Estimating Levenshtein Distance Using Signatures

Peter Coates

November 24, 2015

Abstract: This document presents a heuristic that infers an estimate of the Levenshtein Distance of a pair of documents by comparing variable-length document signatures. The size of the signatures is configurable and is typically between 0.003 to 0.01 of the original document size. Using such signatures, the Levenshtein distance of pairs of documents such as Web pages, articles, etc., can be usually be estimated in a few tens of μ-seconds on generic processors.

This function of this heuristic differs from that of locality-sensitive hashing and similar approaches in to way: (1) It gives a numeric estimate of the degree of similary of pairs of texts, rather than a binary classification of related/not-related and (2) The locations of differences in the signatures correspond roughly to the locations of the corresponding differnces in the underlying documents.

1 Levenshtein Distance

Levenshtein distance (LD) is a measure of the similarity of two sequences. It can be applied to any kind of sequence, but in the interest of clarity, we will discuss it in terms of text strings. For strings S_1 and S_2, $LD(S_1, S_2)$ is their "edit distance," i.e., the number of single-character additions, deletions, or substitutions, required to turn S_1 into S_2.

Pairs of identical strings have an LD of zero. The upper bound for LD of a pair is the length of the longer string, but this can only be reached in special cases, for instance, for pairs of text that have no characters in common, or in which both have length zero, etc. The LD of pairs of random texts is otherwise always less than the upper bound, and for documents of the same length, in the same language, the ratio of the LD to the length of the text for random pairs groups fairly tightly around a characteristic mean.

LD is useful, but it has two drawbacks: (1) Using it requires that you be in possession of the original texts and (2) It is prohibitively slow for large texts because its performance is quadratic, i.e., it executes in time and space proportional to the product of the lengths of the two inputs ($O(mn)$).

The following example illustrates the implications of quadratic performance. A Java implementation of LD executing on a 2.6 GHZ, 8GB laptop executes LD

on pairs of 40-byte strings at approximately $38,000$ pairs/sec, while strings of roughly 100 times that length, 4KB, are processed at only 3.7 pairs/sec—about $100^2 = 10,000$ times more slowly. Files of length 50K proved too big to process at all because of memory limitations.

2 Estimated LD

Because the ratio of LD to document length varies little, an approximation of the LD of two files can be very useful, for instance, for determining whether two files are near-duplicates, or determining whether it is probable that two documents are more distantly related.

We estimate LD from variable-length signatures that are prepared by compressing the inputs in such a way that the LD of the signatures is in proportion to the LD of the two originals. The process need not be reversible, so very high compression rates can be used—anywhere from 25X to 300X would be used for text.

As LD is quadratic, shrinking the signatures results in a disproportionately large speedup–roughly the square of the compression rate. At such high compression rates, information is necessarily lost, so the estimate is rarely exact, but compression rates high enough for very fast estimates still give reasonably good results with text. Moreover, the results tend to be better in the cases where the LD is relatively small, which is usually what one is interested in.

For instance, the text file backing this document is about 20K characters in length, and the LD of two unrelated text files of that size should be about $14,666$. If the document's actual LD from a second document is 250, and the estimated LD is 190, this error, 0.76, normalized by the original file length is only 0.0351, i.e., about 3%. Conversely, if comparison to another file results in an estimated LD that is close to the expected value for random text, the documents are very unlikely to be near duplicates (although they can still be related in other ways.)

3 How It Works

The compression procedure described below produces an average of approximately one character of signature for C characters of input text. Text is processed in successive overlapping neighborhoods of size N, the size of which limits the amount of the output that can be affected by a local difference in the input.

Any size alphabet can be used for the signatures, even binary 0 and 1, but because LD's run-time is dominated by the number of elements in the sequence, the larger the output alphabet, the more efficient the computation. The only real limit on that principle is that the neighborhood size must be large enough that all possible characters in the alphabet can actually be used.

Eight-bit bytes or 16-bit integers are the easiest elements to deal with, because languages and hardware work most naturally in them. Therefore, for

purposes of illustration we'll talk in terms of characters, which in some languages are 8-bit, and in others are 16-bit. We will further limit the output to the printable subset of ASCII, so that the output can be inspected.

The considerations when choosing N and C in the algorithm described below are that the larger the value of C, the smaller is the signature, and the faster the estimate, but the lower the accuracy. Choose a C that is small enough to generate a meaningful amount of output, and large enough that the resulting signatures can be compared quickly.

A larger N causes the compression to consider more characters as "local" to a given position, which increases the entropy of the hashes that are used internally, but blurs the results, because a difference in the text affects output corresponding to as much as $2n$. Values of 8 to 20 are reasonable.

4 The Heuristic

Two strings S_1 and S_2 are processed as follows:

1. Choose c, a compression factor, e.g., $c = 100$.
2. Choose n, a neighborhood size, e.g., $n = 10$.
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. Interpret the result. How depends on what you are trying to achieve.
   (a) An LD estimate can be computed by adding the difference in the lengths of the signatures, to the length of the shorter signature, and multiplying the sum by C and by a "fudge factor" which is the expected proportion by which LD of equal-length random strings is smaller than their length.
   (b) To discover whether two files are related, compare their LD to the value expected for random text of the same lengths. The expected value function will vary by document type and must be determined empirically.

