EE412 Foundation of Big Data Analytics, Fall 2021
HW0*

Due date: September 24, 2021 (11:59pm)

Submission instructions: Use KAIST KLMS to submit your homeworks. Your submission should be one gzipped tar file whose name is `YourStudentID_hw0.tar.gz`. For example, if your student ID is 20210000, and it is for homework #0, please name the file as `20210000_hw0.tar.gz`. You can also use these extensions: tar, gz, zip, tar.zip. Do not use other options not mentioned here.

Your zip file should contain **two** things; one python file (`hw0.py`) and the Ethics Oath pdf file. Before zipping your files, please make a directory named `YourStudentID_hw0` and put your files in the directory. Then, please compress the directory to make a zipped file. **If you violate any of the file name format or extension format, we will deduct 1 point of total score per mistake.**

Ethics Oath: For every homework submission, please fill out and submit the **PDF** version of this document that pledges your honor that you did not violate any ethics rules required by this course and KAIST. You can either scan a printed version into a PDF file or make the Word document into a PDF file after filling it out. Please sign on the document and submit it along with your other files.

Discussions with other people are permitted and encouraged. However, when the time comes to write your solution, such discussions (except with course staff members) are no longer appropriate: you must write down your own solutions independently. If you received any help, you must specify on the top of your written homework any individuals from whom you received help, and the nature of the help that you received. *Do not, under any circumstances, copy another person's solution.* We check all submissions for plagiarism and take any violations seriously.

Purpose of this tutorial: Here you will learn how to write, compile, debug, and execute a simple Spark program. The first part (Section 1–4) of the homework assignment serves as a tutorial, and the second part (Section 5) asks you to write your own Spark program.

*Material adapted from Stanford University CS246.

OVERVIEW

- Section 1: How to download and install a stand-alone Spark instance. All operations done in this Spark instance will be performed against the files in your local file system. *Optional* – You may setup Spark by using Google Cloud Dataproc (instructions). As announced in class, please contact our staff if you want to use this option.
- Section 2: How to launch the Spark shell for interactively building Spark applications.
- Section 3: How to use Spark to launch Spark applications written in an IDE or editor.
- Section 4: An example of writing a simple word count application for Spark.
- Section 5: The actual homework assignment. There are no deliverables for sections 2, 3, and 4. In section 5, you are asked to write and submit your own Spark application based on the word count example.

1 Setting up a stand-alone Spark instance

- Download and install Spark 3.1.2 on your machine:
  https://www.apache.org/dyn/closer.lua/spark/spark-3.1.2/spark-3.1.2-bin-hadoop2.7.tgz
- Unpack the compressed TAR ball:
  tar -zxvf spark-3.1.2-bin-hadoop2.7.tgz

Spark requires **JDK 8** (after version 8u92). Do not install JDK 9; Spark is currently incompatible with JDK 9. For installing JDK, we can follow one of the two ways:

- On Linux or Mac, you can run the following commands for installing Open JDK 8 (Ref: https://docs.datastax.com/en/jdk-install/doc/jdk-install/installOpenJdkDeb.html):

  ```
  $ sudo apt-get update
  $ sudo apt-get install openjdk-8-jdk
  ```

  On Windows, see the link: https://techoral.com/blog/java/openjdk-install-windows.html.
- If you want to directly download the JDK, please visit Oracle's site:
  http://www.oracle.com/technetwork/java/javase/downloads/index.html.

The Haedong Lounge machines already have JDK 8 installed (you can check by typing `java -version`).

Also, you will need **Python 3.6 or higher**. You can get information about installing Python here. Please refer to other Python setting instructions, which are available online.

In case you want to use Ubuntu Linux using **VMware**, here are some instructions.

NOTE: These commands are just guidelines, and what you have to add to your file may vary depending on your specific computer setup. See more details about Spark requirements in here.

2 Running the Spark shell

Spark gives you two different ways to run your applications. The easiest is using the Spark shell, a Read–Eval–Print Loop (REPL) that lets you interactively compose your application. Spark supports Python, which we will use as our language.

To start the Spark shell for Python, do the following:

- Open a terminal window on Linux or Mac or a command window on Windows.
- Change into the directory where you unpacked the Spark binary (i.e., Go into the directory '/spark-3.1.2-bin-hadoop2.7'.)
- Run the following command.
  On Linux or Mac:
  `$ bin/pyspark`
  On Windows:
  `$ bin\pyspark`

As the Spark shell starts, you may see large amounts of logging information displayed on the screen, possibly including several warnings. You can ignore that output for now. Regardless, the startup is complete when you see something like:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.1.2
      /_/

Using Python version 3.9.6 (default, Aug 18 2021 19:38:01)
SparkSession available as 'spark'.
>>>
```

The Spark shell is a full python interpreter and can be used to write and execute regular python programs. For example:

```
>>> print "Hello!"
Hello!
```

The Spark shell can also be used to write Spark applications in python. To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/latest/rdd-programming-guide.html

3 Submitting Spark applications

The Spark shell is great for exploring a data set or experimenting with the API, but it's often best to write your Spark applications outside of the Spark interpreter using an IDE or other smart editor (e.g., emacs, vim). One of the advantages of this approach for this class is that you will have created a submittable file that contains your application,

rather than having to piece it together from the Spark shell's command history.

Python is a convenient choice of language as your Python application does not need to be compiled or linked. Assume you have the following program in a text file called myapp.py:

```
import sys
from pyspark import SparkConf, SparkContext

conf = SparkConf()
sc = SparkContext(conf=conf)
print("%d lines" % sc.textFile(sys.argv[1]).count())
```

This short application opens the file path given as the first argument from the local working directory and prints the number of lines in it. To run this application, do the following:

- Open a terminal window on Linux or Mac or a command window on Windows.
- Change into the directory where you unpacked the Spark binary (i.e., Go into the directory '/spark-3.1.2-bin-hadoop2.7'.)
- Run the following command.
  On Linux or Mac:
  ```
  $ bin/spark-submit YourPathTo/myapp.py YourPathTo/pg100.txt
  ```
  On Windows:
  ```
  $ bin\spark-submit YourPathTo\myapp.py YourPathTo\pg100.txt
  ```

See Section 4 for where to get `pg100.txt`.

As Spark starts, you may see large amounts of logging information displayed on the screen, possibly including several warnings. You can ignore that output for now. Regardless, near the bottom of the output you will see the output from the application:

```
21/09/01 13:28:37 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1)
in 1330 ms on 143.248.41.178 (executor driver) (1/2)
21/09/01 13:28:37 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 1355 ms on 143.248.41.178 (executor driver) (2/2)
21/09/01 13:28:37 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have
all completed, from pool
21/09/01 13:28:37 INFO PythonAccumulatorV2: Connected to AccumulatorServer at host:
127.0.0.1 port: 57459
21/09/01 13:28:37 INFO DAGScheduler: ResultStage 0
(count at /home/yuji/EE412/codes/myapp.py:6) finished in 1.460 s
21/09/01 13:28:37 INFO DAGScheduler: Job 0 is finished. Cancelling potential
speculative or zombie tasks for this job
21/09/01 13:28:37 INFO TaskSchedulerImpl: Killing all running tasks in stage 0:
Stage finished
21/09/01 13:28:37 INFO DAGScheduler: Job 0 finished: count at
/home/yuji/EE412/codes/myapp.py:6, took 1.546809 s
```

```
124787 lines
21/09/01 13:28:37 INFO SparkUI: Stopped Spark web UI at http://dilabserver:4040
21/09/01 13:28:37 INFO MapOutputTrackerMasterEndpoint:
MapOutputTrackerMasterEndpoint stopped!
21/09/01 13:28:37 INFO MemoryStore: MemoryStore cleared
21/09/01 13:28:37 INFO BlockManager: BlockManager stopped
21/09/01 13:28:37 INFO BlockManagerMaster: BlockManagerMaster stopped
21/09/01 13:28:37 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
OutputCommitCoordinator stopped!
21/09/01 13:28:37 INFO SparkContext: Successfully stopped SparkContext
21/09/01 13:28:37 INFO ShutdownHookManager: Shutdown hook called
21/09/01 13:28:37 INFO ShutdownHookManager: Deleting directory
/tmp/spark-263fac70-4d0f-4d4b-b239-4b1ed7e9f07b
21/09/01 13:28:37 INFO ShutdownHookManager: Deleting directory
/tmp/spark-263fac70-4d0f-4d4b-b239-4b1ed7e9f07b/pyspark-969d077c-1052-480c-8301-10dc9
21/09/01 13:28:37 INFO ShutdownHookManager: Deleting directory
/tmp/spark-0453726d-b0f8-4057-99d1-8250daca3b63
```

Executing the application this way causes it to be run single-threaded. To run the application with 4 threads, launch it as
`bin/spark-submit --master 'local[4]' YourPathTo/myapp.py YourPathTo/pg100.txt`.
You can replace the "4" with any number. To use as many threads as are available on your system, launch the application as
`bin/spark-submit --master 'local[*]' YourPathTo/myapp.py YourPathTo/pg100.txt`.
To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/latest/rdd-programming-guide.html

4 Word Count

The typical "Hello, world!" app for Spark applications is known as word count. The map/reduce model is particularly well suited to applications like counting words in a document. In this section, you will see how to develop a word count application in Python. Prior to reading this section, you should read through the Spark programming guide if you haven't already.

All operations in Spark operate on data structures called RDDs, Resilient Distributed Datasets. An RDD is nothing more than a collection of objects. If you read a file into an RDD, each line will become an object (a string, actually) in the collection that is the RDD. If you ask Spark to count the number of elements in the RDD, it will tell you how many lines are in the file. If an RDD contains only two-element tuples, the RDD is known as a "pair RDD" and offers some additional functionality. The first element of each tuple is treated as a key, and the second element as a value. Note that all RDDs are immutable, and any operations that would mutate an RDD will instead create a new RDD.

For this example, you will create your application in an editor instead of using the Spark shell. The first step of every such Spark application is to create a Spark context:

```
import re
import sys
from pyspark import SparkConf, SparkContext

conf = SparkConf()
sc = SparkContext(conf=conf)
```

Next, you'll need to read the target file into an RDD:

```
lines = sc.textFile(sys.argv[1])
```

You now have an RDD filled with strings, one per line of the file.

Next you'll want to split the lines into individual words:

```
words = lines.flatMap(lambda l: re.split(r'[^\w]+', l))
```

The flatMap() operation first converts each line into an array of words, and then makes each of the words an element in the new RDD. If you asked Spark to count the number of elements in the words RDD, it would tell you the number of words in the file.

Next, you'll want to replace each word with a tuple of that word and the number 1. The reason will become clear shortly.

```
pairs = words.map(lambda w: (w, 1))
```

The map() operation replaces each word with a tuple of that word and the number 1. The pairs RDD is a pair RDD where the word is the key, and all of the values are the number 1.

Now, to get a count of the number of instances of each word, you need only group the elements of the RDD by key (word) and add up their values:

```
counts = pairs.reduceByKey(lambda n1, n2: n1 + n2)
```

The reduceByKey() operation keeps adding elements' values together until there are no more to add for each key (word).

Finally, you can store the results in a file and stop the context:

```
counts.saveAsTextFile(sys.argv[2])
sc.stop()
```

The complete file should look like:

```
import re
import sys
from pyspark import SparkConf, SparkContext

conf = SparkConf()
sc = SparkContext(conf=conf)
```

```
lines = sc.textFile(sys.argv[1])
words = lines.flatMap(lambda l: re.split(r'[^\w]+', l))
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(lambda n1, n2: n1 + n2)
counts.saveAsTextFile(sys.argv[2])
sc.stop()
```

Save it in a file called wc.py. To run this application, do the following:

1. Download a copy of the complete works of Shakespeare from the following link: http://www.di.kaist.ac.kr/~swhang/ee412/pg100.txt
2. Open a terminal window on Linux or Mac or a command window on Windows.
3. Change into the directory where you unpacked the Spark binary.
4. Run the following command.
   On Linux or Mac:
   ```
   $ bin/spark-submit YourPathTo/wc.py YourPathTo/pg100.txt YourPathTo/output
   ```
   On Windows:
   ```
   $ bin\spark-submit YourPathTo\wc.py YourPathTo\pg100.txt YourPathTo\output
   ```

After the application completes, you will find the results in the output directory you specified as the last argument to the application.
For example, in the TA's case, the output directory contains the following files:

```
part-00000  part-00001  _SUCCESS
```

Note that each of the RDD partitions is written to one `part-xxxxx` file. You can modify how many partitions you want to use. See more details in here.

5 Write your own Spark Job

Now you will write your first Spark job to accomplish the following task:

- Write a Spark application which outputs the number of words that start with each letter. This means that for every letter we want to count the total number of ***unique*** words that start with that letter. Please ignore the letter case, i.e., consider all words as lower case (e.g., 'AppLE' == 'Apple' =='apple' and 'BIG_2' == 'biG_2'). You should ignore the words that start with non-alphabetic characters (e.g., 3apple).
- Run your program over the same input data as above.
- Use the same parsing code in tutorial.

  ```
  words = lines.flatMap(lambda l: re.split(r'[^\w]+', l))
  ```

  After you split the text by given parsing term (with \w), you do not have to consider any other special cases. Please only consider whether the word starts with alphabet.
- Please **use command-line** arguments to obtain the file path of the dataset. **Do not fix the path in your code.** The input file path should be received by the argument. For example on Linux or Mac, run:

```
bin/spark-submit hw0.py YourPathTo/pg100.txt
```

- Your code should ***print*** the results (total 26 lines) which are sorted in alphabetical order, as following format:

```
<Alphabet in lower case><TAB><Count>
<Alphabet in lower case><TAB><Count>
...
<Alphabet in lower case><TAB><Count>
```

Answers to Frequently Asked Questions

1) You should print all alphabets even if some alphabets have zero count (e.g., `z 0`).
2) 'AppLE', 'Apple', and 'apple' should be counted as 1 unique word for alphabet `a`.
3) "appLe_3" should be count as a word for alphabet `a`, but "3apple" is not a word.
4) Please use the `print` **built-in function in Python** to produce outputs.
5) Also, note that `<TAB>` means `\t` in the Python print function.
6) In this homework, you do not need to save the output files.
7) When you need to test your code: Our recommendation is to create a 'small' test file that you can check the answers on your own.

Please submit the following:

- Your source code in one file (with file name `hw0.py`).