Massive data processing
Homework 2
Gaétan Morand

Configuration

Mac OS 10.11
Hadoop 2.7.3 installed with Homebrew

Here is a typical list of commands that I use :

```
javac SetSim1.java -cp $(hadoop classpath)
mv *.class org/myorg
jar -cvf ss1.jar org/myorg
hdfs dfs -rmr output
hadoop jar ss1.jar org/myorg/SetSim1 input/preprocessing_output.txt output
```

Preprocessing

First I programmed a WordCount job which removes all special characters. The code is in the file “WordCount.java”.
The output is in the file “wordcount.csv”.

In order to remove special characters I split every line this way:

```
String[] words = line.split("[^A-Za-z0-9]");
```

I reuse the “stopwords.txt” file from homework 1.

Obviously it is important to use exactly the same code as in Wordcount.java to tokenize lines. To order the words by frequency, I import the wordcount.csv file as a dictionary and I programmed a loop that goes through each line to put the words in the ascending frequency order.

```
for(Text word : values) {

            String toInsert = word.toString();
            Integer index = new Integer(0);
            Integer frequency = wordcount.get(toInsert);
            Integer len = sortedLine.size();

            for (int i = 0; i < len; i++){
                Integer indexFrequency = wordcount.get(sortedLine.get(i));
                try
                {
                    if (frequency <= indexFrequency){
                    index++;
                }
                }
                catch (NullPointerException nullPointer)
                {
                   System.out.println(sortedLine.get(i));
                }
            }

            sortedLine.add(Math.min(index + 1, len), toInsert);
        }
```

For a reason that I was not able to point out in the due time, there are approximately 20 words that raise a NullPointerException, even though they are in the Wordcount.csv file. Since it only concerns 20 words, I decided not to investigate further.
The results are in the file “preprocessing_output.txt”. The code is in the file “Preprocessing.java”.

The custom counter writes the line count in a specific file: it is 115105.

Set-similarity join: naive method

First we need to implement a custom format reader to use the line number as a key in the map function. I found some documentation here:
https://developer.yahoo.com/hadoop/tutorial/module5.html#fileformat

I did not manage to have it work, so I will just drop the line numbers that are in the file and use the actual line numbers. In practice they are the same.

In the map function we limit the number of lines arbitrarily to reduce calculation time.

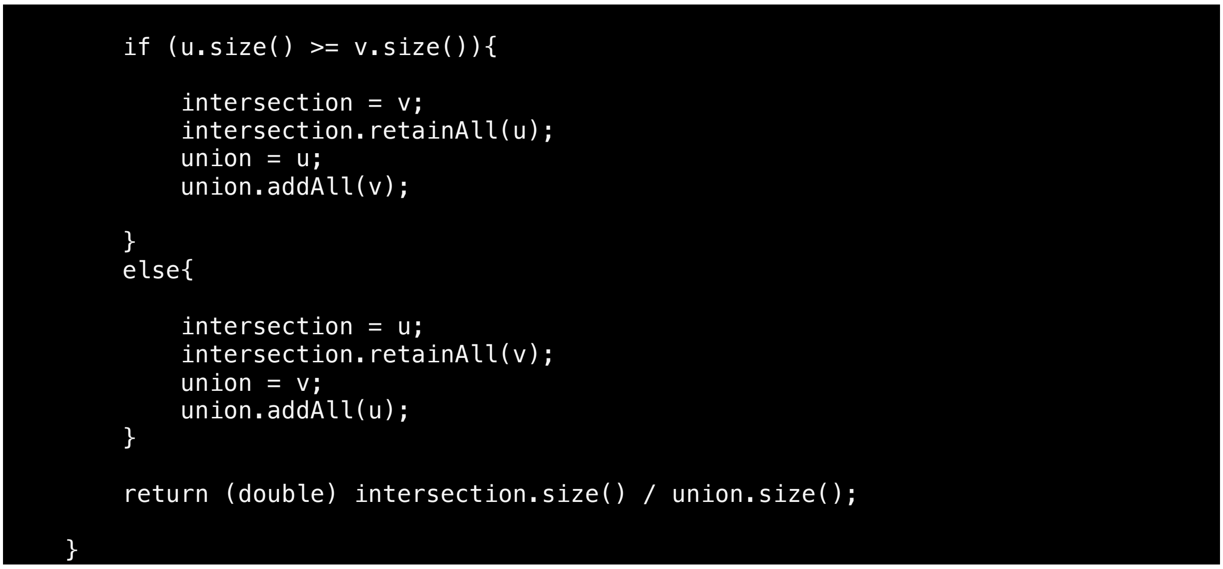
```
lineCount = Math.min(Integer.parseInt(line_count[1]), Nmax);
```

Then we go through all the keys, and we associate every other key that is possible. The new key will be a list of the two document IDs, and the new value will be the words associated to the first key. When we are done, every couple will be sent twice to the reduce function, one with each line of words. It is important to store the keys in ascending order in the list so that the reduce function recognizes that they are the same.

```
for (int i=0; i < lineCount; i++){
            if( key != i){
                List<LongWritable> new_key = new List<LongWritable>(Math.min(key, 
i), Math.max(key,i));
                Text new_value = new Text(line[1]);
                context.write(new_key, new_value);
            }

        }
```

Then in the reduce function we just have to calculate the Jaccard score and output the couple if it is higher than the threshold.

```
private double jaccard(Text s1, Text s2){

        HashSet<String> u = new HashSet<String>(Arrays.asList(s1.toString().split(" 
")));
        HashSet<String> v = new HashSet<String>(Arrays.asList(s2.toString().split(" 
")));

        HashSet<String> intersection = new HashSet<String>();
        HashSet<String> union = new HashSet<String>();
```



```
        if (u.size() >= v.size()){

            intersection = v;
            intersection.retainAll(u);
            union = u;
            union.addAll(v);

        }
        else{

            intersection = u;
            intersection.retainAll(v);
            union = v;
            union.addAll(u);
        }

        return (double) intersection.size() / union.size();

    }
```

I ended up using only textual keys, as I was having too many problems taking care of the formatting.

I did not manage to fully finish the program. It is frustrating because it runs but there are still some small problems that prevent the program's right behavior.

Set-similarity join: advanced method

I did not have enough time to even start this part of the project.

Analysis

In the first part we perform all the comparisons, so the complexity is always N^2, where N is the number of lines.
In the second part, we manage to do less comparisons, by not doing useless ones. Indeed, just comparing a subset of words can be enough to know for sure that two lines are not that similar.
By noticing this, we save a lot of processing power, as well as temporary storage since we don't have to store all the data in memory between the map and the reduce.