Aims:

This exercise aims to get you to:

1. Practice with the use of Apache Spark and Spark-Submit
2. Implement and build a stand-alone Spark application using sbt
3. Run your stand-alone Spark application using Spark-Submit
4. Practice with the use of Apache Spark MLlib for running Machine Learning tasks

PART I: Implementing a stand-alone Spark Application

Running sbt to check the installed version

sbt is an open-source build tool for Scala and Java projects (similar to Ant and Maven). It allows you to build and package Scala projects into Jars that can be run by a JVM. sbt is installed on the lab computers. In order to run the tool, first you need to open a terminal in VLab and run the following program:

```
$ 9313
```

This program will setup the environment needed build your Spark application and run it with spark-submit. You should see an output like the following:

```
$ 9313
Welcome to COMP9313!
newclass starting new subshell for class COMP9313...
```

Next, run the command to check the installed version (notice that running `sbt` for the first time may take some time).

```
$ sbt sbtVersion
...
[info]   Loading   project   definition   from   /tmp_amd/cage/export/cage/3/z9999999/sbt-
workspace/project
[info]      Set      current      project      to      sbt-workspace      (in      build
file:/tmp_amd/cage/export/cage/3/z9999999/sbt-workspace/)
[info] 1.2.8
```

If everything goes fine, you should see an output similar to the one above (the installed version of `sbt` is 1.2.8).

Create your sbt project for the wordcount example

Next, we are going to create an `sbt` project for the stand-alone version of the wordcount example. In order to do this, first create a directory where you will keep your project files, and change to that directory:

```
$ mkdir WordCount

$ cd WordCount
```

Inside this directory, create and save a file named `build.sbt` with the following content:

```
name := "wordcount"
version := "1.0"
scalaVersion := "2.11.12"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.3"
```

The file `build.sbt` contains the details of your project, including a name (name), version of your application (`version`), Scala version (`scalaVersion`) used in your application, and the dependencies (`libraryDependencies`) on which your project relies.

Next, create the directory where your source code file will be stored. To do so, you can use the following command (run this command from inside the project directory `WordCount` you just created in the previous step). Then change to this newly created directory:

```
$ mkdir -p src/main/scala

$ cd src/main/scala
```

Inside the directory above, create and save the file `wordcount.scala` with the following code:

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

object wordcount {
  def main(args: Array[String]) {
    val inputFile = args(0)
    val outputFolder = args(1)
    val conf = new SparkConf().setAppName("wordcount").setMaster("local")
    // Create a Scala Spark Context.
    val sc = new SparkContext(conf)
    // Load our input data.
    val input = sc.textFile(inputFile)
    // Split up into words.
    val words = input.flatMap(line => line.split(" "))
    // Transform into word and count.
    val counts = words.map(word => (word, 1)).reduceByKey(_+_)
    counts.saveAsTextFile(outputFolder)
  }
}
```

Building your sbt project

In order to build your `sbt` project you will need to change to the directory where your `build.sbt` file is located (replace the path below with the full path of your `WordCount` directory):

```
$ cd /import/cage/3/z9999999/WordCount
```

Next, run the program `sbt package` to build and package your project:

```
$ sbt package
```

The command above will run sbt and will start packaging your project (it may take some time). If everything goes well you should see an output similar to the following:

```
$ sbt package
...
[info]   Loading   project   definition   from   /tmp_amd/cage/export/cage/3/z3519822/sbt-workspace/WordCount/project
[info] Loading settings for project wordcount from build.sbt ...
[info]    Set    current    project    to    Word    Count    (in    build file:/tmp_amd/cage/export/cage/3/z3519822/sbt-workspace/WordCount/)
[success] Total time: 2 s, completed 08/07/2019 2:30:32 PM
```

If the command above is completed successfully, you should end up with a directory (within your WordCount directory) named target. The actual jar file that contains your project/application can be found within this folder (see the file word-count_2.11-1.0.jar below):

```
$ ls target/scala-2.11
classes  resolution-cache  wordcount_2.11-1.0.jar
```

Running your Spark stand-alone application with spark-submit

You are now ready to run your Spark stand-alone application using spark-submit.

spark-submit is the program used by Apache Spark to submit jobs (applications) to the cluster (if you have one) for their execution. Assuming you currently are in the WordCount directory (your project's directory), run the spark-submit command as shown below:

```
$ spark-submit --class "wordcount" --master local[2] target/scala-2.11/wordcount_2.11-1.0.jar test-input.txt wc-test-output
```

In the command above you will need to replace the first argument (test-input.txt) with the full path of the input text file on which the word count will be performed (you can create a test input file yourself), and the second argument (wc-test-output) with the full path of the output directory where Spark will write the output.

PART II: Using Spark's MLlib

In this part of this lab, we will practice with the use of MLlib for classification. More specifically, we will focus on Logistic Regression, which is implemented as one of the MLlib libraries in Spark.

For this exercise we will train a Logistic Regression model for the task of classifying iris flowers based on their botanical characteristics. We will use an excerpt of the original dataset published in the UCI Machine Learning Repository. For this exercise, we will consider only two species of flowers, namely, *setosa* and *versicolor*. The CSV file containing the training data can be found here:

https://webcms3.cse.unsw.edu.au/COMP9313/19T2/resources/28578

In this exercise we will be using Spark-shell to build our Logistic Regression model step-by-step. First, we need to import all the libraries that we will need to train our model:

```
scala> import org.apache.spark.sql.SQLContext

scala> import org.apache.spark.ml.classification.LogisticRegression

scala> import org.apache.spark.ml.feature.VectorAssembler
```

The first import (SQLContext) provides the functionalities needed to load the CSV file containing our training data and convert it into a DataFrame to be used with the Logistic Regression library (which correspond to the second import in the code above). The third import provides functionalities to transform our DataFrame into the correct format needed by the Logistic Regression library.

In the next step, we will create a SQLContext object and load the CSV file containing our training data:

```
scala> val sqlContext = new SQLContext(sc)

scala> val df =
sqlContext.read.format("csv").option("header","true").option("inferSchema","true"
).load("FULL_PATH_OF_TRAINING_DATA_FILE")
```

The code above loads the data provided in the string (path) provided to the load() method. Here, you will need to replace with the full path (`FULL_PATH_OF_TRAINING_DATA_FILE`) of the iris dataset in the local file system. The code above produces as result a DataFrame (`df`) that contains our training dataset. You can show the content of `df` with the code below (you should get an output similar to one shown below):

```
scala> df.show(10)

+------------+-----------+------------+-----------+-------+
|sepal_length|sepal_width|petal_length|petal_width|species|
+------------+-----------+------------+-----------+-------+
|         5.1|        3.5|         1.4|        0.2|      0|
|         4.9|        3.0|         1.4|        0.2|      0|
|         4.7|        3.2|         1.3|        0.2|      0|
|         4.6|        3.1|         1.5|        0.2|      0|
|         5.0|        3.6|         1.4|        0.2|      0|
|         5.4|        3.9|         1.7|        0.4|      0|
|         4.6|        3.4|         1.4|        0.3|      0|
|         5.0|        3.4|         1.5|        0.2|      0|
|         4.4|        2.9|         1.4|        0.2|      0|
|         4.9|        3.1|         1.5|        0.1|      0|
+------------+-----------+------------+-----------+-------+

only showing top 10 rows
```

The library we use for building a Logistic Regression model requires that the input dataset includes two columns: features and label. Since our original dataset does not include such columns, we need to create them with the code below:

```
scala> val vecAssemb = new VectorAssembler().setInputCols(Array("sepal_length",
"sepal_width", "petal_length", "petal_width")).setOutputCol("features")

scala> val df2 = vecAssemb.transform(df)

scala> val df3 = df2.withColumnRenamed("species", "label")

scala> val df4 =
df3.drop("sepal_length").drop("sepal_width").drop("petal_length").drop("petal_wid
th")
```

The first two lines of code above packages the original columns `sepal_length`, `sepal_width`, `petal_length` and `petal_width` into a single column named `features`. The third line renames the column `species` to `label`. Finally, the last column deletes the original columns used for features so that we keep only the two required columns (`features` and `label`) in the DataFrame.

We are now ready to train our Logistic Regression model, which can be done with the code below:

```
scala> val logReg = new LogisticRegression()

scala> val model = logReg.fit(df4)
```

After executing the code shown above, `model` contains the resulting Logistic Regression model, which is now ready to make predictions (i.e. classify data). Below we run the predictions on the very same dataset we used for training purposes:

```
scala> val predict = model.transform(df4)

scala> val predictSummary = model.transform(df4).select("features","prediction",
"label")

scala> predictSummary.show(100)
```

If everything goes well, you should see an output similar to the one shown below:

```
+-----------------+----------+-----+
|         features|prediction|label|
+-----------------+----------+-----+
|[5.1,3.5,1.4,0.2]|       0.0|    0|
|[4.9,3.0,1.4,0.2]|       0.0|    0|
|[4.7,3.2,1.3,0.2]|       0.0|    0|
|[4.6,3.1,1.5,0.2]|       0.0|    0|
|[5.0,3.6,1.4,0.2]|       0.0|    0|
...
|[5.7,2.9,4.2,1.3]|       1.0|    1|
|[6.2,2.9,4.3,1.3]|       1.0|    1|
|[5.1,2.5,3.0,1.1]|       1.0|    1|
|[5.7,2.8,4.1,1.3]|       1.0|    1|
+-----------------+----------+-----+
```

Notice that our Logistic Regression model was able to properly classify (predict) each tuple (the values of the `prediction` and `label` columns are identical). Notice that this is just a quick test, and, in more realistic scenarios, you would need to use other techniques (such as cross-validation) to measure the performance of the resulting classification model.