Assignment 1

Quentin Moret

A variation of an inverted index

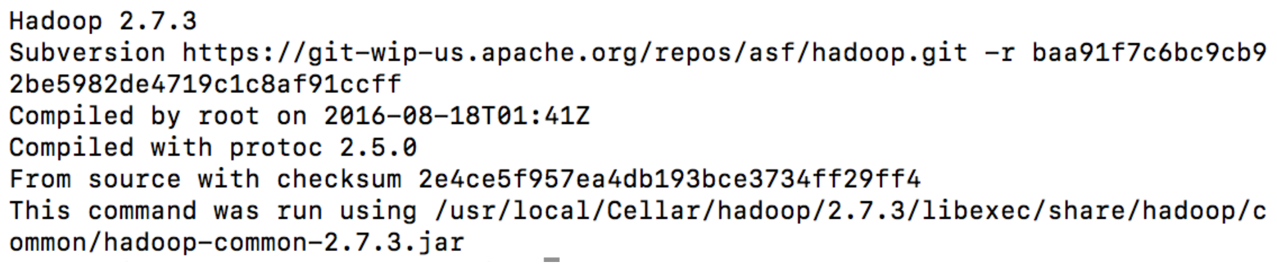
Abstract

We implement an inverted index on the complete works of William Shakespear, Mark Twain and Jane Austen.

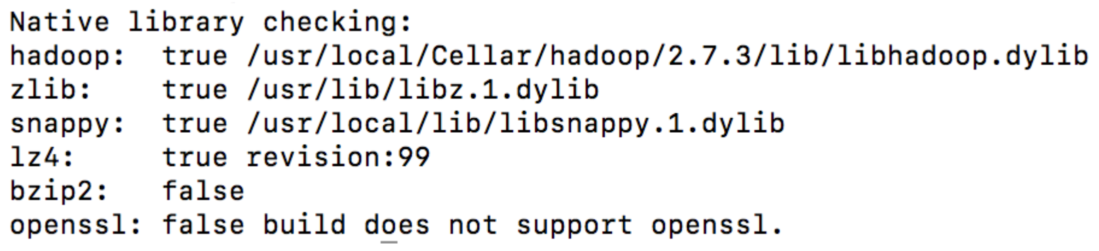
First we identify stop words to then remove them, then we implement a basic inverted index without the stop words, finally we add a frequency count to the inverted index

1. Hadoop setup

Hadoop is configured in pseudo-distributed mode on MacOS 10.12, using version 2.7.3. Native libraries have been installed:



```
Hadoop 2.7.3
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r baa91f7c6bc9cb9
2be5982de4719c1c8af91ccff
Compiled by root on 2016-08-18T01:41Z
Compiled with protoc 2.5.0
From source with checksum 2e4ce5f957ea4db193bce3734ff29ff4
This command was run using /usr/local/Cellar/hadoop/2.7.3/libexec/share/hadoop/c
ommon/hadoop-common-2.7.3.jar
```



```
Native library checking:
hadoop:  true /usr/local/Cellar/hadoop/2.7.3/lib/libhadoop.dylib
zlib:    true /usr/lib/libz.1.dylib
snappy:  true /usr/local/lib/libsnappy.1.dylib
lz4:     true revision:99
bzip2:   false
openssl: false build does not support openssl.
```

Java files are compiled in the shell (using *–cp* command including Hadoop class path), then Jar file is created, and executed in Hadoop. Eventually, the output file is copied to local storage.

Example for StopWords:

```
$ javac StopWords.java -cp $(hadoop classpath)
$ jar cf sw.jar StopWords*.class
$ hadoop jar sw.jar StopWords StopWords/input StopWords/output
$ hdfs dfs -get StopWords/output/part-r-00000 out.txt
```

2. Stop words

To detect the stop words, we use a word count map reducer to which we add a check of the number of occurrences of the word. If this number of occurrences gets greater than 4000, we return this word as a stop word.

Input : Data is uploaded on HDFS using bash command

```
$ hdfs dfs –put input StopWords/input
```

Data preprocessing: we remove special characters in the string tokenizer, by declaring them as delimiters.

