CprE 419 Lab 2: Text Analysis using Hadoop Mapreduce

Shuo Wang

Experiment 2 (50 points):

Java codes are attached at the end.

1. Final results on “Shakespeare”:

Note that for initial letter “z”, there are two bigrams “zeal to” and “zounds i” having the highest frequency of 5.

```
[shuowang@n0 ~]$ hls /scr/shuowang/lab2/exp2/output
Found 3 items
-rw-r--r--   3 shuowang 419x          0 2015-01-31 03:43 /scr/shuowang/lab2/exp2
/output/_SUCCESS
-rw-r--r--   3 shuowang 419x        562 2015-01-31 03:43 /scr/shuowang/lab2/exp2
/output/part-r-00000
-rw-r--r--   3 shuowang 419x        555 2015-01-31 03:43 /scr/shuowang/lab2/exp2
/output/part-r-00001
[shuowang@n0 ~]$ hcat /scr/shuowang/lab2/exp2/output/part-r-00000
a       most frequent bigram(s): 'and the'              797
c       most frequent bigram(s): 'come to'              334
e       most frequent bigram(s): 'exeunt king'          192
g       most frequent bigram(s): 'give me'              370
i       most frequent bigram(s): 'i am'         1889
k       most frequent bigram(s): 'king henry'           694
m       most frequent bigram(s): 'my lord'              1669
o       most frequent bigram(s): 'of the'               1550
q       most frequent bigram(s): 'queen margaret'               152
s       most frequent bigram(s): 'shall be'             554
u       most frequent bigram(s): 'upon the'             303
w       most frequent bigram(s): 'with the'             622
y       most frequent bigram(s): 'you are'              715
[shuowang@n0 ~]$ hcat /scr/shuowang/lab2/exp2/output/part-r-00001
b       most frequent bigram(s): 'by the'               639
d       most frequent bigram(s): 'do not'               488
f       most frequent bigram(s): 'for the'              570
h       most frequent bigram(s): 'he is'                664
j       most frequent bigram(s): 'joan la'              60
l       most frequent bigram(s): 'like a'               554
n       most frequent bigram(s): 'no more'              534
p       most frequent bigram(s): 'pray you'             379
r       most frequent bigram(s): 'richard iii'          160
t       most frequent bigram(s): 'to the'               1671
v       most frequent bigram(s): 'very well'            62
x       most frequent bigram(s): 'xi then'              3
z       most frequent bigram(s): 'zeal to' & 'zounds i'         5
[shuowang@n0 ~]$
```

2. Final results on “Gutenberg”:

```
[shuowang@n0 ~]$ hcat /scr/shuowang/lab2/exp2/output/part-r-00000
a       most frequent bigram(s): 'and the'              373178
c       most frequent bigram(s): 'could not'            61126
e       most frequent bigram(s): 'end of'               31859
g       most frequent bigram(s): 'going to'             33285
i       most frequent bigram(s): 'in the'               751197
k       most frequent bigram(s): 'kind of'              23086
m       most frequent bigram(s): 'may be'               78394
o       most frequent bigram(s): 'of the'               1422537
q       most frequent bigram(s): 'question of'          4842
s       most frequent bigram(s): 'she had'              74872
u       most frequent bigram(s): 'upon the'             71796
w       most frequent bigram(s): 'with the'             202363
y       most frequent bigram(s): 'you are'              63810
[shuowang@n0 ~]$ hcat /scr/shuowang/lab2/exp2/output/part-r-00001
b       most frequent bigram(s): 'by the'               226443
d       most frequent bigram(s): 'did not'              91616
f       most frequent bigram(s): 'for the'              230315
h       most frequent bigram(s): 'he was'               179060
j       most frequent bigram(s): 'just as'              16760
l       most frequent bigram(s): 'like a'               46703
n       most frequent bigram(s): 'not to'               46036
p       most frequent bigram(s): 'project gutenberg'            60450
r       most frequent bigram(s): 'ready to'             12720
t       most frequent bigram(s): 'to the'               564351
v       most frequent bigram(s): 'very well'            11149
x       most frequent bigram(s): 'x trillionthis'               718
z       most frequent bigram(s): 'zipcorrected editions'                1984
```

To consider that bigrams might span lines of input, one possible solution is using [‘.’ ‘?’ ‘!’] to split atoms so that a sentence will not in different atoms. Within each atom, we can restructure the lines to find all the bigrams.

