A Fast Metric of Document Similarity

Peter Coates

September 13, 2011

Abstract: This document presents a heuristic for quickly estimating the edit-distance of document pairs, by comparing metadata associated with each file. Metadata for typical Web pages is from five to twenty bytes in size, and edit-distance for a pair of documents can be estimated in less than 100μ *seconds on generic processors.*

Locality-sensitive hashing, shingling, and similar heurisics for high-speed duplicate detection, tend to excel at making a fast, binary decision about whether a document duplicates one that has been seen before, but they have relatively high rates of errors of both the first and second kind.

This heuristic differs from such algorithms in that it gives a flexible measurement of the degree of similary of pairs *of texts, with very low error rates. Therefore, it is ideally suited for removing type-one errors from the ouput of de-duplication algorithms. It is also capable of detecting more complex relationships between texts than near-duplication. For instance, it can be used to recognize that one text embeds another, to estimate where and/or how diffuse the differences between two documents are, or to recognize that two files are textually related, despite being very different.*

1 Introduction

Determining whether or not two documents are exactly the same is a linear-time operation, i.e, two texts can be compared more or less as fast as they can be read.

If you will be testing the same texts repeatedly, and can tolerate a minute possibility of error, you can do much better. Texts can be hashed to smaller values, such as 64-bit or 128-bit integers, in linear time. Thereafter, duplicates can be detected by comparing the fixed-size hashed metadata. If two documents are identical, the hashes will always be identical too (although there is a vanishingly small chance of a false positive, i.e., that non-identical documents will be categorized as identical.)

Not only is this technique computationally efficient, once the hashes have been prepared, you no longer need to be in possession of the originals to execute the comparison. Moreover, you can structure your metadata collection for efficient search, e.g., by sorting, or by insertion into a hash table or set.

Unfortunately, this strategy only works for documents that are identical down to the last bit.

Detecting near-duplicates, i.e., pairs of documents that are almost the same, is a closely related, but more difficult task. The need for such near-duplicate detection arises in many applications including: Web crawling (the Internet is rife with duplicate pages,) merging large document databases, detecting plagiarism and other misuse of textual data, etc.

There are numerous techniques for generating metatdata that achieve for near-duplicates much of what hashing does for exact duplicate detection; SimHash, shingling and locality-sensitive hashing are examples. While these techniques allow incredibly fast detection near duplicates, unlike hashing, they tend to suffer from high error rates, both for false-positives and for false negatives (a.k.a, errors of precision and recall, or type-1 and type-2 errors.) For instance, the duplicate detection algorithm used by Google's Web crawlers, which must quickly find duplicates among tens of billions of already-seen Web pages, reportedly has about a 25% error rate for both false postiives and false negatives.

1.1 Levenshtein Distance

Levenshtein distance (LD) is one of many metrics of sequence similarity that could in principle be used for accurate near-duplicate detection. Most often applied to text strings, LD 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[1].

Edit distance is a great way to define whether two documents are near-duplicates, but it has two drawbacks: (1) it is not metadata based, meaning you need possession of the original texts, and (2) it is very slow to compute for large texts, executing in time and space proportional to the product of the lengths of the two inputs ($O(m*n)$). 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 $100^2 = 10,000$, times more slowly.

2 Estimating LD

Even a rough approximation of LD would be more than enough to determine whether two files are near-duplicates. Surprisingly, in the most useful cases, it is possible make a reasonably accurate estimate the LD of pairs of documents very quickly, from a small amount of metadata.

[1]The upper bound is rarely reached unless the files are deliberately constructed to be maximally distant, e.g., texts that have no characters in common, or one text has length zero, etc.

The metadata is prepared by compressing the inputs in such a way that the LD of the compressed strings is approximately proportional to the LD of the originals. Because we don't need be able to re-inflate the compressed strings, we can choose a very lossy algorithm.

The compression procedure described below would typically be set to produce approximately one bit of metadata[2] for C bytes of text, with C chosen between 25 and 200[3] The main condition is that the files must be large enough for the chosen C.

A value of C=100, adequate for Web pages, and would yield approximately 100:1 logical compression, but about 800:1 of physical compression. Therefore, a 10KB Web page would yield about 13 bytes metadata.

The time savings for compression is large, because LD is a quadratic operation. This means that compressed inputs accelerate the computation by the *square* of the compression factor (C). For the example above, with $C = 100$, pairs of 10KB Web pages, can be compared at a rate of several thousand pairs/second.

3 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[4] by c.

3.1 Compression

We want a signature string that has the following properties:

1. The signature should be a pseudo-random string of bits that is much smaller than the orginal text.

[2] Although the algorithm can use any size alphabet for output, we describe here the use of only two output characters, 0 and 1. Two characters are convenient because they are easy to pack into bit-strings, and provide the most information per bit of output, which is advantageous for transmission and storage. On the other hand, the two-character alphabet is not optimal for processing speed, which is primarily a function of the number of characters in the two strings. There is more information per character in a diverse alphabet.

[3] For text—much higher compression rates can be used for binaries.

[4] 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. If a substring of the input results in an output bit, no change to the input that is more than a fixed number of characters distant should affect it.

3. A source string should compress to the same sequence of bits, regardless of whether it is compressed in isolation, or compressed when embedded in a larger string, except for at most a few bits in at either end.

3.1.1 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) Remove the $p-1$'th character from s.

   (b) Append 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 a bit that is a function of i.

   (e) Otherwise emit nothing.

4. The resulting sequence of bits is the compressed string. They would ordinarily be stored in a bit field.

At each position in the input, this procedure will emit either a pseudo-random bit (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.

