Ingeniería para el Procesado Masivo de Datos

Dr. Pablo J. Villacorta

Tema 5. Apache Spark III

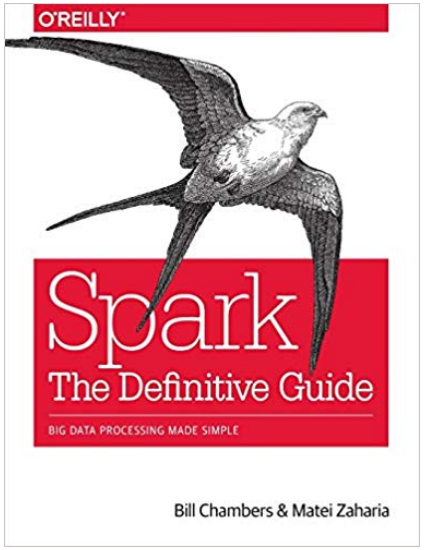
Diciembre de 2022

Objetivos del tema

- Entender cómo funciona el paquete Spark ML, uno de los grandes módulos que componen Spark
  - Comprender la diferencia respecto a mllib
- Introducir los conceptos fundamentales de Spark ML: *estimador, transformador* y *pipeline*
- Ver ejemplos prácticos de funcionamiento
- Entender la filosofía de Spark Structured Streaming
  - El mismo código que funcionaba en batch, convertido a streaming

Spark MLlib

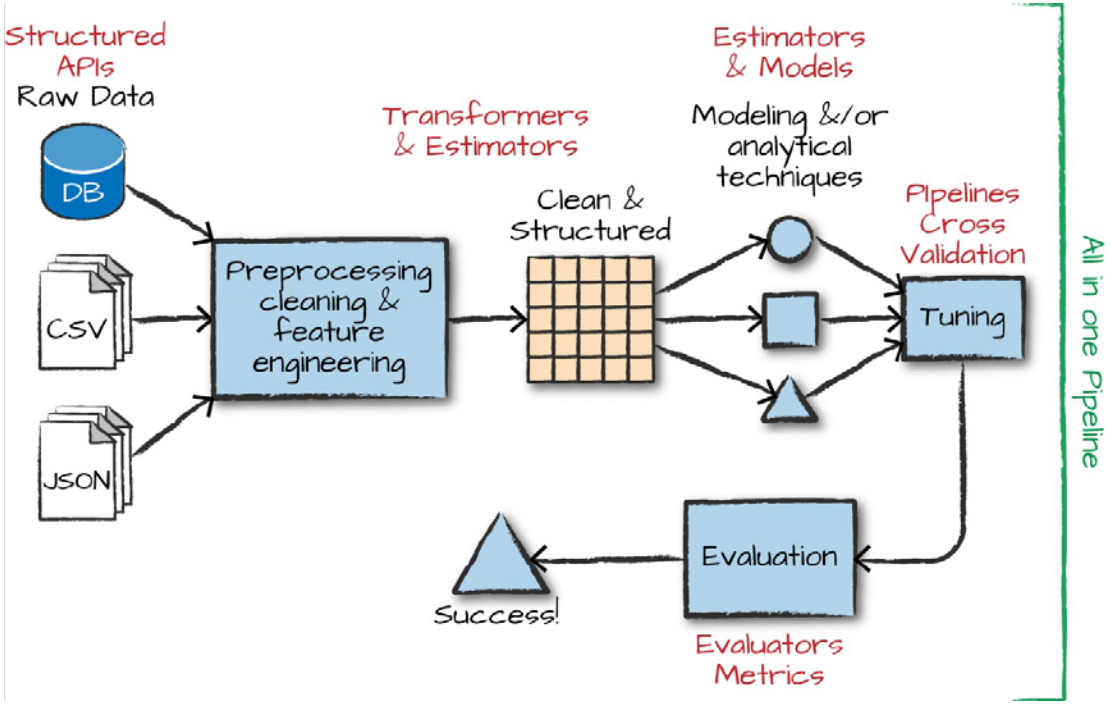

APACHE
Spark
TM


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia

El módulo Spark MLlib

- Spark MLlib: módulo de Spark para
  - 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 (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



Spark MLlib vs Spark ML

- En la API de Spark (Java/Scala/Python) se distinguen:
  - Paquete org.apache.spark.mllib (pyspark.mllib): API *antigua* basada en RDD de una estructura llamada LabeledPoint: *LabeledPoint(etiqueta, [vector de atributos]).* Obsoleta.
  - Paquete org.apache.spark.ml (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.

