SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 007
Accredited by NAAC-UGC with 'A' Grade
Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

Department of Artificial Intelligence and Data Science

Course Name – 16AD601 – Natural Language Processing

III Year / VI Semester

Unit 1 – Introduction

Topic 8- Dynamic Programming Edit Distance

Dynamic Programming Edit Distance

Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to subproblems. Some of the most commonly used algorithms in natural language processing make use of dynamic programming.

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems.

The minimum edit distance algorithm was named by Wagner and Fischer but independently discovered by many people.

Dynamic Programming Edit Distance

Let’s first define the minimum edit distance between two strings. Given two strings, the source string X of length n, and target string Y of length m, we’ll define D[i; j] as the edit distance between X[1::i] and Y[1:: j], i.e., the first i characters of X and the first j characters of Y. The edit distance between X and Y is thus D[n;m].

We’ll use dynamic programming to compute D[n;m] bottom up, combining solutions to subproblems.

In the base case, with a source substring of length i but an empty target string, going from i characters to 0 requires i deletes.

With a target substring of length j but an empty source going from 0 characters to j characters requires j inserts. Having computed D[i; j] for small i; j we then compute larger

Dynamic Programming Edit Distance

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

If we assume the version of Levenshtein distance in which the insertions and deletions each have a cost of 1 (ins-cost(.) = del-cost(.) = 1), and substitutions have a cost of 2 (except substitution of identical letters have zero cost), the computation for D[i; j] becomes:

Dynamic Programming Edit Distance

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

   n ← LENGTH(source)
   m ← LENGTH(target)
   Create a distance matrix D[n+1,m+1]

   # Initialization: the zeroth row and column is the distance from the empty string
   D[0,0] = 0
   for each row i from 1 to n do
       D[i,0] ← D[i-1,0] + del-cost(source[i])
   for each column j from 1 to m do
       D[0,j] ← D[0, j-1] + ins-cost(target[j])

   # Recurrence relation:
   for each row i from 1 to n do
         for each column j from 1 to m do
            D[i, j] ← MIN( D[i−1,j] + del-cost(source[i]),
                           D[i−1, j−1] + sub-cost(source[i], target[j]),
                           D[i, j−1] + ins-cost(target[j]))
   # Termination
   return D[n,m]
```

Figure 2.17 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x$, ins-cost$(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., sub-cost$(x,x) = 0$).

Dynamic Programming Edit Distance

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Figure 2.18 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.17, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions.

THANK YOU