Procesamiento Distribuido para Big Data

Tema 4: Spark ML

Autor: Dr. Pablo J.
Villacorta Iglesias

Actualizado Marzo 2020

INTRODUCCIÓN

Spark MLlib es el módulo de Spark para tareas de

- Limpieza de datos
- Ingeniería de variables (creación de variables desde datos en crudo)
- Aprendizaje de modelos sobre datasets muy grandes (distribuidos)
- Ajuste de parámetros y evaluación de modelos

No proporciona métodos para despliegue en producción de modelos entrenados. En la actualidad, es muy frecuente desplegar modelos como microservicios que usan el modelo entrenado para realizar predicciones sobre un nuevo ejemplo que han recibido por medio de una llamada HTTP de una aplicación externa que les ha enviado el dato para predecir. Comentaremos más detalles en la próxima sección.

La esencia de MLlib es la implementación de modelos de manera distribuida utilizando Spark, y capaces de aprender sobre datasets muy grandes almacenados de manera distribuida. No obstante, también es muy frecuente utilizar solo ciertas funcionalidades de MLlib para pre-procesar variables en datasets masivos, limpiar, normalizar, etc y finalmente filtrar y pasar el dataset resultante (asumiendo que ya no es masivo) al driver para guardarlo en el sistema de archivos local. Posteriormente se puede utilizar una biblioteca de Machine Learning no distribuida, como por ejemplo Scikit-learn de Python o paquetes específicos de R como *caret* o *e1071*. Tanto Python como R son capaces de leer ficheros no distribuidos tanto en formato texto como CSV entre otros, y usarlos como entrada para un algoritmo de aprendizaje automático.

La Fig. 1 muestra una idea general del ciclo de ajuste de un modelo, junto a las herramientas que proporciona Spark para cada etapa (en color rojo). Además, Spark tiene una herramienta adicional llamada *pipelines* para encapsular todas estas etapas en un solo objeto indivisible, y asegurarnos de que los nuevos datos recibidos y con los que queremos realizar predicciones pasen por exactamente las mismas etapas de pre-procesamiento que el dataset estático con el que se entrenó el modelo. Por eso la figura se refiere a estos pasos como un *all-in-one Pipeline*.

La etapa de pre-procesamiento incluye, además de las operaciones típicas de un proyecto de Data Science, cierto procesamiento específico de Spark para pasar los datos al formato adecuado de columnas que esperan recibir las implementaciones de cada algoritmo en la API de Spark ML. MLlib incorpora herramientas para esto también.

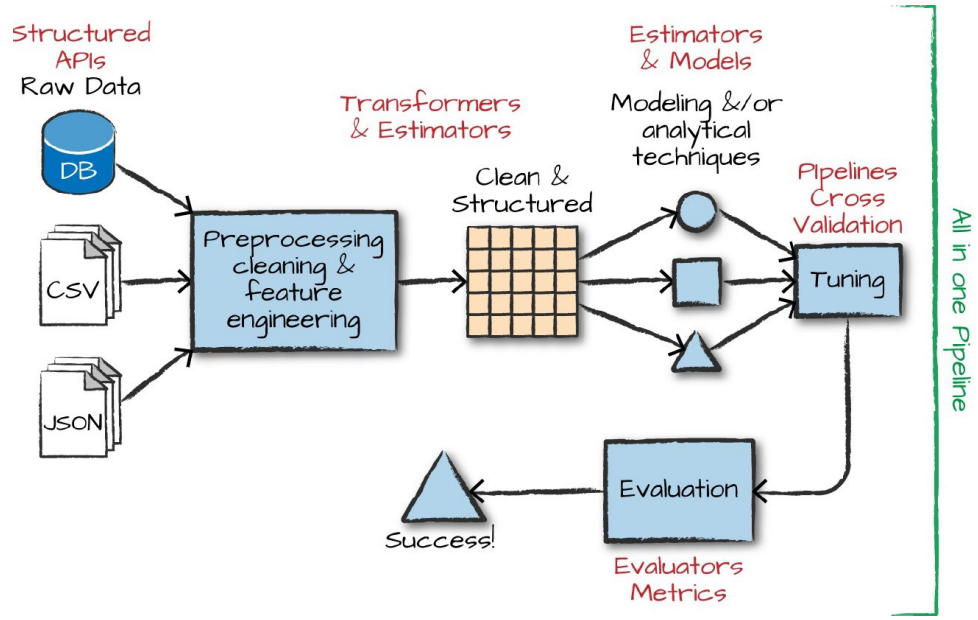


Fig 1. Etapas en el proceso de entrenamiento de un modelo a partir de datos. Imagen tomada de Spark: The Definitive Guide (2018)

