Levenshtein distance

From Wikipedia, the free encyclopedia

In information theory and computer science, the **Levenshtein distance** is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. It is named after Vladimir Levenshtein, who considered this distance in 1965.[1]

Levenshtein distance may also be referred to as **edit distance**, although that may also denote a larger family of distance metrics.[2]:32 It is closely related to pairwise string alignments.

Contents

Definition

Mathematically, the Levenshtein distance between two strings a, b (of length $|a|$ and $|b|$ respectively) is given by $\operatorname{lev}_{a,b}(|a|, |b|)$ where

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $\operatorname{lev}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b.

Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

Example

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. **k**itten → **s**itten (substitution of "s" for "k")
2. sitt**e**n → sitt**i**n (substitution of "i" for "e")
3. sittin → sittin**g** (insertion of "g" at the end).

Upper and lower bounds

The Levenshtein distance has several simple upper and lower bounds. These include:

- It is always at least the difference of the sizes of the two strings.
- It is at most the length of the longer string.
- It is zero if and only if the strings are equal.
- If the strings are the same size, the Hamming distance is an upper bound on the Levenshtein distance.
- The Levenshtein distance between two strings is no greater than the sum of their Levenshtein distances from a third string (triangle inequality).

An example where the Levenshtein distance between two strings of the same length is strictly less than the Hamming distance is given by the pair "flaw" and "lawn". Here the Levenshtein distance equals 2 (delete "f" from the front; insert "n" at the end). The Hamming distance is 4.

Applications

In approximate string matching, the objective is to find matches for short strings in many longer texts, in situations where a small number of differences is to be expected. The short strings could come from a dictionary, for instance. Here, one of the strings is typically short, while the other is arbitrarily long. This has a wide range of applications, for instance, spell checkers, correction systems for optical character recognition, and software to assist natural language translation based on translation memory.

The Levenshtein distance can also be computed between two longer strings, but the cost to compute it, which is roughly proportional to the product of the two string lengths, makes this impractical. Thus, when used to aid in fuzzy string searching in applications such as record linkage, the compared strings are usually short to help improve speed of comparisons.

Relationship with other edit distance metrics

There are other popular measures of edit distance, which are calculated using a different set of allowable edit operations. For instance,

- the Damerau–Levenshtein distance allows insertion, deletion, substitution, and the transposition of two adjacent characters;
- the longest common subsequence metric allows only insertion and deletion, not substitution;
- the Hamming distance allows only substitution, hence, it only applies to strings of the same length.

Edit distance is usually defined as a parameterizable metric calculated with a specific set of allowed edit operations, and each operation is assigned a cost (possibly infinite). This is further generalized by DNA sequence alignment algorithms such as the Smith–Waterman algorithm, which make an operation's cost depend on where it is applied.

Computing Levenshtein distance

Recursive

This is a straightforward, but inefficient, recursive C implementation of a `LevenshteinDistance` function that takes two strings, *s* and *t*, together with their lengths, and returns the Levenshtein distance between them:

```
// len_s and len_t are the number of characters in string s and t respectively
int LevenshteinDistance(char *s, int len_s, char *t, int len_t)
{
  int cost;
  /* base case: empty strings */
  if (len_s == 0) return len_t;
  if (len_t == 0) return len_s;

  /* test if last characters of the strings match */
  if (s[len_s-1] == t[len_t-1])
      cost = 0;
  else
      cost = 1;

  /* return minimum of delete char from s, delete char from t, and delete char from both */
  return minimum(LevenshteinDistance(s, len_s - 1, t, len_t    ) + 1,
                 LevenshteinDistance(s, len_s    , t, len_t - 1) + 1,
                 LevenshteinDistance(s, len_s - 1, t, len_t - 1) + cost);
}
```

This implementation is very inefficient because it recomputes the Levenshtein distance of the same substrings many times.

A more efficient method would never repeat the same distance calculation. For example, the Levenshtein distance of all possible prefixes might be stored in an array `d[][]` where `d[i][j]` is the distance between the first `i` characters of string `s` and the first `j` characters of string `t`. The table is easy to construct one row at a time starting with row 0. When the entire table has been built, the desired distance is `d[len_s][len_t]`.

Iterative with full matrix

Note: This section uses 1-based strings instead of 0-based strings

Computing the Levenshtein distance is based on the observation that if we reserve a matrix to hold the Levenshtein distances between all prefixes of the first string and all prefixes of the second, then we can compute the values in the matrix in a dynamic programming fashion, and thus find the distance between the two full strings as the last value computed.

This algorithm, an example of bottom-up dynamic programming, is discussed, with variants, in the 1974 article *The String-to-string correction problem* by Robert A. Wagner and Michael J. Fischer.[3]

This is a straightforward pseudocode implementation for a function *LevenshteinDistance* that takes two strings, *s* of length *m*, and *t* of length *n*, and returns the Levenshtein distance between them:

```
function LevenshteinDistance(char s[1..m], char t[1..n]):
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t
  // note that d has (m+1)*(n+1) values
  declare int d[0..m, 0..n]

  set each element in d to zero

```

```
// source prefixes can be transformed into empty string by
// dropping all characters
for i from 1 to m:
    d[i, 0] := i

// target prefixes can be reached from empty source prefix
// by inserting every character
for j from 1 to n:
    d[0, j] := j

for j from 1 to n:
    for i from 1 to m:
        if s[i] = t[j]:
          substitutionCost := 0
        else:
          substitutionCost := 1
        d[i, j] := minimum(d[i-1, j] + 1,                   // deletion
                           d[i, j-1] + 1,                   // insertion
                           d[i-1, j-1] + substitutionCost)  // substitution

return d[m, n]
```

Two examples of the resulting matrix (hovering over a number reveals the operation performed to get that number):

		k	**i**	**t**	**t**	**e**	**n**
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

The invariant maintained throughout the algorithm is that we can transform the initial segment `s[1..i]` into `t[1..j]` using a minimum of `d[i,j]` operations. At the end, the bottom-right element of the array contains the answer.

