Lab Lecture #3

Introduction

Suppose we have a set of English text documents and wish to determine which document is most relevant to the query "*the brown cow*". A simple way to start out is by eliminating documents that do not contain all three words "*the*", "*brown*" and "*cow*", but this still leaves many documents. To further distinguish them, we might count the number of times each term occurs in each document and sum them all together; the number of times a term occurs in a document is called its **term frequency**. However, because the term "*the*" is so common, this will tend to incorrectly emphasize documents which happen to use the word "*the*" more, without giving enough weight to the more meaningful terms "*brown*" and "*cow*". Also the term "*the*" is not a good keyword to distinguish relevant and non-relevant documents and terms like "*brown*" and "*cow*" that occur rarely are good keywords to distinguish relevant documents from the non-relevant documents. Hence an **inverse document frequency** factor is incorporated which diminishes the weight of terms that occur very frequently in the collection and increases the weight of terms that occur rarely.

The *term count* in the given document is simply the number of times a given term appears in that document. This count is usually normalized to prevent a bias towards longer documents (which may have a higher term count regardless of the actual importance of that term in the document) to give a measure of the importance of the term t_i within the particular document d_j. Thus we have the **term frequency**, defined as follows.

$$tf_{ij} = \frac{n_{ij}}{N_j}$$

where n_{ij} is the number of occurrences of the considered term (t_i) in document d_j, and the denominator N_j is the sum of number of occurrences of all terms in document d_j (the document length).

The **inverse document frequency** is a measure of the general importance of the term (obtained by dividing the total number of document by the number of documents containing the term, and then taking the logarithm of that quotient).

$$idf_i = \log \frac{|D|}{M_i}$$

where:

- $|D|$ is total number of documents in the corpus;
- $M_i=|\{d : t_i \in d\}|$ is the number of documents where the term t_i appears in.

Then

$$\left(tf - idf\right)_{ij} = tf_{ij} \cdot idf_i$$

A high weight in *tf–idf* is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms. The *tf-idf* value for a term will always be greater than or equal to zero.

Consider a document containing 100 words wherein the word "*cow*" appears 3 times. Following the previously defined formulas, the term frequency (TF) for "*cow*" is then 0.03. Now, assume we have 10 million documents and "*cow*" appears in one thousand of these. Then, the inverse document frequency is calculated as ln(10000000/1000) = 9.21. The *tf-idf* score is the product of these quantities: 0.03 × 9.21 = 0.28.

Compute TF-IDF using MapReduce

Given a small collection of documents, we are going to implement *tf–idf* scores using Hadoop. We will need the following information:

- number of times term t_i appears in a given document (n)
- number of terms in each document (N)
- number of documents term t_i appears in (m)
- total number of documents ($|D|$)

We use multiple rounds of Map/Reduce to gradually compute *tf–idf*:

1. **Word frequency in document**: starting from a directory containing a set of text files, we will produce another set of files associating the couple (t_i, d_j) to the number n of times the term t_i appears in the document d_j.

   Mapper: input: (doc name, doc contents)
   output: ((word, doc name), 1)
   Reducer: sums counts for word in document
   outputs ((word, doc name), n)

2. **Word count in document**: starting from the directory containing the output of the previous job, we will produce another set of files associating the couple (t_i, d_j) to the couple (n, N), where N represents the total number of terms in the document d_j.

   Mapper: input: ((word, doc name), n)
   output: (doc name, (word, n))
   Reducer: sums frequencies n for individual terms in the same document
   outputs ((word, doc name), (n, N))

3. **Word frequency in collection**: starting from the directory containing the output of the previous job, we will produce another set of files associating the couple (t_i, d_j) to the triple (n, N, m), where m represents the number of documents the term t_i appears in the document d_j.

   Mapper: input: ((word, doc name), (n, N))
   output: (word, (doc name, n, N))
   Reducer: sums counts for word in collection
   output: ((word, doc name), (n, N, m))

4. **Calculate *tf-idf***: starting from the directory containing the output of the previous job, and assuming $|D|$ is known, we will produce another set of files associating the couple (t_i, d_j) its *tf-idf* score.

