UNIVERSIDAD COMPLUTENSE DE MADRID

ntic master School

Spark MLlib

Dr. Pablo J. Villacorta

Mayo de 2021

Agenda

- Introducción a Spark MLlib
  - Paquete pyspark.mllib vs pyspark.ml
- Formas de explotar un modelo entrenado con Spark
- Estimadores, transformadores y pipelines

1 Introducción a Spark MLlib

El módulo Spark MLlib

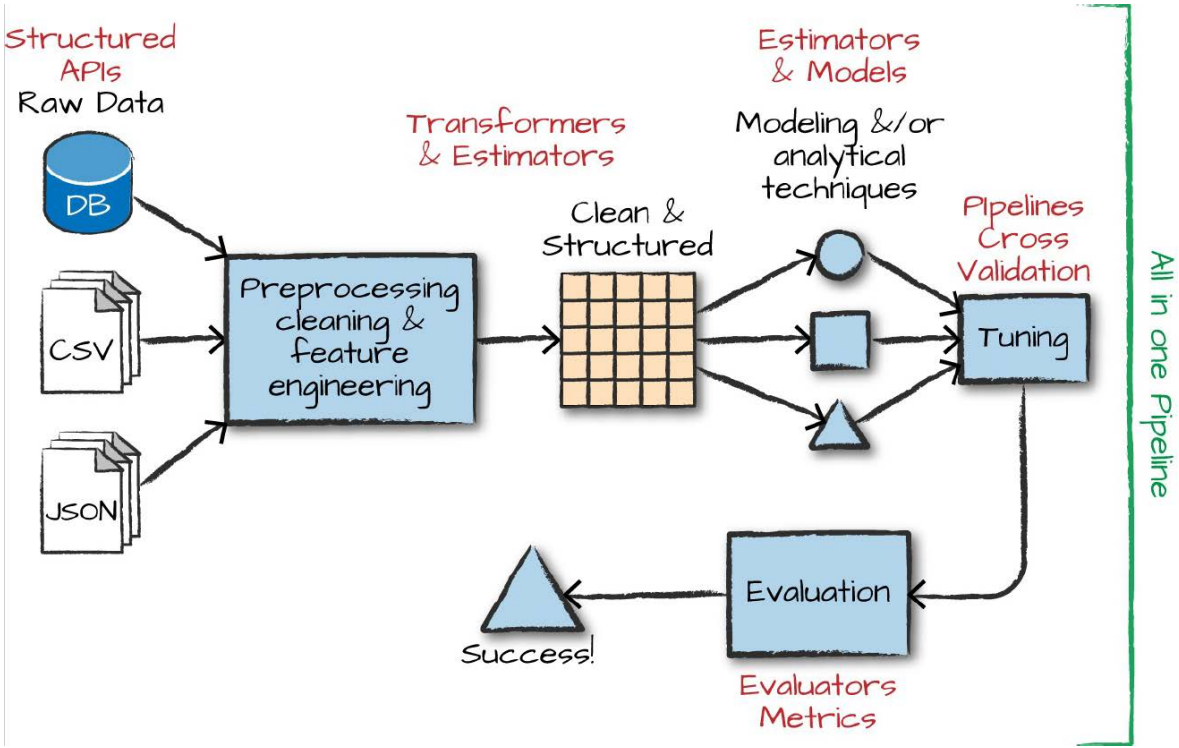
- Spark MLlib: módulo de Spark para ayudarnos en operaciones de
  - Limpieza de datos (aunque la mayoría se realizan con Spark SQL)
  - Ingeniería de variables (creación de variables desde datos en crudo)
  - Aprendizaje de modelos sobre datasets muy grandes (distribuidos)
  - Ajuste de híper-parámetros y evaluación de modelos

- **No proporciona métodos para despliegue en producción** de modelos entrenados (hoy, microservicios que usan el modelo entrenado para predecir un ejemplo, online)

- Puede ayudar en el proceso de **ingeniería de variables** y **entrenamiento**, incluso si el modelo entrenado va a ser explotado después con otro software no distribuido

El módulo Spark MLlib

- Fases del proceso de exploración y ajuste de un modelo, y herramientas de Spark para asistirnos en cada una de ellas



Despliegue de modelos con Spark

- Spark **no** fue concebido para explotación online de modelos entrenados, es decir, dar respuestas rápidas (predicciones) a un ejemplo nuevo que se recibe, por ejemplo, desde un sitio web, sino para entrenar modelos con grandes cantidades de datos
- Aun así, hay varias formas de aprovechar el modelo entrenado obtenido por Spark:

1. Entrenar en batch con datos offline almacenados en HDFS (probablemente el proceso de creación de variables + entrenamiento llevará un tiempo), y usar el modelo entrenado para predecir en modo batch sobre otro conjunto de datos preparados para la predicción (esta etapa es mucho más rápida).

