Gopalakrisna Jeremy

Homework 2: Massive Data Processing

Table des matières

I. Pre-processing

I.1 Initialisation

To get the global frequency of each word, I ran wordcount from the previous assignment on pg100.txt. (wordcount.txt in input folder on git)

For the rest of this assignment I used only an abstract of the file, from the beginning until the end of the 78th sonnet. This is because there are more than 120,000 lines in the document which would result in more than 60billions comparisons for SimilarityA which my computer cannot handle.

I.2 Remove all stopwords (you can use the stopwords file of your previous assignment), special characters (keep only [a-z],[A-Z] and [0-9]) and keep each unique word only once per line. Don't keep empty lines.

For this question, in my mapper I used similar code than in the first homework.

To remove stopwords, I used Scanner to read the file and add each word to a list:

```
Scanner s = new Scanner(new File("/home/cloudera/workspace/stopwords.txt"));
        ArrayList<String> list = new ArrayList<String>();
        while (s.hasNext()){
            list.add(s.next());
        }
        s.close();
```

For special characters, I used the StringTokenizer to split the input value (line) according to the following list:

```
" \t\n\r\f,.:;?![]'#--()_\"*/$%&<>+=@"
```

I then check that the words are not in the stopword list before adding them to my mapper output:

```
while (tokenizer.hasMoreTokens()) {
               String stri = tokenizer.nextToken();
                    if (list.contains(stri)) {
                    } else {
                    context.write(key, new Text(stri));
```

Finally, my mapper's output is composed of:

- Key: Byte offset of beginning of the line
- Value: a single word that is on that line

To keep words only once per line I put them in an Hashset in my reducer therefore removing duplicates.

I.3 Store on HDFS the number of output records (i.e., total lines).

To do this part, I implemented a custom counter:

```
public static enum COUNTER {
         LINES,
       };
```

I increment the counter at the end of the reducer :

```
context.getCounter(COUNTER.LINES).increment(1);
```

To save the output of this counter, I add the following lines to the driver:

```
File file = new File("/home/cloudera/workspace/lines_output.txt");
      file.createNewFile();
      BufferedWriter writer = new BufferedWriter(new FileWriter(file));
      long lines = job.getCounters().findCounter(COUNTER.LINES).getValue();
writer.write(Long.toString(lines));
      writer.flush();
      writer.close();
```

I create a new file then, thanks to BufferedWriter I write the value of the counter to the file before flushing and closing the writer.

I.4 Order the tokens of each line in ascending order of global frequency.

In the reducer, I received unique words of each line and put them into a list.

To be able to order them by ascending order, I created a Hashmap<String,String>. Reading wordcount.txt, for each word, I output the word and its frequency into my Hashmap. Then I decided to copy this information into a List of pairs to be able to use Collections.sort to order them by frequency:

```
Collections.sort(words, new Comparator<Pair<String, Integer>>() {
                        @Override
                        public int compare(final Pair<String, Integer> o1, final
Pair<String, Integer> o2) {
                                    return
Integer.compare(o1.getSecond(),o2.getSecond());

                        }
                    });
```

Finally, my reducer's output is composed of:

- Key: the line number (using the custom counter created and adding 1 because we want to start at one and not zero while keeping the counter as the last action)

```
long nbline = context.getCounter(COUNTER.LINES).getValue()+1;
```

- Value: each unique word of the line ordered by global frequency and separated by a space

II. Set-similarity joins (a)

II.1 The map method should emit, for each document, the document id along with one other document id as a key (one such pair for each other document in the corpus) and the document's content as a value.

In my mapper, I first started by splitting the input into key/value:

```
String[] parts = value.toString().split(",");
               String key1 = parts[0];
               String value1 = parts[1];
```

From this, I already got the value output of the mapper.

For the key output, I created a loop which, for the actual key, returns the key and an integer corresponding to one of the lines after it until the last line which I get from the counter in the pre-processing:

```
// get the value from the counter in preprocessing (number of lines in input)
        Scanner scanner = new Scanner(new
File("/home/cloudera/workspace/lines_output.txt"));
        int nblines = scanner.nextInt();
        scanner.close();
```

Therefore the first key(1) gets matched with all the lines from 2 to the last line, the second(2) with all the lines from 3 to the last one etc. In each pair outputted the left value is smaller than the right one therefore avoiding to have duplicates. Thus, the total number of pairs outputted equals $(n^2-n)/2$ where n is the total number of lines:

```
if (Integer.parseInt(key1)<nblines){
               for (int i = Integer.parseInt(key1)+1; i<=nblines; i++){
             StringBuilder Stringbuilder = new StringBuilder();
             Stringbuilder.append(key1 +","+ i);
```

Finally, my mapper's output is composed of:

- Key: pair of line number separated by a comma
- Value: unique words of the left-keyth line.