Ingeniería de variables y preprocesado

- Creación de variables desde datos raw: API habitual de Spark SQL
- Spark exige un formato de entrada concreto (y rígido) en los DataFrames para poder generar un modelo sobre ellos.
  - Por eso es necesario un pre-procesamiento adicional para preparar el DF para Spark ML después del habitual de limpieza de datos y creación de variables
  - Clasificación: la variable target debe ser *Double* empezando en 0.0
  - Las features del problema deben venir en una **única** columna de tipo vector

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.

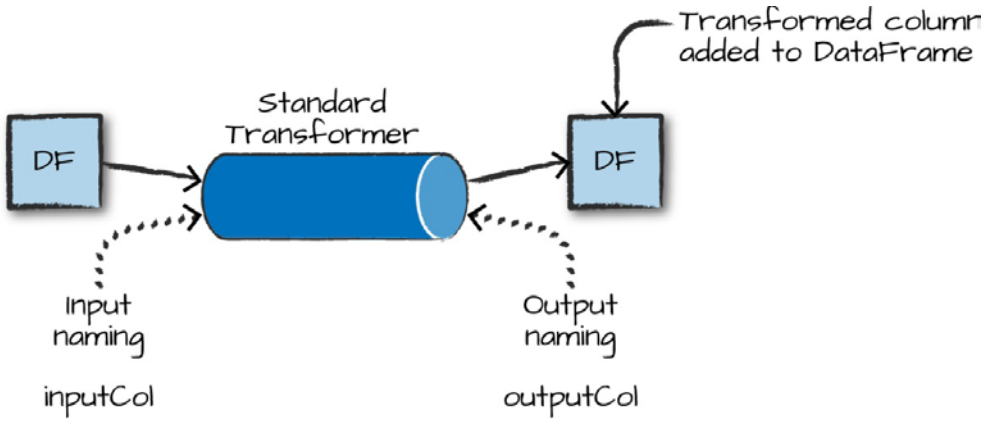
Table of Contents

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

Aunque el epígrafe sea *Feature Transformers*, no todos los que aparecen son *Transformers*… algunos son *Estimators*

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

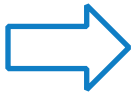
- Los transformadores *no tienen ningún parámetro que aprender* de los datos. Su salida es un DataFrame

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

Transformadores muy comunes

- *Binarizer*: transforma una columna continua en una columna categórica con dos *identificadores* 0.0 y 1.0.

```
from pyspark.ml.feature import Binarizer

continuousDataFrame = spark.createDataFrame([
    (0, 0.1),
    (1, 0.8),
    (2, 0.2)
], ["id", "feature"])

# El umbral por defecto es 0.0 si no indicamos nada
binarizer = Binarizer().setThreshold(0.5).setInputCol("feature")\
                       .setOutputCol("binarized_feature")

binarizedDataFrame = binarizer.transform(continuousDataFrame)

print("Binarizer output with Threshold = %f" % binarizer.getThreshold())
binarizedDataFrame.show()
```

```
Binarizer output with Threshold = 0.500000
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
+---+-------+-----------------+
```

Transformadores muy usuales

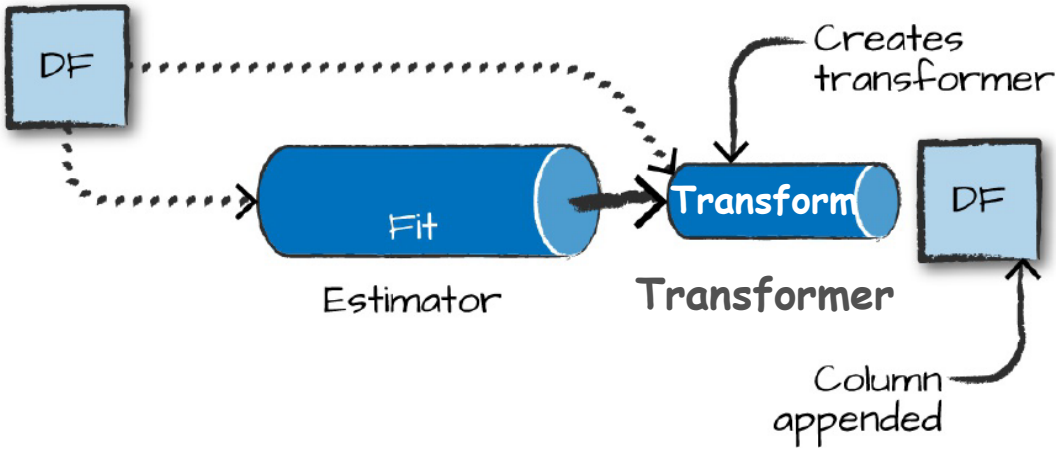
- *Cualquier modelo entrenado*: el resultado de entrenar un modelo sobre un DF es un objeto **Model* de la subclase específica del modelo que hayamos ajustado, que también es un *Transformer* por lo que es capaz de *transformar* (hacer predicciones) un DF de ejemplos, siempre que tenga el mismo formato (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