Despliegue de modelos en producción 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. Por el contrario, la fortaleza de Spark está en entrenar modelos en modo batch (offline) con datasets de gran cantidad de datos.

Aun así, hay varias formas de aprovechar el modelo entrenado obtenido por Spark:

- 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 (también en modo batch) sobre otro conjunto de datos nuevos 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.

- 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.

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

- Entrenar en modo online usando Structured Streaming para recoger datos. Siempre re-entrenar desde 0 salvo si el modelo no está preparado para esto (modelos que soporten *online learning*, que en la actualidad son una minoría).

ESTIMADORES Y TRANSFORMADORES

Antes de describir en detalle la API del módulo Spark MLlib, hay que tener en cuenta que en la API de Spark (Java/Scala/Python) se distinguen:

- Paquete *org.apache.spark.mllib* (en Python: *pyspark.mllib*): API *antigua* basada en RDDs de una estructura llamada LabeledPoint:
  LabeledPoint(etiqueta, [vector de atributos]). Obsoleta, no debe usarse.

- Paquete *org.apache.spark.ml* (en Python: *pyspark.ml*): API actual, sobre DataFrames. En la medida de lo posible, se debe utilizar siempre.

  - *Casi* todo el contenido del módulo spark.mllib ya está migrado al módulo spark.ml, excepto algunas clases en *métricas de evaluación* y algún algoritmo de recomendación.

Para la creación de variables y el preprocesamiento, en general se utiliza la API de Spark SQL que vimos en el tema anterior. No obstante, Spark ML ofrece una transformación en la que puede escribirse código SQL arbitrario e incluir dicha transformación en un pipeline. Además, ciertas transformaciones relacionadas con el pre-procesamiento estadístico de datos (normalización, estandarización, codificación one-hot, etc) también están disponibles en Spark ML. Por último, existen varias transformaciones cuyo propósito no es realmente modificar los datos sino prepararlos (dar a las columnas del DataFrame los tipos adecuados) para la entrada de los algoritmos de Machine Learning de Spark. En concreto, Spark requiere estos formatos:

Problemas de clasificación
(clase codificada como número real)

features	label
[-32.2, 4.5, 1.0, 6.7] [-40.8, 2.25, 4.0, 2.3]	1.0 0.0

Problemas de regresión
(el target ya es un número real)

features	target
[-32.2, 4.5, 1.0, 6.7] [-40.8, 2.25, 4.0, 2.3]	0.72 -4.56

Problemas de clustering
(no existe columna label)

features
[-32.2, 4.5, 1.0, 6.7] [-40.8, 2.25, 4.0, 2.3]

Problemas de recomendación
con filtrado colaborativo

user	item	rating
2 1	3 19	4.5 3.21

Como vemos, los valores de las variables deben presentarse en una sola columna de tipo vector. Por otro lado, si se trata de un problema de aprendizaje supervisado (clasificación o regresión), la columna *target* debe ser siempre de tipo real (Double). En

el caso de los problemas de clasificación, cada clase se indica con un número real empezando en 0.0 y con la parte decimal del número siempre a 0 (es decir, si tenemos un problema con 5 clases, se tienen que codificar como 0.0, 1.0, 2.0, 3.0 y 4.0). En el caso de los problemas de regresión, la columna target puede ser cualquier número real. Los nombres de las columnas no tienen por qué ser *features* y *label* como en los ejemplos de arriba, sino que pueden ser cualesquiera, siempre que le indiquemos al algoritmo cómo se llama la columna (de tipo vector) que contiene las features en el DataFrame que le pasamos para entrenar, y cómo se llama la columna target, si la hay.