II.2 In the reduce phase, perform the Jaccard computations for all/some selected pairs.

II.2.1 Jaccard Method

I implement a Jaccard method which takes two strings as input and returns a double.

The first step is to split the two strings to get the individual words and put them into lists. Then I create a Hashset to get each words that appear but no duplicates.

The total number of words counting duplicates is the sum of the size of the lists.

The total number of words not counting duplicates is the size of the Hashset i.e. the union. The difference between the two is the number of words that appear more than once. As one string has no duplicates, this is the number of words that appear in both documents i.e. the intersection. Dividing the intersection by the union gives us the Jaccard Similarity:

```
public final double Jaccardsim (String s1, String s2){
List<String> list1 = Arrays.asList(s1.split(" "));        List<String> list2 =
Arrays.asList(s2.split("
             HashSet<String> sim = new HashSet<>(list1);
      sim.addAll(list2);
             double total = list1.size()+list2.size();
             double union = sim.size();
             double inter = total - union;
return inter/union;
```

II.2.2 Reducer's steps

After receiving the mapper output, I split the pair of keys. I then used the pre-processing output that I previously put into a Hashmap (to avoid computing the Hashmap at every reducer's run) to get the value associated with the key:

```
public static HashMap<String, String> map = new HashMap<String, String>();{
                   BufferedReader reader;
```

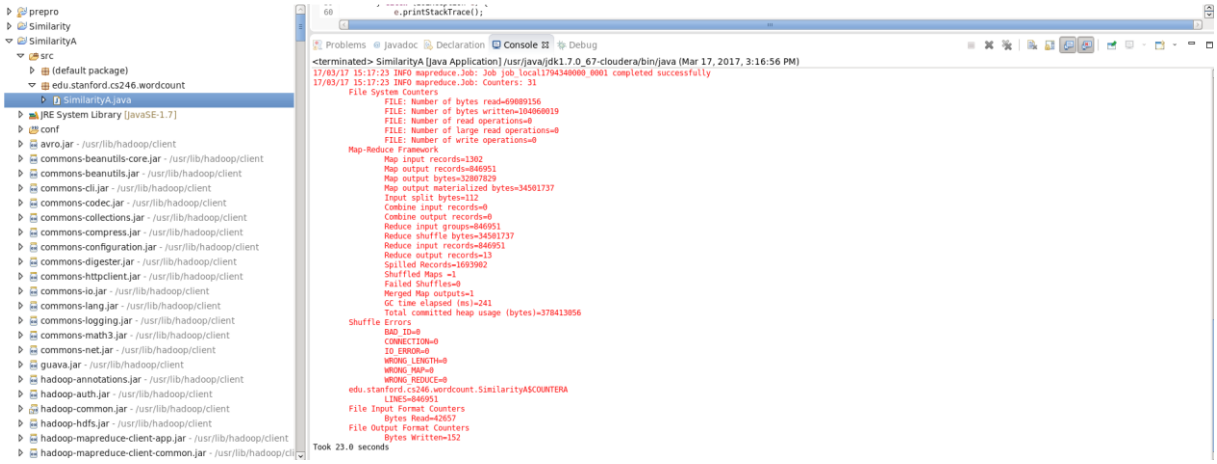
```
                    try {
                          reader =new BufferedReader(new
FileReader("/home/cloudera/workspace/output_prepro.txt"));
                           String line = "";
                                while ((line = reader.readLine()) != null) {
                                      String[] parts1 = line.split(","); //split
the line into key and value
                                      map.put(parts1[0], parts1[1]);
                                }
                                reader.close();
                    } catch (IOException e) {
                          e.printStackTrace();
                    }
                    }


String file1 = map.get(key2.toString());
String file = map.get(key1.toString());
```

I then input the two strings obtained into the Jaccard method.
As each pair of keys is a comparison, we have the same number of comparisons as the number of mapper outputs.
Finally, my reducer's output is composed of:

- Key: pair of line number separated by a comma
- Value: Jaccard Similarity (if it is superior to 0.8)

II.3 Report the execution time and the number of performed comparisons.



