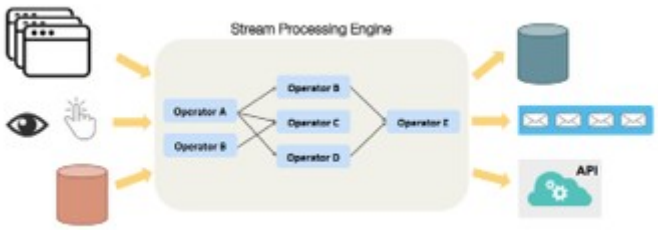
UIMP
Universidad Internacional
Menéndez Pelayo

AEPIA
Spanish Association for
Artificial Intelligence

Hortonworks
Partner
ACADEMIC
TRAINING

Ejercicios de Streaming



Spark Streaming

Configurar el entorno de desarrollo de Spark Streaming

Objetivo Configurar un entorno para desarrollar aplicaciones de Spark Streaming en Scala usando SBT e IntelliJ

Resultado Tendrá un nuevo proyecto en IntelliJ definido y configurado para desarrollar aplicaciones Spark Streaming

Antes de comenzar Tenga la máquina virtual instalada

Descargue el projecto inicial que le servirá de base para comenzar

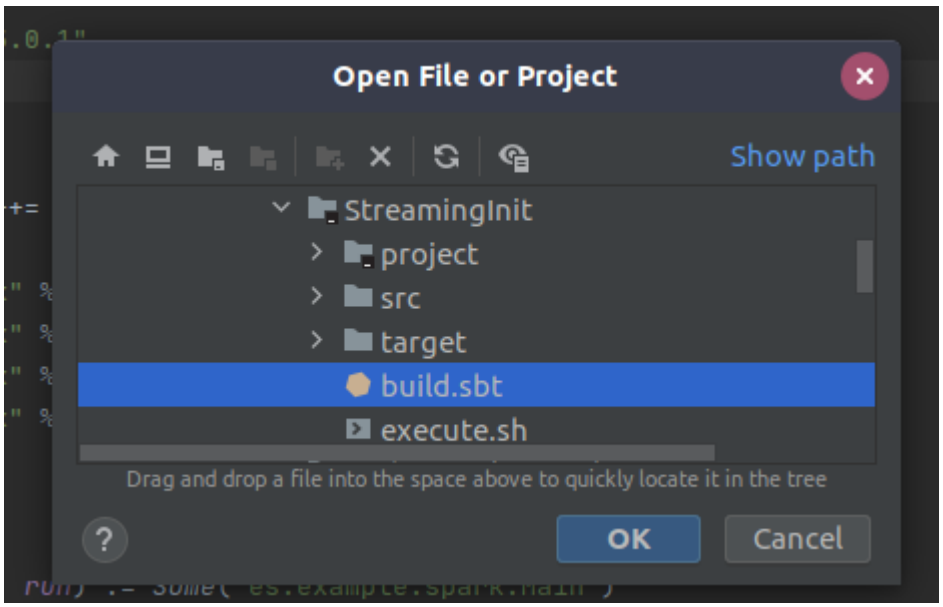
```
Carpeta "StreamingInit" del archivo disponible en
PoliformaT.
```

Entre en el directorio y compruebe que puede compilar el proyecto

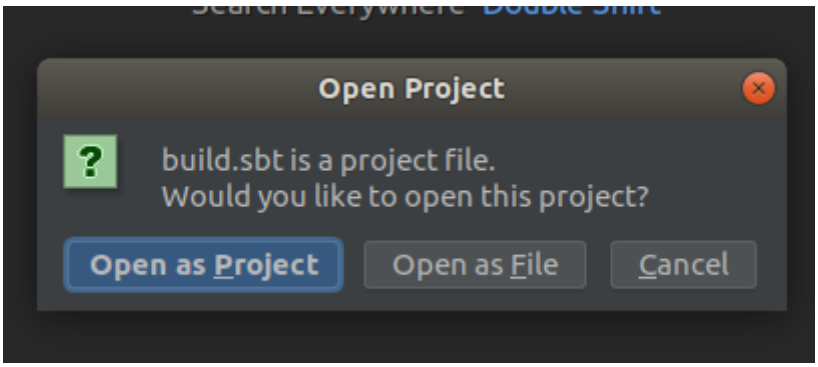
```
$ cd ~/StreamingInit

$ sbt clean assembly
```

Finalmente, importe el proyecto normalmente el InltelliJ en File > Open y seleccione el fichero build.sbt



En la ventana emergente, seleccione “Import as Project”



Resultado: Tiene configurado el entorno.

