Massive data processing 2nd assignment

Ningxin GUO DSBA
Background: ESSEC grande ecole

Pre-processing the input (10)

You will use the document corpus of pg100.txt, as in your previous assignments, assuming that each line represents a distinct document (treat the line number as a document id). Implement a pre-processing job in which you will:

- (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.
- (1) Store on HDFS the number of output records (i.e., total lines).
- (7) Order the tokens of each line in ascending order of global frequency.

Output the remaining lines. Store them on HDFS in TextOutputFormat in any order.

As we already had the stopwords from last assignment, we import it to eliminate high frequency word and apply the pattern to filter out.

```
HashSet<String> stopWords = new HashSet<String>();
rdr = new BufferedReader(new FileReader(new
File("/home/cloudera/workspace/MDPassignment1/stopWords.csv")));
String pattern;
```

To do the counter part, I refer to the following example code:

```
FileWriter writer = new FileWriter("MyFile.txt", true);
BufferedWriter bufferedWriter = new BufferedWriter(writer);
bufferedWriter.write("Hello World");
bufferedWriter.newLine();
bufferedWriter.write("See You Again!");
```

We change the wirte part and apply the close right after.

In the reducer, before writing the output in HDFS, we increment the counter by 1 by using the method

```
context.getCounter(CUSTOM_COUNTER.NB_LINES).increment(1);.
```

In order to rank the output in ascending order, we list the integers with prefix, which is unique, and do the ranking job based on these integers.

```
for (String num : rangingData) {
       if (num.length() > 0);{
       if (newsortedbyFreq.length() > 0) newsortedbyFreq.append(" ");{
          newsortedbyFreq.append(num.replaceAll("#\\d+", ""));
                } } }
```

And

```
context.write(new Text(newstring.toString()), new
Text(newsortedbyFreq.toString().replaceAll("#\\d+", "").replaceAll(",",
"").replaceAll("__#null", "")));
```

The output looks like:

```
0, ebook#6, works#8, complete#9, william#10, shakespeare#14, gutenberg#15,
project#17
81, william#10, shakespeare#14
104, anywhere#1, anyone#2, cost#3, use#6, ebook#6
170, almost#1, restrictions#1, whatsoever#1, away#2, copy#4, give#5
```

Second Part : Set-similarity joins

You are asked to efficiently identify all pairs of documents (d1, d2) that are similar (sim(d1, d2) >= t), given a similarity function sim and a similarity threshold t. Specifically, assume that:

-each output line of the pre-processing job is a unique document (line number is the document id), documents are represented as sets of words,

-sim(d1, d2) = Jaccard(d1, d2) = $|d1 \cap d2| / |d1 \cup d2|$,

-t=0.8.

a)

After adapting the previous pre-processing job, we need to make sure that the same pair of documents is compared no more than once.

We apply a compareto function in our code to solve this problem, and implement the WritableComparable class for all the pair keys:

```
    public int compareTo(TextPair other) {
        int cmpFirstFirst = first.compareTo(other.first);
        int cmpSecondSecond = second.compareTo(other.second);
        int cmpFirstSecond = first.compareTo(other.second);
        int cmpSecondFirst = second.compareTo(other.first);

        if (cmpFirstFirst == 0 && cmpSecondSecond == 0 || cmpFirstSecond ==
0
                && cmpSecondFirst == 0) {
            return 0;
        }
        Text thisSmaller;
        Text otherSmaller;
        Text thisBigger;
        Text otherBigger;
```

When trying to create a function to compute the Jaccard Similarity between 2 sets, we are inspired by the code and discussion:

```
a.addAll(A.removeAll(B)?A:A);
a.retainAll(b); // intersection
a.removeAll(b); // difference
a.addAll(b); // union
```

https://nickgrattan.wordpress.com/2014/02/18/jaccard-similarity-index-for-measuring-document-similarity/(though in c#)

After that, we output only the similar pairs and the execution time is shown below:

b)

We create the inverted index only for the first |d| - [t |d|] + 1 words of each document.

For the mapper part, we split the whole string by space, and apply the threshold which equals t 0.8:

```
long threshold_number = Math.round(Words.length - (Words.length * 0.8) +
1);
```

For the reducer part, we implement the same method as we used before to compute the Jaccard Similarity:

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {
    private BufferedReader reader;
    public double jaccardsim(TreeSet<String> s1, TreeSet<String> s2) {
        if (s1.size() < s2.size()) {
            TreeSet<String> s1bis = s1;
            s1bis.retainAll(s2);
            int inter = s1bis.size();
            s1.addAll(s2);
            int union = s1.size();
            return (double) inter / union;
        } else {
            TreeSet<String> s1bis = s2;
```

```
            s1bis.retainAll(s1);
            int inter = s1bis.size();
            s2.addAll(s1);
            int union = s2.size();
            return (double) inter / union;
        }
    }
```

c)

In order to accelerate the process, we only focus the result of the top 500 lines. The first method takes 42 seconds and the second needs 27 seconds (as can be seen in the screenshot above). The latter one is definitely quicker than the first one, as the size of the hashset is reduced shorter by the equation:

```
|d| - [t |d|] + 1
```