Massive Data Processing

Assignment 2

Flavie VAMPOUILLE
MSc in Data Sciences & Business Analytics
Centrale Supélec & ESSEC Business School

Pre-processing the input

The input of this work is the pg100.txt file available on:
http://www.gutenberg.org/cache/epub/100/pg100.txt

As ask in the subject I keep [a-z], [A-Z] and [0-9] and remove all others special characters. I also change all upper case to lower case as I don't want words like "then" and "Then" to be considered as different.

To find the stop words I first look at the words appearing more than 4 000 times like in the first assignment. My output was of length 32. As I think that there should be more words to remove in English and as we only have one text instead of three in the previous project I try lower values for the number of appearances that characterize stop words. With 2000 I remove words like "king" or "lord", so this value is too small. I then test 3000 but it removes words like "lord" too. Hence I keep the value of the first assignment of 4000.

I first create a WordCount.txt file in which I store the word count of pg100.txt in the form:

```
Word1 number1
Word2 number2
…
```

Then I write a java code which will:

- Remove special characters (keep only [a-z],[A-Z] and [0-9]) and put everything in lower case;
- Remove all stop words;
- Keep each unique word only once per line.
- Don't keep empty lines;
- Order the tokens of each line in ascending order of global frequency;
- Store on HDFS the number of output records (i.e., total lines).

In my mapper class I first import the WordCount.txt file I previously create and store it in a HashMap

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

    // HashMap of WordCount
    public static HashMap<String,Integer> WordCount = new HashMap<String, Integer>();
    public static void importWordCount () throws IOException, InterruptedException {
        BufferedReader BR = new BufferedReader(new FileReader(
                new File("/home/cloudera/workspace/Assignment2/WordCount/WordCount.txt")));
        String line1;
        while((line1 = BR.readLine()) != null){
            String[] arr = line1.split(" ");
            String word = new String(arr[0]);
            Integer count = new Integer(Integer.valueOf(arr[1]));
            WordCount.put(word,count);
        }
    }
    static {
        try {
            importWordCount();
        }
        catch (IOException|InterruptedException e) { throw new ExceptionInInitializerError(e); }
    }
```

I also create a count for the lines to keep the “ID” of the lines as they are the ID of the documents

```java
Integer line_count = 0;
```

Then I read the input text line by line, remove all undesirable characters and remove the stop words (words that appear more than 4 000 times in the whole text)

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

    // pre-process the input
    for (String line : value.toString().toLowerCase().replaceAll("[^a-zA-Z0-9\\s]", "").split("[\\r\\n]+")) {
        String[] words_in_line = line.split(" ");
        // Map of words in line with their global frequency, non sorted
        // remove stop words and words not in WordCount
        HashSet<String> map = new HashSet<String>();
        for (int i = 0; i < words_in_line.length; i++) {
            if (WordCount.containsKey(words_in_line[i])) {
                if (WordCount.get(words_in_line[i]) < 4000 ) {
                    map.add(words_in_line[i]);
                }
            }
        }
```

For each of the lines hence obtain, I ordered the set of words it contains by ascending order of global frequency and write the result (if the line in non-empty) in the mapper output with

- Key = ID of the line
- Value = words in the line by ascending order of global frequency

```
// create the comparator for global frequency
Comparator<String> compare_frequency = new Comparator<String>() {
    public int compare (String word1, String word2) {
        return WordCount.get(word1) - WordCount.get(word2);
    }
};
// create the new output
if (!map.isEmpty()) {
    line_count++;
    List<String> list = new ArrayList<String>(map);
    Collections.sort(list, compare_frequency);
    StringBuilder SB = new StringBuilder();
    String separator = " ";
    for (String word : list) {
        SB.append(word);
        SB.append(separator);
    }
    // write line ID and line content's
    context.write(new Text(String.valueOf(line_count)),new Text(SB.toString()));
}
```

As the goal is to compare similarity in the lines I don't use a reducer. If a line appears several time (identical map output are identical documents) I don't want to delete some but to keep each line one time.