4.1 Compression

We want a signature string that has the following properties:

1. The signature should be a pseudo-random string that is much smaller than the orginal text.
2. The signature output for a given substring of the input should not be affected by any character in the input that is more than a fixed distance away.

3. An input 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 at most a bounded number of characters at either end.

4.1.1 Compression Steps

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

1. Choose an arbitrary integer, k, $0 \leq k < c$.
2. For each position, p, in the input, from 0 to *input.length-n*:
   (a) Hash the n-length substring starting at position p to a pseudo-random integer, H_p.
   (b) If $k = H_p \pmod{C}$, emit a character that is a pseudo-random function of H_p.
   (c) Otherwise emit nothing.
3. The resulting sequence of characters is signature.

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

The hash function itself can be anything that thoroughly scrambles n characters into a pseudo-random value. For the demo quoted below, for each neighborhood, each of the *char* values is successively XOR'd into a long integer, at a position shifting left by eight bits each time, with the position wrapping around if the number of shifts exceeds eight.

If the output alphabet is a subset of ASCII, of size α, the pseudo-random function to choose an output *char* from H_p can be implemented as simple array of chars, and the choice made by selecting the value at the index $H_p \pmod{\alpha}$

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.

5 Test Results

Tests were performed on text pre-loaded into memory on a MacBook Pro with 2.6 GHZ processor. All tests were executed on a single thread.

A 4,865 character text string was compressed 10,000 times, at $C = 101$, in 3,178 milliseconds, i.e., 3146 *file compressions/sec*, or, 15,308,239 *chars/sec*. The effect of using different values of c on compression is negligible.

Estimated LD was computed for pairs of versions of the a 27KB text files (from a Gutenberg book) that had been modified in various ways—a few random lines deleted, blocks of lines deleted, random words deleted, etc. The table below

gives the original file sizes, the actual LD, the rate of computation for LD, the estimated LD, the rate of computation for the estimate, and the normalized error of the estimate[1].

The compression rate was $C = 101$ and $n = 9$. Notice that the estimates are about 9,700 times faster than computing LD itself, which is about what would be expected from a compression rate of 101.

f-1	f-2	LD-orig	LD/sec	Est	Est/sec	Err
24,767	24728	39	0.4026	0	3906	0.0016
24,728	23,624	1192	0.4013	1364	4032	0.0069
24,728	23,979	767	0.4002	876	3898	0.0044
24,453	22,111	2948	0.4755	4236	5277	0.0527

Here is the same table again, with $C = 200$. Note that the error has increased somewhat, and the speed has quadrupled.

f-1	f-2	LD-orig	LD/sec	Est	Est/sec	Err
24,767	24728	39	0.4075	0	12,738	0.0016
24,728	23,624	1192	0.4075	607	13,422	0.0237
24,728	23,979	767	0.4523	810	16,806	0.0017
24,453	22,111	2948	0.5128	6162	20,408	0.1314

And again, with $C = 307$. Note that the error rate doesn't seem much worse, but the speed has increased by a factor of 2.6, which is almost exactly what you'd expect for a 50% increase in compression rate (2.25.)

f-1	f-2	LD-orig	LD/sec	Est	Est/sec	Err
24,767	24728	39	0.3891	0	33,898	0.0016
24,728	23,624	1192	0.4307	1192	42,553	0.0338
24,728	23,979	767	0.3942	356	44,444	0.0166
24,453	22,111	2948	0.4335	4286	46,511	0.0547

[1]Normalized, in this context, means adjusted for how much of the file was affected—e.g, an estimate of zero, and an actual LD of 39 is a very small error in a 24K file.

6 Limitations and Considerations

6.1 Unrelated Documents

A significant percentage of the characters in real text will almost always randomly line up in such a way as to not require an edit operation. Therefore, a small amount of similarity cannot be distinguished from 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.

6.2 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 result in a non-null character.) Therefore, if detecting that there is *any* difference is also important, a separate identity test, such as comparing cryptographic digests, should be used. Using both can be useful because estimated LD will usually report that the inputs are the same if they differ only very slightly.

The number of differences can have a bigger effect on accuracy than the cumulative size of the differences, because $n - 1$ positions on either side can be affected. Adjusting N and C can help tune the sensitivity.

7 Applications

The technique may be useful for:

- De-duplicating: detecting variant Web-pages, text files, etc, and estimating their textual similarity, as well as verifiying and categorizing relationships detected by other duplicate detection algorthms.
- 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[2].
  - Files including arbitrary fragments of text or other data from a target document can be quickly identified[3].

[2] *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. Many trivially different versions of a given file often exist, differing only by dates or versioning information.

[3] As noted elsewhere, if shared fragments of the target are small 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.

- 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 present, even in partial or modified form.

  The same 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 in useful form.

8 Conclusion

After a one-time linear-time pre-processing step, this heuristic estimates the true Levenshtein edit-distance of text documents up to a few hundred times larger than would be practical to compare using LD alone.

The technique can be used for detecting near-duplicates, verifying the results of other duplicate detection algorithms, detecting more distant relationships between text files, such as inclusion of one file in another, and highly edited document variants.

When applied to detecting near-duplicates, errors of both the first and second kinds are negligible.