UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Big Data Management (BDMA & MIRI Masters)
Session: Apache Spark (Training)

Sergi Nadal

The purpose of this document is that you get familiar with the environment and tools required for the lab session. This is an optional task, thus no delivery is expected and no grade will be assigned. In this session we will explore Apache Spark, nowadays the most popular Big Data processing engine. We will experiment with its core functionalities for processing batches of data that are, for instance, stored in files. You are free to check any external reference, for instance the Spark programming guide (`https://spark.apache.org/docs/latest/rdd-programming-guide.html`).

If you encounter any problems during the preparation of the training, please contact by email the lecturer to clarify any issues before the lab session (`petar@essi.upc.edu`).

Instructions

In this session, as Spark jobs will run locally, you do not need to start the Spark service and everything can be run from IntelliJ or Eclipse. Once you have fetched the code from LearnSQL, it can be imported as a *Maven Project* into eclipse. All executions can be run locally, and thus you do not need to be concerned with the host connection. In the *Main* class (in package *bdm.lab3*) you can see that different arguments are expected, and will drive the behaviour of the program. The simplest way to define such arguments is by creating a *Run Configuration* of type *Java Application* from the *Run* menu. In the tab menu called *Arguments* you will be able to edit such arguments, in each exercise you will be guided with the required arguments.

Java reminder

We will first recall on some advanced Java topics that might come handy when programming in Spark. Note that they will not be necessarily present in the lab session. If you are already familiar with this concepts you might skip this section.

Data structures manipulation with Guava

Notice: The content and tasks contained in this section are inspired or have been extracted from the following book: *Bejeck, B. (2013). Getting Started with Google Guava. Packt Publishing Ltd.* All credit goes to the authors.

Guava is originally from 2007, where it started as the *Google Collections Library*, which provided utilities for working with Java collections. The Google Guava project has evolved into being an essential toolkit for developers working in Java. There is something for everyone in Guava. There are classes for working with strings, collections, concurrency, I/O, and reflection. Here, we will focus in the management of such collection data structures. A complete reference on Guava's support for data structures is available in `https://github.com/google/guava/wiki/CollectionUtilitiesExplained`.

Lists

A list (`https://docs.oracle.com/javase/8/docs/api/java/util/List.html`) is an ordered collection whose elements can be accessed by their integer index. In Guava (`https://google.github.io/guava/releases/23.0/api/docs/com/google/common/collect/Lists.html`) a list of elements of type `T` is defined as:

```
List<String> emptyList = Lists.newArrayList();
emptyList.add("A");
String first = emptyList.get(0);
```

A relevant method is the `Lists.partition()`, which returns sublists of size n from a given list. For instance, assume we have created a list with four elements with the static method that populates as many instances as those in the parameters:

```
List<String> aList = Lists.newArrayList("A","B","C","D");
```

Then, we can call the `Lists.partition()` method specifying two partitions:

```
List<List<String>> partitions = Lists.partition(aList,2)
```

In this example, the list `partitions` would contain `[[A,B],[C,D]]`. The method returns consecutive sublists of the same size, with the exception of the last sublist, which may be smaller.

Sets

A set (`https://docs.oracle.com/javase/8/docs/api/java/util/Set.html`) is an unordered collection that contains no duplicate elements. In Guava (`https://google.github.io/guava/releases/23.0/api/docs/com/google/common/collect/Sets.html`) a set of elements of type `T` is defined as:

```
Set<String> emptySet = Sets.newHashSet();
emptySet.add("A");
String first = emptyList.get(0);
```

Guava provides methods from set theory to operate with sets.

Difference The `Sets.difference` method takes two set instance parameters and returns a `SetView` of the elements found in the first set, but not in the second. `SetView` is a static, abstract inner class of the `Sets` class and represents an unmodifiable view of a given `Set` instance. Any elements that exist in the second set but not in the first set are not included.

Symmetric difference The `Sets.symmetricDifference` method returns elements that are contained in one set or the other set, but not contained in both. The returned set is an unmodifiable view `SetView`.

Intersection The `Sets.intersection` method returns an unmodifiable `SetView` instance containing elements that are found in two `Set` instances.

Union The `Sets.union` method takes two sets and returns a `SetView` instance that contains elements that are found in either set.

