Hands-on 10:

Spark and Open Standard Data Formats

SUPERCOMPUTERS ARCHITECTURE

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

Barcelona, Fall 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hands-on 10

Spark and Open Standard Data Formats

SUPERCOMPUTERS ARCHITECTURE

Facultat d'Informàtica de Barcelona (FIB)

UPC Barcelona Tech & Barcelona Supercomputing Center

Version 1.0 - 28 November 2015

Professor: Jordi Torres

Table of Contents

1 Hands-on content

One important thing in Spark framework is that it has good functions for parsing popular Open Data Standard Formats as JSON or CVS. However, for XML it is necessary to write several additional lines of code to create a data frame by specifying the schema programmatically.

In this hands-on we will experiment with JSON and XML information formats as a input data. For this hands-on we are going to use Scala language for explaining the details based in a couple of examples. Finally we will present the same code with Python.

For XML this hands-on uses text, data and code presented by Dimitry Petrov in his blog Full Stack ML: http://fullstackml.com. Thank you Dimitry!

2 Setup environment

For this hands-on it is required to have the version 1.5 of Spark. If you don't have this version of Spark in your machine you can simply download it from the Spark web page http://spark.apache.org/. A direct link of a pre-built version 1.5.1 used for preparing this hands-on can be downloaded from this direct :

http://d3kbcqa49mib13.cloudfront.net/spark-1.5.1-bin-hadoop2.6.tgz

You also need to be sure that you are using the latest version of Java. If not install it.

```
jorditorres$ java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed
mode)
You are ready to run Spark in Standalone mode if Java is
installed in your computer. If not – install Java.
```

For linux systems and Macs, uncompress the downloaded .tgz file and copy to any directory. This is a Spark directory now. Run Spark shell for executing the code presented in this hands-on. Spark shell can run your Scala command in interactive mode:

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

3 Parsing JSON

JSON or JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. Today it is very popular and we thing that we should consider also this format.

In order to practice with JSON files, we will use the same MLlib learning function as Hands-on 9. However, now we will create a new file named `example3.txt` that contains the following JSON data representing tags for pictures obtained from 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"]}
```

First, we will process the data, and store it in a SqlContext. The reason is because K-means doesn't works with tags, we need to transform them as a vectors.

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val photos = sqlContext.jsonFile("example3.txt")
photos.printSchema

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

photos.count
photos.registerTempTable("pt")
val tags = sqlContext.sql("select tags from pt where tags is
not null")
tags.collect.foreach(println)
```

After we will vectorize the data with TF:

```
val tagsAsArray = tags.map(aRow =>
aRow(0).asInstanceOf[scala.collection.mutable.ArrayBuffer[St
ring]])
tagsAsArray.collect.foreach(println)
```

Once we have an array with the data, we can repeat the same steps that we did in the previous hads-on to create the TF and clustering with KMeans.

