Big Data Analytics (SOEN 498/691)
Laboratory sessions

Tristan Glatard, Valerie Hayot-Sasson
Department of Computer Science and Software Engineering
Concordia University, Montreal
tristan.glatard@concordia.ca, valeriehayot@gmail.com

January 20, 2017

Contents

Part I
Prerequisites

1 Notations

This is a command you are supposed to type in a Linux terminal (don't type the $ sign):

```
$ echo Welcome
```

This command is split in two lines:

```
$ echo Welcome to this tutorial,\
  I hope you will enjoy it!
```

This is a piece of Java code:

(Link to file)

```
int a=7;
```

This is a piece of bash script:

(Link to file)

```
#!/bin/bash
for i in 1 2 3
do
   echo $i
done
```

And this is a piece of XML file:

```
<one>
  <two>Hello</two>
</one>
```

2 Java

Java is the primary language used in Hadoop. Make sure that Java version 7 or higher is installed on your system. If it is not, follow the instructions from the Oracle website and check that the Java Virtual Machine (JVM) works correctly:

```
$ java -version
```

Make sure that you are able to create and run a simple Java program such as:

(Link to file)

```
public class Test{
  public static void main(String[] args){
    System.out.println(''Hello, World!'');
  }
}
```

This program is compiled as follows:

```
$ javac Test.java
```

The compilation produces a `Test.class` file that contains the compiled class. The program can be executed as follows:

```
$ java Test
```

`Test.class` must be located in the directory where `java` is executed, or in a directory listed in the `CLASSPATH` environment variable.

This tutorial can be done with Java command-line tools (`javac` and `java`) a text editor such as `vim` or `emacs`. For larger applications, for instance your course project, it is recommended to use an Integrated Development Environment (IDE) tailored for Java, e.g., Eclipse or Netbeans.

3 Linux

This tutorial is based on Linux and it uses the bash shell. You don't have to install bash as it is already present on your workstation.

3.1 Useful commands

- `cat`: prints the content of a file on the standard output.
- `chmod`: changes the access permissions of a file or directory.
- `echo`: prints a message on the standard output.
- `export`: exports environment variables to the environment of subsequently executed commands.
- `ls`: lists directory content.
- `man`: gets help on any command.
- `mkdir`: creates a directory.
- `seq 1 1000`: prints the integers from 1 to 1000.
- `tar`: handles `tar`, `tgz` and many other types of archives.
- `wget`: downloads the content of a URL.

3.2 Standard streams

A Linux process is connected to three particular files called *streams*:

- The standard *output* is used by the process to write normal information.
- The standard *error* is used by the process to write error information.
- The standard *input* is used to send information to the process.

We will use streams extensively in the tutorial.

3.3 Useful operators

- `>` (redirection operator): redirects the standard output of its left argument to a file. For instance, `echo Hello > a.txt` writes "Hello" in file `a.txt` (if `a.txt` already exists, it is overwritten).
- `>>`: same as `>` but the standard output of the left operand is *appended* to the file in the right operand.
- `|` (pipe operator): redirects the standard output of its left argument to the standard input of its right argument. For instance, `cat a.txt | sort` prints the sorted content of a file.

Here is also a list of useful environment variables:

- `PATH`: tells the system which directories to search for executables.
- `CLASSPATH`: tells the JVM where classes must be searched.

Part II
Getting started with Hadoop

This part of the tutorial will guide you through practical steps to install Hadoop and write simple MapReduce programs. For a theoretical presentation of Hadoop and MapReduce, refer to Chapter 2 of [1] (available here) or to Chapter 2 of [2] (available here) . The content presented hereafter is adapted from the following sources:

- Apache Hadoop documentation
- MapReduce tutorial
- Tom White, "Hadoop: The Definitive Guide". Chapter 2 (MapReduce).

For now we will run Hadoop in *standalone mode*, i.e., in a single JVM. This mode is useful for debugging programs and testing things out.