    - *Es un enfoque **muy** frecuente en clientes*. Es posible cuando los datos que hay que predecir ya se conocen en el momento de entrenar, por ejemplo series temporales para predecir una ventana a futuro.
    - Las predicciones pre-calculadas se almacenan en HDFS o en una BD, indexadas para que sea muy rápido servirlas desde un microservicio

Despliegue de modelos con Spark

2. Entrenar, guardar el modelo entrenado, y usarlo desde Spark para hacer predicciones. Poco recomendable: lanzar un job nuevo para cada ejemplo que predecir implica sobrecarga. Balanceo de carga con réplicas del modelo

3. Entrenar y exportar el modelo a un formato de intercambio, ej. PMML, para leerlo y explotarlo con otra herramienta no distribuida (Python en especial)

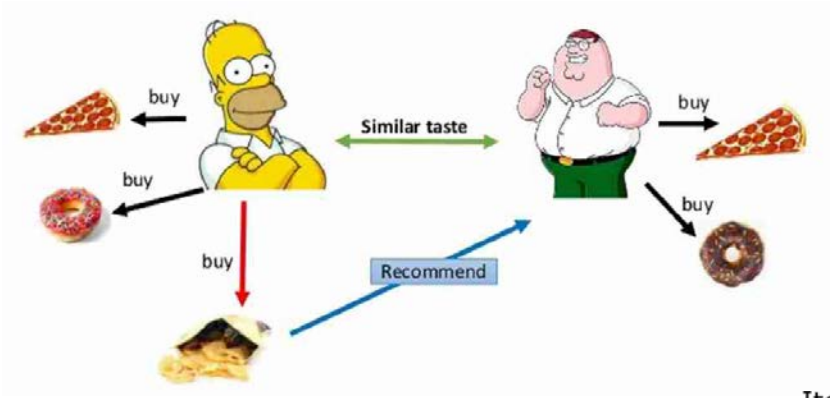
4. Entrenar y guardar el modelo, y usarlo desde Spark con Structured Streaming para hacer predicciones en mini-batches. Válido si vamos a recibir muchos datos de manera continua y no es muy relevante el tiempo de respuesta sino el throughput (métrica que optimiza Structured Streaming)

Spark MLlib vs Spark ML

- En la API de pyspark se distinguen:

  - Paquete pyspark.mllib: API *antigua* basada en **RDD** de una estructura llamada LabeledPoint. **Obsoleta**.
  - Paquete pyspark.ml: API actual, sobre **DataFrames**. En la medida de lo posible, se debe utilizar siempre.
  - *Casi* todo está migrado del módulo mllib al módulo ml, excepto algunas clases en *métricas de evaluación* y algún algoritmo de recomendación.

Filtrado colaborativo



Item

User	W	X	Y	Z
A		4.5	2.0	
B	4.0		3.5	
C		5.0		2.0
D		3.5	4.0	1.0

Rating Matrix

=

A	1.2	0.8
B	1.4	0.9
C	1.5	1.0
D	1.2	0.8

User Matrix

X

W	X	Y	Z
1.5	1.2	1.0	0.8
1.7	0.6	1.1	0.4

Item Matrix

Ingeniería de variables y preprocesado

- La mayoría de las operaciones de manipulación, limpieza y transformación de datos en Spark se llevan a cabo con la API de **Spark SQL** (API estructurada, DataFrames)
  - Incluye la creación de nuevas variables (feature engineering)
  - Existe un transformador para poder insertar código SQL en un pipeline: *SQLTransformer*

- Hay **ciertos tipos de pre-procesamiento** para los que MLlib nos facilita herramientas:
  - Escalado / estandarización / normalización
  - Codificación One Hot
  - Tokenización (separación de textos en palabras)
  - Representación de textos mediante vectores numéricos con TF - IDF

- Además, es necesario un pre-procesamiento específico para los algoritmos de MLlib: exige un formato de entrada concreto y rígido en los DataFrames de entrenamiento
  - Pre-procesamiento adicional para preparar el DF para Spark ML después del habitual de limpieza de datos y creación de variables

Ingeniería de variables y preprocesado

- Modelos de clasificación: la variable target que representa las clases debe ser *Double,* con parte decimal a 0 y empezando en 0.0
- Cualquier modelo: las variables (los predictores) deben ir en **una sola columna de tipo vector**
  - Excepción: recomendación por filtrado colaborativo (ALS)

