Detection of similarity between documents

Axel Bengtsson
Department of Computer Science
University of Lund
axel.bengtsson@gmail.com

Ola Olsson
Department of Computer Science
University of Lund
ola@matematik.nu

Abstract

This document describes an implemented GUI application for detection of syntactic similarities between documents.

1 Introduction

Similarities between documents is interesting in many different kind of areas. The purpose can stretch from different kind of areas such as:

- Let the application choose articles such that we don't read the same kind of article twice.
- Easily detect cheating at assignments.
- Easily detect changes between two revisions of papers, source code etc.

2 How to detect syntactic similarities between documents

To detect similarities, we choose to implement a vector based algorithm called the cosine similarity. This algorithm lets all document represent a vector in the space. To see if two texts are equal or near equal, they should have a cosine similarity as near 1 as possible.

3 The program

Our program consists of three modules, Topic detection and tracking, TDT which is a module that counts the document vectors, LCS, the longest common subsequence counter and the third module which is the GUI.

The first module TDT, reads the articles and counts the word frequency. Then it calculates the weights of the words and counts the cosine similarity described above. All the words are included in this calculation except words described in "stoplist.txt". It returns an ordered list with all the pairs of articles in descending order of the rank. [1] A XML-file will be created at this time with all the weights and word frequency that the document contains. The name of the XML-file is the same as the filename and then the suffix ".XML" is added.

[1]This is the vector analysis number, where 1 is very close to each other and 0 is not.

The GUI starts up and calls the LCS with the two documents with highest rank. Every time the up/down button is pushed, a new call to LCS will be made.

3.1 Class diagram

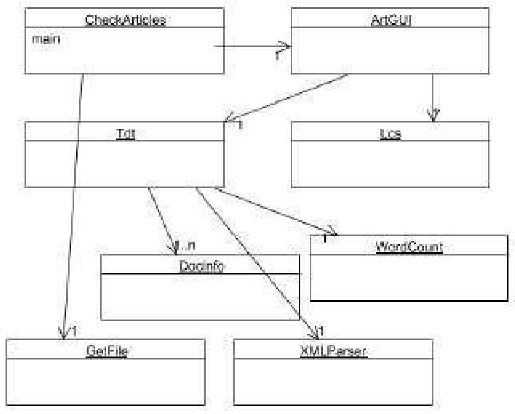


Figure 1: *Representation of the classes.*

3.2 TF-IDF

To give every word in the text a weight, we implemented the TF-IDF term frequency, inverted document frequency algorithm. This is seen as (Downie, 1997)

$$Weight(w_{ij}) = f_{w_{ij}} \times log_2 \left(\frac{N}{n_{w_i}} \right)$$

Term	Meaning
w_{ij}	i:th word in document j
$f_{w_{ij}}$	Frequency for word w_{ij}
N	Total number of documents
n_{w_i}	Number of documents w_i occurs in

This is trivial math but does explain some effects of the formula:

- If a word occur in every text, the weight for that word will be zero in every document.
- Two words can have different weights, it depends on which document it is in. The log term is constant in this sense but the frequency of the word in the document may differ.
- If a word only occur in one text that certain word will (of course) get $f_{ij} \times log(N)$ which is the biggest weight a word can get.

This means that, the more a word appears in all the texts, the less weight it will get. To get a high weight, the word should appear very often in as few texts as possible. When the TF-IDS has selected the two articles which is most equal, we are running these articles through a LCS algorithm. This algorithm detects which words are the same in both articles and paints these red in the GUI. Observe that the words found in the algorithm doesn't have to be consecutive. We have slightly modified the well known recursive algorithm in two ways, first we have made it iterative and secondly, we are running it through words instead of single characters.

3.2.1 Cosine similarity rate

After the TF-IDF algorithm, every word in every document has got a weight. Now we define the similarity between two documents as:

$$Rate(X,Y) = \frac{\sum_{w_i \in (X \cap Y)} x_{w_i} \times y_{w_i}}{\|X\| \times \|Y\|}$$

Term	Meaning
w_{ij}	i:th word in document j
X	A document
Y	A document
x_{w_i}	The weight of word w_i in X
y_{w_i}	The weight of word w_i in Y
$\|X\|$	$\sqrt{x_{w_1}^2 + x_{w_2}^2 + ...}$
$\|Y\|$	$\sqrt{y_{w_1}^2 + y_{w_2}^2 + ...}$

First, the program looks up all words which appears in both documents. The word weights are multiplied together and sums up for each word. This sum is divided by the product of both document norms. This formula will run through all $\binom{n}{2}$ pairs of documents.