4 Installation

Download Hadoop release 2.7.3 from one of the Apache Download Mirrors and unpack the distribution:

```
$ wget http://apache.mirror.rafal.ca/hadoop/common/hadoop-2.7.3/hadoop-2.7.3.tar.gz
$ tar xf hadoop-2.7.3.tar.gz
$ export HADOOP=$PWD/hadoop-2.7.3
```

Edit `$HADOOP/etc/hadoop/hadoop-env.sh` and define the path to your Java installation:

```
export JAVA_HOME=/path/to/your/java/installation/
```

Add the Hadoop executables to your PATH:

```
$ export PATH=$PATH:${HADOOP}/bin:${HADOOP}/sbin
```

Make sure Hadoop is installed properly:

```
$ hadoop version
```

Run a first example:

```
$ mkdir input
# creates a test file containing integers from 1 to 1000
$ for i in $(seq 1 1000); do echo $i >> input/file.txt ; done
$ hadoop jar ${HADOOP}/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar grep \
                                                                    input output .*1.*
$ cat output/*
```

This program prints the lines of the input file that contain the string "1".

5 First MapReduce program

As a first MapReduce program, write the classical WordCount example:

(Link to file)

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
```

```
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  // The class that will be used for the map tasks. Has to extend ''Mapper''.
  // In general, overrides the ``map'' method.
  // < > denotes the use of a Generic Type. Text and IntWritable are Hadoop
  // classes used for keys and values. In Hadoop, any key or value has to
  // implement specific interfaces (Java's String or Integer cannot
  // be used directly).
  public static class TokenizerMapper
  extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    // Called once for each key/value pair in the input split.
    // Context is a class defined in Mapper, that gives access to
    // input and output key/value pairs.
    public void map(Object key, Text value, Context context
    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        // Emits key/value pair in the form <word, 1>
        context.write(word, one);
      }
    }
  }

  // The class that will be used for the reducers and combiners.
  public static class IntSumReducer
  extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
    Context context
    ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  // Main method
  public static void main(String[] args) throws Exception {
    // Configuration provides access to configuration parameters
    Configuration conf = new Configuration();
    // Create a new Hadoop job named ``word count''
    Job job = Job.getInstance(conf, "word␣count");
    // Sets the Jar by finding where the WordCount class came from
    job.setJarByClass(WordCount.class);
    // Sets mapper class for the job.
    job.setMapperClass(TokenizerMapper.class);
    // Sets combiner class for the job.
    job.setCombinerClass(IntSumReducer.class);
    // Sets reducer class for the job.
```

```
    job.setReducerClass(IntSumReducer.class);
    // Set the key class for the job output data.
    job.setOutputKeyClass(Text.class);
    // Set the value class for job outputs.
    job.setOutputValueClass(IntWritable.class);
    // Add a Path to the list of inputs for the map-reduce job.
    FileInputFormat.addInputPath(job, new Path(args[0]));
    // Set the Path of the output directory for the map-reduce job.
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

The code above is commented so that you should be able to understand what it does. Refer to the Apache Hadoop API reference if anything is unclear.

We will now compile and run this program. Add Hadoop's jars to the CLASSPATH:

```
export CLASSPATH=$CLASSPATH:${HADOOP}/share/hadoop/common/hadoop-common-2.7.3.jar:\
          ${HADOOP}/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.7.3.jar:.
```

Compile the program:

```
$ javac WordCount.java
```

Create a jar containing all the classes in WordCount:

```
$ jar cvf wordcount.jar WordCount*.class
```

Create a test file and run the program:

```
$ echo one two three two three three > test.txt
$ hadoop jar wordcount.jar WordCount test.txt output
```

6 Hadoop Streaming

While the Hadoop framework is developed in Java, it can also execute programs written in any language using the Hadoop Streaming tool. In this section, we are going to implement the WordCount example in bash. You may also implement it in any other language (Python, Ruby, C, etc) if you wish.

Hadoop Streaming works as follows:

1. The key/value pairs are passed to the mapper executable through its standard input.
2. The mapper executable writes output key/value pairs to its standard output.
3. The sorted key/value pairs are passed to the reduce executable through its standard input.