   Mapper: input: ((word, doc name), (n, N, m))
   output: ((word, doc name), *tf-idf*)
   Reducer: do nothing
   output: ((word, doc name), *tf-idf*)

Lab Lecture #3.1

Word frequency in document

This phase is designed in a job whose task is to count the number of words in each of the documents in the input directory. The mapper will receive as input a key that is, by default, the byte offset of the current line in the current file processed (`LongWritable` object) and a value that is the line read from the file (`Text` object). It must output another (key, value) pair. The problem is coding the (word, doc name) pair in a single object. While it is possible to implement a custom class to do the job, we will use a "string trick", emitting a simple string composed by the word, the special character "`@`" and the doc name. In order to obtain the document name from the `Context` object, use the following statement:

```
String fileName = ((FileSplit)context.getInputSplit()).getPath().getName();
```

Then, use the following statements to remove punctuation and other word anomalies:

```
Pattern p = Pattern.compile("\\w+");
Matcher m = p.matcher(value.toString());

while (m.find()) {
      String word = m.group().toLowerCase();
      // remaining code
}
```

During the mapper execution, each word in the line should be lower-cased, and ignored if it does not start with a letter or if it contains the character "_".

The reducer behaves as the standard, well-known `WordCount` reducer. In this case, keys are represented by `Text` objects, and values by `IntWritable` objects. The output will be a set of files (one per reducer): each line of each file will contain a word@document string, a tab character and an integer coded as string. Please remember that Hadoop requires the same classes for keys in the mapper output and the reducer input, as well as the same classes for relative values, although key and value classes can be different.

```java
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordFrequencyInDocument
{
   public static class NewMapper extends Mapper<LongWritable, Text, Text, IntWritable>
   {
      private final static IntWritable one = new IntWritable(1);

      public void map(LongWritable key, Text value, Context context)
         throws IOException, InterruptedException
      {
         // Compile all the words using regex
         Pattern p = Pattern.compile("\\w+");
         Matcher m = p.matcher(value.toString());

         // Get the name of the file from the inputsplit in the context
         String fileName = ((FileSplit) context.getInputSplit()).getPath().getName();

         // Build the values and write pairs through the context
         StringBuilder valueBuilder = new StringBuilder();
         while (m.find()) {
            String matchedKey = m.group().toLowerCase();

            // remove words starting with non letters, digits or containing other chars
            if (!Character.isLetter(matchedKey.charAt(0)) ||
                Character.isDigit(matchedKey.charAt(0)) ||
                matchedKey.contains("_"))
               continue;

            valueBuilder.append(matchedKey);
            valueBuilder.append("@");
            valueBuilder.append(fileName);

            context.write(new Text(valueBuilder.toString()), one);
            valueBuilder.setLength(0);
         }
      }
   }

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

      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);
      }
   }
   public static void main(String[] args) throws Exception
   {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "word frequency in document");
      job.setJarByClass(WordFrequencyInDocument.class);

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      job.setMapperClass(NewMapper.class);
      job.setCombinerClass(NewReducer.class);
      job.setReducerClass(NewReducer.class);

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

Lab Lecture #3.2

Word count in document

The goal of this phase is to count the total number of words for each document, in a way to compare each word with the total number of words. The mapper will receive as input a key that is, by default, the byte offset of the current line in the current file processed (`LongWritable` object) and a value that is the line read from the file (`Text` object). It must output another (key, value) pair. The problem is again coding the (word, *n*) pair in a single object. We will use the "string trick", emitting a simple string composed by the word, the special character "`/`" and a string representing the number of occurrences of the word in the document. In order to split a string `line` in an array of strings with a custom char you can use the following statement:

```
String[] tokens = line.split("@");
```

The reducer will receive two `Text` objects as keys and values representing the (doc name, (word, *n*) couple. It just needs to sum the total number of values in a document and pass this value over to the next step, along with the previous number of keys and values, as necessary data for the next step, writing in the files a line for each ((word, doc name), (*n*, *N*)) couple's couple ☺.

