CS5425 Assignment 1 Task (How to do?)

Name: Ong Jian Ying Gary

Matriculation Number: A0155664X

Date: 17/09/2022

Task description:

- Task 1: Given two textual files, count the number of words that are common.
- Caveat, we are also given a third file, titled stopwords.txt that we have to input as an argument so that we are able to avoid these stopwords that are defined within the list to identify similarity between both documents of inputs.
- We are provided two input files:
  - Task1-input1.txt
  - Task1-input2.txt
- Output
  - Output directory is specified in <output>
  - Top-20 output of the result using the data files listed above (you only need to extract these 20 output from the sorted output), with each line:
  - **Freq**\t**word**\n, where \t is tab and \n creates a new line. **(IMPORTANT!!!)**
- Where we are supposed to identify top 20 common words between these two files ASIDE from the stop words that were provided.
- TopkCommonWords <input1> <input2> <input3> <output>
- An example command can be:
  - hadoop jar cm.jar TopkCommonWords commonwords/input/task1-input1.txt commonwords/input/task1-input2.txt commonwords/input/stopwords.txt commonwords/cm_output/
  - Note that the **./compile_run script already contains this command**. So you can simply test your code using ./compile_run (on the SoC cluster)
- If your answer is correct, the output of your file should be the same as that given in the **answer.txt** file.
- **Problem definition**
  - Given TWO textual files, for each common word between the two files, **find the *smaller* number of times that it appears between the two files.** **(Possible approach: For each matching key, take the minimum value of the**

two files). Output the top 20 common words with highest such frequency (For words with the same frequency, there's no special requirement for the output order).

- Example: if the word "John" appears 5 times in the 1st file and 3 times in the 2nd file, the smaller number of times is 3

- **Requirements**
- Split the input text with "(space)\t\n\r\f". (Possible approach: https://www.geeksforgeeks.org/split-string-java-examples/). Any other tokens like ",.:`" will be regarded as a part of the words
- Remove stop-words as given in Stopwords.txt, such as "a", "the", "that", "of", … (case sensitive)
- Sort the common words in descending order of the smaller number of occurrences in the two files.
- In general, words with different case or different non-whitespace punctuation are considered different words.

Methodology / approach:

- Utilize the stopwords.txt file as an additional argument and figure out how to extract all the words within the list that we need to exclude from our map reduce search for each document.
- Run the mapreduce word count program on both documents and get their top words count and return them as a list for each document
- Use the variables for stored result for both and **exclude the stop words that were captured from the stopwords.txt file**
- With the remaining words, we would then want to use both list of words from the wordcount output and then return the top 20 common words as the output answer to be written as a file titled ''
- For words that exist within the stopwords.txt string list, use the following to exclude words in the stopwords.txt file:
    - If (!mywordlist.contains(word.toString())) {

