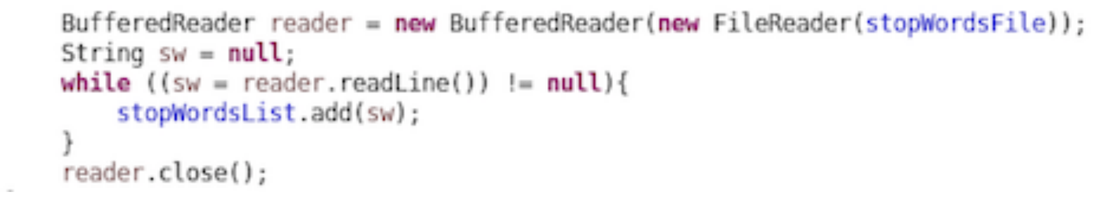
Nandita Nagar

Big Data Processing & Analytics

Assignment 2

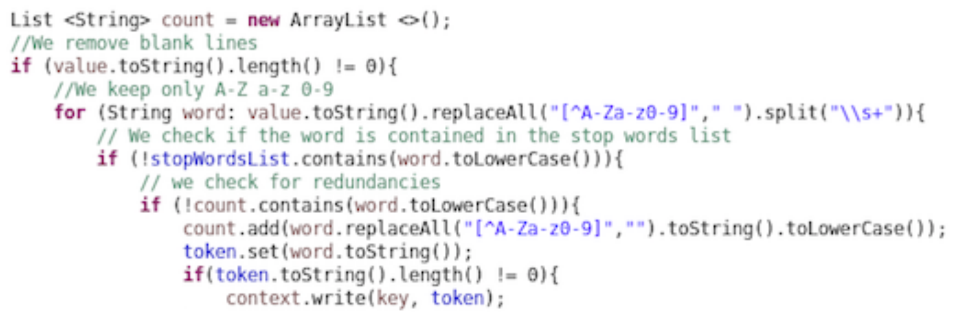
Part 1 Pre Processing

First the requirement is to remove all stop words – the words that were identified in the previous assignment. This had been implemented using the stop words same logic i.e. the stop words are the ones with frequency more than 4000. We read that file in the mapper:



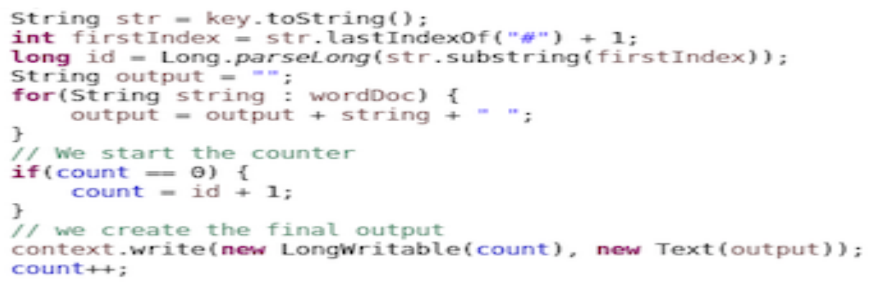
```
BufferedReader reader = new BufferedReader(new FileReader(stopWordsFile));
String sw = null;
while ((sw = reader.readLine()) != null){
    stopWordsList.add(sw);
}
reader.close();
```

As per instructions we also need to remove empty lines. To remove special characters, keeping only [a-z], [A-Z] and [0-9]. In addition, the word redundancies are to be removed from each line and only unique words are to be retained once per line:



```
List <String> count = new ArrayList <>();
//We remove blank lines
if (value.toString().length() != 0){
    //We keep only A-Z a-z 0-9
    for (String word: value.toString().replaceAll("[^A-Za-z0-9]"," ").split("\\s+")){
        // We check if the word is contained in the stop words list
        if (!stopWordsList.contains(word.toLowerCase())){
            // we check for redundancies
            if (!count.contains(word.toLowerCase())){
                count.add(word.replaceAll("[^A-Za-z0-9]","").toString().toLowerCase());
                token.set(word.toString());
                if(token.toString().length() != 0){
                    context.write(key, token);
```