En el caso de algoritmos de recomendación, Spark solo permite filtrado colaborativo[1]. En ese caso hay que pasarle un DataFrame que contenga, al menos, tres columnas con los identificadores de un usuario, un ítem y el rating que ha dado ese usuario a ese ítem, ya sea implícito o explícito. Los identificadores de usuarios y de ítems tienen que ser códigos de tipo entero, mientras que el rating puede ser un número real en cualquier rango.

Según la documentación oficial de Spark, actualmente nos ofrece las siguientes posibilidades para preprocesamiento, divididas en varios grupos:

- Extracción: extraer variables a partir de datos en crudo
- Transformación: re-escalar, convertir o modificar variables
- Seleccionar: seleccionar un buen subconjunto de variables de otro más grande
- Locality Sensitive Hashing (LSH): combinación de trasnformaciones de variables con otros algoritmos.

Extracción de variables

TF-IDF	Word2Vec	CountVectorizer	FeatureHasher

Transformación de variables

Tokenizer	StopWordsRemover	n-gram
Binarizer	PCA	PolynomialExpansion
Discrete Cosine Transform (DCT)	StringIndexer	IndexToString
OneHotEncoder (Deprecated since 2.3.0)	OneHotEncoderEstimator	VectorIndexer
Interaction	Normalizer	StandardScaler
MinMaxScaler	MaxAbsScaler	Bucketizer
ElementwiseProduct	SQLTransformer	VectorAssembler
VectorSizeHint	QuantileDiscretizer	Imputer

Selección de variables

VectorSlicer	RFormula	ChiSqSelector

[1] Spark también implementa la descomposición SVD de una matriz, pero este algoritmo no está adaptado en la API al problema de la recomendación y tendría que ser el usuario el que utilizase las matrices U, S y V devueltas para implementar recomendaciones.

Locality Sensitive Hashing	LSH Operations

Transformación de variables

Approximate Similarity Join	Approximate Nearest Neighbor Search
LSH Algorithms	Bucketed Random Projection for Euclidean Distance
MinHash for Jaccard Distance	

Transformadores en Spark ML

Un *transformer* es un objeto que recibe como entrada un DataFrame de Spark y uno o varios nombres de columna existentes (por ejemplo *inputCol*), y las transforma de alguna manera. Su salida es el mismo DataFrame con una nueva columna añadida a por la derecha, con el nombre que hayamos indicado en el parámetro correspondiente (generalmente *outputCol* pero a veces puede ser *predictionCol*).

La interfaz *Transformer* tiene un único método: *transform(df: DataFrame)* que recibe un DataFrame y devuelve otro DataFrame. Los transformadores ***no necesitan aprender ningún parámetro del DataFrame de entrada***. Simplemente están preparados (tienen toda la información) para transformar un DataFrame, y esto es lo que hacen cuando llamamos a *transform()* tal como muestra la Fig. 2.

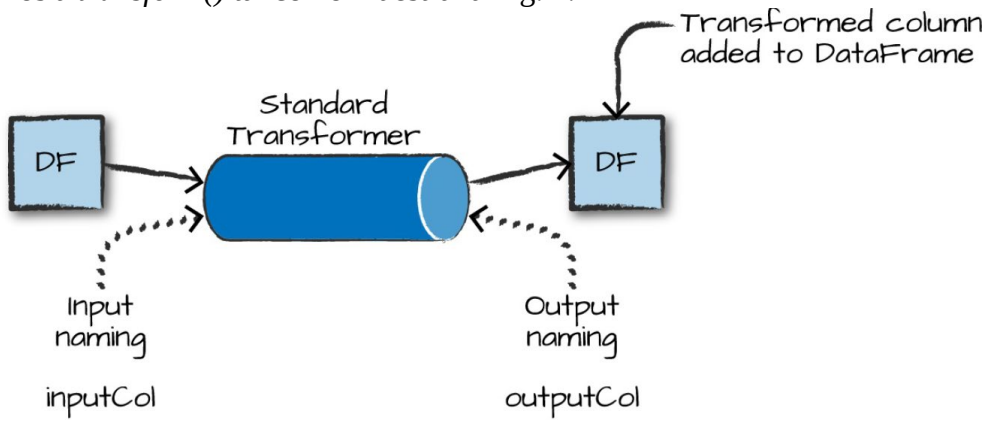


Fig 2. Comportamiento de un transformador en Spark.
Imagen tomada de Spark: The Definitive Guide (2018)

