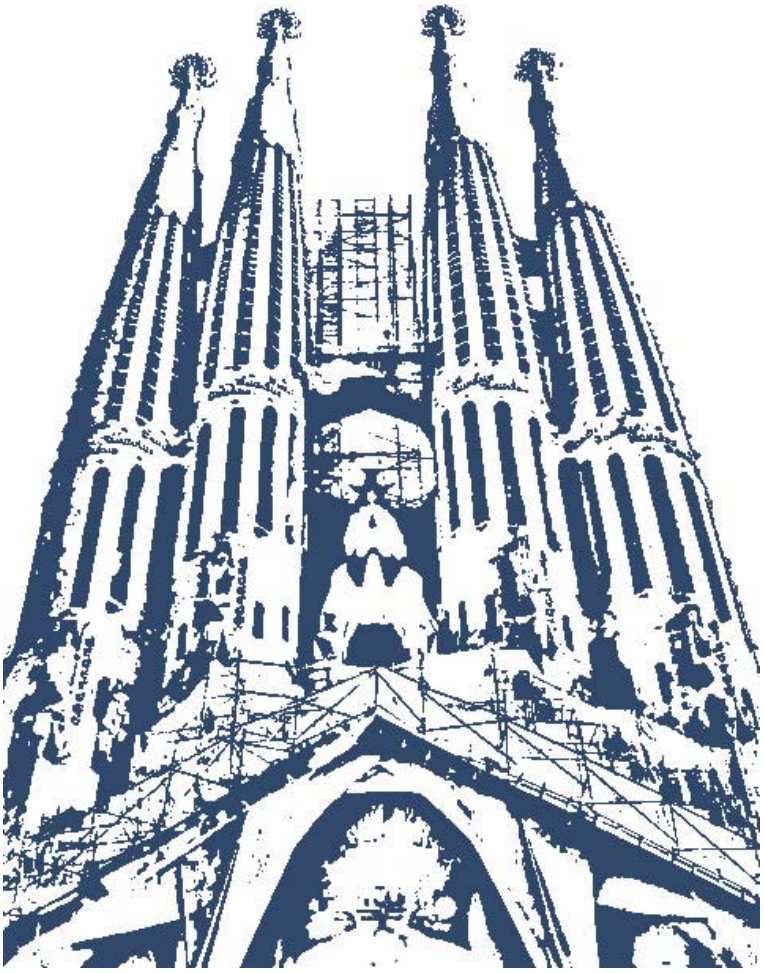

Introducción a Apache Spark

Mauro Gómez, Mario Macias, Rubèn Tous y Jordi Torres

Part III

Aprender haciendo

CAPÍTULO 10

Manos a la obra: clasificando tweets con MLlib

10.1 Contenido

Este hands-on se ha creado con el fin de ayudar a estudiantes, o cualquier otra persona interesada, a iniciarse con Apache Spark. Spark es una poderosa herramienta de código libre que permite procesar grandes cantidades de datos a muy alta velocidad y con una mucha facilidad de uso. Para realizar esta pequeña introducción vamos a usar Python 2.6 y Scala sbt v0.13.7 junto con Apache Spark v1.3.0.

10.2 Configuración

Para empezar a usar Apache Spark tenemos que descargarnos el entorno; tenemos disponible en la página oficial (`https://spark.apache.org/`) todos los ficheros necesarios. Con el fin de evitar la compilación, recomendamos descargar una versión *pre-built*. Una vez se disponga del fichero *tarball* tenemos que descomprimirlo y navegar hasta la carpeta *bin*, donde se encuentran todos los ejecutables. En esta carpeta están disponibles los *scripts* para acceder a la consola interactiva que ofrece Spark, tanto para Scala como para Python. Con el fin de que la ejecución sea más fácil para el usuario Spark proporciona un fichero de configuración, $SPARK_DIR\conf\log4j.properties , en el cual se pueden configurar difer-

entes parámetros del entorno. En este caso vamos a modificar la propiedad *log4j.rootCategory=INFO, console* para que nos muestre en la consola solo los mensajes de error, y podamos ver con más claridad, la ejecución de nuestro código. Por tanto, es necesario cambiar la propiedad para que, finalmente, quede tal que así: *log4j.rootCategory=ERROR, console* . Por último, es necesario indicarle a Spark la IP a la que queremos asociar el nodo, por tanto es necesario modificar el archivo $SPARK_DIR\conf\env.sh y configurar la variable *SPARK_LOCAL_IP=LOCALHOST*.

