A Fast Metric of Document Similarity

Peter Coates

September 13, 2011

Abstract: This document describes a heuristic for estimating the Levenshtein Distance *(LD) metric of sequence-similarity. With this technique, estimates of LD can be quickly computed for for large input strings such as Web pages, articles, book-length texts, etc, that may be orders of magnitude larger than the practical size input size limitation for LD itself.*

The heuristic works by first pre-processing the inputs into highly compressed string signatures, then running LD on the reduced inputs, and finally scaling the result by the compression factor. For typical text applications, the signatures may be compressed by a factor of from 25 to 100, which extends the size range over which LD is practical by the same factor.

Equivalenty, the time and space requirements to compute an estimate are inverse to the square of the compression rate. For example, with a compression factor of 100, the estimate is computed approximately $10,000$ *times faster than computing LD directly.*

1 Introduction

Levenshtein distance (LD) is a metric of sequence similarity. Most often applied to text strings, it is the "edit distance" between the strings. I.e., for strings S_1 and S_2, $LD(S_1, S_2)$ is the number of single-character additions, deletions, or substitutions, that are required to convert S_1 into S_2. Pairs of identical strings have an LD of zero, and the upper bound for LD is the length of the longer string.

Unfortunately, LD executes in time and space proportional to the product of the lengths of the two inputs ($O(m*n)$), making it an impractical metric for long input strings. For example, a simple implementation executing on a 2.6 GHZ laptop processed pairs of 40-byte strings at approximately $38,000$ pairs/sec, while strings of roughly 100 times that length, 4KB, were processed at only 3.7 pairs/sec—about 100^2, times more slowly.

Fortunately, for many purposes, a good approximation is enough. For such cases, it is possible to estimate LD to within a few percent by first compressing the inputs, then executing the algorithm on the shortened strings. The time saving is large—operating on shorter inputs accelerates the computation by the

square of the compression factor. The compression step is also quite fast, the run-time being linear in the length of the original string.

Moreover, because we don't need be able to re-inflate the compressed strings, we can choose an extremely lossy algorithm that compresses by factors of as much as 100 or more.

The most obvious processing advantage is relative speed of estimation over processing the string pairs naively with LD. Two less obvious speed advantages are also noteworthy: in applications where the cost of compression can be amortized over many distance tests, the load-time from disk is reduced, and the number of documents that can be represented in memory increased, by the compression factor[1].

2 The Heuristic

Two strings S_1 and S_2 are processed as follows:

1. Choose c, a compression factor, say, $c = 100$.
2. Choose n, a neighborhood size, say, $n = 10$.
3. Compress S_1 and S_2 into signatures Sc_1 and Sc_2, using the algorithm given below, parameterized by c and n.
4. Execute the standard LD algorithm on results of the previous step, i.e., $LD(Sc_1, Sc_2)$.
5. Scale the result by muliplying[2] by c.

2.1 Compression

We want a signature string that has the following properties:

1. The signature should be a pseudo-random string, the length of which is roughly proportional to the length of the input divided by c.
2. If a substring of the source results in an output character at a given position, changing or deleting the source substring should not change any character of the output that results from a source substring more than n characters away.
3. A source string should compress to the same sequence of characters, regardless of whether it is compressed in isolation, or compressed when embedded in a larger string, except for a maximum of n characters in at either end of the substring of the signature that results from the source.

[1] For instance, one-gigabyte of memory is sufficient to represent about 6.25 million documents the size of this one—less memory than would be required to store the same number of integers in a Java Map.

[2] When using LD as a metric of gross similarity, it is only the ratio of the calculated LD to the upper bound that we actually care about, rather than the integer LD score per se.

2.1.1 Compression Steps

A compressed signature with these properties can be computed as follows:

1. Choose an integer, k, $0 \leq k < c$.
2. Initialize an accumulator variable, s, to the sum of the first n characters.
3. Thereafter, starting at the n'th position and proceeding from left to right, one character at a time, until the string is exhausted, at each position, p:
   (a) Subtract the $p-1$'th character from s.
   (b) Add the the $p+n$'th character to s.
   (c) Compute a pseudo-random integer, $i = s \pmod{c}$, which is a psuedo-random hash of the current n characters.
   (d) If $i = k \pmod{c}$, emit character that is a function of i.
   (e) Otherwise emit nothing.
4. The resulting sequence of characters is the compressed string.

At each position in the input, this procedure will emit either a pseudo-random character ($1/c$ of the time) or nothing ($1 - 1/c$ of the time.) Thus, on average, we get a signature that is $1/c$ the size of the input.

If Σ is the input alphabet, and $\sigma = |\Sigma|$ is the cardinality of the input alphabet, only $\sigma n/c$ distinct sums will result in output.

For most purposes, in the useful range of c and n, both are small enough that each of the values in the set, $\{i \mid 0 \leq i < \sigma n, \quad i \equiv k \pmod{c}\}$, can be mapped in a look-up table to a distinct printable ASCII character.

Thus, compression is a fast, linear time operation, requiring only three arithmetic operations and a lookup, per character of input.

Note that given reasonable values of n and c, most small differences between two inputs do not result in *any* difference in the corresponding signatures, because the procedure emits a character on average for only $1/c$ of the character positions in the source.

3 Results

The heuristic was tested on an an earlier version of this LaTeX document having 7441 characters and 173 lines.

