

PDA - PRACTICA 1

Similarity between documents

Yusuf Hisil

In this practice, you are asked to implement some basic methods for the processing of text documents in order to be able to look for similarities between them. The notion of similarity between documents is complex and there are numerous approaches. In this practice we will use a fairly simple one. To begin with, we will try to find the coefficient of similarity between some books. What is asked in this first part of the practice is that you implement a similar program such that given two files tells us the similarity between them. To achieve this, we will use a measure called cosine similarity over the vectors of normalized frequencies of the figs, which we will describe below. To do this we want you to implement the following methods:

1. ****WORD FREQUENCY****: given a string, this method returns a List[(String, Int)], where each tuple holds a word that appears in the text, and its' absolute frequency.

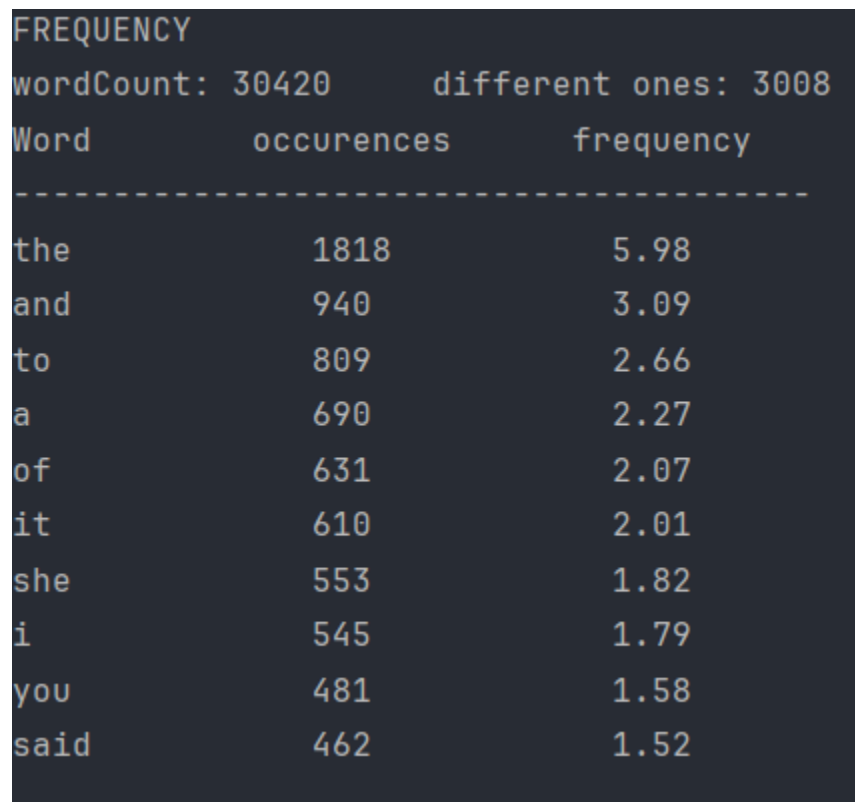
```
```
```

```
def freq(input: String): List[(String, Int)] = {
 input
 .toLowerCase()
 .replaceAll("[^a-zA-Z\\s]", " ")
 .split("\\s+")
 .groupBy(x=>x)
 .map(tuple => (tuple._1, tuple._2.length))
 .toList
 .sortWith((e1, e2) => e1._2 > e2._2)
}
```

```
```
```

The input is first normalized to have only lowercase characters and stripped of anything that isn't a letter or a white space, and finally split into words with a white space delimiter. Then, using `groupBy`, the method takes the list and transforms it into a map where the keys are the words and the values are arrays of the occurrences of the key. Using `.map()`, the method transforms the entries into (word, frequency), by counting the number of occurrences. Finally it's made into a list and sorted to easily display the 10 most frequent words.

