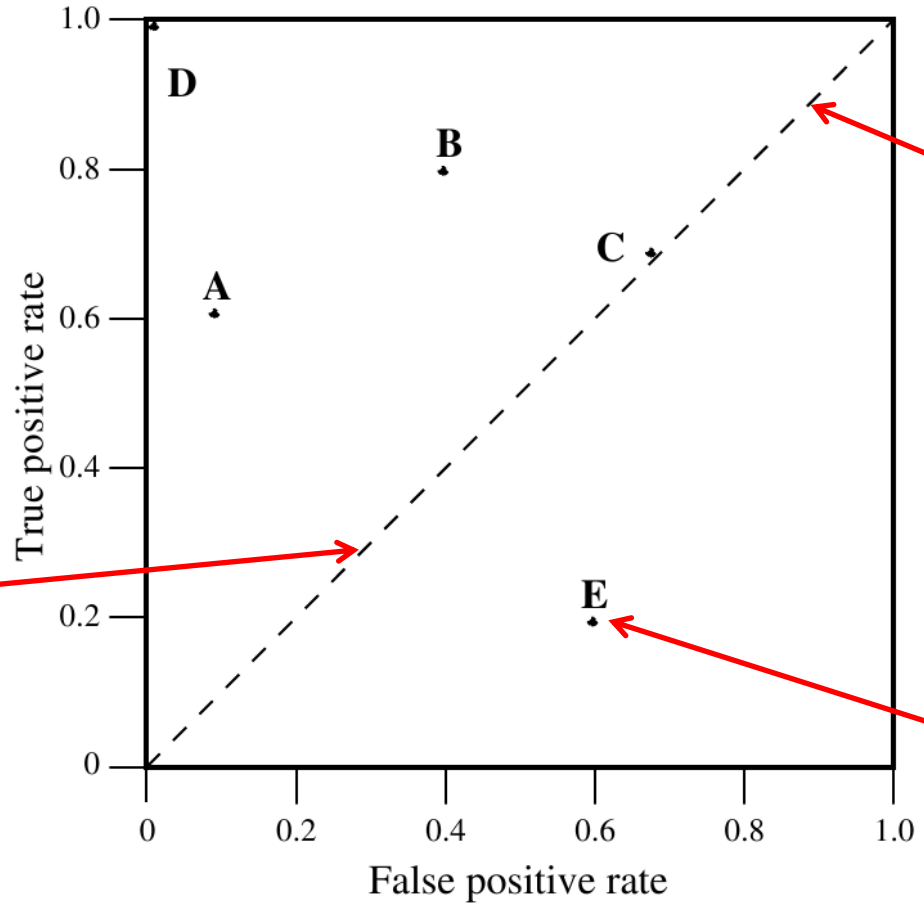
Clasificador aleatorio: diagonal



Aleatorio,
90% positivo
(0.9, 0.9)

Aleatorio,
30% positivo
(0.3, 0.3)

Peor que aleatorio: bajo la diagonal



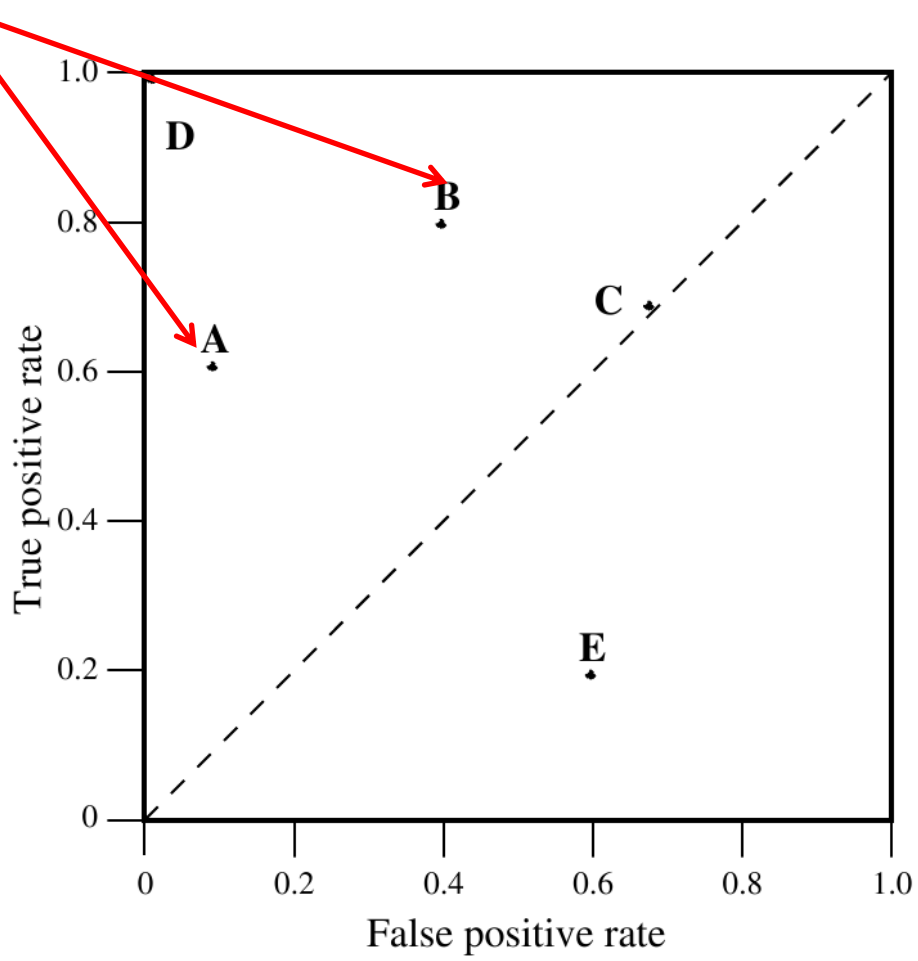
Aleatorio,
90% positivo
(0.9, 0.9)

Aleatorio,
30% positivo
(0.3, 0.3)

Peor que
aleatorio,
(su complementario
es B)

Comparación de clasificadores

A es más conservador que B



Si el coste de los falsos positivos es elevado, preferiríamos A

Si el coste de los falsos positivos es asumible, preferiríamos B

Generación de curvas ROC

- Un clasificador binario produce un punto en el espacio ROC.
- Hay clasificadores que proporcionan algún tipo de "puntuación continua" ($score$): Naïve Bayes, Regresión logística, Redes de Neuronas.
- Utilizando un umbral para decidir la clase, tenemos una familia de clasificadores
  - Clase positiva si $score > umbral$.
- Variando el umbral, generamos una curva.