Implementation:
Mapper:

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

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

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
        String line = value.toString().toLowerCase();
        StringTokenizer itr = new StringTokenizer(line , " \t\n\r&\\-
_!.;,()\"\'/:+=$[]§?#*");
  while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        }
    }
  }
```

Reducer

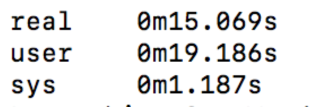
```
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      if (sum>4000){
      result.set(sum);
      context.write(key, null);
      }
    }
  }
```

Main

```
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "stopwords");
    job.setJarByClass(StopWords.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Execution time



```
real    0m15.069s
user    0m19.186s
sys     0m1.187s
```

In order to concatenate the generated output in a single file, we use the following shell command:

```
$ hadoop fs -cat StopWords/output/part\* | hadoop fs -put -
StopWords/stopwords.csv
```

Extract from `stopwords.csv`:

```
a                                    could
about                                d
after                                day
again                                did
all                                  do
am                                   down
an                                   ever
and                                  every
any                                  first
are                                  for
as                                   from
at                                   get
away                                 go
be                                   good
been                                 got
before                               great
but                                  had
by                                   has
can                                  have
come
```

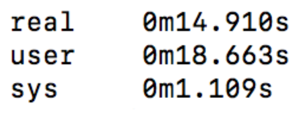
We notice that the majority of words are indeed stop words, but some are debatable ("day, away, great…). We could adjust this by increasing the minimum number of occurences required to be flagged as stop word (here 4000).

i. Using 10 reducers

We add the following code to our main:

```
job.setNumReduceTasks(10);
```

Execution time:



```
real	0m14.910s
user	0m18.663s
sys	0m1.109s
```

(Obtained with MacOS bash `time` function

We notice a slight increase in performance, because the job is parallelized.

ii. Using a combiner

We add the following code to our main:

```
job.setCombinerClass(MyCombiner.class);
```

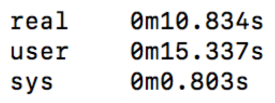
And we define the following combiner class:

```
  public static class MyCombiner
       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();
      }
      if (sum>4000){
      result.set(sum);
      context.write(key, result);
      }
    }
  }
```

Which is basically the same class as our reducer except that it outputs the value (the sum) so that the reducer can use it.

Execution time



```
real	0m10.834s
user	0m15.337s
sys	0m0.803s
```

We get an even greater increase in performance, because the reducer work on pre-aggregated results.

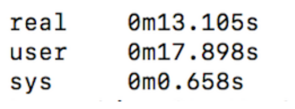
iii. Using compression

We add the following code to our main:

```
conf.set("mapreduce.map.output.compress", "true");
conf.set("mapreduce.map.output.compress.codec",
       "org.apache.hadoop.io.compress.SnappyCodec");
```

The choice to use SnappyCodec was perfectly arbitrary.

Execution time



```
real	0m13.105s
user	0m17.898s
sys	0m0.658s
```

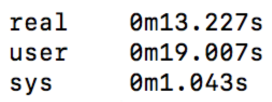
We don't get an increase here, which may be explained by the fact that we need to compress and decompress, and do not gain performance on data transfer since we use a single machine configuration.

iv. Using 50 reducers

We remove compression, and add the following code to our main:

```
job.setNumReduceTasks(50);
```

Execution time



```
real	0m13.227s
user	0m19.007s
sys	0m1.043s
```

As compared with 10 reducers with combiner, this is much slower. The number of reducers is here too high for good performance.

3. Inverted index

The inverted index gives for a given word the list of documents in which it is present.
We thus need to modify both our mapper and our reducer. The mapper needs to get the file in which the word have been found, and set it as value. And the reducer needs to concatenate the list of unique files in which the word have been found.
To do this, we use

```
String fileName = ((FileSplit) context.getInputSplit()).getPath().getName();
```

In order to skip stop words, we need to add an access to the csv output of the StopWords jar that is located in HDFS, load this file, parse it and save it as a string, and then test for each word if it is contained in this string.