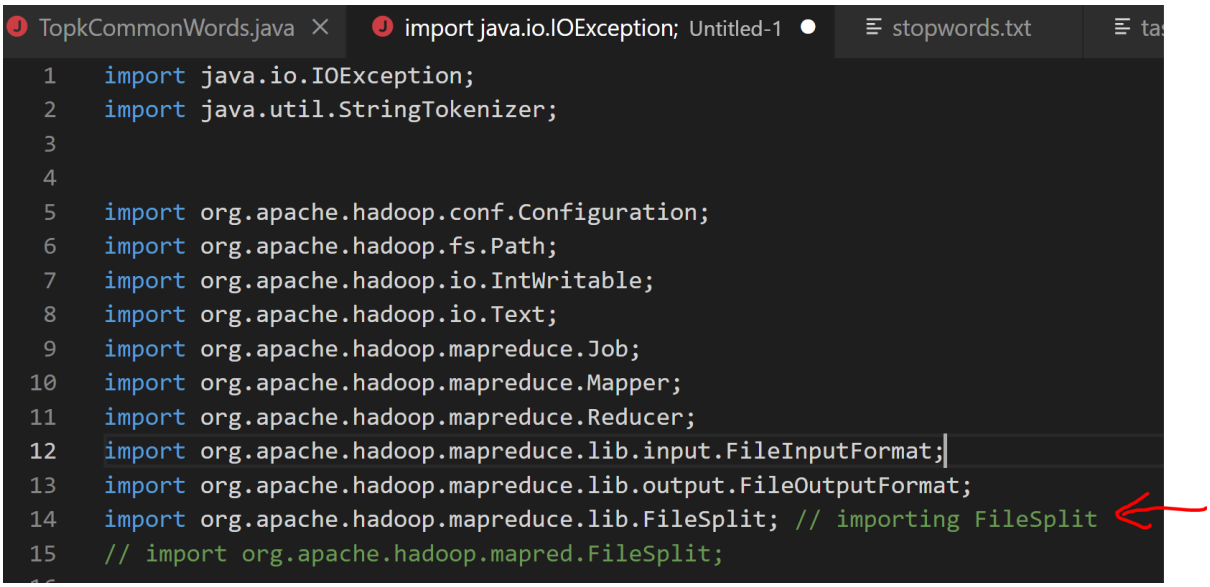
      Context.write(word,one)

      }
    - What the above is saying: if my stopwordlist contains the current word I am looking at now for each of the input document, I will not want to write the output and get the sums.
- We also need to change our configuration to accept 4 arguments from the user.
-

```java
// Driver program
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs(); // get all args
  // edit this part to expect 3 arguments from user, last one being the s3 bucket name
  if (otherArgs.length != 3) {
    System.err.println("Usage: WordCount <in> <out> <s3 bucket name>");
    System.exit(2);
  }
```

Steps taken:

1. The first task that I did was to include and import FileSplit library into my code by putting this at the top:



This library is important for us to **get the current input file name.**
https://stackoverflow.com/questions/19012482/how-to-get-the-input-file-name-in-the-mapper-in-a-hadoop-program

Import all the libraries required for all the tasks here:

```java
import java.io.IOException;
import java.util.StringTokenizer;
import java.util.Dictionary;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.Comparator;
```

```
import java.util.Iterator;
import java.util.Collection;
import java.util.Collections;
import java.util.stream.Collectors;

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.input.MultipleInputs; // to allow
for multiple inputs.
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit; // importing FileSplit
import org.apache.hadoop.util.GenericOptionsParser; // use
GenericOptionsParser to allow us to identify the arguments given in the file
to consider stopwords.txt file as well.

// Important for reading file from HDFS.
import java.io.BufferedReader;
import java.io.FileReader;
```

2. Another edit done was changing the output index in the driver program to the following:

// set the HDFS path of the output data, in this case, **we are putting the output at the fourth argument.**

Hadoop also allows for multiple input paths, and we have to change the output file path as well at the fourth argument (index 4)

```
      // HADOOP ALLOWS FOR MULTIPLE INPUT PATHS, so we can just specify the
changes here.
      FileInputFormat.addInputPath(job, new Path(args[0])); // set the HDFS
path of the input data for task1-input1.txt
      FileInputFormat.addInputPath(job, new Path(args[1])); // set the HDFS
path of the input data for task1-input2.txt
      FileInputFormat.addInputPath(job, new Path(args[2])); // set the HDFS
path of the input data for stopwords.txt

      // set the HDFS path of the output data, in this case, we are putting
the output at the fourth argument.
      FileOutputFormat.setOutputPath(job, new Path(args[3]));
```

3. Always make sure the mapper / reducer return types are correctly specified, i.e. "MyMapper extends Mapper<input key, input value, output key, output value>", and similarly for the reducer.

Since we are using mappers that emit different types than a reducer, we have to set the types emitted by the mapper.

