Hw0 - This homework contains questions of mining massive datasets.

Mining Massive Dataset (Wichita State University)

CS246: Mining Massive Datasets **Winter 2018**

Spark Tutorial

Due Thursday January 25, 2018 at 11:59pm Pacific time

General Instructions

The purpose of this tutorial is (1) to get you started with Spark and (2) to get you acquainted with the code and homework submission system. Completing the tutorial is optional but by handing in the results in time students will earn 5 points. This tutorial is to be completed individually.

Here you will learn how to write, compile, debug and execute a simple Spark program. First part of the assignment serves as a tutorial and the second part asks you to write your own Spark program.

Section 1 explains 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. You may setup a full (single-node) spark cluster if you prefer; the results will be the same. You can find instructions online. If you'd like to experiment with a full Spark environment where Spark workloads are executed by YARN against files stored in HDFS, we recommend either the Cloudera Quickstart Virtual Machine: http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms for a pre-installed single-node cluster or Cloudera Manager: http://archive.cloudera.com/cm5/installer/latest/cloudera-manager-installer.bin to install a multi-node cluster on machines you control.

Section 2 explains how to launch the Spark shell for interactively building Spark applications.

Section 3 explains how to use Spark to launch Spark applications written in an IDE or editor.

Section 4 gives an example of writing a simple word count application for Spark.

Section 5 is 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.

This assignment requires you to upload the code and hand-in the output for Section 5.

All students should submit the output via Gradescope and upload the code

Gradescope: create account at https://gradescope.com class code: MKYXN5

Upload the code: http://snap.stanford.edu/submit/

Questions

1 Setting up a stand-alone Spark instance

- Download and install *Spark 2.2.1* on your machine: https://www.apache.org/dyn/closer.lua/spark/spark-2.2.1/spark-2.2.1-bin-hadoop2.7.tgz
- Unpack the compressed TAR ball.

Spark requires JDK 8, which is installed by default on the rice.stanford.edu clusters. **Do not install JDK 9**; Spark is currently incompatible with JDK 9. If you need to download the JDK, please visit Oracle's download site: http://www.oracle.com/technetwork/java/javase/downloads/index.html. If you plan to use Scala, you will also need Scala 2.11, and if you plan to use python, you will need python 2.7 or higher (which is also preinstalled on rice) or 3.4 or higher.

To make Spark, Scala, Maven, etc. work on rice, we had to add the following to our .profile file:

```
JAVA_HOME="/usr/bin/java"
SCALA_HOME="$HOME/scala-2.12.4"
SPARK_HOME="$HOME/spark-2.2.1-bin-hadoop2.7"
SPARK_LOCAL_IP="127.0.0.1"
PATH="$HOME/bin:$HOME/.local/bin:$SCALA_HOME/bin:$SPARK_HOME/bin:
    $HOME/apache-maven-3.5.2/bin:$PATH"
```

These commands are just guidelines, and what you have to add to your file may vary depending on your specific computer setup.

2 Running the Spark shell

Spark gives you two different ways to run your applications. The easiest is using the Spark shell, a REPL that let's you interactively compose your application. The Spark shell supports two languages: Scala/Java and python. In this tutorial we will only discuss using python and Scala. We highly recommend python as both the language itself and the python Spark API are straightforward.

2.1 Spark Shell for Python

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

1. Open a terminal window on Mac or Linux or a command window on Windows.
2. Change into the directory where you unpacked the Spark binary.
3. Run: `bin/pyspark` on Mac or Linux or `bin\pyspark` on Windows.

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 2.2.1
      /_/

Using Python version 2.7.10 (default, Feb  7 2017 00:08:15)
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. (Surprise!) To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

2.2 Spark Shell for Scala

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

1. Open a terminal window on Mac or Linux or a command window on Windows.
2. Change into the directory where you unpacked the Spark binary.
3. Run: `bin/spark-shell` on Mac or Linux or `bin\spark-shell` on Windows.