We implement this by overwriting the `setup()` function of our mapper, and by getting the path `to stopwords.csv` in the `main` with `conf.set("stopWords", args[2]);`

Mapper:

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

        private Text word = new Text();

        String stopWords = "";

     protected void setup(Context context) throws IOException,
InterruptedException {
            String stopWordsPath = context.getConfiguration().get("stopWords");
            Path ofile = new Path(stopWordsPath);
            FileSystem fs = FileSystem.get(new Configuration());
            BufferedReader br =new BufferedReader(new
InputStreamReader(fs.open(ofile)));
            String line;
            line=br.readLine();
            try{
                while (line != null){
                    stopWords += line + ",";
                    line = br.readLine();
                    }
            } finally {
              br.close();
            }
        }

public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
            String line = value.toString().toLowerCase();
            StringTokenizer itr = new StringTokenizer(line , " \t\n\r&\\-
_!.;,()\"\'/:+=$[]§?#*|{}~0123456789<>@`");
            while (itr.hasMoreTokens()) {
                String fileName = ((FileSplit)
context.getInputSplit()).getPath().getName();
                word.set(itr.nextToken());
                if (!stopWords.contains(word.toString())){
                    Text file = new Text(fileName);
                    context.write(word, file);
                }
            }
        }
    }
```

Reducers (including the counter of next question)

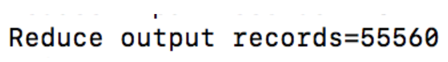
```
public static class ConcatReducer
         extends Reducer<Text,Text,Text,Text> {
    private Text result = new Text();

    public void reduce(Text key, Iterable<Text> values,
                                        Context context
                                        ) throws IOException,
InterruptedException {
        int ndoc = 0;
        String list = "";
        for (Text val : values) {
             String file = val.toString();
             if (list == "") {
                     list = file;
                     ndoc = 1;
             }
             if (!(list.contains(file))){
                     list = list + ", " + file;
                     ndoc+=1;
            }
        }
        if (ndoc == 1){
            context.getCounter(counters.APPEAR_IN_ONE_DOC_ONLY).increment(1);
        }
        Text result = new Text(list);
        result.set(sum);
        context.write(key, result);
    }
}
```

We get the expected output :

```
aachen   pg3200.txt
aar      pg3200.txt
aaron    pg3200.txt, pg100.txt
aart     pg3200.txt
aartist  pg3200.txt
aback    pg3200.txt
abaft    pg3200.txt
abaissiez      pg100.txt
abana    pg3200.txt
abandon  pg3200.txt, pg100.txt
abandoned      pg100.txt, pg31100.txt, pg3200.txt
abandoning     pg3200.txt
abandonment    pg3200.txt
abandons pg3200.txt
abase    pg100.txt
abasement      pg3200.txt
abash    pg3200.txt, pg100.txt
abashed  pg3200.txt, pg31100.txt
abate    pg31100.txt, pg100.txt, pg3200.txt
abated   pg3200.txt, pg100.txt
abatement      pg100.txt, pg31100.txt, pg3200.txt
```

b) The counter that counts the number of unique words that exist in the document corpus (excluding stop words) is the following :



```
Reduce output records=55560
```

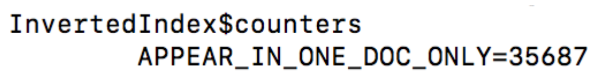
We define our own counter for the number of word that appear in only one document by first defining a counter class :

```
enum counters {APPEAR_IN_ONE_DOC_ONLY};
```

Then we increment the counter in the main (as shown in reducer above), using a variable ndoc that counts the number of docs in which the word is present :

```
if (ndoc == 1){
          context.getCounter(counters.APPEAR_IN_ONE_DOC_ONLY).increment(1);
      }
```

We find the requested information in the logs :



```
InvertedIndex$counters
        APPEAR_IN_ONE_DOC_ONLY=35687
```

We want to get the value of our counter written in a file
For this, we add in the main, after the `job.waitForCompletion(true) :`

```
Path pt = new Path(  "hdfs://localhost:9000/user/quentin/InvertedIndex/counter");
FileSystem fs = pt.getFileSystem(conf);
BufferedWriter bw=new BufferedWriter(new OutputStreamWriter(fs.create(pt,true)));
long cnt =
job.getCounters().findCounter(counters.APPEAR_IN_ONE_DOC_ONLY).getValue();
bw.write("Number of words that appear only in one document : " +
String.valueOf(cnt));
bw.close();
```

We access it with

