# Filip Ekberg's *Blog (/)*

***Author. Blogger. Speaker. C# MVP. Software Engineer. Geek.***

# Calculating Document Distance (/2014/02/17/calculating-document-distance/)

***Posted by Filip Ekberg (/about-filip/) on 17 Feb 2014***

Previously we looked at the first part in my Back to Basics series where we understood and implemented Peak-Finding. This time we are going to talk about something slightly different; Calculating Document Distance. I really recommend you to take a look at the MIT course on Introduction to Algorithms (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-2-models-of-computation-document-distance/), for this post I really recommend watching the part about document distance.

Practicing algorithms is both fun and educating, even if you are 20 years into your career you will most certainly always learn something new when analyzing algorithms. There's no greater feeling than when you've tackled a problem for a long time and suddenly you understand it deeply enough to optimize it and play with the edge cases.

## What is Document Distance?

Consider that you have two documents containing a huge amount of text in them, be it essays or websites. Now you want to know how similar these documents are, in the sense of: how many words overlap in these documents. Conceptual the algorithm is really simple there's just a few steps that you'll have to go through:

1. Open and read both documents that you are going to compare. Only read words and numbers, skip special characters (spaces, dots, etc..) and convert the words to lower case
2. Calculate the word frequency in both collections of words, this means how many times each word occur in each document
3. Compare the frequencies from both computations and calculate the distance

The distance itself is calculated using a predefined formula that you don't really have to pay too much attention too at this moment, unless you really fancy computations on

Both the first two steps are pretty trivial, we'll make use of some built in data structures that are fast for random access and for inserts and try to not make it all too complex. The second step is the one with the fancy math in it. We're going to find something called inner product (http://mathworld.wolfram.com/InnerProduct.html), which basically takes the frequencies of the words that occur in both documents, multiplies those and adds them up.

So let's say that our frequencies are represented using a dictionary, that way we have O(1) access time and O(1) inserts. To compute the product we then produce a method with the following signature: `int ComputeInnerProduct(Dictionary<string, int> first, Dictionary<string, int> second)`. This means that we have the frequencies for both the first and the second document and now we can add these together where the words intersect.

The implementation of this is quiet simple when you know what the formula expects (we'll get to that in a moment), here's the entire method that finds the inner product:

```
public int ComputeInnerProduct(Dictionary<string, int> first,
Dictionary<string, int> second)
{
    var sum = 0;
    foreach (var key in first.Keys)
    {
        if (second.ContainsKey(key)) sum += first[key]*second[key];
    }

    return sum;
}
```

As we only care about intersection, we just have to go over one of the lists, we don't care if the other one is longer or not. For each element we have in our dictionary, we simply check if the other document have the same word and then perform the multiplication which we add into our sum.

Ready to take a look at the formula? Well, ready or not here it comes!

$$\arccos\left(\frac{L1 \cdot L2}{\sqrt{(L1 \cdot L1)(L2 \cdot L2)}}\right)$$

This equation is taken from the MIT course material from the open courseware course I linked above, think of L1 as the first document and L2 as the second documents

frequencies that intersect. Basically where it says `L1 * L2` it refers to the inner product of these two. The equation finds an angle in a vector based on our inputs. So you might have figured out that `L1 * L1` means the inner product of these two and the same goes for `L2 * L2`. We basically just have to call **`ComputeInnerProduct`** three times, with slightly different input.

Let's just take a look at what the implementation of this method does, the method that computes the distance.

```
public double ComputeDistance(Dictionary<string, int> first,
Dictionary<string, int> second)
{
    var numerator = ComputeInnerProduct(first, second);

    var denominator = Math.Sqrt(ComputeInnerProduct(first, first) *
ComputeInnerProduct(second, second));

    return Math.Acos(numerator / denominator);
}
```

As you can see in the code sample we are doing exactly what the formula expresses, at least as long as we trust our **`int ComputeInnerProduct`** implementation.

We've got the two hardest parts done that has the most of math inside it, the rest is just the trivial methods that loads the files and processes the text. Let's take a look at how we calculate the frequency of the words in our document. I want this operation to be fast as well, so again I will make use of the dictionary to get O(1) (constant) lookup and insert.

```
public Dictionary<string, int> ComputeFrequency(string[] input)
{
    var result = new Dictionary<string, int>();

    for (var i = 0; i < input.Length; i++)
    {
        if (result.ContainsKey(input[i]))
        {
            result[input[i]]++;
        }
        else
        {
            result.Add(input[i], 1);
        }
    }

    return result;
}
```

Let's take an example input and see what happens, consider that we have the following text:

> Algorithms are fun and educating! Solving the algorithms are as fun as writing about them.