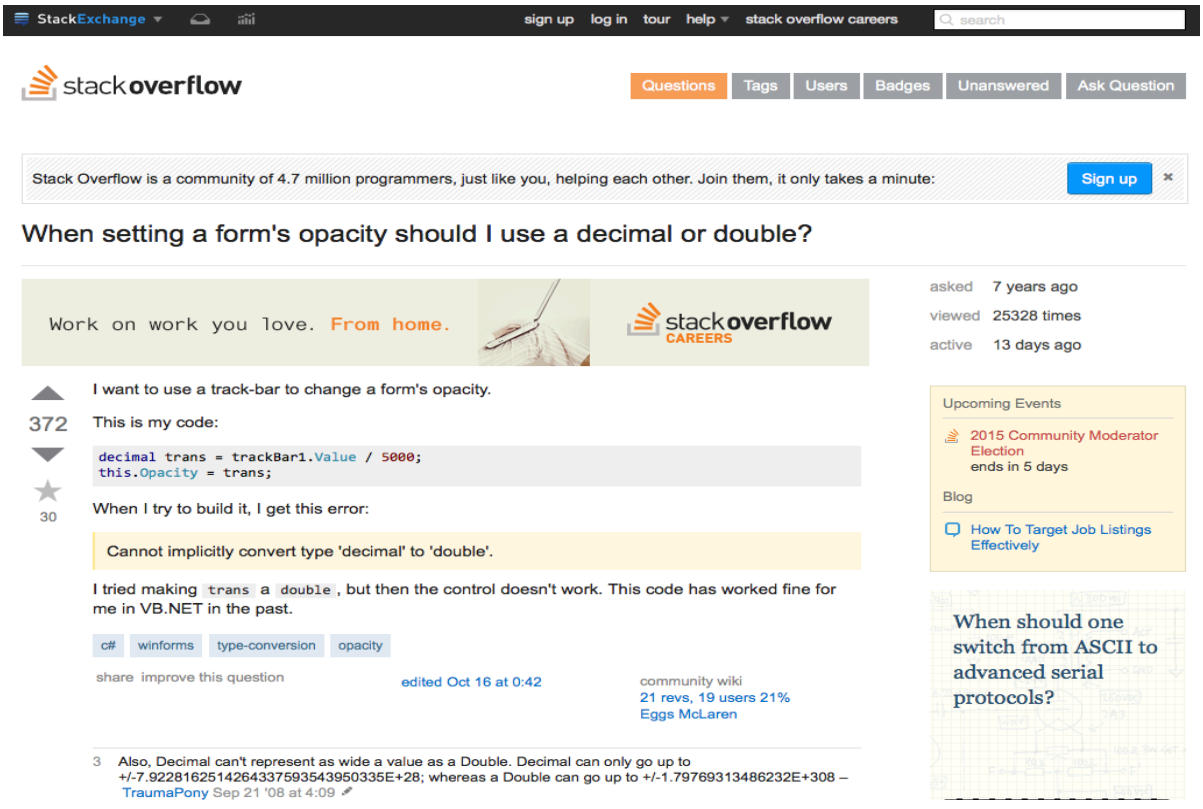
4 Parsing XML

For describing how to parse a XML input data we will use data from Stackoverflow.com. Stackoverflow is a question and answers web site about programming. I assume that all of you are familiar with stackoverflow and some of you have an account for this web site.

4.1 Input dataset

Stackoverflow dataset is published under an open creative common license. You might find the dataset on https://archive.org/details/stackexchange. The dataset contains all data including stackoverflow and the overall size of the archive is 27 gigabytes. The size of the uncompressed data is more than 1 Terabyte. If you are interested to download and extract the complete dataset you can visit this webpage: http://fullstackml.com/2015/10/29/beginners-guide-apache-spark-machine-learning-scenario-with-a-large-input-dataset.

In order to understand a little bit more the data format of the file let me show you how look like the XML version of one post, for example: http://stackoverflow.com/questions/4/when-setting-a-forms-opacity-should-i-use-a-decimal-or-double:



Let's take a look how the XML version of this stackoverflow question presented will look like in the file:

```
<ROW
      ID="4"
      POSTTYPEID="1"
      ACCEPTEDANSWERID="7"
      CREATIONDATE="2008-07-31T21:42:52.667"
      SCORE="322"
      VIEWCOUNT="21888"
      BODY="&LT;P&GT;I WANT TO USE A TRACK-BAR TO CHANGE A FORM'S
      OPACITY.&LT;/P&GT; &LT;P&GT;THIS IS MY CODE:&LT;/P&GT;
      &LT;PRE&GT;&LT;CODE&GT;DECIMAL TRANS = TRACKBAR1.VALUE / 5000;
      THIS.OPACITY = TRANS; &LT;/CODE&GT;&LT;/PRE&GT; &LT;P&GT;WHEN I
      TRY TO BUILD IT, I GET THIS ERROR:&LT;/P&GT; &LT;BLOCKQUOTE&GT;
      &LT;P&GT;CANNOT IMPLICITLY CONVERT TYPE 'DECIMAL' TO
      'DOUBLE'.&LT;/P&GT; &LT;/BLOCKQUOTE&GT; &LT;P&GT;I TRIED MAKING
      &LT;CODE&GT;TRANS&LT;/CODE&GT; A &LT;CODE&GT;DOUBLE&LT;/CODE&GT;,
      BUT THEN THE CONTROL DOESN'T WORK. THIS CODE HAS WORKED FINE FOR
      ME IN VB.NET IN THE PAST. &LT;/P&GT; "
      OWNERUSERID="8"
      LASTEDITORUSERID="451518"
      LASTEDITORDISPLAYNAME="RICH B"
      LASTEDITDATE="2014-07-28T10:02:50.557"
      LASTACTIVITYDATE="2014-12-20T17:18:47.807"
      TITLE="WHEN SETTING A FORM'S OPACITY SHOULD I USE A DECIMAL OR
      DOUBLE?"
      TAGS="&LT;C#&GT;&LT;WINFORMS&GT;&LT;TYPE-
      CONVERSION&GT;&LT;OPACITY&GT;"
      ANSWERCOUNT="13"
      COMMENTCOUNT="1"
      FAVORITECOUNT="27"
      COMMUNITYOWNEDDATE="2012-10-31T16:42:47.213"
   />
```

The file contains mainly body text which is a text of questions from the web site. In the file each post is presented by one single row. Note that because the text is HTML ,the opening and closing p tags (and) are written as <p> and </p> respectively.

This file contains the stackoverflow.com posts data as xml attributes:

- Title – post title
- Body – post text
- Tags – list of tags for post

10+ more xml-attributes that we won't use.

To simplify the task and reduce the amount of code, we are going to concatenate Title and Body and use that as a single text column.

For this hands-on we will use a small part of this dataset, only 100.000 lines of file Post.xml (133Mb of data build by Dimitry Petrov). This size is enough for experimenting in our hands-on. You can download the file from this web page: https://www.dropbox.com/s/n2skgloqoadpa30/Posts.small.xml?dl=0

4.2 Simple Predictive model as an example

Our goal is to create a predictive model that predicts post Tags based on Body and Title. It might be easy to imagine how this model should work in the stackoverflow.com web site – the user types a question and the web size automatically gives tags suggestion.

Assume that we need as many correct tags as possible and that the user would remove the unnecessary tags. Because of this assumption we are choosing recall as a high priority target for our model.

The real problem of stackoverflow tag prediction is a multi-label classification one because the model should predict many classes, which are not exclusive. The same text might be classified as “Java” and “Multithreading”.

In order to simplify this hands-on, instead of training a multi-label classifier, we will train a simple binary classifier for a given tag. For instance, for the tag “Java” one classifier will be created which can predict a post that is about the Java language.

By using this simple approach, many classifiers might be created for almost all frequent labels (Java, C++, Python, multi-threading etc…). This approach is simple and good for studying. However, it is not perfect in practice because by splitting predictive models by separate classifiers, you are ignoring the correlations between classes.

Different libraries need to be imported for our example with XML. For Spark data manipulation in our hands-on we need to import the following libraries:

```
import org.apache.spark.sql.catalyst.plans._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
```

Finally, the machine learning libraries required in this hands-on are:

```
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.classification
    .LogisticRegression
import org.apache.spark.mllib.evaluation
    .BinaryClassificationMetrics
```

```
import org.apache.spark.ml.Pipeline
```

4.3 Parsing XML

The scala.xml library. Scala can process XML literals and we don't need to put quotes around XML strings. We can just write them directly, and Scala will automatically interpret them as XML elements (of type scala.xml.Element). First we need to import it:

```
import scala.xml._
```

We need to extract Body, Text and Tags from the input xml file and create a single data-frame with these columns. Let's remove the xml header and footer (note locate the input file in the same directory where you run the spark shell command):

```
val fileName = "Posts.small.xml"
val textFile = sc.textFile(fileName)
val postsXml = textFile.map(_.trim).
   filter(!_.startsWith("<?xml version=")).
   filter(_ != "<posts>").
   filter(_ != "</posts>")
```

Note: Scala provides trim method in its string class which helps to remove both leading and trailing white spaces.