I implemented the same counter as in the Pre-processing code. As stated earlier, the number of comparison is equal to $(n^2-n)/2$ where n is the number of lines. Here we have 1302 lines in total so 846951 comparisons (same number as obtained by the counter, available in NBComparisonA.txt).

For the running time, my computer froze when I tried to run it in the console therefore I outputted the time elapsed when running it in Eclipse. I used the following lines in my driver:

```
long start = System.nanoTime();
long end = System.nanoTime();
long elapsedTime = end - start;
double seconds = elapsedTime/1000000000
System.out.println("Took " + seconds + "seconds");
```

The execution time was 23 seconds.

III. Set-similarity joins (b)

III.1 Create an inverted index, only for the first |d| - ⌈t |d|⌉ + 1 words of each document d (remember that they are stored in ascending order of frequency).

I start the same way as in SimilarityA, by splitting the input file into key and value.

I then split the value obtained into individual words and put them into a list. I compute an integer for the number of words to keep:

int keep = (int) Math.round((list.size() - (list.size()*0.8) + 1));
I then create a loop to keep only the required number of words starting from the last one in the list as words are stored in ascending order, therefore keeping the ones with higher frequency:

```
for (int i=list.size();i>list.size()-keep;i--){
context.write(new Text(list.get(i-1)),new Text(key1.toString()));
        }
```

Finally, my mapper's output is composed of:

- Key: a word to keep
- Value: the current line read by the mapper

III.2 Compute the similarity of the document pairs.

To have an output format similar to the one of SimilarityA, I get the values into a list reordering them so that in my pairs of keys, the left one is always smaller than the right one:

```
ArrayList<Integer> list = new ArrayList<Integer>();
                    for (Text value : values) {
                          list.add(Integer.parseInt(value.toString()));
                    }
                    Collections.sort(list);
```

Then I created the pairs of keys by outputting the values of the list above:

```
List<Pair<String, String>> words = new ArrayList<Pair<String, String>>();
      for (int val=0;val<list.size()-1;val++){
            for (int val1=val+1;val1<=list.size()-1;val1++){
                  String key1 = list.get(val).toString();
                  String key2 = list.get(val1).toString();
                  words.add(new Pair<String, String>(key1,key2));
                        }
                  }
```

As in SimilarityA, I then compare the strings of each pair and output the Jaccard Similarity.

One issue is that as I compare lines based on individual words, if two lines have more than one word in common, the similarity will be computed each time they have a similar word. If two input keys (the words) have the same values (line number), the similarity will be outputted for the two keys meaning we will have duplicates. To avoid that, In the beginning of my Reduce class I introduced an empty HashMap in which I add the pairs of keys for each output. I then check each time if the paie of key is already in the HashMap or not before computing the similarity:

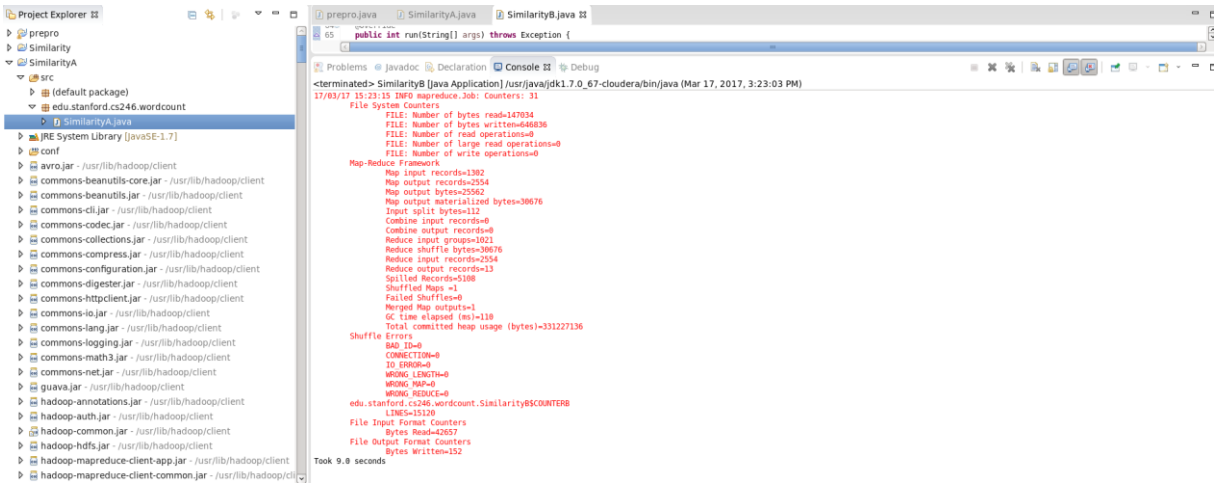
```
for (int i=0;i<=words.size()-1;i++){
if (uniquepair.containsKey(new
Text(words.get(i).getFirst()+","+words.get(i).getSecond()))){
                        }
      else{
            String file1 = map.get(words.get(i).getFirst());
            String file2 = map.get(words.get(i).getSecond());
            double sim = Jaccardsim(file1,file2);
            if (sim>0.8){
                  context.write(new
Text(words.get(i).getFirst()+","+words.get(i).getSecond()),sim);
                                uniquepair.put(new
Text(words.get(i).getFirst()+","+words.get(i).getSecond()),sim);
                                }
```