In this case, our mapper emits <Text – word in each input file, Text – file num that the word belonged to>

Our reducer emits <IntWritable – minimum counts of each common word, Text – common word in both documents>

We also need to change the name of our setJarByClass, setMapperClass, setReducerClass to what I have named it in the code.

Hence, we need to do the following:

```
      job.setJarByClass(TopkCommonWords.class);
      // changing our code to CommonWordsMapper.class instead of
TokenizerMapper.class
      job.setMapperClass(CommonWordsMapper.class);
      // changing our code to MinCountReducer.class instead of
IntSumReducer.class
      job.setReducerClass(MinCountReducer.class);
      // Since our mapper emits different types than the reducer, we can set
the types emitted by the mapper, as mentioned by prof.
      // setting mapper output key type
      job.setMapOutputKeyClass(Text.class);
      // setting mapper output value type
      job.setMapOutputValueClass(Text.class);
      // setting reducer output key type
      job.setOutputKeyClass(IntWritable.class);
      // setting reducer output value type
      job.setOutputValueClass(Text.class);
```

How can I get the name of the input file within a mapper? I have multiple input files stored in the input directory, each mapper may read a different file, and I need to know which file the mapper has read.

4. Within the map function, our aim here is to emit tuples <word, filenum>

   If word comes from input1, we will give a filenum of 1
   If word comes from input2, we will give a filenum of 2

   This will later allow us to make use of 2 dictionaries in the reduce function as inputs and count the total number of occurrences of each word in each input file.

   Hence, we will make use of the FileSplit library and its method .getInputSplit(). We will define a string for the current file name being looked at by the map function.

   For all the tokens of words within our map function, we will iterate through each word and emit each word with the file num corresponding to them.

   In the shuffle phase, all words with the same key will go to the same reducer.

5. However, there is incorporation of the stopwords.txt file from input3 as well. We need to ensure that if the word currently being read from the input files (input1 and input2), if the word exists inside stopwords.txt, we will not emit those tuples in our map function.

   We first need to make sure to ensure that our string of words coming from stopwords.txt is stored in the Configuration conf object.

   In our main() function, we will first use the following code:

```
      // incorporate the stopwords.txt file inside our code now within the
main method.
      // 1. read the stopwords into a string in the main method
      String stopwords_path = new String(args[2]); // getting the stopwords
path from args[2] as a string.
      BufferedReader br = new BufferedReader(new FileReader(stopwords_path));

      // Declaring a StringBuilder and String variable to be used in while
loop, where lines are read and string is built and words are delimited by
commas.
      // String builder used to build up our stopwords_string, since
conf.set() can be used on Strings. Then we can apply conf.get() in map
function to incorporate stopwords.
      StringBuilder sb = new StringBuilder();
      String st;
      // Condition holds true as long as there is a character in string.
      while ((st= br.readLine()) != null) {
        // add read word line by line into our stopwords ArrayList
```

```
        sb.append(st); // use StringBuilder to append to list.
        sb.append(","); // append a comma as a delimiter
      }

      String stopwords_string = sb.toString(); // getting the stopwords string
from StringBuilder
      // With our stopwords list, we can use conf.set() to set the variable in
configurations, with title 'stopwords'.
      conf.set("stopwords", stopwords_string);
      // 2. Thereafter, pass the string to the mappers using conf.set() and
conf.get() in our Configurations object with setup() method.
```

We first assign stopwords path as our third argument args[2].
Thereafter, we make use of a BufferedReader and FileReader to read the data from our stopwords path.

We utilize a stringbuilder to ensure that while each line is being read in the bufferedreader, that we are appending each unique stopwords to a final stopwords string, with each word being delimited by a comma ‘,’.

