Conceptos y Aplicaciones de Big Data

Spark streaming

Prof. Waldo Hasperué
(whasperue@lidi.info.unlp.edu.ar)

Temario

- Spark streaming

Spark streaming

- Spark streaming no es 100% streaming.
- Por cuestiones de eficiencia y compatibilidad, Spark streaming guarda el stream en pequeños "chunks" ejecutando pequeños procesos batch (micro-batch)

Spark streaming

- Spark streaming está diseñado para alimentarse desde varias fuentes de datos:
  - Apache Kafka
  - Apache Flume
  - Amazon Kinesis
  - Twitter
  - Sensores u otros dispositivos via TCP sockets

Spark streaming

- En Spark un stream es representado como un stream discreto (DStream) el cual es una secuencia de RDDs.
- Cada RDD es un snapshot de todos los datos recolectados durante un período de tiempo, el cual luego se procesa como un batch.

Spark streaming

- Los intervalos de cada chunk o micro-batch es configurable, el intervalo más pequeño es de 500ms

- Cuanto más chico es el intervalo, más RDDs se deben crear, lo cual podría tener un impacto negativo en el rendimiento del cluster.

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "My program")

ssc = StreamingContext(sc, 15)

stream = ssc.socketTextStream("localhost", 7777)

# transformaciones

ssc.start()
ssc.awaitTermination()
```

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "My program")

s   = StreamingContext(sc,  )

s   am = ssc.socketTextStrea   ocalhost", 7777)

    ormaciones

    t()
    tTermination()
```

Creamos un contexto que representa al framework de Spark

"local[2]" hace referencia al modo de ejecución. En nuestro caso será local y usará 2 procesadores (necesita al menos 2 para que funcione)

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "My program")

ssc = StreamingContext(sc, 15)

st  eam = ssc.socketTextStream("  calhost", 7777)

#   sformaciones

    ()
Termination()
```

A partir del contexto de Spark creamos el contexto que representa el procesamiento de streaming

Le indicamos la cantidad de segundos que dura la ventana temporal para la recolección de datos del stream

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "My program")

ssc = StreamingContext(sc, 15)

stream = ssc.socketTextStream("localhost", 7777)

# transformaciones

ssc.start()
ssc.awaitTermination()
```

Nos conectamos a la fuente de streaming. En nuestros ejercicios escucharemos el puerto 7777 de localhost

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "My program")

ssc = StreamingContext(sc, 15)

stream = ssc.socketTextStream("localhost", 7777)

# transformaciones

ssc.start()
ssc.awaitTermination()
```

El procesamiento del flujo.

Aquí están todas las transformaciones y acciones que arman el DAG a ejecutar

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.stre

sc = SparkContex

ssc = StreamingC

stream = ssc.soc

# transformaciones

ssc.start()
ssc.awaitTermination()
```

Las transformaciones y acciones que aplicamos en nuestro procesamiento no se ejecutan hasta tanto le damos la orden al contexto del stream. Como la acción de *start* crea un thread para el procesamiento del flujo, necesitamos esperar la finalización de ese thread con el método *awaitTermination*.

Proyecto en Spark streaming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "My program")

ssc = StreamingContext(sc, 15)

stream = ssc.socketTextStream("localhost", 7777)

# transformaciones
```

```
ssc.start()
ssc.awaitTermination()
```

Estas instrucciones hará que el DAG armado se ejecute una y otra vez indefinidamente, hasta que el thread se detenga (con CTRL+C o con el comando kill)

Streaming WordCount

```
# transformaciones

counts = stream.flatMap(lambda line:
                        line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)

counts.pprint()

# transformaciones
```

Streaming WordCount

```
# transformaciones

counts = stream.flatMap(lambda line:
                        line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)

counts.pprint()

# transformaciones
```

Las acciones y transformaciones son las mismas que el WordCount en batch.
La única diferencia es que se procesan las RDD leídas del stream

Streaming WordCount

```
# transformaciones

counts = stream.flatMap(lambda line:
                        line.split(" ")) \
                .map(lambda word: (word, 1)) \
                .reduceByKey(lambda a, b: a + b)

counts.pprint()

# transformaciones
```

El DStream resultante lo podemos imprimir en consola. También se puede escribir en el HDFS o ser enviado a otra fuente de datos.

Simulando el flujo

- Para que nuestra aplicación funcione necesitamos un servidor que envíe datos al puerto 7777.
- Eso lo conseguimos con el comando nc de linux

```
nc -lk 7777
```

- Este comando nos permitirá ingresar texto por consola y cada línea de texto ingresada será procesada por nuestra aplicación de spark streaming.

WordCount - Ejemplo

Chunk T1

buen día vida, buen día mundo

Ejecución del DAG

buen	2
día	2
vida	1
mundo	1

Chunk T2

un día en la vida del mundo

Ejecución del DAG

un	1
día	1
en	1
la	1
vida	1
del	1
mundo	1

Persistencia entre ventanas

- Cada nuevo "chunk" leído sobreescribe el anterior.
- Si queremos acumular datos de una ventana a otra necesitamos hacer persistencia en los DStream necesarios.
- La persistencia se realiza sobre DStream de la forma clave-valor.

Persistencia entre ventanas

- Para lograr la persistencia debemos hacer dos cosas.
- La primera es crear un checkpoint.