Transformadores más habituales

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. Ejemplo:

```
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)],
```

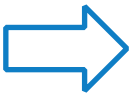
```
      ["id", "hour", "mobile", "userFeatures", "clicked"])

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]
```

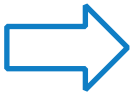
SQLTransformer: recibe una sentencia SQL de tipo *SELECT* como string, y la aplica al DF que se le pase como argumento a *transform()*. Útil para dejar fijadas series de transformaciones arbitrarias que luego sean incluidas en un pipeline.

```
from pyspark.ml.feature import SQLTransformer

df = spark.createDataFrame([
      (0, 1.0, 3.0),
      (2, 2.0, 5.0)
], ["id", "v1", "v2"])

sqlTrans = SQLTransformer(
      statement = "SELECT *, (v1 + v2) AS v3,
                   (v1 * v2) AS v4 FROM __THIS__"
                  )
sqlTrans.transform(df).show()
```

```
 id |  v1 |  v2
----|-----|-----
 0  | 1.0 | 3.0
 2  | 2.0 | 5.0
```



```
 id |  v1 |  v2 |  v3 |  v4
----|-----|-----|-----|-----
 0  | 1.0 | 3.0 | 4.0 | 3.0
 2  | 2.0 | 5.0 | 7.0 |10.0
```

Bucketizer: es un discretizador. 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")

# Transformamos los datos originales en el índice de su intervalo
bucketedData = bucketizer.transform(dataFrame)

# Obtenemos la longitud del vector de límites de intervalos
nBuckets = len(bucketizer.getSplits())-1)
print("Bucketizer output with %d buckets" % nBuckets)

bucketedData.show()

+--------+----------------+
|features|bucketedFeatures|
+--------+----------------+
|  -999.0|             0.0|
|    -0.5|             1.0|
|    -0.3|             1.0|
|     0.0|             2.0|
|     0.2|             2.0|
|   999.9|             3.0|
+--------+----------------+
```

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, que también es un *Transformer* por lo que es capaz de *transformar* (hacer predicciones) 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.

- Las predicciones se añaden junto a cada ejemplo en una nueva columna
- Para facilitar que se mantenga el mismo formato, se suele entrenar un pipeline completo y utilizar su salida (*pipeline entrenado*) como transformador

```
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)
lrModel = lr.fit(training)

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

from pyspark.ml import Transformer
```

```
isinstance(lrModel, Transformer) # Comprobamos que el modelo ajustado
True                             # pertenece a la clase Transformer

