Massive Data Assignment 2

SAMOU Timothée

March 16, 2017

Abstract

This assignment is the second application of Hadoop Mapreduce framework. We are going to implement two set similarity joins methods with a preprocessing part.
Additional informations : MSc Data Sciences & BA student.

The Github can be found at this address where you will find the commit tree :
https://github.com/timothee001/BDPA_Assign2_TSAMOU/

Contents

1 Pre-processing the input

At the beginning I wanted to run two jobs to complete this task (taking a WordCount to get the frequency of each word). But I realized that it was inefficient and time computation was very bad, so I decided to start from scratch without using previous WordCount job code and compute frequency in the same job.

The preprocessing part has been well optimized and run in about 20 seconds for the whole document.

Figure 1: Preprocessing job run

1.1 Removing Stop words

We use part of code of previous assignment to do this. I imported my previous class 'ReadCSV' that return a ArrayList of String containing all words that have frequency superior to 4000 taken from the three texts corpus of the previous assignment :

Listing 1: Removing Stopwords

```
private ArrayList<String> stopwords = ReadCSV.getStopWords();

//in the mapper
if(!words.contains(token) && !stopwords.contains(token.toLowerCase())){
              context.write(new Text(token), key);
}
```

1.2 Storing on HDFS

In the cleanup method we added the following code, it just creates a file on HDFS and write our records count that we incremented in our job.

Listing 2: Storing on HDFS Code

```
try{
              Path pt=new Path("outputPreprocessingLines");
              FileSystem fs = FileSystem.get(new Configuration());
              BufferedWriter br=new BufferedWriter(new
                  OutputStreamWriter(fs.create(pt,true)));

              String line="Number of outputs records " +linesCount;
              br.write(line);
              br.close();
        }catch(Exception e){
               System.out.println("File not found");
        }
```

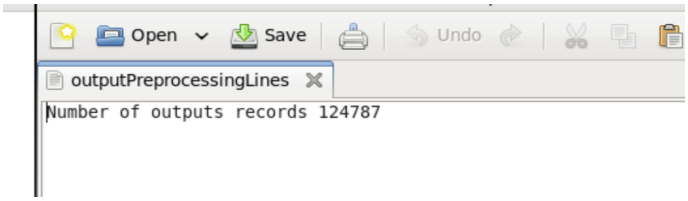


Figure 2: Storing on HDFS the number of output records

1.3 Ordering in ascending order

This is the tricky part, to order the word by frequency, I created a class WordDocs that implements the comparable interface with the mandatory compareTo method. If one object a has bigger frequency property than b a.compareTo(b) will return true, false otherwise.

Then, a key is associated with a list of corresponding docIds. To get the frequency, we return the size of this list. At the end, those WordDocs will be fully ranked in a TreeSet when we do the insertion and the search. This is way more convenient.

Listing 3: WordDocs class

```
public class WordDocs implements Comparable<WordDocs>{

   TreeSet<Long> tree;
   String key;

   public WordDocs(String key,TreeSet<Long> tree){
      this.key=key;
      this.tree=tree;
   }

   public String GetKey(){
      return this.key;
   }

   public int Size(){
      return this.tree.size();
   }

   public TreeSet<Long> GetTree(){
      return this.tree;
   }

   @Override
   public int compareTo(WordDocs o) {
      // TODO Auto-generated method stub
       if (this.Size() > o.Size()) {
              return 1;
          } else {
              return -1;
          }
   }

}
```

In the mapper, each key that corresponds to a word is emitted with his corresponding docId. We also created a Hashmap that store doc content : static HashMap <Long, String> docString = new HashMap <Long, String>(); with

Then in the mapper, I added this small call at the end of map function : docString.put(key.get(), "");

This hashmap will be filled in the cleanup function that is called once in the reducer :

Listing 4: Cleanup

```
protected void cleanup(Context ctxt) throws IOException,InterruptedException {
      //we call this fonction once at the end
        while (!wordDocs.isEmpty()) {

         WordDocs biggestWordDoc = wordDocs.pollFirst();
        // System.out.println(biggestWordDoc.Size());
         TreeSet<Long> docIds = biggestWordDoc.GetTree();

         for(int i = 0;i<docIds.size();i++){
            long docid = docIds.pollFirst();
           docString.put(docid, docString.get(docid) + " " + biggestWordDoc.key);
         }


        }

        for (Entry<Long, String> entry : docString.entrySet()) {
         Long docid = entry.getKey();
         String doccontent = entry.getValue();
         if (doccontent.matches(".*[a-zA-Z0-9]+.*"))
            ctxt.write(new Text(doccontent.trim()), new Text());
               linesCount++;
      }
```

As you can see, the WordDocs files that have been ordered by frequency are taken one by one starting from the lowest frequency at the left of the stack. And we increment every hashmap content that represent our documents. And we do a final loop to write them in the output file.