Transformadores muy usuales

```
%> from pyspark.ml.regression import LinearRegression
%> training = spark.read.format("libsvm")\
              .load("sample_linear_regression_data.txt")

%> 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)
True

%> pred = lrModel.transform(training)
%> pred.show()
+-------------------+-----------------+--------------------+
|              label|         features|          prediction|
+-------------------+-----------------+--------------------+
| -9.490009878824548|[1.81, -6.56, ...| 0.39922280427864854|
|         ...       |    ...          |         ...        |
|         ...       |    ...          |         ...        |
|-19.782762789614537|[0.32, -4.81, ...|  0.7839239258929726|
| -0.250102447941961|[1.09, -3.78, ...|  1.1299316224433398|
+-------------------+-----------------+--------------------+
```

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: e*stimador 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!

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

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

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* 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 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).

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)
>>> from pyspark.ml.linalg import Vectors, VectorUDT  # convertir vectores sparse a dense
>>> from pyspark.sql import functions as F      # porque el show() de sparse se ve peor en
>>> toDenseUDF = F.udf(lambda r: Vectors.dense(r), VectorUDT()) # para visualizar mejor
>>> 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]|
+-------------+-------------+-------------+-----------+-----------+-----------------+
```

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.

Pipelines

- 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).

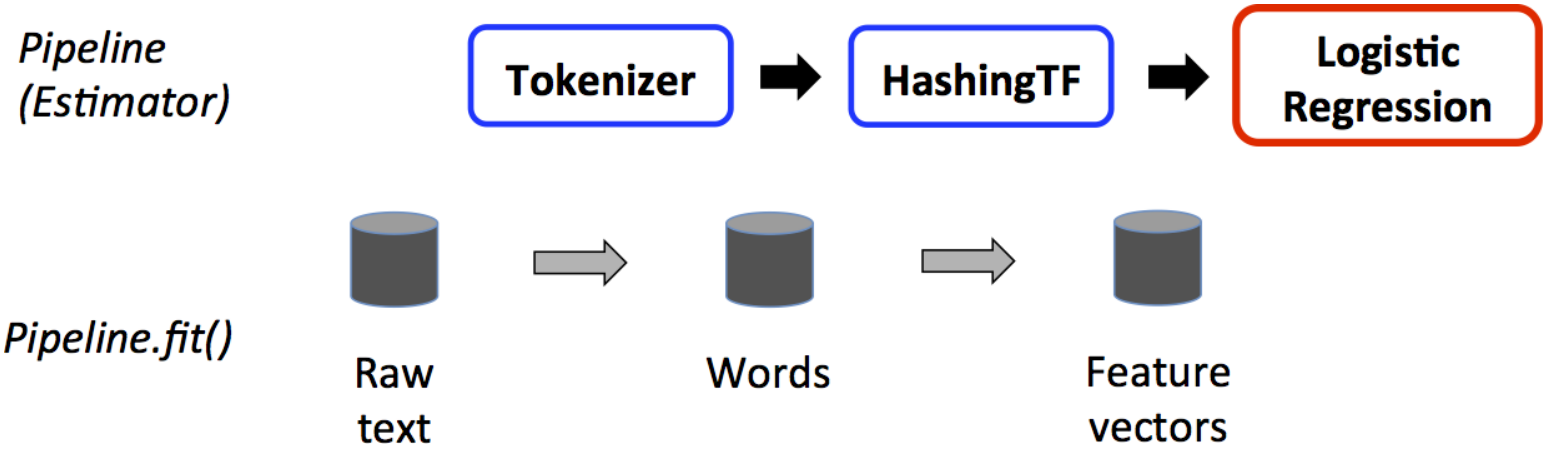
  - Es habitual (pero no obligatorio) que la última etapa sea un algoritmo de ML, aunque podría haber varios a lo largo de un pipeline

  - El objeto devuelto por fit() sobre un pipeline es un PipelineModel (objeto pipeline ajustado) que es un transformador, en el cual *todas las etapas se han convertido en transformadores* ya que todas han sido ajustadas

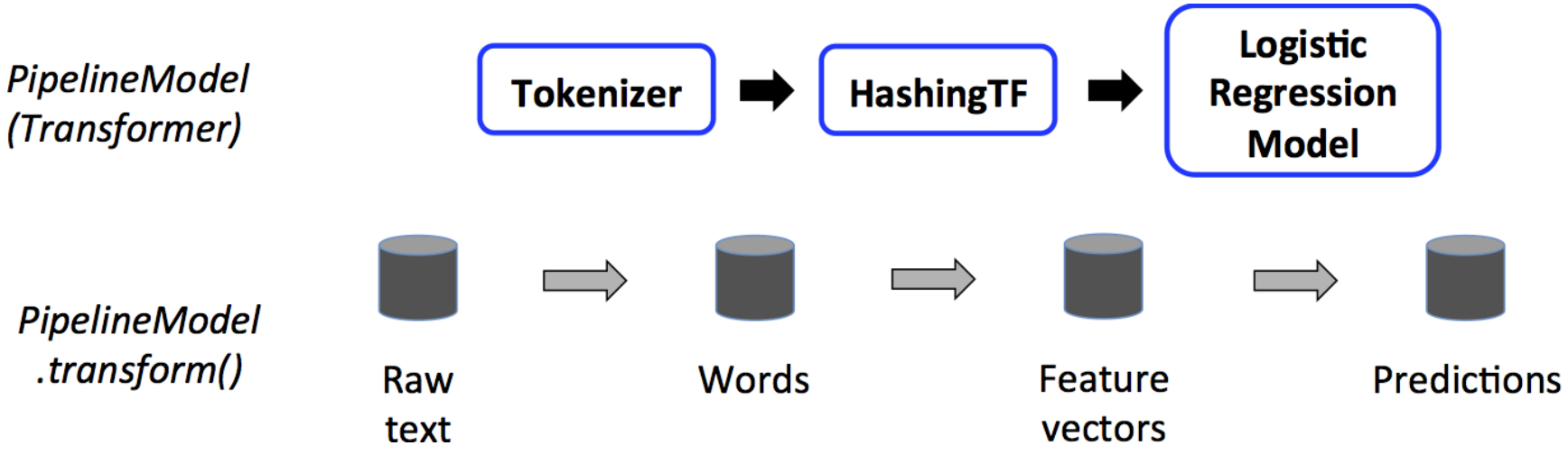
  - IMPORTANTE: un mismo objeto no puede ser añadido a dos pipelines

Pipelines

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



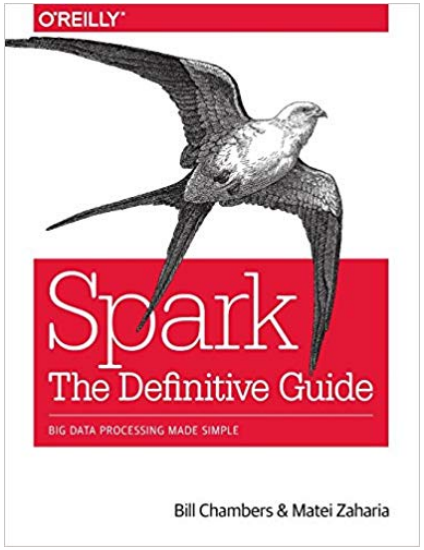
- PipelineModel (ajustado) devuelto por la llamada a *fit(df)*: todo son Transformers



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("features")

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, binarizerClassifier, lr])

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

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

Spark Streaming



El módulo Spark Streaming

- Es el acto de incorporar de manera continua nuevos datos para ir actualizando el resultado del cálculo
  - Los datos de entrada no tienen principio ni fin
  - Los datos son series de eventos que llegan a la aplicación

- El programa calcula nuevas versiones del resultado según van llegando nuevos datos