- Si el clasificador solo devuelve la clase, adaptarlo para que devuelva un $score$.

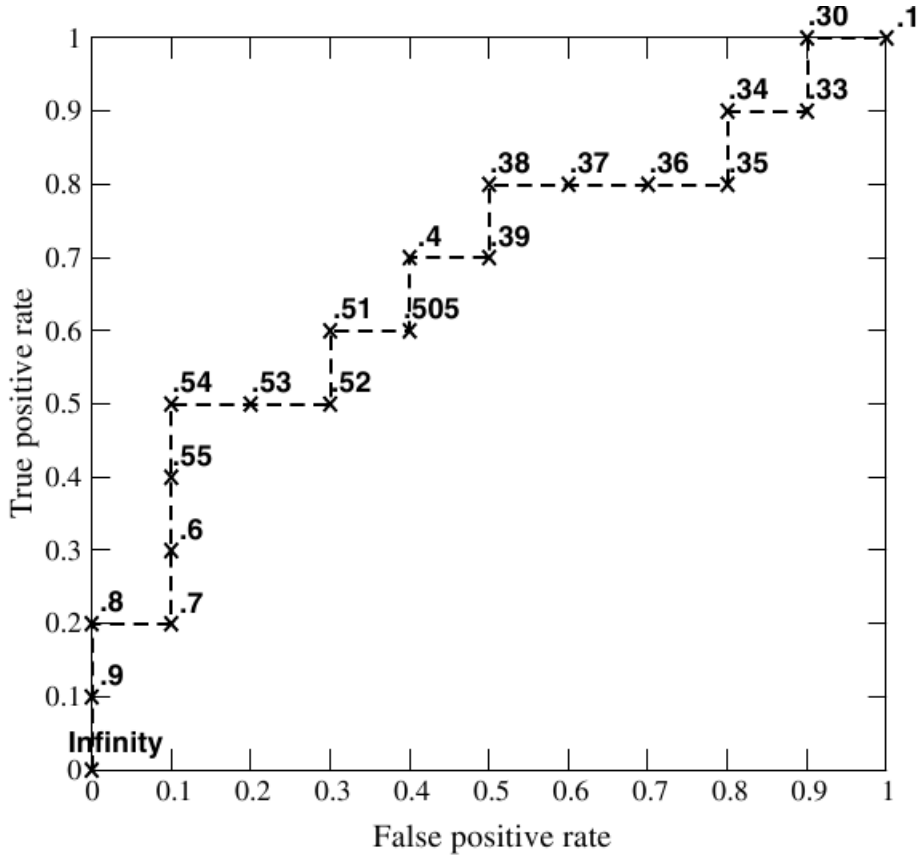
Ejemplo: curva ROC sobre conjunto de Prueba con 20 instancias

- Crear el clasificador.
- Evaluarlo sobre el conjunto de prueba.
- Ordenar los ejemplos de prueba por orden decreciente de $score$.
- Mover el umbral entre $scores$ consecutivos.
- Cada elección de umbral define un clasificador y un punto en el espacio ROC.

Inst#	Class	Score	Inst#	Class	Score
1	p	.9	11	p	.4
2	p	.8	12	n	.39
3	n	.7	13	p	.38
4	p	.6	14	n	.37
5	p	.55	15	n	.36
6	p	.54	16	n	.35
7	n	.53	17	p	.34
8	n	.52	18	n	.33
9	p	.51	19	p	.30
10	n	.505	20	n	.1

Curva ROC sobre un conjunto de prueba

Inst#	Class	Score	Inst#	Class	Score
1	p	.9	11	p	.4
2	p	.8	12	n	.39
3	n	.7	13	p	.38
4	p	.6	14	n	.37
5	p	.55	15	n	.36
6	p	.54	16	n	.35
7	n	.53	17	p	.34
8	n	.52	18	n	.33
9	p	.51	19	p	.30
10	n	.505	20	n	.1