Scala automatically converts all xml codes like "<a>" to actual tags "<a>". Also we are going to concatenate title and body and remove all unnecessary tags and new line characters from the body and all space duplications.

```
val postsRDD = postsXml.map { s =>
   val xml = XML.loadString(s)

   val id = (xml \ "@Id").text
   val tags = (xml \ "@Tags").text

   val title = (xml \ "@Title").text
   val body = (xml \ "@Body").text
   val bodyPlain = ("<\\S+>".r).replaceAllIn(body, " ")
   val text = (title + " " + bodyPlain).replaceAll("\n",
      " ").replaceAll("( )+", " ");

   Row(id, tags, text)
}
```

We will use a DataFrame. In Spark, to create a DataFrame, a schema should be applied to RDD.

```
val schemaString = "Id Tags Text"
val schema = StructType(
   schemaString.split(" ").map(fieldName =>
      StructField(fieldName, StringType, true)))

val postsDf = sqlContext.createDataFrame(postsRDD, schema)
```

Now, in order to be sure that everything is fine, you can display the content of the DataFrame to stdout with:

```
postsDf.show()
```

5 Training and Testing the model

5.1 Preparing training datasets

For this code example, we are using “java” as a label that we would like to predict by a binary classifier. All rows with the “java” label should be marked as a “1” and rows with no “java” as a “0”. Let’s identify our target tag “java” and create binary labels based on this tag.

```
val targetTag = "java"
val myudf: (String => Double) = (str: String) =>
    {if (str.contains(targetTag)) 1.0 else 0.0}
val sqlfunc = udf(myudf)
val postsLabeled = postsDf.withColumn("Label",
    sqlfunc(col("Tags")) )
```

Dataset can be split into negative and positive subsets by using the new label.

```
val positive = postsLabeled.filter('Label > 0.0)
val negative = postsLabeled.filter('Label < 1.0)
```

We are going to use 90% of our data for the model training and 10% as a testing dataset. Let’s create a training dataset by sampling the positive and negative datasets separately.

```
val positiveTrain = positive.sample(false, 0.9)
val negativeTrain = negative.sample(false, 0.9)
val training = positiveTrain.unionAll(negativeTrain)
```

In order to be sure that everything is going fine I suggest to inspect the DataFrames content:

```
negative.show()
positive.show()
```

5.2 Preparing testing datasets

The testing dataset step should include all rows that are not included in the training datasets, and again, positive and negative examples separately.

```
val negativeTrainTmp = negativeTrain
    .withColumnRenamed("Label", "Flag").select('Id, 'Flag)

val negativeTest = negative.join( negativeTrainTmp,
    negative("Id") === negativeTrainTmp("Id"),
    "LeftOuter").filter("Flag is null")
    .select(negative("Id"), 'Tags, 'Text, 'Label)

val positiveTrainTmp = positiveTrain
    .withColumnRenamed("Label", "Flag")
    .select('Id, 'Flag)

val positiveTest = positive.join( positiveTrainTmp,
    positive("Id") === positiveTrainTmp("Id"),
    "LeftOuter").filter("Flag is null")
    .select(positive("Id"), 'Tags, 'Text, 'Label)

val testing = negativeTest.unionAll(positiveTest)
```

Again, if you will check that everything is going fine you can execute:

```
testing.show()
```

Remember that transformations in Spark are “lazy” and are only computed when an action requires a result to be returned to the driver program. For this reason executing `testing.show()` requires many seconds.

5.3 Training a model

The `ml.feature` package provides common feature transformers that help convert raw data or features into more suitable forms for model fitting, e.g., `HashingTF`. That maps a sequence of terms to their term frequencies using the hashing trick.

Spark API creates a model based on columns from the DataFrame and the training parameters. We will consider the Logistic regression. Currently, this class only supports binary classification (It will support multiclass in the future).

Going into more detail with the classifier model is out of scope of this hands-on, due we are centering it in XML format, we are only centering the explanation on XML (however I prefer to use a real cases for using in this hands-on).

