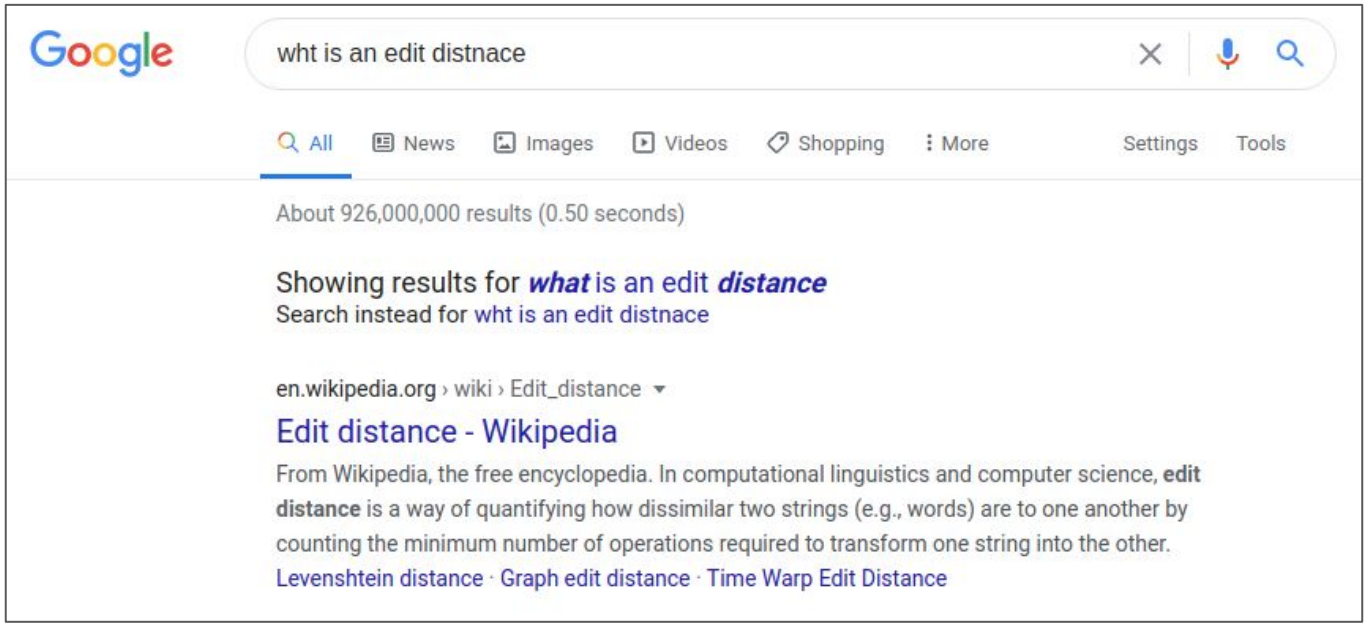
Neighbor

Edit Distances

Neighbor Mind Garden

Clark Brown

What is an Edit Distance?



How does Google know what we mean even though we don't know how to spell because autocorrect is ubiquitous or we type too fast because we are always in a hurry to get answers?

Edit distances are metrics quantify the difference between strings.

The distance is directly related to the minimum number of edit operations that would transform one string into the other.

Examples:

Cat and Cart - Add one letter to “cat” to get “cart”

Form and From - Switch two adjacent letters around in “form” to get “from”

Neighbor and Neiybor - Remove two letters and add one to “neighbor” to get “neiybor”

Common Operations:

Insertion of a single symbol

- If $a = uv$, then inserting the symbol x produces uxv
- This can be denoted $\varepsilon \to x$, using ε to denote the empty string

Deletion of a single symbol

- Changes uxv to uv
- This can be denoted $x \to \varepsilon$

Substitution of a single symbol

- Substitute x for a symbol $y \neq x$ changes uxv to uyv
- This can be denoted $x \to y$

Additional Operations:

Transposition of two adjacent symbols

- If $a = uv$, then transposing u and v to get $b = vu$

Merge of two identical symbols

- Merging the two u changes uuv to uv

Split of a single symbol

- Splitting the single v changes uv to uvv

Levenshtein Distance (Vladimir Levenshtein, 1965) only accounts for Insertion, Deletion, and Substitution.

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

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

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 $\mathrm{lev}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b. i and j are 1-based indices.

Wikipedia, Levenshtein Distance

Levenshtein Distance

```
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
  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]
```

Wikipedia, Levenshtein Distance

Although the distance is defined recursively, most recursive approaches for implementation are inefficient.

This pseudocode calculates the distance using the idea of a matrix where comparisons are made by a bottom-up dynamic programming approach like ones used for knapsack problems.

The last value of the matrix is the Levenshtein Distance.

(This is the Wagner–Fischer algorithm which is O(nm) where n,m are the lengths of the strings.)

