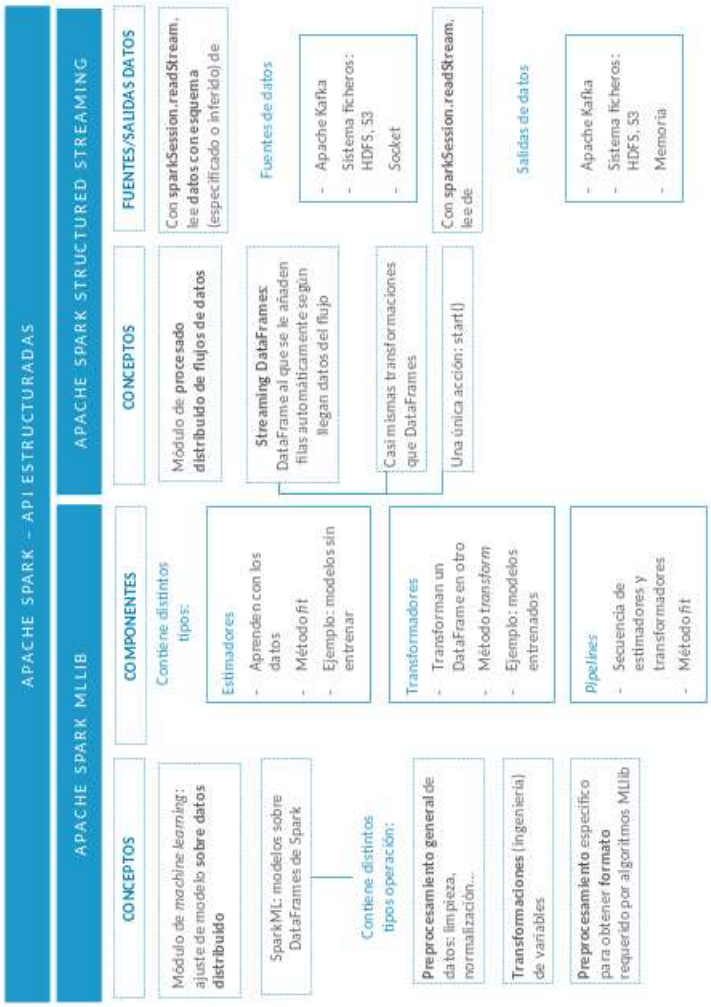
Ingeniería para el Procesado Masivo de Datos

Tema 5. Spark III

Índice

Esquema



5.1. Introducción y objetivos

Una vez expuestas las características fundamentales de Spark, en este tema nos detendremos en los módulos que proporcionan funcionalidad de más alto nivel. Empezaremos describiendo Spark MLlib y explicaremos los métodos básicos que ofrece para ajustar modelos; a continuación, examinaremos el módulo Spark Streaming para procesar datos en tiempo real. Con esto cerraremos el capítulo dedicado a Spark. Finalizaremos el tema presentando Apache Hive, otra herramienta del ecosistema Hadoop para hacer consultas SQL sobre datos distribuidos, y que puede utilizar como motor de ejecución MapReduce, Spark o Tez indistintamente. Con todo ello, los objetivos que persigue este tema son:

- Introducir al alumno en los módulos de alto nivel de Spark.
- Presentar las capacidades de Spark MLlib mediante ejemplos de uso que servirán de base para desarrollar la actividad entregable.
- Mostrar las ideas básicas y un ejemplo sencillo de Spark Structured Streaming.

5.2. Spark MLlib

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. Estos usan el modelo entrenado para realizar predicciones sobre un nuevo ejemplo, el cual reciben 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. Dichos modelos son capaces de aprender sobre datasets muy grandes almacenados de manera distribuida. No obstante, también es muy frecuente utilizar solo ciertas funcionalidades de MLlib para preprocesar variables en *datasets* masivos, limpiar, normalizar, etc., y, finalmente, para filtrar y pasar el *dataset* resultante (asumiendo que ya no es masivo) al *driver*, a fin de 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 en formato texto o CSV, entre otros, y de usarlos como entrada para un algoritmo de aprendizaje automático.

La figura 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 preprocesamiento 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 preprocesamiento incluye, además de las operaciones típicas de un proyecto de *data science,* cierto procesamiento específico de Spark. La finalidad es 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.