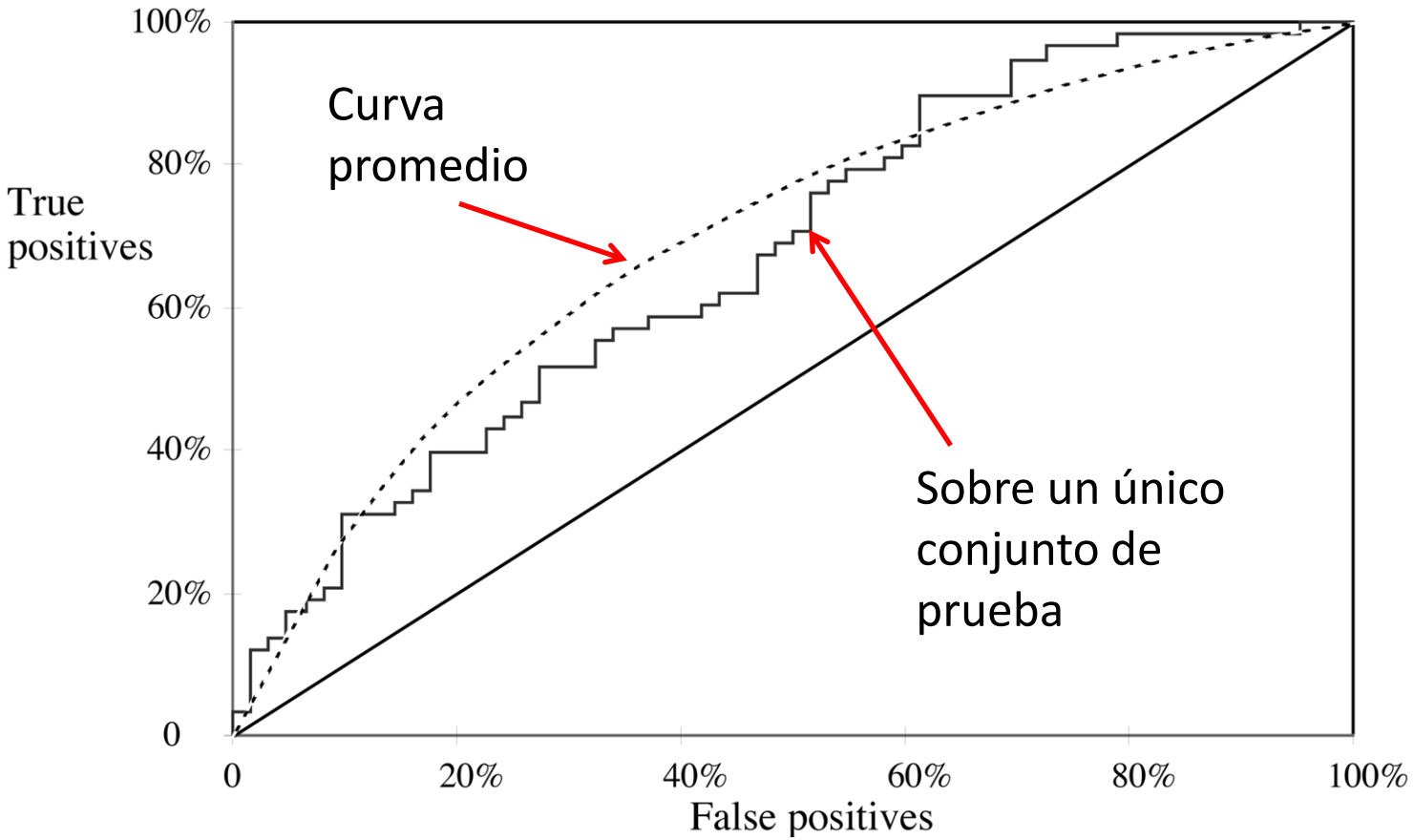


Generación de curvas ROC

- La forma de la curva ROC depende del conjunto de prueba utilizado.
- Se puede obtener una curva más suave promediando valores de varios experimentos.

- Promediando sobre tasa tp (promedio vertical).
- Mediante validación cruzada.
- En cada partición
  - Obtener tasa tp para cada valor fp.
- Obtener tasa tp media para cada valor fp.

Ejemplo curva promedio

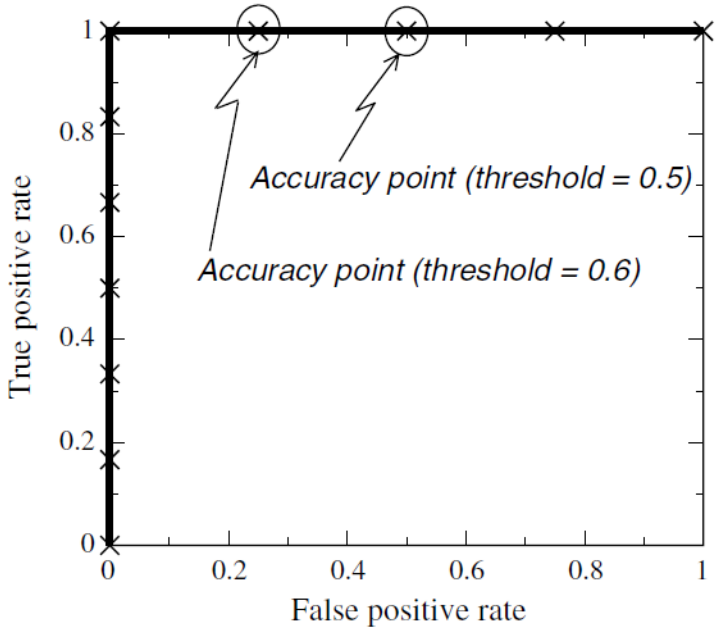


Propiedades de las curvas ROC

- Las curvas ROC miden la capacidad del clasificador para producir buenas puntuaciones relativas.

- Las curvas ROC son invariantes ante cambios en la distribución de clases (en el conjunto de prueba).

Scores relativos



Inst no.	Class True	Class Hyp	Score
1	p	Y	0.99999
2	p	Y	0.99999
3	p	Y	0.99993
4	p	Y	0.99986
5	p	Y	0.99964
6	p	Y	0.99955
7	n	Y	0.68139
8	n	Y	0.50961
9	n	N	0.48880
10	n	N	0.44951

- Tasa de error 0.2 (umbral 0.5, por defecto) pero curva ROC ideal
- Las curvas ROC mide la capacidad del clasificador para ordenar las instancias positivas frente a las negativas.

Propiedades de las curvas ROC

- Las curvas ROC son invariantes ante cambios en la distribución de clases (en el conjunto de prueba)
  - Si la proporción de ejemplos positivos/negativos cambia en el conjunto de prueba, la curva ROC no cambia.
  - Motivo: las tasas tp (y fp) son ratios respecto a los positivos (negativos) totales, P (N)
    - cada ratio se calcula con elementos de la misma fila.