Ahora que ya tenemos el entorno configurado podemos iniciar la consola interactiva, tanto en Scala como en Python:

Iniciando consola virtual en Scala

```
$ ./bin/spark-shell
```

Iniciando consola virtual en Python

```
$ ./bin/pyspark
```

Ámbas consolas nos proporcionan un *Spark Context* mediante la variable **sc**. Este objeto es el que nos proporciona todas las funciones para interactuar con el núcleo de Spark. Para comprobar que todo funciona, vamos a realizar un pequeño código que cuente las palabras de un fichero, README.md por ejemplo, y nos diga cuantas repeticiones hay de cada una de ellas.

WordCount con Scala

```
val file = sc.textFile("README.md")
val count = file.flatMap(line => line.split(" ")).
                map(word => (word, 1).reduceByKey(_ + _)

count.saveAsTextFile("wc_out")
```

WordCount con Python

```
from operator import add
f = sc.textFile("README.MD")
wc = f.flatMap(lambda x: x.split(' ')).
                map(lambda x: (x, 1)).reduceByKey(add)

wc.saveAsTexFile("wc_out")
```

Una vez ejecutado, si revisas el directorio *\wc_out* podrás ver el resultado de la ejecución.

10.3 Ejemplo sencillo de agrupamiento

Para entrar un poco más en profundidad, veremos un ejemplo de agrupación simple. La agrupación, también conocida como *clustering*, es un problema de aprendizaje no supervisado; por el que nuestro objetivo es agrupar una serie de subconjuntos de entidades relacionados con alguna base de conocimiento. El agrupamiento se utiliza a menudo para el análisis exploratorio y/o como un componente de una estructura jerárquica del aprendizaje supervisado (en el que los clasificadores, o distintos modelos de regresión, son entrenados para cada grupo).

10.3.1 K-Means

El módulo MLlib ofrece una serie de métodos para que podamos usar *K-Means*, el algoritmo de agrupamiento más usado que nos ofrece la posibilidad de agrupar una serie de datos en un número predefinido de grupos. La implementación que ofrece MLlib incluye una variante del algoritmo tradicional, que ofrece paralelización en la ejecución, este método se llama *K-Means||*. La implementación de dicho método recibe los siguientes parámetros

- *k* el número de grupos deseados.
- *maxIterations* el número máximo de iteraciones.
- *initializationMode* especifica si la inicialización es aleatoria o vía *K-Means||*.
- *runs* número de veces que queremos ejecutar el algoritmo. Como *K-Means* no asegura encontrar una solución óptima global, se puede

ejecutar varias veces sobre el mismo conjunto de datos y el algoritmo devolverá la solución más optima encontrada.

- *initializationSteps* determina el número de pasos en el algoritmo.
- *epsilon* determina el umbral de distancia dentro de la cual consideramos que el algoritmo pueda haber convergido.

En los siguientes ejemplos, después de cargar y analizar los datos, usaremos el algoritmo *K-Means* para dividir los datos en cinco grupos. Una vez lo tengamos, calcularemos el error cuadrático (conocido como *Within Set Sum of Squared Error*). Este error se puede minimizar buscando una *k* óptima para el problema.

10.3.2 Cargando los datos en RDDs

Crea un fichero, *ejemplo.txt*, con el siguiente contenido:

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

Ahora ya podemos cargarlo en Spark como un objeto *RDD*:

Scala

```
val data = sc.textFile("ejemplo.txt")
```

Python

```
data = sc.textFile("ejemplo.txt")
```

Si queremos comprobar qué contenido tiene nuestro objeto *RDD*, podemos hacerlo de la siguiente manera:

Scala

```
data.foreach(println)
```

Python

```
def show (x): print x
data.foreach(show)
```

10.3.3 Convirtiendo texto a float

En esta sección, vamos a ver como podemos convertir el texto que hemos cargado de nuestro fichero, y convertirlo a *float* para poder tratarlo.