Damerau-Levenshtein Distance

Damerau-Levenshtein Distance builds on Levenshtein Distance by also accounting for Transposition as well as Insertion, Deletion, and Substitution.

Useful for DNA comparisons because it supports these four operations.

To express the Damerau–Levenshtein distance between two strings a and b a function $d_{a,b}(i,j)$ is defined, whose value is a distance between an i–symbol prefix (initial substring) of string a and a j–symbol prefix of b.

The *restricted distance* function is defined recursively as:,[7]:A:11

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0 \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0 \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0 \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} & \text{if } i,j > 0 \\ d_{a,b}(i-2,j-2) + 1 & \text{if } i,j > 1 \text{ and } a[i] = b[j-1] \text{ and } a[i-1] = b[j] \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

Wikipedia, Damerau-Levenshtein Distance

Neighbor | Damerau-Levenshtein Distance

```
algorithm DL-distance is
    input: strings a[1..length(a)], b[1..length(b)]
    output: distance, integer

    da := new array of |Σ| integers
    for i := 1 to |Σ| inclusive do
        da[i] := 0

    let d[−1..length(a), −1..length(b)] be a 2-d array of integers, dimensions length(a)+2, length(b)+2
    // note that d has indices starting at −1, while a, b and da are one-indexed.

    maxdist := length(a) + length(b)
    d[−1, −1] := maxdist
    for i := 0 to length(a) inclusive do
        d[i, −1] := maxdist
        d[i, 0] := i
    for j := 0 to length(b) inclusive do
        d[−1, j] := maxdist
        d[0, j] := j

    for i := 1 to length(a) inclusive do
        db := 0
        for j := 1 to length(b) inclusive do
            k := da[b[j]]
            ℓ := db
            if a[i] = b[j] then
                cost := 0
                db := j
            else
                cost := 1
            d[i, j] := minimum(d[i−1, j−1] + cost,  //substitution
                               d[i,   j−1] + 1,     //insertion
                               d[i−1, j  ] + 1,     //deletion
                               d[k−1, ℓ−1] + (i−k−1) + 1 + (j-ℓ−1)) //transposition
        da[a[i]] := i
    return d[length(a), length(b)]
```

Wikipedia, Damerau-Levenshtein Distance

Similar implementation to the Levenshtein Distance with the Wagner–Fischer algorithm calculation from the bottom up.

Jaro Similarity

Jaro Similarity compares two strings to see how *similar* they are, whereas distances are usually measuring how *dissimilar* two strings are. It uses Transposition.

The Jaro Similarity sim_j of two given strings s_1 and s_2 is

$$sim_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases}$$

Where:

- $|s_i|$ is the length of the string s_i;
- m is the number of *matching characters* (see below);
- t is half the number of *transpositions* (see below).

Characters from the two strings are only considered matching if their placements (index) are no more than

$$\left\lfloor \frac{\max(|s_1|, |s_2|)}{2} \right\rfloor - 1$$

apart from each other.

Wikipedia, Jaro-Winkler Distance

Jaro-Winkler Distance

Jaro-Winkler Distance is the inverse of the Jaro-Winkler Similarity between two strings. The Jaro-Winkler Similarity is related to the Jaro Similarity and is weighted by a prefix scale/length.

Jaro–Winkler similarity uses a prefix scale p which gives more favorable ratings to strings that match from the beginning for a set prefix length ℓ. Given two strings s_1 and s_2, their Jaro–Winkler similarity sim_w is:

$$sim_w = sim_j + \ell p(1 - sim_j),$$

where:

- sim_j is the Jaro similarity for strings s_1 and s_2
- ℓ is the length of common prefix at the start of the string up to a maximum of four characters
- p is a constant scaling factor for how much the score is adjusted upwards for having common prefixes. p should not exceed 0.25, otherwise the similarity could become larger than 1. The standard value for this constant in Winkler's work is $p = 0.1$

The Jaro-Winkler distance d_w is defined as $d_w = 1 - sim_w$.

Wikipedia, Jaro-Winkler Distance

Other Similarity/Distance Metrics

Cosine Similarity
- popular in machine learning
- information retrieval

Trigram Similarity
- compare groups of three
- included in PostgreSQL

Hamming Distance
- only allows substitution

Graph Edit Distance
- treat the string as a graph or network
- edit operations are graph operations (e.g. remove node, add node)

Time Warp Edit Distance
- time series matching

Dynamic Time Warping
- time series matching
- used with beat sequences in musical data comparisons

In general, understanding the calculation of each distance metric is the best way to know **which one you should use** in your specific situation.

Do you need transpositions?

Don't use Levenshtein. Use something else.

Do you want to account for the prefix?

Use Jaro-Winkler or a variant.

Do you want to account for merges or splits?

Use an OCR accuracy metric.