Maps

A map (`https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`) is a collection that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value (i.e., an object, which in general can be of an arbitrary complexity). In Guava (`https://google.github.io/guava/releases/23.0/api/docs/com/google/common/collect/Maps.html`) a map is defined as follows:

```
Map<String,String> emptyMap = Maps.newHashMap();
emptyMap.put("k1","A");
emptyMap.put("k2","B");
List<String> AandB = Lists.newArrayList(emptyMap.values());
```

Lambda expressions

Notice: The content and tasks contained in this section are inspired or have been extracted from the following book: *Warburton, R. (2014). Java 8 Lambdas: functional programming for the masses. O'Reilly.*. All credit goes to the authors.

Lambda expressions, also known as anonymous functions, allow to write quick throw away functions (one use) without naming them. Here is an expression, which tests if the parameter is even.

```
private boolean isEven(int x) {
       return x % 2 == 0;
}
boolean result = isEven(4);
```

Now, here is the same method implemented as a lambda expression.

```
Predicate<Integer> pred = x -> x % 2 == 0;
boolean result = pred.test(4);
```

We will mainly use lambda expressions with iterators over collections. For instance, the following code prints the length of each element in the list:

```
List<String> s = Lists.newArrayList("AAA","AA","A");
s.forEach(t -> System.out.println(t.length()));
```

Note that in the previous expression the variable `t` is just a name we decide to give to each `String` to be evaluated. Also, if your lambda expression is longer than a one-line expression, then it needs to be encapsulated within brackets { }. Now, instead of just printing content in the console we might want to generate a new list with such lengths for further processing. Note that this is not possible using the `forEach` method. But first we need to introduce the `streams` interface.

Streams allow us to write collections-processing code at a higher level of abstraction. The `Stream` interface (`https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`) contains a series of functions, each corresponding to a common operation that you might perform on a `Collection`. Streams allow us to perform *internal iteration*, which allows to focus only on the function we want to apply to the elements of a collection and dismiss the boilerplate (i.e., extra code) to iterate over it. Thus, the method that returns a list of integers containing the length of each elements would look as follow:

```
List<String> s = Lists.newArrayList("AAA","AA","A");
List<Integer> lenghts = s.stream().map(t -> t.length())
                                  .collect(Collectors.toList());
```

Note we have used the `map` function, whose semantics are that given an element of type `T` it returns an element of type `R`. In this case, our `map` receives `String` and returns `Integer`. Also, note that we have not specified to the function its return type, it is automatically inferred. Some common `stream` operators are:

- `collect(toList())`: generates a list from the values in a `Stream`.
- `filter`: the central idea of filter is to retain some elements of the `Stream`, while throwing others out. It must return either true or false for a given value
- `flatMap`: lets you replace a value with a `Stream` and concatenates all the streams together.

- `max` and `min`: which are implemented using `Comparators` where you encode the logic of finding the maximum or minimum on a given `Stream`.

The complete set of operators supported over streams is available at `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`.

Apache Spark

Warmup

To start, let's take a look at class `Exercise_1.java`, you do not need to implement anything here, as the code is already provided for you. Such method provides basic analysis on top of the `wines` file we used in previous sessions. Note that, for simplicity, a sample of 10MB of such file is provided in the *resources* folder.

MapReduce in Spark

In `Exercise_2.java`, you are supposed to implement the group by and aggregation functions on top of the same `wines` file. You may follow a similar approach to that in the MapReduce session. Precisely, you have to return the sum of attribute *ash* (position 5) grouped by *type* (position 1).

Predicting risk of cancer using kNN

In this exercise we are going to use Spark to implement the classification of breast tumors as malign or benign. We will follow the same approach as in the *MapReduce* session, thus we will perform predictions using the one nearest neighbor (1NN) algorithm. Next, we provide you with the high-level guidelines on how to implement the algorithm, for the details refer to the previously mentioned session.

1. Split the train and test sets
2. Generate all possible combinations of instances in the training and test sets
3. For each combination of training and test instances:
    (a) Calculate the euclidean distance of their predictor variables
4. For each test instance:
    (a) Use as prediction that from the training set that has the minimum euclidean distance.
5. Generate the confusion matrix

Similarly as the previous session, we provide you with the *Utils1_NN.java* class which contains methods to obtain the values of a given attribute and to obtain the list of predictors.

Your task consists on implementing the complete dataflow 1NN and generation of the confusion matrix in a single Spark program. Use class *Exercise_3.java* to write your implementation, the expected output is the confusion matrix as below:

```
B_B           94
B_M           5
M_B           4
M_M           57
```