I finally store the results in TextOutputFormat, with separator ";" between the number of the line and its mapped content.

```
job.getConfiguration().set("mapreduce.output.textoutputformat.separator", ";");
```

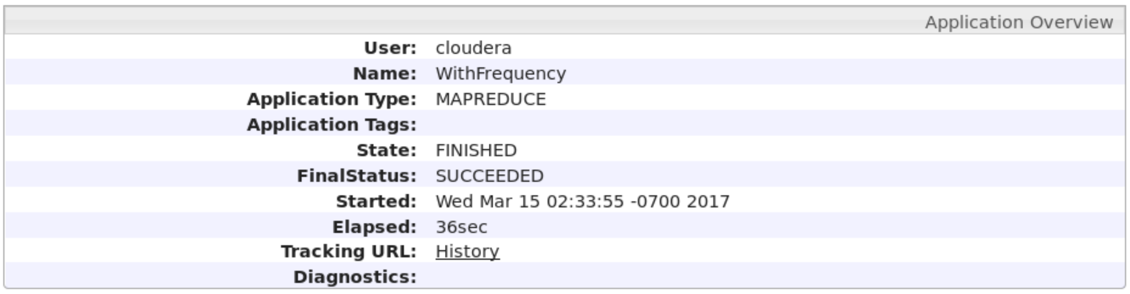
The Reduce output records (which is the same as the map output records as I don't use a reducer) is 114 984.

The final output is of the form:

```
1;ebook complete gutenberg shakespeare works project william
2;shakespeare william
3;anyone anywhere ebook cost use at no
4;restrictions whatsoever copy almost away give may or no
5;reuse included license terms gutenberg under project
6;wwwgutenbergorg online ebook at or
7;details copyrighted ebook below gutenberg project
8;guidelines file copyright follow please
9;title complete shakespeare works william
10;author shakespeare william
```

The code is in “java code / Pre-processing the data” and the output file is in the “Pre-process input” folder. The code is named “WithFrequency.java” and the output “PreProcessInput.txt” (or PreProcessPart1000.txt” for the first 1 000 lines alone).

The execution time is 36 seconds



Application Overview

User:	cloudera
Name:	WithFrequency
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Wed Mar 15 02:33:55 -0700 2017
Elapsed:	36sec
Tracking URL:	History
Diagnostics:	

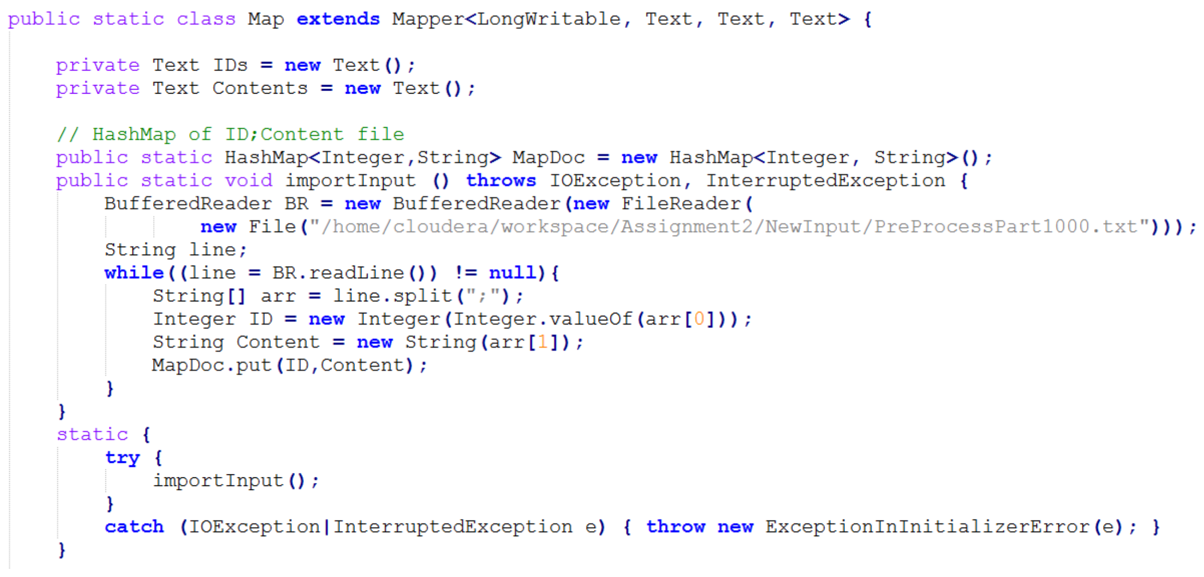
Set-similarity joins

Task a. Naïve approach

We want to perform all pair-wise comparison between documents. To do that we will need two steps:

- A mapper that will associate each document by pair (we want them unique) with a key output of the form id1/id2 and an associate output value of the form content1/content2;
- A reducer that will calculate for each of these pairs the similarity between the associated contents and return a key output of the form id1/id2 associated with a similarity value s of the two contents. As we are only interested in documents that are similar, we add a filter on the reducer to write only in the final output the id pairs that have a similarity above a threshold value that is given as 0.8 for this work.

In the mapper, I first create a HashMap with the (ID,Content) that were create in the Pre-process part

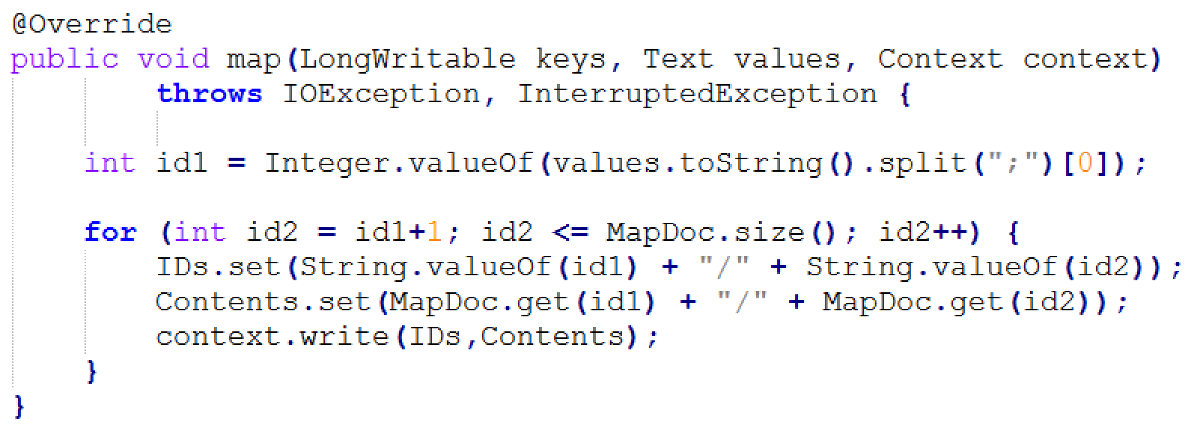


```java
public static class Map extends Mapper<LongWritable, Text, Text, Text> {

    private Text IDs = new Text();
    private Text Contents = new Text();

    // HashMap of ID;Content file
    public static HashMap<Integer,String> MapDoc = new HashMap<Integer, String>();
    public static void importInput () throws IOException, InterruptedException {
        BufferedReader BR = new BufferedReader(new FileReader(
                new File("/home/cloudera/workspace/Assignment2/NewInput/PreProcessPart1000.txt")));
        String line;
        while((line = BR.readLine()) != null){
            String[] arr = line.split(";");
            Integer ID = new Integer(Integer.valueOf(arr[0]));
            String Content = new String(arr[1]);
            MapDoc.put(ID,Content);
        }
    }
    static {
        try {
            importInput();
        }
        catch (IOException|InterruptedException e) { throw new ExceptionInInitializerError(e); }
    }
```

Then I iterate through all the IDs two times, keeping id2 > id1, and write as output of the mapper:

- Key output : id1/id2
- Value output : content1/content2

As I only take id2 > id1, this output will contain each pair a single time.



```java
@Override
public void map(LongWritable keys, Text values, Context context)
        throws IOException, InterruptedException {

    int id1 = Integer.valueOf(values.toString().split(";")[0]);

    for (int id2 = id1+1; id2 <= MapDoc.size(); id2++) {
        IDs.set(String.valueOf(id1) + "/" + String.valueOf(id2));
        Contents.set(MapDoc.get(id1) + "/" + MapDoc.get(id2));
        context.write(IDs,Contents);
    }
}
```

Now that all pairs are created only once, I write a reducer that will perform the Jaccard computations for the similarity between these pairs (cardinal of the intersection of the compared contents divided by the cardinal of their union). To achieve that goal, I

- iterate through the pairs created with the mapper
- transform each one of document's content in the pair in a HashSet of its words
- create a set for the union of the two above sets using ".addAll" and a set for their intersection using ".retainAll".
- calculate the similarity wich is $\frac{size\ intersection}{size\ union}$

- write as a key output of the reducer the ID “id1/id2” of the two documents of the pair and as a value output the similarity between these documents.

