# Spark Streaming

ASR7

09/12/2022

## 1 Setup

You can run the code in this lab in two environments:

- The Hadoop cluster we built together.
- Directly on your machine. In this case, you need to download a stand-alone version of Spark that you can find on this page: https://spark.apache.org/downloads.html. Untar the file you downloaded with *tar -xvf spark....tgz*. Then, to run a Spark shell, you can go to *spark_your_version/bin* and execute *./sparkshell* like in our previous lab.

In this lab, we will run the code in a *spark-shell* environment, but we could also compile our code in Scala and submit our job.

## 2 Word Count

In this first section, we want to understand how a Spark-streaming app runs. We will write a simple word count that counts the words on a TCP socket.

2.1. When using Spark-streaming, we need to define a specific context to run our code as follows:

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._


val ssc = new StreamingContext(sc, Seconds(10))
```

(in the *spark-shell*, the SparkContext is predefined for us. If you want to compile and run the program, you need to define it as we saw last time.)

What does the second argument of the StreamingContext's constructor represent?

2.2. We want our application to connect on a TCP socket running on the *localhost*, on the port 9999 (on the Docker cluster, we will instead use *spark-master* as the hosting server). To get the lines out of this socket, we can use:

```
val lines = ssc.socketTextStream("localhost", 9999)
```

*socketTextStream* returns a DStream. Go and check the documentation. Can you find all the functions we explored last time when manipulating RDDs? What is the output type of these functions?

2.3. Now, we can manipulate our text stream *lines* as we did with files in the last lab. Write a code to compute a word count. To show the results, you will need to add:

```
wordCounts.print(10) // Print the 10 first elements
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

2.4. If you start the computation now, you should have many runtime errors as no server runs on localhost on port 9999. To correct that, we will use Netcat with the command *nc -vv -l -s 0.0.0.0 -p 9999*. Now, everything you will type in the Netcat terminal will be sent over TCP to whoever is connected. Test your program and check it works.

2.5. Open the runtime WebUI (most likely accessible with localhost:4040). When does Spark run a new job? At the top, a tab was not here last time. What is it? What kind of metrics do you get?

2.6. In the lecture, you have seen what *windows* are. What are the two characteristics of a window? What are their "default" values? From the *lines* DStream, create a new DStream with different window parameters. Test your program.

# 3 Word Count with Kafka

In this exercise, we will test the word count but this time the data is produced by kafka.

3.1. Install Kafka: this lab supposes that kafka is available on your machine. If not, you should download and install it.

3.2. Start a Zookeeper service: run in a terminal a single-node Zookeeper instance

3.3. Start the Kafka broker: launch a single Kafka broker in a new terminal

Note: The Zookeeper will be in port 2181 and Kafka in 9092.

3.4. Create a topic with one partition using –create

3.5. Publish the first messages in the topic. To publish messages, use the Kafka client console.

3.6. Create a consumer group to associate a consumer to this group

3.7. Now we want to build our Spark Streaming application (do not forget to import the JARs from Kafka and Spark like in the previous labs if you want to use your IDE).

```
import org.apache.kafka.clients.consumer.ConsumerConfig
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka010._


object SparkKafkaWordCount {
  def main(args: Array[String]) {
    if (args.length < 3) {
      System.err.println(s"""
        |Usage: SparkKafkaWordCount <brokers> <groupId> <topics>
        |
        """.stripMargin)
      System.exit(1)
    }
    val Array(brokers, groupId, topics) = args
  }
}
```

3.8. Create a context with 2 second batch interval

3.9. Create kafka stream with brokers and topics

```
val topicsSet = topics.split(",").toSet
val kafkaParams = Map[String, Object](
  ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG -> brokers,
  ConsumerConfig.GROUP_ID_CONFIG -> groupId,
  ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG -> classOf[StringDeserializer],
  ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG -> classOf[StringDeserializer])
val messages = KafkaUtils.createDirectStream[String, String](
  ssc,
  LocationStrategies.PreferConsistent,
  ConsumerStrategies.Subscribe[String, String](topicsSet, kafkaParams))
```

3.10. Get the lines, split them into words, count the words and print.

3.11. Start the computation

3.12. Start the new Spark application using spark-submit. Here, you have two options. You can either use the spark-shell with the following command:

```
./bin/spark-shell --packages org.apache.spark:spark-streaming-kafka-0-10_2.12:3.0.1
```

and then use *:load YourScalaFile.scala*. To start the program, you have to simulate a call to the main function with:

```
SparkKafkaWordCount.main(Array("localhost:9092", groupId, topicId))
```

Here, the exit in the main function will close the spark-shell. Remove it if you want. You can also generate a JAR and specify the group and the topic.

```
./bin/spark-submit --packages org.apache.spark:spark-streaming-kafka-0-10_2.12:3.0.1
    --master local[2] ./[PATH TO YOU JAR].jar localhost:9092 groupId topicId


groupId : is a consumer group name to consume from topics
topicId : is a list of one or more kafka topics to consume from
```

3.13. Cleaning: Don't forget to stop all the processes

- Stop the existing Kafka broker (ctrl C or kill the corresponding process)
- Stop the Zookeeper service
- Delete the logs of Zookeeper and Kafka using the following command:

```
rm -rf /tmp/zookeeper/* /tmp/kafka-logs/*
```

- Delete a topic:

```
./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic topic1 --delete
```

# 4 Kmeans On Numerical Data

In this exercise, we will learn how to perform clustering on numeric data using the famous KMeans algorithm that you saw in a previous lecture. The difference is; that we will work on a stream of data. Please go on the Moodle page and download the file *full.zip*. Unzip it. It contains 1,000 files, and each of them includes 100 lines representing a 2D vector.