The code that we can execute in our computer in order to obtain the model is:

```
val numFeatures =   64000
val numEpochs =     30
val regParam =      0.02

val tokenizer = new Tokenizer().setInputCol("Text")
    .setOutputCol("Words")

val hashingTF = new  org.apache.spark.ml.feature
    .HashingTF().setNumFeatures(numFeatures).

    setInputCol(tokenizer.getOutputCol)
    .setOutputCol("Features")

val lr = new LogisticRegression().setMaxIter(numEpochs)
    .setRegParam(regParam)setFeaturesCol("Features")
    .setLabelCol("Label").setRawPredictionCol("Score")
    .setPredictionCol("Prediction")

val pipeline = new Pipeline()
    .setStages(Array(tokenizer, hashingTF, lr))

val model = pipeline.fit(training)
```

5.4 Testing a model

Now we have our `model` that is our binary "Java" classifier, which returns a prediction with values 0.0 or 1.0. Try with this example:

```
val testTitle =
 "Easiest way to merge a release into one JAR file"

val testBoby =
 """Is there a tool or script which easily merges a bunch
 of href="http://en.wikipedia.org/wiki/JAR_%28file_format
 %29" JAR files into one JAR file? A bonus would be to
 easily set the main-file manifest and make it executable.
 I would like to run it with something like: As far as I
 can tell, it has no dependencies which indicates that it
 shouldn't be an easy single-file tool, but the downloaded
 ZIP file contains a lot of libraries."""


val testText = testTitle + testBody

val testDF = sqlContext
   .createDataFrame(Seq( (99.0, testText)))
   .toDF("Label", "Text")

val result = model.transform(testDF)

val prediction = result.collect()(0)(6)
   .asInstanceOf[Double]

print("Prediction: "+ prediction)
```

In order to check the quality of the prediction, one metric that we could use is the area under the ROC curve. The ROC curve is a graphical plot that illustrates the performance of a binary classifier system.

If you use the small dataset then the quality of your model is probably not the best. Area under the ROC value will be very low (close to 50%) which indicates a poor quality of the model. With an entire Posts.xml dataset, the quality will be better. The code that evaluate the quality of the model based on the training dataset could be:

```
val testingResult = model.transform(testing)

val testingResultScores = testingResult
   .select("Prediction", "Label").rdd
   .map(r => (r(0).asInstanceOf[Double], r(1)
   .asInstanceOf[Double]))

val bc =
   new BinaryClassificationMetrics(testingResultScores)

val roc = bc.areaUnderROC

print("Area under the ROC:" + roc)
```

6 Python code

Python solution looks similar to the last Scala solution because when you look “under the hood” you have the same Spark library and engine. As there aren’t many difference between Python and Scala, I will highlight only the major ones and you can refer back to the last post for the code in it’s entirety. Enclosed you will find the code in Python.

6.1 JSON

```
sqlContext = SQLContext(sc)
photos = sqlContext.jsonFile("example3.txt")
def show(x): print x
photos.foreach(show)

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



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

6.2 XML

```
import re
from pyspark.sql.types import *
from pyspark.sql.functions import *

from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.mllib.evaluation import
BinaryClassificationMetrics

targetTag = "java"
textFile = sc.textFile("Posts.small.xml")
```

```
postsXml = textFile.map( lambda line: line.strip() ).\
    filter( lambda line: line != "<posts>" and line !=
"</posts>").\
    filter( lambda line: not line.startswith("<?xml
version=") ).\
    filter( lambda line: line.find("Id=") >= 0 ).\
    filter( lambda line: line.find("Tags=") >= 0 ).\
    filter( lambda line: line.find("Body=") >= 0 ).\
    filter( lambda line: line.find("Title=") >= 0 )

