Massive Data Processing - Lab 2

Emilie Leblanc, emilie.leblanc@student.ecp.fr

March 18, 2017

I am initially a student from ESSEC and I follow the MSc. in Data Sciences and Business Analytics (Centrale-ESSEC) as well as Centrale's Applied Mathematics (OMA) option.
My git repository is here : https://github.com/EmilieLeb/BDPA_Assign2_ELEBLAN and there should be the commit tree inside.

1 Introduction - Hadoop setup

For my Hadoop setup, I have used a VirtualBox in which I have installed Ubuntu 16.04 and Java. I have also created a new user, hduser, on which I installed Hadoop 2.7.3.

2 Pre-processing the input

2.1 Do simple wordcount

I have started by doing a simple wordcount because it will be useful for us later. It also allows me to show the process I followed.

After formating the namenode and starting the dfs and the yarn, I checked that everything was up and running with jps.

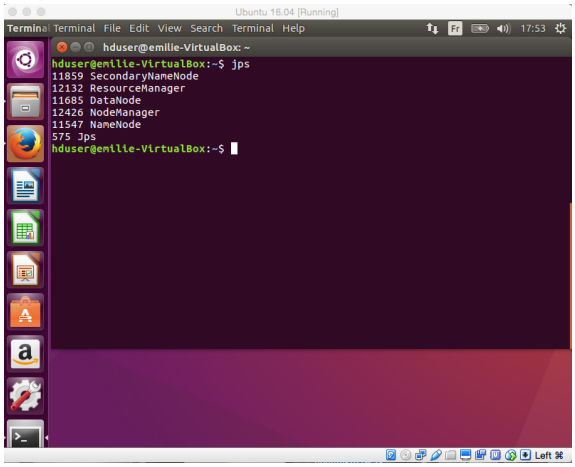

Figure 1: Jps command line.

Since everything was fine, I downloaded with curl that pg100.txt file from gutenberg, as instructed in the assignment. I then put the inputdata in the HDFS.

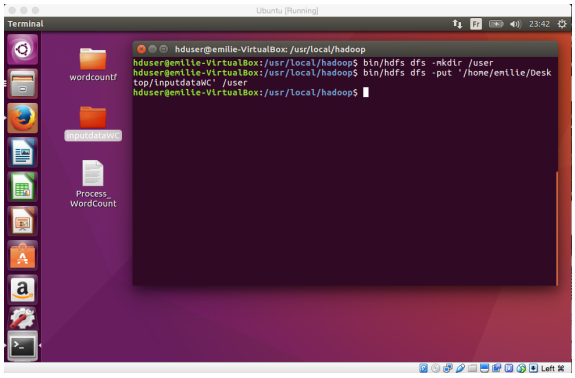

Figure 2: Inserting inputdata in HDFS.

On my Desktop, I had a file containing the WordCountpro.java code I wished to run.

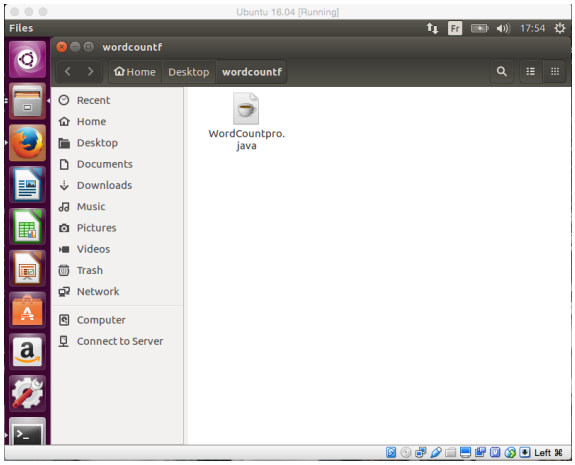

Figure 3: Wordcount initial folder containing code.

