Lecture 2 Project: Document Distance

Yanuar Heru Prakosa

31-05-2021

The Document Distance

The assignment from the Lecture 2 in 6-006 is related to one called document distance. Thus I need to find out first what is exactly document distance is. Form this MIT discussion webpage, I learn that document distance will be measured using vector equations. Let D be a text, then a word is a consecutive of alphanumeric characters. We will not distinguish upper case and lower case letters, but we use non alphanumeric characters as delimiter between words. For example the word "can't" consist of 2 words: can and t.

Okay now we go to the formulation: The word word frequency distribution of a document D ia a mapping from words w to their frequency count, denoted as $D(w)$. We can view the frequency distribution D as vector, with one component per possible word. Each component will be a non-negative integer ≥ 0.

The norm of this vector is defined by:

$$N(D) = \sqrt{D \cdot D} = \sqrt{\sum_{w} D(w)^2}$$

Alright this calculation still does not make any sense right now. But let's move on, I need to get to how to calculate document distance first. Now when we have two documents to be compared to each other, let's name them D and D'. The inner product between D and D' is defined as:

$$D \cdot D' = \sum_{w} D(w)D'(w)$$

So basically this is the sum of products on all word frequencies in two documents. Thus, if a word exist 1000 times in one document but never existed in other document the inner product of that part is 0!

Okay, since our objective is to define distance which is defined here as angle between two documents we need to go back to definition of vector dot products:

$$D \cdot D' = N(D)N(D')cos\theta$$

As we looking for an angle we are focusing on θ right? Therefore:

$$cos\theta = \frac{D \cdot D'}{N(D)N(D')}$$

$$\theta = arccos\left(\frac{D \cdot D'}{N(D)N(D')}\right)$$

Deriving from other equations above we will get:

$$\theta = arccos\left(\frac{\sum_w D(w)D'(w)}{\sqrt{\sum_w D(w)^2}\sqrt{\sum_w D'(w)^2}}\right)$$

From those formula we can infer that if both Documents are the same then the $angle(D, D') = \theta = 0$. On the other hand if both documents are completely different in the sense there are no similarity in terms of word used in it then the $angle(D, D') = \theta = \frac{\pi}{2}$. Please note we are using radians in this matter.

Pre Example

To make it clearer I think a simple practical example is mandatory. Let us have two sentences that we will use as documents. One is "To be or not to be" and the other is "Doubt truth to be a liar". We will calculate the distance between these two documents.

Now we already list all unique set of words and it can be seen in this table plus its frequencies:

word	D1(w)	D2(w)
to	2	1
be	2	1
or	1	0
not	1	0
doubt	0	1
truth	0	1
a	0	1
liar	0	1

Now let's get back to the formula:

$$\theta = arccos\left(\frac{\sum_w D(w)D'(w)}{\sqrt{\sum_w D(w)^2}\sqrt{\sum_w D'(w)^2}}\right)$$

we begin with the denominator part first.

To calculate the denominator we need to find $\sum_w D(w)^2$ and $\sum_w D'(w)^2$ first. Here is the results of the calculation:

word	D1(w)	D2(w)	D1(w)^2	D2(w)^2
to	2	1	4	1
be	2	1	4	1
or	1	0	1	0
not	1	0	1	0
doubt	0	1	0	1
truth	0	1	0	1
a	0	1	0	1
liar	0	1	0	1
		sums	10	6

As you can see the sums already being included in the table above. We can use it as our denominators, but remember those sum products must be subject to a square root operations. Now we need to calculate the nominator side: $sumD(w)D'(W)$, which we have already calculated also using spreadsheet:

word	D1(w)	D2(w)	D1(w)^2	D2(w)^2	D1(w).D2(w)
to	2	1	4	1	2
be	2	1	4	1	2
or	1	0	1	0	0
not	1	0	1	0	0
doubt	0	1	0	1	0
truth	0	1	0	1	0
a	0	1	0	1	0
liar	0	1	0	1	0
		sums	10	6	4

As you can see on the table above the $sumD(w)D'(W) = 4$, now we can put all of them in the formula:

$$\theta = arccos\left(\frac{4}{\sqrt{10}\sqrt{6}}\right)$$

$$\theta = arccos\left(\frac{4}{\sqrt{10.6}}\right)$$

$$\theta = arccos\,(0.52)$$

$$\theta = 1.028157225$$

There you are, that is in summary how we will measure the distance between to documents.