I include a counter that returns the number of performed comparison. It was not essential as this number is also given by “Reduce input groups” in the Map-Reduce Framework, but it will be useful in task b, with the inverted index prefix.

```java
public static enum Number {
    PerformedComparison,
};

public static class Reduce extends Reducer<Text, Text, Text, Text> {

    private Text SimilarityOutput = new Text();

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

        String Docs = values.iterator().next().toString();
        String[] splitted = Docs.split("/");

        context.getCounter(Number.PerformedComparison).increment(1);

        List<String> list1 = Arrays.asList(splitted[0].toString().split(" "));
        HashSet<String> set1 = new HashSet<String>(list1);

        List<String> list2 = Arrays.asList(splitted[1].toString().split(" "));
        HashSet<String> set2 = new HashSet<String>(list2);

        HashSet<String> union = new HashSet<String>(set1);
        union.addAll(set2);

        HashSet<String> intersection = new HashSet<String>(set1);
        intersection.retainAll(set2);

        float sim = (float) intersection.size() / union.size();

        if (sim >= 0.8) {
            SimilarityOutput.set(String.valueOf(sim));
            context.write(keys,SimilarityOutput);
        }
    }
}
```

I run a test on the first 1 000 lines of the pre-process input and obtain:

- Number of Performed Comparisons = 499 500 (or again Reduce input groups = 499 500)
- Reduce output records = 14

Hence 499 500 pairs have been compared and only 14 have a similarity above the threshold of 0.8.

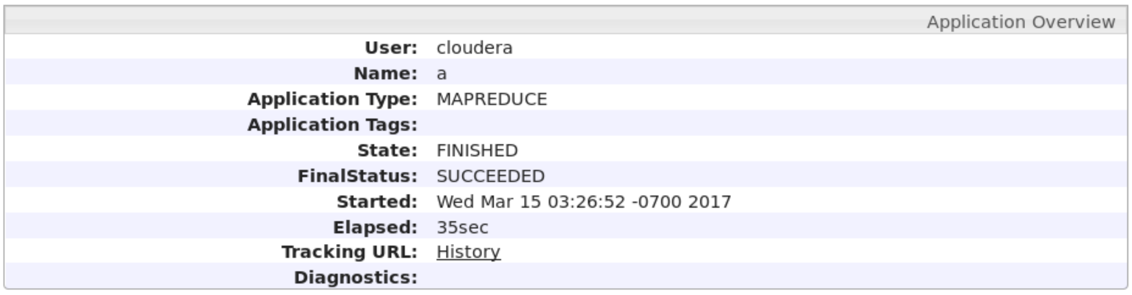
The output is of the form

```
2/132 : 1.0
22/122 : 1.0
22/38 : 1.0
…
```

I then check with the pre-process input for some lines and it appears that the ones with similarity 1 are the same:

2: shakespeare william / 132: shakespeare William
22: version complete works william electronic / 122: version complete works william electronic
…

The execution time with 1 000 lines is 35 seconds



Application Overview

User:	cloudera
Name:	a
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Wed Mar 15 03:26:52 -0700 2017
Elapsed:	35sec
Tracking URL:	History
Diagnostics:	

<u>Note</u>: When I try to run my code in HDFS the first time, the task failed due to the fact that there was not enough memory allow, even with 1 000 lines only. Thus, to be able to run it, I add the following lines in my code

```
job.getConfiguration().set("mapreduce.map.memory.mb", "4096");
job.getConfiguration().set("mapreduce.map.java.opts", "-Xmx3500m");
```

I then try with the whole pre-process input but my computer crash out of memory. So I will not be able to perform it with my restricted means.