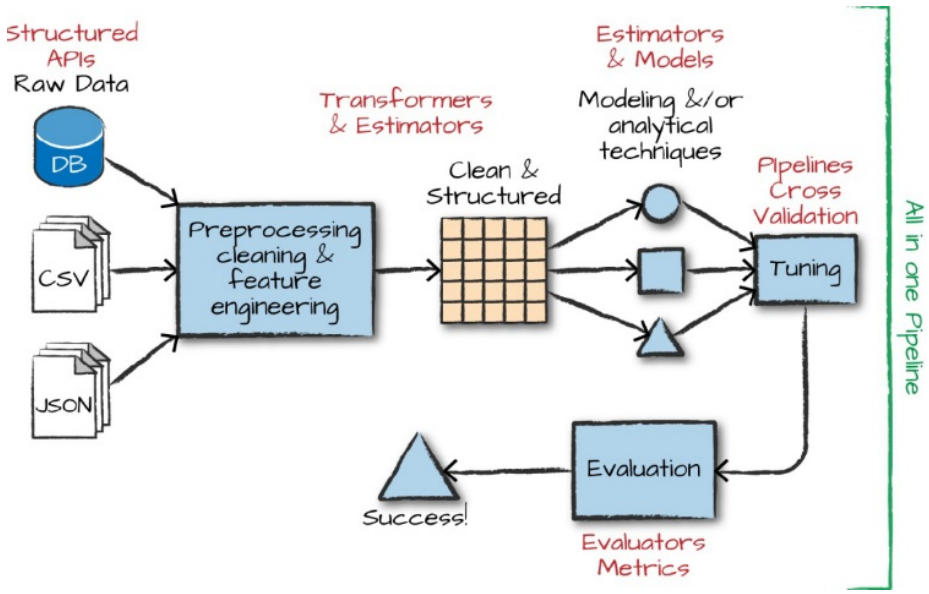


Figura 1. Etapas en el proceso de entrenamiento de un modelo a partir de datos. Fuente: Apache Spark 2.4.5.

Despliegue de modelos en producción con Spark

Spark no fue concebido para explotación *online* de modelos entrenados, es decir, para 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 conjuntos de datos muy grandes.

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

- **Entrenar con datos *offline* almacenados en HDFS** (el proceso de creación de variables y entrenamiento llevará cierto tiempo) y, justo después, usar el modelo entrenado para predecir (también en modo *batch*) otro conjunto de datos nuevos. La etapa de predicción es mucho más rápida que la de entrenamiento.
  - Es un enfoque muy frecuente. Es posible cuando los datos que hay que predecir (excepto la columna objetivo) ya se conocen en el momento de entrenar, por ejemplo, series temporales para predecir una ventana a futuro.
  - Las predicciones precalculadas se almacenan en HDFS o en bases de datos indexadas para que sea muy rápido servirlas desde un microservicio.
- También es posible hacer una predicción en *batch* en otro momento que no sea inmediatamente después del entrenamiento, sino cuando tengamos suficientes datos nuevos (un nuevo lote, considerable) sobre los que predecir.
- Entrenar, guardar el modelo entrenado y usarlo desde Spark para hacer predicciones una a una. Poco recomendable: lanzar un *job* para cada ejemplo que predecir implica sobrecarga. Balanceo de carga con réplicas del modelo.
- Entrenar y exportar el modelo a un formato de intercambio. Ejemplo: PMML, para leerlo y explotarlo con otra herramienta no distribuida (Python en especial, aunque también R soporta archivos en formato PMML).
- Entrenar en modo *online* usando Structured Streaming para recoger datos. Exige reentrenar desde cero, salvo que el algoritmo esté preparado para entrenamiento incremental (modelos de *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 RDD 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 preprocesamiento 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:

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

Tabla 1. Problemas de clasificación (clase codificada como número real).

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

Tabla 2. Problemas de regresión (el *target* ya es un número real).

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

Tabla 3. Problemas de *clustering* (no existe columna *label*).

user	*item*	*rating*
2	3	4.5
1	19	3.21

Tabla 4. Problemas de recomendación (con filtrado colaborativo).