Summary

Here is what you should do when measuring the distance between two documents:

1. list all words in a set, meaning there are no double words in a set all words are unique
2. count the frequency of occurance for each word in each document (WARNING: EACH NOT TOTAL)
3. use vector inner product and normalization also dot products to calculate the angle (SEE THE FORMULAS AND EXAMPLE ABOVE)
4. that angle when is closer to 0 (the minimum angle is 0) then the chance both documents are the same is higher (some use this as an indication of plagiarism)
5. Vice versa when the angle is closer towards $\frac{\pi}{2}$ then the documents are less similar.
6. Just remember this is only a basic algorithm on how to measure document distance, many ways can be used to cheat this algorithm thus this kind of measurements always evolving in terms of algorithms.
7. Stay tune and keep learning!

1 Coding in Python

Now after we know how to formulate the document distance, we can start use it to formulate pseudocode. The algorithm will be mesured after the pseudocode is implemented into code in python. Each python file here uses different algorithm or should I say method on how to handle the words inside the documents. These words are what used to measure the document distance later on using vector normalized multiplication.

Well it is better if we research each python file to understand the way they were calculated. I need to have method in order to debug the code later on to make sense of the algorithms they use. However, preparing method to debug the code is not as simple as it might sounds. For once the built in VSCode debugger does not allow the *args input from the user. Meanwhile the Python own debug library is lacking in the user interface features.

If I need to choose between the two, right now I more lean towards the VSCode internal debugger. I must make some adjustment in the main function in order to put the *args into the system. I need to make the test document (t1 and t2) still passed into the argument after the main function is called.

```
1         def main():
2             if len(sys.argv) != 3:
3                 print ("Usage: docdist1.py filename_1 filename_2")
4             else:
5                 filename_1 = sys.argv[1]
6                 filename_2 = sys.argv[2]
7                 sorted_word_list_1 = word_frequencies_for_file(filename_1)
8                 sorted_word_list_2 = word_frequencies_for_file(filename_2)
9                 distance = vector_angle(sorted_word_list_1,sorted_word_list_2)
10                print ("The distance between the documents is: %0.6f (radians)"%distance)
11
```

As you can see in the main function if the user does not include additional arguments in the CLI when invoking the docdist program it will invoke warning and stop the program altogether. In order to solve this problem I need to make the run will include the t1.verne.txt and t2.bobsey.txt in the initial parameters. Why we use t1.verne.txt and t2.bobsey.txt? Well because both files are the smallest of all documents in the project. This is merely a test run and debug run to see how the code works. Thus, choosing the smallest file as example for all docdist program (docdist1 to docdist6) will result comparable and comprehensive results.

In order to make the VSCode built in debugger works I need to modify the main function a bit. The main idea here is to make the test documents files part of the arguments from the first time debug run is initiated. As we cannot use CLI to run the debug state of the program then I need to modify the main function as this is the function called first on run. Here is the modification of the code:

```
1         def main():
2     # import pdb; pdb.set_trace() # <--- this is for debug purpose only
3     if len(sys.argv) != 3:
4         print ("Usage: docdist4.py filename_1 filename_2")
5         #-- FOR DEBUG PURPOSES ONLY
6         filename_1 = 'E:\\python_me\\6-006_python\\lec02_code\\t1.verne.txt'
7         filename_2 = 'E:\\python_me\\6-006_python\\lec02_code\\t2.bobsey.txt'
```

```
        sorted_word_list_1 = word_frequencies_for_file(filename_1)
        sorted_word_list_2 = word_frequencies_for_file(filename_2)
        distance = vector_angle(sorted_word_list_1,sorted_word_list_2)
        print ("The distance between the documents is: %0.6f (radians)"%distance)
        # --- comment out this after finish debugging!
    else:
        filename_1 = sys.argv[1]
        filename_2 = sys.argv[2]
        sorted_word_list_1 = word_frequencies_for_file(filename_1)
        sorted_word_list_2 = word_frequencies_for_file(filename_2)
        distance = vector_angle(sorted_word_list_1,sorted_word_list_2)
        print ("The distance between the documents is: %0.6f (radians)"%distance)

```

NOTE: the file paths are in Windows format since the files are stored in the local Windows storage. I think it will be safer to use Windows path format.

1.1 UML Sequence Diagram

I need tool to help me understand how the program works. As takin notes are too random and the contents are easier to forget, I think I need better methods to make sense how the program works. The Sequence Diagram from UML sounds like a good tool for this job. Unlike static Class Diagram the Sequence Diagram will record the interaction between function(?) in the program.

