Big Data Analytics Final

Pubmed Central Topic Visualization

**Problem Statement: T**o produce a visualization of the topics that most commonly co-occur in the pubmed documents with a user provided term or phrase.

**Pre-processing the text:**

I tried different libraries to pre-process the text file.

**WEKA:**

The most common components you might want to use are

* *Instances* - your data
* *Filter* - for pre-processing the data
* *Classifier/Clusterer* - built on the processed data
* *Evaluating* - how good is the classifier/clusterer?
* *Attribute selection* - removing irrelevant attributes from your data

**Approach 1: Data Intensive text processing through Map Reduce “*Calculating Supervised Weighting on Hadoop*”**

“Supervised weighting [Debole and Sebastiani, 2003] have tested and compared some supervised weighting approaches that leverages on the training data. These approaches are variants of TFIDF weighting where the idf part is modified using common functions used to conduct feature selection.”

As per this paper <http://www.ijcai.org/papers/0304.pdf> it has been proved that supervised weighting is much better in performance than TF-IDF, no much documentation exists. I couldn’t proceed further.

**Approach 2: Data Intensive text processing through Map Reduce “*Calculating TF-IDF on Hadoop*”**

**Tf–idf**, short for **term frequency–inverse document frequency**, is a numerical statistic that is intended to reflect how important a word is to a [document](https://en.wikipedia.org/wiki/Document) in a collection or [corpus](https://en.wikipedia.org/wiki/Text_corpus). It is often used as a weighting factor in [information retrieval](https://en.wikipedia.org/wiki/Information_retrieval) and [text mining](https://en.wikipedia.org/wiki/Text_mining). The tf-idf value increases [proportionally](https://en.wikipedia.org/wiki/Proportionality_(mathematics)) to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general..

**Example:**

Suppose we have term frequency tables for a collection consisting of only two documents, as listed on the right, then calculation of tf–idf for the term "this" in document 1 is performed as follows.

|  |  |
| --- | --- |
| **Document 2** | |
| **Term** | **Term Count** |
| this | 1 |
| is | 1 |
| another | 2 |
| example | 3 |

|  |  |
| --- | --- |
| **Document 1** | |
| **Term** | **Term Count** |
| this | 1 |
| is | 1 |
| a | 2 |
| sample | 1 |

Tf, in its basic form, is just the frequency that we look up in appropriate table. In this case, it's one.

Idf is a bit more involved:

\mathrm {idf} ({\mathsf {this}},D)=\log {\frac {N}{|\{d\in D:t\in d\}|}}

The numerator of the fraction is the number of documents, which is two. The number of documents in which "this" appears is also two, giving

\mathrm {idf} ({\mathsf {this}},D)=\log {\frac {2}{2}}=0

So tf–idf is zero for this term, and with the basic definition this is true of any term that occurs in all documents.

A slightly more interesting example arises from the word "example", which occurs three times but in only one document. For this document, tf–idf of "example" is:

\mathrm {tf} ({\mathsf {example}},d_{2})=3

\mathrm {idf} ({\mathsf {example}},D)=\log {\frac {2}{1}}\approx 0.3010

\mathrm {tfidf} ({\mathsf {example}},d_{2})=\mathrm {tf} ({\mathsf {example}},d_{2})\times \mathrm {idf} ({\mathsf {example}},D)=3\times 0.3010\approx 0.9030

**Calculating Co-occurrence:**

A [co-occurrence](http://en.wikipedia.org/wiki/Co-occurrence) matrix could be described as the tracking of an event, and given a certain window of time or space, what other events seem to occur. Here “events” are the individual words found in the text. For example, considering the phrase “The quick brown fox jumped over the lazy dog”. With a window value of 2, the co-occurrence for the word “jumped” would be [brown, fox, over, the]. A co-occurrence matrix could be applied to other areas that require investigation into when “this” event occurs, what other events seem to happen at the same time. To build the text co-occurrence matrix, I tried with two methods. Pairs and Stripes algorithms

**Method 1: Pairs**

For each line passed in when the map function is called, we will split on spaces creating a String Array. The next step would be to construct two loops. The outer loop will iterate over each word in the array and the inner loop will iterate over the “neighbors” of the current word. The number of iterations for the inner loop is dictated by the size of our “window” to capture neighbors of the current word. At the bottom of each iteration in the inner loop, we will emit a WordPair object (consisting of the current word on the left and the neighbor word on the right) as the key, and a count of one as the value.

**Method 2: Stripes**

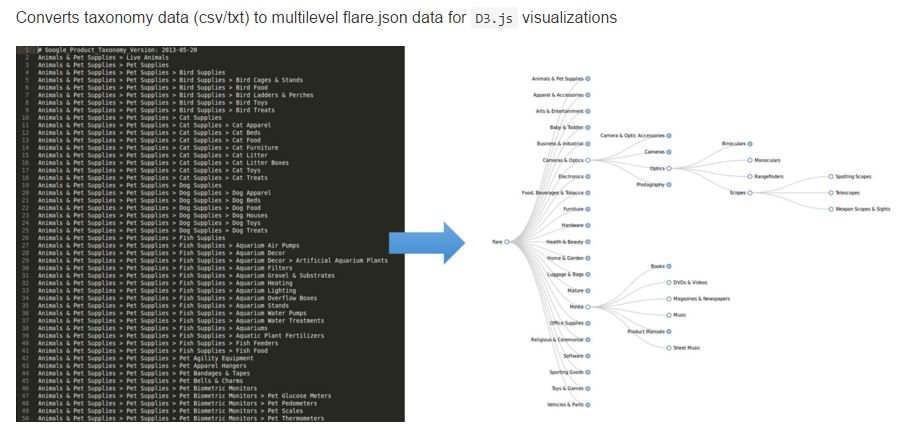
The approach is the same, but all of the “neighbor” words are collected in a HashMap with the neighbor word as the key and an integer count as the value. When all of the values have been collected for a given word (the bottom of the outer loop), the word and the hashmap are emitted. The Reducer for the Stripes approach is a little more involved due to the fact we will need to iterate over a collection of maps, then for each map, iterate over all of the values in the map.