```
ssc.checkpoint("buffer")
```

- Esta instrucción creara en el HDFS un directorio llamado "buffer" (puede ser cualquier nombre válido) donde spark ira haciendo la persistencia de los Dstream que indiquemos.

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValues, history):
     if(history == None):
          history = 0
     if(newValues == None):
          newValues = 0
     else:
          newValues = sum(newValues)
     return newValues + history

history = counts.updateStateByKey(fUpdate)
```

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValues, history):
     if(history == None):
          history = 0
     if(newValues == None):
          newValues = 0
     else:
          newValues = sum(newValues)
     return newValues + history

history = counts.updateStateByKey(fUpdate)
```

counts es el Dstream resultado del procesamiento (por ejemplo WordCount)

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValues, history):
     if(history == None):
          history = 0
     if(newValues == None):
          newValues = 0
     else:
          newValues = sum(newValues)
     return newValues + history

history = counts.updateStateByKey(fUpdate)
```

La actualización se realiza con el método updateStateByKey

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValu
      if(history ==
            history =
      if(newValues == None):
            newValues = 0
      else:
            newValues = sum(newValues)
      return newValues + history

history = counts.updateStateByKey(fUpdate)
```

La actualización se realiza con la implementación de una función que reciba los nuevos pares <K, V> (los recolectados en la última ventana) más el Dstream persistido.

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValues, history):
    if(history == None):
        history = 0
    if(newValues == N
        newValues =
```

Por cada clave (nueva y persistida) se invoca a esta función.
En newValues se recibe una lista con los nuevos valores recolectados en la última ventana (podría ser nula o vacía).
En history se recibe el último valor persistido (podría ser nulo)

```
history = counts.updateStateByKey(fUpdate)
```

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValues, history):
    if(history == None):
        history = 0
    if(newValues == None):
        newValues = 0
    else:
        newValues = s   (newValues)
    return newValues +   tory

history = counts.updat       Key(fUpdate)
```

Hay que hacer todos los chequeos pertinentes porque tano newValues como history podrían no existir

Persistencia entre ventanas

- La segunda es actualizar el DStream correspondiente.

```
def fUpdate(newValues, history):
    if(history == None):
        history = 0
    if(newValues == None):
        newValues = 0
    else:
        newValues = sum(newValues)
    return newValues + history

history = counts.updateS          ey(fUpdate)
```

newValues es una lista con lo nuevos valores recolectados en el último "chunk"

Persistencia entre ventanas

- La segunda es actualiz

```
def fUpdate(newValu
    if(history ==
        history =
    if(newValues ==
        newValues = 0
    else:
        newValues = sum(newValues)
    return newValues + history

history = counts.updateStateByKey(fUpdate)
```

Luego de los chequeos pertinentes devolvemos el nuevo acumulado. El valor devuelto será el que se persista para la clave que se está procesando. Este valor será el que se reciba como "*history*" en la próxima ventana.

WordCount ejemplo

Chunk T1

buen día vida, buen día mundo

Chunk T2

que hermosa es la vida, que hermoso es el mundo

WordCount ejemplo

Chunk T1

buen día vida, buen día mundo

Resultado del WordCount

- buen 2
- día 2
- vida 1
- mundo 1

Key	newValues	history	return
buen	[2]	None	2
día	[2]	None	2
vida	[1]	None	1
mundo	[1]	None	1

WordCount ejemplo

Chunk T2

que hermosa es la vida, que hermoso es el mundo

Resultado del WordCount

- que 2
- hermosa 1
- es 2
- la 1
- vida 1
- hermoso 1
- el 1
- mundo 1

Key	new Values	history	return
buen	None	2	2
día	None	2	2
vida	[1]	1	2
mundo	[1]	1	2
que	[2]	None	2
hermosa	[1]	None	1
es	[2]	None	2
la	[1]	None	1
hermoso	[1]	None	1
el	[1]	None	1

WordCount ejemplo (sin reduceByKey)

Chunk T2

que hermosa es la vida, que hermoso es el mundo

Resultado del WordCount

- que 1
- que 1
- hermosa 1
- es 1
- es 1
- la 1
- vida 1
- hermoso 1
- el 1
- mundo 1

Key	new Values	history	return
buen	None	2	2
día	None	2	2
vida	[1]	1	2
mundo	[1]	1	2
que	[1,1]	None	2
hermosa	[1]	None	1
es	[1,1]	None	2
la	[1]	None	1
hermoso	[1]	None	1
el	[1]	None	1

Persistencia entre ventanas

```
def fUpdate(newValues, history):
    if(history == None):
        history = 0
    if(newValues == None):
        newValues = 0
    else:
        newValues = newValues[0]
    return newValues + history

history = counts.updateStateByKey(fUpdate)
```

Estructuras de control en Spark streaming

¿Qué hace?