Breaking this up into a frequency table will give us something like this:

| | |
|---|---|
| algorithms | 2 |
| are | 2 |
| fun | 2 |
| and | 1 |
| educating | 1 |
| solving | 1 |
| the | 1 |
| as | 1 |
| writing | 1 |
| about | 1 |
| them | 1 |

This is just one of the documents that we processed, we still would need to do the same for a second one. You can see from the above how the frequencies are calculated. When having the frequencies from two different documents, this is what we pass to our distance computation. We've still got one final step before adding them all together and executing them: we need to load the file and process the text. Notice that everything from above was lower case and that there were no spaces, line-breaks, commas or punctuation? That is because we strip the text from all that.

Honestly the only method so far that I feel is a bit messy is the one that reads the text and processes it. I'd be happy to hear how you'd clean it up, doing it in Python is so much nicer as you get more help from the framework doing the boilerplate stuff. Remember, if you think about using regex that can be a slower operation and you need to keep that in mind when designing your algorithms.

I've created a method that retrieves all the words for a single file and it looks like this:

```
public string[] GetWords(string filename)
{
  var words = new List<string>();
  var characters = new List<char>();

  var input = new StreamReader(filename).ReadToEnd();
  var seperators = new List<char> { ' ' };
  seperators.AddRange(Environment.NewLine);
  foreach (var word in input.Split(seperators.ToArray()))
  {
      foreach (var character in word.ToCharArray())
      {
          if (char.IsLetterOrDigit(character)) characters.Add(character);
      }

      if (characters.Count > 0)
      {
          words.Add(string.Join("", characters).ToLowerInvariant());
          characters.Clear();
      }
  }

  return words.ToArray();
}
```

Consider the word "don't" when processing this you'll get it as "dont", which is exactly what we want as we want to strip it from everything that is not an alphanumeric. The method is quiet trivial, it splits the text that it loaded by lines and spaces, then it adds each character in the current word that is an alphanumeric to our character collection, when it has processed all the characters, it adds the string representation as a lower case invariant to the collection of words. When all words are processed we are ready for the frequency calculation.

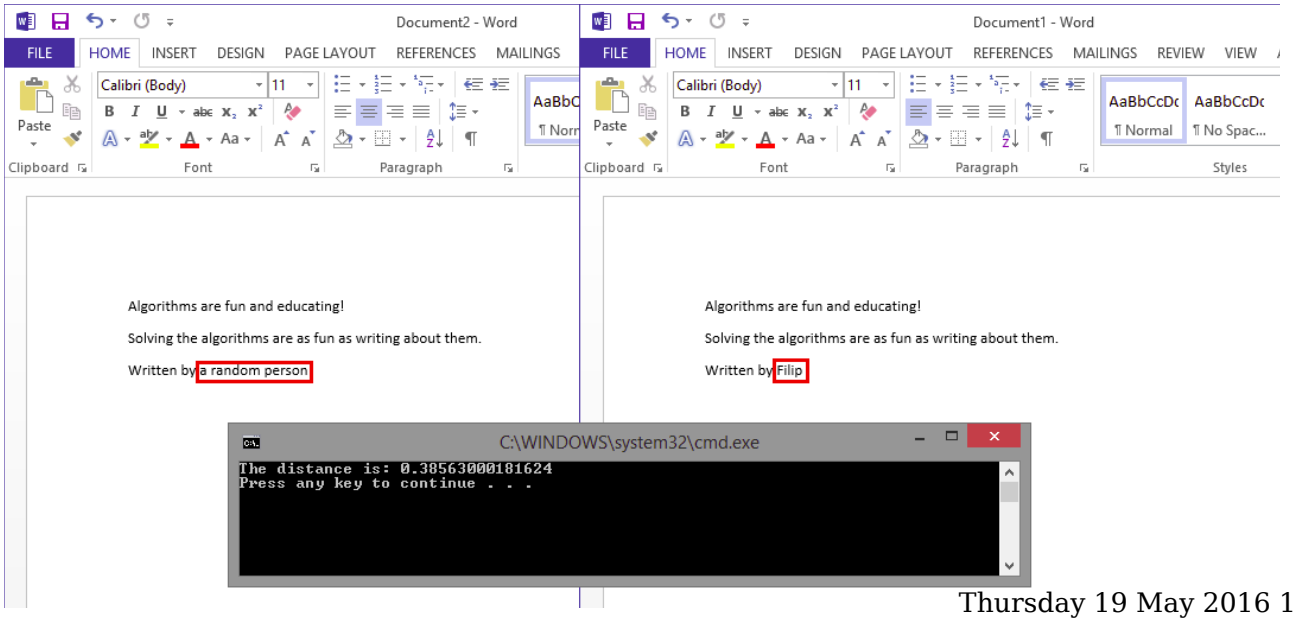
To tie it all together we can run this from the programs main method in this case and the sequence of invocations will look like this:

```
static void Main(string[] args)
{
    var program = new Program();
    var first = program.GetWords("file1.txt");
    var second = program.GetWords("file2.txt");

    var firstFrequencies = program.ComputeFrequency(first);
    var secondFrequencies = program.ComputeFrequency(second);

    var distance = program.ComputeDistance(firstFrequencies,
secondFrequencies);

    Console.WriteLine("The distance is: {0}", distance);

}
```