Hence, mappers and reducers can be tested as follows for any MapReduce program executed with Hadoop Streaming:

```
$ cat input.txt | mapper.sh | sort | reducer.sh
```

(Replace `input.txt` with the name of your input file, `mapper.sh` with the bash script containing the mapper code and `reducer.sh` with the bash script containing the reduce code). The mapper code is straightforward:

(Link to file)

```
#!/bin/bash

# Read all the lines passed on stdin
while read line
do
    # Split the words in the line
    for word in $line
```

```
    do
        # Emit a <word,1> key/value pair for every word
        echo $word 1
    done
done
```

The reducer code is a bit more complex:

(Link to file)

```
#!/bin/bash

count=0
while read line
do
    # Declare an array containing all the elements on the line
    # About bash arrays: http://www.tldp.org/LDP/abs/html/arrays.html
    tokens=( $line )
    # The word (key) is the first element in the array
    new_word=${tokens[0]}
    # The number of occurrence of this word (value) is the second element in the array
    increment=${tokens[1]}
    # Remember that the key/value pairs produced by mappers
    # are sorted before being passed to the reducer.
    # Here we detect if the word in the key/value pair is the same
    # as at the previous iteration or if it has changed.
    # ${word+x} is a bash parameter expansion that is substituted to word if
    # and only if word is set (see https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.htm
    if [[ ${new_word} = ${word} ]] || [ -z ${word+x} ]
    then
        # Word hasn't changed or it was not defined at the previous iteration:
        # increment the counter by the value in key/value pair.
        count=$((count+${increment}))
    else
        # Word has changed: emit the key/value pair and set the counter to the
        # value in key/value pair.
        echo $word $count
        count=${increment}
    fi
    # Save the word for next iteration
    word=$new_word
done
# Emit last word
echo $word $count
```

Run the program with Hadoop Streaming as follows:

```
hadoop jar ${HADOOP}/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar\
-input test.txt\
-output output\
-mapper ./mapper.sh\
-reducer ./reducer.sh\
-file wordcount-reducer.sh -file wordcount-mapper.sh
```

Create a larger text file and run the program again. The number of mappers and reducers can be changed with the following options: `mapreduce.job.reduces` and `mapreduce.job.maps`. For instance, to use two mappers and two reducers:

```
hadoop jar ${HADOOP}/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
-D mapreduce.job.reduces=2 \
-D mapreduce.job.map=2 \
-input test.txt \
-output output \
-mapper ./mapper.sh \
-reducer ./reducer.sh \
-file reducer.sh -file mapper.sh
```

You can also add a combiner with option `-combiner`.

The framework still uses only a single mapper and a single reducer because it is configured in standalone mode, i.e., as a single Java process. In the next part of the tutorial, we will configure the pseudo-distributed mode where Hadoop runs in several processes.

Part III
HDFS and YARN

In standalone mode, Hadoop was using the local file system to store files and all the jobs were executed in the same JVM. In pseudo-distributed mode, it uses the Hadoop Distributed File System (HDFS) and YARN (Yet Another Resource Negotiator).

The content presented hereafter is adapted from the following sources:

- Apache Hadoop documentation
- MapReduce tutorial
- Tom White, "Hadoop: The Definitive Guide". Chapter 2 (MapReduce), Chapter 3 (HDFS).

7 ssh setup

Make sure that you can connect to your own workstation using ssh *with no password*:

```
$ ssh localhost
```

If the ssh command prompts for a password, create a password-less ssh key and authorize it in your account:

```
$ mkdir -p $HOME/.ssh
$ chmod 700 $HOME/.ssh
$ ssh-keygen
# press 'Enter' at every question
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
$ chmod 600 $HOME/authorized_keys
```

8 HDFS

8.1 Configuration

Configure the HDFS by adding the following to `$HADOOP/etc/hadoop/core-site.xml`:

```
<configuration>
   <property>
      <name>fs.defaultFS</name>
      <value>hdfs://localhost:9000</value>
   </property>
   <property>
      <name>hadoop.tmp.dir</name>
      <value>/some/path/on/your/workstation</value>
   <description>A base for other temporary directories.</description>
  </property>
</configuration>
```

and to `$HADOOP/etc/hadoop/hdfs-site.xml`:

```
<configuration>
   <property>
      <name>dfs.replication</name>
      <value>1</value>
   </property>
</configuration>
```

