Identifying document similarity using a fast estimation of the Levenshtein Distance based on compression and signatures

Peter Coates, **Frank Breitinger**

Presentation for DFRWS EU, Oxford

DFRWS
DIGITAL FORENSIC RESEARCH CONFERENCE

UNIL | Université de Lausanne

Are these files related?

Filesize 1.2 MB

Filesize 1.2 MB

Idea 1 – crypto hashes
Problem: Crypto hashes only work for exact matches

Idea 2 – approximate matching
Problem: work but only return only a “certainty score”

Idea 3 – Levenshtein distance

Filesize 1.2 MB

Filesize 1.2 MB

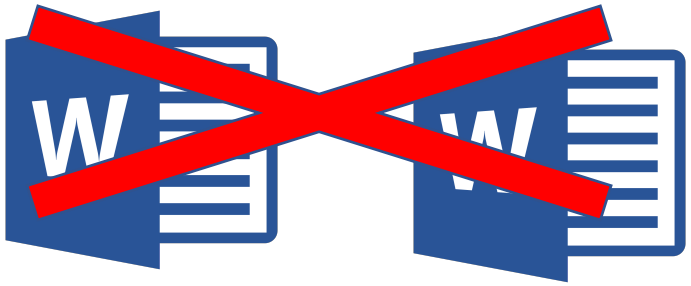
Precise measure of the (dis)similarity of two strings:

Given a pair of strings, the LD is the number of single-character edits (i.e., insertions, deletions, or substitutions) that are required to turn one string into the other.

Example 1: AAA → BBB LD=3

Example 2: Beer → Fear LD=2

Levenshtein distance cont'd



Filesize 1.2 MB

Filesize 1.2 MB

Problem: Quadratic runtime complexity $O(|a| * |b|)$

Practical impact: two strings with 50'000 characters each requires already 3.5 sec on a modern Laptop → comparing larger strings, such as Web pages, long articles, or books, becomes impractically slow

1.2 MB ~ 1'200'000 characters

There are some optimizations which we will ignore for this presentation

Contribution

A heuristic to **estimate the LD** of texts pairs over a size range many hundreds of times larger than is practical for calculating an exact value

- Performance (Settings) can be adjusted using parameters
- A GoLang implementation is available (Python / Java exists)
  - Python is slooooooow

The Algorithm

The LD estimation of two documents is done in three steps:

(1) compressing each document into a signature using a Lossy Compression Algorithm (LCA)
   - Output is a pseudo random looking “text” based on a defined alphabet

(2) applying the (original!) **LD algorithm** to the compressed signatures (they are text)

(3) scaling the result back by the “compression rate”

```
Text Documents
        ↓
Sig1: 8aHks8+Sv …
Sig2: 8aHKs6zSl…
        ↓
LD(sig1, sig2) = eLD
        ↓
eLD + some math =
approximate LD
```

Lossy Compression Algorithm (LCA)

- Desired properties for LCA:
  - Compression, Determinism, Runtime efficiency, and Concatenation
    - Concatenation: $dig(A) + dig(B) = dig(A + B)$
      - Proposed algorithm does not fully satisfy the fourth property; we settle for "good enough"
- Algorithm uses three parameters:
  - C = Compression rate (common values are 20, 50, 100, 1000)
    - If C is too large, too much information is lost, and the digest provides little value
  - N = Neighborhood (common values are 11 to 21)
    - The larger N, the more sensitive the heuristic will be to small differences because each character is part of the N distinct neighborhoods
  - ALPHABET
    - Given an Alphabet = [a..zA..z0..9], signature could look like AeVVgCAe2aZUpa6dnEkK..
      - Longer Alphabet is desired wherefore we include more characters e.g., +&#

LCA procedure

Let S be the input of length L, and let S_p denote the current position in the input

1. dig (lossy compression digest) denotes a string containing the digest, which is **initially empty**.
2. If $p + N \geq L$, print dig and quit (p *is a counter an* initially zero).
3. Compute $H_N(S_p)$ and store the result in T .
4. If $(T\ mod\ C) == 0$,
   1. (a) generate $tmp = T\ mod\ len(ALPHABET)$, and
   2. (b) append $ALPHABET[tmp]$ to dig.
5. Set $p = p + 1$ and return to step 2.

For hashing (H), our implementation uses Rabin-Karb but any fast algo is possible

$p = 0;\ \ sig = „“;\ \ N = 7;\ \ C = 101;\ \ ALPHABET = \{A, C, a, b, 1,2, +\}$

S = H E L L O _ W O R L D !