However as the number of operations must be quite huge, I take interest in finding it. Considering that my code considered all unique pairs in the input set, this means that if the set is of length n, the number of pairs will be $\frac{n(n-1)}{2}$. Thus for

- 1 000 lines : number of pairs = 499 500 (as we find above)
- 114 984 lines (total number) : number of pairs = 6 610 602 636 ! Which explains why my computer crash out of memory.

The code for this question in in the “java code / Set similarity joins” folder and is named “a.java”. The output is in the “Similarities output” folder and is named “a similarity - 1000 first lines.txt”.

Task b. With prefix filtering principle

For this task, we first want to create an inverted index, for the first |d| - ceil(t . |d|) +1 words in each document d. Hence the mapper will output:

- Words with low frequency as keys
- The whole input value as value : “id ; content” of each document containing the key word (if the position of the key word is before the limit)

As the value of the similarity threshold will be used both in the mapper (to calculate the number of words to keep in the inverted index) and in the reducer (to filtrate the documents pairs to write in the final output) I store its value in a public float variable.

```java
public static Float SimilarityThreshold = 0.8f;

public static class Map extends Mapper<LongWritable, Text, Text, Text> {

    private Text Index = new Text();

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

        String line = value.toString().split(";")[1];
        String[] WordsLine = line.toString().split(" ");

        int KeepWords = (int) (WordsLine.length - Math.ceil(SimilarityThreshold * WordsLine.length) + 1);

        for (int i = 0; i < KeepWords; i++) {
            Index.set(WordsLine[i]);
            context.write(Index, value);
        }
    }
}
```

Once again I create a counter to know the number of perform comparisons, and it will be more useful this time as this value will no more be in the map-reduce framework.

```java
public static enum Number {
    PerformedComparison,
};
```

With the reducer I want to compute the Jaccard similarity between documents that share the same less frequent words (the prefix filtering principle states that with a well choose number of words we will not miss any similar documents).

The first step is to create a list of IDs and Contents for each for each key word created in the mapper phase. To create the IDs and Contents lists in the reducer will put them back empty at each new element of “values” (remind that each value contain id1/id2;Content1/Content2 elements associated with the inverted index word).

```java
public static class Reduce extends Reducer<Text, Text, Text, Text> {

    private Text outputKey = new Text();
    private Text outputValue = new Text();
    private HashSet<String> duplicate = new HashSet<String>();

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

        List<Integer> IDs = new ArrayList<Integer>();
        List<String> Contents = new ArrayList<String>();

        for (Text val : values) {
            IDs.add(Integer.valueOf(val.toString().split(";")[0]));
            Contents.add(val.toString().split(";")[1]);
        }
```

However we need to be careful because this time we don’t have unicity of the pair-wise comparisons. If an id/content document has several prefix it will be used for comparison several time.

An example is that 39/23 : 1.0 could appear several times in the output because the contents of these documents are for both: “19901993 inc library copyright shakespeare world”. As there is six words and the similarity threshold is 0.8

6 - ceil (0.8 * 6) + 1 = 2.2

Hence we will keep two words in the inverted index and both “19901993” and “inc” will be used as key in the mapper, and documents 39 and 23 similarity will be computed two times. So, I add a filter on the IDs pair (HashSet “duplicate” in the code below).

```java
if (IDs.size() > 1) {
    for (int i = 0; i < IDs.size()-1; i++) {
        for (int j = i+1; j <= IDs.size()-1; j++) {

            Integer id1 = IDs.get(i);
            Integer id2 = IDs.get(j);
            String id = String.valueOf(id1) + "/" + String.valueOf(id2);
            String id_reverse = String.valueOf(id2) + "/" + String.valueOf(id1);

            if (!(duplicate.contains(id))) {

                duplicate.add(id);
                duplicate.add(id_reverse);

                context.getCounter(Number.PerformedComparison).increment(1);
```

Once I obtain unicity of the compared pairs of documents, I compute the Jaccard similarity for each unique pair (associated to the map key word) in the same way as with the reducer of the naïve method.