Hence, we use the following code here:

```
      // incorporate the stopwords.txt file inside our code now within the
main method.
      // 1. read the stopwords into a string in the main method
      String stopwords_path = new String(args[2]); // getting the stopwords
path from args[2] as a string.
      BufferedReader br = new BufferedReader(new FileReader(stopwords_path));

      // Declaring a StringBuilder and String variable to be used in while
loop, where lines are read and string is built and words are delimited by
commas.
      // String builder used to build up our stopwords_string, since
conf.set() can be used on Strings. Then we can apply conf.get() in map
function to incorporate stopwords.
      StringBuilder sb = new StringBuilder();
      String st;
      // Condition holds true as long as there is a character in string.
      while ((st= br.readLine()) != null) {
        // add read word line by line into our stopwords ArrayList
        sb.append(st); // use StringBuilder to append to list.
        sb.append(","); // append a comma as a delimiter
      }

      String stopwords_string = sb.toString(); // getting the stopwords string
from StringBuilder
      // With our stopwords list, we can use conf.set() to set the variable in
configurations, with title 'stopwords'.
```

```
      conf.set("stopwords", stopwords_string);
      // 2. Thereafter, pass the string to the mappers using conf.set() and
conf.get() in our Configurations object with setup() method.
```

After we get our stopwords_string. We are then able to finally perform conf.set() to ensure that our map function is able to utilize this stopwords string by using conf.get().

We make use of a setup() function, which basically allows us to do something before the map function, to transform this stopwords string into a stopWords list defined throughout the Map class, so that it can be used within the map function.

Our setup() function will look like the following:

```
    // utilizing setup() method in CommonWordsMapper class to incorporate our
stopwords fom stopwords.txt file.
    protected void setup(Context context) throws IOException,
InterruptedException {
      Configuration conf = context.getConfiguration(); // get configurations
that we have set in the main method.
      stopWords = new HashSet<String>(); // define our stopwords as a new list
variable
      // use conf.get() to get our stopwords_string that we created in main
method to use in MAP, then split it with "," and iterate to get all stop words
into a list.
      for (String word: conf.get("stopwords").split(",")) {
        stopWords.add(word); // adding each word to the stopWords set.
      }
    }
```

After we have gotten our stopwords list, we will then proceed to include this stopwords in our logic of map function:

```
// MAP FUNCTION
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString(), "
\t\n\r\f"); // converts line to string token, add new argument for delimiter.
      // file1 name:
      String file1 = new String("task1-input1.txt");
      // file2 name:
      String file2 = new String("task1-input2.txt");

      // get the current input file name being processed by map function by
using FileSplit.
      FileSplit fileSplit = (FileSplit)context.getInputSplit();
      String filename = fileSplit.getPath().getName();
```

```
      // We need to keep track of which file a tuple came from, you can
consider emitting a tuple like (keyword, filenum), where filenum is '1' or '2'
depending on which file it came from.
      // start of while loop to loop through the documents.
      String filenum = new String(""); // let file num be an empty string
initially.

      // Start iterating through the text file for tokens.
      while (itr.hasMoreTokens()) {
        String token = itr.nextToken(); // store the token as a string so that
we do not go to the next word.
        // the current word token needs to be used as a string to check
against the stopwords list.
        // If Y, we continue the loop. If N, then we want to set the word in
map.
        if (stopWords.contains(token)) {
          continue; // skip current token if the token is within the stopwords
list defined.
        } else {
          word.set(token); // setting the value of word (Text) with a token
(String)
          // for the rest of the tokens that are NOT in stopwords list, we
proceed with our normal job.
          if (filename.equals(file1)) {
            filenum = "1"; // write the integer value of 1 indicating that the
word came from input1.
          } else if (filename.equals(file2)) {
            filenum = "2"; // write the integer value of 2 indicating that the
word came from input2.
          }
          context.write(word, new Text(filenum));
        }
      }
    }
```

6. in the reduce function, modify the code to calculate the min count between the two files

   Our aim within the reduce function now is to ensure that we retrieve two dictionaries, dict1 and dict2, which are of format: Map<String,Integer>. The string here is referring to individual words in both dict1 and dict2. The integer value here refers to the number of occurrences in each of the dictionaries.