Then, I created the jar file and ran it by using the following commands (all of these commands, for every process, can be found in the Process file). And the job ran correctly.

```
#############################
### CREATE WORDCOUNT ###
#############################

#Compile

javac -classpath /usr/local/hadoop/share/hadoop/common/hadoop-common
    -2.7.3.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-
    mapreduce-client-core-2.7.3.jar:/usr/local/hadoop/share/hadoop/
    common/lib/commons-cli-1.2.jar -d /home/emilie/Desktop/wordcountf
    *.java

#Convert into Jar File

jar -cvf wordcountj.jar -C /home/emilie/Desktop/wordcountf/wordcountc .

#Run JAR File

bin/hadoop jar /home/emilie/Desktop/wordcountf/wordcountj.jar
    WordCountpro /user/inputdata/pg100.txt output
```

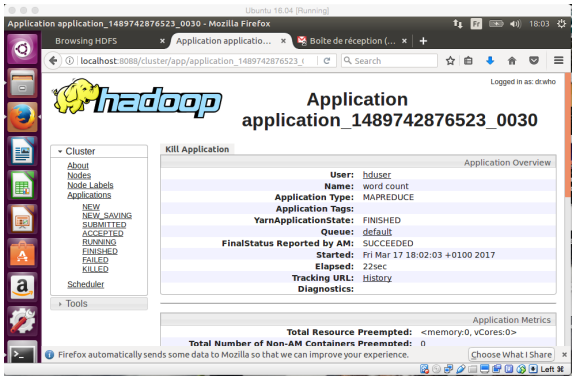

Figure 4: Success of the Wordcount job.

Extract of the obtained wordcount file:

```
address 19
addresses       2
addressing      1
addrest 2
adds    5
adhere  2
adheres 3
adieu   101
```

2.2 Create stopwords file

I proceeded the same way to create the stopwords file.

Extract of the obtained stopwords file:

```
a       14800
and     26847
as      5988
be      7146
but     6287
by      4490
d 8961
```

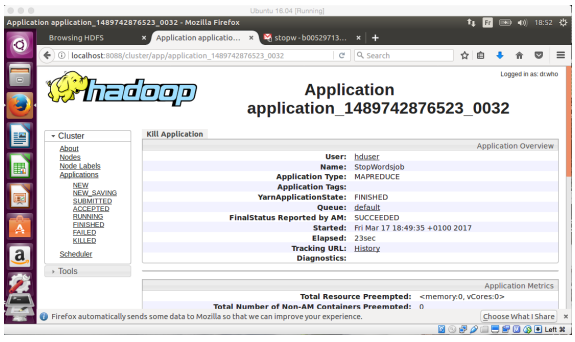

Figure 5: Success of the Stopwords job.

2.3 Preprocessing

In order to finalize the preprocessing, I proceeded in the following manner.

Main function :

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

                // Setting the configuration such that we can upload the documents in command line
                Configuration conf = new Configuration();
                conf.set("stopWords", args[2]);
                conf.set("wordCount", args[3]);

                Job job = Job.getInstance(conf, "PreProcessing");
                job.setJarByClass(PreProcessingpro.class);
                job.setMapperClass(Map.class);
                job.setReducerClass(Reduce.class);

                job.setOutputKeyClass(IntWritable.class);
                job.setOutputValueClass(Text.class);

                FileInputFormat.addInputPath(job, new Path(args[0]));
                FileOutputFormat.setOutputPath(job, new Path(args[1]));

                job.waitForCompletion(true);

                // Problem with the number of lines: does not seem to work
                System.out.println("Number of lines = " + totalLines);
}
```

Mapper :

```
        public static class Map extends Mapper<Object, Text, IntWritable, Text>{

                // Initialise the stop words
                private Text wordText = new Text();
                private String word = new String();
                String stopWords = "";

                // Upload the stop words
                protected void setup(Context context) throws IOException, InterruptedException {

                        // Go get the file
                        String stopWordsPath = context.getConfiguration().get("stopWords");
                        Path openStopWords = new Path(stopWordsPath);
                        FileSystem fs = FileSystem.get(new Configuration());
                        BufferedReader stopWordsFile = new BufferedReader(new InputStreamReader(fs.open(openStopWords)));
                        String line;
                        line= stopWordsFile.readLine();

                        // Upload each word one by one
                        try{
                                while (line != null){
                                        stopWords += line + ",";
```