		Clase predicha	
		sí	no
Clase real	sí	TP	FN
	no	FP	TN

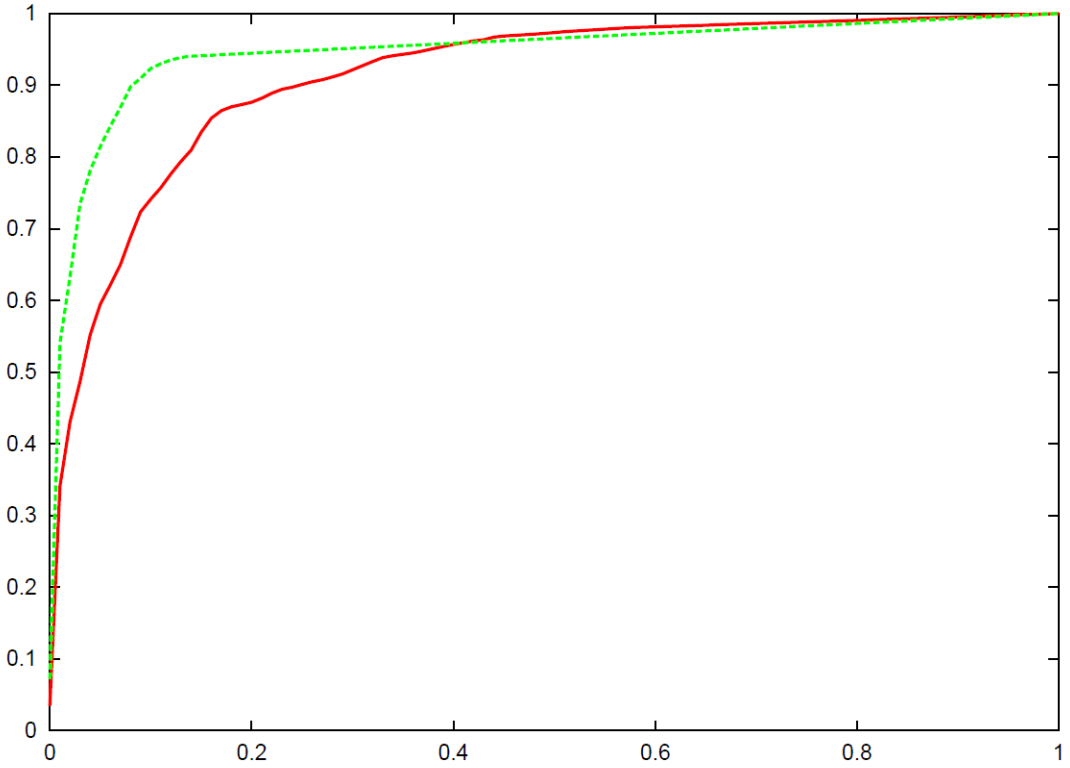
$$tp = \frac{TP}{TP + FN}$$

$$fp = \frac{FP}{FP + TN}$$

- Frente a otras métricas, que mezclan elementos de ambas filas.

Curvas ROC para dos clasificadores

- Punto de trabajo: numero de fp.
- En cada punto de trabajo, es preferible el clasificador con tasa tp más alta.
- Rara vez el mismo clasificador se comporta mejor en todos los puntos de trabajo.



Área bajo la curva ROC: AUC

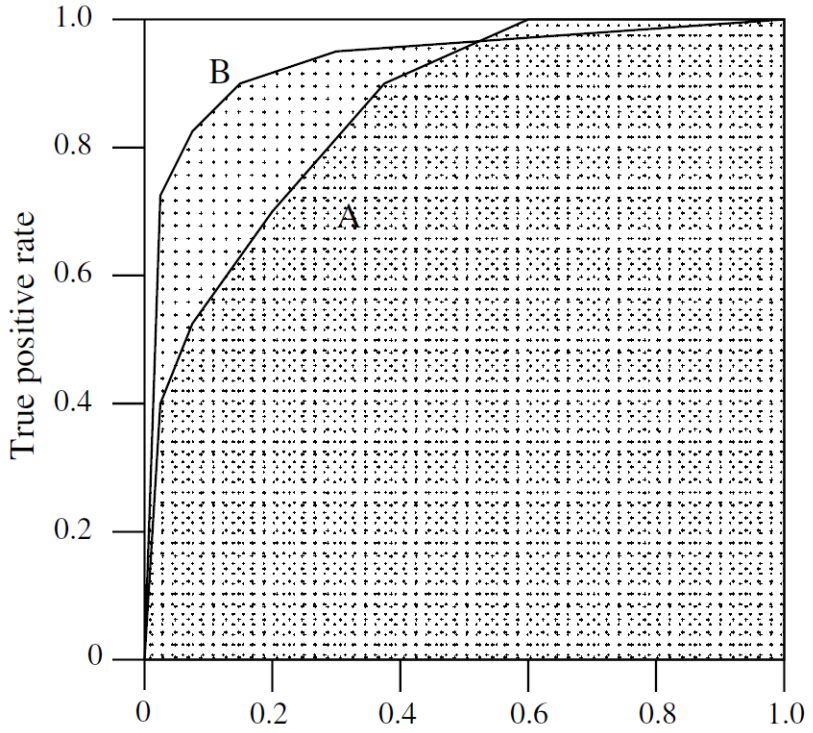
- Único valor numérico que facilita la comparación de dos clasificadores.

- AUC: porción del área bajo la curva en un cuadrado unitario
  - Entre 0 y 1.
  - Debería ser mayor que 0.5
    - Correspondiente a un clasificador aleatorio.

- Preferibles clasificadores con mayor AUC
  - Pero un clasificador con menor AUC puede comportase mejor en alguna región.

Ejemplo AUC

- $AUC_B > AUC_A$
- Pero si estamos dispuestos a aceptar el coste de más fp con tal de identificar más positivos, el clasificador A es preferible.



Propiedad AUC

- El área bajo la curva es equivalente a la probabilidad de que el clasificador asigne un score superior a una instancia positiva que a una negativa.

- AUC es un buen indicador de la capacidad de predicción de un clasificador.