Scala

```
import org.apache.spark.mllib.linalg.Vectors
val parsedData = data.map(
        s => Vectors.dense(s.split(" ").map(_.toDouble))
).cache()

parsedData.foreach(println)
```

Python

```
from numpy import array
parsedData = data.map(
        lambda line : array([float(x) for x in line.split(' ')])
).cache()

parsedData.foreach(show)
```

Una vez se hayan ejecutado estos comandos, la variable *parsedData* contendrá un array con los datos en tipo *float*.

10.3.4 Entrenando el algoritmo

Ya disponemos de los datos en el formato correcto, ahora vamos a entrenar el algoritmo y computar el coste de error generado. Para esto, necesitamos importar una serie de librerías como *KMeans* o *sqrt*, que nos facilitarán las funciones para llevar a cabo estos cálculos. Lo primero que haremos será crear una instancia del objeto *KMeans* y agrupar los datos en dos categorías para luego calcular el error cuadrático que ha generado el algoritmo. El algoritmo estándar de *KMeans* tiene como objetivo minimizar la suma del cuadrado de las distancias entre los puntos de cada grupo, es decir la

distancia Euclídea al cuadrado. Como ya comentamos anteriormente, este error se puede disminuir buscando el valor óptimo de la variable *k*.

Para realizar el entrenamiento del algoritmo necesitamos ejecutar el siguiente código:

Scala

```
import org.apache.spark.mllib.clustering.KMeans

// Build the model (cluster the data)
val clusters = KMeans.train(parsedData, 5, 10)

# Evaluate clustering by computing
# Within Set Sum of Squared Errors
val WSSSE = clusters.computeCost(parsedData)

print("Within Set Sum of Squared Error = " + WSSSE)
```

Python

```
from pyspark.mllib.clustering import KMeans
from numpy import array
from math import sqrt

# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10,
runs=10, initializationMode="random")

# Evaluate clustering by computing
# Within Set Sum of Squared Errors
def error(point):
        center = clusters.centers[clusters.predict(point)]
        return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = parsedData.map(lambda point:error(point))
        .reduce(lambda x, y: x + y)

print("Within Set Sum of Squared Error = " + str(WSSSE))
```

10.4 Frecuencia de términos (TF) en textos

La frecuencia de términos, conocida como TF, es una método de vectorización utilizado en la minería de datos que refleja la importancia de cada término de documento. Si se quiere profundizar más sobre los algoritmos de TF, la siguiente página explica con detalle la implementación sobre Spark, y el funcionamiento del algoritmo. `https://spark.apache.org/docs/latest/mllib-feature-extraction.html#tf-idf`

El algoritmo TF está implementado en la librería *HashingTF* de Spark, y el método recibe como parámetro un *RDD* con una lista de los términos, y devuelve un objeto *SparseVector* con las frecuencias de cada uno de ellos.

10.4.1 Ejemplo con texto plano

Para comenzar tenemos que crear un fichero, *ejemplo2.txt*, con el siguiente contenido:

messi	barcelona	madrid	instagram	barcelona
barcelona	paella	playa	sangria	instagram
paella	barcelona	instagram	vermut	champions
ramblas	paella	ramblas	instagram	iniesta
messi	madrid	instagram	gol	porteria
barcelona	messi	instagram	twitter	balon
messi	gol	barcelona	instagram	arena
sangria	paella	instagram	agua	barcelona

10.4.2 Pre-procesado del texto

Para calcular el TF de nuestro ejemplo necesitamos, primero obtener un vector con las palabras de cada línea y luego llamar a la función que lo calcula pasándole dicho vector.

Scala

```
val documents = sc.textFile("ejemplo2.txt")
        .map(_.split(" ").toSeq)
documents.foreach(println)

...

WrapedArray(messi, barcelona, madrid, instagram, barcelona)

...
```

Python

```
documents = sc.textFile("ejemplo2.txt")
        .map(lambda line:line.split(" "))
documents.foreach(show)

...

[u'messi',u'barcelona',u'madrid',u'instagram',u'barcelona']

...
```

10.4.3 Calculo TF

Ahora que ya tenemos los datos preparados y en el formato que necesitamos, crearemos una instancia de la clase *HashingTF* para transformar los datos y obtener la frecuencia de los términos.