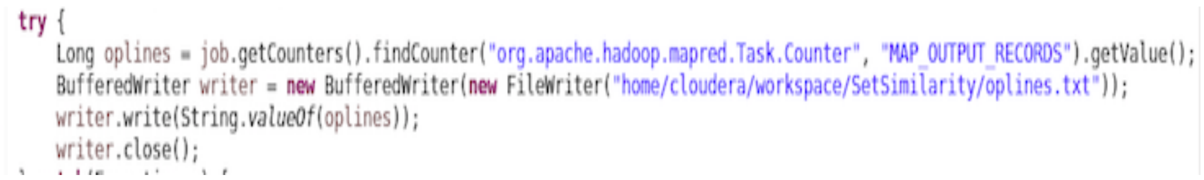
In the reducer we find the frequency of each word and remove redundancies to keep only unique words in the output:



```
String str = key.toString();
int firstIndex = str.lastIndexOf("#") + 1;
long id = Long.parseLong(str.substring(firstIndex));
String output = "";
for(String string : wordDoc) {
    output = output + string + " ";
}
// We start the counter
if(count == 0) {
    count = id + 1;
}
// we create the final output
context.write(new LongWritable(count), new Text(output));
count++;
```

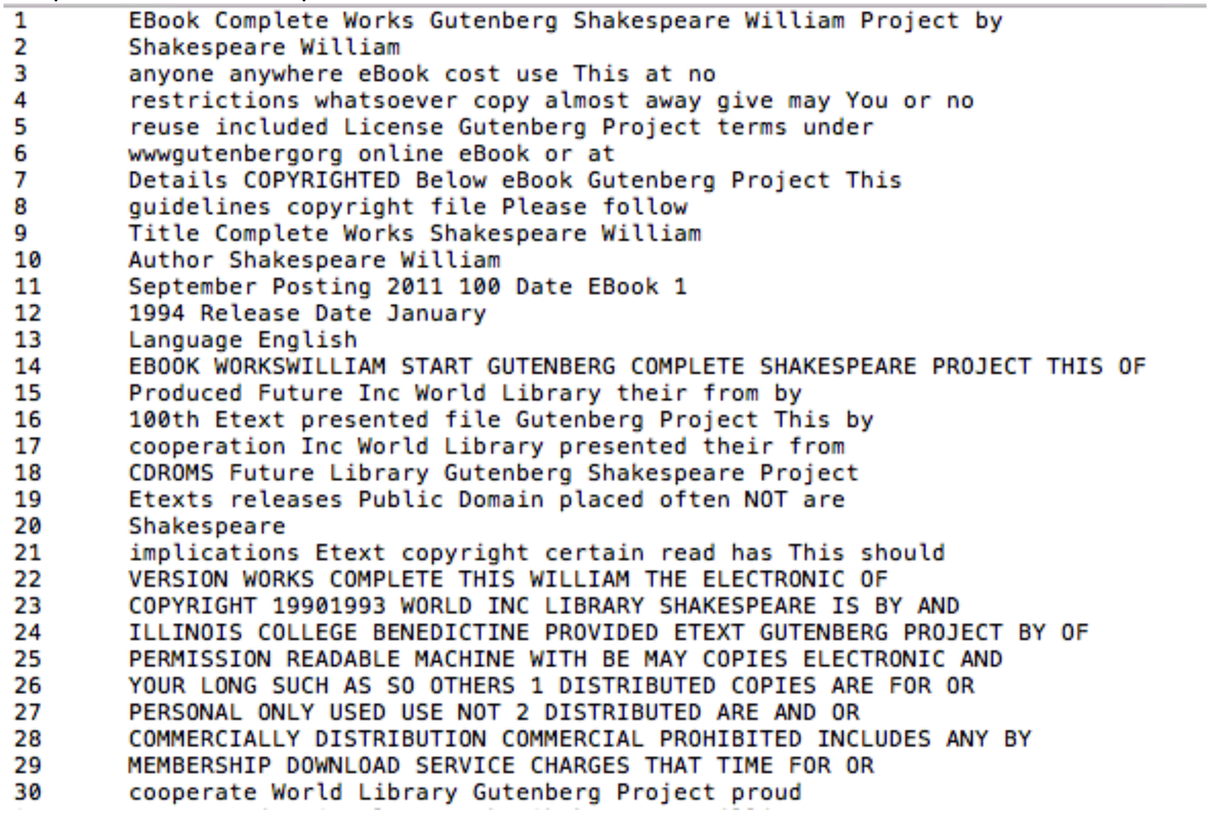
We also created a counter that will give the number of output lines once the job is completed. Next is to store the number of output lines. That has been stored in local