Como vemos, en todos los algoritmos, 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 (sea de clasificación o de 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, que ha de empezar en 0.0 y con la parte decimal del número siempre a 0 (es decir, si tenemos un problema con cinco clases, se tienen que codificar como 0.0, 1.0, 2.0, 3.0 y 4.0).

En relación con 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.

Si hablamos de algoritmos de recomendación, Spark solo permite filtrado colaborativo. 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 dicho 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.

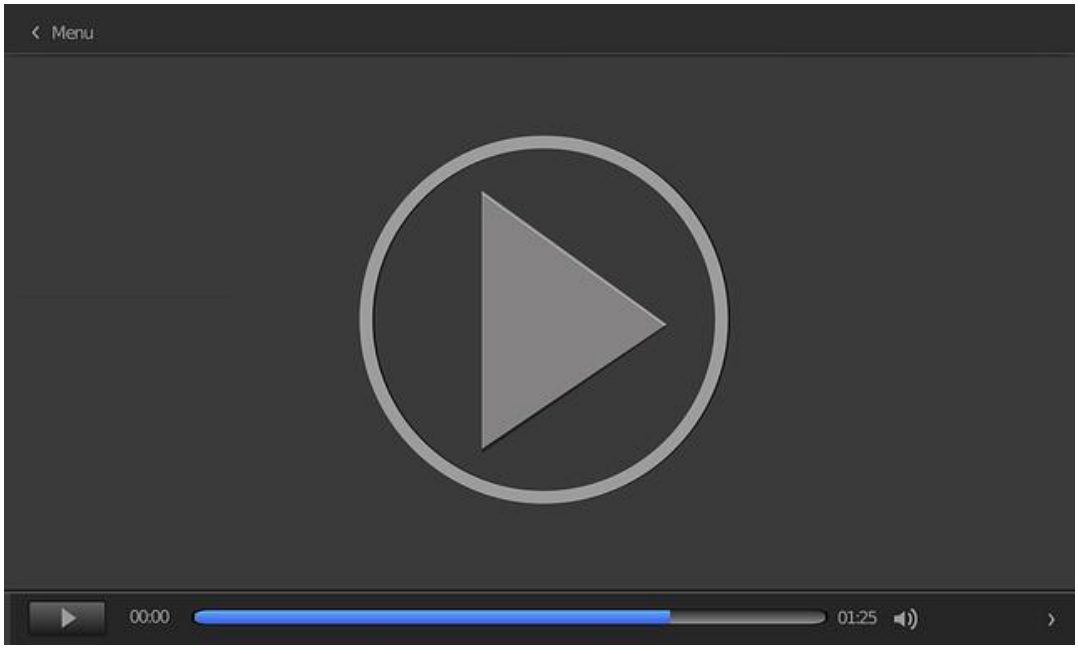
Según la documentación oficial de Spark, actualmente nos ofrece las siguientes posibilidades para preprocesamiento, divididas en varios grupos:

- Extracción: extraer variables a partir de datos en crudo.
- Transformación: reescalar, convertir o modificar variables.
- Seleccionar: seleccionar un buen subconjunto de variables de otro más grande.
- *Locality sensitive hashing* (LSH): combinación de transformaciones de variables con otros algoritmos.

Para más detalle consultar la documentación de Spark: https://spark.apache.org/docs/latest/ml-features.html

---

Para finalizar este apartado, en el siguiente vídeo, vamos a mostrar las capacidades de Spark ML en un problema de análisis de sentimiento.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=1007c11e-1c7e-443a-9dbb-b01100a57e6f

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 los transforma de alguna manera. Su salida es el mismo DataFrame con una nueva columna añadida a la derecha, con el nombre que hayamos indicado en el parámetro correspondiente (generalmente, *outputCol,* pero, a veces, puede ser *predictionCol*).