Iterative with two matrix rows

It turns out that only two rows of the table are needed for the construction if one does not want to reconstruct the edited input strings (the previous row and the current row being calculated).

The Levenshtein distance may be calculated iteratively using the following algorithm:[4]

```
int LevenshteinDistance(string s, string t)
{
    // degenerate cases
    if (s == t) return 0;
    if (s.Length == 0) return t.Length;
    if (t.Length == 0) return s.Length;

    // create two work vectors of integer distances
    int[] v0 = new int[t.Length + 1];
    int[] v1 = new int[t.Length + 1];

    // initialize v0 (the previous row of distances)
    // this row is A[0][i]: edit distance for an empty s
    // the distance is just the number of characters to delete from t
    for (int i = 0; i < v0.Length; i++)
        v0[i] = i;

    for (int i = 0; i < s.Length; i++)
    {
        // calculate v1 (current row distances) from the previous row v0

        // first element of v1 is A[i+1][0]
        //   edit distance is delete (i+1) chars from s to match empty t
        v1[0] = i + 1;

        // use formula to fill in the rest of the row
        for (int j = 0; j < t.Length; j++)
        {
            var cost = (s[i] == t[j]) ? 0 : 1;
            v1[j + 1] = Minimum(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost);
        }

        // copy v1 (current row) to v0 (previous row) for next iteration
        for (int j = 0; j < v0.Length; j++)
            v0[j] = v1[j];
    }

    return v1[t.Length];
}
```

Hirschberg's algorithm combines this method with divide and conquer. It can compute the optimal edit sequence, and not just the edit distance, in the same asymptotic time and space bounds.[5]

Approximation

The Levenshtein distance between two strings of length n can be approximated to within a factor

$$(\log n)^{O(1/\varepsilon)}$$

where $\varepsilon > 0$ is a free parameter to be tuned, in time $O(\mathrm{n}^{1+\varepsilon})$.[6]

Computational hardness

It has been shown that the Levenshtein distance of two strings of length n cannot be computed in time $O(n^{2 - \varepsilon})$ unless the strong exponential time hypothesis is false.[7]

See also

- agrep
- diff
- MinHash
- Dynamic time warping
- Euclidean distance
- Homology of sequences in genetics
- Hunt–McIlroy algorithm
- Jaccard index
- Locality-sensitive hashing
- Longest common subsequence problem
- Lucene (an open source search engine that implements edit distance)
- Manhattan distance
- Metric space
- Most frequent k characters
- Optimal matching algorithm
- Similarity space on Numerical taxonomy
- Sørensen similarity index

References

1. Влади́мир И. Левенштейн (1965). Двоичные коды с исправлением выпадений, вставок и замещений символов [Binary codes capable of correcting deletions, insertions, and reversals]. *Доклады Академий Наук СССР* (in Russian). **163** (4): 845–8. Appeared in English as: Levenshtein, Vladimir I. (February 1966). "Binary codes capable of correcting deletions, insertions, and reversals". *Soviet Physics Doklady*. **10** (8): 707–710.
2. Navarro, Gonzalo (2001). "A guided tour to approximate string matching" (PDF). *ACM Computing Surveys*. **33** (1): 31–88. doi:10.1145/375360.375365.
3. Wagner, Robert A.; Fischer, Michael J. (1974), "The String-to-String Correction Problem", *Journal of the ACM*, **21** (1): 168–173, doi:10.1145/321796.321811
4. Hjelmqvist, Sten (26 Mar 2012), *Fast, memory efficient Levenshtein algorithm*
5. Hirschberg, D. S. (1975). "A linear space algorithm for computing maximal common subsequences". *Communications of the ACM*. **18** (6): 341–343. doi:10.1145/360825.360861. MR 0375829.

6. Andoni, Alexandr; Krauthgamer, Robert; Onak, Krzysztof (2010). *Polylogarithmic approximation for edit distance and the asymmetric query complexity*. IEEE Symp. Foundations of Computer Science (FOCS). arXiv:1005.4033. CiteSeerX: 10.1.1.208.2079.
7. Backurs, Arturs; Indyk, Piotr (2015). *Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false)*. Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOCS). arXiv:1412.0348.

External links

- Black, Paul E., ed. (14 August 2008), "Levenshtein distance", *Dictionary of Algorithms and Data Structures [online]*, U.S. National Institute of Standards and Technology, retrieved 3 April 2013

The Wikibook *Algorithm implementation* has a page on the topic of: ***Levenshtein distance***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=735379831"

Categories: String similarity measures | Dynamic programming | Quantitative linguistics