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 2.2.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

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

```
scala> print("Hello!")
Hello!
```

The Spark shell can also be used to write Spark applications in Scala. (Surprise!) To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/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. 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. Spark accepts applications written in four languages: Scala, Java, python, and R. In this tutorial we will discuss using Java, Scala, and python. If you choose to use R, you will find resources online, but the TA staff may not be able to help you if you run into issues. We highly recommend python as both the language itself and the python Spark API are straightforward.

3.1 Submitting Python Applications

Python is a convenient choice of language as your python application doesn't 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:

1. Open a terminal window on Mac or Linux or a command window on Windows.
2. Change into the directory where you unpacked the Spark binary.
3. Run: `bin/spark-submit path/to/myapp.py path/to/pg100.txt` on Mac or Linux or `bin\spark-submit path\to\myapp.py path\to\pg100.txt` on Windows.

(See section 4 for where to find the `pg100.txt` file.)

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:

```
17/12/18 11:55:40 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1)
in 633 ms on localhost (executor driver) (2/2)
17/12/18 11:55:40 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
17/12/18 11:55:40 INFO DAGScheduler: ResultStage 0 (count at myapp.py:6)
finished in 0.690 s
17/12/18 11:55:40 INFO DAGScheduler: Job 0 finished: count at myapp.py:6,
took 0.841068 s
124787 lines
17/12/18 11:55:40 INFO SparkContext: Invoking stop() from shutdown hook
17/12/18 11:55:40 INFO SparkUI: Stopped Spark web UI at
http://192.168.86.218:4040
17/12/18 11:55:40 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint
stopped!
17/12/18 11:55:41 INFO MemoryStore: MemoryStore cleared
```

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]' path/to/myapp.py path/to/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[*]' path/to/myapp.py path/to/pg100.txt`.

To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

3.2 Submitting Java Applications

Building a Spark application in Java will require you to have Maven installed: https://maven.apache.org/install.html. Follow these steps:

1. Create (or clone) a Maven project. If you're using an IDE, this step can typically be accomplished by choosing to create a new project from the IDE's menus and selecting "Maven project" as the type. If you're not using an IDE, you can follow these steps: https://maven.apache.org/guides/getting-started/. Using an IDE is highly recommended. IntelliJ is generally a good choice, though NetBeans and Eclipse will also work.

2. Modify the `pom.xml` file to include the Spark artifact. In an IDE this can typically be accomplished through a menu-driven process or by editing the `pom.xml` file directly. You can find instructions for IntelliJ and Eclipse here: https://sparktutorials.github.io/2015/04/02/setting-up-a-spark-project-with-maven.html. If you're not using an IDE, you'll need to add the following snipet to the `pom.xml` file using an editor:

```
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.2.1</version>
  </dependency>
</dependencies>
```

Your resulting `pom.xml` file should look something like:

```
<?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.stanford.cs246</groupId>
  <artifactId>MyApp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

```
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
  </dependencies>
</project>
```

Note that the exact file contents may vary depending on how you created your project.

3. Create your application. Assume that you have the following Java file in your Maven project:

```
package edu.stanford.cs246;

import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.SparkConf;

public class MyApp {
  public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf();
    JavaSparkContext sc = new JavaSparkContext(conf);
    System.out.printf("%d lines\n", sc.textFile(args[0]).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.

4. Build your project. Most IDEs provide a way to build a Maven project from the menu or toolbar. You can also build your project from the command line in a terminal or command window by changing to the directory that contains your `pom.xml` file and running: `mvn clean package`. This command will create a JAR file from your project in the `target` directory. Note that the first time you build your Maven project, Maven will try to download all needed dependencies. For Spark that list can be quite long. Be sure you're on a network with reasonable bandwidth and allow yourself enough time.

5. To run your application, do the following:

   (a) Open a terminal window on Mac or Linux or a command window on Windows.

   (b) Change into the directory where you unpacked the Spark binary.

   (c) Run:

```
bin/spark-submit --class edu.stanford.cs246.MyApp
      path/to/MyApp-1.0-SNAPSHOT.jar path/to/pg100.txt
```

on Mac or Linux or

```
bin\spark-submit --class edu.stanford.cs246.MyApp
      path\to\MyApp-1.0-SNAPSHOT.jar path\to\pg100.txt
```

on Windows.

