longest_common_subsequence

May 5, 2020

1 Longest Common Subsequence

In text analysis, it is often useful to compare the similarity of two texts (imagine if you were trying to determine plagiarism between a source and answer text). In this notebook, we'll explore one measure of text similarity, the **Longest Common Subsequence** or LCS.

> The Longest Common Subsequence is the longest string of letters that are *the same* between to strings.

A short example: * For two input strings, A and B * A = 'ABCD' * B = 'BD' * The LCS is 'BD', which has a length of **2** characters ---

1.1 Storing pre-computed values

The LCS algorithm depends on looking at two strings and comparing them letter by letter. You can solve this problem in multiple ways. You can iterate through each letter in the strings and compare them, adding to your value for LCS as you go.

The method I recommend for implementing an efficient LCS algorithm is: using a matrix and dynamic programming. Recall that **dynamic programming** is all about breaking a larger problem into a smaller set of subproblems, and building up a complete result without having to repeat any subproblems.

This approach assumes that you can split up a large LCS task into a combination of smaller LCS tasks. Let's look at the short example in more detail:

- A = 'ABCD'
- B = 'BD'

We can see right away that the longest subsequence of *letters* here is 2 (B and D are in sequence in both strings). And we can calculate this by looking at relationships between each letter in the two strings, A and B.

Here, I have a matrix with the letters of A on top and the letters of B on the left side:

This starts out as a matrix that has as many columns and rows as letters in the strings S and O **+1** additional row and column, filled with zeros on the top and left sides. So, in this case, instead of a 2x4 matrix it is a 3x5.

Now, we can fill this matrix up by breaking it into smaller LCS problems. For example, let's first look at the shortest substrings: the starting letter of A and B. We'll first ask, what is the Longest Common Subsequence between these two letters "A" and "B"?

Here, the answer is zero and we fill in the corresponding grid cell with that value.

Then, we ask the next question, what is the LCS between "AB" and "B"?

Here, we have a match, and can fill in the appropriate value 1.

If we continue along this row, we can actually see that B only matches this one time, and any further questions, such as — What is the LCS between "ABCD" and "B"? — will have that same value, 1, due to the initial B-B match.

Then, we move on to the second row. "A" and "BD" have 0 matches.

But "AB" and "BD" have a B-B match, which we've already noted in the cell above. Finally, we have a match at the end D-D. Where we can add one to our current highest match (1) to get a final LCS of 2.

The final LCS will be that value **2**.

1.1.1 The matrix rules

One thing to notice here is that, you can efficiently fill up this matrix one cell at a time. Each grid cell only depends on the values in the grid cells that are directly on top and to the left of it, or on the diagonal/top-left. The rules are as follows: * Start with a matrix that has one extra row and column of zeros. * As you traverse your string: * If there is a match, fill that grid cell with the value to the top-left of that cell *plus* one. So, in our case, when we found a matching B-B, we added +1 to the value in the top-left of the matching cell, 0. * If there is not a match, take the *maximum* value from either directly to the left or the top cell, and carry that value over to the non-match cell.

- After completely filling the matrix, **the bottom-right cell will hold the non-normalized LCS value**.

1.2 Calculate the longest common subsequence

Implement the function `lcs`; this should calculate the *longest common subsequence* of characters between two strings.

```
In [ ]: def lcs(string_a, string_b):
            pass
```

Hide Solution

```
In [ ]: def lcs(string_a, string_b):
            lookup_table = [[0 for x in range(len(string_b) + 1)] for x in range(len(string_a) +

            for char_a_i, char_a in enumerate(string_a):
                for char_b_i, char_b in enumerate(string_b):
                    if char_a == char_b:
                        lookup_table[char_a_i + 1][char_b_i + 1] = lookup_table[char_a_i][char_b
                    else:
                        lookup_table[char_a_i + 1][char_b_i + 1] = max(
                            lookup_table[char_a_i][char_b_i + 1],
                            lookup_table[char_a_i + 1][char_b_i])

            return lookup_table[-1][-1]
```

Test your function on a few test strings by running the cell, below.

```
In [ ]: ## Test cell

        # Run this cell to see how your function is working
        test_A1 = "WHOWEEKLY"
        test_B1 = "HOWONLY"

        lcs_val1 = lcs(test_A1, test_B1)

        test_A2 = "CATSINSPACETWO"
        test_B2 = "DOGSPACEWHO"

        lcs_val2 = lcs(test_A2, test_B2)

        print('LCS val 1 = ', lcs_val1)
        assert lcs_val1==5, "Incorrect LCS value."
        print('LCS val 2 = ', lcs_val2)
        assert lcs_val2==7, "Incorrect LCS value."
        print('Tests passed!')
```

1.3 Complexity

What is the complexity?

```
In [ ]: # The time complexity of the above implementation
        # is dominated by the two nested loops,
        # which give us an O(N^2) time complexity.
```

Show Solution