Format the file system:

```
$ hdfs namenode -format
```

And start the HDFS daemons:

```
$ start-dfs.sh
```

8.2 Command-line usage

We will now review the main HDFS commands. Directory creation:

```
  $ hdfs dfs -mkdir /user
  $ hdfs dfs -mkdir /user/<username>
```

Directory listing:

```
  $ hdfs dfs -ls /user
```

File copy to HDFS:

```
  $ echo This is a test > test.txt
  $ hdfs dfs -put test.txt /user/<username>/test.txt
```

File copy from HDFS:

```
  $ hdfs dfs -get /user/<username>/test.txt
```

File content display:

```
  $ hdfs dfs -cat /user/<username>/test.txt
```

File removal:

```
  $ hdfs dfs -rm /user/<username>/test.txt
```

8.3 Web interface

HDFS can be browsed through a web interface which is by default available at http://localhost:50070.

8.4 Java API

The `FileSystem` API is the safest way to access files on HDFS from a Java program. A file in a Hadoop file system is represented by a Hadoop `Path` object. You can think of a Path as a Hadoop file system URI, such as `hdfs://localhost/user/tom/quangle.txt`.

The following program reads a file on HDFS:

(Link to file)

```
import java.io.IOException;
import java.io.InputStream;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;


public class HdfsCat {

   public static void main(String[] args) throws IOException {
```

```
        // URI to 'cat' will be passed as first argument
        String hdfsUri = args[0];
        // Create the FileSystem object
        FileSystem fs = FileSystem.get(URI.create(hdfsUri), new Configuration());
        // Create HDFS Path from URI
        Path path = new Path(hdfsUri);
        InputStream in = null;
        try{
            // Open an InputStream from the Path
            in = fs.open(path);
            // Copy bytes from InputStream to stdout
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally{
            // Close the InputStream
            IOUtils.closeStream(in);
        }
    }
}
```

After having compiled this file, create a jar and run it with Hadoop:

```
$ hadoop jar <path to jar> HdfsCat <HDFS URI of file to cat>
```

A similar program can be written to write files to HDFS:

(Link to file)

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;


public class HdfsPut {
  public static void main(String[] args) throws FileNotFoundException, IOException{
      String localFilePath = args[0];
      String dest = args[1];
      InputStream in = new BufferedInputStream(new FileInputStream(localFilePath));
      FileSystem fs = FileSystem.get(URI.create(dest),new Configuration());
      OutputStream out = fs.create(new Path(dest));
      IOUtils.copyBytes(in, out, 4096, true);
  }
}
```

Compile this class and make sure you can use it to write files to HDFS.

9 YARN

YARN (Yet Another Resource Negotiator) is the resource management system used in pseudo-distributed mode.

9.1 Configuration

Edit `$HADOOP/etc/hadoop/mapred-site.xml` to add the following configuration properties:

```
<configuration>
  <property>
```

```
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Edit `$HADOOP/etc/hadoop/yarn-site.xml` to add the following configuration properties:

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
    <description>Whether virtual memory limits will be enforced for containers.</description>
  </property>
</configuration>
```

Start the YARN daemon:

```
$ start-yarn.sh
```

The MapReduce job history daemon is another useful service to start:

```
$ mr-jobhistory-daemon.sh start historyserver
```

Re-run the Hadoop Streaming example of Section 6 using multiple mappers and reducers.

9.2 Web interfaces

Browse the YARN monitoring interface (`http://localhost:8088`) and the job history server (`http://localhost:19888`).

9.3 Improved WordCount

Here is a more complete WordCount example that adds the following features to the initial example presented in Section 5:

- It uses the distributed cache to distribute a "skip" file to all the mappers. The skip file contains patterns to be removed from words.
- The Mapper uses a `setup` method.
- It uses Counters to track statistics related to the execution.
- It uses Configuration to read configuration parameters.

The code below has been commented so that you should be able to understand what it does. Refer to the Apache Hadoop API reference if anything is unclear.

(Link to file)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
```

```java
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.StringUtils;

public class WordCount2 {

  // Class that will be used for the map tasks.
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    // Definition for a group of 1 counter that
    // will be used to count all the input words
    static enum CountersEnum { INPUT_WORDS }

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    // Specifies whether matching is case sensitive.
    // Value will be set from configuration.
    private boolean caseSensitive;
    // Specifies the patterns that should be ignored in the words.
    // Patterns are contained in files stored in the distributed cache.
    // A configuration parameter specifies whether patter skipping is
    // activated.
    private Set<String> patternsToSkip = new HashSet<String>();

    private Configuration conf;
    private BufferedReader fis;

    // Called once at the beginning of the task, i.e.,
    // once for all key-value pairs. Typically used to set
    // configuration parameters and perform other common tasks.
    @Override
    public void setup(Context context) throws IOException,
       InterruptedException {
      // Read configuration parameters
      conf = context.getConfiguration();
      // First config parameter specifies whether matching will be case sensitive
      caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
      // Second config parameter specifies the file that contains the patterns to skip
      // It is set by the main method when command-line option ``-skip'' is used.
      if (conf.getBoolean("wordcount.skip.patterns", true)) {
        // This is how file names is retrieved from the distributed cache
        // In this case all the files in the cache are assumed to be skip files
        URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
        if(patternsURIs != null)
        for (URI patternsURI : patternsURIs) {
          Path patternsPath = new Path(patternsURI.getPath());
          String patternsFileName = patternsPath.getName().toString();
          parseSkipFile(patternsFileName);
        }
```

```
    }
  }

  // This method is called from setup. It parses a skip file
  // and puts the patterns to skip in the patternsToSkip class attribute.
  private void parseSkipFile(String fileName) {
    try {
      // A file in the distributed cache is read as any other file.
      fis = new BufferedReader(new FileReader(fileName));
      String pattern = null;
      while ((pattern = fis.readLine()) != null) {
        patternsToSkip.add(pattern);
      }
    } catch (IOException ioe) {
      System.err.println("Caught␣exception␣while␣parsing␣the␣cached␣file␣'"
          + StringUtils.stringifyException(ioe));
    }
  }

  // Called once for each key/value pair in the input split.
  @Override
  public void map(Object key, Text value, Context context
       ) throws IOException, InterruptedException {
    // Igonores case in the line if class attributed was set in setup.
    String line = (caseSensitive) ?
       value.toString() : value.toString().toLowerCase();
    // Skips all the patterns on the line
    for (String pattern : patternsToSkip) {
      line = line.replaceAll(pattern, "");
    }
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
      // Gets a counter called ``INPUT_WORDS'' with group ``CountersEnum''
      Counter counter = context.getCounter(CountersEnum.class.getName(),
          CountersEnum.INPUT_WORDS.toString());
          // Increment the counter. Counters are shared among all map tasks.
          // When the application completes, the counter will contain the total
          // number of words that have been processed.
      counter.increment(1);
    }
  }
}

// Reducer. Identical to the initial WordCount example.
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

```
  // Main method.
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    // GenericOptionsParser helps parsing attributes passed to the application.
    GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
    String[] remainingArgs = optionParser.getRemainingArgs();
    if (!(remainingArgs.length != 2 || remainingArgs.length != 4)) {
      System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
      System.exit(2);
    }
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount2.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    // Adds skip file to cache if available
    List<String> otherArgs = new ArrayList<String>();
    for (int i=0; i < remainingArgs.length; ++i) {
      if ("-skip".equals(remainingArgs[i])) {
        job.addCacheFile(new Path(remainingArgs[++i]).toUri());
        job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
      } else {
        otherArgs.add(remainingArgs[i]);
      }
    }
    FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Compile this example and check that it works as expected:

- Create a text file with uppercase and lowercase characters and special characters such as ',', '.' and '!'.
- Create a skip file containing the special characters.
- Make sure that the program counts words as if there was no special character.
- Make sure that the INPUT_WORDS counter is properly reported in the standard output at the end of the execution.
- Use option -Dwordcount.case.sensitive=false to perform case-insensitive matching.

Troubleshooting:

- Jar `$HADOOP/share/hadoop/hdfs/lib/commons-cli-1.2.jar` must be added to the `CLASSPATH` variable to compile the code.
- Special characters such as '.' must be escaped as in '\.' in the skip file.

References

[1] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[2] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.