location. I invested disproportionate time in attempting to put that on HDFS and chose the alternate approach eventually:
Record Lines number is ‘114461’:



```
try {
    Long oplines = job.getCounters().findCounter("org.apache.hadoop.mapred.Task.Counter", "MAP_OUTPUT_RECORDS").getValue();
    BufferedWriter writer = new BufferedWriter(new FileWriter("home/cloudera/workspace/SetSimilarity/oplines.txt"));
    writer.write(String.valueOf(oplines));
    writer.close();
```

Output: PreProcess Ouptut.txt



```
1       EBook Complete Works Gutenberg Shakespeare William Project by
2       Shakespeare William
3       anyone anywhere eBook cost use This at no
4       restrictions whatsoever copy almost away give may You or no
5       reuse included License Gutenberg Project terms under
6       wwwgutenbergorg online eBook or at
7       Details COPYRIGHTED Below eBook Gutenberg Project This
8       guidelines copyright file Please follow
9       Title Complete Works Shakespeare William
10      Author Shakespeare William
11      September Posting 2011 100 Date EBook 1
12      1994 Release Date January
13      Language English
14      EBOOK WORKSWILLIAM START GUTENBERG COMPLETE SHAKESPEARE PROJECT THIS OF
15      Produced Future Inc World Library their from by
16      100th Etext presented file Gutenberg Project This by
17      cooperation Inc World Library presented their from
18      CDROMS Future Library Gutenberg Shakespeare Project
19      Etexts releases Public Domain placed often NOT are
20      Shakespeare
21      implications Etext copyright certain read has This should
22      VERSION WORKS COMPLETE THIS WILLIAM THE ELECTRONIC OF
23      COPYRIGHT 19901993 WORLD INC LIBRARY SHAKESPEARE IS BY AND
24      ILLINOIS COLLEGE BENEDICTINE PROVIDED ETEXT GUTENBERG PROJECT BY OF
25      PERMISSION READABLE MACHINE WITH BE MAY COPIES ELECTRONIC AND
26      YOUR LONG SUCH AS SO OTHERS 1 DISTRIBUTED COPIES ARE FOR OR
27      PERSONAL ONLY USED USE NOT 2 DISTRIBUTED ARE AND OR
28      COMMERCIALLY DISTRIBUTION COMMERCIAL PROHIBITED INCLUDES ANY BY
29      MEMBERSHIP DOWNLOAD SERVICE CHARGES THAT TIME FOR OR
30      cooperate World Library Gutenberg Project proud
```

Part 2 Set-similarity joins

We 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 assumption is 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□ d2| / |d1 U d2|,
- t = 0.8.

Question a)
Naïve Approach – The program exhaustively checks for all pairwise comparisons between documents i.e. each line and then only the pairs that are similar as given as output.

We give the mapper input to the mapper as a key value pair. Key is a line number that has an associated text value to it:

```
private Text opKey = new Text();
private Text opValue = new Text();
```

We specify the max number of document ids. We know that each line is on input is treated as a separate document id. Here there are 100 for consideration. L signifies the long type:

```
public Long maxdID = 100L;
```

The map method has two if loops – one to check for the current document id and emits a key value pair and another to check for the max document id as specified earlier and emits a key value pair:

```
for (Long i = 1L; i < dId; i = i + 1L) {
    opKey.set(Long.toString(i) + "$" + Long.toString(dId));
    context.write(opKey, opValue);
}

for (Long i = dId + 1L; i < maxdID + 1L; i = i + 1L) {
    opKey.set(Long.toString(dId) + "$" + Long.toString(i));
    context.write(opKey, opValue);
}
```

