Massive Data Analysis Assignment #2

Li Shi Date: 3/5/2017

Terminal Workflow

- Creating new package in terminal

```
[cloudera@quickstart ~]$ cd workspace/Lishi
[cloudera@quickstart Lishi]$ mkdir input
[cloudera@quickstart Lishi]$ mkdir output
```

- Sampling: We only test on a sample of input files in order to decrease execution times. Choose first 200 lines of the file and named it as `pg100_sample.txt`.

```
hadoop fs –put workspace/Lishi/input/pg100_sample.txt
hadoop fs –put workspace/Lishi/input/StopWord.txt
```

-Now files are stored in HDFS, after implement 2 classes in Eclipse, export the jar file and run in the terminal.

Part A) Data Preprocessing

Question 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.**

We will put the file StopWords.txt in HDFS and add some parameters in the driver, so that we are able to load the stopwords file directly from the command line. Now, we just need to specify the argument -skip stopwordsfileHDFSpath to the hadoop jar command, as follows:

```
hadoop jar Lishi.jar Lishi.mdp.1_with_frequency
input/pg100_sample.txt output –skip input/StopWords.txt
```

In order to handle the -skip method, we also need to add some additional configurations in the driver:

```
public int run(String[] args) throws Exception {
```

```
        Job job = Job.getInstance(getConf(), "1_with_frequency");

        for (int i = 0; i < args.length; i += 1) {
            if ("-skip".equals(args[i])) {
                job.getConfiguration().setBoolean(
                    "skip.patterns", true);
                i += 1;
                job.addCacheFile(new Path(args[i]).toUri());
                LOG.info("Added: " + args[i]);
            }

        }
```

2. In the following code, two private void methods were setup to load file and to parse it.

```
public static class Map extends
        Mapper<LongWritable, Text, LongWritable, Text> {
    private Set<String> patternsToSkip = new HashSet<String>();
    private BufferedReader fis;

    protected void setup(Mapper.Context context) throws
IOException,
            InterruptedException {
        if (context.getInputSplit() instanceof FileSplit) {
            ((FileSplit)
context.getInputSplit()).getPath().toString();
        } else {
            context.getInputSplit().toString();
        }
        Configuration config = context.getConfiguration();
        if (config.getBoolean("skip.patterns", false)) {
            URI[] localPaths = context.getCacheFiles();
            parseSkipFile(localPaths[0]);
        }
    }

    private void parseSkipFile(URI patternsURI) {
        LOG.info("Added:" + patternsURI);
        try {
            fis = new BufferedReader(new FileReader(new File(
                patternsURI.getPath()).getName())));
            String pattern;
            while ((pattern = fis.readLine()) != null) {
                patternsToSkip.add(pattern);
            }
        } catch (IOException ioe) {
            System.err
                    .println("Caught exception while parsing the
cached file '"
                        + patternsURI
```

```
                              + "' : "
                              + StringUtils.stringifyException(ioe));
                }

        }
```

3. Remove the special characters and empty lines: In the Map function `[^A-Za-z0-9]` in order to keep special characters (keep only [a-z],[A-Z] and [0-9]) **((in green labeled area)..** Also special characters, empty words and empty line were removed in **if() loop (in yellow labeled area).**

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

        for (String word : value.toString().split("\\s*\\b\\s*"))
{

            Pattern p = Pattern.compile("[^A-Za-z0-9]");

            if (value.toString().length() == 0
                    || word.toLowerCase().isEmpty()
                    || patternsToSkip.contains(word.toLowerCase())
                    || p.matcher(word.toLowerCase()).find()) {
                continue;
            }

            context.write(key, new Text(word.toLowerCase()));
        }
    }

}
```

4) In the reducer:
In **Green labeled** area, this part shows how to keep each unique word only once per line through `HashSet` function.

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

    private BufferedReader reader;

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

        ArrayList<String> wordsL = new ArrayList<String>();
```

```java
        HashMap<String, String> wordcount = new HashMap<String,
String>();
        reader = new BufferedReader(
            new FileReader(
                new File(

  "/home/cloudera/workspace/Lishi/output/StopWords.txt")));
        String pattern;
        while ((pattern = reader.readLine()) != null) {
            String[] word = pattern.split(",");
            wordcount.put(word[0], word[1]);
        }

        for (Text word : values) {
            wordsL.add(word.toString());
        }

        HashSet<String> wordsHS = new HashSet<String>(wordsL);
```

Question 2:(1) Store on HDFS the number of output records (i.e., total lines).