Finally, my reducer's output is composed of:

- Key: pair of line number separated by a comma
- Jaccard Similarity (if it is superior to 0.8)

III.3 Report the execution time and the number of performed comparisons

For SimilarityB the number of comparisons I had was 15120 (available in NBComparisonB.txt). The execution time was 9 seconds.



IV. Explain and justify the difference between a) and b) in the number of performed comparisons, as well as their difference in execution time.

First of all, in the mapper, for SimilarityA, we have an output for each pair, which means one for each comparison we are going to make. As we output each pair only once, there is no real reducer job in the sense of putting together values with the same key. Indeed, in our reducer we are going to have the same number of output than in our mapper and the reducer is going to run once for each pair individually. The complexity increases with each added line by the value of the total lines before adding it. For example, if we have 10 lines, adding another line adds 10 comparisons, then another line adds 11 comparisons etc.

In SimilarityB however, the number of outputs of both the mapper and the reducers are smaller. Indeed, in the mapper output, we have the number of different words that appear in the document. In the reducer, we have words associated with the different lines in which they appear which means we compare lines that have at least a word in common therefore making the process much faster as there are no useless comparisons that are going to result in a similarity of 0. To compare, for SimilarityB to perform as much comparisons as SimilarityA it would need every line to have a word in common with every other line. Furthermore, we keep only $|d| - \lceil t\,|d| \rceil + 1$ words for each line we further reduce the number of comparisons while still making sure that similar documents get compared.

In conclusion we could say that SimilarityA has no pre-processing part to it and just performs brute force comparison to find similarities while SimilarityB performs a pre-processing to avoid useless comparisons of non-similar documents which is the reason why SimilarityA takes twice the time to execute.

Sources:

http://stackoverflow.com/questions/4777622/creating-a-list-of-pairs-in-java

http://stackoverflow.com/questions/109383/sort-a-mapkey-value-by-values-java

http://www.programcreek.com/2013/03/java-sort-map-by-value/

http://stackoverflow.com/questions/16939773/get-arraylistnamevaluepair-value-by-name

http://stackoverflow.com/questions/521171/a-java-collection-of-value-pairs-tuples

http://stackoverflow.com/questions/18395998/hadoop-map-reduce-secondary-sorting

http://www.bigdataspeak.com/2013/02/hadoop-how-to-do-secondary-sort-on_25.html

http://stackoverflow.com/questions/10455840/sorting-a-list-consisting-of-integer-pairs

http://stackoverflow.com/questions/29526643/counting-frequency-of-words-from-a-txt-file-in-java

http://stackoverflow.com/questions/10158793/sorting-words-in-order-of-frequency-least-to-greatest

http://stackoverflow.com/questions/8886103/read-from-a-text-file-into-a-hash-map-or-list

http://stackoverflow.com/questions/32038221/in-textinputformat-in-hadoop-mapreduce-what-is-byte-offset-and-how-key-is-as-by

http://stackoverflow.com/questions/7771303/getting-the-character-returned-by-read-in-bufferedreader

http://stackoverflow.com/questions/2340106/what-is-the-purpose-of-flush-in-java-streams

http://stackoverflow.com/questions/6349376/java-creating-new-file-how-do-i-specify-the-directory-with-a-method

http://stackoverflow.com/questions/5450290/accessing-a-mappers-counter-from-a-reducer

http://stackoverflow.com/questions/34171191/cannot-access-counter-in-the-reducer-class-of-mapreduce

https://diveintodata.org/2011/03/15/an-example-of-hadoop-mapreduce-counter/