   We can interate through the list of val in values given by the mapper function in a for loop to insert values into each of the dictionaries through the following code:

```
      // We need to iterate through all the tuples (key: Text (word), value:
Text (filenum)) from the mapper.
```

```
      // use a HashMap to keep track of the counts in the reduce function for
each word and counts from each file.
      Map<String, Integer> dict1 = new HashMap<String, Integer>();
      Map<String, Integer> dict2 = new HashMap<String, Integer>();

      // get word and total count for each input file in two dictionaries,
dict1 & dict2.
      for (Text val: values) {

        // for values of filenum = "1"
        if (val.toString().equals("1") && dict1.get(key.toString()) == null) {
          dict1.put(key.toString(), 1); // initialize first occurrence of word
with integer of 1 in dict1.
        } else if (val.toString().equals("1") && dict1.get(key.toString()) !=
null) {
          dict1.put(key.toString(), dict1.get(key.toString()) + 1); // update
value of dictionary for word in dict1
        }

        // for values of filenum = "2"
        if (val.toString().equals("2") && dict2.get(key.toString()) == null) {
          dict2.put(key.toString(), 1); // initialize first occurrence of word
with integer of 1 in dict2.
        } else if (val.toString().equals("2") && dict2.get(key.toString()) !=
null) {
          dict2.put(key.toString(), dict2.get(key.toString()) + 1); // update
value of dictionary for word in dict2
        }
      }
```

Once we have these 2 dictionaries, we can start looking for the common words in each dictionary. The pseudocode is:

Start iterating through key value pairs in dict1

Check if the word exists in dict2

If it does not, skip the word by using continue;

If it exists, it is a common word, and we have to utilize 2 maps unSortedMap1 and unSortedMap2 to store the common word as the key, and the respective counts of common word in each input file.

These 2 unSortedMaps are important, we ultimately want to retrieve the minimum count of word occurrence for each common word between the 2 files in our cleanup() function of Reducer class.

```
      // start finding the common words in both dictionaries, by iterating
through the keys of dict1, and checking if they are in dict2.
      for (Map.Entry<String, Integer> entry: dict1.entrySet()){
```

```
        String word_in_dict1 = entry.getKey().toString(); // get word in dict1
        // check if the key (word_in_dict1) exists in dict2!
        if (dict2.containsKey(word_in_dict1) == false) {
          // In this case, since the word is in dict1, but not in dict2, we
will not care about them, as it is not a common word.
          continue;
        } else if (dict2.containsKey(word_in_dict1)) {
          // since sorting later will require all the keys, it has to be done
in the cleanup() of reducer.
          // Instead of emitting the output in reduce(), we will store them
into 2 HashMaps to get mincount later in cleanup().

          // Our ultimate goal here is to get 2 dictionaries, one dictionary
for words & counts in dict1, and another for words & counts in dict2.
          // store all the common words into unSortedMap1 - key: String
(word_in_dict1) - common word, value: Integer (counts from dict1)
          unSortedMap1.put(word_in_dict1, dict1.get(word_in_dict1));
          // store all the common words into unSortedMap2 - key: String
(word_in_dict1) - common word, value: Integer counts from dict2)
          unSortedMap2.put(word_in_dict1, dict2.get(word_in_dict1));

          // We will not emit the output here, because we need to do sorting
in the cleanup() function.
          // In our cleanup() function, we look forward to getting the minimum
counts of words between the two dictionaries to a new dictionary
unSortedMapFinal
          // then, we will look to sort by key to get the top 20 common words
by using a LinkedHashMap, sorted by descending order of VALUE (use
.comparingByValue(Comparator.reverseOrder())
        }
      }
```

7. Specify our cleanup() function, to get the unSortedMapFinal, which contains all the common words as the key, and the minimum count of word from both input1 and input2.