In reduce we start with a document pair of A and B:

```
String docA = values.iterator().next().toString();
String docB = values.iterator().next().toString();
```

We initialize a counter to check for the number of comparisons:

```
Counter count = context.getCounter(CountersEnum.class.getName(),
CountersEnum.NUM_COMP.toString());
//The counter increments by one after each comparison
count.increment(1);
```

We compute the similarity between the document pair A and B using the compute similarity method:

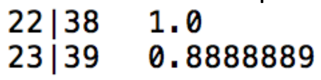
```
Float sim = computeSimilarity(docA, docB);
```

This if loop check if the calculated similarity is more than the threshold similarity. If yes, we consider this pair:

```
if (sim > simT) {
        opVal.set(sim);
      context.write(key, opVal);
  }
```

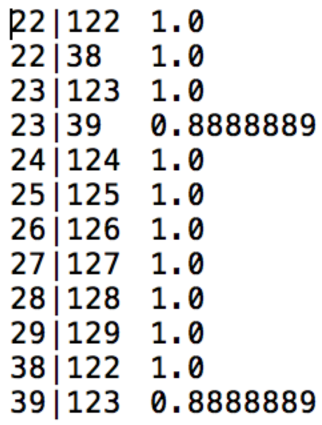
Output:

First we tried the input for a small sample file of 100 documents:



```
22|38   1.0
23|39   0.8888889
```

Since that worked fine we scaled up to 1000 documents and that worked as well:

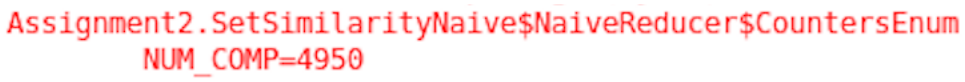


```
22|122  1.0
22|38   1.0
23|123  1.0
23|39   0.8888889
24|124  1.0
25|125  1.0
26|126  1.0
27|127  1.0
28|128  1.0
29|129  1.0
38|122  1.0
39|123  0.8888889
```

Execution time: 29 seconds for 1000 lines; 44 seconds for 100 lines

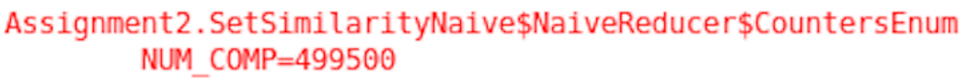
Number of performed comparisons:

For 100 lines:



```
Assignment2.SetSimilarityNaive$NaiveReducer$CountersEnum
        NUM_COMP=4950
```

For 1000 lines



```
Assignment2.SetSimilarityNaive$NaiveReducer$CountersEnum
        NUM_COMP=499500
```

Question b)

Indexing approach – According to this approach it is adequate to index only the first |d| - ⌈t *|d|⌉ + 1 words of each document d, without missing any similar documents (known as the prefix-filtering principle), where t is the Jaccard similarity threshold and |d| is the number of words in d.

We index only a single word for each document, guarantee that all similar documents will be compared and skip comparisons between some non-similar documents.

We use a straightforward mapper implementation that reads each document i.e. each line and calculates the number of words as per the formula and assigns it to the index Count. The idea is to create key value pairs of words and docs:

```
int Count = ((Double) (words.length - Math.ceil(simT * words.length) + 1)).intValue();

for (int i = 0; i < Count; i += 1) {
    opKey.set(words[i]);
    context.write(opKey, value);
```

The key value pairs of word indexes and each line as a doc, created after the mapper are given to the reducer. As per the logic of this approach the reducer will iterate over all the pair and calculate similarities:

```
List<Long> dId = new ArrayList<Long>();
List<String> d = new ArrayList<String>();

for (Text val : values) {
    dId.add(Long.parseLong(val.toString().split("\t")[0]));
    d.add(val.toString().split("\t")[1]);
}

for (int i = 0; i < dId.size(); i += 1) {
    Long dId1 = dId.get(i);
    String d1 = d.get(i);

    for (int j = i + 1; j < dId.size(); j += 1) {
        Long dId2 = dId.get(j);
        String d2 = d.get(j);
```