If we still use each line as atoms, the bigrams spanning different lines can only meet each other in the reduce step. So we need to assign the same key to the last word of last sentence and the first word in the next sentence in the map step.
For example, for the input <4,line4> and <5,line5>,
after map step we get:
<34,”first word of line4” + “4”>
<45,”last word of line4” + “4” + “whether there is [.?!] in the end”>
<45,”first word of line5” + “5”>
<56,”last word of line5” + “5” + “whether there is [.?!] in the end”>
Then in reduce step, the reducer for key “45” will decide whether the last word of line4 and the first word of line5 could form a bigram based on the presence of [.?!] at the end of line4.
By this extra mapreduce round, all bigrams spans lines can be found.

So to do the same thing in exp2 considering the bigrams spanning lines, I might use three rounds:
1. Find all bigrams spanning lines and attach it to the input file of round 2 (each bigram in a single line)
2. Do the count for each bigram
3. Group the bigrams by the initial letters and find the most frequent one within each group.

3. Java Code:

```
import java.io.*;
import java.lang.*;
import java.util.*;
import java.net.*;

import org.apache.hadoop.fs.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.util.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class Driver extends Configured implements Tool {

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

                int res = ToolRunner.run(new Configuration(), new Driver(), args);
                System.exit(res);

        } // End main

        public int run ( String[] args ) throws Exception {

                String input = "/class/s15419x/lab2/gutenberg";    // Change this accordingly
                String temp = "/scr/shuowang/lab2/exp2/temp";      // Change this accordingly
                String output = "/scr/shuowang/lab2/exp2/output/";  // Change this accordingly

                int reduce_tasks = 2;  // The number of reduce tasks that will be assigned to the job
                Configuration conf = new Configuration();

                // Create job for round 1: round 1 counts the frequencies for each bigram

                // Create the job
                Job job_one = new Job(conf, "Driver Program Round One");

                // Attach the job to this Driver
                job_one.setJarByClass(Driver.class);

                // Fix the number of reduce tasks to run
                // If not provided, the system decides on its own
                job_one.setNumReduceTasks(reduce_tasks);

                // The datatype of the Output Key
                // Must match with the declaration of the Reducer Class
                job_one.setOutputKeyClass(Text.class);

                // The datatype of the Output Value
                // Must match with the declaration of the Reducer Class
                job_one.setOutputValueClass(IntWritable.class);

                // The class that provides the map method
                job_one.setMapperClass(Map_One.class);

                // The class that provides the reduce method
                job_one.setReducerClass(Reduce_One.class);

                // Decides how the input will be split
                // We are using TextInputFormat which splits the data line by line
                // This means each map method receives one line as an input
```

```
		job_one.setInputFormatClass(TextInputFormat.class);

		// Decides the Output Format
		job_one.setOutputFormatClass(TextOutputFormat.class);

		// The input HDFS path for this job
		// The path can be a directory containing several files
		// You can add multiple input paths including multiple directories
		FileInputFormat.addInputPath(job_one, new Path(input));
		// FileInputFormat.addInputPath(job_one, new Path(another_input_path)); // This is legal

		// The output HDFS path for this job
		// The output path must be one and only one
		// This must not be shared with other running jobs in the system
		FileOutputFormat.setOutputPath(job_one, new Path(temp));
		// FileOutputFormat.setOutputPath(job_one, new Path(another_output_path)); // This is not allowed

		// Run the job
		job_one.waitForCompletion(true);


		// Create job for round 2: round 2 groups all the bigrams by their initial letters and choose the most
frequent bigram
		// The output of the previous job can be passed as the input to the next
		// The steps are as in job 1

		Job job_two = new Job(conf, "Driver Program Round Two");
		job_two.setJarByClass(Driver.class);
		job_two.setNumReduceTasks(reduce_tasks);

		job_two.setOutputKeyClass(Text.class);
		job_two.setOutputValueClass(Text.class);

		// If required the same Map / Reduce classes can also be used
		// Will depend on logic if separate Map / Reduce classes are needed
		// Here we show separate ones
		job_two.setMapperClass(Map_Two.class);
		job_two.setReducerClass(Reduce_Two.class);

		job_two.setInputFormatClass(TextInputFormat.class);