$p = 0$ (iteration 0)

$T = H_7(S_0)\ mod\ C$

$H_7(HELLO\_W) = 1069\ mod\ 101 = 59$

$(T\ mod\ C)\ != 0$

continue

$p = p + 1$

$T = H_7(S_1)\ mod\ C$

$H_7(ELLO\_WO) = 715\ mod\ 101 = 8$

$p = p + 1$

$T = H_7(S_2)\ mod\ C$

$H_7(LLO\_WOR) = 303\ mod\ 101 = 0$

$tmp = \ ALPHABET[303\ mod\ 7]$

$sig = sig + tmp$

$7 = len(ALPHABET)$

$sig = „a“$

$p = p + 1$

$T = H(S_3)\ mod\ C$

$H_7(LLO\_WOR) = 801\ mod\ 101 = 94$

$p = p + 1$

....

A bit similar to ssdeep and its rolling hash

$sig = „aAC1 + +2B“$

eLD Signature

Final eLD signature comprises of a *header* as well as the *LCA digest*

Header necessary to keep track of the “parameters”

fileLength + digest are mandory

```
$ go run *.go -in test-data/doc.txt
#filename, fileLength, C, N, digestLength, digest

test-data/doc.txt,13680,101,20,152,7n(n[(9&dU7;aRZaPgSGWzoFCC_r
{FUB;A]v?dGrL.bQZ!3GCT)V0r>XWpPNmQ6>#8YhTN]h-X598u+qw1Q1K:+&CVI
gIhCK//j2 ...
```

Estimating the LD

- To estimate the LD of two documents given their eLD signatures, we first compute the LD between the digests and then perform a scaling where ideally $LD(A, B) \approx eLD(A, B)$

- <u>Unscientific description</u> of the scale: multiple by the compression rate

- Three considerations impacting the scaling:
  1. Documents have roughly the same size and therefore the generated digests shall also have approximately the same size.
  2. Documents have a large difference in size (e.g., ratio 1:5) which also results in a large difference of the digests length → difference is the minimum LD
  3. Unrelated documents (usually) have a LD shorter than their length

All details about the estimation are motivated and explained in the paper.

$$\texttt{digDiff} = |dig_A| - |dig_B|$$

$$\texttt{effectiveC} = (|A| + |B|)/(|dig_A| + |dig_B|)$$

$$\texttt{digLD} = LD(dig_A, dig_B)$$

$$\texttt{scaledDigLD} = (\ \texttt{digLD} - \texttt{digDiff}\ ) * \texttt{effectiveC}/(1 + R_{words})$$

$$\texttt{fileLengthDiff} = |A| - |B|$$

$$eLD = \texttt{round}(\ \texttt{scaledDigLD} + \texttt{fileLengthDiff}\ )$$

Does it actually work?

Evaluation

(based on the GoLang implementation)

Preliminary remarks

- Signature generation runtime is impacted by the Hash function and N
  - Compression C does not have an impact on the generation
- Signature comparison runtime impacted by the length of the signature
  - Compression C has an impact
  - N does have no impact
- Precision is impacted by C, N and other parameters

Signature generation runtime efficiency

- Depends on the efficiency of the hashing function (remember each neighborhood is hashed)
- Average runtime for different hashing algorithms using three different N and a constant C = 301
  - *Tests for FNV and MD5 utilized GoLang libraries and required type-casts which may have impacted runtime

18,216 files ranging from 10 KB to 12 MB totaling in approximately 7.2 GB.

N	7	14	21
Rabin Karb	42s	43s	42s
djb2	1m 55s	2m 35s	3m 18s
FNV*	2m 13s	2m 39s	3m 21s
MD5*	19m 18s	19m 41s	19m 39s

Signature comparison runtime efficiency

- Time for an all-against-all comparison (i.e., 780 comparisons total) for various C's
  - N = 11
  - csv size is the file size of the signature

C	duration	csv size
51	2m 12s	342 KB
101	34s	175 KB
301	4.2s	59 KB
501	2.0s	37 KB
1001	0.9s	19 KB

Randomly selected 40 files (between 22 KB and 1.2 MB) from our dataset with a total of 17.4 MB

Estimated LD vs. LD (related files)