```
23                                          line = stopWordsFile.readLine()
                                                ;
24                                  }
25                          } finally {
26                                  stopWordsFile.close();
27                          }
28  }
29
30                  public void map(Object key, Text value, Context context
                        ) throws IOException, InterruptedException {
31                          // Initialisation
32                          String wordsofline = value.toString();
33                          String seenwords = "";
34                          StringTokenizer token = new StringTokenizer(
                                wordsofline , "
                                thesevalueswontshowcorrecltyinlatex");
35
36                          // If word has not been seen before in the line
                                and is not a stop word, keep it
37                          while (token.hasMoreTokens()) {
38                                  wordText.set(token.nextToken());
39                                  word = wordText.toString();
40                                  if (!stopWords.contains(word) && !
                                        seenwords.contains(word)){
41                                  seenwords = seenwords + word + " ";
42                                  }
43                          }
44                          if (seenwords.length()>0) {
45                                  totalLines.set(totalLines.get()+1);
46                                  seenwords = seenwords.substring(0,
                                        seenwords.length()-1);
47                                  Text seenwordsText = new Text(seenwords
                                        );
48                                  context.write(totalLines, seenwordsText
                                        );
49                          }
50  }
```

Reducer :

```
1   public static class Reduce extends Reducer<IntWritable,Text,IntWritable
        ,Text> {
2
3           // Create a sort of dictionary to count the occurrences of
                words to order them
4           private HashMap<String, Integer> dict = new HashMap<String,
                Integer>();
5
6           // Import word count
7           protected void setup(Context context) throws IOException,
                InterruptedException {
8
9                   // Load file
10                  String wordCountPath = context.getConfiguration().get("
                        wordCount");
11                  Path openWordCount = new Path(wordCountPath);
12                  FileSystem fs = FileSystem.get(new Configuration());
13                  BufferedReader wordCountFile =new BufferedReader(new
                        InputStreamReader(fs.open(openWordCount)));
14                  String line;
```

```
15                  line = wordCountFile.readLine();
16
17                  // For each line, get the word and the number of values
18                  try{
19                  while (line != null){
20                      String keyvalue[] = line.split("\t");
21                      String word = keyvalue[0];
22                      int nbcount = Integer.parseInt(keyvalue[1]);
23                      dict.put(word, nbcount);
24                      line = wordCountFile.readLine();
25                  }
26                  } finally {
27                          wordCountFile.close();
28                  }
29          }
30
31          public void reduce(IntWritable key, Iterable<Text> values,
                Context context) throws IOException, InterruptedException {
32
33              for (Text value : values){
34
35                  // Create a sort of sub dictionary per line associating
                        word - count
36                  HashMap<String, Integer> subdict = new HashMap<String,
                        Integer>();
37
38                  // Take the words and counts of the associated line
39                  String line = value.toString();
40                  StringTokenizer token = new StringTokenizer(line);
41
42                  while (token.hasMoreTokens()) {
43                      String word = token.nextToken();
44                      subdict.put(word.toLowerCase(), dict.get(word));
45                  }
46
47                  // Order the words of that line
48                  List<String> HMKeys = new ArrayList<String>(subdict.
                        keySet());
49                  List<Integer> HMValues = new ArrayList<Integer>(subdict
                        .values());
50                  Collections.sort(HMValues);
51
52                  // Use that ordered list
53                  LinkedList<String> list = new LinkedList<String>();
54                  Iterator<Integer> HMItvalue = HMValues.iterator();
55
56                  while (HMItvalue.hasNext()) {
57
58                      Integer v = HMItvalue.next();
59                      Iterator<String> HMItkey = HMKeys.iterator();
60
61                      while (HMItkey.hasNext()) {
62
63                          String sortedWords = HMItkey.next();
64                          Integer comp1 = subdict.get(sortedWords);
65                          Integer comp2 = v;
66
67                          if (comp1.equals(comp2)) {
```

```
                HMItkey.remove();
                list.add(sortedWords);
            }

}
```

Noticeably, these settings allow us to directly set the source, the wordcount and the stopwords files in a simple command line, such as:

```
bin/hadoop jar /home/emilie/Desktop/preprocessingf/preprocessingj.jar
    PreProcessingpro /user/inputdata/pg100.txt output /user/inputdata/
    stopwords.txt /user/inputdata/wordcount.txt
```

As seen in the screenshot, the preprocessing job took 23sec.

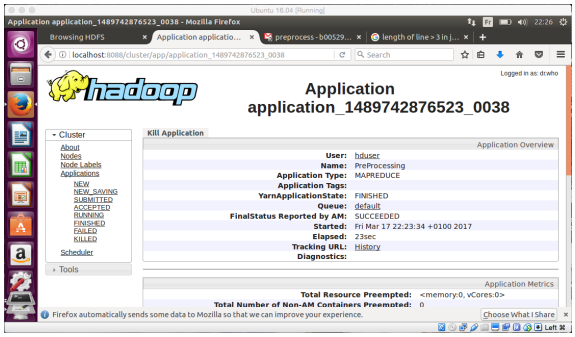

Figure 6: Preprocessing job success.

Extract of the preprocessing output file :

```
1       ebook complete works shakespeare william gutenberg project the
2       shakespeare william
3       anyone anywhere ebook cost use this
4       restrictions whatsoever copy almost away give may you
5       included license re gutenberg project terms under use
6       online gutenberg www ebook org
7       copyrighted details below ebook gutenberg project this
8       guidelines copyright file please follow
9       title complete works shakespeare william the
10      author shakespeare william
```

Ordering the words in the individual lines was a quite laborious task... Moreover, as you can see, I did not manage to extract the total number of lines.

3 Set-similarity joins

In this part, I only had time to do the pairwise comparisons with Jaccard similarity, so I did not implement the Inverted index. There also was not many space left on my VM so I limited the number of lines taken for the pairwise similarities to 50 and then to 1 000.

3.1 PairWise comparisons

I proceeded as following.

Main function :

```
        public static void main(String[] args) throws Exception {
                // As usual..
                Configuration conf = new Configuration();
                Job job = Job.getInstance(conf, "PairWise");
                job.setJarByClass(PairWisepro.class);
                job.setMapperClass(Map.class);
                job.setReducerClass(Reduce.class);
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(Text.class);
                FileInputFormat.addInputPath(job, new Path(args[0]));
                FileOutputFormat.setOutputPath(job, new Path(args[1]));

                job.waitForCompletion(true);
}
```

Mapper :

```
public static class Map extends Mapper<Object, Text, Text, Text>{

                // Since we did not manage to find the exact line
                   number, we set it arbitrarily (50, then 1000 and
                   more if we can)
                private int fullSize = 1000;

                // Create another kind of counter to be sure we don t
                   exceed the fullSize
                private IntWritable lineKey = new IntWritable(0);

                public void map(Object key, Text value, Context context
                   ) throws IOException, InterruptedException {
                        lineKey.set(lineKey.get()+1);

                        // For each line as long as we are under
                           fullSize limit
                        if (lineKey.get() <= fullSize){

                                // We split the line between the line
                                   number and the string
                                String [] lineValueIt = value.toString
                                   ().split("\t");
                                String lineValue = lineValueIt[0];
                                int lineKey = Integer.parseInt(
                                   lineValueIt[0]);
                                String line = lineValueIt[1];
                                Text lineText = new Text(line);

                                // We then create all pairs of lines
                                   that we will use for the similarity
                                for (int compLineKey = 1; compLineKey<=
                                   fullSize; compLineKey++){
```

```
24                                          String newKey = new String();
25                                          String compLineValue = String.
                                                valueOf(compLineKey);
26                                          // According to their positions
                                                 we put the first one first
                                                 in the comparison couple
27                                          if (compLineKey != lineKey) {
28                                                  if (lineKey <
                                                        compLineKey){
29                                                          newKey =
                                                                lineValue +
                                                                 "," +
                                                                compLineValue
                                                                ;
30                                                  } else {
31                                                          newKey =
                                                                compLineValue
                                                                 + "," +
                                                                lineValue;
32                                                  }
33                                                  Text newKeyText = new
                                                        Text(newKey);
34                                                  context.write(
                                                        newKeyText,
                                                        lineText);
35                                          }
36                                  }
37                          }
38                  }
39  }
```

Reducer :

