Mining, Measuring, and Managing Textual Variation with TAN Diff+

Preliminary Comments

Joel Kalvesmaki

In my seminar paper I will be presenting TAN Diff+, an XSLT-based application for text comparison and analysis. These preliminary comments provide important background.

Text Comparison

When we compare texts, we are looking for similarities and differences. Our expectations are shaped by the questions and goals that motivate us.

Many authors, editors, and writers use text comparison to understand specifically how a document has changed from one version to another, and to reconcile those differences. Lawyers or forensic specialists may be interested not in reconciling differences but in tracing their history. Companies may want to measure the accuracy of keyboard operators, or to compare and grade the output from different optical character recognition configurations. Philologists may want to compare multiple versions of a particular text, to establish the phylogenetic relationships between manuscripts, and to create a *stemma codicum*. Computer programmers may want to assess different versions of the same source file, perhaps to identify when a particular bug entered the code base. Biologists may want to compare DNA sequences as textual differences, to identify mutations and abnormalities between test samples.

With so many different motives and fields, it is difficult to say precisely what text comparison should entail. Should a comparison ignore case or space or certain types of spelling? How granular should the textual atom be: stroke, character, word, line, or something else? If quantification of text difference is required, upon what metric? (See Huitfeld and Sperberg-McQueen 2020 for an overview.) Should scores be weighted, and if so, how?

Computer-based methods of comparing texts are ubiquitous and indispensable. They are also abundant (see below). For 95% or more of situations that require close comparison of texts, one or more of the many customer off-the-shelf software packages are sufficient. But pre-packaged software is commonly inadequate for complex needs, for example, the need to…

- …compare three or more versions of the same text;
- …normalize or ignore specific differences in spacing, punctuation, or language-specific orthography;
- …collect, process, and analyse statistical differences;
- …adjust scoring differences, based on weighted criteria;
- …modify results in light of detectable transpositions;
- …compare each text not as a linear sequence but as a hierarchical tree.

In such cases, one nearly always needs to start with a text-difference function library in a programming language (see below), and develop an application tailored to the needs of a specific project or workflow. Such an approach can be challenging to implement by anyone who is uncomfortable outside of a graphic user interface, because it almost always requires elementary competency in a given programming language, or collaboration with colleagues who have such expertise. It also requires extra time. But if the endeavour is successful, the investment pays off handsomely, producing results and insights that pre-packaged solutions cannot.

Algorithms

The sciences and the humanities have a great deal of common interest in the area of textual differences. Although one might think that text difference is fundamentally a problem in the humanities, science has pioneered the technological solutions, particularly through biologists and mathematicians (Sankoff and Kruskal 1983). Scientists frequently need to compare protein sequences, birdsong samples, gas chromatography readings, and other data that can be expressed as sequences. If sequences members are limited (such as G, A, T, and C in DNA sequences), the task can be framed as a textual difference problem, with the sequence items expressed as letters in an alphabet.

Calculation of the difference between two sequences has sometimes been expressed as a search for the shortest edit script, i.e., the least number of steps required to go from sequence 1 to sequence 2. This manner of phrasing the problem, however, raises questions about how steps are calculated and quantified. If the deletion or insertion of a single character counts as one edit step, does a replacement of a single character count as two edit steps or one? The problem is framed in terms of ergonomics, so one might justifiably complain that replacements should not be treated equally, because some edits are more costly than others (e.g., a change from 3 to 四 versus one from 3 to 4). Cannot one simply do everything in two steps? Delete the entire first sequence, insert the second.

Therefore the task of text differencing is more commonly framed as the search for a *longest common subsequence* (LCS), i.e., a common subsequence that is not surpassed in length by any other common subsequence. For example, (a, b, c) and (a, d, c) have three common subsequences—(a), (c), and (a, c)—with (a, c) being the longest. Do not confuse the LCS for the longest common contiguous subsequence (LCCS), which is the longest uninterrupted common subsequence. For example, in the sequence pair (a, b, ., x, y, z, ., c, d) and (x, y, z, _, a, b, _, c, d), the LCS is (a, b, c, d) but the LCCS is (x, y, z).