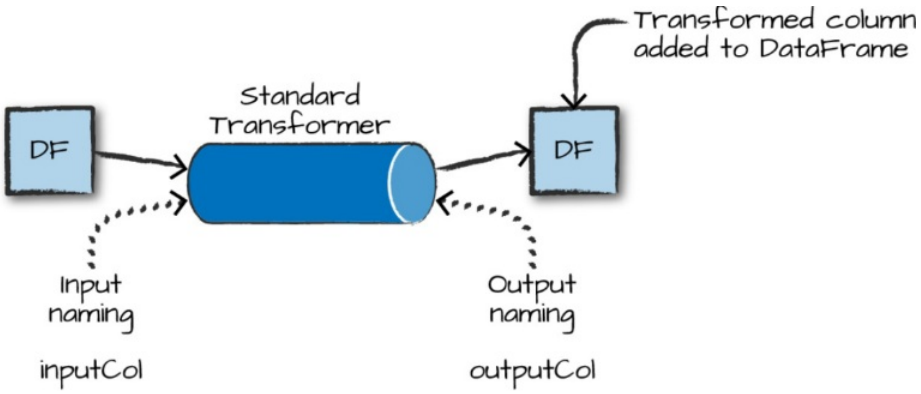


Figura 2. Funcionamiento de un transformador de Spark ML. Fuente: Apache Spark 2.4.5.

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 figura 2.

Algunos transformadores 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", "mobi1e", "userFeatures", "c1icked"])

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]
```

- **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. 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
from pyspark.ml import Transformer

training = spark.read.format("csv")\
                .load("sample_linear_regression_data.csv")

lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(training)
lrModel.__class__  # comprobamos la clase del modelo ajustado
 # Devuelve: <class 'pyspark.ml.regression.LinearRegressionModel'>

pred = lrModel.transform(training)
pred.show()
```

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 precisan 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. Es importante notar que Spark llama *modelo* a cualquier cosa que requiera un *fit* previo, no solo a los algoritmos de aprendizaje automático que conocemos como modelos propiamente.

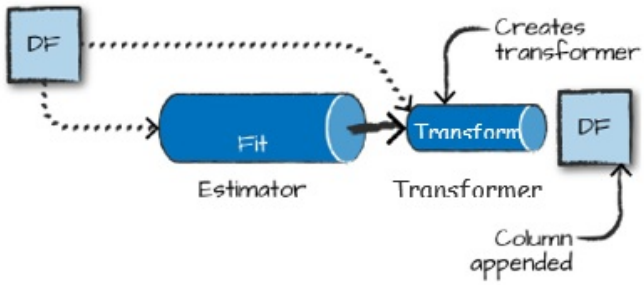


Figura 3. Estimador que genera un transformador. Fuente: Chambers y Zaharia, 2018.

Estimadores más comunes

- **StringIndexer*:*** estimador para preprocesar variables categóricas. Es el más utilizado. Convierte una columna (de cualquier tipo, ya que los valores serán interpretados como categorías) en números reales (*double*), con la parte decimal a 0 y empezando en 0.0. Las categorías se representan mediante 0.0, 1.0, 2.0, etc.

- Además, añade metadatos al DataFrame transformado devuelto por transform(), con los que indica que esa columna es categórica y no como cualquier otra columna numérica. Esta información es útil para los algoritmos.

- Los algoritmos que sí soportan variables categóricas (ejemplo: DecisionTree, RandomForest, GradientBoostedTrees) requieren que estas las pasemos indexadas.

- Los algoritmos que no soportan variables categóricas (LinearRegression, LogisticRegression) requieren el uso de OneHotEncoder, tal como veremos.

Es importante recordar que, al emplear un modelo entrenado 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 sentido 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"])
```

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

```
indexer = StringIndexer(inputCol =
  "category", outputCol = "categoryIndex")
indexed = indexer.fit(df).transform(df)
indexed.show()

# Guardo el transformer ajustado (indexerModel) para usarlo después:
indexerModel = indexer.fit(df)
indexed = indexerModer.transform(df)