Resumen curvas ROC

- Herramienta para visualizar el comportamiento de un clasificador.
- Interesante porque no es sensible a la distribución de clases (en el conjunto de prueba) ni al coste.
- Conocido el coste de los errores, permite seleccionar el punto de trabajo
  - En combinación con un umbral.
- Utilizadas tradicionalmente en comunicaciones (radar) y diagnosis médica.

7. Otras métricas

- Marketing
  - Clientes que responden: TP.
  - Tamaño de la muestra: $(TP + FP)/(TP + FP + TN + FN)$ % clientes a los que se le envía.
  - Gráfico de elevación ($lift\ charts$).
- Recuperación de información
  - ***Precision*** ($TP/(TP + FP)$): % de documentos relevantes entre los que son recuperados.
  - ***Recall*** ($\mathrm{tp} = TP/(TP + FN)$): % de documentos relevantes que son recuperados.
  - Curvas *precision/recall.*
- Diagnóstico médico: resultados de test
  - Sensibilidad (tp): porcentaje con enfermedad y test positivo.
  - Especificidad ($1 - fp$): porcentaje sin enfermedad y test negativo.

precission/recall

- Utilizadas en el ámbito de la recuperación de información.
  - También de utilidad en clasificación si la distribución de valores de la clase es muy desequilibrada.
- Documentos recuperados: equivalente a predicción positiva de un clasificador.

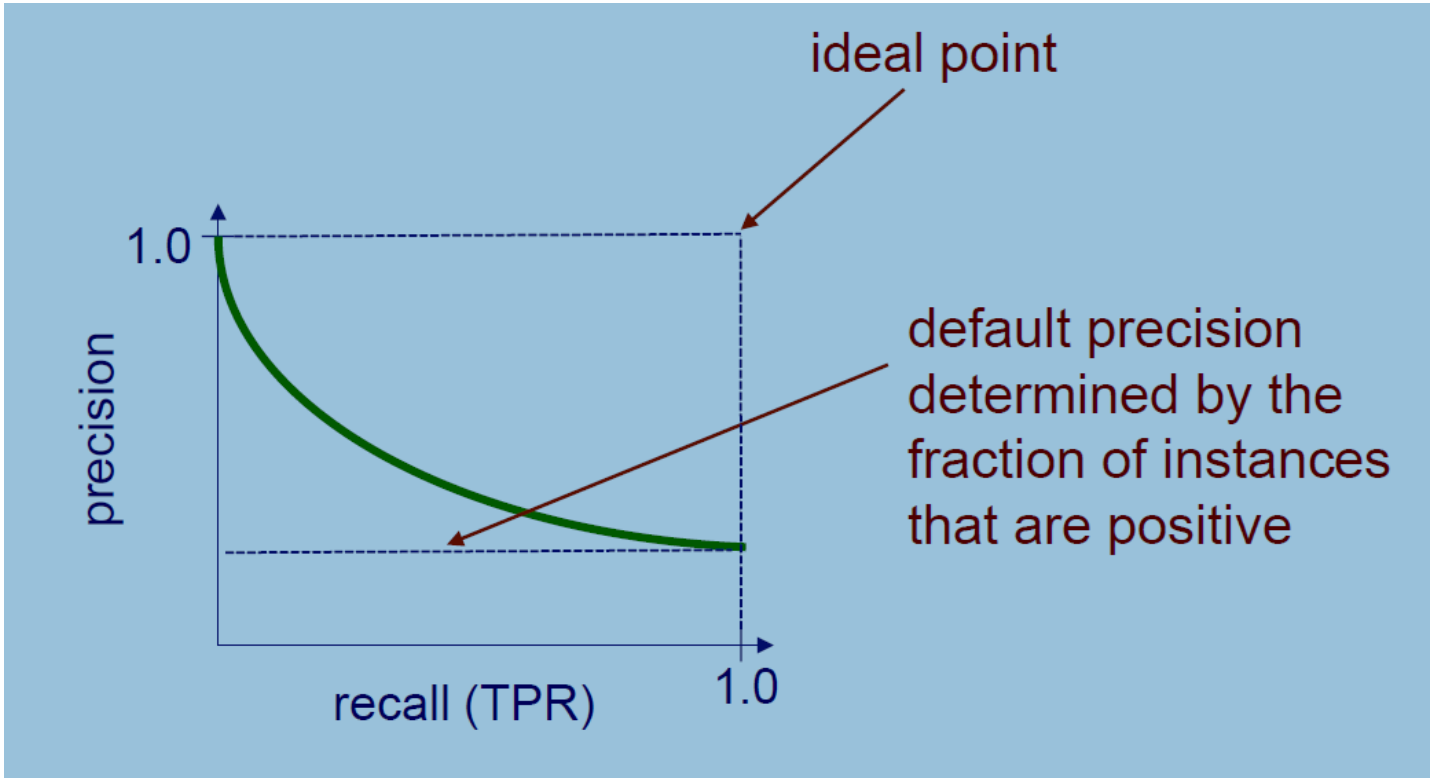
- *Precision* ($TP/(TP+FP)$): % de documentos relevantes entre los que son recuperados
  - *Precision* perfecto: 100, todos los documentos recuperados son relevantes (aunque no estén todos, FN; no hay FP).
- *Recall* (tp $= TP/(TP+FN)$): % de documentos relevantes que son recuperados
  - Recall perfecto: 100, se recuperaron todos los documentos relevantes (y quizás muchos no interesantes también, FP; no hay FN).

Espacio PR: curvas *precision/recall*

- Gráficos bidimensionales.
- Eje Y: *precision.*
- Eje X: *recall.*
- Punto en el espacio: pares $(recall, precision)$
  - *Precision:* $TP/(TP + FP)$: en %.
  - *Recall*: $\text{tp} = TP/(TP + FN)$*:* en %.
- Relación entre la calidad (Ciertos Positivos entre todos los recuperados) y la exhaustividad (*recall:* tasa de recuperados entre todos los relevantes).