Algorithms designed to find the LCS begin by casting the two sequences as axes of a matrix, and then populating that matrix with values. Paths are drawn through the matrix of values. Those paths with the lowest score are the LCS's. A classic description of this algorithm is provided by Myers 1986, the starting point for many implementations. The procedure is at best quadratic in time complexity (i.e., based upon the product of the lengths of the two input sequences). When the length of the two input strings are around tens of thousands of characters or fewer, results can be had in a split second. But when the length exceeds one million characters, the classic algorithm can be punishing to run.

In 2021 I presented what I called a staggered-sample algorithm. To my knowledge this algorithm has not been published or implemented before. (I still find this difficult to believe, and I would welcome feedback.) The algorithm imitates how I think humans normally compare two long texts, by eyeballing the big picture, then gradually drilling down into details. We start with a large sample from the smaller string, and look for it in the longer one. If there is no match, we repeat the process with several more samples of the same size. Normally such samples overlap each other. If that produces no fruit, we reduce the size of our sample, but take more of them. We continue this process, quickly reducing the size of the samples (but increasing the number of samples of a given size) until we find a significant match. Once we find a common anchor of substantial length, we expand it maximally and repeat the process with the pairs of texts on either side of the anchor. The staggered-sample algorithm is highly efficient, operating normally in logarithmic time (i.e., a tenfold increase in the length of the input only doubles the amount of processing time). The staggered-sample algorithm does not guarantee the LCS, but results are almost always optimal. See Kalvesmaki 2021. In the near future, I intend (1) to implement the algorithm in C# and (2) to present a paper describing a function that applies the algorithm to three or more versions of the same text (N-way or octopus diff).

Text Differencing Software

An entire Wikipedia article is devoted to comparison of 24 software packages that compare files: https://en.wikipedia.org/wiki/Comparison_of_file_comparison_tools

Of the packages mentioned in that list, I have personally used and benefitted from **diff**, **fc**, the **Notepad++** plugin, and **WinMerge**.

There are many other options not included in the Wikipedia's list worth mentioning (a name that is underlined represents package or function library in a programming language):

CollateX. Designed specifically for collating multiple versions of a text. Comparisons are word-based, not character-based. https://collatex.net/ A wrapper of the program is provided by **ITSEE Collation Editor:** https://github.com/itsee-birmingham/standalone_collation_editor

Delta XML features a number of products designed for comparing structured documents such as XML and JSON. https://www.deltaxml.com/

Diff-match-patch. Developed by Google, this algorithm is built upon the algorithm described in Myers 1986. The function library has been implemented in a variety of languages: C++, C#, Dart, Java, JavaScript, Lua, Objective C, Python. Source code: https://github.com/google/diff-match-patch

Juxta provides an environment to compare multiple versions of a text. https://www.juxtasoftware.org/

NDiff is Python's package for text comparison.

Oxygen XML Editor has a built in differencing feature for sets of two or three files, and can even be turned to comparison of sets of directories. https://oxygenxml.com/

OpenOffice and **Microsoft Word** have native file comparison tools. **Google documents** now also has this feature.

rdiff is a core component of the Unix utility rsync. It compares files (plain text, binary, etc.) and produces a delta that summarizes the difference between the two. https://rsync.samba.org/

Saktumiva was written primarily for Sanskrit texts, but was generalized to allow for the merging of two or more TEI XML files into a single one with apparatus. https://saktumiva.org/

TAN Diff+ is an XSLT application that will handle any number of input files, supports customized normalization, and returns comparative statistics in XML or HTML. It relies upon the staggered-sample algorithm (see above). A TAN Diff+ tutorial is under preparation. https://textalign.net

Some websites provide text difference APIs: **diffchecker.com**, **text-compare.com**, **textdiff.com**. You may find others.

Bibliography

Aldous, David, and Persi Diaconis. “Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson Theorem.” *Bulletin of the American Mathematical Society* 36, no. 04 (July 21, 1999): 413–33. https://doi.org/10.1090/S0273-0979-99-00796-X.

Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. “Basic Local Alignment Search Tool.” *Journal of Molecular Biology* 215, no. 3 (October 5, 1990): 403–10. https://doi.org/10.1016/S0022-2836(05)80360-2.

Birnbaum, David J. “Sequence Alignment in XSLT 3.0.” In *XML Prague 2020 – Conference Proceedings*, 45–65, 2020. https://archive.xmlprague.cz/2020/files/xmlprague-2020-proceedings.pdf.

Boes, Olivier. “Improving the Needleman-Wunsch Algorithm with the Dynamine Predictor.” University of Brussels, 2014.

Fraser, Neil. “Writing: Diff Strategies,” 2006. https://neil.fraser.name/writing/diff/.

Gotoh, O. “Optimal Alignment Between Groups of Sequences and Its Application to Multiple Sequence Alignment.” *Comput. Appl. Biosci.*, 1993. https://doi.org/10.1093/BIOINFORMATICS/9.3.361.

Gotoh, Osamu. “Alignment of Three Biological Sequences with an Efficient Traceback Procedure.” *Journal of Theoretical Biology* 121, no. 3 (August 7, 1986): 327–37. https://doi.org/10.1016/S0022-5193(86)80112-6.

Heckel, Paul. “A Technique for Isolating Differences between Files.” *Communications of the ACM* 21, no. 4 (April 1, 1978): 264–68. https://doi.org/10.1145/359460.359467.

Huang, Xiaoqiu, and Webb Miller. “A Time-Efficient, Linear-Space Local Similarity Algorithm.” *Advances in Applied Mathematics* 12, no. 3 (January 1, 1991): 337–57. https://doi.org/10.1016/0196-8858(91)90017-D.

Huitfeldt, Claus, and C. M. Sperberg-McQueen. “Document Similarity: Transcription, Edit Distances, Vocabulary Overlap, and the Metaphysics of Documents.” Washington, DC, 2020. https://doi.org/10.4242/BalisageVol25.Huitfeldt01.

Hunt, J W, and M D McIlroy. “An Algorithm for Differential File Comparison,” July 1976, 9.

Jhaver, Shagun, Latifur Khan, and Bhavani Thuraisingham. “Calculating Edit Distance for Large Sets of String Pairs Using MapReduce.” In *Proceedings of the ASE International Conference on Big Data*, 9, 2014.

Kalvesmaki, Joel. “String Comparison in XSLT with tan:diff(),” Vol. 26. Washington, DC, 2021. https://doi.org/10.4242/BalisageVol26.Kalvesmaki01.

Lipman, D J, S F Altschul, and J D Kececioglu. “A Tool for Multiple Sequence Alignment.” *Proceedings of the National Academy of Sciences of the United States of America* 86, no. 12 (June 1, 1989): 4412–15. https://doi.org/10.1073/pnas.86.12.4412.

Myers, Eugene W. “An O(ND) Difference Algorithm and Its Variations.” *Algorithmica* 1, no. 1–4 (November 1986): 251–66. https://doi.org/10.1007/BF01840446.

Myers, Eugene W., and Webb Miller. “Optimal Alignments in Linear Space.” *Bioinformatics* 4, no. 1 (March 1, 1988): 11–17. https://doi.org/10.1093/bioinformatics/4.1.11.

Ratcliff, John W., and David E. Metzener. “Pattern Matching: The Gestalt Approach.” Dr. Dobb’s, 1988. http://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970.

Sankoff, David, and Joseph B. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Reading, Mass.: Addison-Wesley PubCo, Advanced Book Program, 1983.

Stephen, Graham A. *String Searching Algorithms*. Lecture Notes Series on Computing 3. Singapore: World Scientific Publishing CoPteLtd, 1994. https://doi.org/10.1142/2418#t=toc.

Ukkonen, Esko. “Algorithms for Approximate String Matching.” *Information and Control*, International Conference on Foundations of Computation Theory, 64, no. 1 (January 1, 1985): 100–118. https://doi.org/10.1016/S0019-9958(85)80046-2.