The Output is COUNTERLINE.TXT. The following code is show how to create the counter:

```java
    public static enum COUNTER {
        COUNTERLINE,
    };

job.waitForCompletion(true);
        long counter =
job.getCounters().findCounter(CUSTOM_COUNTER.COUNTERLINE)
            .getValue();
        Path outFile = new Path("COUNTERLINE.txt");
        BufferedWriter br = new BufferedWriter(new
OutputStreamWriter(
            fs.create(outFile, true)));
        br.write(String.valueOf(counter));
        br.close();

        return 0;
```

We increment the counter by 1 by using the method as followed:

```java
context.getCounter(COUNTER.COUNTERLINE).increment(1);
```

Question 3: (7) Order the tokens of each line in ascending order of global frequency.

Firstly, we do basic wordcount to get the global frequency of each words without having computing a second time. Call a HashMap function to get the count of given words using **WordCount.get** function. After we get the output, we can do a sorting in the reducer. We sort the list by comparing integer (with prefix #). The function.

```java
HashMap<String, String> WordCount = new HashMap<String, String>();
        reader = new BufferedReader(
                new FileReader(
                        new File(

    "/home/cloudera/workspace/Lishi/output/WordCount.txt")));
        String pattern;
        while ((pattern = reader.readLine()) != null) {
            String[] word = pattern.split(",");
            wordcount.put(word[0], word[1]);

        }



HashSet<String> wordsHS = new HashSet<String>(wordsL);

        StringBuilder countedword = new StringBuilder();

        String firstprefix = "";
        for (String word : wordsHS) {
            countedword.append(firstprefix);
            firstprefix = ", ";
            countedword.append(word + "#" + WordCount.get(word));
        }

java.util.List<String> countedword_xx= Arrays

    .asList(countedword.toString().split("\\s*,\\s*"));

        Collections.sort(countedword_xx, new Comparator<String>()
{
            public int compare(String o1, String o2) {
                return extractInt(o1) – extractInt(o2);
            }

            int extractInt(String s) {
                String num = s.replaceAll("[^#\\d+]", "");
                num = num.replaceAll("\\d+#", "");
                num = num.replaceAll("#", "");
                return num.isEmpty() ? 0 : Integer.parseInt(num);
            }
```

Part B) Set Similarity Joint

Question 4: (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.

There are three ways to do pairwise comparison.

Brief introduction:

1) Naïve approach: perform all pairwise comparison and output similar pairs.
2) Second approach: group together documents that have at least one common word and then perform comparisons only between those pairs.
3) **Indexing approach: skip some words below similarity threshold (this threshold is deisgned by user), and then only compare those words upon this threshold.**

Since we do not need frequency, so I did another word without frequency input through the **code sheet two_without_frequency. JAVA.** I use **replaceAll** method to output word without associated frequency. The **output is two_without_frequency.TXT.**

Then, Since comparing the (doc2,doc1) is identical as (doc1,doc2), we need to remove duplicated comparing to eliminate unnecessary running time. The following code is a way to define a class named **Pairing** which is implemented with **WritableComparable** class for pair keys.

```
public class Pairing implements WritableComparable<Pairing> {

    private Text first;
    private Text second;

    public Pairing(Text first, Text second) {
      set(first, second);
    }

    public Pairing() {
      set(new Text(), new Text());
    }

    public Pairing(String first, String second) {
      set(new Text(first), new Text(second));
    }

    public Text FFirst() {
```

```java
        return first;
    }

    public Text SSecond() {
        return second;
    }

    public void set(Text FFirst, Text SSecond) {
        this.first = first;
        this.second = second;
    }


    @Override
    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        first.readFields(in);
        second.readFields(in);
    }

    @Override
    public String SString() {
        return first + " " + second;
    }

    @Override
    public int compareTo(Pairing 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 thatSmaller;

        Text thisBigger;
        Text thatBigger;

        if (this.first.compareTo(this.second) < 0) {
            thisSmaller = this.first;
```

```
                thisBigger = this.second;
        } else {
            thisSmaller = this.second;
            thisBigger = this.first;
        }

        if (other.first.compareTo(other.second) < 0) {
            otherSmaller = other.first;
            otherBigger = other.second;
        } else {
            otherSmaller = other.second;
            otherBigger = other.first;
        }

        int cmpThisSmallerOtherSmaller =
thisSmaller.compareTo(otherSmaller);
        int cmpThisBiggerOtherBigger =
thisBigger.compareTo(otherBigger);

        if (cmpThisSmallerOtherSmaller == 0) {
            return cmpThisBiggerOtherBigger;
        } else {
            return cmpThisSmallerOtherSmaller;
        }

    }
```

Secondly, after we load preprocessed two_without_frequency.TXT into **Hashup<>** object
(yellow area), we create a TreeSet<> which take words value list of the doc2 in the key pair of
(doc1, doc2) **(green area).** The Treeset<> class provides an implementation of the set interface
that uses the tree as storage and sorted according to the nature ordering of its elements in
TreeSet. Then, we create a TreeSet which take string value list of doc2 in the key pair of
(doc1,doc2) **(grey area).** Finally, we use the Jaccard similarity between 2 TreeSets and save the
similar pairs on HDFS **(white area).** Also, and the number of performance is also saved as txt
format on HDFS.