Clasificación y regresión

features	target
[-32.2, 4.5, 1.0, 6.7]	1.0

Clustering

features
[-32.2, 4.5, 1.0, 6.7]

Recomendación (filtrado colaborativo)

user	item	rating
3	27	2.8

Ingeniería de variables en Spark

Extracting, transforming and selecting features

This section covers algorithms for working with features, roughly divided into these groups:

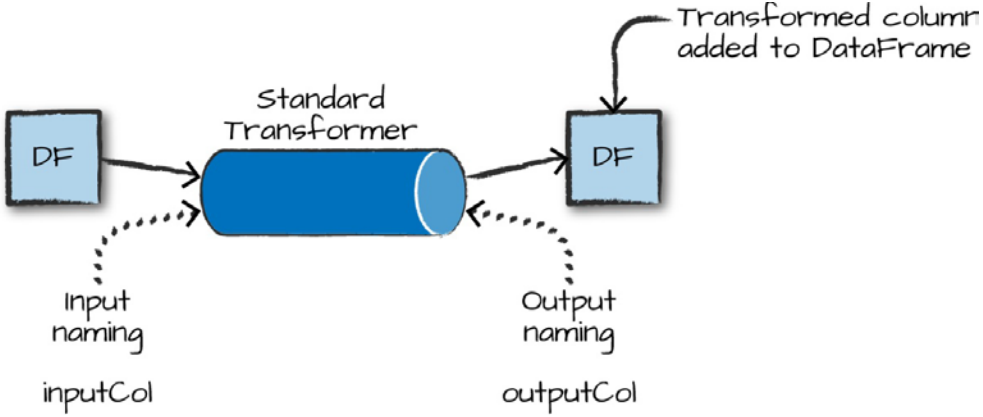
- Extraction: Extracting features from “raw” data
- Transformation: Scaling, converting, or modifying features
- Selection: Selecting a subset from a larger set of features
- Locality Sensitive Hashing (LSH): This class of algorithms combines aspects of feature transformation with other algorithms.

Table of Contents

https://spark.apache.org/docs/latest/ml-features.html

Transformadores en Spark ML

- **Transformer**: función que recibe como entrada un DF y uno o varios nombres de columna existentes (*inputCol*), y las transforma de alguna manera. Salida: el mismo DF con una nueva columna añadida, con el nombre que hayamos fijado (*outputCol*)



- La interfaz *Transformer* tiene un único método: *transform(df: DataFrame)* que recibe un DF y devuelve otro DF

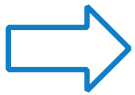
- Los transformadores *no tienen ningún parámetro que aprender* de los datos. Su salida es un DataFrame

Transformadores muy comunes

- **VectorAssembler**: recibe varias columnas y las concatena en una sola de tipo vector, de longitud igual al número de columnas que se quieran ensamblar. Necesario para calcular la columna (única) de *features* en los algoritmos de aprendizaje supervisado.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
dataset = spark.createDataFrame(
     [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.51]), 1.0)],
     ["id", "hour", "mobile", "userFeatures", "clicked"])

assembler = VectorAssembler(
     inputCols=["hour", "mobile", "userFeatures"],
     outputCol="features")

output = assembler.transform(dataset)
print("Assembled 'hour', 'mobile', 'userFeatures' to column 'features'")
output.select("features", "clicked").show(truncate = False)
```

```
 id | hour | mobile | userFeatures     | clicked
----|------|--------|------------------|---------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0
```



```
 id | hour | mobile | userFeatures     | clicked | features
----|------|--------|------------------|---------|-----------------------------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0     | [18.0, 1.0, 0.0, 10.0, 0.5]
```

Transformadores muy comunes

- **Bucketizer**: transforma una columna continua en una columna con *identificadores de intervalo (bucket).* Recibe como parámetro el vector con los límites de cada intervalo

```
from pyspark.ml.feature import Bucketizer
splits = [-float("inf"), -0.5, 0.0, 0.5, float("inf")]

data = [(-999.9,), (-0.5,), (-0.3,), (0.0,), (0.2,), (999.9,)]
dataFrame = spark.createDataFrame(data, ["features"])

bucketizer = Bucketizer(splits=splits, inputCol="features",
outputCol="bucketedFeatures")