- Un clasificador binario produce un punto en el espacio PR.
- Variando el umbral se obtiene una curva.
- La curva PR muestra la fracción de las predicciones que son ciertos positivos.
- (1, 1): clasificador ideal.

Ejemplo curva PR



www.cs.wisc.edu/~dpage/cs760/

Área bajo la curva PR y *F-measure*

- Ambas permiten resumir el comportamiento con una única medida.

- AUC mide la *precision* media de un clasificador.
- AUC=1: clasificador perfecto, con 100% *precision* y *recall.*
- Pero *precision* y *recall* suelen tener una relación inversa.

- *F-measure* es la media harmónica de *precision* y *recall.*
- $F\text{-}measure = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} = \frac{2 \times TP}{2 \times TP + FP + FN}$
- Varía entre 0 y 1.
- Utilizada en recuperación de información.
- Discutible en clasificación, pues no considera TN.
  - Pero de utilizad en localización en imágenes (hay muchos TN).

8. Referencias

- Ian H. Witten, Eibe Frank, Mark A. Hall, Christopher Pal. Data Mining: practical machine learning tools and techniques (4th Edition). Morgan Kaufmann, 2016. ISBN: 9780128042915
- Tom Fawcett. An introduction to ROC analysis. Pattern Recognition Letters 27, 861-874, 2006.
- Davis, J. and Goadrich, M. The relationship between Precision-Recall and ROC curves. ICML '06 Proceedings of the 23rd international conference on Machine learning, pp 233-240, Pittsburgh, PA, 2006.
- Thomas Kautz, Bjoern M. EskoÞer, Cristian F. Pasluosta. Generic Performance Measure for Multiclass-Classifiers. Pattern Recognition , 8, 111-125, 2017. doi: 10.1016/j.patcog.2017.03.008
- David Page. Evaluating Machine Learning Methods. http://pages.cs.wisc.edu/~dpage/cs760/evaluating.pdf. Último acceso: octubre 2025.

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

Metodología experimental de evaluación y selección de modelos

Metodología experimental: creación y evaluación de clasificadores mediante la tasa de error

Contenido

1. Motivación.
2. Error.
3. Estimación del error.
4. Metodología experimental para estimar la tasa de error de una hipótesis.
5. Evaluación y selección de modelos.
6. Sobre la tasa de error.
7. Referencias.

1. Motivación

- Nos limitamos al ámbito de los clasificadores
  - Aprendizaje inductivo basado en ejemplos, supervisado, predicción valor categórico.
- Recordar: cada modelo inducido es una posible hipótesis.

- ¿Cuál es la calidad de una hipótesis?
  - Múltiples criterios
    - Conocimiento aprendido: calidad, novedad, utilidad…
    - Capacidad para describir ejemplos no utilizados para entrenar: calidad como clasificador, coste aplicación,…

- Primera aproximación: tasa de error.

- ¿Cómo utilizar los datos para crear la hipótesis y al mismo tiempo evaluarla?

2. Error

- Medida natural del rendimiento de un clasificador.
- Acierto: el clasificador predice correctamente la clase.
- Error: el clasificador predice incorrectamente la clase.
- Tasa de error: proporción del número de errores cometidos sobre un conjunto de instancias.

Error verdadero, $e_D(h)$

- Def. El error verdadero, $e_D(h)$, de una hipótesis h respecto a un concepto objetivo c y distribución de probabilidad del espacio de instancias D es:

$$e_D(h) = Pr_{x \in D}[c(x) \neq h(x)]$$

Tasa de error, $e_S(h)$

- Def. La tasa de error, $e_S(h)$, de una hipótesis h respecto a un concepto objetivo c y una muestra $S \subset X$ es:

$$e_S(h) = 1/card(S) \times \sum_{x \in S} \delta(c(x), h(x))$$

$\delta(c(x), h(x)) = 1$ *sii* $c(x) \neq h(x)$
$\delta(c(x), h(x)) = 0$ *sii* $c(x) = h(x)$

Error de resubstitución, e_r

- Tasa de error calculada sobre el conjunto de entrenamiento.

- Muy optimista.
- Estima tasas de error menores que el error verdadero.

Error verdadero y tasa de error

- La tasa de error es una estimación del error verdadero.

- ¿En qué circunstancias es una buena estimación?
- No basta con $card(S)$ "suficientemente grande".

- Cuando **la hipótesis h y el conjunto S** con el que se obtiene la tasa de error **son independientes**.

- El conjunto S debe seleccionarse con independencia de la hipótesis y sus elementos no deben utilizarse para la creación del clasificador de ninguna manera.

3. Estimación del error

- Proceso de Bernoulli y distribución binomial.
- Ejemplo: ¿probabilidad de obtener 3 caras lanzando una moneda 5 veces al aire?

- Ensayo de Bernoulli o Experimento base: lanzar moneda al aire.
  - La variable aleatoria solo toma uno de dos valores (distribución binomial)
- Proceso de Bernoulli: repetición del experimento base
  - Sucesos independientes, probabilidad constante.

- Problema: determinar la probabilidad del experimento base conocido el resultado de un Proceso de Bernoulli.

Tasa de error: Proceso de Bernoulli

- El problema de determinar el error verdadero, $e_D(h)$, conocida la tasa de error, $e_S(h)$, se puede plantear como un Proceso de Bernoulli:
  - Experimento base: determinar el error verdadero de h sobre una instancia cualquiera.
    - La variable aleatoria solo toma uno de dos valores.
  - Proceso de Bernoulli: repetición sobre los elementos de S
    - Sucesos independientes, probabilidad constante.

- Resultado del Proceso de Bernoulli: Errores cometidos sobre S (a partir del cual obtenemos $e_S(h)$)
- Probabilidad de error al realizar experimento base: error verdadero, $e_D(h)$

- Siempre que los sucesos sean **independientes**: h independiente de S

Estimación de la tasa de error con suficientes ejemplos

- Si S contiene n ejemplos seleccionados de forma aleatoria según probabilidad D, no utilizados de ninguna manera para la creación de la hipótesis h que clasifica mal r ejemplos de S.
- Si $n >= 30$ $(np(1-p) >= 5)$, aproximamos binomial por normal.