```
1   public static class Reduce extends Reducer<Text,Text,Text,Text> {
2
3                   public void reduce(Text key, Iterable<Text> values,
                        Context context) throws IOException,
                        InterruptedException {
4
5                           // We then create an array with the lines of
                                the same key
6                           List<String> linesOfKey = new ArrayList<String
                                >(2);
7                           for (Text value : values){
8                                   linesOfKey.add(value.toString());
9                           }
10
11                          // After checking that we compute a couple only
                                 once . .
12                          if (linesOfKey.size() == 2){
13                                  String d1 = linesOfKey.get(0);
14                                  String d2 = linesOfKey.get(1);
15                                  String [] wordsOf1 = d1.split(" ");
16                                  String [] wordsOf2 = d2.split(" ");
17
18                                  // .. We compute the Jacquard
                                        similarity as enunciated in the
                                        assignment
19                                  int intersection = 0;
20                                  int union = 0;
```

```
21
22                                  // inspiration from : https://github.
                                        com/tdebatty/java-string-similarity
                                        /blob/master/src/main/java/info/
                                        debatty/java/stringsimilarity/
                                        Jaccard.java
23                                  for (String word : wordsOf1){
24
25                                          // If a word is in both, add
                                                one to intersection. Always
                                                 add one to union.
26                                          if (d2.contains(word)){
27                                                  intersection ++;
28                                          }
29                                          union++;
30                                          // Now check the leftover part:
                                                 only add to union if we
                                                have not seen the word
                                                before, i.e. if it is only
                                                in wordsOf2
31                                          for (String word : wordsOf2){
32                                                  if (!(d1.contains(word)
                                                        )) {
33                                                          union++;
34                                                  }
35
36                                  float Jacsim = (float) intersection /
                                        union;
37
38                                  // We now only return the couples that
                                        have a Jacquard similarity over 0.8
39                                  if (Jacsim > 0.8){
40                                          String out = d1 + " - " + d2 +
                                                " - Jacsim = " + String.
                                                valueOf(Jacsim);
41                                          Text value = new Text(out);
42                                          context.write(key, value);
43                                  }
44                                  // Increment our counter !!
45                                  context.getCounter(counting.
                                        NUMBER_OF_COMPARISONS).increment(1)
                                        ;
46                          }
47                  }
48  }
```

Of course, I am here taking the preprocessed pg100.txt as input, not simply pg100.txt (see Process file if necessary).

As you can see in the two following screenshots, running the job for 50 lines took 25sec, and for lines it took 30sec.

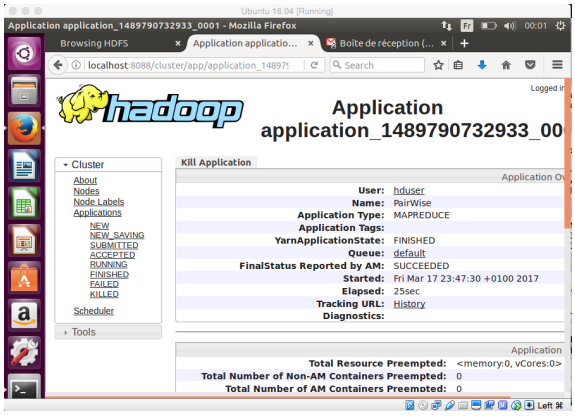

Figure 7: Pairwise job success with 50 lines.

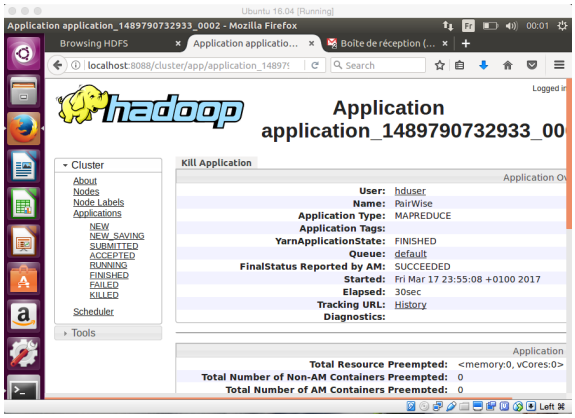

Figure 8: Pairwise job success with 1 000 lines.

The number of comparisons vary: 1 225 for 50 lines and 499 500 for 1 000 lines (as seen in the two following screenshots). This is coherent with the fact that there should be $\frac{n*(n-1)}{2}$ computations possible (so to do), with n the number of lines.

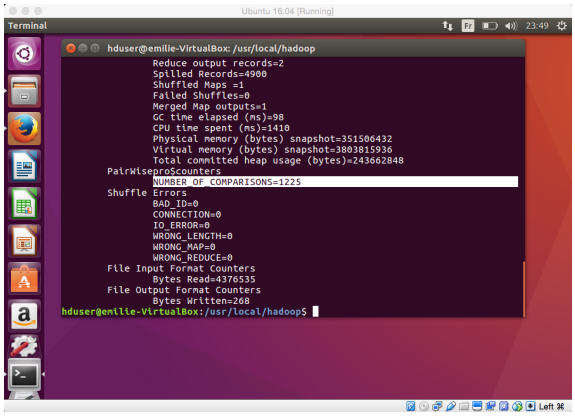

Figure 9: Number of comparisons for 50 lines.

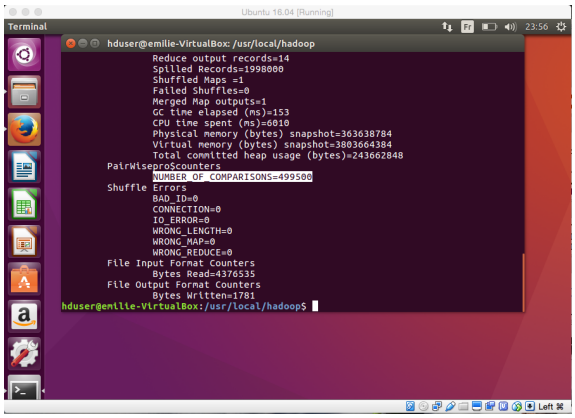

Figure 10: Number of comparisons for 1 000 lines.

I chose only to output the lines with a Jaccard similarity over 0,8 to skip comparisons between non-similar lines.

Extract of the pairwise comparison for 50 lines :