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

        HashMap<String, String> linesHP = new HashMap<String,
String>();
        reader = new BufferedReader(
                new FileReader(
                        new File(
    "/home/cloudera/workspace/Lishi/output/two_without_frequency.
txt")));
```

```java
        TreeSet<String> wordsoflinetreeset = new
TreeSet<String>();
        String wordsoflinetreeset = linesHP.get(key.getSecond()
            .toString());
        for (String word : wordsoflinetreeset.split(" ")) {
          wordsofline.add(word);
        }


        TreeSet<String> wordstreeset = new TreeSet<String>();

        for (String word :
values.iterator().next().toString().split(" ")) {
            wordsTS.add(word);
        }




        String pattern;
        while ((pattern = reader.readLine()) != null) {
          String[] line = pattern.split(",");
          linesHP.put(line[0], line[1]);
        }


context.getCounter(COUNTER.COUNTER_COMPARISIONS_ONE).increment(1)
;
        double sim = jaccard(wordtreeset, wordsoflinetreeset);

        if (sim >= 0.8) {
            context.write(new Text("(" + key.getFirst() + ", " +
key.getSecond() + ")"),
                    new Text(String.valueOf(sim)));
        }

      }

    }

}
```

Question 5: (40) Create an inverted index, only for the first $|d| - \lceil t\,|d|\rceil + 1$ 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.

In the map class:

$|d| - \lceil t\,|d|\rceil + 1$ of each document is represented as

```
long keepedwords = Math.round(wordsLong.length- (wordsLong.length *
0.8) + 1);
```

In here: t=0.8. Then the inverted index for selected words is output.

In the reducer:

The Jaccard similarity is computed

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

        private BufferedReader reader;

        public double jaccard(TreeSet<String> s1, TreeSet<String> s2)
{

            if (s1.size() < s2.size()) {
                TreeSet<String> jac = s1;
                jac.retainAll(s2);
                int inter = jac.size();
                s1.addAll(s2);
                int union = s1.size();
                return (double) inter / union;
            } else {
                TreeSet<String> jac = s2;
                jac.retainAll(s1);
                int inter = jac.size();
                s2.addAll(s1);
                int union = s2.size();
                return (double) inter / union;
            }
```

The following is the key pair processing.

And TXT file is output on HDFS with similarity over 0.8 **(grey area).**

```java
            for (Text word : values) {
            wordsLong.add(word.toString());
            }

        if (wordsLong.size() > 1) {
            ArrayList<String> pairs = new ArrayList<String>();
            for (int i = 0; i < wordsLong.size(); ++i) {
                for (int j = i + 1; j < wordsLong.size(); ++j) {
                    String pair = new String(wordsLong.get(i) + " "
```

```
                          + wordsLong.get(j));
                  pairs.add(pair);
              }
          }

          for (String pair : pairs) {
              TreeSet<String> pair1st = new TreeSet<String>();
              String pair1stS = linesLS
                      .get(pair.split(" ")[0].toString());
              for (String word : pair1stS.split(" ")) {
                  pair1st.add(word);
              }

              TreeSet<String> pair2nd = new TreeSet<String>();
              String pair2ndS = linesLS
                      .get(pair.split(" ")[1].toString());
              for (String word : pair2ndS.split(" ")) {
                  pair2nd.add(word);
context.getCounter(COUNTER.COUNTER_COMPARISIONS_TWO).increment(1)
;
              double sim = jaccard(pair1st,pair2nd);

              if (sim >= 0.8) {
                  context.write(new Text( pair.split(" ")[0] + ",
"
                          + pair.split(" ")[1] +),
                          new Text(String.valueOf(sim)));
```

Question 5: (10) Explain and justify the difference between a) and b) in the number of performed comparisons, as well as their difference in execution time.

The execution time of a) and b)

MapReduce Job job_1489424115020_0005

	Job Overview
Job Name:	SetSimilarityJoins_Questiontwo
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Mar 13 10:35:45 PDT 2017
Started:	Mon Mar 13 10:36:00 PDT 2017
Finished:	Mon Mar 13 10:36:31 PDT 2017
Elapsed:	31sec
Diagnostics:	
Average Map Time	12sec
Average Shuffle Time	11sec
Average Merge Time	0sec
Average Reduce Time	1sec

The number of performed comparison: **a)11446 b)80**

The execution time of similarity: **a)57sec b)12sec**

The second method is obviously **faster** than first one. The reason is because second method targets on reducing the number of compared words, which use more time to compute. Also, the first method has much **more comparison** than second method; this is because the latter method only uses less frequency words for doc in the corpus to do indexing. The purpose of this step is to eliminate large amount of unnecessary non-similar doc comparison and only keep similar doc to do further comparing.