   We can do it through the following:

```
      // Iterate through unSortedMap1 and get the min count and store the
result in the unSortedMapFinal map.
      for (Map.Entry<String, Integer> entry: unSortedMap1.entrySet()) {
        // First check if the word being looked at is exactly the same:
        Integer min_count = Math.min(entry.getValue(),
unSortedMap2.get(entry.getKey()));
        unSortedMapFinal.put(entry.getKey(), min_count); // store the common
word with the appropriate minimum counts into unSortedMapFinal.
```

```
    }
```

8. Thereafter we need to sort the unSortedMapFinal by reverse order (DESCENDING) by VALUE (count of each common word), so that we are able to retrieve the top 20 common words by descending count in our final written output of Reducer.
9. We do the following for sorting: (it is important that our key is String here instead of count of word). IF you use count of words, you will eliminate all other tied values of words from your output, which is undesirable. Hence the common word should be used as the key here when we sort by values (.comparingByValue(Comparator.reverseOrder()))

   We use the following code:

```
    unSortedMapFinal.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
    .forEachOrdered(x -> reverseSortedMap.put(x.getKey(), x.getValue()));
```

10. Thereafter, we have our reverseSortedMap for descending order of key – value pairs.
11. We finally just need to output top 20 words. Set a counter for this in the code:

```
12.       // define counter variable to keep track of the total number of
    emitted outputs to be 0 first, to ensure 20 outputs of words.
13.       int counter = 0;
14.       for (Map.Entry<String, Integer> entry:
    reverseSortedMap.entrySet()) {
15.         // check for counter variable to be < 20, since we start from
    0.
16.         if (counter < 20) {
17.           // for loop will run as long as counter is < 20. Once counter
    = 20, we will stop emitting outputs in reducer.
18.           // set the result (IntWritable) from an Integer.
19.           result.set(entry.getValue());
20.           context.write(result, new Text(entry.getKey()));
21.           // System.out.println(entry.getKey().toString() + "," +
    entry.getValue());
22.         }
23.         counter = counter + 1; // increment counter for every
    iteration.
24.       }
```

We are done with our task! (END)

This thread will be updated with FAQs or clarifications about Assignment 1.

FAQ:

Will the grading consider speed? Or coding style?

- No, only correctness will be considered for grading.

When logging in to the school cluster in VSCode, it asked me to choose between linux, windows, or mac; which should I choose?

- Please choose Linux (the school cluster uses Linux).

compile_run script gave the error: "ValueError: invalid literal for int() with base 10: 'l'"

- Make sure in each line you are printing the count first before the keyword, i.e. <count>\t<keyword>.
- In general, if the checking script returns any error, you can figure out the cause of the problem by comparing "output.txt" (which is the output of your method) with "answer.txt" (which is the correct answer). If you see any differences, you can try to understand the reason for the differences.

Can I modify compile_run?

- No; The submission script only copies over your .java file, so it will be graded using the original version of ./compile_run.

Will input filenames always be task1-input1.txt and task1-input2.txt?

- Yes.

Do I need to worry about the case where the input file sizes are extremely large?

- No. During grading, the test files used for grading will not be significantly larger (or have extreme differences in the distribution of words) from the given data.

Do I need to worry about possible changes to the stopword file?

- No; the stopword file is fixed.

Could you explain the meaning of "different non-whitespace punctuation are considered different words"?

- You can interpret a "word" as any sequence of non-whitespace characters, including characters like '*', '.' and so on. Basically, other than tokenizing by whitespace, your program doesn't have to do any other string processing, and all non-whitespace characters can be handled the same.