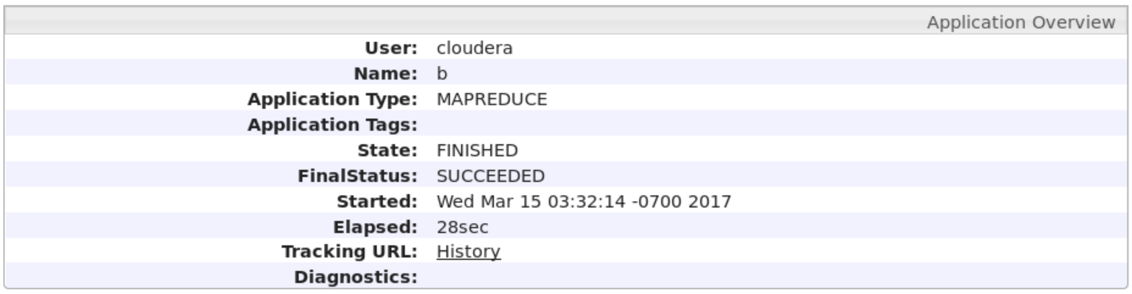
```
                    String content1 = Contents.get(i);
                    List<String> list1 = Arrays.asList(content1.toString().split(" "));
                    HashSet<String> set1 = new HashSet<String>(list1);

                    String content2 = Contents.get(j);
                    List<String> list2 = Arrays.asList(content2.toString().split(" "));
                    HashSet<String> set2 = new HashSet<String>(list2);

                    HashSet<String> union = new HashSet<String>(set1);
                    union.addAll(set2);

                    HashSet<String> intersection = new HashSet<String>(set1);
                    intersection.retainAll(set2);

                    float sim = (float) intersection.size() / union.size();

                    if (sim >= SimilarityThreshold) {
                        outputKey.set(id);
                        outputValue.set(String.valueOf(sim));
                        context.write(outputKey, outputValue);
                    }
                }
            }
        }
}
```

I run a test on the first 1 000 lines of the pre-process input and obtain:

- Number of Performed Comparisons = 578
- Reduce output records = 14

Hence this method does less comparison than the naïve one, is really faster and produce the same output.

The execution time is 28 seconds



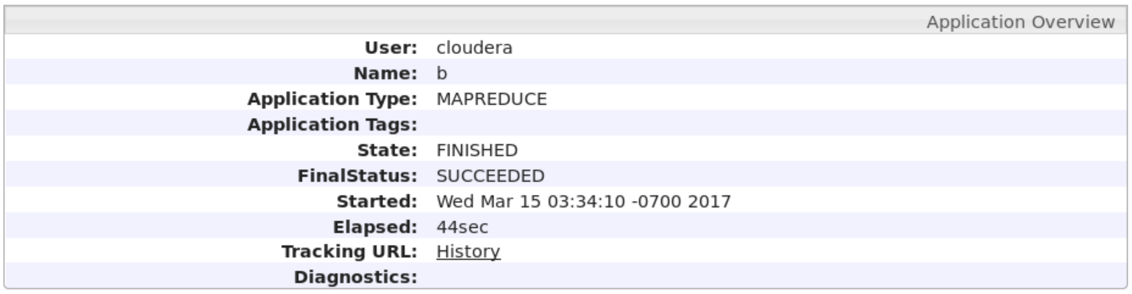
Application Overview

User:	cloudera
Name:	b
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Wed Mar 15 03:32:14 -0700 2017
Elapsed:	28sec
Tracking URL:	History
Diagnostics:	

I then run it on the whole input and it works fine. I obtain

- Number of Performed Comparisons = 2 721 392
- Reduce output records = 231 227

The execution time is 44 seconds



```
                                                   Application Overview
               User:  cloudera
               Name:  b
   Application Type:  MAPREDUCE
   Application Tags:
              State:  FINISHED
        FinalStatus:  SUCCEEDED
            Started:  Wed Mar 15 03:34:10 -0700 2017
            Elapsed:  44sec
       Tracking URL:  History
        Diagnostics:
```

The code for this question in in the “java code / Set similarity joins” folder and is named “b.java”. The outputs are in the “Similarities output” folder and are named “b similarity - 1000 first lines.txt” and “b similarity”.