```java
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountInDocument
{
   public static class NewMapper extends Mapper<LongWritable, Text, Text, Text>
   {
      public void map(LongWritable key, Text value, Context context)
         throws IOException, InterruptedException
      {
         String[] wordAndDocCounter = value.toString().split("\t");
         String[] wordAndDoc = wordAndDocCounter[0].split("@");
         context.write(new Text(wordAndDoc[1]),
                       new Text(wordAndDoc[0] + "=" + wordAndDocCounter[1]));
       }
   }

   public static class NewReducer extends Reducer<Text, Text, Text, Text>
   {
      protected void reduce(Text key, Iterable<Text> values, Context context)
         throws IOException, InterruptedException
      {
         int sumOfWordsInDocument = 0;
         Map<String, Integer> tempCounter = new HashMap<String, Integer>();
         for (Text val : values) {
            String[] wordCounter = val.toString().split("=");
            tempCounter.put(wordCounter[0], Integer.valueOf(wordCounter[1]));
            sumOfWordsInDocument += Integer.parseInt(val.toString().split("=")[1]);
         }
         for (String wordKey : tempCounter.keySet())
            context.write(new Text(wordKey + "@" + key.toString()),
                          new Text(tempCounter.get(wordKey) + "/" + sumOfWordsInDocument));
       }
   }

   public static void main(String[] args) throws Exception
   {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "word count in document");
      job.setJarByClass(WordCountInDocument.class);

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(Text.class);

      job.setMapperClass(NewMapper.class);
      job.setReducerClass(NewReducer.class);

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

Lab Lecture #3.3

Word frequency in collection

```java
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordFrequencyInCollection
{
   public static class NewMapper extends Mapper<LongWritable, Text, Text, Text>
   {
      public void map(LongWritable key, Text value, Context context)
         throws IOException, InterruptedException {
         String[] wordAndCounters = value.toString().split("\t");
         String[] wordAndDoc = wordAndCounters[0].split("@");
         context.write(new Text(wordAndDoc[0]),
                       new Text(wordAndDoc[1] + "=" + wordAndCounters[1]));
      }
   }

   public static class NewReducer extends Reducer<Text, Text, Text, Text>
   {
      public void reduce(Text key, Iterable<Text> values, Context context)
         throws IOException, InterruptedException {
         // total frequency of this word
         Map<String, String> tempMap = new HashMap<String, String>();
         int numberOfDocumentsInCorpusWhereKeyAppears = 0;
         for (Text val : values) {
            String[] docAndCounter = val.toString().split("=");
            tempMap.put(docAndCounter[0], docAndCounter[1]);
            numberOfDocumentsInCorpusWhereKeyAppears++;
         }
         for (String docKey: tempMap.keySet())
            context.write(new Text(key.toString() + "@" + docKey),
                          new Text(tempMap.get(docKey) + "/" +
                              numberOfDocumentsInCorpusWhereKeyAppears));
       }
   }

   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "word frequency in collection");
      job.setJarByClass(WordFrequencyInCollection.class);

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(Text.class);
      job.setMapperClass(NewMapper.class);
      job.setReducerClass(NewReducer.class);

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

Lab Lecture #3.4

Calculate *tf-idf*

```java
import java.io.IOException;
import java.text.DecimalFormat;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordDocumentTfIdf
{
   public static class NewMapper extends Mapper<LongWritable, Text, Text, Text>
   {
      private static final DecimalFormat DF = new DecimalFormat("###.########");
      private static final int numberOfDocumentsInCorpus = 782;

      public void map(LongWritable key, Text value, Context context)
         throws IOException, InterruptedException
      {
         String[] wordAndCounters = value.toString().split("\t");
         String[] numbers = wordAndCounters[1].split("/");

         // Term frequency is the quotient of the number of terms
         // in document and the total number of terms in doc
         double tf = Double.valueOf(numbers[0]) / Double.valueOf(numbers[1]);

         // Inverse document frequency is the quotient between
         // the number of docs in corpus and number of docs the term appears
         double idf = (double)numberOfDocumentsInCorpus / Double.valueOf(numbers[2]);

         double tfIdf = tf * Math.log10(idf);

         context.write(new Text(wordAndCounters[0]), new Text(DF.format(tfIdf)));
       }
   }

   public static void main(String[] args) throws Exception
   {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "word frequency in collection");
      job.setJarByClass(WordDocumentTfIdf.class);

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(Text.class);

      job.setMapperClass(NewMapper.class);
      job.setReducerClass(Reducer.class);

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```