Checklist if you have difficulties logging in to cluster:

- You are using SoC VPN (e.g. using Forticlient, which you can download from https://webvpn.comp.nus.edu.sg/remote/login?lang=en, or https://www.fortinet.com/support/product-downloads#vpn)
- You activated cluster permission (https://mysoc.nus.edu.sg/~myacct/)
- You are using your SoC account userid and password (may be different from your NUS account details). If unsure about your account ID, you can go to https://mysoc.nus.edu.sg/~myacct/, log in, and you should see "Your current email addresses are:" then an email like xyz@comp.nus.edu.sg, then xyz would be your SoC ID. If you need to reset the password, you can then go to My SoC Account -> Reset Password. (if you haven't logged in to it for a while, your previous password may have expired, so you may have to reset it).
- Check that ssh command should be: “ssh <SOC ID>@xcnc24.comp.nus.edu.sg” (or xcnc25, ..., xcnc31)
- If there are still issues, please email me (bhooi@comp.nus.edu.sg)

Where to store temporary / intermediate files generated during mapreduce? (if needed; these are optional)

- The easiest way is to store them in "commonwords/wc_output". The compile_run script automatically deletes the previous version of this folder (if any) before it runs. So, storing your temporary files here ensures that running compile_run multiple times doesn't cause errors due to the intermediate folder already being present.
- If you used some other folder to store your intermediate files, running compile_run multiple times may cause errors due to the folder already existing, unless you delete the folder yourself, e.g. in your Java code (which is also fine). In any case, for grading we will make sure to run your code in a clean directory, so this won't be an issue.

How to handle ties in frequency?

- For ties in frequency, you can order them arbitrarily. This applies to the selection of top 20 output as well. The grading script will automatically consider these cases when marking.

How do I know if I submitted successfully?

- If you see "You have successfully submitted" and with no error messages, it has been submitted. There is no need to submit anything on LumiNUS.

Hints for the assignment

Note that the following is just based on one approach; there are many other valid approaches. You can solve the problem in steps:

Step 1: ignore the stopwords and the sorting; just try to get the min count for each word for the two files. Use the assignment 0 code as a starting point and try to modify it for this case. How should the map function's emitted tuples be modified? (Hint: to get the current input file name, see https://stackoverflow.com/questions/19012482/how-to-get-the-input-file-name-in-the-mapper-in-a-hadoop-program and import FileSplit for this to work) Then in the reduce function, modify the code to calculate the min count between the two files. Some tips for this part:

- For the program to run, the file paths (addInputPath, setOutputPath) must be changed from assignment 0, since we now have 4 arguments (args[0] = input file 1, args[1] = input file 2, args[2] = stopwords, args[3] = output). Hadoop allows multiple input paths. Also, always make sure the mapper / reducer return types are correctly specified, i.e. "MyMapper extends Mapper<input key, input value, output key, output value>", and similarly for the reducer. You also have to correctly specify the types in your main method, i.e. "setMapOutputKeyClass(...), setOutputKeyClass(...)", and so on.
- Text is hadoop's wrapper around String. You can convert it to/from strings using toString() and "new Text(mystring)". Similarly, IntWritable is a wrapper around int (you can use get() / set() on it).
- If you are correct for this part, it will still give "Wrong Answer" since we ignored the stopwords / sorting, but you can check that "you" appeared a minimum of 55 times.
- When testing your code on the cluster, if there are compile errors, you can press ctrl-c a few times to stop the program (so you can read the error messages more easily).

Step 2: Incorporate the stop words list. One way is to read the stopwords into a string in the main method, then pass it to the mappers using the configuration object: see https://stackoverflow.com/questions/25432598/what-is-the-mapper-of-reducer-setup-used-for/25450627 for an example. This example also shows how to use the setup / cleanup methods.

- If you are correct for this part, you should see that the word "you" is no longer in the list of output, but "you," and "you." are.
- (There is an alternative approach using DistributedCache, but this is optional).

Step 3: Incorporate sorting. Since this requires all the keys, it has to be done either in the cleanup() of the reducer, or in a separate mapreduce job. For the former (which is easier), instead of emitting output in the reduce() function, you can instead store it into 2 maps, then in the reducer's cleanup(), get the minimum counts for each keyword, sort, and emit the top entries. You should be able to find snippets online to help you do these steps in java, e.g. sorting a map: https://howtodoinjava.com/java/sort/java-sort-map-by-values/.