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|
+-------------------+--------------------+--------------------+
```

Estimadores en Spark ML

Un *Estimator* es un objeto de Spark capaz de realizar transformaciones que primero 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 de datos que se desea transformar.

La interfaz *Estimator* tiene un único método: *fit(df: DataFrame)* que recibe un DataFrame y devuelve un objeto de tipo **Model*, el *modelo entrenado*, que es además un *Transformer* tal como explicamos anteriormente ("cualquier modelo entrenado").
Es importante notar que Spark llama *modelo* a cualquier cosa que requiera un *fit* previo, no sólo a los algoritmos de aprendizaje automático que conocemos como modelos propiamente.

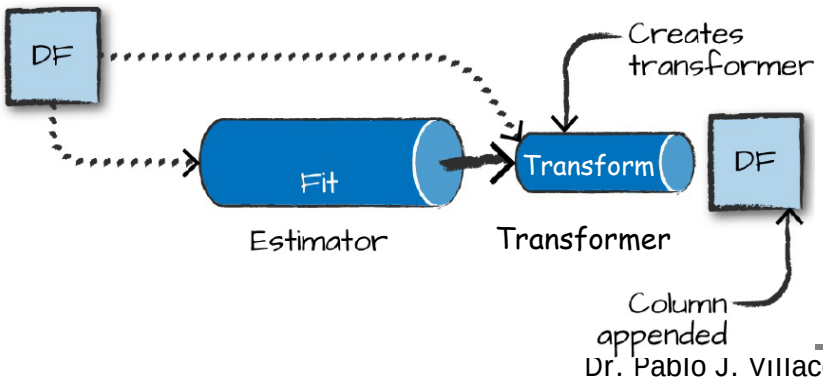


Fig. 3. Estimador que genera un transformador.
Tomado de Spark: The Definitive Guide (2018)

Estimadores más 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 números reales (*Double*) con la parte decimal a 0 y 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 como cualquier otra columna de números reales)

- 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 usar un modelo de machine learning para predecir ejemplos nuevos, primero hay que codificar sus variables categóricas siguiendo exactamente la misma codificación que se utilizó con los datos de entrenamiento con los que se entrenó el modelo. Por eso cobra utilidad la estructura de *Pipeline* que comentaremos más adelante.

```
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")
indexed = indexer.fit(df).transform(df)
indexed.show()
```

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

```
# otra opción: guardo el transformer para usarlo después
indexerModel = indexer.fit(df)
indexed = indexerModer.transform(df)

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

***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* 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

```
from pyspark.ml.feature import OneHotEncoderEstimator

df = spark.createDataFrame([
     (0.0, 1.0, 2.0),
     (1.0, 0.0, 3.0),   # Spark asume que la 3ª col tiene 5 categorías
     (2.0, 1.0, 2.0),   # porque el máximo valor que aparece es 4.0
     (0.0, 2.0, 1.0),  # y asume que vienen de una indexación previa
     (0.0, 1.0, 4.0),  # con StringIndexer y por tanto empiezan en 0.0
     (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)

from pyspark.ml.linalg import Vectors, VectorUDT  # convertir vectores
                                                  # sparse a dense
                  # porque el show() de sparse se ve peor en pantalla

from pyspark.sql import functions as F
toDenseUDF = F.udf(lambda r: Vectors.dense(r), VectorUDT())

encoded.withColumn("categoryVec1", toDenseUDF("categoryVec1"))\
...      .withColumn("categoryVec2", toDenseUDF("categoryVec2"))\
...      .withColumn("categoryVec3", toDenseUDF("categoryVec3"))\
...      .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]|
+--------------+--------------+--------------+-------------+-------------+-----------------+
```

***Cualquier modelo de predicción** (machine learning)*: 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).

- En el caso de los algoritmos de Machine Learning, los parámetros que se pueden ajustar *antes* de entrenar el modelo (aparte de los nombres de columnas necesarios) se denominan *híper-parámetros* y afectan a la manera en la que se entrenará el modelo. Por ejemplo, el híper-parámetro que controla la fuerza de la regularización (lambda), el número de iteraciones máximo del algoritmo de descenso en gradiente que se aplicará para minimizar el error al aprender los parámetros del modelo, o el número de árboles que se ajustarán en un algoritmo RandomForest.

PIPELINES

Es frecuente en Machine Learning extraer características de datos en crudo ("raw") y prepararlas antes de llamar a un algoritmo de aprendizaje. Sin embargo, puede ser difícil tener control de todos los pasos de pre-procesamiento que hemos hecho al entrenar, para luego replicarlos en otros conjuntos de datos o en el momento de hacer predicciones para nuevos datos con el modelo entrenado. Por 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.

Spark nos proporciona un mecanismo para esto.

Pipeline: secuencia de etapas (PipelineStage, superclase común 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)

Un Pipeline es un *Estimator*. El método *fit(df)* recorre cada etapa, **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 Es importante recordar que un mismo objeto (sea un estimador o un transformador) no puede ser añadido como a dos pipelines diferentes.