# Transform original data into its bucket index
bucketedData = bucketizer.transform(dataFrame)

print("Bucketizer output with %d buckets"
      % (len(bucketizer.getSplits())-1))

bucketedData.show()
```

```
+--------+----------------+
|features|bucketedFeatures|
+--------+----------------+
|  -999.9|             0.0|
|    -0.5|             1.0|
|    -0.3|             1.0|
|     0.0|             2.0|
|     0.2|             2.0|
|   999.9|             3.0|
+--------+----------------+
```

Transformadores muy usuales

- *Cualquier modelo entrenado*: el resultado de entrenar un modelo sobre un DF es un objeto **Model* de la subclase específica del modelo que hayamos ajustado
- Todo modelo **entrenado** es también un *Transformer* por lo que es capaz de *transformar* (hacer predicciones con) un DF de ejemplos, siempre que tenga el mismo formato (mismos nombres de columnas y tipos de datos) que el DF que se utilizó para entrenar
- En general, *transformar* quiere decir añadir una o varias columnas nuevas al DF, en este caso con las predicciones
- Para facilitar que se mantenga el mismo formato, se suele entrenar un pipeline completo y utilizar su salida (*pipeline entrenado*) como transformador

Transformadores muy usuales

```
%> from pyspark.ml.regression import LinearRegression
%> training = spark.read.format("libsvm")\
               .load("sample_linear_regression_data.txt")
%> training.show()
+-------------------+--------------------+
|              label|            features|
+-------------------+--------------------+
| -9.490009878824548|(10,[0,1,2,3,4,5,...|
| 0.2577820163584905|(10,[0,1,2,3,4,5,...|
      ...
| 1.5299675726687754|(10,[0,1,2,3,4,5,...|
| -0.250102447941961|(10,[0,1,2,3,4,5,...|
+-------------------+--------------------+
%> lr = LinearRegression(maxIter=10, regParam=0.3,
     elasticNetParam=0.8, featuresCol = "features",
     labelCol = "label")
%> lrModel = lr.fit(training)