10.4.4 Pre-procesado del texto

Para calcular el TF de nuestro ejemplo necesitamos, primero obtener un vector con las palabras de cada línea y luego llamar a la función que lo calcula pasándole dicho vector.

Scala

```
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.feature.HashingTF

val hashingTF = new HashingTF()

val tf = hashingTF.transform(documents)
tf.foreach(println)

...

(1048576,[388745,503816,618478,929925],[1.0,1.0,2.0,1.0])

...
```

Python

```
from pyspark.mllib.feature import HashingTF

hashingTF = HashingTF()
tf = hashingTF.transform(documents)

def show (x): print x
tf.foreach(show)

...

(1048576,[388745,503816,618478,929925],[1.0,1.0,2.0,1.0])

...
```

En este instante, ya tenemos una variable, *tf*, que contiene un vector con la frecuencia de los términos del documento (p. ej. barcelona con frecuencia 2.0).

10.4.5 Clustering (KMeans)

Una vez ya tenemos los vectores con las frecuencias de cada palabra, vamos a usar el algoritmo *KMeans* para crear dos grupos con dichas frecuencias.

Scala

```
import org.apache.spark.mllib.clustering.KMeans

val clusters = KMeans.train(tf, 2, 1)
val results = documents.map(x => Vector(x,
        clusters.predict(hashingTF.transform(x))))

results.collect.foreach(println)

...

Vector(WrappedArray(barcelona,messi,instagram), 0)
Vector(WrappedArray(barcelona,paella,playa,sangria,instagram] 1]

...
```

Python

```
from pyspark.mllib.clustering import KMeans

clusters = KMeans.train(tf, 2, 1, 1)

results = documents.map(lambda x : array([x,
        clusters.predict(hashingTF.transform(x))]))

results.foreach(show)

...

[[u'barcelona', u'messi', u'instagram'] 0]

[[u'barcelona', u'paella', u'playa', u'sangria', u'instagram'] 1]

...
```

10.5 Frecuencia de términos (TF) con JSON

El formato JSON, o Javascript Object Notation, es un formato estándar que se usa un estilo fácilmente comprensible para transmitir objetos consistentes en pares clave-valor. Este formato se usa, principalmente, para transmitir datos entre servidores y aplicaciones web, y se ha postulado como una muy buena alternativa el XML. Con el fin de comprender la

ejecución con este formato, vamos a crear un pequeño archivo llamado *ejemplo3.txt*; este fichero contiene los siguientes datos en formato JSON, que representan las etiquetas de una serie de fotos obtenidas de *Instagram*.

```
{"tags": ["messi", "barcelona", "madrid"]}
{"tags": ["barcelona", "paella", "playa", "sangria"]}
{"tags": ["paella", "barcelona"]}
{"tags": ["ramblas", "paella", "ramblas"]}
{"tags": ["messi", "madrid"]}
{"tags": ["barcelona", "messi"]}
{"tags": ["messi", "gol", "barcelona"]}
{"tags": ["sangria", "paella"]}
```

Una vez tenemos el fichero con el formato específico, tenemos que procesarlos y guardarlos para luego obtener los vectores que necesitamos. Para esto, vamos a usar el *SQLContext* que nos proporciona Spark, ya que nos facilita todo el pre-procesado de los datos en este formato.

