Big Data Processing and Analytics: Assignment 2

Louis MARTIN
louis.martin@student.ecp.fr

March 17, 2017

In this assignment we are going to compare documents using the Jaccard similarity. For the sake of simplicity we are going to run our algorithms on the works of William Shakespeare, each line being considered as a document.

1 Setup

This section is the same as for the first assignment.

1.1 System specifications

- **Operating system**:
  Ubuntu 16.04 (Native)
- **System specifications**:
  Model: Dell Inspiron 17R 5720
  Processor: i5-3210M
  Cores: 2
  Threads: 4
  Ram: 8 GB
  Storage: 256GB SSD (MLC)
- **Java version**:
  openjdk version "1.8.0_121"
  OpenJDK Runtime Environment (build 1.8.0_121-8u121-b13-0ubuntu1.16.04.2-b13)
  OpenJDK 64-Bit Server VM (build 25.121-b13, mixed mode)
- **Haddop version**:
  Hadoop 2.7.3
  Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r baa91f7c6bc9cb92be5982de4719c1c8af91ccff
  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/hadoop/share/hadoop/common/hadoop-common-2.7.3.jar

Hadoop was installed using this tutorial and configured using this tutorial.

1.2 Configuration

The configuration comes from the official documentation for a single Hadoop node cluster. The following configuration files allows Hadoop and YARN to run in a pseudo-distributed mode.

- **core-site.xml:**

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

- **hdfs-site.xml:**

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

- **mapred-site.xml:**

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

- **yarn-site.xml:**

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!--
    To prevent unhealthy nodes and jobs hanging due to low disk space
    http://stackoverflow.com/questions/29131449/why-does-hadoop-report-unhealthy-node-local-dirs-and-log-dirs-are-bad
    -->
    <property>
        <name>yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage</name>
        <value>98.5</value>
    </property>
</configuration>
```

- **Commands to set up HDFS and YARN:**

```
    # Format the filesystem
    hdfs namenode -format
    # Start HDFS
    start-dfs.sh

    # Create directories to execute MapReduce jobs
    hdfs dfs -mkdir /user
    hdfs dfs -mkdir /user/louis

    # Put the data in HDFS
    hdfs dfs -put ~/dev/bdpa/a1/data data
```

```

# Start YARN Ressource manager
start-yarn.sh
```

1.3 Workflow

As I ran everything from the command line interface, I created helper shell scripts in order to facilitate the compilation and execution of the mapreduce jobs.

- **compile.sh**:
  The *compile.sh* script which takes as input parameter the name of the main class of the *.java* file to compile it into a *.jar* file.
  Usage: ./compile.sh Preprocess.

```
#!/bin/bash
rm -rf *.class *.jar;
hadoop com.sun.tools.javac.Main $1.java;
jar cf $1.jar $1*.class;
rm $1*.class;
```

- **run.sh**:
  The compilation and execution of the map-reduce job is then wrapped into this script. It takes as input the main class and the input file or directory.
  Usage: ./run.sh Preprocess data/corpus.

```
#!/bin/bash
./compile.sh $1;
hadoop jar $1.jar $1 $2 out/;
hdfs dfs -getmerge out/ ${1,,}.csv;
rm .*.crc;
```

2 Pre-processing the input (10)

The following preprocessing is done with a mapreduce job in **Preprocess.java**. Each line is read by a single map call, processed and the written to a file. The key we used to uniquely identify a line is its byte offset, which is natively provided in the map call. The value is the processed line.

2.1 (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.

2.1.1 WordCount.java

In order to remove the stopwords we get the number of occurences of each word in the corpus and store it in wordcount.csv. We will reuse these wordcounts for sorting the words.

- **WordCount.java** (inspired from the official documentation and the first assignment):

```
  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 {
      // Splits a string to tokens (here words).
```

```
        // We split on anything that is not an alphanumerical character
        String[] tokens = value.toString().toLowerCase().split("[^a-z0-9]");
        for (int i=0; i<tokens.length; i++){
          word.set(tokens[i]);
          // Write one (key, value) pair to context
          context.write(word, one);
        }
      }
    }

    public static class IntSumReducer
         extends Reducer<Text,IntWritable,Text,IntWritable> {
      private IntWritable result = new IntWritable(10);