```
valor = stream.count()


if valor > 5:
    data = stream.map(lambda t: t * 2)
else:
    data = stream.map(lambda t:     / 2)



ssc.start()
```

Genera un nodo u otro en el DAG dependiendo de una condición que AUN no se evaluó

Estructuras de control en Spark streaming

¿Qué hace?

```
for i in range(100)
    stream = stream.map(lambda t: t * 2)
```

```
ssc.start()
```

Genera un DAG de 100 nodos maps. En cada ventana del stream se ejecutarán los 100 maps

Estructuras de control en Spark streaming

¿Qué hace?

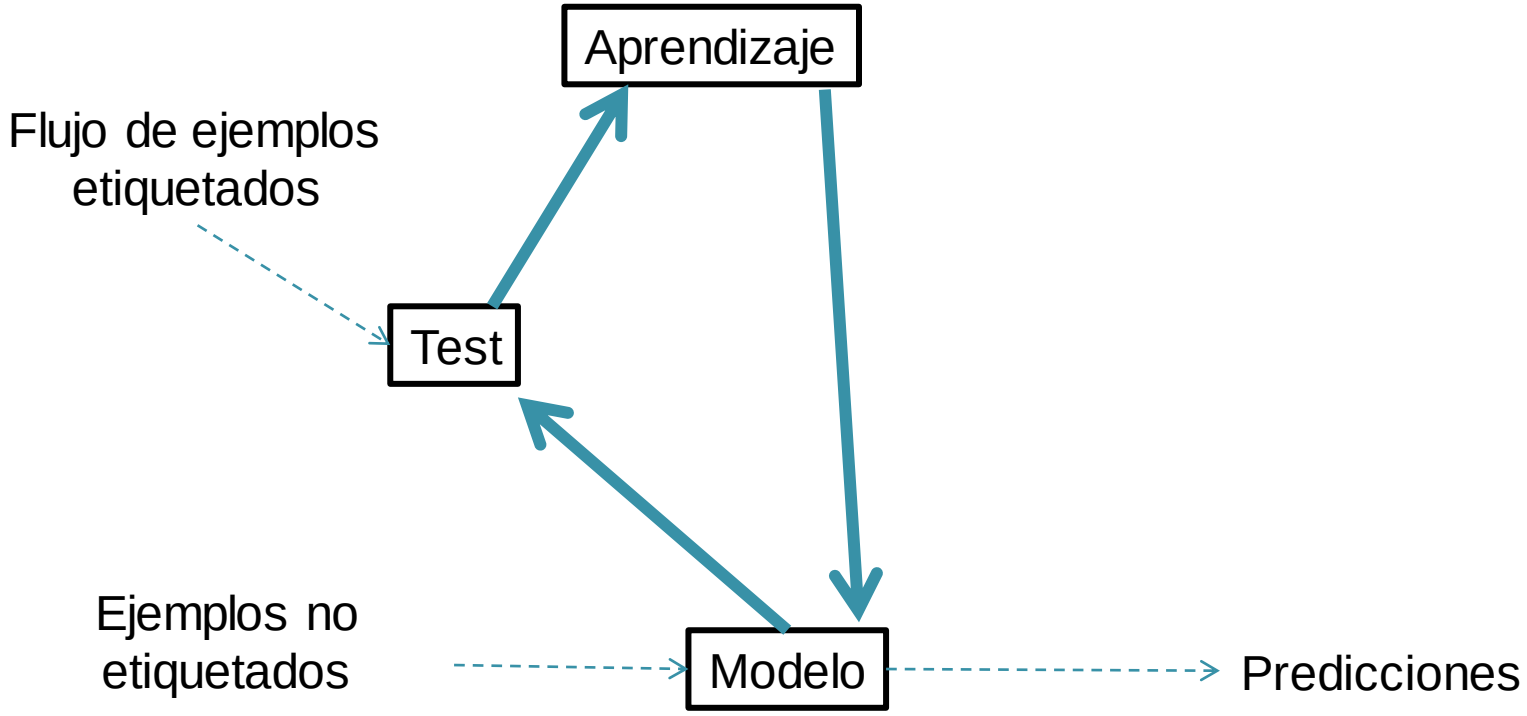
```
error = 1; tolerancia = 0.0001

while error > tolerancia:
    stream = stream.map(funcionMap)
    error = stream.reduce(funcReduc)

ssc.start()
```

Probablemente se quede en un loop infinito, ya que el error que se intenta calcular no se va a saber hasta que comience la ejecución

Entrenamiento on-line



Entrenamiento on-line

- Se deben hacer dos scripts que se ejecutan por separado.
  - El primero (entrenamiento) obtiene y refina el modelo continuamente. Como el modelo se almacena en un Dstream se puede persistir en el HDFS.
  - El segundo script (predictor) lee el modelo del HDFS y lo utiliza para hacer las predicciones.

Spark streaming - Resumen

- No deben utilizarse las estructuras de control
- El script principal solo debería armar el DAG. Éste se ejecuta permanentemente en un "while(True)" interno de Spark, después de invocar al comando "*start*" del streaming context.
- Toda información que desee almacenarse entre la ejecución de distintas ventanas debe ser persistido con la función "*updateStateByKey*"