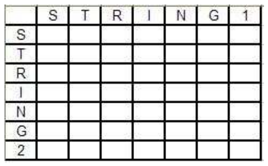
3.2.2 LCS

The LCS is often based upon a recursive algorithm. Imagine two strings, $X =< x_1,...,x_i >$ and $Y =< y_1,...,y_j >$. The recursive solution is based on the fact that if x_i and y_j is the same character, then the LCS of the two strings is the same as the LCS of $x_1,...,x_{i-1}$ and $y_1,...,y_{j-1}$ and then add x_i to the solution.
If $x_i \neq y_j$, we say that the LCS of X and Y is the maximum length of one of the two subproblems $LCS(X, y_1,...,y_{i-1})$ and $LCS(x_1,...,x_{i-1}, Y)$. This means that if the last two characters are not equal, the problem can be reduced to two subproblems, one subproblem that runs LCS with whole Y and deletes the last character from X and the other subproblem runs LCS with whole X and deletes the last character from Y. This follows cause we are comparing two strings and if their characters don't match, then one of the characters is worthless.

The base of the recursion is when the argument to the algorithm is the empty string, then the algorithm returns the empty string.

$$LCS(i,j) = \begin{cases} "" & if \quad i=0 \quad or \quad j=0 \\ LCS(i-1,j-1)+x_i & if \quad x_i = x_j \\ max(LCS(i-1,j),LCS(i,j-1)) & otherwise \end{cases}$$ [2]

However, this method is really slow because of the recursive steps. The worst case happens when $i = j$ and when the strings doesn' t have a common character at all. Of course, in this case the strings can have a maximum length of the number of characters in the alphabet. Anyway, the worst case calls the last row in (3.2.2) all the time which splits the problem in two parts. In the worst case, this call will be made i times. This means that the time complexity of the solution is $O(2^n)$ [3] which is totally unacceptable in our program. Of course some of these subproblems are the same and can easily be treated by memoization (probably using a hash) but we agreed on implement it iteratively. To find out how to make an iterative solution we can gain information from the recursive solution. If we think of the problem as a matrix:



	S	T	R	I	N	G	1
S							
T							
R							
I							
N							
G							
2							

The recursive solution starts at the right-bottom corner of the matrix and checks whether the charac-

[2]In this document and context the operator plus is overloaded as concatenation of strings, and the function max returns the argument which has the longest string.

[3]Here, n is the length which is equal to i and j

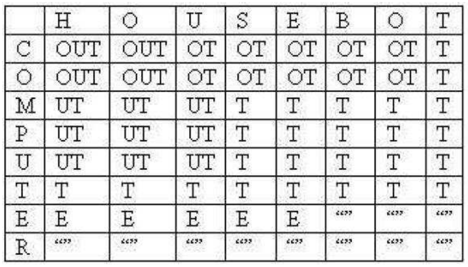
ters are the same or not. If they are the same, the cell$_{i,j}$ is equal to cell$_{i+1,j+1}$ + whats in cell$_{i,j}$. Else, the cell$_{i,j}$ will be equal to max($_{i+1,j\ i,j+1}$).

If we make a matrix of the two strings and follow the recursive solution of this matrix, then we simply see that the recursive solution can be written as two for-loops which starts in the right-bottom corner and works its way up to the left-upper corner where the solution will be stated.

```
for(int x=i;x>=0;++x)
{
for(int y=j;y>0;++y)
{
if (X[x]==0 || Y[y]==0) ResultMatrix[x][y]="";
if (X[x]==Y[y])
ResultMatrix[x][y] = ResultMatrix[x+1][y+1] + X[x];
else
ResultMatrix[x][y] = max(ResultMatrix[x][y+1],ResultMatrix[x+1][y]);


}
}
```

This means that a certain cell in the matrix is either a 0 which means that the letters are not equal, or the letter itself + the letter (or string) in the cell down one step and then one step to the right. This makes the solution a $O(n^2)$ in time which is much better than $O(2^n)$. An example of how our algorithm calculates the LCS of the strings "HOUSEBOT" and "COMPUTER".



	H	O	U	S	E	B	O	T
C	OUT	OUT	OT	OT	OT	OT	OT	T
O	OUT	OUT	OT	OT	OT	OT	OT	T
M	UT	UT	UT	T	T	T	T	T
P	UT	UT	UT	T	T	T	T	T
U	UT	UT	UT	T	T	T	T	T
T	T	T	T	T	T	T	T	T
E	E	E	E	E	E	“”	“”	“”
R	“”	“”	“”	“”	“”	“”	“”	“”

Figure 2: *Note that the result from this algorithm does not provide us a valid result for substrings of these strings. To get that, we should run the algorithm forward instead of backward.*

4 Screen shots and test runs

The program requires Java 2 Standard Edition 5.0 and can be started as follows:

```
java CheckArticles
```

A file chooser window will appear. You choose your multiple articles by pressing the control key (or shift). [4] The program reads and calculates data and a progress bar will guide you through this step. After this, a GUI will appear where the words appearing in both articles in same order are marked red. The articles that will be shown at the start of the GUI are the 2 articles of $\binom{n}{2}$ [5] that got the highest rating with our TF-IDF algorithm. When using the up/down buttons you will go through the documents in ascending/descending rank order.