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

- Output **wordcount.csv**:
  Here is an extract from the output

```
anticipation, 1
antick, 1
anticly, 1
antics, 2
antidote, 1
antidotes, 1
antigonus, 32
antiopa, 1
antipathy, 1
antipholus, 220
antipholuses, 1
```

2.1.2 Preprocess.java

We then preprocess the lines using the wordcounts. The preprocessing is done by a single map task (no reduce taks).

```
        // Disable the reducer, we are implementing a 'map only' job
        job.setNumReduceTasks(0);
        job.setMapperClass(LineCleanerMapper.class);
```

- **Clean the line**:
  We first clean the lines from characters other than alpha-numerical characters.

```
        // Splits a string to tokens (here words).
        // We split on anything that is not an alphanumerical character
        String[] words = value.toString().toLowerCase().split("[^a-z0-9]");
```

- **Keep each word once per line**:

```
        // We use the "no duplicate" property of sets
        HashSet<String> wordSet = new HashSet<String>();
```

- **Remove all stopwords**:
  A stopword is defined by a word with less than 4000 occurences. The wordcounts are stored in a hashmap for O(1) element retrieval.

```
        for (String word : words) {
          // Take only words that are not stopwords
          if (wordCounts.get(word) < 4000){
            wordSet.add(word);
          }
        }
```

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

As we implemented a map only job, the number of output records is given by the Counter **MAP_OUTPUT_RECORDS**. The number of output records is **114982**.

```
        // Retrieve number of lines
        Counters counters = job.getCounters();
        Counter linesCounter = counters.findCounter(TaskCounter.MAP_OUTPUT_RECORDS
            );
        System.out.println("Lines written: " + linesCounter.getValue());
        // Write the count of unique words to a file in HDFS
        Path filePath = new Path("lines_preprocessed.txt");
        if (hdfs.exists(filePath)) {
            hdfs.delete(filePath, true);
        }
        FSDataOutputStream fin = hdfs.create(filePath);
        fin.writeUTF(Long.toString(linesCounter.getValue()));
        fin.close();
```

2.3 (7) Order the tokens of each line in ascending order of global frequency.

We need to sort the resulting words based on their respective count. With this aim in mind, we implemented a custom comparator to sort the array of words based on their counts that we previously stored in a HashMap:

```
    private class CountComparator implements Comparator<String> {
        // Custom comparator that will sort words based on their count
        @Override
        public int compare(String word1, String word2) {
            return wordCounts.get(word1).compareTo(wordCounts.get(word2));
        }
    }
```

We then sort the words using this comparator:

```
        Arrays.sort(uniqueWords, new CountComparator());
```

3 Set-similarity joins (90)

3.1 (40) Perform all pair-wise comparisons between documents, using the following technique: Each document is handled by a single mapper (remember that lines are used to represent documents in this assignment). 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 the reduce phase, perform the Jaccard computations for all/some selected pairs. Output only similar pairs on HDFS, in TextOutputFormat. Make sure that the same pair of documents is compared no more than once. Report the execution time and the number of performed comparisons.

3.1.1 run_naive_similarity.sh

We wrapped the execution of the necessary jobs in the **run_naive_similarity.sh**.

As the number of comparisons is simply too large for my computer to handle (too long and to much storage taken), I reduced the number of lines to be compared to 5000.

```bash
#!/bin/bash
./run.sh WordCount data/corpus
./run.sh Preprocess data/corpus
# Keep only first 5000 lines for faster execution
sed -i "5001,$ d" preprocess.csv
./run.sh NaiveSimilarity preprocess.csv
```

3.1.2 NaiveSimilarity.java

- **Jaccard similarity**:
  The jaccard similarity between two documents is the length of their intersection, divided by their union.

```java
  public static float jaccard(String doc1, String doc2){
    Set words1 = new HashSet<String>(Arrays.asList(doc1.split(" ")));
    Set words2 = new HashSet<String>(Arrays.asList(doc2.split(" ")));
    Set intersection = new HashSet<String>(words1);
    intersection.retainAll(words2);
    Set union = new HashSet<String>(words1);
    union.addAll(words2);
    float similarity = (float) intersection.size() / (float) union.size();
    return similarity;
  }
```

- **Mapper**:
  We first read all keys in an array. Then the Mapper reads each preprocessed lines. It finally outputs a pair of the key of the current line and another key for each other line in the corpus. The value is the content of the line.