%> lrModel.__class__
<class 'pyspark.ml.regression.LinearRegressionModel'>

%> from pyspark.ml import Transformer

%> isinstance(lrModel, Transformer)
True
```

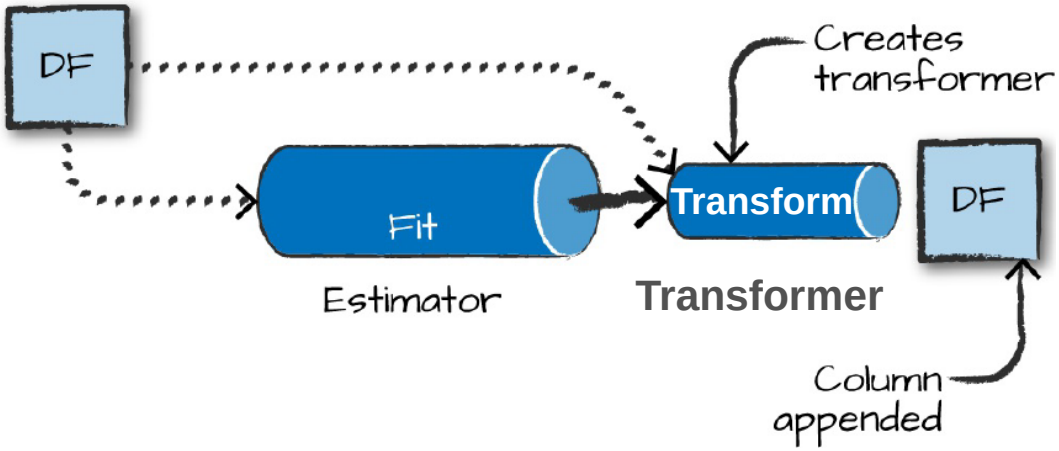
```
%> pred = lrModel.transform(training)
%> pred.show()
+-------------------+--------------------+--------------------+
|              label|            features|          prediction|
+-------------------+--------------------+--------------------+
| -9.490009878824548|(10,[0,1,2,3,4,5,...| 0.39922280427864854|
| 0.2577820163584905|(10,[0,1,2,3,4,5,...|-0.29559741764686487|
| -4.438869807456516|(10,[0,1,2,3,4,5,...|  0.7651496483023066|
|-19.782762789614537|(10,[0,1,2,3,4,5,...|  0.7839239258929726|
| -7.966593841555266|(10,[0,1,2,3,4,5,...|  1.4831466765011345|
| -7.896274316726144|(10,[0,1,2,3,4,5,...| -0.9871618140066576|
| -8.464803554195287|(10,[0,1,2,3,4,5,...|  1.5395124755034428|
| 2.1214592666251364|(10,[0,1,2,3,4,5,...| 0.05906145957465214|
| 1.0720117616524107|(10,[0,1,2,3,4,5,...| -2.0397390816430665|
|-13.772441561702871|(10,[0,1,2,3,4,5,...|  2.1211666677165093|
| -5.082010756207233|(10,[0,1,2,3,4,5,...|-0.04572650153420729|
|  7.887786536531237|(10,[0,1,2,3,4,5,...|  1.4045706595369045|
| 14.323146365332388|(10,[0,1,2,3,4,5,...|  1.8936490662233862|
|-20.057482615789212|(10,[0,1,2,3,4,5,...|  0.2625495742873283|
|-0.8995693247765151|(10,[0,1,2,3,4,5,...|  1.1054144970959854|
| -19.16829262296376|(10,[0,1,2,3,4,5,...| -1.3003908887799676|
|  5.601801561245534|(10,[0,1,2,3,4,5,...| -2.0446543261749612|
|-3.2256352187273354|(10,[0,1,2,3,4,5,...|  -0.960287000485595|
| 1.5299675726687754|(10,[0,1,2,3,4,5,...|  1.6330567770307318|
| -0.250102447941961|(10,[0,1,2,3,4,5,...|  1.1299316224433398|
+-------------------+--------------------+--------------------+
```

lr : Estimador

lrModel : Transformador (modelo entrenado)

Estimadores en Spark ML

- *Estimator*: herramienta de Spark para poder realizar transformaciones que requieren que ciertos parámetros de la transformación se ajusten (o se *aprendan*) a partir de los datos.
- Normalmente requieren una pasada previa (o varias) sobre la columna que se desea transformar
- La interfaz *Estimator* tiene un único método: *fit(df: DataFrame)* que recibe un DF y devuelve un objeto de tipo **Model*, el *modelo entrenado*, que es además un Transformer
  - OJO: Spark llama *modelo* a cualquier cosa que requiera un *fit* previo, no sólo a los algoritmos de machine learning



Estimadores muy comunes

- **StringIndexer**: estimador para pre-procesar variables categóricas*. Es el más utilizado.* Convierte una columna categórica (da igual el tipo ya que los valores serán entendidos como categorías) en *Double* empezando en 0.0, donde las categorías se representan mediante 0.0, 1.0, 2.0…
  - **Además**, añade metadatos al DataFrame transformado (resultante de *transform*) indicando que esa columna es categórica (no es simplemente continua)
  - Los algoritmos que sí soportan variables categóricas (ej: DecisionTree, RandomForest, GradientBoostedTrees) requieren que se las pasemos indexadas
  - Los algoritmos que no soportan variables categóricas (LinearRegression, LogisticRegression) requieren el uso de *OneHotEncoder* (siguiente slide)
  - IMPORTANTE: al predecir ejemplos nuevos, hay que usar la misma codificación!

Estimadores muy comunes: StringIndexer

```
from pyspark.ml.feature import StringIndexer
df = spark.createDataFrame(
     [(0,"a"), (1,"b"), (2,"c"), (3,"a"), (4,"a"), (5,"c")],
     ["id", "category"])

indexer = StringIndexer(inputCol = "category",
                        outputCol = "categoryIndex")

# guardo el transformer para usarlo después
indexerModel = indexer.fit(df)
indexedDF = indexerModel.transform(df)
indexedDF.show()
```

```
id | category | categoryIndex
---|----------|--------------
0  | a        | 0.0
1  | b        | 2.0
2  | c        | 1.0
```