1.2 Docdist1

In the docdist 1 there is a specification at this:

```
        #!/usr/bin/python
        # docdist1.py - initial version of document distance
        #
        # Original version by Ronald L. Rivest on February 14, 2007
        # Revision by Erik D. Demaine on September 12, 2011
        #
        # Usage:
        #    docdist1.py filename1 filename2
        #
        # This program computes the "distance" between two text files
        # as the angle between their word frequency vectors (in radians).
        #
        # For each input file, a word-frequency vector is computed as follows:
        #    (1) the specified file is read in
        #    (2) it is converted into a list of alphanumeric "words"
        #        Here a "word" is a sequence of consecutive alphanumeric
        #        characters.  Non-alphanumeric characters are treated as blanks.
        #        Case is not significant.
        #    (3) for each word, its frequency of occurrence is determined
        #    (4) the word/frequency lists are sorted into order alphabetically
        #
        # The "distance" between two vectors is the angle between them.
```

```
23          # If x = (x1, x2, ..., xn) is the first vector (xi = freq of word i)
24          # and y = (y1, y2, ..., yn) is the second vector,
25          # then the angle between them is defined as:
26          #    d(x,y) = arccos(inner_product(x,y) / (norm(x)*norm(y)))
27          # where:
28          #    inner_product(x,y) = x1*y1 + x2*y2 + ... xn*yn
29          #    norm(x) = sqrt(inner_product(x,x))
30
31
```

I must admit the specification is a bit long, but this is important since these specifications will be the basis to compare to other algorithms. The basic principle here is:

1. read all the lines in both documents that will return list of strings per line in document
2. from all of those list of strings they were split into words, hence return list of words.
3. then calculate the frequency of each word and put it into list in a list ie: [['0', 3],['an', 42]], means the string zero has 3 times occurance while string word an has 42 occurrance in one document.
4. then sort them alphabetically, NOT based on frequency value.
5. then begin finding vector distance using dot based inner products.

Basically, the final steps will be similar across the Docdist algorithms. However, minor changes might prove useful to increase the processing speed. For the docdist1.py the processing time is 3.625 with document distance = 0.582 radians.

1.3 Docdist2

Now in the Docdist2 there is only one small modification as stated in the top part of its specification:

```
1           #!/usr/bin/python
2           # docdist2.py - changed concatenate to extend in get_words_from_line_list
3
```

In the code at the **get_words_from_line_list** it is stated that using the List module extend function will make the process more efficient. The List.extend(List:seq) is a fucntion that will add element of another list into the List that call it. Other steps are basically the same as the Docdist1.py. However, just by modifying one line it has increase the computation speed. For docdist2.py the processing time is 3.188 seconds with document distance = 0.582 radians.

1.4 Docdist3

In docdist3 there will be more modification. Basically it started from the docdist2.py code but with one more modification. Here is as written in the docdist3.py specification:

```
1           #!/usr/bin/python
2           # docdist3.py - improved dot product to exploit sorted order and achieve
3           #               linear instead of quadratic time
4
```

Meaning most of the code in the diagram for docdist2.py still works in the docdist3.py with some adjustment in the inner_product function. In docdist2.py the inner_product function is defined as:

```
def inner_product(L1,L2):
"""
Inner product between two vectors, where vectors
are represented as lists of (word,freq) pairs.

Example: inner_product([["and",3],["of",2],["the",5]],
                       [["and",4],["in",1],["of",1],["this",2]]) = 14.0
"""
sum = 0.0
for word1, count1 in L1:
    for word2, count2 in L2:
        if word1 == word2:
            sum += count1 * count2
return sum

```

For the docdist3.py the inner_product function is defined as:

```
def inner_product(L1,L2):
"""
Inner product between two vectors, where vectors
are represented as alphabetically sorted (word,freq) pairs.

Example: inner_product([["and",3],["of",2],["the",5]],
                       [["and",4],["in",1],["of",1],["this",2]]) = 14.0
"""
sum = 0.0
i = 0
j = 0
while i<len(L1) and j<len(L2):
    # L1[i:] and L2[j:] yet to be processed
    if L1[i][0] == L2[j][0]:
        # both vectors have this word
        sum += L1[i][1] * L2[j][1]
        i += 1
        j += 1
    elif L1[i][0] < L2[j][0]:
        # word L1[i][0] is in L1 but not L2
        i += 1
    else:
        # word L2[j][0] is in L2 but not L1
        j += 1
return sum

```

As the L1 and L2 lists already sorted meaning the order of words in both lists already managed alphabetically. Thus if the same index is not the same then those the inner product of both words will be zero. The word if present in L2 but not present in L1 or vice versa will result zero inner

product, thus can be omitted in the sum. *See the Document Distance section to learn more!*.