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 1 or 0[5].

Thus, compression is fast 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 bit on average for only $1/c$ of the character positions in the source.

[5] We are using 0 and 1, but in general, it can be any alphabet, such as a the set of printable ASCII characters.

4 Test Results

The heuristic was tested primarily on an an earlier version of this LaTeX document having 13,508 characters and 315 lines. Corrupted versions of this file, and other files of the same size, but with unrelated contents, were also used in some tests.

4.1 Speed

With the text pre-loaded into memory on a on 2.6 GHZ processor, plain LD, applied to a pair of uncompressed 13,508 byte test files, executed at 0.87 pairs/second.

Similar documents (also preloaded into memory to eliminate file-reading time) were compressed (on a single thread) at between 7,700 and 10,000 documents per second (104,000,000 to 135,000,000 characters/second,) depending upon compression rate.

Estimaed LD was computed for the same pairs of files, using metadata prepared with a range of compression rates, on unrelated 13,508 character text files. The table blow gives the resulting signature sizes and computation rates.

C	N	Pairs/sec	Metadata-bits	Metadata-bytes
50	4	1,388	316	40
100	4	10,000	72	9
200	4	20,000	22	3

4.2 Accuracy of Estimation

To test sensitivity and recall, comparisons were made with a range of compression levels, with the original 13,508 character file, modified in several ways:

- Random characters were substituted for $1/p$ of the characters in the text, for a range of p from 0.01 to 1.5.
- The text was modified by adding from one to ten blocks of 500 charaters.
- The original 13K text was compared to itself with an equal-length document concatenated onto the beginnning and the end.

4.2.1 Numerous Small Changes

The following table shows the results of estimating LD on pairs of texts that have a range of proportions of single characters changed at random. This test simulates many minor editing differences, light formatting, etc. Results for a range of compression rates are shown.

The fields are: compression rate, neighborhood size, percentage of random chars that differ, the true LD for the documents, the estimated LD, the LD

computed for two completely unrelated files of this size, and the the estimated LD and the LD of unrelated files of the same size.

The LD of the two 13,508 character texts, uncompressed, is 12,430. This is a difficult case for estimated LD, because the numerous small changes affect a large proportion of the neighborhoods in the sample texts.

C	N	% Chars Subst'd	True LD	Est'd LD	Max LD	Error
50	4	1	136	333	12,430	0.0267
50	4	5	777	1,000	12,430	0.0864
50	4	10	1,438	2,416	12,430	0.1943
50	4	20	2,689	2,350	12,430	0.1890
50	4	30	4,083	3,700	12,430	0.2976
100	4	1	119	66	12,430	0.0483
100	4	5	764	600	12,430	0.2976
100	4	10	1,451	1,066	12,430	0.0859
100	4	20	2,740	1,666	12,430	0.1343
100	4	30	3,981	1,833	12,430	0.1478
200	4	1	115	66	12,430	0.0053
200	4	5	814	200	12,430	0.0161
200	4	10	1,461	266	12,430	0.0322
200	4	20	2,841	800	12,430	0.0645

4.2.2 A Few Large Changes

The following table shows the results of estimating LD on pairs of texts that have a several large blocks of random text inserted throughout. The original file is 13,508 characters, and the modified files are up to 5000 characters larger. This test simulates what might be seen when a plain text document is converted to one or more HTML page, a Web page is heavily modified. Results for a range of compression rates are shown.

The fields are: compression rate, neighborhood size, the number of added blocks, the size of the blocks, the actual LD, the estimated LD, and the ratio of the error of the estimate to the LD of the original from a random text document of the same size as the modified file. The smaller the value in the last column, the better the LD estimate.

C	N	Num-Blocks	Block-Size	True-LD	Est'd-LD	Est'd-LD/Max
200	4	1	500	500	200	0.0274
200	4	5	500	2,500	2,200	0.0245
200	4	10	500	5,000	6,000	-0.2401
100	4	1	500	500	700	-0.0182
100	4	5	500	2,500	2,800	-0.0246
100	4	10	500	5,000	4,700	-0.0145
50	4	1	500	500	700	-0.0182
50	4	5	500	2,500	2,100	0.0327
50	4	10	500	5,000	5,450	-0.0690

5 Limitations and Considerations

5.1 Unrelated Documents

LD, whether true or estimated, is not a good measure of similarity for files that are extremely different, because the LD of random text files of the same length is usually far from the theoretical upper bound, which is the length of the longer string. 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,or will align after an insertionor deletion edit is done elswhere. Therefore, a small amount of similarity is hard to distinguish 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.

5.2 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 a cryptographic digest, should be used.

The number of differences, can have a bigger effect on accuracy than the cumulative size of the differences, because each difference is surrounded by a neighborhood of width $n - 1$ characters that can be affected.

5.3 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.

6 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 algorithms.
- 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[6].
  - Files including arbitrary fragments of text or other data from a target document can be quickly identified[7].
- 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, in a large data set.

  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.

7 Conclusion

After a one-time linear-time pre-processing step, this heuristic estimates the true Levenshtein edit-distance of text documents many times larger than would

[6] *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.

[7] 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.

be practical to compare using LD alone. Estimated LD can be used for detecting near-duplicates, removing false postives, and otherwise tightening up the output of other duplicate detection algotrithms, and for detecting more distant relationships between text files, such as inclusion of one file in another, and highly edited document variants.

The metadata for comparing files is small—half a dozen to a dozen bytes per 10K of text—and computationally cheap to prepare. Web pages, articles, and similar text documents of a few tens of kilobytes can be compared at a rate of many thousands of pairs per second.

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