Brandeis University
Department of Computer Science
COSI 129a - Introduction to Big Data Analysis
Fall 2016

Assignment 2: Hadoop on the Cluster.

The aim of this assignment is to get some hands-on experience with Hadoop on a multi-user cluster. You will write some simple Java processing jobs on a Wikipedia dump file using the *hadoop* library. The output of this assignment will be used for the next assignment of this course.

1 Resources

The english wikipedia dump file is provided at */shared* on HDFS of both clusters. Note that this wikipedia file is not exactly the same dump file that you could download online. The size of the file is 46GB. You should not make any changes to the wikipedia dump file on the clusters as it is shared among the groups, but use it as input for your map-reduce job. Don't duplicate the file anywhere in the cluster, as it would consume significant disk resources. You will also work with the following resources provided on the cs129a account:

1. /home/o/class/cs129a/assignment2/wiki-small
2. /home/o/class/cs129a/assignment2/people.txt
3. /home/o/class/cs129a/assignment2/framework

The first two were also used for assignment 1. You should first test your code on wiki-small on your laptop, using the workflow you set up for assignment 1. The third item above contains a code framework, which you are free to ignore as long as you know what you are doing and you have permission from the grading TA.

If you want to use *cloud9*, a basic library to work with wikipedia dump file, we provide it together with the setup for hadoop cluster as a *jar* lib. The library could help you to parse the dump file into individual articles. You could have a look at the library reference at Cloud 9 Library. Another option is to write your own wikipedia parser, using RecordReader (Record Reader introduction)

2 Problem

You need to extract from the dump file all articles about people listed in *people.txt*. Then you have to process the data (tokenizing, removing stop words, indexing), and create an index, which you will use in the next assignment, when you extract some meaningful information from big data employing machine learning.

2.1 Section A: Get the people article files

The dump file is provided in the XML format (check here if you don't know what XML is). You are required to get all articles that contain a name in the provided people list, and save them a separate file. Example of an article header in the wikipedia dump file:

```
<page>
    <title>Albert Gore</title>
    <id>617</id>
    ...
</page>
```

You could create your own way to get the article, or to feed the wikipedia dump file into a convenient library called *cloud9*. You are asked to write your implementation in *articles.GetArticlesMapred.java*:

```
public class GetArticlesMapred {
   public static class  GetArticlesMapper extends
      Mapper<LongWritable, edu.umd.cloud9.collection.wikipedia.WikipediaPage, Text, Text> {
      \\TODO: Implement mapper
   }
}
```

For this task, you're not required to write a Reducer. If you want to write your own implementation of WikipediaPage, you need to get permission from TAs (as it would affect the grading process).

2.2 Section B: Tokenization

2.2.1 Tokenization and cleaning

Generally, a wikipedia article has a number of style markings, hyperlink markings and other meta data inside. You want to get rid of them in order to acquire higher quality data for information extraction tasks.

> For the purpose of [[Biological reproduction—reproduction]] most amphibians require [[fresh water]]. A few (e.g. "[[Fejervarya raja]]") can inhabit [[brackish water]] and even survive (though not thrive) in [[seawater]], but there are no true [[marine (ocean)—marine]] amphibians. Several hundred frog species in [[adaptive radiation]]s (e.g., "[[Eleutherodactylus]]", the Pacific Platymantines, the Australo-Papuan microhylids, and many other tropical frogs), however, do not need any water for [[breeding in the wild]].
> ... {{Link GA|es}}
> {{Link FA|ca}}
> {{Link FA|fi}}
> ...

For each data file, tokenize it into a list of tokens, by splitting on special characters. Generally, you might want to split only on space character, but other characters might need to be considered as token boundaries as well, such as –, — and |. Periods and comma need to be split from tokens as well.

At the same time, you have to clean the data, keeping only relevant data. Your Wikipedia file cleaner should include but not be limited to the following features:

- Remove **bold**, *italic* and page link markers.
- Remove infoboxes, images, sound links, references etc.
- Remove numeric expressions: dates, numbers.

2.2.2 Token normalization

Normalizing the token might involve the following steps:

- Change the string form to lowercase
- Lemmatizing, for example: plurality removal *nations* → *nation*; gerund form of verb to infinitive form *writing* → *write*. You are encouraged to try writing a simple (rule-based) lemmatizer by yourselves, or experimenting with some text processing libraries, such as Stanford CoreNLP.