3.1 Speed

With the text pre-loaded into memory on a on 2.6 GHZ processor, plain LD applied to two copies of the test file executed at 1.08 pairs/second. With both compression and LD executing at each iteration[3], the execution rates on the same pair of files, for a range of compression values, were:

[3]For many applications, such as de-duplicating, the cost of the compressions step would be amortized across many comparisons.

- $c = 25$, 1000 pairs/sec.
- $c = 50$, 3030 pairs/sec.
- $c = 100$, 6666 pairs/sec.

3.2 Accuracy of Estimation

Comparisons were made with a range of compression levels, with the original 7441 character, 173 line, file, modified in several ways:

- Six widely separated lines were deleted from one of the files.
- The last 40% of one of the files was deleted as a block, in addition to six lines.
- A random 50 of the 173 lines were deleted from one of the files.

The results follow, showing the LD scores for the two files, and the percentage difference between the estimate and the actual LD score.

C	N	Changes	Actual LD	Est'd LD	Error
25	12	6 lines	217/7441	308/7441	0.01
50	12	6 lines	217/7441	430/7441	0.02
100	12	6 lines	217/7441	474/7441	0.03
25	12	6 lines + 40% block deleted	3173/7441	3313/7441	0.02
50	12	6 lines + 40% block deleted	3173/7441	3382/7441	0.03
100	12	6 lines + 40% block deleted	3173/7441	3957/7441	0.10
25	12	50 lines of 173 deleted	2606/7441	2864/7441	0.03
50	12	50 lines of 173 deleted	2606/7441	3259/7441	0.09
100	12	50 lines of 173 deleted	2606/7441	3008/7441	0.05

4 Limitations and Considerations

4.1 Unrelated Documents

LD, whether true or estimated, is not a good measure of similarity for files that are extremely different. This is because the LD of random text files of the same length is usually quite far from the theoretical upper bound, which is the length of the longer string. A significant percentage of the characters will almost always randomly line up in such a way as to not require an edit operation.

For example, LD for two different 13,877 character LaTeX *.tex* files was 10,863. The LD of one of these documents and an MS-Word document of the same length (but on a different subject) yielded almost the identical LD score, 10,866, and estimated values of LD for both of these pairs were very close to the true values for a range of n and c.

The closeness of all the values merely reflects the fact that the index-for-index variation between the pairs of strings, whether compressed or uncompressed, is highly random. Thus, a small amount of similarity in most pairs of strings will be lost in the the background noise.

However, either LD or the estimated LD *can* be used to determine the fact that strings are very different, even though neither effectively measures the amount of difference when it is very large.

4.2 Small Differences

The majority of small differences between the input files will cause no difference to the output (because only $1/c$ of the input character positions will result in a non-null character.) Therefore, if detecting that there is *any* difference is also important, a separate identity test, such as a cryptographic digest, should be used.

4.3 Number of Differences

The number of differences, rather than their cumulative size, is what most affects accuracy, because each difference is surrounded by a neighborhood of width $n-1$ characters that can be affected.

4.4 Compression Rate

The value of c affects both speed and sensitivity. The frequency with which the procedure misses small differences, and the relative effect of the differences that are detected both increase with c.

5 Applications

The technique may be useful for:

- De-duplicating: Finding variant Web-pages, text files, etc, and estimating their textual similarity.
- Forensic analysis: speed up searches of disk-drive contents.
  - Multiple versions of known, and therefore uninteresting, executables, system files, or other boiler plate can be identified with a single signature, sidestepping the need for separate cryptographic digests for every version[4].

[4] *The National Software Reference Library* (NSRL) provides a repository of MD-5 and SHA-1 file signatures of millions of software files for use computer forensic investigations.

  - Files including arbitrary fragments of text or other data from a target document can be quickly identified[5].

- Plagiarism detection: Finding occurrences of significant inclusion of target text embedded in other files.

- Intellectual property and data security: A remote service can use these techniques to support double-blind queries concerning whether a significant fragment of the client's content exists in a remote dataset.

  The party providing the service reveals nothing about what it has, other than *true/false* in response to specific queries about presence. Likewise, the client reveals nothing about the search target, except, obviously, in cases where it is found.

  This technique can be used to provide verification to members of the public that any specific piece of IP is or is not being stored in a library.

  The technique can also be used for data security; Such a service running in the background on each of an institution's workstations can provide a means of detecting the unintentional leakage of sensitive or classified information through cut-and-paste, embedding photos, renaming altered versions of a document, etc, and can do so without either the workstation or the institution side revealing sensitive information to the other.

6 Conclusion

Using a linear-time pre-processing step, this heuristic estimates the true value of LD($S1$,$S2$) for text strings many times larger than practical limit for LD alone.

While the runtime and space requirements for LD are both $O(n^2)$ for inputs of length n, the resource requirements for estimating LD are $O((n/c)^2)$, with c typically in the range of 10 to 100. This makes the technique applicable for files c times longer than can be processed with LD, or, seen the other way around, gives a speedup factor of c^2.

The technique is useful for rapidly identifying variant versions of larger text or binary files, Web pages, etc, and estimating the degree of similarity.

Estimated LD works well for detecting that files are similar, and measuring how similar. As with LD, the technique give the most useful metric of similarity when applied to relatively similar strings, because for very dissimilar strings, or strings in which many blocks have been reordered, shared text becomes difficult to distinguish from the distance measurements for mutually random strings.

[5] As noted elsewhere, if shared fragments of the target are small relative to the size of the search document, they may get lost in the background noise. However, a finer resolution can be achieved by breaking large files into blocks of some maximum size appropriate to the granularity at which the target text is interesting.