2 All pair-wise

2.1 Basic

This is the first basic implementation, I couldn't run several times this job on the whole document because of computation time. I run it once on the whole document and it took one hour ! The tests on this job are done on a smaller set of 3000 lines approximatively. We keep the same reduced doc on Inverted Index to perform comparisons.

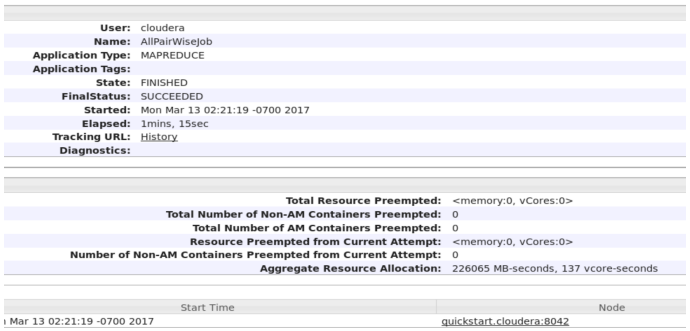


Figure 3: All pair wise job

Listing 5: Mapper

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

        ArrayList<Doc> map = new ArrayList<Doc>();
```

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

    Doc currentDoc = new Doc(key.get(),value.toString());

    int mapSize = map.size();
    for(int i=mapSize-1;i>=0;i--){
       long minId=Math.min(currentDoc.GetId(), map.get(i).GetId());
       long maxId=Math.max(currentDoc.GetId(), map.get(i).GetId());
       context.write(new Text(minId+"_"+maxId), new Text(currentDoc.GetContent()));
       context.write(new Text(minId+"_"+maxId), new Text(map.get(i).GetContent()));
    }

    map.add(currentDoc);

    }


}
```

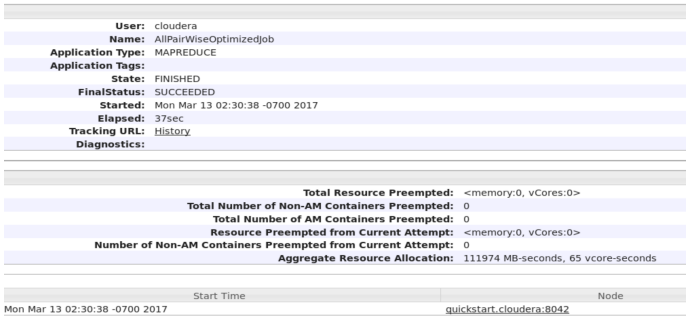
In this mapper we are sure that every possible pair is not duplicated with our ArrayList. Indeed, every time we read a new document we compare it with all other previous existing document and we order the keys ids by ascending order. At the end we emit twice the same key that contains first and second doc Id, but with the two corresponding document corresponding to each key as value separately.

Listing 6: Reducer

```
int comparisonCount =0;
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
           throws IOException, InterruptedException {

      Text s1 = new Text();
      Text s2 = new Text();

      int count = 0;
      for (Text val : values) {
         if(count==0){
            s1.set(val);
         }else{
            s2.set(val);
         }
         count++;
        }


      String [] keys = key.toString().split("_");
      Double sim = AllPairWise.similarity(s1.toString(), s2.toString());
      comparisonCount++;
      if(sim>=AllPairWise.thresold)
         context.write(new Text("(d"+keys[0]+",d"+keys[1]+")"), new
             Text(sim.toString()));
```

In this approach, we are sure that every <Key,<List value» has a unique key with two values in the list. We just have to parse them into Text object and compute similarity.