```
… # resto de código
indexedNuevo = indexerModel.transform(otroDF) # lo uso otra vez!
```

Estimadores muy comunes

- **OneHotEncoderEstimator***:* recibe un conjunto de columnas y convierte cada una (de manera independiente) a un conjunto de variables *dummy* con codificación one-hot
  - Cada variable (con *n* categorías posibles) da lugar a *n*-1 columnas (en una sola columna de tipo vector) donde en cada ejemplo, sólo una de ellas tiene valor 1 y el resto son 0 indicando cuál es el valor de la categoría presente en ese ejemplo
  - Spark siempre asume que los valores provienen de una indexación previa con *StringIndexer*: obligatoriamente números reales con la parte decimal a 0

- **Cualquier algoritmo de predicción** *(Machine Learning*) es un Estimator antes de ser ajustado:
  - Todos los modelos heredan de *Estimator:* el método *fit(df)* lanza el aprendizaje.
  - Los *Estimators* suelen tener muchos parámetros configurables antes de *fit* (en algunos *Transformers* también hay parámetros configurables, pero suelen ser menos).

Estimadores muy comunes

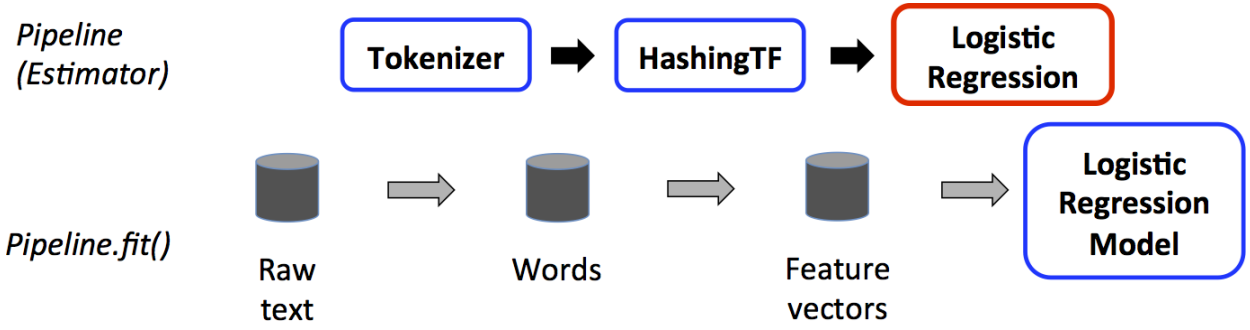
```
>>> from pyspark.ml.feature import OneHotEncoderEstimator
>>> df = spark.createDataFrame([
     (0.0, 1.0, 2.0),
     (1.0, 0.0, 3.0),
     (2.0, 1.0, 2.0),  # Spark asume que la tercera columna tiene 5 categorías!
     (0.0, 2.0, 1.0),
     (0.0, 1.0, 4.0),
     (2.0, 0.0, 4.0)
], ["categoryIndex1", "categoryIndex2", "categoryIndex3"])

>>> encoder = OneHotEncoderEstimator(inputCols = ["categoryIndex1", "categoryIndex2",  "categoryIndex3"],
                                     outputCols = ["categoryVec1", "categoryVec2", "categoryVec3"])

>>> model = encoder.fit(df)
>>> encoded = model.transform(df)
>>> encoded.show()

+-------------+-------------+-------------+-------------+-------------+-----------------+
|categoryIndex1|categoryIndex2|categoryIndex3|categoryVec1|categoryVec2|     categoryVec3|
+-------------+-------------+-------------+-------------+-------------+-----------------+
|          0.0|          1.0|          2.0|   [1.0,0.0]|   [0.0,1.0]|[0.0,0.0,1.0,0.0]|
|          1.0|          0.0|          3.0|   [0.0,1.0]|   [1.0,0.0]|[0.0,0.0,0.0,1.0]|
|          2.0|          1.0|          2.0|   [0.0,0.0]|   [0.0,1.0]|[0.0,0.0,1.0,0.0]|
|          0.0|          2.0|          1.0|   [1.0,0.0]|   [0.0,0.0]|[0.0,1.0,0.0,0.0]|
|          0.0|          1.0|          4.0|   [1.0,0.0]|   [0.0,1.0]|[0.0,0.0,0.0,0.0]|
|          2.0|          0.0|          4.0|   [0.0,0.0]|   [1.0,0.0]|[0.0,0.0,0.0,0.0]|
+-------------+-------------+-------------+-------------+-------------+-----------------+
```

Pipelines

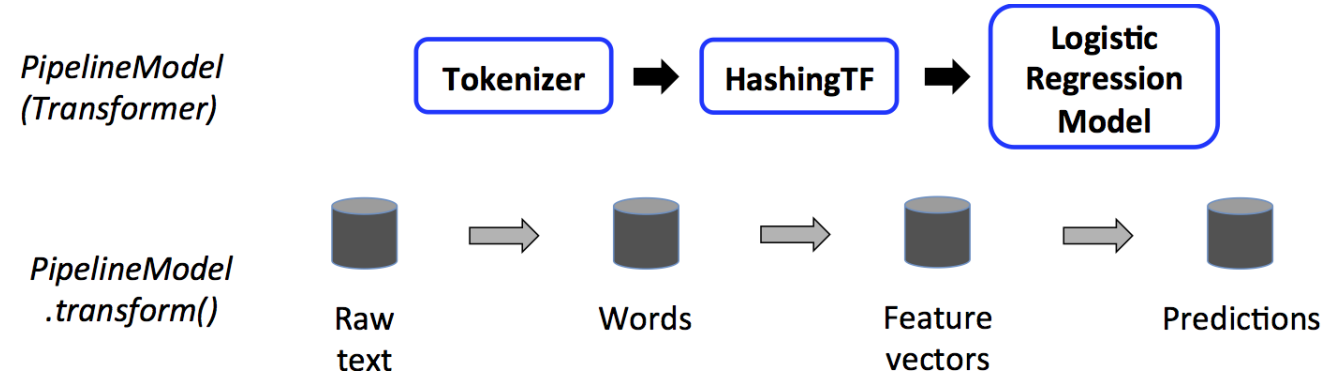
- Es frecuente en ML extraer características de datos raw y prepararlas antes de llamar a un algoritmo de aprendizaje.
- Puede ser difícil tener control de todos los pasos de pre-procesamiento si luego queremos replicarlos en otros conjuntos de datos o en el momento de hacer predicciones con nuevos datos. Ejemplo: para procesar un documento:
  - División en palabras
  - Procesamiento de palabras para obtener un vector de características numéricas
  - Preparación de esas características para el formato que requiere el algoritmo elegido en Spark
  - Finalmente, entrenamiento de un modelo.
- **Pipeline**: secuencia de etapas (*PipelineStage*, superclase de Estimator y Transformer) que se ejecutan en un cierto orden. La salida de una etapa es entrada para *alguna* de las etapas posteriores (no necesariamente la inmediatamente siguiente)
  - Es un **Estimator**. El método *fit(df)* recorre cada etapas, **llama a transform() si la etapa es un Transformer, y llama a fit(df) y luego a transform(df) si es un Estimator**, pasando siempre el DF tal como esté en ese punto (con las columnas originales más las que le hayan añadido las etapas previas).
  - Es habitual (pero no obligatorio) que la última etapa sea un algoritmo de ML, aunque podría haber varios a lo largo de un pipeline
  - IMPORTANTE: un mismo objeto no puede ser añadido a dos pipelines

Pipelines

- Pipeline antes de llamar al método *fit(df)*:



- PipelineModel (ajustado) devuelto por la llamada a *fit(df)*: todo son Transformers



Pipelines