# ... resto del código de nuestro programa. Ahora cargamos nuevos
# datos y los codificamos siguiendo exactamente la misma codificación:
indexedNuevo = indexerModel.transform(datosNuevosDF)
```

Ideas clave

- **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 variables (condensadas en una sola columna de tipo vector), donde, para cada ejemplo, solo una de las n variables tiene valor 1 y el resto son 0. Esto indica 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, la columna de entrada debe estar constituida por 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),  # también 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)

# La siguiente línea se utiliza porque vamos a convertir vectores
# sparse a dense, ya que el show() de sparse se ve peor en pantalla

from pyspark.ml.linalg import Vectors, VectorUDT
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*** y 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 *hiperparámetros* y afectan a la manera en la que se desarrolla dicho entrenamiento. Por ejemplo, el hiperparámetro que controla la fuerza de la regularización (lambda), el número de iteraciones máximo del algoritmo de descenso en gradiente que se aplicará para aprender los parámetros del modelo o el número de árboles que se ajustarán en un algoritmo RandomForest.

Pipelines en Spark ML

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 preprocesamiento que hemos llevado a cabo al entrenar, para luego replicarlos de manera exacta en otros conjuntos de datos o en el momento de hacer predicciones para nuevos datos con el modelo entrenado. Por ejemplo, a la hora de procesar un documento, hemos de llevar a cabo los siguientes pasos:

- División en palabras.

Ideas clave

- 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, denominado ***pipeline***.

Pipeline de SparkML: secuencia de etapas (*estimator* o *transformer*) que se ejecutan en un cierto orden para ir transformando un DataFrame.

En un *pipeline* de Spark, 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) de un *pipeline* recorre cada etapa: **llama a** transform() **si la etapa es un *transformer* o a** fit(df **) y luego a** transform(df) **si es un *estimator,*** pasando siempre el DataFrame *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 del *pipeline* sea un algoritmo de *machine learning,* 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 etapa a dos *pipelines* diferentes.

Veamos un ejemplo. La siguiente figura muestra un *pipeline* antes de llamar a *fit*:

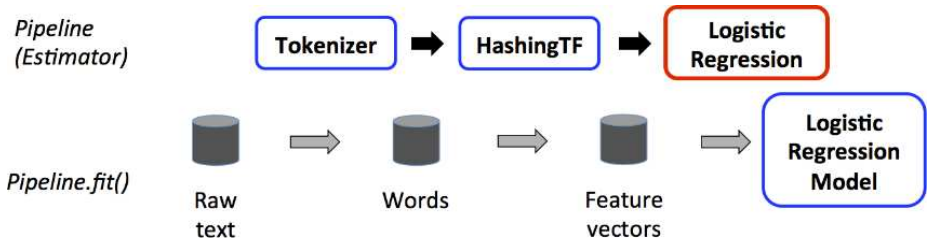


Figura 4. *Pipeline* sin ajustar (arriba) y procesamiento que ocurre al llamar a *fit* (abajo). Fuente: Apache Spark 2.4.5.

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 ejecutamos el método transform() sobre un PipelineModel, este irá llamando a *transform* para cada etapa y el DF de la etapa previa devuelto por *transform* pasará como argumento.

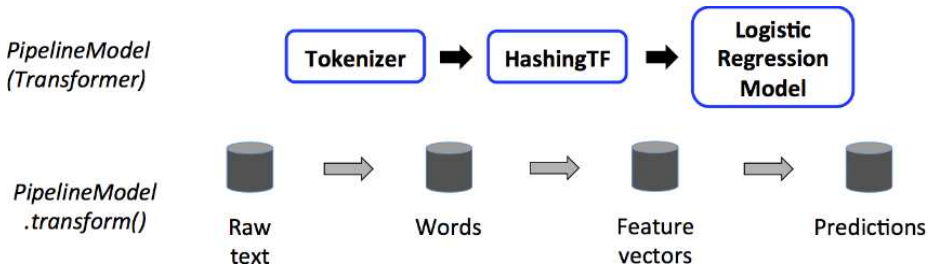


Figura 5. *Pipeline* ajustado (PipelineModel, arriba) y procesamiento que ocurre al llamar a *transform* (abajo). Fuente: Apache Spark 2.4.5.

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), que es un transformador y sobre el que podemos llamar a *transform* tantas veces como queramos, sobre conjuntos de datos nuevos (nunca vistos por el modelo, pero que contengan todas las columnas que esperan cada una las etapas), para realizar predicciones.

El siguiente ejemplo lee un conjunto de datos sobre vuelos y tiempo que han llegado retrasados en minutos, los separa en entrenamiento y test, y actúa sobre los datos de entrenamiento: primero indexa las variables categóricas y las fusiona en una única columna de tipo vector; después estandariza cada variable por separado, binariza la columna *target* (para convertir el retraso en minutos en una variable binaria: retraso *sí* o *no*) y ajusta un modelo de regresión logística para predecir si un vuelo tendrá o no retraso. Todas las etapas se añaden a un *pipeline* y se efectúa el procesamiento en el momento de llamar a `fit()` sobre los datos de entrenamiento. Una vez que tenemos el *pipeline* entrenado, lo aplicamos para predecir los datos de los test, los cuales seguirán exactamente las mismas etapas.