Here, you might need the following imports:

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.clustering.StreamingKMeans
import org.apache.spark.sql._
import org.apache.spark.sql.types._
```

4.1. We will use the *textFileStream(directory: String): DStream[String]* to read files. It takes as input a directory and creates a data stream that monitors the new files in that directory and reads them as text files. How can we emulate a fake data stream with our files?

4.2. Now, use *textFileStream* to read the files and turn the lines into vector using the function *Vectors.parse*. Here, we can consider we create a batch every 10 seconds.

4.3. Now that we have a stream of vectors, we can perform the clustering. To do so, use the class *StreamingKMeans*. First, by reading the documentation, instantiate a new model and set the number of clusters to 5, its decay factor to 0.5, and initialize random centers. Next, find the function to train the model and train it on the vector stream we defined previously.

4.4. Here, we will make the prediction on the same data stream. Ask the model to perform the predictions. We want to save the results in a new directory. To do so, we can use the function *foreachRDD* that applies a given function to all batches, represented by an RDD. We remind you that the function to save to a text file is *saveAsTextFile*.

4.5. You can start the computation using *ssc.start()*. Use the method you found earlier to simulate a stream. You can stop the processing by running *ssc.stop()*.

4.6. We want to clean our data by putting together the data points and the prediction. Open the data points and the predictions in two RDD. Turn the coordinates into Arrays with *Vectors.parse(p).toArray* and clusters into Ints. Merge the two resulting RDDs using the *zip* function. This function outputs an RDD of tuples. In general, we prefer to manipulate DataFrames, which are more or less equivalent to SQL tables. From the result of the *zip*, create a DataFrame with two columns, "cluster" and "coords". Print the schema of your DataFrame.

# 5 Kmeans On Textual Data: Optional

This exercise aims to cluster Tweets using word embeddings called Word2Vec. To simplify, we do not consider that the Tweets come from a stream, but the code could easily be adapted. We will use Kmeans for the clustering part. Go to the page of the Moodle and download *tweets*, *stop_words* and *w2vModel.zip*. Unzip the last one. They contain a list of 800 Tweets, a list of stop words, and a vector representation for the words.

We will need linear algebra to make operations on vectors. So, import and define the following functions:

```
import org.apache.spark._
import org.apache.spark.rdd._
import org.apache.spark.SparkContext._
```

```
import org.apache.spark.mllib.feature.{Word2Vec, Word2VecModel}
import org.apache.spark.mllib.linalg.{Vector, Vectors, DenseVector, SparseVector}
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.util.KMeansDataGenerator
import scala.io.Source
import breeze.linalg.{DenseVector => BDV, SparseVector => BSV, Vector => BV}
import org.apache.spark.mllib.linalg.{Vector => SparkVector}
def toBreeze(v:SparkVector) = BV(v.toArray)
def fromBreeze(bv:BV[Double]) = Vectors.dense(bv.toArray)
def add(v1:SparkVector, v2:SparkVector) = fromBreeze(toBreeze(v1) + toBreeze(v2))
def scalarMultiply(a:Double, v:SparkVector) = fromBreeze(a * toBreeze(v))
```

5.1. First, we want to read the stop words and the Word2Vec vectors. We can do it with the following code:

```
// Read the stopwords
import scala.io.Source
val stopWords = Source.fromFile("stop_words").getLines.toSet
// Read Word2Vec
val w2vModel = Word2VecModel.load(sc, "w2vModel")
// We want to be able to map a string to an array of scala,
// i.e. we want a (serializable) Map[String, Array[Float]]
val vectors = w2vModel.getVectors.mapValues(
    vv => Vectors.dense(vv.map(_.toDouble))).map(identity)
```

As you might notice, we do not have RDDs here. This is problematic as our data will be distributed to several workers on different machines. Therefore, they will not be able to access these local variables. Fortunately, Spark offers a mechanism to create read-only variables shared by all nodes called *Broadcasts*. Using the documentation of *SparkContext*, broadcast the stop words and the word vectors.

5.2. We want to preprocess the Tweets to clean them and turn them into a vector representation. Start by creating an RDD from the *tweets* file and implement the following steps:

1. Remove sentences of length less than 2.
2. Turn the sentences into lower case.
3. Split the sentences into words (do not use *flatMap* here) using as a separator non-alphabetic characters.
4. Turn each Tweet into a vector representation by taking the mean of the representation of all its words. Ignore the stopwords during this process. The size of the vectors in Word2Vec is 100.
5. Remove all the null vectors that represent the case where no word was in the Word2Vec vocabulary (you can use *Vectors.norm(vec, 1.0)*).

These steps can be long to run, and we might not want to do them every time we need the vector representation. That's why you can call the *.persist()* function on the final RDD, so the results of the computations are saved the first time they are done.

5.3. Train a Kmeans model with 5 clusters and 200 iterations. Once it is done, you can inspect what the words associated to the centroids might be with:

```
clustering.clusterCenters.foreach(clusterCenter => {
    w2vModel.findSynonyms(clusterCenter,5).foreach(synonym => print(" %s
(%5.3f),"
            .format(synonym._1, synonym._2)))
    println()
})
```