```
from pyspark.ml.feature import StringIndexer, VectorAssembler, Binarizer, VectorSlicer, StandardScaler
from pyspark.ml import Pipeline from pyspark.ml.classification import LogisticRegression

trainTest = spark.read.parquet("flights.parquet").randomSplit([0.8, 0.2], 12345)
trainingData = trainTest[0]
testingData = trainTest[1]

monthIndexer = StringIndexer().setInputCol("Month").setOutputCol("MonthCat")
dayofMonthIndexer = StringIndexer().setInputCol("DayofMonth").setOutputCol("DayofMonthCat")
dayOfWeekIndexer = StringIndexer().setInputCol("DayOfWeek").setOutputCol("DayOfWeekCat")
uniqueCarrierIndexer = StringIndexer().setInputCol("UniqueCarrier").setOutputCol("UniqueCarrierCat")
originIndexer = StringIndexer().setInputCol("Origin").setOutputCol("OriginCat")

assembler = VectorAssembler().setInputCols(["MonthCat", "DayofMonthCat", "DayOfWeekCat", "UniqueCarrierCat",
       "OriginCat", "DepTime", "CRSDepTime", "ArrTime", "CRSArrTime", "ActualElapsedTime", "CRSElapsedTime",     "AirTime","DepDelay",
"Distance"]).setOutputCol("rawFeatures")

slicer = VectorSlicer().setInputCol("rawFeatures").setOutputCol("slicedfeatures").setNames(["MonthCat",
       "DayofMonthCat", "DayOfWeekCat", "UniqueCarrierCat", "DepTime", "ArrTime", "ActualElapsedTime",    "AirTime", "DepDelay", "Distance"])

scaler = StandardScaler().setInputCol("slicedfeatures").setOutputCol("features")\
                              .setWithStd(True).setWithMean(True)

binarizerClassifier = Binarizer().setInputCol("ArrDelay").setOutputCol("binaryLabel").setThreshold(15.0)
lr = LogisticRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
                              .setLabelCol("binaryLabel").setFeaturesCol("features")

lrPipeline = Pipeline().setStages([monthIndexer, dayofMonthIndexer, dayOfWeekIndexer, uniqueCarrierIndexer,
       originIndexer, assembler, slicer, scaler, binarizerClassifier, lr])

pipelineModel = lrPipeline.fit(trainingData)
lrPredictions = pipelineModel.transform(testingData)