2.2 Small optimization



User:	cloudera
Name:	AllPairWiseOptimizedJob
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Mon Mar 13 02:30:38 -0700 2017
Elapsed:	37sec
Tracking URL:	History
Diagnostics:	

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	111974 MB-seconds, 65 vcore-seconds

Start Time	Node
Mon Mar 13 02:30:38 -0700 2017	quickstart.cloudera:8042

Figure 4: All pair wise optimized job

Listing 7: Small optimization

```
for(int i=mapSize-1;i>=0;i--){



          String [] parts = currentDoc.GetContent().split("\\s+");
          String [] parts2 = map.get(i).GetContent().split("\\s+");


           Set<String> set1 = new HashSet<String>();
           Set<String> set2 = new HashSet<String>();


              for(String p : parts) {
                set1.add(p);
              }


              for(String p : parts2) {
                set2.add(p);
              }
              Set<String> intersect = new HashSet<>();
              intersect.clear();
             intersect.addAll(set1);
             intersect.retainAll(set2);

             long minId=Math.min(currentDoc.GetId(), map.get(i).GetId());
             long maxId=Math.max(currentDoc.GetId(), map.get(i).GetId());

             if(intersect.size()>0){
                context.write(new Text(minId+"_"+maxId), new
                    Text(currentDoc.GetContent().trim()));
                context.write(new Text(minId+"_"+maxId), new
                    Text(map.get(i).GetContent().trim()));
             }

       }
```

As I have observed that the first approach is very bad in term of computation, I decided to also implement the 2nd approach with the small optimization. We emit only couples that have at least a common word (it means the intersection is at least equal to one). As you can see, time computation is much slower for the same result.

3 Inverted Index

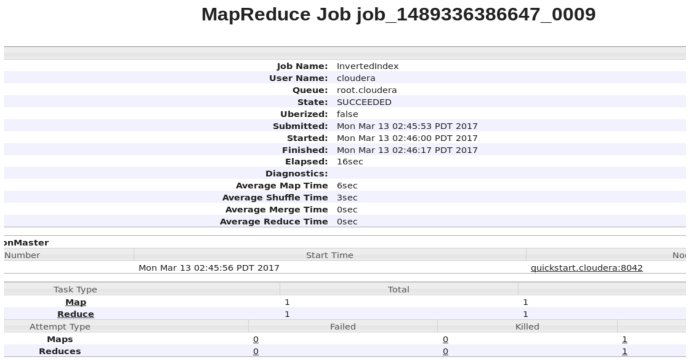


Figure 5: Inverted Index job

Listing 8: Chauduri function

```
public static int getNumberOfWordsToKeep(String document){

   int wordsNumber = countWords(document.trim());
   //System.out.println(document + " "+wordsNumber);
   double td = Math.ceil(thresold * (double)wordsNumber);
   return wordsNumber - (int)td +1;
}

 public static int countWords(String s){

    return s.trim().split("\\s+").length;
}
```

First we implemented the function that allows us to reduce the number of word to keep by the Chauduri rule using the fact that our words are ordered in ascending order.

Listing 9: Class header

```
public class InvertedIndex extends Configured implements Tool{

   static double thresold =0.5;
   static HashMap<Long,String> docs = new HashMap<Long,String>();
```

Moreover, because we emit a limited amount of words, we still have to compare whole documents, this is why we created a static HashMap that can be accessed in the mapper, reducer and even outside.

Listing 10: Mapper

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


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

          docs.put(key.get(), value.toString());
          String[] parts = value.toString().split("\\s+");
        //  System.out.println(value + " : "+getNumberOfWordsToKeep(value.toString()));
          String keytoemit ="";
```

```
        for(int i =0;i<getNumberOfWordsToKeep(value.toString());i++){

           //System.out.println(parts[i]);
           //keytoemit=keytoemit+parts[i]+" ";

           context.write(new Text(parts[i].trim()), key);
        }


      }


   }
```

As you can see, the mapper has two roles :

-Adding the doc in our Hashmap
-Computing the number of word to keep and emitting the corresponding words

Listing 11: Reducer

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

     HashMap<String,String> map = new HashMap<String,String> ();
     int comparisonCount =0;

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

       ArrayList<Long> docsId = new ArrayList<Long>();
       //System.out.println(key +" : ");



       for (LongWritable entry : values) {
           docsId.add(entry.get());
       }

       for(int i=0;i<docsId.size();i++){

          for(int j=i+1;j<docsId.size();j++){

                long iddoc1 = Math.min(docsId.get(i), docsId.get(j));
                long iddoc2 = Math.max(docsId.get(i), docsId.get(j));

                if(iddoc1!=iddoc2){
                   Double sim =
                       InvertedIndex.similarity(docs.get(iddoc1),docs.get(iddoc2));
                   comparisonCount++;
                   if(sim>=InvertedIndex.thresold)
                      map.put("(d"+iddoc1+",d"+iddoc2+")", sim.toString());
                }


                }

          }

       }
```

In my reducer, I decided to not emitting at this part. Indeed, the role of this reducer is to fulfill a HashMap that has the good property that every key is unique ! Then for each key I have

corresponding documents, I can compare documents two by two for each key and put the key (using ordered Ids) and the corresponding similarity.