Running this method on the file "pg11.txt" provides the following result:



```
FREQUENCY
wordCount: 30420      different ones: 3008
Word      occurrences  frequency
-----
the        1818         5.98
and         940         3.09
to          809         2.66
a           690         2.27
of          631         2.07
it          610         2.01
she         553         1.82
i           545         1.79
you         481         1.58
said        462         1.52
```

Word	occurrences	frequency
the	1818	5.98
and	940	3.09
to	809	2.66
a	690	2.27
of	631	2.07
it	610	2.01
she	553	1.82
i	545	1.79
you	481	1.58
said	462	1.52

2. **FREQ WITHOUT STOP-WORDS**: given a string, the method does the same thing as **freq**, but it strips the list of the stop-words found in the provided txt file.

```
def nonstopfreq(input: String): List[(String, Int)] = {  
  val source = Source.fromFile("resources/english-stop.txt")  
  val content = source.mkString.split("\\s+") // returns a list of all the stop words  
  source.close()  
}
```

```

    freq(input).filterNot(e1 => content.contains(e1._1))
  }
  ...

```

Running this method on the file "pg11.txt" provides the following result:

FREQUENCY WITHOUT STOPWORDS		
wordCount: 30420		different ones: 2624
Word	occurences	frequency

alice	403	1.32
gutenberg	93	0.31
project	87	0.29
queen	75	0.25
thought	74	0.24
time	71	0.23
king	63	0.21
turtle	59	0.19
began	58	0.19
tm	57	0.19

3. **WORD DISTRIBUTION**: this method takes a string as an input and returns a List[(Int, Int)] that holds the pair values (frequency, frequency frequency) which shows how often each frequency appears in the document.

```

...

def wordfreqfreq(input: String): List[(Int, Int)] = {
  input.toLowerCase()
    .replaceAll("[^a-zA-Z\\s]", " ")
    .split("\\s+")
    .groupBy(x => x)// => map[word, Array[Word]] where the second array holds every occurrence of
the key

```

```

    .map(tuple => (tuple._1, tuple._2.length))// map[word, frequency] by counting the number of
occurrences

    .groupBy(kv => kv._2)// map[freq, map[word, freq]]; turned into a map based on frequency, the
inner map holds a list of occurrences of each frequency

    .map{case (freq, wordlist) => (freq, wordlist.size)}// map[Int, Int] where the second is the freqfreq,
as counted by the length of the occurrences of each freq

    .toList

    .sortWith((e1, e2) => e1._2 > e2._2)
}
` ``

```

Running this method on the file "pg11.txt" provides the following result:

```
FREQFREQ
the 10 most frequent frequencies
1331 words appear 1 times
468 words appear 2 times
264 words appear 3 times
176 words appear 4 times
101 words appear 5 times
74 words appear 8 times
72 words appear 6 times
66 words appear 7 times
39 words appear 9 times
35 words appear 10 times

the 5 least frequent frequencies
1 words appear 68 times
1 words appear 178 times
1 words appear 227 times
1 words appear 940 times
1 words appear 100 times
```

4. **N-GRAMS**: given a String and an Int n, this method returns List[(String, Int)], where each string is an n-gram of the given size(n), and the int is the frequency with which it appears.

...

```
def ngrams(input: String, n: Int): List[(String, Int)] = {
  val s_stop = Source.fromFile("resources/english-stop.txt")
  val stop_words = s_stop.mkString.split("\\s+") // returns a list of all the stop words
  s_stop.close()
```

```

input
  .toLowerCase()
  .replaceAll("[^a-zA-Z\\s]", " ")
  .split("\\s+")
  .filterNot(stop_words.contains(_))// filters out the stop-words
  .sliding(n)// returns an iterator with a sliding window of n elements that goes over the Array[String]
  .toList
  .map(_._mkString(" "))// merges each array resulted from the iterator into a string, using " " as a
joint
  .groupBy(identity)// makes a map where (k,v) is (n-gram, occurrences)
  .map{case (ngram, occurrences) => (ngram, occurrences.size)}// counts the frequency of
occurrence
  .toList
  .sortWith((e1,e2) => e1._2 > e2._2)// sorts for easier displaying }
` ``