Scala

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val photos = sqlContext.jsonFile("ejemplo3.txt")

photos.printSchema

...

root
  |-- tags: array (nullable = true)
  |     |-- element: string (containsNull = true)

...

photos.count
photos.registerTempTable("pt")
tags = sqlContext.sql("select tags from pt where tags is
        not null")

tags.collect.foreach(println)
```

Una vez tenemos el fichero procesado y almacenado, podemos obtener el vector necesario para calcular la Frecuencia de Términos (TF) de nuestro

Python

```
sqlContext = SQLContext(sc)
def show(x): print x

photos = sqlContext.jsonFile("ejemplo3.txt")
photos.foreach(show)
photos.registerTempTable("pt")

tags = sqlContext.sql("select tags from pt where tags is
        not null")

tags.foreach(show)
```

fichero.

Scala

```
val tagsAsArray = tags.map(aRow => aRow(0)
        .asInstanceOf[
        scala.collection.mutable.ArrayBuffer[String]])

tagsAsArray.collect.foreach(println)
```

Python

```
tagsAsArray = tags.map(lambda x: array(x[0]))
```

Con estos vectores ahora podemos volver al punto anterior (10.4.3) y calcular el TF y crear una serie de *clusters*.

10.6 Creando tu propia aplicación

En esta última sección veremos como crear nuestra propia aplicación independiente con Spark. En esta aplicación obtendremos nuestro propio conjunto de datos de Twitter mediante una librería en *Python* y la API que Twitter pone a nuestra disposición. Con el fin de simplificarlo, y centrarnos solamente en la parte del tratamiento de datos con Spark, se proporciona el *script* para la obtención de los datos así como un pequeño fichero con algunos *hastags* que se han obtenido. Estos ficheros se pueden encontrar en el repositorio `https://github.com/mgparada/sparkmeetup`.

Una vez tengamos nuestro propio fichero de datos, o el que se proporciona, es el momento de crear nuestra aplicación Spark. Para comenzar, crearemos un fichero llamado *standaloneApp.py* en donde es necesario importar los diferentes contextos que nos proporciona Spark. Como vamos a leer el fichero con los datos, es necesario que importemos el paquete *sys*. Por último, y como en anteriores apartados, hemos definido la función *show* para poder obtener resultados por consola.

Python

```
from pyspark.sql import SQLContext
from pyspark import SparkContext
from pyspark.mllib.feature import HashingTF
from numpy import array
from pyspark.mllib.clustering import KMeans
import sys

def show(x): print x
```

Una vez tenemos definidos los paquetes que usaremos, comprobaremos que los parámetros que se le pasan a nuestra aplicación sean los correctos.

Python

```
if len(sys.argv) != 2:
        print >> sys.stderr,
        "Ejecucion: standaloneApp.py <data-filename>.json"
        exit(-1)
```

Como ya hemos visto en la sección anterior, para ejecutar código sobre Spark es necesario crear una instancia del contexto que nos ofrecen. Por tanto, tenemos que crear una variable que lo represente y nos permita usar las funciones de Spark.

Python

```
sc = SparkContext()
```

Ahora ya podemos comenzar a procesar nuestros datos, tal y como hemos hecho en apartados anteriores. Como los datos los hemos obtenido en formato JSON, tenemos que crear una instancia de *SQLContext* para que nos facilite el tratamiento.

Python

```
sqlContext = SQLContext(sc)

tweets = sqlContext.jsonFile(sys.argv[1])
tweets.foreach(show)

tweets.registerTempTable("pt")
words = sqlContext
        .sql("select hashtags from pt where hashtags is not null")

words.foreach(show)
```

Una vez hemos realizado el preprocesado, necesitamos obtener un vector con el que luego calcularemos la Frecuencia de Términos.

Python

```
wordsArray = words.map( lambda x:array(x[0]))
hashingTF = HashingTF()
tf = hashingTF.transform(wordsArray)

tf.foreach(show)
```

Por último, solo nos queda crear los *clusters* con el algoritmo *KMeans* para diferencias los *tweets* que hablan sobre Madrid o Barcelona.

Python

```
clusters = KMeans.train(tf,2,1,1)
results = wordsArray.map(lambda x:
        array([x,clusters.predict(hashingTF.transform(x))]))

results.foreach(show)
```

Ya tenemos nuestra primera aplicación Spark implementada. Para ejecutar esta aplicación, es necesario usar siguiente *script* que nos proporciona Spark:

Python

```
./bin/spark-submit standaloneApp.py tweets.json
```

10.7 Agradecimientos

Gonzalo Pericacho, estudiante de Máster de la UPC, por la elaboración de la primera versión de la sección 10.6.