Calculates similarity by invoking the method at the bottom for calculation as per the formula:

```
if (sim > simT) {
      String idPair = (dId1 < dId2) ? Long.toString(dId1) + "$" + Long.toString(dId2) :
          Long.toString(dId2) + "$" + Long.toString(dId1);
      opKey.set(idPair);
      opValue.set(sim);
    context.write(opKey, opValue);
```

The counter is the same as used in the previous approach. It counts the number of comparisons and increments by one for each successive one. The if loop checks if the calculated similarity is more than the threshold. If yes, we consider that pair:

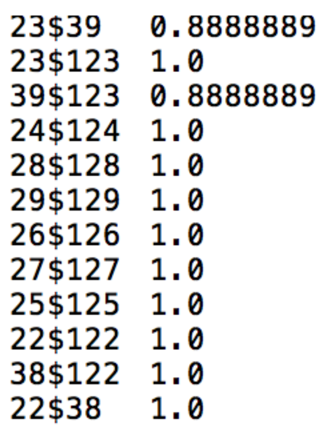
```
Counter count = context.getCounter(CountersEnum.class.getName(),
CountersEnum.NUM_COMP.toString());
count.increment(1);
```

Output:

First we tried the input for a small sample file of 100 documents:

```
23|39   0.8888889
22|38   1.0
```

Since that worked fine we scaled up to 1000 documents and that worked as well:

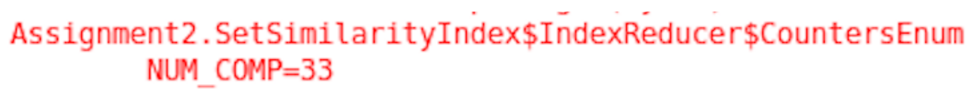


```
23$39   0.8888889
23$123  1.0
39$123  0.8888889
24$124  1.0
28$128  1.0
29$129  1.0
26$126  1.0
27$127  1.0
25$125  1.0
22$122  1.0
38$122  1.0
22$38   1.0
```

Execution time: 53 seconds for 1000 lines; 17 seconds for 100 lines

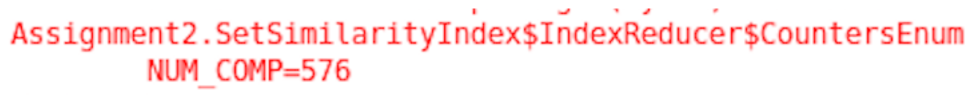
Number of performed comparisons:

For 100 lines:



```
Assignment2.SetSimilarityIndex$IndexReducer$CountersEnum
        NUM_COMP=33
```

For 1000 lines:



```
Assignment2.SetSimilarityIndex$IndexReducer$CountersEnum
        NUM_COMP=576
```

Due to the limited capacity of my machine, software installation issues, previous crashes and the feedback from many colleagues about their machines crashing while running comparisons with the entire corpus I took the precaution of first running the code with 100 lines. When that worked fine, I scaled up 10 folds and the output was satisfactorily demonstrating the concepts that this assignment intends for us to learn. Hence, I decided to use an input of 1000 lines for performing the set similarity joins with both the approaches.

Question c)

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

As we have observed there is a clear difference between the number of performed comparisons and execution time between the Naïve approach and the Indexing approach. The number of comparisons made by the Naïve approach are more making its execution time also higher. This is a brute force approach in the sense that it exhaustively performs all pairwise comparisons between lines or documents, checks the similarity against the given threshold and presents the pairs that have a similarity more than or equal to the same. So by principle, it performs n(n-1)/2 comparisons and has O(n^2) approach to similarity joining.

The second approach uses the most common choice to measure the similarity between two sets A and B i.e. the Jaccard index J (A, B) = |A∩B|/|A□ B|. We have indexed only the first |no. of words in each doc| - ⌈Jaccard similarity *|no. of words in each doc|⌉ + 1 words of each document d. This approach guarantees not missing any similar documents. This allows for comparisons between non-similar documents to be skipped thereby reducing the number of comparisons and resultantly cutting down the execution time considerably.