```
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]  # Dividimos los datos en train y test:
testingData = trainTest[1]   # 80 % para entrenar y 20 % para testear.

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("vetorizedFeatures")


# Normalizamos cada variable para tener media 0 y desviación típica 1:
scaler = StandardScaler().setInputCol("vectorizedFeatures")\
                         .setOutputCol("features")\
                           .setWithStd(True).setWithMean(True)
binarizer = Binarizer().setInputCol("ArrDelay")\
                                  .setOutputCol("binaryLabel")\
                                  .setThreshold(15.0)
# Algoritmo de machine learning (Estimator) para clasificación:
lr = LogisticRegression().setMaxIter(10)\
                           .setRegParam(0.3)\
                           .setElasticNetParam(0.8)\
                           .setLabelCol("binaryLabel")\
                           .setFeaturesCol("features")

lrPipeline = Pipeline().setStages([
     monthIndexer, dayofMonthIndexer, dayOfWeekIndexer,
     uniqueCarrierIndexer, originIndexer, assembler, scaler,
     binarizer, lr])
pipelineModel = lrPipeline.fit(trainingData) # ajustar modelos
lrPredictions = pipelineModel.transform(testingData)  # predecir
lrPredictions.select("prediction", "binaryLabel", "features").show(20)
```

Ideas clave

5.3. Spark Structured Streaming

En esta sección, veremos una breve introducción a Structured Streaming, el módulo de Spark que ha absorbido al obsoleto Spark Streaming, el cual usaba una estructura de datos llamada DStream, basada en RDD. Actualmente, Spark Structured Streaming utiliza los ***streaming* DataFrames**, que conceptualmente son iguales que un DataFrame, pero a los cuales **se les van añadiendo filas automáticamente, en tiempo real, según van llegando**. En realidad, no se implementa de esta manera, pero la analogía es válida para razonar sobre *streaming* DataFrames. Esto permite **disminuir la latencia** con respecto a un proceso *batch* que se ejecute periódicamente, ya que es capaz de hacer el cálculo incremental automáticamente en lugar de recalcular siempre todo el resultado partiendo de 0.

El procesamiento de flujos de datos (*stream processing*) consiste en incorporar continuamente nuevos datos para actualizar en tiempo real un resultado. Es decir, el cálculo se realiza agregando de alguna forma los datos nuevos a los ya existentes. Esta agregación puede incluir descartar en ciertos momentos los datos más antiguos para considerar solo los recibidos en una ventana temporal reciente. Podemos verlo recálculo continuo del resultado, en oposición a lo que ocurre en el procesamiento *batch*, en el que el cálculo se lleva a cabo una sola vez.

Structured Streaming se esfuerza por mantener una API idéntica a los DataFrames. De hecho, el mismo código que calcula la salida para un DataFrame convencional (estático) debería funcionar para un *streaming* DataFrame. Los conceptos de transformación y acción se mantienen, con una salvedad: la única acción disponible en Structured Streaming es la de comenzar un flujo (start()), que iniciará el cálculo y lo ejecutará indefinidamente, actualizando resultados periódicamente.

Fuentes de datos de entradas y salidas permitidas

Spark Structured Streaming permite leer la entrada como flujo de datos desde Kafka (lo veremos en el tema siguiente); desde un sistema de ficheros como HDFS o Amazon S3, en el que una fuente externa va añadiendo ficheros nuevos a un directorio en tiempo real, y desde una fuente *socket* usable solo para desarrollar y testear. La lectura se efectúa con el método `readStream` aplicado al objeto SparkSession: `spark.readStream`. Es importante recordar que, en Structured Streaming, **hay que activar explícitamente la opción de inferencia de esquema o bien especificar siempre el esquema del fichero de entrada**, independientemente del formato. Incluso si es un fichero Parquet, que ya contiene dentro el esquema, es necesario especificar este como argumento si no hemos configurado la propiedad `spark.sql.streaming.schemaInference` a true en las opciones de Spark. Si ya se dispone de una versión inicial de los datos guardada, se puede realizar una lectura previa a un DataFrame convencional (estático), obtener el esquema que se ha leído y usar dicho esquema para el *streaming* DataFrame.

La salida puede asimismo escribirse en Kafka, en ficheros y en otras salidas también restringidas para testeo y depuración como, por ejemplo, una salida a «memoria». El **modo de salida** es relevante en estos casos: ¿queremos solamente añadir información con la salida actualizada o reemplazar completamente el fichero de salida generado en cada actualización? Existen tres modos: **añadir**, **actualizar** y **reemplazo completo**.

Veamos un ejemplo de código sencillo. Supongamos que tenemos un directorio de HDFS donde otro proceso externo va creando ficheros en tiempo real, todos con la misma estructura. Cada nuevo fichero incluye información de retrasos sobre un grupo de vuelos que ha aterrizado en diversos aeropuertos recientemente. Asumimos un *dataset* que incluye información sobre los retrasos de vuelos, similar al de ejemplos anteriores. Queremos ver el retraso medio que sufren los vuelos en cada aeropuerto. El código para calcularlo será el mismo que si tuviésemos un

DataFrame estático, pero Spark Structured Streaming irá actualizando automáticamente el fichero de salida agregado en tiempo real.