1. Tasa de error: $e_S(h) = r/n$
2. Error verdadero: $e_D(h)$
3. $E[e_D(h)] = E[e_S(h)]$
4. Desviación estándar $\sigma_{e_s(h)} \approx \left(\frac{e_{S(h)}(1-e_{S(h)})}{n}\right)^{1/2}$
5. Con probabilidad $N\%$, $e_D(h)$ está en el intervalo

$$e_{S(h)} \pm z_N \times \left(\frac{e_S(h)(1-e_S(h))}{n}\right)^{1/2}$$

   - Donde z_N lo obténtenos de las tablas (de dos colas) de la distribución Normal para la confianza deseada.

4. Metodología experimental para estimar la tasa de error de una hipótesis

Suficientes datos

- T: conjunto de entrenamiento.
  - Con el que se crea la hipótesis.
- V: conjunto de validación.
  - Para ajustar la hipótesis (parámetros, estructura,…)
- P: conjunto de prueba.
  - Con el que se calcula la tasa de error.

- Selección aleatoria de T, V y P. **Los elementos de P no pueden utilizarse en T y V de ninguna forma**.
- Cuanto más grandes, mejor.
- Construir clasificador con T, V (V no es necesario si validación cruzada interna)
- Tasa de error con P (según resumen pag. 11).

- Una vez evaluada la tasa de error, **TODOS** los datos pueden utilizarse para construir el clasificador. No necesario en un contexto Big Data.

Todos los datos para construir el clasificador

- Suposición: el error verdadero disminuye con el tamaño del conjunto de entrenamiento
  - Hipótesis de trabajo.
  - Razonable si los ejemplos se eligen de forma aleatoria.
  - Contrastada experimentalmente
    - Menor disminución con mayor tamaño del conjunto.
- Consecuencia: la tasa de error es una estimación pesimista
  - El error verdadero con más datos será menor que el error verdadero con el conjunto de entrenamiento.
- No son admisibles las estimaciones optimistas.

Insuficientes datos /ajuste de parámetros

- Es necesario reservar algunos datos para la evaluación.
- Que no se hayan utilizado para el ajuste de parámetros.

- Big data: no es problemático, pues dispone de suficientes datos.
- Problemático si pocos datos.
  - Normalmente, cuanto más grande sea el conjunto de entrenamiento mejor será el clasificador (mejoras cada vez más pequeñas).
  - Cuanto más grande sea el conjunto de test, más precisa será la estimación del error.

- Una vez que la evaluación se ha completado, se pueden usar **todos los datos** disponibles para construir el clasificador final.
- De nuevo, si la escala es Big Data, esto puede no ser necesario (ni posible).

Holdout (reserva)

- Dividir los datos originales en entrenamiento y prueba.
  - Dilema: idealmente, ambos conjuntos deberían ser grandes.
  - Buen clasificador o buena estimación del error.

- Típicamente $1/3$: $2/3\ T$, $1/3\ P$, de forma aleatoria.
- Big data 80%, 20% es razonable pues el conjunto de prueba es suficientemente grande.

- Inconveniente: las muestras podrían no ser representativas.
  - E.g., una clase podría no estar presente.
  - Incluso en big data.

- **Estratificación**: asegura que cada clase está representada con aproximadamente las mismas proporciones en los dos subconjuntos.

Holdout repetido

- La estimación de *hold out* depende de la partición aleatoria y tiene mucha variabilidad.
- Solución: repetir el proceso y promediar la tasa de error.

- Repetir el proceso k veces con diferentes muestras
  - En cada iteración se elijen nuevos T y P de forma aleatoria, manteniendo la proporción.

- Tasa de error: promedio, $e(h)=\sum_{i=1,k} e_i(h)/k$

- Varianza, estimación normal: optimista, no recomendable pues *h* y *P* no son tan independientes (peor aún si *n<30*).

Holdout repetido, varianza

- Más realista: estimar varianza muestral, S^2:

$$S^2_{e(h)} = 1/(k-1) \times \sum_{i=1,k} (e_i(h) - e(h))^2$$

- Intervalos de confianza: t-student.

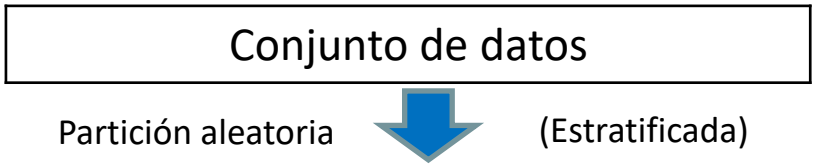
- Con probabilidad $N\%$, $e_D(h)$ está en el intervalo:

$$e(h) \pm t_{N,k-1} \times S_{e(h)}/\sqrt{k}$$

- $t_{N,k-1}$: *t-student* de $k-1$ grados de libertad para confianza $N\%$
  - Si $k \gg$, t tiende a la distribución normal.

Validación cruzada

- *Holdout* repetido no es óptimo: los conjuntos de prueba se solapan
- Validación cruzada evita el solapamiento de los conjuntos de prueba
- **Validación cruzada de k particiones** (k-*fold cross validation*, $k - XV$)
  - Primer paso: repartir los datos en k subconjuntos del mismo tamaño.
  - Segundo paso: usar cada subconjunto como prueba, el resto para entrenamiento.



Iteración	Entrenamiento	Prueba
1	S_2, S_3	S_1
2	S_1, S_3	S_2
3	S_1, S_2	S_3

- Variante estratificada: se estratifican los conjuntos en el primer paso.