lrPredictions.select("prediction", "binaryLabel", "features").show(20)
```

Ajuste de híper-parámetros

- Casi todos los algoritmos tienen híper-parámetros cuyos valores debe fijar el usuario antes de iniciar el algoritmo. La búsqueda del valor óptimo se hace por ensayo-error de entre un conjunto de valores habituales factibles
- Spark proporciona herramientas para buscar la mejor combinación de valores de los híper-parámetros, indicándole el conjunto de posibles valores que debe probar con cada uno.
  - Se fija una combinación, se entrena el modelo con un subconjunto de los datos (datos de **entrenamiento**: el mismo para todas las combinaciones que se irán probando) y se evalúa con otro subconjunto (datos de **validación**, también los mismos para todas las combinaciones)
  - **Si hay suficientes datos**, basta con un solo conjunto de entrenamiento y otro de validación. Si no, se lleva a cabo Cross Validation.
  - Con cualquiera de las dos herramientas, Spark devuelve el modelo entrenado con la mejor combinación de parámetros, usando para ello todo el dataset
- Antes de empezar todo el proceso, se separa un subconjunto de **test** que no participa en el proceso, y será utilizado solamente una vez al final, para saber cómo se comportará el mejor modelo con datos nunca vistos, y obtener así una métrica que se entregará junto con el modelo entrenado.

Ajuste de híper-parámetros

Separamos un subconjunto de test para usarlo al final una sola vez

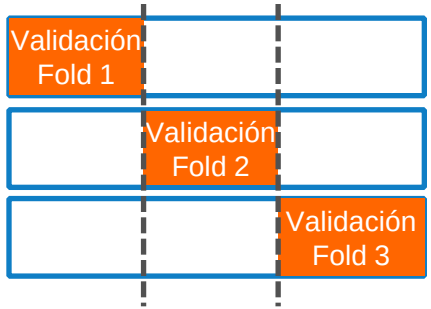
¿El conjunto de entrenamiento es suficientemente grande?

Sí

Spark: **`TrainValidationSplit`**

El conjunto de entrenamiento se divide a su vez en train + validación. Se entrena con train y se valida con el cjto de validación y esa es la métrica

No



El cjto. de entrenamiento se divide en K folds (K impar). Se entra con K-1 y se evalúa en el fold K-ésimo. Se toma la media, y ese es el valor de la métrica para cada combinación de híper-parámetros

Spark: **`CrossValidator`**

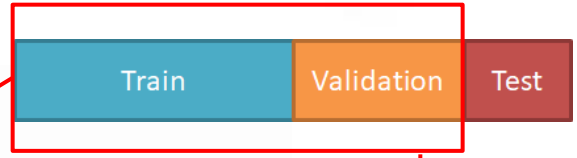
Ajuste de híper-parámetros: TrainValidationSplit

Train-Validation Split

In addition to `CrossValidator` Spark also offers `TrainValidationSplit` for hyper-parameter tuning. `TrainValidationSplit` only evaluates each combination of parameters once, as opposed to k times in the case of `CrossValidator`. It is therefore less expensive, but will not produce as reliable results when the training dataset is not sufficiently large.

Unlike `CrossValidator`, `TrainValidationSplit` creates a single (training, test) dataset pair. It splits the dataset into these two parts using the `trainRatio` parameter. For example with $trainRatio = 0.75$, `TrainValidationSplit` will generate a training and test dataset pair where 75% of the data is used for training and 25% for validation.

Like `CrossValidator`, `TrainValidationSplit` finally fits the `Estimator` using the best `ParamMap` and the entire dataset.



Ajuste de híper-parámetros: TrainValidationSplit

Cross-Validation

`CrossValidator` begins by splitting the dataset into a set of *folds* which are used as separate training and test datasets. E.g., with $k = 3$ folds, `CrossValidator` will generate 3 (training, test) dataset pairs, each of which uses 2/3 of the data for training and 1/3 for testing. To evaluate a particular `ParamMap`, `CrossValidator` computes the average evaluation metric for the 3 `Models` produced by fitting the `Estimator` on the 3 different (training, test) dataset pairs.

After identifying the best `ParamMap`, `CrossValidator` finally re-fits the `Estimator` using the best `ParamMap` and the entire dataset.

Examples: model selection via cross-validation

The following example demonstrates using `CrossValidator` to select from a grid of parameters.

Note that cross-validation over a grid of parameters is expensive. E.g., in the example below, the parameter grid has 3 values for `hashingTF.numFeatures` and 2 values for `lr.regParam`, and `CrossValidator` uses 2 folds. This multiplies out to $(3 \times 2) \times 2 = 12$ different models being trained. In realistic settings, it can be common to try many more parameters and use more folds ($k = 3$ and $k = 10$ are common). In other words, using `CrossValidator` can be very expensive. However, it is also a well-established method for choosing parameters which is more statistically sound than heuristic hand-tuning.