  - Spark automáticamente actualiza el resultado

- Spark Streaming: procesamiento en tiempo real de flujos de datos
- Dos APIs:
  - DStream API: RDDs (bajo nivel, muy poca ayuda)
  - **Structured Streaming**: la API Estructurada que ya hemos usado (DataFrames) aplicada a datos en streaming. Será la que usaremos.

Casos de uso para datos en streaming

- Alertas: un nuevo dato causa que se actualice cierto resultado y a consecuencia, se envíe una alerta a una persona

- Reporting en tiempo real: cuadros de mando construidos sobre agregaciones de datos que se actualizan en tiempo real

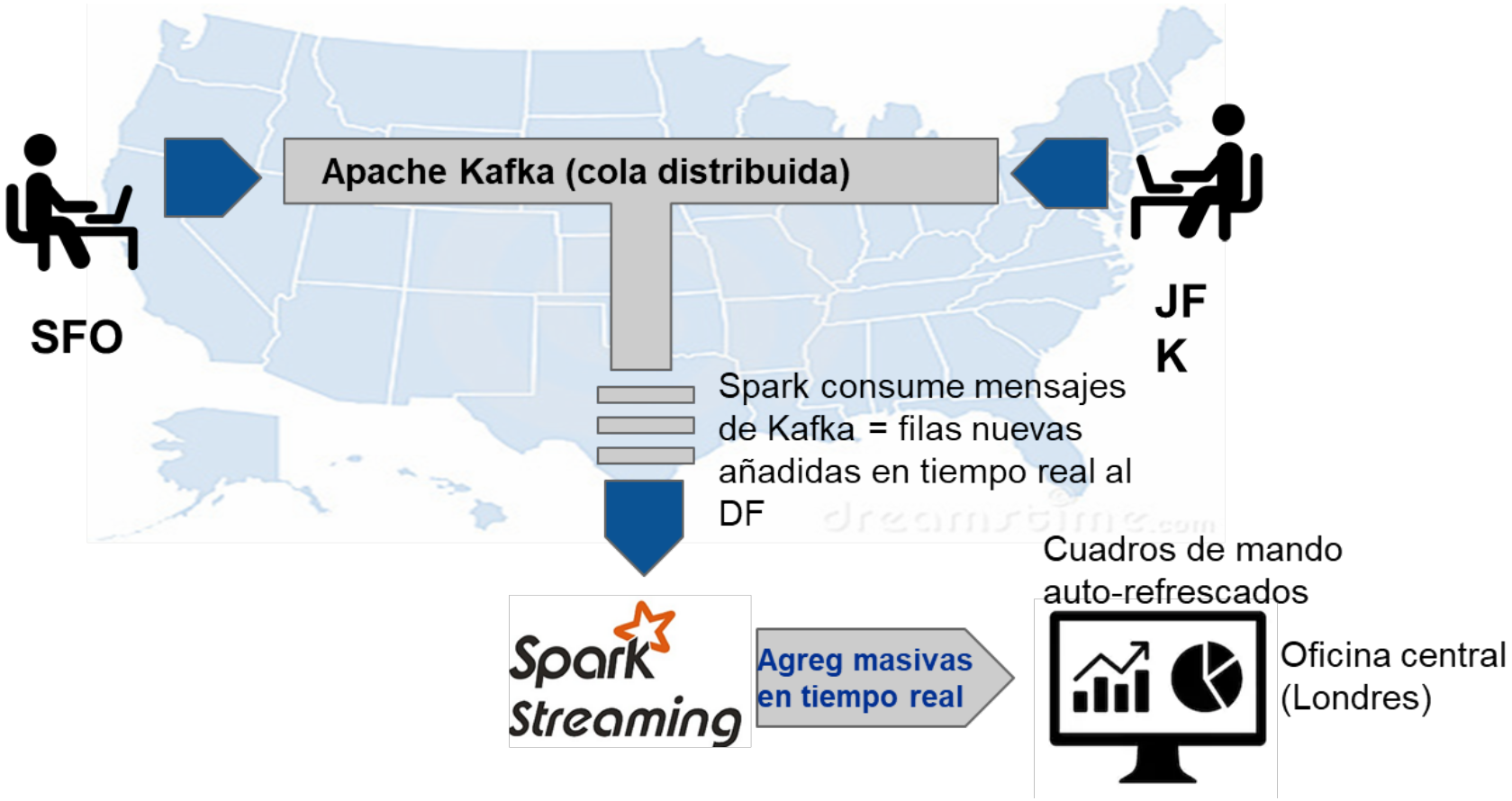
- ETL incremental: “mi trabajo batch pero en streaming”.
  - Limpiar y procesar datos en crudo según van llegando, para escribirlos a un data warehouse.

- Decisiones en tiempo real: ¿es fraudulenta esta transacción? Denegar sobre la marcha en caso de serlo
  - Usando reglas hardcodeadas, o un modelo de ML

Retos del procesamiento en streaming

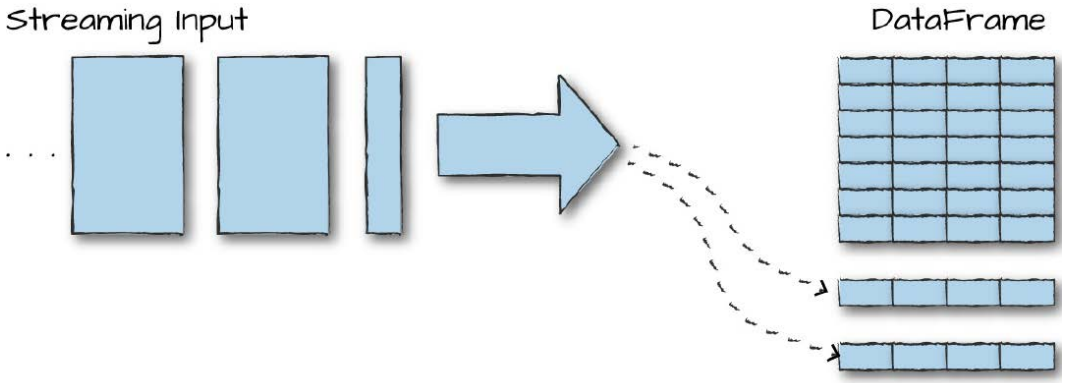
- Baja latencia (respuesta rápida a un solo evento)
- Alto throughput (procesar rápidamente un volumen alto de entrada. Se mide en salidas por unidad de t). Opuesto a baja latencia habitualmente.
- Procesar datos desordenados (recibidos en orden diferente al que se generaron)
- Mantener un estado interno para determinar qué está pasando
- Ser capaz de unir datos en streaming con datos batch históricos
- Robustez pero sin duplicar salidas

Casos de uso para datos en streaming



Structured Streaming