		job_two.setOutputFormatClass(TextOutputFormat.class);

		// The output of previous job set as input of the next
		FileInputFormat.addInputPath(job_two, new Path(temp));
		FileOutputFormat.setOutputPath(job_two, new Path(output));

		// Run the job
		job_two.waitForCompletion(true);

		return 0;

	} // End run

	// The Map Class
	// The input to the map method would be a LongWritable (long) key and Text (String) value
	// Notice the class declaration is done with LongWritable key and Text value
	// The TextInputFormat splits the data line by line.
	// The key for TextInputFormat is nothing but the line number and hence can be ignored
	// The value for the TextInputFormat is a line of text from the input
	// The map method can emit data using context.write() method
	// However, to match the class declaration, it must emit Text as key and IntWribale as value
	public static class Map_One extends Mapper<LongWritable, Text, Text, IntWritable>  {

		// Reuse objects to save overhead of object creation.
		private IntWritable one = new IntWritable(1);
```

```
		private Text bigram = new Text();

		// The map method
		public void map(LongWritable key, Text value, Context context)
                                                        throws IOException, InterruptedException  {

		// The TextInputFormat splits the data line by line.
		// So each map method receives one line from the input
		String line = value.toString();

		// Split the input line by sentence ending marks ".!?"
		String[] inputlines = line.toLowerCase().split("[.!?]");

		// a loop to deal with each part of the input line
		for (int i = 0; i < inputlines.length; i++)
		{

		// Remove every thing except letters, spaces and tabs, convert to lower case and tokenize to get
the individual words

		StringTokenizer tokens = new StringTokenizer(inputlines[i].replaceAll("[^a-zA-X    ]",
"").toLowerCase());

		String previous = null;

		while (tokens.hasMoreTokens()) {

		String current = tokens.nextToken();

		if (previous != null) {
                                        bigram.set(previous + " " + current);
                                        // Use context.write to emit values
                                        context.write(bigram, one);
                                                                        }
		previous = current;

                        } // End while

		} // End for loop

		} // End method "map"

	} // End Class Map_One

	// The reduce class
	// The key is Text and must match the datatype of the output key of the map method
	// The value is IntWritable and also must match the datatype of the output value of the map method
	public static class Reduce_One extends Reducer<Text, IntWritable, Text, IntWritable>  {

		// The reduce method
		// For key, we have an Iterable over all values associated with this key
		// The values come in a sorted fasion.
		public void reduce(Text key, Iterable<IntWritable> values, Context context)
                                                                                        throws
IOException, InterruptedException  {

                        int sum = 0;

                        for (IntWritable val : values) {
                                context.progress();
                                sum += val.get();
                        }

                        // Use context.write to emit values
                        context.write(key, new IntWritable(sum));
```

```
		} // End method "reduce"

	} // End Class Reduce_One

	// The second Map Class
	public static class Map_Two extends Mapper<LongWritable, Text, Text, Text>  {


		private Text outpart1 = new Text();
		private Text outpart2 = new Text();

		public void map(LongWritable key, Text value, Context context)
				throws IOException, InterruptedException  {


			// The TextInputFormat splits the data line by line.
			// So each map method receives one line from the input
			String line = value.toString();
			outpart1.set(line.substring(0,1));
			outpart2.set(line);

			context.write(outpart1, outpart2);


		}  // End method "map"

	}  // End Class Map_Two

	// The second Reduce class
	public static class Reduce_Two extends Reducer<Text, Text, Text, Text>  {

		private Text out = new Text();

		public void reduce(Text key, Iterable<Text> values, Context context)
				throws IOException, InterruptedException  {

			int frequency = 0;
			String frequentbigram = "";
			String[] buffer;

			for (Text val : values) {
				context.progress();
				 buffer = val.toString().split("		");
				if ( frequency==Integer.parseInt(buffer[1])){
					frequency = Integer.parseInt(buffer[1]);
					frequentbigram = frequentbigram + "' & '" + buffer[0];
				} // end if
				if ( frequency<Integer.parseInt(buffer[1])){
					frequency = Integer.parseInt(buffer[1]);
					frequentbigram = buffer[0];
				} // end if


				out.set("most frequent bigram(s): '" + frequentbigram + "'			" +
Integer.toString(frequency));

			}
			context.write(key, out);

		}  // End method "reduce"

	}  // End Class Reduce_Two

}
```