```
# flights.parquet es una carpeta de archivos, todos con mismo esquema:
staticDF = spark.read.parquet("/path/to/flights/folder")
schema = staticDF.schema
# 1 para que se lea solamente un fichero de la carpeta en cada lectura:
streamingDF = spark.readStream.schema(schema)\
                     .option("maxFilesPerTrigger", 1)
\
                   .parquet("/path/to/flights.parquet")

# Usamos la API estructurada como haríamos con un DF convencional:
resultDF = streamingDF.where("delay > 15").groupBy("airport").count()

# Invocamos la única acción disponible. Nótese que writeStream no es
query = resultDF.writeStream\ # acción, sino que la acción es start()
            .queryName("retrasosPorAeropuerto")\ # ¡un nombre único!
            .format("memory")\ # salida a memoria (para pruebas solo)
             .outputMode("complete")\ # reemplazo completo
            .start()    # esto desencadena el cálculo en segundo plano

# IMPRESCINDIBLE para evitar que el driver finalice sin esperarnos:
query.awaitTermination()
```

En el código anterior, Spark procesará un fichero, hará la agregación (conteo) y volcará el resultado a la salida indicada, que, en el caso anterior, es memoria. Inmediatamente después de terminar de procesar un fichero, volverá a leer de la carpeta otro distinto. Se irán leyendo de uno en uno los ficheros de la carpeta, debido a que hemos configurado maxFilesPerTrigger a 1. Asumimos que existe un proceso externo que está añadiendo en tiempo real ficheros nuevos a esa carpeta y, por este motivo, Spark los va procesando también en tiempo real.

5.4. Referencias bibliográficas

Apache Spark 2.4.5. (s. f.). *Machine Learning Library (MLlib) Guide*.

Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.

Documentación oficial de Apache Spark

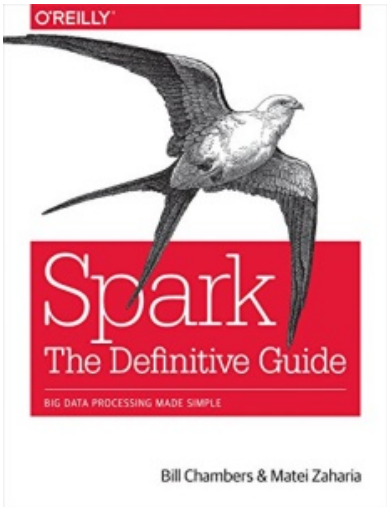
Apache Spark. Página web official: https://spark.apache.org/docs/latest/

Documentación *online*, detallada y de muy buena calidad sobre el sistema de computación Apache Spark.

Spark: the definitive guide

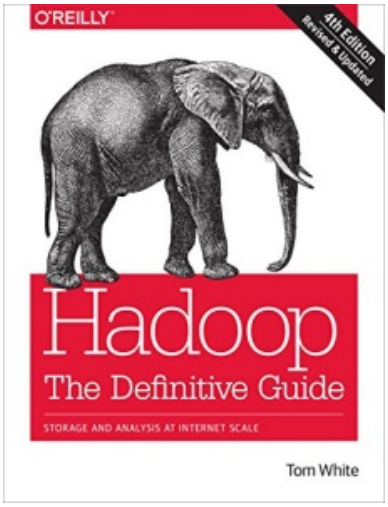
Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.

Guía detallada de Spark, en su versión más actualizada. Contiene numerosos ejemplos y muestra exhaustivamente todas las capacidades de Spark. Son relevantes los capítulos del 20 al 22, el 24 y el 25.



Hadoop: the definitive guide

White, T. (2015). *Hadoop: the definitive guide* (4. a edición). O'Reilly.



El capítulo 17 al completo está dedicado a Apache Hive.

Test

1. ¿Qué diferencia Spark MLlib de Spark ML?

A. Spark MLlib ofrece interfaz para DataFrames en todos sus componentes, mientras que Spark ML sigue utilizando RDD y ha quedado obsoleta.

