BU-ALI SINA UNIVERSITY

REPORT

Document Similarity (Using LCS)

Mehran Shahidi
Mehrdad Shahidi

supervised by
Dr. Moharam Mansorizade

July 30, 2020

Summary

In this project, we are going to impelement a similar document finder that will try to find similar document on the web based on the given document, and it's going to give a score how much the document are similar to each other.

https://github.com/m3hransh/similar_document_finder

Contents

1 Introduction **1**

2 Implementing Similarity Rater Using LCS **2**
2.1 Python Implementatoin 1
2.1.1 Runtime and Memory Complexity 2

3 Similarity Score **2**
3.1 Implementing Similarity Score in Python 3

4 Dependency Score **3**
4.1 Partial Inclusion 3
4.2 Implementing Inclusion Score in Python 4

5 Execution of Similarity and Inclusion Score using LCS **5**

1 Introduction

Similarity is a complex concept. In this project we are going to use LCS (Longest Common Sequence) to find some scores to decide how similar two documents are to each other. Then we improve the functionality and try to solve a harder problem of searching similar documents based on a document at hand on the Internet to see if there exists contents that are close or they contain in the document. The second part of the project needs web scraping that is a complex problem by itself. At first, to simplify the task we only look on some specified websites that we know the format of their contents and can extract those. Then we make the problem harder and try to find some optimized method to solve the problem in a general way.

2 Implementing Similarity Rater Using LCS

The **longest common subsequence** [1] (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from the longest common substring problem: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. For example, consider the sequences (ABCD) and (ACBAD). They have 5 length-2 common subsequences: (AB), (AC), (AD), (BD), and (CD); 2 length-3 common subsequences: (ABD) and (ACD); and no longer common subsequences. So (ABD) and (ACD) are their longest common subsequences.

Let two sequences be defined as follows: $X = (x_1 x_2 \cdots x_m)$ $X = (x_1 x_2 \cdots x_m)$ and $Y = (y_1 y_2 \cdots y_n)$ $Y = (y_1 y_2 \cdots y_n)$. The prefixes of X are $X_{1,2,\ldots,m}$ $X_{1,2,\ldots,m}$; the prefixes of Y are $Y_{1,2,\ldots,n}$ $Y_{1,2,\ldots,n}$. Let $LCS(X_i, Y_j)$ $LCS(X_i, Y_j)$ represent the set of longest common subsequence of prefixes X_i and Y_j. This set of sequences is given by the following.

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(x_{i-1}, y_{j-1})^\frown x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \tag{1}$$

2.1 Python Implementatoin

Using the LCS algorithm we can find the longest common sequence in the two or more strings. But applying this method to long documents can be irrelevant. Because in document similarity problems we mostly care how semantically those are the same and finding a common sequence of characters has little to do with that. Instead, we can split the documents into words and find the longest common sequence of words.

Now, lets implement this in Python. The LCS function is going to take two lists of strings and find LCS among those. Using a dynamic programming approach and with use of tabulation we solve the LCS problem.

As an example, lets say, we have two following lists of strings.

$$x = [Hey, Mehran, how, are, you, doing, today]$$

$$y = [Hey, Mehrdad, how, are, doing?]$$

Using recursive formula in the equation 1 and taking a bottom up approach we can fill a table similar to what shown in the 1 and find the LCS by returning the bottom right cell in the table.

Table 1: LCS of two lists. Blue colored list is the answer.