Task c. Explain differences between a and b in the number of performed comparison and in the execution time

In task a., the number of output for the mapper is:

$$\frac{n(n-1)}{2}$$

where n is the number of documents (here the number of non-empty lines after pre-processing pg100.txt). The reducer then performs comparisons for each one of these pairs and the number of performed comparisons grows extremely fast:

- For 1 000 lines we have: number of performed comparison = 499 500 (~ $5.\,10^5$)
- For 114 984 lines (total number) we have: number of performed comparisons = 6 610 602 636 (~ $6.\,10^9$)

If the execution time is pretty fast with 1 000 lines (35 seconds) which represent a little less than $5.\,10^5$ performed comparison, for the whole input my computer run for about thirty minutes before crashing out of memory. Considering the execution time mainly depends on the number of performed comparisons and seeing that this number is about 6.10^9 for only 114 984 lines (which is really not a big deal compare to the number of document we can find on the web), comparing all pair-wise documents is rapidly no more an option to study similarities in a corpus of documents.

In Task b. we add a filter on the documents for which we want to compute similarity. The *prefix-filtering principle* states that for a well choose number Nd (depending of document d), comparing only documents which share the firsts Nd words with the lower frequency will allow us to find all the documents that are similar (have a similarity above a certain threshold).

If the threshold has value t, then [Chaudhuri et al. 2006] proves that it is adequate to fix

Nd = |d| - ceil (t . |d|) + 1

It means that indexing only the first Nd words of each document d, we will not miss any similar documents. The interest of doing so is that it will drastically decrease the number of performed comparisons.

In our case, most sentences are below ten words (counting very large) and the similarity threshold is 0.8. Hence we will keep 10 – 8 + 1 = 3 words maximum in the inverted index for each document. If the total number of documents is n, we will have a maximum of 3^n words in the inverted index (which far lower than the $\frac{n(n-1)}{2}$ of the mapper in task a.).

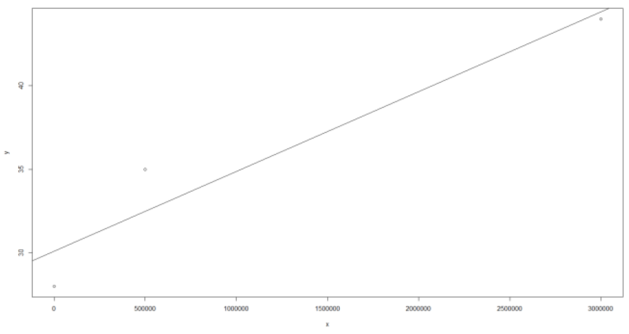
Then the similarity is computed for pairs of documents that share a key word, but only once (whether they share one or several key words). I will look through two extreme cases: if all documents are the same and if all the documents are different.

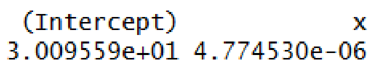
- If all documents are the same then there is only 3 words in the inverted index and all pair-wise comparisons are performed. Thus it is worse than with the previous case as we first look at all the documents to create the inverted index and then compute all pair-wise comparisons anyway.
- If all documents are strictly different (which will rapidly be impossible due to the limited number of words, even if current English count not far than 600 000 words) then there will be zero performed comparisons.

Hopefully, both those cases will be quite rare and we can hope to lie somewhere between them.

	Similarity.a 1 000 lines	**Similarity.a 110 000 lines**	**Similarity.b 1 000 lines**	**Similarity.b 110 000 lines**
Number performed comparisons	5.10^5	6.10^9	5.10^2	3.10^6
Execution time	35	*crash*	28 s	44 s

Making the rough assumption that the execution time linearly depends in the number of performed comparisons and performing a simple regression in R on the (number of comparisons , execution time) pairs, we can see that for 10^9 comparisons the execution time would be really huge: about 5.10^3 seconds = 1,4 hours.





```
 (Intercept)            x 
3.009559e+01 4.774530e-06 
```

We can conclude that if doing a all pair-wise comparison is acceptable for a very small number of documents, it quickly becomes no more an option as this number increase. While creating an inverted index of the share words with lower frequency before computing the similarities will allow to reduce significantly the number of performed comparisons and thus the execution time, without missing any similar documents.