postsRDD = postsXml.map( lambda s: pyspark.sql.Row(\
                Id = re.search('Id=".+?"', s).group(0)[4:-
1].encode('utf-8'),\
                Label = 1.0 if re.search('Tags=".+?"', s) !=
None\
                            and re.search('Tags=".+?"',
s).group(0)[6:-1].encode('utf-8').find(targetTag) >= 0 else
0.0,\
                Text = (re.search('Title=".+?"',
s).group(0)[7:-1] if re.search('Title=".+?"', s) != None
else "" + " " +\
                        re.search('Body=".+?"',
s).group(0)[6:-1]) if re.search('Body=".+?"', s) != None
else ""))

postsLabeled = sqlContext.createDataFrame(postsRDD)

positive = postsLabeled.filter(postsLabeled.Label > 0.0)
negative = postsLabeled.filter(postsLabeled.Label < 1.0)

# Sample without replacement (False)
positiveTrain = positive.sample(False, 0.9)
negativeTrain = negative.sample(False, 0.9)
training = positiveTrain.unionAll(negativeTrain)

negTrainTmp1 = negativeTrain.withColumnRenamed("Label",
"Flag")
negativeTrainTmp = negTrainTmp1.select(negTrainTmp1.Id,
negTrainTmp1.Flag)


negativeTest = negative.join( negativeTrainTmp, negative.Id
== negativeTrainTmp.Id, "LeftOuter").\
                        filter("Flag is null").\
                        select(negative.Id, negative.Text,
negative.Label)
```

```
posTrainTmp1 = positiveTrain.withColumnRenamed("Label",
"Flag")
positiveTrainTmp = posTrainTmp1.select(posTrainTmp1.Id,
posTrainTmp1.Flag)

positiveTest = positive.join( positiveTrainTmp, positive.Id
== positiveTrainTmp.Id, "LeftOuter").\
                        filter("Flag is null").\
                        select(positive.Id, positive.Text,
positive.Label)
testing = negativeTest.unionAll(positiveTest)


# CREATE MODEL
numFeatures = 20000
numEpochs = 20
regParam = 0.02

tokenizer =
Tokenizer().setInputCol("Text").setOutputCol("Words")
hashingTF = HashingTF().setNumFeatures(numFeatures).\

setInputCol(tokenizer.getOutputCol()).setOutputCol("Features
")
lr =
LogisticRegression().setMaxIter(numEpochs).setRegParam(regPa
ram).\

setFeaturesCol("Features").setLabelCol("Label").\

setRawPredictionCol("Score").setPredictionCol("Prediction")
pipeline = Pipeline().setStages([tokenizer, hashingTF, lr])


model = pipeline.fit(training)
```

```
testTitle = "Easiest way to merge a release into one JAR
file"
testBody = """Is there a tool or script which easily merges
a bunch of

href=&quot;http://en.wikipedia.org/wiki/JAR_%28file_format%2
9&quot;
    &gt;JAR&lt;/a&gt; files into one JAR file? A bonus would
be to easily set the main-file manifest
    and make it executable. I would like to run it with
something like:
    &lt;/p&gt;&#xA;&#xA;&lt;blockquote&gt;&#xA;
&lt;p&gt;java -jar

rst.jar&lt;/p&gt;&#xA;&lt;/blockquote&gt;&#xA;&#xA;&lt;p&gt;
    As far as I can tell, it has no dependencies which
indicates that it shouldn't be an easy
    single-file tool, but the downloaded ZIP file contains a
lot of libraries."""
testText = testTitle + testBody
testDF = sqlContext.createDataFrame([ ("0", testText, 1.0)],
["Id", "Text", "Label"])
result = model.transform(testDF)
prediction = result.collect()[0][7]
print("Prediction: ", prediction)


testingResult = model.transform(testing)
testingResultScores = testingResult.select("Prediction",
"Label").rdd.map( lambda r: (float(r[0]), float(r[1])))
bc = BinaryClassificationMetrics(testingResultScores)
roc = bc.areaUnderROC
print("Area under the ROC:",  roc)
```

7 Exercise

Find/create four examples of stackoverflow questions, which are classified into two different groups by a binary classifier of the tag “Spark”.

You have one week to deliver a report with the results, including a description of the input, the code and the output.