**Analysis:**

When looking at the two approaches, we can see that the Pairs algorithm will generate more key value pairs compared to the Stripes algorithm. Also, the Pairs algorithm captures each individual co-occurrence event while the Stripes algorithm captures all co-occurrences for a given event. Both the Pairs and Stripes implementations would benefit from using a Combiner. Because both produce commutative and associative results, we can simply re-use each Mapper’s Reducer as the Combiner.

**Generating the flare Json:**

For data visualization, we need flare Json as the input format. I have used a library that converts the comma separated value to flare json format. This can be further used for D3 Visualization.



**D3 Data Visualization:**

I used the Bubble chart library for D3 visualization. Bubble charts encode data in the area of circles. Although less perceptually-accurate than bar charts, they can pack hundreds of values into a small space.

**Error Analysis: (Edge Cases)**

**The below is a test output for input “spawned”**

{

"name": "spawned",

"children": [

{

{"name": "tablets", "size": 3938},

{"name": "symptom", "size": 3812},

{"name": "Hierarchical", "size": 6714},

{"name": "Merge", "size": 743}

{"name": "spearman", "size": 1983},

{"name": "special", "size": 2047},

{"name": "undergo", "size": 1375},

{"name": "understand", "size": 8746},

{"name": "universe", "size": 2202},

]

}

}

I picked this word from one of the file. I could find some of the co-occurrence word in the json but I do find some other words along with it.

**Another example: In the below example, for the input “data”, the co-occurrences are repeated more than once.**

{

"name": "data",

"children": [

{

"name": "converters",

"children": [

{"name": "reliable", "size": 721},

{"name": "reliable", "size": 721},

{"name": "reliable", "size": 721},

{"name": "Count", "size": 781},

{"name": "Date", "size": 4141},

{"name": "Distinct", "size": 933},

{"name": "Expression", "size": 5130},

{"name": "Expression", "size": 5130},

{"name": "Fn", "size": 3240},

{"name": "DataConverter", "size": 1314},

{"name": "Converter", "size": 2220}

]

},

**Example 3: (No co-occurrences) But I believe there are co-occurrences for this word.**

{

"name": "university",

"children": [

]

},

**Below are some of the common occurences across the files:**

Big\_Data\_2015\_Jun\_1\_3(2)\_103-113.txt

------------------------------------------

schools procedure billed school fisher

Big\_Data\_2015\_Sep\_1\_3(3)\_173-188.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_203-208.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_59-66.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_138-147.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_198-202.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_59-66.txt

------------------------------------------

chws collection workers organizations health

Big\_Data\_2015\_Sep\_1\_3(3)\_173-188.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_203-208.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_189-192.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_193-197.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_198-202.txt

Big\_Data\_2015\_Mar\_1\_3(1)\_22-33.txt

------------------------------------------

\usepackage democracy \begin traps equations

Big\_Data\_2015\_Sep\_1\_3(3)\_138-147.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_173-188.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_59-66.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_103-113.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_193-197.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_138-147.txt

------------------------------------------

election watch \usepackage county counties

Big\_Data\_2015\_Mar\_1\_3(1)\_22-33.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_173-188.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_103-113.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_59-66.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_193-197.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_173-188.txt

------------------------------------------

synoptic retirement khp participant hrs

Big\_Data\_2015\_Sep\_1\_3(3)\_203-208.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_198-202.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_189-192.txt

Big\_Data\_2015\_Sep\_1\_3(3)\_193-197.txt

Big\_Data\_2015\_Jun\_1\_3(2)\_59-66.txt

**Approach 3: Using Spark**

The list of tokens becomes input for processing such as parsing or text mining. Although tokenization is a slow process. But, with the help of Spark we can make it fast by running it in chunks/parallel.

I tried to tokenize the words in a text file and count the number of times they occur in the text file (i.e., term frequency).

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

object TokenizerApp {

 def main(args: Array[String]) {

 val logFile = "src/data/sample.txt" // Should be some file on your system

 val sc = new SparkContext("local", "Tokenizer App", "/path/to/spark-0.9.1-incubating",

 List("target/scala-2.10/simple-project\_2.10-1.0.jar"))

 val logData = sc.textFile(logFile, 2).cache()

 val tokens = sc.textFile(logFile, 2).flatMap(line => line.split(" "))

 val termFrequency = tokens.map(word => (word, 1)).reduceByKey((a, b) => a + b)

 termFrequency.collect.map(tf => println("Term, Frequency: " + tf))

 tokens.saveAsTextFile("src/data/tokens")

 termFrequency.saveAsTextFile("src/data/term\_frequency")

 }

}

While processing “**textFile**” to obtain tokens, the “**logFile**” is split into 2 parts.

val tokens = sc.textFile(logFile, 2).flatMap(line => line.split(" "))

This makes processing of text file, faster as Spark can process these 2 parts of file in chunks to obtain tokens. So, more the number of splits, the faster execution is.

I could not proceed further due to lack of time.