  In order to retrieve the content of both documents, we need to make sure that they will end up in the same reducer and hence have the same key. We put the lowest key first for always having a single key. This also ensures that each pair is compared only once (more details in the comments):

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

    private static ArrayList<Long> keys = NaiveSimilarity.readAllKeys();
    private Text pair = new Text();
    private Text currentValue = new Text();
```

```

    public void map(LongWritable key, Text value, Context context
                    ) throws IOException, InterruptedException {
      String[] split = value.toString().split(",");
      Long currentKey = Long.parseLong(split[0]);
      currentValue.set(split[1]);
      // We will compare this document to only previous documents in order to
      // only compare a given pair of documents once.
      for (Long otherKey : keys){
        // The following pair is a quick hack to pass a pair of keys, instead
        // of defining a more elegant ComparableWritable.
        // This new class would not have brought much more (YAGNI principle).
        // We put the lowest key first to be sure that the same pair of
        // documents always have the same key and thus end up in the same reducer.
        // Note that we don't take into account the case where the keys are the same.
        if (currentKey < otherKey){
          pair.set(currentKey.toString() + "\t" + otherKey.toString());
          context.write(pair, currentValue);
        } else if (currentKey > otherKey) {
          pair.set(otherKey.toString() + "\t" + currentKey.toString());
          context.write(pair, currentValue);
        }
      }
      }
    }
```

- **Reducer**:
  Each reduce call is supposed to receive exactly two values, one for each document.

```
  public static class CompareReducer
       extends Reducer<Text, Text, Text, Text> {
    private Text value1 = new Text();
    private Text value2 = new Text();

    public void reduce(Text key, Iterable<Text> values,
                       Context context
                       ) throws IOException, InterruptedException {
      // Values is supposed to contain two elements. The content of each of the
      // two documents to be compared.
      Iterator<Text> itr = values.iterator();
      value1.set(itr.next());
      value2.set(itr.next());
      if (itr.hasNext()){
        throw new RuntimeException("More than one value for a given pair of document ids was received.");
      }
      float similarity = NaiveSimilarity.jaccard(value1.toString(), value2.toString());
      if (similarity > 0.8) {
        Text value = new Text(similarity + "\t\"" + value1.toString() + "\" - \"" + value2.toString() + "\"");
```

```
                context.write(key, value);
            }
        }
    }
```

The number of comparison is simply the number of reduce calls given by the **REDUCE_INPUT_GROUPS** counter. The number of comparison is **12497500** which is exactly $\frac{(n-1)n}{2}$ with $n = 5000$ being the number of documents. The first document is compared to $n-1$ documents, the second to $n-2$ documents. It is therefore equal to $\sum_1^{n-1} = \frac{(n-1)n}{2}$ (not that the upper bound is $n-1$).

The execution time on 5000 documents is **131.62 s**

3.2 (40) Create an inverted index, only for the first $|d| - ceil(t \times |d|) + 1$ words of each document d (remember that they are stored in ascending order of frequency). In your reducer, compute the similarity of the document pairs. Output only similar pairs on HDFS, in TextOutputFormat. Report the execution time and the number of performed comparisons.

3.2.1 run_fast_similarity.sh

We wrapped the execution of the necessary jobs in the **run_fast_similarity.sh**. For consistency with the previous method, we also only keep the first 5000 lines.

```
#!/bin/bash
./run.sh WordCount data/corpus
./run.sh Preprocess data/corpus
# Keep only first 5000 lines for faster execution
sed -i "5001,$ d" preprocess.csv
./run.sh FastSimilarity preprocess.csv
```

3.2.2 FastSimilarity.java

- **Mapper**:
  The mapper is based in the inverted index mapper from the previous assignment. We output a key value pair for the first words of each document with the key being the current word, and the value, the document it appears in. The first words to take into account is given by the cutoff index: $|d| - ceil(t \times |d|) + 1$. We don't need to index more words because if all those first words don't match, then it was shown that the similarity is below our threshold, hence we can skip the comparison.