```
22,38  version works complete this william the electronic of - version works complete this william the electronic of - Jacsim = 1.0
23,39  copyright world 1993 1990 inc library shakespeare is by - copyright world 1993 1990 inc library shakespeare is by and - Jacsim = 0.9
```

Extract of the pairwise comparison for 1 000 lines :

```
2,132  shakespeare william - shakespeare william - Jacsim = 1.0
22,122  version works complete this william the electronic of - version works complete this william the electronic of - Jacsim = 1.0
22,38  version works complete this william the electronic of - version works complete this william the electronic of - Jacsim = 1.0
23,123  copyright world 1993 1990 inc library shakespeare is by and - copyright world 1993 1990 inc library shakespeare is by and - Jacsim = 1.0
23,39  copyright world 1993 1990 inc library shakespeare is by - copyright world 1993 1990 inc library shakespeare is by and - Jacsim = 0.9
24,124  college illinois benedictine provided etext project gutenberg by of - college illinois benedictine provided etext project gutenberg by of - Jacsim = 1.0
25,125  readable machine permission with be may copies electronic and - readable machine permission with be may copies electronic and -
```

```
    Jacsim = 1.0
8 26,126  your long such as so others distributed copies are for - your
    long such as so others distributed copies are for - Jacsim = 1.0
```

3.2 Inverted Index and prefix filtering

As said previously, I did not have the time to implement it.

3.3 Explain the difference in the number of comparisons and the computation time

I am guessing that the prefix filtering will generate a drop in execution time and reduce the number of comparisons. Indeed, as I understood it, it allows us to compute only the couples likely to be similar (thus removing many unnecessary comparisons). It is based on comparing the least common words only.