Since my docsId have unique key, even if we add a value that already exist in the HashMap, only one value will be keep avoiding double docs couple ids.

Listing 12: Cleanup

```
protected void cleanup(Context ctxt) throws IOException,InterruptedException {
   //we call this fonction once at the end

  for(Entry<String, String> entry : map.entrySet())
  {
    ctxt.write(new Text(entry.getKey()), new Text(entry.getValue()));
  }


  System.out.println("Number of comparison performed : " + comparisonCount);

 }

}
```

I called this cleanup method at the end, we have now a clean hashmap with a list of ordered unique docsId couple in the key and their similarity. We just have to loop on it and emit every <Key,Value> pair.

4 Comparisons

For the execution of this all pair wise approach, the job has been run on a smaller set of data than the inverted index. Indeed, for the whole corpus obtained after the preprocessing part, the job took more than one hour to run ! For the same text corpus the Inverted Index took only about one minute !

Then, in order to compare the two approaches, we will perform it on smaller sets of approximatively 3000 lines. We could have done also a "cross validation approach" by doing job on several loads of 3000 lines to average the result of several trials.

4.1 Checking the results are the sames

We compare the size of two files obtained in the file system and hopefully they are the same. Moreover, we looked at the number of bytes written in the job execution and they are also same.

```
                Physical memory (bytes) snapshot=9792
                Virtual memory (bytes) snapshot=31420
                Total committed heap usage (bytes)=92
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=48503
        File Output Format Counters
                Bytes Written=25525
ing
oudera@quickstart HM]$ hadoop jar hm2.jar setsimila
```

Figure 6: Byte written check

4.2 Time execution and number of comparison performed

We took a threshold value of 0.5 for the example.

As you can see on the previous screen shot, the all pair wise approach took one minute and 15 seconds while the inverted Index took for the same input file 16 seconds ! We also added a counter to compute the number of comparison :

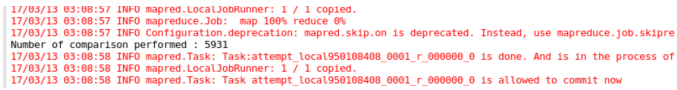


Figure 7: Number of comparison for Inverted Index

Result : 5931 comparisons

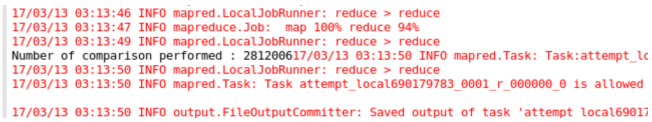


Figure 8: Number of comparison for All Pair

Result : 2812006 comparisons

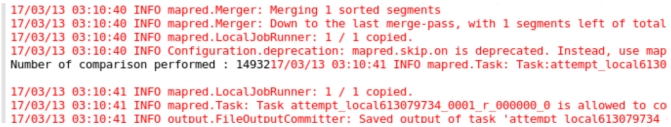


Figure 9: Number of comparison for All Pair with small optimization

Result : 14932 comparisons

As you can see, the inverted index is much better than the other approach for the computation time as well as the number of comparisons performed because of his indexation method that emit much less amount of data. Indeed according to [Chaudhuri et al. 2006] he proves that it is "adequate to index only the first $|d| - \lceil t \bullet |d| \rceil + 1$ words of each document d, without missing any similar documents (known as the prefix-filtering principle)". It reduces quite well the needed data to emit.
Moreover the all pair approach doesn't use at all the fact that sentences are ordered by increasing order of frequencies, it just send all possibles combination of pair. Then the more the threshold value is high the less data is used in the inverted index method. The pair approach has the same amount of data whatever the threshold.
The only drawbacks is that as the inverted index name said, it is a index, that means we got index number that allows us to get easier access to raw document. But those documents still have to be all store in a list.