Conteo de palabras en stream

Objetivo Escribir un programa que reciba cadenas de texto por red y mantenga una estadística de conteo

Resultado Un primer proyecto completo de WordCount en Spark Streaming

Antes de comenzar Configure el entorno de desarrollo.

Realice los siguientes pasos:

1. Partiendo del proyecto en blanco complete la clase principal NetworkWordCount
    1. Conéctese al cluster de Spark. Esta línea genérica serviría tanto para conectarse a un cluster (YARN/MESOS) o para ejecutar en local. Sobre la conexión al cluster, creamos un StreamingContext, que nos permite la planificación de trabajos continuamente.

```
val sparkConf = new SparkConf().setAppName("NetworkWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(1))
```

    2. Establezca los niveles de log para evaluar el funcionamiento continuo del pipeline. Observe que este método que ya está incluido solamente incrementa el nivel de log hasta WARNING

```
ssc.sparkContext.setLogLevel("WARN")
```

    3. Conecte la fuente de datos al socket. El host viene determinado por el primer argumento y el puerto por el segundo. Los datos se van a mantener en memoria y disco de manera serializada para ahorrar espacio

```
val lines = ssc.socketTextStream(args(0), args(1).toInt,
    StorageLevel.MEMORY_AND_DISK_SER)
```

    4. Divida los bloques de texto que llegan y cuente el número de palabras.

```
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
```

    5. Imprima el resultado por pantalla para poder evaluar el funcionamiento

```
wordCounts.print()
```

    6. Inicie la ejecución del pipeline. Este se mantendrá funcionando mientras se mantenga la conexión a la fuente de datos o hasta que el trabajo sea terminado

```
ssc.start()
```

```
ssc.awaitTermination()
```

1. Ejecute el trabajo
    1. Para ejecutar el trabajo, primero tiene que abrir un servidor en el que enviar datos. Para ello abra la línea de comandos y ejecute

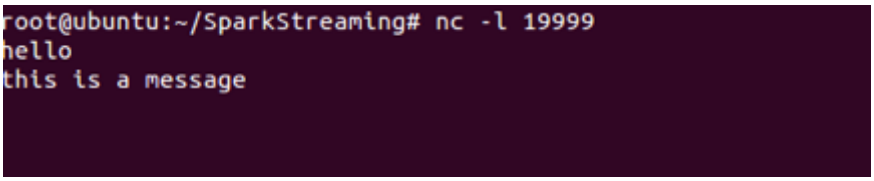
```
$ nc -l 19999
```

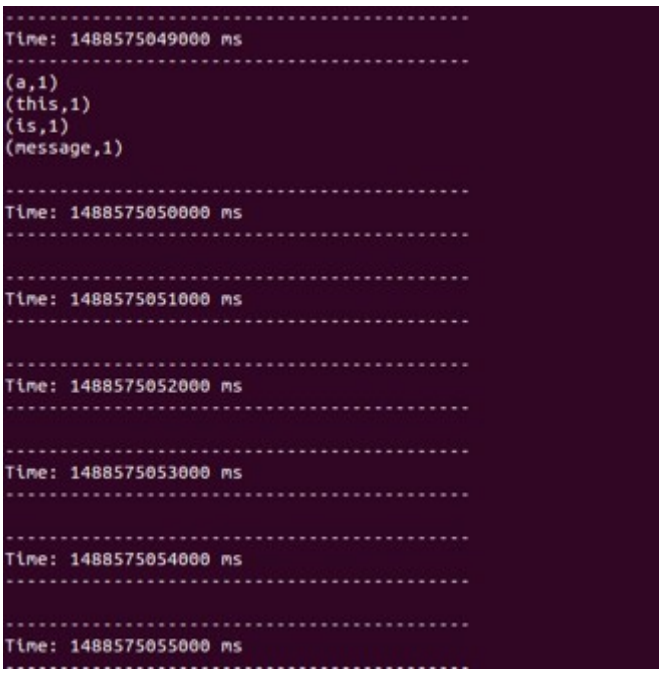
    2. Compile la aplicación y ejecute. Compruebe como execute.sh envia el trabajo a un cluster local. Para enviar este trabajo a un cluster remoto, solo habría que cambiar el argumento **master**

```
$ sbt clean assembly

$ ./execute.sh
es.dmr.uimp.spark.streaming.NetworkWordCount localhost
19999
```

    3. Al escribir en el terminal del servidor debería de poder ver el conteo ejecutándose en Spark.





Enhorabuena, acaba de ejecutar su primer trabajo de Spark.