Now I know why they must be sorted first. This make sense now since using the older inner_product from docdist2.py will cost quadratic time $\mathcal{O}(n)^2$. Meanwhile, the inner_product function in docdist3.py will cost linear time $\mathcal{O}(n)$. The processing time for docdist3.py is 2.031 seconds, much faster compared to docdist2.py. The difference between quadratic and linear is much significant compared to the difference between docdist1.py to docdist2.py. This is because the docdist2.py still has the same running time or at least similar.

1.5 Docdist4

Now this is the first bug arise. This is because the docdist4.py uses dictionary to store the frequency data of each word. However, the latest Python 3.x will not treat the dictionary items object as indexable. This will make sorting the data impossible.

```
1        def count_frequency(word_list):
2        """
3        Return a list giving pairs of form: (word,frequency)
4        """
5        D = {}
6        for new_word in word_list:
7            if new_word in D:
8                D[new_word] = D[new_word]+1
9            else:
10               D[new_word] = 1
11       return list(D.items())
12
```

This is the main difference and source of the problem. As mentioned in the docdist4.py specification:

```
1        #!/usr/bin/python
2        # docdist4.py - changed count_frequency to use dictionaries instead of lists
3
```

the result of the count frequency will be contained in a dictionary rather than list like in previous docdist files. Let's learn about dictionary items object first. Here is one example on that:

```
1        car = {
2        "brand": "Ford",
3        "model": "Mustang",
4        "year": 1964
5        }
6
7        x = car.items()
8
9        print(x)
10
```

The code above will result an object described as:

```
1        dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
2
```

Although it seems like a list consist of tuples which contains key value combination of the whole dictionary, it is not a list. List will be able to be indexed, but this object cannot. In order to make it indexable I need to modify it back to list:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = list(car.items())

print(x)

```

The code will returns:

```
[('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)]

```

This is a list just like before. However, it is different since inside this list is tuples. In previous docdist files it is list inside a list. Moreover, it is supposed to be faster to access dictionary compared to list. *This is need to be verified further!*

The main question here is the docdist4.py still uses docdist3.py inner product formulation. Since the words for both documents are already put into dictionary it is will be faster just to check if certain word is available in one or another dictionary. However, I need to run debug to find out what happen during the count_frequency algorithm.

In the debug run I can see that the list is now list of tuples of (word, frequency) pair sorted alphabetically. This is the only difference between the previous lists from previous docdist. The inner product also use the same algorithm as the docdist3.py thus will have linear time complexity $\mathcal{O}(n)$ as before. Therefore, as the docdist4.py performance is better compared to the docdist3.py is more because of the dictionary logging compared to the list used in previous algorithm. The docdist4.py took 1.547 seconds to finish the processing document distance. This is significantly faster compared to docdist3.py which requre 2.013 seconds to finish the process.

Pause for a minute

The debug on the docdist4.py suppose to decide on how to solve the bug in docdist5.py and docdist6.py. Both algorithms uses the same dictionary principle thus resulting similar bug as the docdist4.py. So far by converting the dictionary.items object into list of tuples of dictionary items solve the bug. However, does the result validate the solution?

Is it suppose to be faster using dictionary to log the information compared to append it in a list? Well it supposedly so. For once the indexing in dictionary is not as strict as in a list nor tuple. Dictionary have keys as pointers to certain values. Thus it is not depending on the index position to address certain value.

I say it is a good chance the solution is having merit. Changing it to the list form only supplied the items the indexable feature. It is used in the sort process. However, the latest solution still using the same inner product as previous docdist3 algorithm. Therefore it is by merit that appending to dictionary should be faster compared to list.

1.6 Docdist5

Here is the first file I need to really debug. Now let's begin with its specification first:

```
#!/usr/bin/python
# docdist5.py - change get_words_from_string to use string translate and split
#

```

This means I need to compare the previous get_words_form_string to the one in the docdist5.py. This is the previous version:

```
def get_words_from_string(line):
"""
Return a list of the words in the given input string,
converting each word to lower-case.