2.2.3 Stop word removal

Generally speaking, the frequencies of words/lemmas in a corpus are said to follow Zipf's law. Some of the highly frequent words such as articles (*the*, *a*, *an*) and conjunctions (*and*, *or*) should be excluded from the index, because they don't have much semantic content, while taking a lots of indexing space. A good strategy to find these stop words is to run Word Count over the the data and exclude the most frequent words. You could also use a list of stop words provided elsewhere online. The details of implementation for stop words removal will be left for you to decide.

2.2.4 Code framework

You are asked to implement your tokenizer in *lemma.Tokenizer*:

```
public class Tokenizer {
   public List<String> tokenize(String sentence){
      \\TODO: Implement tokenizing
   }
}
```

2.3 Section C

2.3.1 Lemma Indexing

After processing the articles files, we could build an indexing on people articles in the following format (bag of words format):

```
ARTICLE_INDEX -> ARTICLE_TITLE: (LEMMA_FREQ)*
LEMMA_FREQ -> <LEMMA, FREQUENCY_OF_LEMMA_IN_ARTICLE>
```

In short, the result of indexing will have multiple lines. Each line is an index corresponding to one article, containing the frequencies of all the lemmas in that article. For example, if a document has *title* = *Dogg_Catt* and has text content **woof meow woof woof**, the index of that document would be:

```
   Dogg_Catt <woof, 3> <meow, 1>
```

Your result index from now on will be called ARTICLE_LEMMA_INDEX. Again, assuming that you use *cloud9*, you are asked to write your implementation in the file *lemma.LemmaIndexMapred.java*:

```
public class LemmaIndexMapred {
   public static class  LemmaIndexMapper extends
      Mapper<LongWritable, WikipediaPage, Text, StringIntegerList> {
      \\TODO: Implement mapper
   }
}
```

Note that StringIntegerList.java is a supported class that implement *org.apache.hadoop.io.Writable*. Every thing you can write and read from file using MapReduce is required to implement that interface. In short, it enables you to write a list of pairs $\langle String, Integer \rangle$ (used for indexing) by Map Reduce. If you want to write your own implementation of index, or just use Text, you need to get permission from a TA.

2.3.2 Inverted Indexing

Inverted index could be built from ARTICLE_LEMMA_INDEX. Your inverted index need to follow the following format:

```
   INVERTED_INDEX -> LEMMA: (ARTICLE_FREQ)*
   ARTICLE_FREQ -> <ARTICLE_TITLE, FREQUENCY_OF_LEMMA_IN_ARTICLE>
```

For example, given that a lemma **woof** occurs 3 times in document *Dogg_Woof*, and occurs 1 times in document *Catt_Meow*, the inverted index should be:

```
  woof <Dogg_Woof, 3> <Catt_Meow, 1>
```

You are asked to write your implementation in *inverted.InvertedIndexMapred.java*:

```
public class InvertedIndexMapred {
   public static class  InvertedIndexMapper extends
      Mapper<Text, Text, Text, StringInteger> {
      \\TODO: Implement mapper
   }

   public static class InvertedIndexReducer extends MapReduceBase implements
Reducer<Text, StringInteger, Text, StringIntegerList> {
      \\TODO: Implement reducer
   }
}
```

3 Submission

We strongly suggest that within a week of getting this assignment you write a design document specifying how you would approach this task in a MapReduce way and that you email this document to one of the

TAs. An updated version of this report can be part of your report. You are required to submit a zip file containing the following materials:

- Your source code.
- Anything required to run your code on the cluster, including third-party libraries and stop word lists.
- A short report (about 2 pages) of your project. It should include:
  - Your overall design.
  - The timing of your run on your cluster.
  - Notes on any difficulties implementing your project.

4 Grading

- The quality of your source code (reflecting your understanding of the task; code has to be properly commented). (30%)
- Your source codes works on the cluster on some small inputs when we test them. (30%)
- The quality of your report (including the completion of your runs on the cluster). (40%)

Note. Again, though we give you a framework, your implementation could be different, but you need to get permission from a TA to make any change to that framework.

5 Tips

Break down your solution into small tasks. Think about each task with respect to the required inputs and outputs. You can then combine some tasks in one MapReduce job in order to save time and resource or you can implement a pipeline of MapReduce jobs each corresponding to a task.

It is noted that keeping a large lemma index in memory could result in heap space problem for JVM. Combining some steps could help to solve the problem, for example, stop words removal could be incorporated into the indexing step by ignoring an index if its size surpasses a certain threshold.