```

Running this method on the file "pg11.txt" with n=3 provides the following result:

```

NGRAMS
project gutenber tm          57
gutenberg tm electronic      18
project gutenber literary    13
literary archive foundation  13
gutenberg literary archive   13
tm electronic works          12
gutenberg tm license         8
full project gutenber        7
mock turtle alice            6
tm electronic work           6

```

5. **VECTOR SPACE MODEL**: this method takes two strings and an int as parameters: the strings are the paths to the documents to be compared and the int is the size of the n-gram.

```
`` `def cos_sim(book1: String, book2: String, n: Int): Double =  
{  
  //opens the documents and transforms them into Strings for easy processing  
  val s1 = Source.fromFile(book1)  
  val s2 = Source.fromFile(book2)  
  val c1 = s1.mkString  
  val c2 = s2.mkString  
  s1.close()  
  s2.close()  
  
  //gathers the ngrams from each document with the ngrams method  
  val words1: List[(String, Int)] = ngrams(c1, n)  
  val words2: List[(String, Int)] = ngrams(c2, n)  
  
  //maximum frequencies, used to calculate normalized frequencies later  
  val max_freq1: Int = words1.maxBy(_._2)._2  
  val max_freq2: Int = words2.maxBy(_._2)._2  
  
  // obtain a map of all the n-grams in both the documents, but only with the frequencies  
  // of the first document, so that they can be compared for all the words  
  val map1: Map[String, Int]  
= words1.toMap  
  val complete_map1 :Map[String, Int] = words2.map {  
    case (word, freq) => (word, map1.getOrElse(word, 0))  
  }.toMap ++ map1  
  
  val map2 = words2.toMap  
  val complete_map2: Map[String, Int] = words1.map {
```

```
case (word, freq) => (word, map2.getOrElse(word, 0))
}.toMap ++ map2
```

```
// turn the maps into vectors for cosinus similarity analysis
```

```
// the normalized frequency is calculated // the vectors are sorted by the words so that both
vectors have the same words on the same indices val a: Vector[(String, Float)] = complete_map1
```

```
.toVector
.map{
case (word, freq) => (word, freq.toFloat/max_freq1)
}
.sortBy(_._1)
```

```
val b: Vector[(String, Float)] = complete_map2
```

```
.toVector
.map {
case (word, freq) => (word, freq.toFloat / max_freq2)
}
.sortBy(_._1)
```

```
// calculate the components of the cosinus similarity formula separately, for enhanced readability
```

```
val dotProduct: Double = (a zip b).map { case (x, y) => x._2 * y._2 }.sum
```

```
val length1: Double = math.sqrt(a.map(x => x._2*x._2).sum)
```

```
val length2: Double = math.sqrt(b.map(x => x._2*x._2).sum)
```

```
// calculate and return the similarity index
```

```
dotProduct/(length1 * length2)
```

```
}
` ``
```


Using the ngram method from earlier, each document is processed for a List of (n-gram, frequency) pairs. Then the lists are turned into maps and merged together to ensure uniqueness of the words. The merging is done twice, each resulting map holding the frequencies of only one of the documents, the rest being 0. Then the maps are turned into vectors and sorted alphabetically for easy and correct calculation of the cosinus similarity, which is done in multiple steps for easier reading.

Running this on the files "pg11.txt" and "pg12.txt" with n=1, gives the expected result of 0.87637517.

The following table shows the results of running the method between 3 different files: pg11.txt, pg12.txt, and a copy of the book "Alice in Wonderland" found online at the link: <https://gist.github.com/phillipj/4944029>

	Pg11.txt/pg12.txt	Pg11.txt/online	Pg12.txt/online
N=1	0.8763751701061564	0.9516884285143099	0.8324168404928453
N=2	0.5033006526146986	0.7353000534682589	0.09597166781801553
N=3	0.39463078360602993	0.770303879616064	0.004438326190802574