Input:  line (a string)
Output: a list of strings
        (each string is a sequence of alphanumeric characters)
"""
word_list = []          # accumulates words in line
character_list = []     # accumulates characters in word
for c in line:
    if c.isalnum():
        character_list.append(c)
    elif len(character_list)>0:
        word = "".join(character_list)
        word = word.lower()
        word_list.append(word)
        character_list = []
if len(character_list)>0:
    word = "".join(character_list)
    word = word.lower()
    word_list.append(word)
return word_list

```

Now we compare with the one in the docdist5.py, here:

```
def get\_words\_from_string(line):
"""
Return a list of the words in the given input string,
converting each word to lower-case.

Input:  line (a string)
Output: a list of strings
        (each string is a sequence of alphanumeric characters)
"""
line = line.translate(translation_table)
word_list = line.split()
return word_list
```

```

```

Basically, the docdist5.py uses the Python's built in function to split the senteces into list of words. As for the string translate function this is new for me. To understand this I need to check to the translation_table variable.

```
        # global variables needed for fast parsing
        # translation table maps upper case to lower case and punctuation to spaces
        # for Python 3.x the string module does not have maketrans method anymore as it is being deprecated
        # it is being substituted with the str (built in text sequence type).=> str. maketrans is static thus we can use it right away
        # also the string.uppercse is being deprecated to string.ascii_uppercase, same as string.lowercase to ascii_lowercase
        translation_table = str.maketrans(string.punctuation+string.ascii_uppercase," "*len (string.punctuation)+string.ascii_lowercase)

```

Basically the tranlation_table convert all punctuations (white spaces, commas, dots, and alike) to just white spaces (" ") times the length of the punctuations list. Also it converts uppercase into lowercase. Note that the translation_table uses ascii module since it exchange one character to another based on its ASCII code.

Now on the comment to the snippet also mentions bugs. The Python 3.x does not support the direct maketrans method. Now the Python will use the str built in library and then use the maketrans method.

Since docdist5.py also uses dictionary to store the frequency then it has the same bug as the previous docdist4.py. The solution is also the same by adding list(D.items()) code into the count_frequency function. As this also use the same inner product calculation as the sorted thus the time complexity at that function will be $\mathcal{O}(n)$ or linear. Thus the difference will only be from the get_words_from_string function and the use of built in method translate and split. Apparently it is a significant improvement once again wiht docdist5.py complete the process in just 0.625 seconds.

1.7 Docdist6

Here is the specification of the docdist6.py:

```
        #!/usr/bin/python
        # docdist6.py - changed sorting from insertion sort to merge sort
        #

```

This means the main different in the docdist6.py compared to the previous docdist is in the operation 4 section: sorting alphabetically. The old insertion_sort function is no longer in use, although in the code this function is still present. For comparison sake perhaps? Well here is the function that taking its place:

```
        def merge_sort(A):
        """
        Sort list A into order, and return result.
        """
```

```
        n = len(A)
        if n==1:
            return A
        mid = n//2     # floor division
        L = merge_sort(A[:mid])
        R = merge_sort(A[mid:])
        return merge(L,R)

    def merge(L,R):
        """
        Given two sorted sequences L and R, return their merge.
        """
        i = 0
        j = 0
        answer = []
        while i<len(L) and j<len(R):
            if L[i]<R[j]:
                answer.append(L[i])
                i += 1
            else:
                answer.append(R[j])
                j += 1
        if i<len(L):
            answer.extend(L[i:])
        if j<len(R):
            answer.extend(R[j:])
        return answer

```

As you can see it takes two functions to handle the merge sort. One is the merge_sort(A) function which uses recursion on both direction. After all the part is divided into their smallest unit the function will call the merge(L,R) to merge it in sorted order, in this case alphabetically. Same as docdist5.py, the docdist6.py also have the same problem in the dictionary items object. I need to convert it into list first.

```
        def count_frequency(word_list):
            """
            Return a list giving pairs of form: (word,frequency)
            """
            D = {}
            for new_word in word_list:
                if new_word in D:
                    D[new_word] = D[new_word]+1
                else:
                    D[new_word] = 1
            return D.items()

```

After fixing the problem, the run of the docdist6.py is faster with needed time to complete 0.281 seconds. By using merge sort it basically reduce the time complexity of the sorting process from $\mathcal{O}(n)$ or linear to $\mathcal{O}(log_2 n)$ or logaritmic.