#			Operation	org. FS	LD	C=11	C=21	C=51	C=101
1	10348.txt	10348mod.txt	Deleted 10 lines	24226	663	718	798	709	663
2	ltplt10.txt	ltplt10mod.txt	Deleted 10 lines	22869	634	672	652	717	719
3	7563.txt	7563mod.txt	Deleted 51 lines in the beginning	23414	1024	1024	1024	1024	1024
4	7565.txt	7565mod.txt	Deleted 101 lines in the middle	25983	2234	2234	2234	2234	2330
5	7906.txt	7906mod.txt	Deleted 90 lines (some beginning; some end)	21586	3329	3329	3329	3329	3329
6	10630.txt	10630mod.txt	Deleted 7 chunks	32185	7085	7095	7103	7086	7086
7	17282.txt	17282mod.txt	Deleted 3 large chunks	25534	7436	7436	7436	7436	7436
8	18232.txt	18232mod.txt	Deleted 15 times 3 lines	31635	2964	3117	3022	3034	3165
9	8528.txt	8528mod.txt	Deleted approx first half	33625	15811	15811	15811	15811	15811
10	11592.txt	11592mod.txt	Deleted 10 paragraphs	36253	4224	4262	4259	4224	4224
11	16637.txt	16637mod.txt	Deleted 5 paragraphs at the beginning	39864	1731	1740	1731	1731	1731
12	wlett10.txt	wlett10mod.txt	Deleted big middle chunk	32346	8968	8968	8968	8968	8968
13	lf17w10.txt	lf17w10mod.txt	Deleted all (450) 'the'	38902	1215	5409	5749	7128	7156
14	haw4610.txt	haw4610mod.txt	Inserted 10 random 'A'	31340	10	93	113	96	10
15	haw7810.txt	haw7810mod.txt	Inserted 40 random 'A'	36281	40	531	562	819	390
16	lf20w10.txt	lf20w10mod.txt	Swapped 6 paragraphs around	37765	3218	2831	2969	2897	1559
17	14814.txt	14814mod.txt	Swapped first and second half (approx)	28050	17517	14128	14454	15134	17721
18	12337.txt	12337mod.txt	All 'b' (338) replaced with 'B'	31241	338	4204	3172	3004	3248
19	17195.txt	17195mod.txt	All 'e' (3080) replaced with 'E'	35158	3080	21629	21605	22510	20106
20	13322.txt	13322mod.txt	All spaces (6572) replaced with double-space	38838	6572	34083	35883	31128	32218

Well for deletions but problems when spread out

Random inserts are a problem

Swapping is okay

Problem with many minor changes throughout

Estimated LD vs. LD (loosely related files)

	LD	$C = 11$	$C = 21$	$C = 51$	$C = 101$	$C = 201$
Duration	9min 09s	3.5s	1.2s	0.6s	0.4s	0.3s
Avg Error (abs/%)	–	1223 / 6.5%	1166 / 6.4%	1519 / 9.0%	1578 / 9.0%	1650 / 9.4%
Min Error (abs/%)	–	16 / 0.1%	10 / 0.1%	10 / 0.1%	3 / 0.0%	16 / 0.1%
Max Error (abs/%)	–	3498 / 23.2%	3484 / 23.1%	5177 / 34.3%	4187 / 40.7%	6942 / 35.6%
Std Dev. (abs/%)	–	822 / 3.6%	767 / 3.8%	1114 / 6.4%	1075 / 7.0%	1424 / 7.3%
Error rate (avg/std dev)		0.03 / 0.02	0.03 / 0.02	0.04 / 0.03	0.04 / 0.02	0.05 / 0.04

Randomly selected 20 files (between 20 KB and 40 KB) from our dataset with a total of 627 KB

Last row puts “error rates in perspective” where low is good

$$ER = abs(\frac{LD(A,B)}{max(|A|,|B|)} - \frac{eLD(A,B)}{max(|A|,|B|)}).$$

Discussion of evaluation

- Evaluation focused on C, more tests are needed for N and R
  - R is the expected_overlap_text; a parameter for the upscaling describing the average ”overlap” (
    - R is a calculated value
    - $\mathrm{R} = 1 - \frac{LD(S_1,S_2)}{\mathrm{len}(\mathrm{S}_2)}$) (note: $len(S_1) = len(S_2)$)
- Algorithm could output $\mathrm{LD} = 0$ although there are minor differences; it will never output $\mathrm{LD} > 0$ if the texts are identical
- Parameters allow to adjust the algorithm to the given problem
- Does not work well for short texts
  - Fall back: use original LD

Significance score

How do we know how similar / dissimilar two inputs are?

- Correlate the estimated LD with the length of inputs
- Assuming $len(dig_A) \geq len(dig_B)$ then $delta = \frac{(len(dig_A) - ld)}{len(dig_B)}$
  - Similar if delta < Threshold
  - Weakness: does not work for digests with a large difference in size

Temporary fix: only for files that do not differ too much; future work.

		$len(dig_A)$	$len(dig_B)$	$LD(dig_A, dig_B)$	$delta$
"good matches"	1	700	700	0	1.000
	2	700	700	10	0.986
	3	700	350	400	0.857
	4	700	100	600	1.000
"poor matches"	5	700	700	600	0.143
	6	700	350	650	0.143
	7	700	100	696	0.040
	8	700	200	700	0.000
fails	9	70'000	700	70'000	0.000
	10	70'000	700	69'650	0.500
	11	70'000	700	69'300	1.000

L9 and 10 do not happen with real world text; it will always be like L11

Take home messages

- Levenshtein distance estimation (eLD) is a novel concept of comparing large text documents
  - Prototypes (various languages) are available
- While more testing/evaluation is required, this technique may complement hashing/approximate matching providing more granular results
- Optimizations with respect to the estimation precision may be possible (i.e., get “closer” to the real LD)

Questions

Thank you!

vCard

Frank Breitinger

Frank.Breitinger@unil.ch