	$\emptyset$	**Hey**	**Mehran,**	**how**	**are**	**you**	**doning**	**today?**
$\emptyset$	[]	[]	[]	[]	[]	[]	[]	[]
Hey	[]	$['hey']$	$\leftarrow ['hey']$	$\leftarrow ['hey']$	$\leftarrow ['hey']$	$\leftarrow ['hey']$	$\leftarrow ['hey']$	$\leftarrow ['hey']$
Mehrdad,	[]	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey']$
how	[]	$\uparrow ['hey']$	$\uparrow ['hey']$	$['hey','how']$	$\leftarrow ['hey','how']$	$\leftarrow ['hey','how']$	$\leftarrow ['hey','how']$	$\leftarrow ['hey','how']$
are	[]	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey','how']$	$['hey','how','are']$	$\leftarrow ['hey','how','are']$	$\leftarrow ['hey','how','are]$	$\leftarrow ['hey','how','are']$
doing?	[]	$\uparrow ['hey']$	$\uparrow ['hey']$	$\uparrow ['hey','how']$	$\uparrow ['hey','how','are']$	$\uparrow ['hey','how','are']$	$\uparrow ['hey','how','are]$	$\uparrow ['hey','how','are']$

[1] https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

The code implemented in Python is shown in Listing 1. First, the table is filled with empty lists. Looping through the table, strting with second row and second column (first row and first column initialaized with empty before) using similar formula as equation 1 fill the table with approporiate values. *indx* function is used to map each index of texts with right index in table. Because the talble have one extra row and column to find the right index for the table we need to add 1 to the index of each character in the text. For convenience a *lambda* function is written for that.

```python
def LCS(text1, text2):
    '''
    text1, text2 : two string or list of strings.
    return: list of similar characters or strings.
    '''
    n = len(text1)
    m = len(text2)

    # create (n+1)*(m+1) table
    table = [[[]]*(m+1)]*(n+1)

    # map index of texts to table
    indx = lambda i: i+1

    for i in range(n):
        for j in range(m):
            if text1[i] == text2[j]:
                # LCS(i,j) = LCS(i-1, j-1) ^ text1[i]
                table[indx(i)][indx(j)] = table[indx(i-1)][indx(j-1)] + [text1[i]]
            else:
                # LCS(i, j) = max(LCS(i, j-1), LCS(i-1, j))
                table[indx(i)][indx(j)] = max(table[indx(i)][indx(j-1)], table[indx(i
    -1)][indx(j)], key= lambda x:len(x))

    return table[n][m]
```

Listing 1: Implementatoin of LCS in python

2.1.1 Runtime and Memory Complexity

The runtime complexity of function is shown in Listing 1 is $O(m \times n)$, because it needs to loop through the table of size $m \times n$. There is subtlty in calculating Memory Complexity. In Python, lists only have refrences to their elements and don't store whole elements. And in total there is $m \times n$ refrences in the table with number of LCS distinct strings refrences. That in the worst case gives $O(m \times n \times min\{n, m\})$.

3 Similarity Score

Now that we can find the LCS, how we can say how much the documents are similar? One of the methods is used in statistics to estimate similarity of two sets is **Jaccard similarity coefficient**. [2] The Jaccard index, also known as Intersection over Union and the Jaccard similarity coefficient (originally given the French name coefficient de communauté by Paul Jaccard), is a statistic used for gauging the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

There is only a subtlety here, in LCS we don't find the intersectons. So we can amend the forumla this way:

$$J'(A, B) = \frac{|LCS(A, B)|}{|A| + |B| - |LCS(A, B)|} \tag{3}$$

[2]https://en.wikipedia.org/wiki/Jaccard_index

3.1 Implementing Similarity Score in Python

Python implementation is shown in the Listing 2.

```python
def sim_rate(t1, t2, lcs):
    '''
    t1: list of words
    t2: list of words
    lcs: longest common sequence of t1 and t2
    return: similarity rate and including rate.'''
    rate = len(lcs)/(len(t1)+len(t2)-len(lcs))

    return rate
```

Listing 2: jaccard similarity rate

4 Dependency Score

Other way to look at similarity is to see how much a set includes elments of the other set compare to all the elements in the set. Let's say we have two documents and one of those is subset of the other. if the second document has lots of strings compare to the first one. This causes low smilarity score. So we introduce another score to recognize how much one document depends on other. So the score is defined as following:

$$S(A,B) = \frac{|LCS(A,B)|}{|A|} \tag{4}$$

shows the dependency of set A on set B.