Pipeline antes de llamar a *fit*:

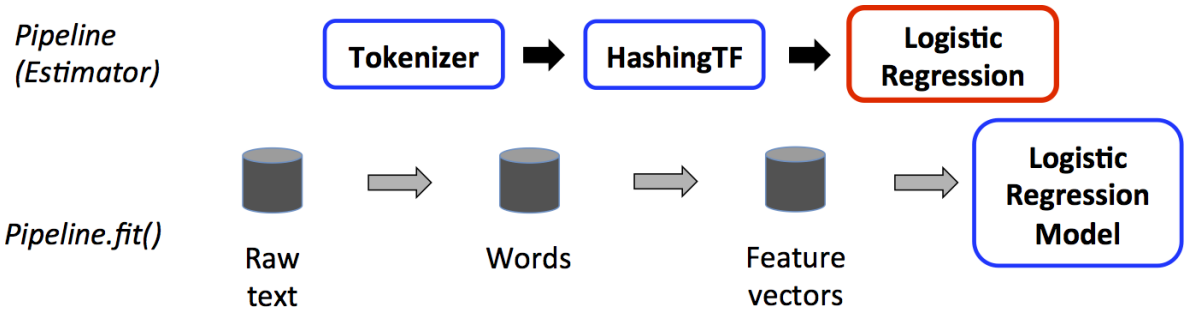


Fig. 4. Pipeline sin ajustar (arriba), y procesamiento que ocurre al llamar a fit (abajo). Imagen tomada de la documentación oficial de Spark.

Las etapas en azul son transformadores mientras que las rojas son estimadores. A continuación vemos el objeto PipelineModel (pipeline ajustado), donde las etapas que eran estimadores han pasado a ser transformadores. Si llamamos al método *transform()* sobre un PipelineModel, éste irá llamando a *transform* para cada etapa, pasando como argumento el DF resultante del *transform* de la etapa previa.

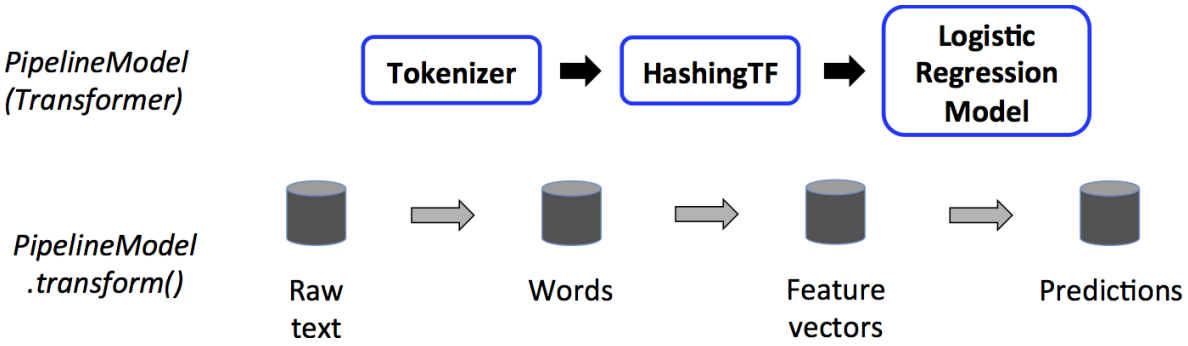


Fig. 5. Pipeline ajustado (PipelineModel, arriba), y procesamiento que ocurre al llamar a transform (abajo). Imagen tomada de la documentación oficial de Spark.

Lo habitual es llamar a *fit* sobre el objeto Pipeline una sola vez con los datos de entrenamiento. Esto devuelve un objeto PipelineModel (modelo ajustado), sobre el que podemos llamar a *transform* tantas veces como queramos sobre conjuntos de datos nuevos (nunca vistos por el modelo, pero contengan las columnas que esperan cada una las etapas) para realizar predicciones sucesivamente.

Ejemplo:

```
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)
```

BIBLIOGRAFÍA RECOMENDADA

- Bill Chambers y Matei Zaharia. *Spark: The Definitive Guide* O'Reilly, 2018.
- La web oficial de Spark, https://spark.apache.org/docs/latest/ contiene una extensa documentación.