```
$ hdfs dfs -get InvertedIndex/counter counter.txt
```

We get the expected text :

```
Number of words that appear only in one document : 35687
```

d) The variation of the inverted index consists in adding the number of occurences of each word in each of the documents in which it appears.

The mapper remains the same. The combiner needs to concatenate values from the mapper (files), and add a frequency. We use for this dynamic counters, that need to be reset at each call to the combiner.

Then the reducer needs to take the combiner outputs, parse them, and add the frequencies corresponding to the same file.

Combiner :

```
public static class ConcatCombiner
         extends Reducer<Text,Text,Text,Text> {

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

            // Reset counters
            context.getCounter("nPerDoc", "pg100.txt").setValue(0);
            context.getCounter("nPerDoc", "pg31100.txt").setValue(0);
            context.getCounter("nPerDoc", "pg3200.txt").setValue(0);

            List<String> list = new ArrayList<String>();
            for (Text val : values) {
                 String file = val.toString();
                 if (list.isEmpty()) {
                         list.add(file);
                 }
                 if (!(list.contains(file))){
                    list.add(file);
                 }
                 context.getCounter("nPerDoc", file).increment(1);
            }

            ListIterator<String> it = list.listIterator();
            String s = "";
            while(it.hasNext()){
                String str = it.next();
                String fileFreq = str + "#" +
String.valueOf(context.getCounter("nPerDoc", str).getValue());
                if (s.equals("")) {
                    s = fileFreq;
                } else {
                s = s + "," + fileFreq;
                }
            }

            Text result = new Text(s);
            context.write(key, result);
    }
}
```

Reducer

```
public static class ConcatReducer
         extends Reducer<Text,Text,Text,Text> {

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

            int ndoc = 0;
            List<String> list = new ArrayList<String>();
            String file = "";
            int count = 0;
            for (Text val : values) {
                 String line = val.toString();
                 StringTokenizer itr = new StringTokenizer(line , ",#");
                 while (itr.hasMoreTokens()) {
                     file = itr.nextToken();
                     count = Integer.parseInt(itr.nextToken());
                     context.getCounter("nPerDocRed", file).increment(count);
                     if (!list.contains(file)){
                         list.add(file);
                     }
                 }
             }
            ListIterator<String> it = list.listIterator();
            String s = "";
            while(it.hasNext()){
                String str = it.next();
                String fileFreq = str + "#" +
String.valueOf(context.getCounter("nPerDocRed", str).getValue());
                if (s.equals("")) {
                    s = fileFreq;
                } else {
                s = s + "," + fileFreq;
                }
            }
            Text result = new Text(s);
            context.write(key, result);
        }
}
```

The result is satisfying :

```
aachen   pg3200.txt#1
aar      pg3200.txt#4
aaron    pg3200.txt#7,pg100.txt#97
aart     pg3200.txt#8
aartist  pg3200.txt#9
aback    pg3200.txt#10
abaft    pg3200.txt#17
abaissiez      pg100.txt#98
abana    pg3200.txt#23
abandon  pg3200.txt#38,pg100.txt#108
abandoned      pg100.txt#110,pg31100.txt#5,pg3200.txt#87
abandoning     pg3200.txt#88
abandonment    pg3200.txt#91
```

Conclusion

This assignment was a great first glance into Hadoop. We saw the potential of managing high data load, but also the difficulty to implement such programs : it takes a lot of time to master Java, where implementing such an algorithm in Python for example would only require a few lines.

We also see the importance of correctly configuring our job, as it can have quite an impact on execution time.

It was however very tough to have to do all this without having ever learnt anything of Java, and with only very little notions of Hadoop. Searching for everything on the web and self-learning allows one to overcome this, but proves to be very time-consuming.

Sources

1. Countless searches on www.stackoverflow.com and www.google.com
2. Tom White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly Media, March 2015
3. Hadoop documentation : http://hadoop.apache.org/docs/current/