4.1 Partial Inclusion

What if the documents are not the subset of each other nor similar but still there is some parts in them that are exactly the same. In LCS, we don't care about if the similar words occurs in dense or in a sparse space. As an example consdier two following random texts. The colored parts shows the parts that are similar. As you can see if the uncolored text increase the Similarity and Inclusion score is going to decrease, although they have exactly similar parts in them.

Text 1	Text 2
Both rest of know draw fond post as. It agreement defective to excellent. Feebly do engage of narrow. Extensive repulsive belonging depending if promotion be zealously as. Preference inquietude ask now are dispatched led appearance. Small meant in so doubt hopes. Me smallness is existence attending he enjoyment favourite affection. Delivered is to ye belonging enjoyment preferred. Astonished and acceptance men two discretion. Law education recommend did objection how old. **To sure calm much most long me mean. Able rent long in do we. Uncommonly no it announcing melancholy an in. Mirth learn it he given. Secure shy favour length all twenty denote. He felicity no an at packages answered opinions juvenile.**	**Both rest of know draw fond post as. It agreement defective to excellent. Feebly do engage of narrow. Extensive repulsive belonging depending if promotion be zealously as.**Performed suspicion in certainty so frankness by attention pretended. Newspaper or in tolerably education enjoyment. Extremity excellent certainty discourse sincerity no he so resembled. Joy house worse arise total boy but. Elderly up chicken do at feeling is. Like seen drew no make fond at on rent. Behaviour extremely her explained situation yet september gentleman are who. Is thought or pointed hearing he. **To sure calm much most long me mean. Able rent long in do we. Uncommonly no it announcing melancholy an in. Mirth learn it he given. Secure shy favour length all twenty denote. He felicity no an at packages answered opinions juvenile.**

Table 2: Two random texts. Colored parts are similar

For the complexity of recognizing this part we are not going to define a score, and only use diagrams to gain the insight about the existance of this. You can see the occurance distribution graph of LCS in Figure 1.