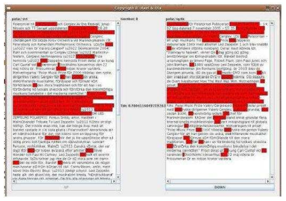


Figure 3: *Two documents are shown and the red marked text is the LCS.*

5 Quality assurance

To see whether our functions are work as they should we made some test cases and predicted the result before running them on our program. The main items we wanted to test is the cosine similarity function and the LCS.

To test them, we made three files containing this information [6]

File1

```
Hello everybody. this is a test Ola and Axel.
```

File2

```
Hello anyone. What may be the deal.
```

File3

```
Hello anybody. Great program Ola and Axel.
```

Before we ran these files in the program we expected four things.

1. The word "Hello" must get word weight 0 in every document because $log_2 1 = 0$.
2. File1 and File3 is the pair of files which should get the highest cosine similarity rate. This is because of the words "Ola and Axel".
3. File1 to File2 and File2 to File3 are the rest of pairs and these pairs are worth nothing.

[4] Files have to be chosen from the same directory.

[5] If n is the total number of documents, this will be the number of pairs

[6] Dont care about the semantics of the sentences, it is just test cases.

4. The LCS of File1 and File3 should be ”Hello Ola and Axel” because the other words isnt contained by the other string respectively.

We ran all the tests and the predicted results were correct.

Another test to run through the LCS is at text which contains a word only one, against the same text but reversed. The LCS algorithm should display one word in red. Imagine the text:

```
This is a test.
```

If we run this text against the reverse

```
.test a is This
```

, the red marked word could be any of the words in the two sentences depending on how one has implemented the *max* function. The important thing is to understand that not two words will be red. This is because of the reversing text. Suppose that the strings are build like $X = < x_1, x_2, ..., x_n >$, the other string Y will look like $< x_n, x_{n-1}, .., x_1 >$. Suppose we find a word which must be in our LCS. Say x_k where $k \in 1 \rightarrow n$. Then, this word must be the word y_{n-k} in Y. After some iteration we find a new word to be in our LCS, say x_l where $l > k$. This means that this word is found in y_{n-l} where $l < k$. Because $l < k$, it means that the second word is before the first word which is not possible in LCS.
The word which got red in our program was: ”a” This is because we didnt implement a max function ourselves.

6 Statistics

To get a good understanding of how good our application is, we tried it on several documents, some of them were intended to give some result while other documents were copied directly from newspapers. We chose two subjects from the newspapers that have been pretty large in the last weeks, namely ”wilma” and ”the bird disease”. 10 documents from each subject were collected from DN, Aftonbladet, Expressen, Sydsvenskan, TT.se and other big news sources. First, both of us, independently of each other ranked all the pairs of documents based upon the LCS and give the pairs a grade between 1-5. After that we compared our ratings and they were exactly the same but one or two pairs. We ran the articles through the program and the result was that almost every document got a TDT rating very good compared to our LCS rating except those pairs where one of the document was much larger than the other one.

7 Conclusions

We think that the assignment was perfect in time measure. We also liked the subject and we had absolutely no problem with coding whatsoever. The problem was to get the idea of how the TF-IDF works but Pierre explained it very good.

What we could conclude from the statistics is that the program is very good at finding similar documents when the papers are approximately in the same length. Otherwise, if one of the documents are a proper subset of the other but far more smaller, the TDT will be very low. Maybe it is good, maybe not, it depends on the purposes. It is now possible for a student to copy another students paper and keep on writing without the notice of the TDT, but the LCS will cover it, if the teacher will search through all the pairs graphically.

Something notable is also that it was the first time we used CVS and that worked very good as well.

8 Acknowledgments

We would like to thank Pierre for the help and for assistance with books and Rolf Karlsson who sent us the lecture notes of Dynamic Programming (Karlsson, 2005).

References

J. Stephen Downie. 1997. Term weighting: tf*idf. Webpage, September24 . `http://instruct.uwo.ca/gplis/601/week3/tfidf.html`.

R. Karlsson. 2005. Lecture 5: Dynami programming.