Conteo total de palabras

Objetivo: Mantener un conteo total de palabras a lo largo del tiempo

Resultado: Un pipeline que mantiene actualizado el total de apariciones hastael momento

Antes de comenzar: Configure el entorno de desarrollo

Realice los siguientes pasos:

1. Construya el pipeline de calculo en la clase StatefulNetworkWordCount
    1. Establezca la conexión con Spark y el entorno de Streaming

```
val sparkConf = new
    SparkConf().setAppName("StatefulNetworkWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(1))
```

    2. Igual que antes, configure el nivel de log

```
ssc.sparkContext.setLogLevel("WARN")
```

    3. En este ejercicio, el pipeline va a mantener una cuenta total del número de veces que un término ha aparecido en el stream. Para que la computación sea fiable, es necesario que periódicamente se haga un backup del estado del pipeline. Establecemos la localización donde este backup se va a mantener

```
ssc.checkpoint("./checkpoint")
```

    4. Creamos una conexión al socket como en el ejercicio anterior y distribuimos las palabras encontradas

```
val lines = ssc.socketTextStream(args(0), args(1).toInt)
val words = lines.flatMap(_.split(" "))
val wordsDstream = words.map(x => (x, 1))
```

    5. Construimos la actualización del conteo total. Para ello, es necesario una función que toma como entrada la clave, el valor y el estado actual para esa clave, y lleva a cabo la actualización. La salida es el nuevo valor para el estado

```
val mappingFunc = (word: String, one: Option[Int], state:
State[Int]) => {
    val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
    val output = (word, sum)
    state.update(sum)
    output
}
```

6. Conectamos la actualización del valor con el origen de datos.
    1. Comenzamos con un estado inicial

```
val initialRDD = ssc.sparkContext.parallelize(List(("hello", 1),
("world", 1)))
```

    2. Conectamos el calculo con el stream de entrada

```
val stateDstream = wordsDstream.mapWithState(
    StateSpec.function(mappingFunc).initialState(initialRDD)
)
```

7. Mostramos el nuevo estado por pantalla

```
stateDstream.print()
```

8. Arrancamos el pipeline

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

2. Ejecuta el pipeline
    1. Este pipeline funciona igual que el ejemplo anterior, solo tienes que cambiar la clase por StatefulNetworkWordCount

```
$ sbt clean assembly

$ ./execute.sh
es.dmr.uimp.spark.streaming.StatefulNetworkWordCount
localhost 19999
```

    2. Ahora debería observar que el conteo publicado es el total acumulado, y no el conteo dentro de cada batch únicamente.

Resultado: Ha completado su primer calculo con estado en Spark Streaming.

Integrar Spark con Kafka

Objetivo Conectar un pipeline de Spark con el topic “sentences” que utilizamos antes en la práctica de Spark y mantener una media de apariciones de términos

Resultado Un pipeline que mantiene actualizada una media móvil por segundo de términos hasta el momento

Antes de comenzar Configure el entorno de desarrollo

Realice los siguiente pasos

1. Introduzca el código para conectarse a Kafka y hacer el cálculo de la media
    1. Abra la clase inicial KafkaWordCount.scala, introduzca la conexión a Spark e inicie el entorno de Streaming.

```
val sparkConf = new
    SparkConf().setAppName("KafkaNetworkWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(1))
```

    2. Añada un directorio de checkpoint para que el pipeline guarde el estado

```
ssc.checkpoint("./checkpoint")
```

    3. Conéctese a Kafka. Para ello es necesario indicar
        1. Dónde se encuentra el cluster de Zookeeper con la información acerca del cluster de Kafka y todos los topics, partitions, etc
        2. El identificador de consumer group que los lectores van a compartir
        3. La lista de temas de la que se quieren consumir mensajes
        4. El número de consumidores que van a leer de Kafka y crear el Dstream

```
val Array(zkQuorum, group, topics, numThreads) = args
val topicMap = topics.split(",")

val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> zkQuorum,
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> group,
  "auto.offset.reset" -> "earliest",
  "enable.auto.commit" -> (false: java.lang.Boolean)
)

val lines = KafkaUtils.createDirectStream[String, String](
  ssc,
  PreferConsistent,
  Subscribe[String, String](topicMap, kafkaParams)
```

```
).map(_.value)
```