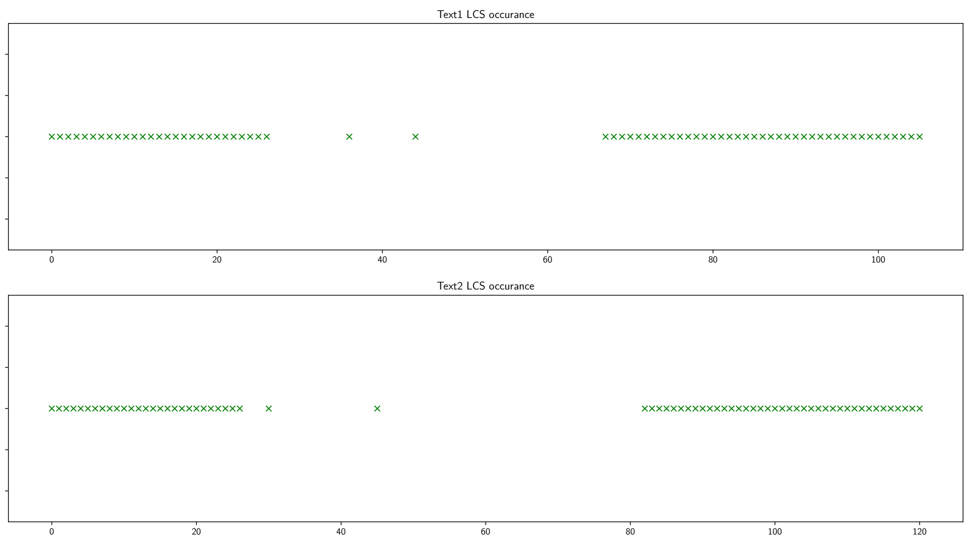

Figure 1: LCS occurance distribution of texts in Table 2

4.2 Implementing Inclusion Score in Python

Now let's implement the the Dependency Score using formula in the Equation 4. For the partial inclusion, the occurance indexes of LCS will be found to draw the the graph later in another part of the program. The code is shown in the Listing 3.

```python
def depend_score(t1, t2, lcs):
    '''
    t1: list of words
    t2: list of words
    lcs: longest common sequence of t1 and t2
    return: return dependency rate of t1 on t2'''
    return  len(lcs)/len(t1)


def dist_finder(text, words):
    '''
    text: list of words
    words: List of word occurances in text
    return: index of words occurances in text'''
    counter = 0
    dist_list = []

    for i, word in enumerate(text):
        if len(words) == counter:
            break
        if words[counter] == word:
            dist_list.append(i)
            counter += 1

    return dist_list
```

Listing 3: Inclusion Score

5 Execution of Similarity and Inclusion Score using LCS

In this section, we write a Python program that uses previous implementations. The program is going to read two texts from files that are given by system arguments through the execution terminal and is going to print the Similarity and Dependency score with the LCS strings. Afterward, will plot the distribution of LCS occurrence in both text1 and text2.

```python
#! /usr/bin/python3
import sys
import numpy as np
import matplotlib.pyplot as plt
from lcs import LCS, sim_rate, contain_rate, dist_finder


def plot_at_y(ax, arr, val, **kwargs):
    '''Plot 1-d scatter'''
    ax.plot(arr, np.zeros_like(arr) + val, 'x',color='green', **kwargs)
    ax.set_yticklabels([])


if __name__ == "__main__":
    # getting file names from Terminal args
    if len(sys.argv) >= 3:
        f_name1 = sys.argv[1]
        f_name2 = sys.argv[2]
    else:
        # default values of filenames
        f_name1 = 'text1.txt'
        f_name2 = 'text2.txt'

    #opening files
    with open(f_name1, 'r') as f1:
        with open(f_name2, 'r') as f2:
            # Read and split text to words
            t1 = f1.read().split()
            t2 = f2.read().split()

            lcs = LCS(t1, t2)
            sim = sim_rate(t1,t2,lcs)
            contain = contain_rate(t1, t2, lcs)

            print("Similarity Score: {:.2%}  \n"
                    "Inclusion Score: {:.2%}  \n"
                    "\nLCS Words: \n{}".format(sim, contain, ' '.join(lcs)))

            # Plotting distribution of LCS occurances in t1 and t2
            fig, (ax1, ax2) = plt.subplots(2)
            ax1.set_title("Text1 LCS occurance")
            ax2.set_title("Text2 LCS occurance")
            plot_at_y(ax1, dist_finder(t1, lcs), 0)
            plot_at_y(ax2, dist_finder(t2, lcs), 0)
            plt.show()
```

Listing 4: Python program using previous Implementation

The execution of the program is shown in Figure 2. *text1.txt* and *text2.txt* contains Text1 and Text2 that were shown in the Table 1. The plot is going to be similar to Figure 1. As you can see, the Similarity score of texts is 42.77% and Inclusion score (the rate the smaller text depends on the bigger text) is 64.15%.

Figure 2: Execution of program in Listing 4

```
$ ./main.py text1.txt text2.txt
Similarity Score: 42.77%
Dependency Score of text1 on text2: 64.15%
Dependency Score of text2 on text1: 56.20%

LCS Words:
Both rest of know draw fond post as.
It agreement defective to excellent. Feebly do engage of narrow.
Extensive repulsive belonging depending if promotion be zealously
as. so he To sure calm much most long me mean. Able rent long in
do we. Uncommonly no it announcingmelancholy an in. Mirth learn
it he given. Secureshy favour length all twenty denote.
He felicity noan at packages answered opinions juvenile.
```

If you have cloned the applications Git repository on GitHub, you can run `git checkout phase#1` to check out this version of the application.