- Son microbatches (aunque hay una propuesta para soportar verdadero procesamiento continuo, aún sin hacer): esperamos intervalos de tiempo para formar un DataFrame
- Idea: misma API que la API Estructurada (DataFrames)
  - El mismo código batch, transformado a streaming!
- Conceptualmente, un DataFrame al que se le añaden filas en tiempo real
  - Se conoce como un **Streaming DataFrame**
  - El mismo código que funciona para un trabajo batch debería servir sin cambios en Structured Streaming
- Restricción: solo hay una acción disponible: arrancar un flujo que permanecerá ejecutando continuamente dando resultados (**start**)



Lectura de un StreamingDF desde Kafka

```
inputDF = spark.readStream\
  .format("kafka")\
  .option("kafka.bootstrap.servers", "192.168.99.100:9092")\
  .option("subscribe", "retrasos")\
  .load() # nos suscribimos al topic "retrasos" y leemos mensajes de él

processedDF = preprocesar(inputDF) # hacemos legibles los mensajes binarios

retrasoMedioStreamingDF = processedDF.withColumn(...) # transformaciones
consoleOutput = retrasoMedioStreamingDF\
                    .writeStream\
                    .queryName("nombreTabla")\
                    .outputMode("complete")\
                    .format("memory")\
                    .start()  # escribimos la salida como tabla de Hive en memoria

contenidoDF = spark.sql("select * from nombreTabla")
contenidoDF.show()
time.sleep(5)
contenidoDF.show()
contenidoDF.show()
```

…eso es todo por hoy…

:-)

Cualquier duda, consulta o comentario:

mensaje a través de la plataforma!

www.unir.net