(See section 4 for where to find the `pg100.txt` file.)

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:

```
17/12/18 13:18:16 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 406 ms on localhost (executor driver) (2/2)
17/12/18 13:18:16 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
17/12/18 13:18:16 INFO DAGScheduler: ResultStage 0 (count at MyApp.java:10)
finished in 0.441 s
17/12/18 13:18:16 INFO DAGScheduler: Job 0 finished: count at MyApp.java:10,
took 0.694674 s
124787 lines
17/12/18 13:18:16 INFO SparkContext: Invoking stop() from shutdown hook
17/12/18 13:18:16 INFO SparkUI: Stopped Spark web UI at
http://192.168.86.218:4040
17/12/18 13:18:16 INFO MapOutputTrackerMasterEndpoint:
MapOutputTrackerMasterEndpoint stopped!
17/12/18 13:18:16 INFO MemoryStore: MemoryStore cleared
```

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]' --class edu.stanford.cs246.MyApp path/to/MyApp-1.0-SNAPSHOT.jar path/to/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[*]' --class edu.stanford.cs246.MyApp path/to/MyApp-1.0-SNAPSHOT.jar path/to/pg100.txt`.

To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

3.3 Submitting Scala Applications

Building a Spark application in Scala will require you to have Maven installed: https://maven.apache.org/install.html. Follow these steps:

1. Create (or clone) a Maven project. If you're using an IDE, this step can typically be accomplished by choosing to create a new project from the IDE's menus and selecting "Maven project" as the type. If you're not using an IDE, you can follow these steps: https://maven.apache.org/guides/getting-started/. Using an IDE is highly recommended. IntelliJ is generally a good choice, though Eclipse will also work. If you want to use Netbeans, you will need to install the Scala plugin: https://github.com/dcaoyuan/nbscala.

2. Modify the `pom.xml` file to include the Spark and Scala artifacts and enable the Scala build process. In an IDE this can typically be accomplished through a menu-driven process or by editing the `pom.xml` file directly. You can find instructions for IntelliJ here: http://knowdimension.com/en/data/create-a-spark-application-with-Scala-using-maven- If you're not using an IDE, you'll need to add the following snipet to the `pom.xml` file using an editor:

```
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.2.1</version>
  </dependency>
  <dependency>
    <groupId>org.scala-tools</groupId>
    <artifactId>maven-scala-plugin</artifactId>
    <version>2.11</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
```

```
          </executions>
        </plugin>
      </plugins>
    </build>
```

Your resulting `pom.xml` file should look something like:

```
<?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.stanford.cs246</groupId>
  <artifactId>MyApp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
    <dependency>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <version>2.11</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.scala-tools</groupId>
        <artifactId>maven-scala-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
```

```
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Note that the exact file contents may vary depending on how you created your project.

3. Create your application. Assume that you have the following Scala file in your Maven project:

```
package edu.stanford.cs246

import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.SparkContext._

object MyApp {
  def main(args: Array[String]) {
    val conf = new SparkConf();
    val sc = new SparkContext(conf)
    printf("%d lines\n", sc.textFile(args(0)).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.

4. Build your project. Most IDEs provide a way to build a Maven project from the menu or toolbar. You can also build your project from the command line in a terminal or command window by changing to the directory that contains your `pom.xml` file and running: `mvn clean package`. This command will create a JAR file from your project in the `target` directory. Note that the first time you build your Maven project, Maven will try to download all needed dependencies. For Spark that list can be quite long. Be sure you're on a network with reasonable bandwidth and allow yourself enough time.

5. To run your application, do the following:

   (a) Open a terminal window on Mac or Linux or a command window on Windows.

   (b) Change into the directory where you unpacked the Spark binary.