Validación cruzada: estimación del error

- Tasa de error, promedio: $e(h) = \sum_{i=1,k} e_i(h)/k$

- Estimar varianza muestral, S^2:

$$S^2_{e(h)} = 1/(k-1) \times \sum_{i=1,k} (e_i(h) - e(h))^2$$

- Intervalos de confianza: *t-student.*
- Con probabilidad $N\%, e_D(h)$ está en el intervalo:

$$e(h) \pm t_{N,k-1} \times S_{e(h)}/\sqrt{k}$$

Validación cruzada

- Estándar: 10-fold *stratified cross validation*.
  - Habitual, pero nada especial con el 10.
  - En un contexto Big Data, 5, incluso 3 *folds*, pueden ser suficientes.
- La estratificación reduce la varianza del estimador.
- Ni la estratificación ni la división tienen que ser exactas.
- La estima del error se ve afectada por la partición aleatoria.

- Validación cruzada repetida.
  - Para paliar la influencia de la partición aleatoria.
  - E.g.: $10 \times 10, 5 \times 2 \cdots$

Validación cruzada repetida: estimación del error

- Validación cruzada repetida $R \times k$ *(Repetitions* $\times$ *k-folds).*
- Tasa de error, promedio: $e(h)=[\sum_{i=1,R\times k} e_i(h)]/(R \times k)$

- Estimar varianza muestral, S^2:

$$S^2_{e(h)} = 1/(R \times k-1) \times \sum_{i=1,R\times k} (e_i(h)- e(h))^2$$

- Intervalos de confianza: *t-student.*
- Con probabilidad $N\%$, $e_D(h)$ está en el intervalo:

$$e(h) \pm t_{N,R\times k-1} \times S_{e(h)}/\sqrt{R \times k}$$

- Ligeramente optimista: las repeticiones no son totalmente independientes.

Comparación de métodos de estimación del error.

Método	Características
Error de resubstitución	Optimista
Holdoud	Pesimista, muy muy variable
Holdoud repetido	Pesimista, muy variable
Validación cruzada	Menos sesgado, variable
Validación cruzada repetida	Menos variable, más costoso

5. Evaluación y selección de modelos

- Necesario para el ajuste de parámetros de un clasificador.

- *D*: conjunto de datos disponibles.
- Realizamos una partición aleatoria: ***T, P***.

- Utilizamos **T** para **entrenar y seleccionar el modelo.**
- Por lo tanto, hay que realizar evaluaciones honestas sobre T para determinar buenos valores de los parámetros del modelo
  - ***Entrenamiento y validación***.
  - ***Validación cruzada interna***.

- Una vez fijados los parámetros, crear clasificador con ***T***.
- **Evaluar con *P*.**
- Crear clasificador final con *D (quizás no en Big data).*

Entrenamiento y validación (y evaluación)

- En Big Data, de utilidad si no tenemos suficientes recursos

- *D*: conjunto de datos disponibles.
- Realizamos una participan aleatoria: ***T, P.***

- Utilizamos **T** para **entrenar y seleccionar el modelo**
- Mediante un proceso de ***Entrenamiento y validación***
  - Partición aleatoria de ***T*** *en* ***T'*** *y* ***V***
  - Creamos modelos con T'
  - Los evaluamos sobre V, para encontrar unos buenos valores de los parámetros del modelo

- Una vez fijados los parámetros**, crear clasificador con *T***.
- **Evaluar con *P***.
- Crear clasificador final con *D (quizás no en Big data)*.

Validación cruzada interna (y evaluación)

- Habitual para el ajuste de parámetros de un clasificador.

- *D*: conjunto de datos disponibles.
- Realizamos una participan aleatoria: ***T, P***.

- Utilizamos **T** para **entrenar y seleccionar el modelo**
- Mediante un proceso de ***Validación cruzada interna***
  - Creamos y evaluamos modelos mediante validación cruzada sobre ***T,*** para encontrar unos buenos valores de los parámetros del modelo.

- Una vez fijados los parámetros, **crear clasificador con *T***.
- **Evaluar con *P*.**
- Crear clasificador final con *D (quizás no en Big Data)*

6. Sobre la tasa de error

- La tasa de error de un clasificador permite evaluar la calidad de una hipótesis (un modelo).
- No proporciona información sobre la calidad de un algoritmo de aprendizaje.
- El mismo algoritmo podría generar hipótesis con mayor/menor tasa de error
  - Misma hipótesis sobre otro conjunto de prueba.
  - Otro conjunto de aprendizaje.
  - Otro dominio de aplicación.
- La comparación de algoritmos requiere test de hipótesis.
- **Ningún algoritmo es mejor que otro sobre cualquier dominio de aplicación.**

Limitaciones de la tasa de error

- Hasta ahora, hemos utilizado la tasa de error como criterio para evaluar hipótesis.
- Suposiciones:
  - Distribuciones de clases no muy desequilibradas.
  - Los costes de los errores son los mismos.
- No siempre es así
  - 99,99% de la población no es terrorista
    - «No terrorista» cierto 99,99%
  - El 97% de los días del mes las vacas no están en celo
    - «No celo» cierto 97,00%
- ¿Coste de diagnosticar erróneamente la presencia de una enfermedad frente a no diagnosticar una enfermedad real?

7. Referencias

- Tom M. Mitchell. Machine Learning. McGraw-Hill, 1997.
- Ian H. Witten, Eibe Frank, Mark A. Hall, Christopher Pal. Data Mining: practical machine learning tools and techniques (4th Edition). Morgan Kaufmann, 2016. ISBN: 9780128042915,
- David Page. Evaluating Machine Learning Methods. https://pages.cs.wisc.edu/~dpage/cs760/evaluating.pdf. Último acceso: octubre 2025.

Para los más teóricos:

- Arlot, S. and Celisse, A. A survey of cross-validation procedures for model selection. Statistics Surveys, 4, 40-79, 2010. DOI:10.1214/09-SS054. Disponible en: https://projecteuclid.org/journals/statistics-surveys/volume-4/issue-none/A-survey-of-cross-validation-procedures-for-model-selection/10.1214/09-SS054.full. Último acceso: octubre 2025.