B. Spark MLlib no permite cachear los resultados de los modelos, mientras que Spark ML sí.

C. Spark MLlib es más rápida entrenando modelos que Spark ML.

D. Ninguna de las respuestas anteriores es correcta.

2. ¿Qué tipo de componentes ofrece Spark ML?

A. Estimadores y transformadores para ingeniería de variables y para normalizar datos.

B. Estimadores y transformadores para preparar los datos para el formato requerido por los algoritmos de aprendizaje automático de Spark.

C. Solo *pipelines* que no dan acceso a los estimadores internos.

D. Las respuestas A y B anteriores son correctas.

3. ¿Cuál es el método principal de un *estimator* de Spark ML?

A. El método *fit.*

B. El método *transform.*

C. El método *estimate.*

D. El método *describe*.

4. ¿A qué interfaz pertenecen los algoritmos de *machine learning* de Spark cuando aún no han sido entrenados?

A. Transformer.

B. Estimator.

C. Pipeline.

D. DataFrame.

5. ¿A qué interfaz pertenecen los modelos de Spark ML cuando ya han sido entrenados con datos?

A. Transformer.

B. Estimator.

C. Pipeline.

D. DataFrame.

6. ¿Qué ocurre si creamos un StringIndexer para codificar las etiquetas de una variable en el *dataset* de entrenamiento y después creamos otro StringIndexer para codificar los datos de test en el momento de elaborar predicciones?

A. Obtendremos la misma codificación en los dos.

B. Da un error, porque un mismo StringIndexer no puede añadirse a dos pipelines.

C. Podríamos obtener codificaciones distintas de la misma etiqueta en los datos de entrenamiento y en los de test, lo que falsearía los resultados de las predicciones.

D. Ninguna de las respuestas anteriores es correcta.

7. ¿Cuál es la estructura principal que maneja Spark Structured Streaming?

A. DStreams.

B. DStreams DataFrames.

C. *Streaming* DataFrames.

D. *Streaming* RDD.

8. Spark Streaming permite leer flujos de datos:

A. Solo desde tecnologías de ingesta de datos como Apache Kafka.

B. Desde cualquier fuente de datos, siempre que contenga un esquema, como, por ejemplo, una base de datos.

C. Desde fuentes como Apache Kafka y HDFS, si activamos la inferencia de esquema.

D. Las respuestas A, B y C son incorrectas.

9. En Spark Streaming, una vez se ejecuta la acción *start*:

A. El *driver* espera automáticamente a que concluya la recepción de flujo para finalizar su ejecución.

B. Hay que ejecutar un método para indicar al *driver* que no finalice automáticamente y que espere a que concluya la recepción del flujo.

C. Un flujo de datos no tiene fin y, por tanto, el *driver* nunca puede finalizar.

D. Ninguna de las opciones anteriores es correcta.

10. ¿Qué acciones pueden realizarse en Spark Structured Streaming?

A. take.

B. show.

C. start.

D. collect.