   (c) Run:

```
bin/spark-submit --class edu.stanford.cs246.MyApp
      path/to/MyApp-1.0-SNAPSHOT.jar path/to/pg100.txt
```

on Mac or Linux or

```
bin\spark-submit --class edu.stanford.cs246.MyApp
      path\to\MyApp-1.0-SNAPSHOT.jar path\to\pg100.txt
```

on Windows.

(See section 4 for where to find the `pg100.txt` file.)

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:

```
17/12/18 13:18:16 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 406 ms on localhost (executor driver) (2/2)
17/12/18 13:18:16 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
17/12/18 13:18:16 INFO DAGScheduler: ResultStage 0 (count at MyApp.java:10)
finished in 0.441 s
17/12/18 13:18:16 INFO DAGScheduler: Job 0 finished: count at MyApp.java:10,
took 0.694674 s
124787 lines
17/12/18 13:18:16 INFO SparkContext: Invoking stop() from shutdown hook
17/12/18 13:18:16 INFO SparkUI: Stopped Spark web UI at
http://192.168.86.218:4040
17/12/18 13:18:16 INFO MapOutputTrackerMasterEndpoint:
MapOutputTrackerMasterEndpoint stopped!
17/12/18 13:18:16 INFO MemoryStore: MemoryStore cleared
```

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]' --class edu.stanford.cs246.MyApp path/to/MyApp-1.0-SNAPSHOT.jar path/to/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[*]' --class edu.stanford.cs246.MyApp path/to/MyApp-1.0-SNAPSHOT.jar path/to/pg100.txt`.

To learn about writing Spark applications, please read through the Spark programming guide: https://spark.apache.org/docs/2.2.0/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, Java, and Scala. 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.

4.1 Word Count in python

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 course website: http://web.stanford.edu/class/cs246/homeworks/hw0/pg100.txt.
2. Open a terminal window on Mac or Linux or a command window on Windows.
3. Change into the directory where you unpacked the Spark binary.
4. Run:

   ```
   bin/spark-submit path/to/wc.py path/to/pg100.txt path/to/output
   ```

on Mac or Linux or

```
bin\spark-submit path\to\wc.py path\to\pg100.txt
  path\to\output
```

on Windows.

After the application completes, you will find the results in the output directory you specified as the second argument to the application.

4.2 Word Count in Java

Before you start, create a new Maven project for your application. The bare minimum requirements are a `pom.xml` file and a source code directory. The source code directory should be `path/to/project/src/main/java`, and the `pom.xml` file should contain something like:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.stanford.cs246</groupId>
  <artifactId>WordCount</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
  </dependencies>
</project>
```

Once you have your project created, you can begin writing the application. The first step of every Java Spark application is to create a Spark context:

```
package edu.stanford.cs246;

import java.util.Arrays;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import scala.Tuple2;

public class WordCount {
  public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf();
    JavaSparkContext sc = new JavaSparkContext(conf);

    ...
  }
}
```

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

```
    JavaRDD<String> lines = sc.textFile(args[0]);
```

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:

```
    JavaRDD<String> words =
        lines.flatMap(l -> Arrays.asList(l.split("[^\\w]+")).iterator());
```

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. Note that the lambda argument to the method must return an iterator, not a list or array.

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

```
    JavaPairRDD<String, Integer> pairs =
        words.mapToPair(w -> new Tuple2<>(w, 1));
```

The `mapToPair()` 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. Note that the type of the RDD is now `JavaPairRDD`. Also note that the use of the Scala `Tuple2` class is the normal and intended way to perform this operation.

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:

```
JavaPairRDD<String, Integer> counts =
    pairs.reduceByKey((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(args[1]);
sc.stop();
```

The complete file should look like:

```
package edu.stanford.cs246;

import java.util.Arrays;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import scala.Tuple2;

public class WordCount {
  public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf();
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<String> lines = sc.textFile(args[0]);
    JavaRDD<String> words =
        lines.flatMap(l -> Arrays.asList(l.split("[^\\w]+")).iterator());
    JavaPairRDD<String, Integer> pairs =
        words.mapToPair(w -> new Tuple2<>(w, 1));
    JavaPairRDD<String, Integer> counts =
        pairs.reduceByKey((n1, n2) -> n1 + n2);

    counts.saveAsTextFile(args[1]);
    sc.stop();
  }
}
```

Save it in a file called `edu/stanford/cs246/WordCount.java` in your project's source code directory. To run this application, do the following:

1. Download a copy of the complete works of Shakespeare from the course website: http://web.stanford.edu/class/cs246/homeworks/hw0/pg100.txt.
2. Open a terminal window on Mac or Linux or a command window on Windows.
3. Change into the project directory.
4. Build the project by running `mvn clean package`.
5. Change into the directory where you unpacked the Spark binary.
6. Run:

   ```
   bin/spark-submit --class edu.stanford.cs246.WordCount
     path/to/WordCount-1.0-SNAPSHOT.jar path/to/pg100.txt path/to/output
   ```

   on Mac or Linux or

   ```
   bin\spark-submit --class edu.stanford.cs246.WordCount
     path\to\WordCount-1.0-SNAPSHOT.jar path\to\pg100.txt path\to\output
   ```

   on Windows.

After the application completes, you will find the results in the output directory you specified as the second argument to the application.

4.3 Word Count in Scala

Before you start, create a new Maven project for your application. The bare minimum requirements are a `pom.xml` file and a source code directory. The source code directory should be `path/to/project/src/main/scala`, and the `pom.xml` file should contain something like:

```
<?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.stanford.cs246</groupId>
```

```
  <artifactId>WordCount</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.1</version>
    </dependency>
    <dependency>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <version>2.11</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.scala-tools</groupId>
        <artifactId>maven-scala-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Once you have your project created, you can begin writing the application. The first step of every Scala Spark application is to create a Spark context:

```
package edu.stanford.cs246

import org.apache.spark.{SparkConf,SparkContext}
```

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

object WordCount {
  def main(args: Array[String]) {
    val conf = new SparkConf();
    val sc = new SparkContext(conf)

    ...
  }
}
```

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

```
    val lines = sc.textFile(args(0))
```

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:

```
    val words = lines.flatMap(l => l.split("[^\\w]+"))
```

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.

```
    val pairs = words.map(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:

```
    val counts = pairs.reduceByKey((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(args(1))
    sc.stop
```

The complete file should look like:

```
package edu.stanford.cs246

import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.SparkContext._

object WordCount {
  def main(args: Array[String]) {
    val conf = new SparkConf();
    val sc = new SparkContext(conf)

    val lines = sc.textFile(args(0))
    val words = lines.flatMap(l => l.split("[^\\w]+"))
    val pairs = words.map(w => (w, 1))
    val counts = pairs.reduceByKey((n1, n2) => n1 + n2)

    counts.saveAsTextFile(args(1))
    sc.stop
  }
}
```

Save it in a file called `edu/stanford/cs246/WordCount.scala` in your project's source code directory. To run this application, do the following:

1. Download a copy of the complete works of Shakespeare from the course website: http://web.stanford.edu/class/cs246/homeworks/hw0/pg100.txt.
2. Open a terminal window on Mac or Linux or a command window on Windows.
3. Change into the project directory.
4. Build the project by running `mvn clean package`.
5. Change into the directory where you unpacked the Spark binary.
6. Run:

```
bin/spark-submit --class edu.stanford.cs246.WordCount
  path/to/WordCount-1.0-SNAPSHOT.jar path/to/pg100.txt path/to/output
```

on Mac or Linux or

```
bin\spark-submit --class edu.stanford.cs246.WordCount
  path\to\WordCount-1.0-SNAPSHOT.jar path\to\pg100.txt path\to\output
```

on Windows.

After the application completes, you will find the results in the output directory you specified as the second argument to the application.

5 Task: 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 (non-unique) words that start with that letter. In your implementation ignore the letter case, *i.e.*, consider all words as lower case. You can ignore all non-alphabetic characters.
- Run your program over the same input data as above.

What to hand-in: Submit the printout of the output file to Gradescope at https://gradescope.com, and upload the source code at http://snap.stanford.edu/submit.