```
    public static class TokenizerMapper
            extends Mapper<Object, Text, Text, LongWritable>{
        private Text word = new Text();
        private LongWritable docKey = new LongWritable();

        public void map(Object key, Text value, Context context
                        ) throws IOException, InterruptedException {
            String[] split = value.toString().split(",");
            docKey.set(Long.parseLong(split[0]));
            // Splits a string to tokens (here words)
            String[] words = split[1].split(" ");
            int cutoff = words.length - (int) Math.ceil((double) words.length *
                threshold) + 1;
            for (int i=0; i < cutoff; i++) {
                word.set(words[i]);
                context.write(word, docKey);
            }
        }
    }
```

- **Reducer**:
  For each reduce call, the reducer will compare all documents for which the considered word appears.

  Some comparisons will be made multiple times for documents that have several words in common. However, having sorted the words by ascending order of frequency ensures that the most common words are not indexed, and that we make the minimum of comparisons possible.

  In order to skip the duplicated comparisons, we could just have kept track of the pairs that were already compared in a static variable. This could however defeat the purpose of distributed computing (difficult to have one shared static variable between all devices).

```java
  public static class CompareReducer
       extends Reducer<Text, LongWritable, Text, Text> {
    private Text outputKey = new Text();
    private Text outputValue = new Text();
    private HashMap<Long, String> allDocs = readAllDocs();

    public void reduce(Text word, Iterable<LongWritable> docKeys,
                       Context context
                       ) throws IOException, InterruptedException {
      ArrayList<LongWritable> processedKeys = new ArrayList<LongWritable>();
      String doc1 = new String();
      String doc2 = new String();

      for (LongWritable docKey : docKeys) {
        for (LongWritable processedKey : processedKeys) {
          doc1 = allDocs.get(processedKey.get());
          doc2 = allDocs.get(docKey.get());
          float similarity = FastSimilarity.jaccard(doc1, doc2);
          context.getCounter(ComparisonCounter.COMPARISONS).increment(1);
          if (similarity > threshold){
            outputKey.set("(" + processedKey.toString() + ", " + docKey.toString() + ")");
            outputValue.set(similarity + "\t\"" + doc1 + "\" - \"" + doc2 + "\"");
            context.write(outputKey, outputValue);
          }
        }

        LongWritable processedKey = new LongWritable(docKey.get());
        processedKeys.add(processedKey);
      }
    }
  }
```

- **Output**:
  Here is an extract of the most similar documents (the output file is cluttered with licence lines which are 100% similar).

```
(156814, 156587) : 0.8333333    "spare clown sir o lord" - "thick spare clown sir o lord"
(234568, 154478) : 0.75 "lafeu countess clown enter" - "countess clown enter "
(184063, 238872) : 1.0  "re clown enter" - "re clown enter"
(234568, 130241) : 0.6  "lafeu countess clown enter" - "steward countess clown enter"
(154478, 130241) : 0.75 "countess clown enter" - "steward countess clown enter"
```

```
314 (234568, 182395) : 0.75 "lafeu countess clown enter" - "countess clown enter
    "
315 (154478, 182395) : 1.0 "countess clown enter" - "countess clown enter"
316 (130241, 182395) : 0.75 "steward countess clown enter" - "countess clown
    enter"
317 (239762, 180616) : 1.0 "illinois benedictine college etext provided
    gutenberg project" - "illinois benedictine college etext provided
    gutenberg project"
318 (239762, 113060) : 1.0 "illinois benedictine college etext provided
    gutenberg project" - "illinois benedictine college etext provided
    gutenberg project"
319 (180616, 113060) : 1.0 "illinois benedictine college etext provided
    gutenberg project" - "illinois benedictine college etext provided
    gutenberg project"
```

Our custom counter gives us a number of comparison is **107200**. The execution time is **2.54 s**. This is a huge improvement over the previous method.

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

- In the first algorithm, we naively compared all documents with every other. The computation time grows in n^2 which is not feasible for a large set of documents as is common in big web companies.
- The second smarter method, starts from the idea that documents with very few words in common shouldn't even be compared (this is the case for a majority of pairs).

  The first idea is to index only the first $|d| - ceil(t \times |d|) + 1$ which ensures that documents with no words in common in those words will have a similarity lower than the threshold.

  A Second idea is that indexing the rare words before the more common ones will also lead to less comparisons. These words will indeed appear in a smaller subset of documents hence less comparisons.

The difference in execution time is directly linked to the number of comparisons and the number of key value pairs that are passed between the mappers and reducers.

4 Conclusion

This approach can used for information retrieval with search engines for example. The query being the document we want to compare to all others. This could also be used for documents clustering, the jaccard similarity giving us distances between documents.

In order to improve the relevance of the results we could use more advanced similarity measures such as the well know TF-IDF measure.