So looking at the conceptual definition of the algorithm you'll see that we are doing just that.

1. Open and read both documents that you are going to compare. Only read words and numbers, skip special characters (spaces, dots, etc..) and convert the words to lower case
2. Calculate the word frequency in both collections of words, this means how many times each word occur in each document
3. Compare the frequencies from both computations and calculate the distance

I wrote two text files that contains almost the exact same data, except the author name in the end of the text is different. I opened it up in word and highlighted the difference then I ran the document distance algorithm on it and it shows us that the distance is not far at all, hence this is almost an exact copy.



You will notice that as the documents approach similarity the distance decreases and when the are identical it will be a 0 distance. If the are completely different the distance will be the maximum distance which is 1.5707963267949. This method of finding distances between documents can be used to find cheaters on essays, help with searching through documents and sort by relevance and much more. It's a really interesting algorithm that lets you think a bit about what is going on.

The complete code is available on GitHub in my Algorithms repository. (https://github.com/fekberg/Algorithms/blob/master/Document%20Distance/Document%20Distance/Program.cs)

It's fun and educating to try and solve it on paper first by both sketching and coding on paper, you'll find lots of interesting bugs and edge cases that you didn't think about when the compiler isn't there to help you out and it will greatly improve your analysis skills.



| Algorithms & Data structures (/category/algorithms-and-data-structures) | C# (/category/c) | Programming (/category/programming)

3 Comments Filip Ekberg's blog 1 Login

Recommend Share Sort by Best

Join the discussion…

**Daouda Traoré** • a year ago

Hey Filip,

Here is a piece of code that i took from MIT Open Course Ware, from the recitation 2 class in algorithm.
please hlp me to understand what happen exactly from line 12 to 16.
They said that it's to not miss up the word at the end of the line, but i don't know how that could happen when i look at the code!!!!
Let's go with the example they gave in the recitation:

The fax is outside.

Let's pretend that we reach the last word. At line 5 it will check and start getting the characters for OUTSIDE till reach the DOT at he end of the line. The not being alnum, then it will execute line 7 to 11. So for me the algorithm won't miss up the last word. But they said it will, then it should be a point that i am not understanding. Is there different interpretation of "space " and other NONE alnum characters ?
At line 5 the algorithm check for "ALNUM", it's not the case for the (DOT), so why the program should act differently? What happen exaclty from the last word to the end of the line???

see more

• Reply • Share ›

**Filip Ekberg** Mod ➚ Daouda Traoré • a year ago

Hey Daouda,

Not sure I exactly understand what you are asking, could you tell me what you've tried and what's not working properly?

Cheers

• Reply • Share ›

**Daouda Traoré** ➚ Filip Ekberg • a year ago

Hey Filip,

I eventually understood what was the point. If you look at the code i send in my first post, you'll notice that they repeated some line of code at the end which was in the for loop, and they said it was too care about the lasst word. but as i told you got it now. Something i don't understand yet, is how the code lead with word composed of non alphanumeric characters. Look at the script below from 5 to 11, and try to figure out how its behavior in the case you have a word like "DIDN'T". At 5 the