4. Ahora vamos a realizar el cálculo de la media de apariciones. Para ello, vamos a hacer un calculo rolling. Cada RDD de entrada es un batch de 1 segundo y lo que queremos es tener el número de apariciones por segundo. Por lo tanto, por cada batch se genera un valor

```
val words = lines.flatMap(_.split(" "))
val wordsDstream = words.map(x => (x, 1))
                        .reduceByKey(_ + _)
                        .map{case (k:String, c:Int) => (k, (c, 1))}
```

5. Ahora vamos a hacer el calculo de la media. En cada batch se genera un valor con clave con (count, 1). Para hacer el calculo de la media, los estadísticos suficientes son (media actual, numero de batches). En base a esto se puede hacer la actualización

```
val mappingFunc = (word: String, c: Option[(Int, Int)], state:
State[(Float, Int)]) => {
  val s = state.getOption.getOrElse((0.0f, 0))
  val v = c.getOrElse((0, 0))
  val sum = s._1*s._2 + v._1
  val count = s._2 + v._2

  if (count == 0) {
    val output = (0.0f, count)
    state.update(output)
    (word, output)
  }else{
    val output = (sum/count.toFloat, count)
    state.update(output)
    (word, output)
  }
}
```

6. Conecte el calculo de la media de apariciones con el stream de entrada

```
val stateDstream =
wordsDstream.mapWithState(StateSpec.function(mappingFunc))
```

7. Imprima el resultado y comience la ejecución del pipeline de cálculo.

```
stateDstream.print()
ssc.start()
ssc.awaitTermination()
```

2. Arrancar Kafka
    1. Arranque los servidores de Zookeeper y Kafka (en caso de no estar ya arrancados).
    2. Abra una ventana Terminal
    3. Entre en la carpeta /opt/Kafka/kafka_2.11-2.3.0/:

    $ cd /opt/Kafka/kafka_2.11-2.3.0/

3. Arranque el servidor Zookeeper. Para ello, ejecute el siguiente comando:

   $ sudo bin/zookeeper-server-start.sh config/zookeeper.properties &

4. Arranque el servidor de Kafka. Para ello, ejecute el siguiente comando:

   $ sudo bin/kafka-server-start.sh config/server.properties &

2. Definir un topic:
   1. Utilice el script para crear un nuevo topic:

      $ bin/kafka-topics.sh --create bin/kafka-topics.sh --topic my_topic --partitions 1 --replication-factor 1 --zookeeper localhost:2181

   2. Utilice la opción --list de kafka-topics.sh para verificar que su topic se ha creado correctamente.

      $ bin/kafka-topics.sh --list --zookeeper localhost:2181

   3. Prueba del topic
      1. Ejecute el siguiente comando, que le permite enviar mensajes a un tema desde la consola:

      $ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my_topic

      2. Escriba un texto y pulse Enter. Cada línea de texto que ingrese envía un mensaje a *my_topic*.
      3. Pulse **Ctrl + c** para detener el proceso.
      4. Para consumir mensajes, ejecute el programa siguiente:

      $ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my_topic --from-beginning

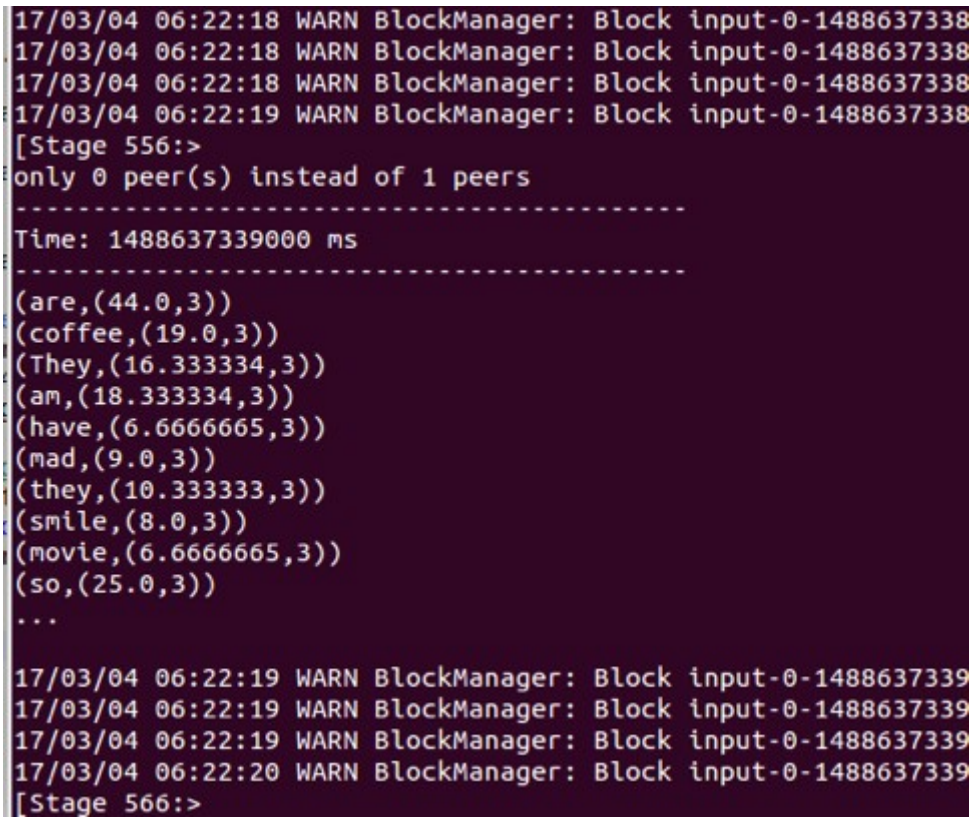
      5. Debería ver los mensajes que escribió anteriormente.
      6. Pulse **Ctrl + c** para detener el proceso.

3. Ejecute el pipeline
   1. Primero tiene que crear el tema “sentences” en el cluster Kafka
   2. Una vez creado el tema, puede introducir datos utilizando el *kafka-console-producer.sh*
   3. Compile el proyecto y ejecute el pipeline

```
$ sbt clean assembly

$ ./execute.sh
es.dmr.uimp.spark.streaming.KafkaWordCount
localhost:9092 kafkaConsumer sentences 1
```

4. Debería ver la salida que actualiza el estado medio cada vez que introduzca datos en el tema de Kafka. El conteo de numero de batches debería de aumentar continuamente y la media mantenerse actualizada.



Resultado: Ha conectado una aplicación de Spark Streaming a Kafka y mantenido actualizado el calculo de apariciones medias. El estado del pipeline se mantiene guardado en el sistema de ficheros. Para desplegar este código en producción, sólo tendría que cambiar la localización del directorio de checkpoint a un lugar en HDFS, de manera que este estado se guarde replicado y pueda recuperarse ante un fallo de nodo.

Cálculos de ventana en Spark Streaming

Objetivo Realizar un cálculo que mantenga el total de apariciones en los últimos 10 segundos, actualizando cada segundo

Resultado Un pipeline que mantiene actualizada un conteo móvil por segundo de términos para los últimos 10 segundos

Antes de comenzar Realice el ejercicio anterior

Realice los siguientes cambios

1. Duplique la clase KafkaWordCount.scala, vamos a cambiar el cálculo sobre el pipeline de Kafka anterior.
    1. Genere un evento por cada aparición de un término

```
val wordDstream = words.map(x => (x, 1))
```

    2. Cree una función que sume evento cada vez que estos entren en la ventana

```
val addCount = (x: Int, y: Int) => x + y
```

    3. Añada una función que elimine conteos que caigan fuera de la ventana cuando esta se mueva hacia delante

```
val removeCount = (x: Int, y: Int) => x - y
```

    4. Añada una función que filtre términos que hayan desaparecido. Esto nos ayuda a que la memoria del cálculo en el cluster no crezca indefinidamente y sin necesidad

```
val filterEmpty = (x: (String, Int)) => x._2 != 0
```

    5. Conecte el cálculo a la entrada e imprima el resultado

```
val runningCountStream =
wordDstream.reduceByKeyAndWindow(addCount, removeCount, Seconds(10),
Seconds(1), 2, filterEmpty)
```

2. Ejecute el pipeline
    1. Introduzca los datos en el topic de kafka igual que en el tema anterior y ejecute el cálculo. Debería ver un calculo móvil (Esto se hace más evidente si espera un tiempo, verá que la función de filtro termina por vaciar el estado y deja de imprimir).