string2string: A Modern Python Library for String-to-String Algorithms

Mirac Suzgun
Stanford University

Stuart M. Shieber
Harvard University

Dan Jurafsky
Stanford University

In memory of Lynn.

Abstract

We introduce **string2string**, an open-source library that offers a comprehensive suite of efficient algorithms for a broad range of string-to-string problems. It includes traditional algorithmic solutions as well as recent advanced neural approaches to tackle various problems in string alignment, distance measurement, lexical and semantic search, and similarity analysis—along with several helpful visualization tools and metrics to facilitate the interpretation and analysis of these methods. Notable algorithms featured in the library include the Smith-Waterman algorithm for pairwise local alignment, the Hirschberg algorithm for global alignment, the Wagner-Fischer algorithm for edit distance, BARTScore and BERTScore for similarity analysis, the Knuth-Morris-Pratt algorithm for lexical search, and Faiss for semantic search. In addition, it wraps existing efficient and widely-used implementations of certain frameworks and metrics, such as sacreBLEU and ROUGE. Overall, the library aims to provide extensive coverage and increased flexibility in comparison to existing libraries for strings. It can be used for many downstream applications, tasks, and problems in natural-language processing, bioinformatics, and computational social sciences. It is implemented in Python, easily installable via pip, and accessible through a simple API. Source code, documentation, and tutorials are all available on our GitHub page: https://github.com/stanfordnlp/string2string.[1]

1 Introduction

String-to-string problems have a wide range of applications in various domains and fields, such as natural-language processing (e.g., information extraction, spell checking, and semantic search), computational molecular biology (e.g., DNA sequence alignment), programming languages and compilers (e.g., parsing and compiling), as well as computational social sciences and digital humanities (e.g., lexical and semantic analysis of literary texts).

The current state of string-to-string processing, alignment, distance, similarity, and search algorithms is marked by a multitude of implementations in many programming languages, such as C++, Java, and Python, but these implementations are not unified and lack flexibility, modularity, and comprehensive documentation, hindering their accessibility to users. Thus, there is a need for a unified platform that combines these functionalities into one accessible and comprehensive system.

In this work, we present an open-source library that offers a broad collection of algorithms and techniques for the alignment, manipulation, and evaluation of string-to-string mappings.[2] These problems include measuring the lexical distance between two strings (e.g., under the Levenshtein edit distance metric), computing the local or global alignment between two DNA sequences (e.g., based on a substitution matrix such as BLOSUM), calculating the semantic similarity between two texts (e.g., using BART-embeddings), and performing efficient semantic search (e.g., via the Faiss library by FAIR (Johnson et al., 2019)).

The string2string library has been purposefully crafted to prioritize key design principles, including modularity, completeness, efficiency, flexibility, and clarity. As an open-source initiative, the library will continue to grow and adapt to meet the evolving of its user community in the future, and we are committed to ensuring that the library remains a flexible, accessible, and dynamic resource, capable of accommodating the changing landscape of string-to-string problems and tasks.

[1]Correspondence to: msuzgun@cs.stanford.edu.

[2]We define a *string* as an ordered collection of characters—such as letters, numerals, symbols—which serves as a representation of a unit of information, text, or data. Strings can be used to represent anything, from simple sentences to complex nucleic acid sequences or elaborate computer programs.

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 278–285

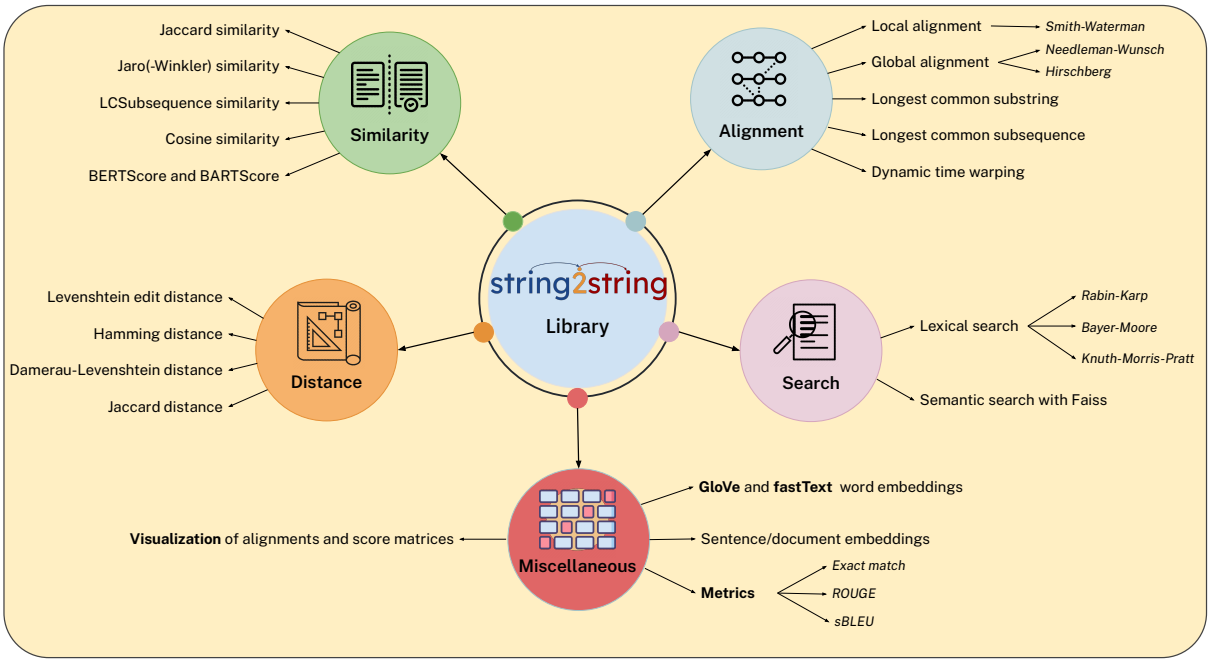

Figure 1: The `string2string` library provides a broad set of algorithms and techniques to tackle a variety of problems and tasks involving the pairwise alignment, comparison, evaluation, manipulation, and processing of string-to-string mappings. It includes the implementation of widely-used algorithms, such as Smith-Waterman (Smith and Waterman, 1981) for local alignment, Knuth-Morris-Pratt (Knuth et al., 1977) for identical string matching (search), and Wagner-Fisher (Wagner and Fischer, 1974) for edit distance, as well as recent neural approaches, such as BERTScore (Zhang et al., 2020) and BARTScore (Yuan et al., 2021) for semantic similarity measurements and Faiss (Johnson et al., 2019) for semantic search. The library has been designed to support not only individual strings but also lists of strings so that users can align and compare strings at the token, word, or sentence levels. It further contains visualization features to allow users to visualize alignments and score matrices of strings.

2 Related Work

The fields of natural-language processing and machine learning have a long-standing and exemplary tradition of fostering a culture that values open-source tools and libraries. While designing our own string-to-string library, which includes both traditional algorithmic and neural approaches to various problems, we have drawn inspirations from Natural Language Toolkit (NLTK; Bird and Loper (2004)), Gensim (Řehůřek and Sojka, 2010), OpenGrm Ngram (Roark et al., 2012), Stanford CoreNLP Toolkit (Manning et al., 2014), OpenNMT (Klein et al., 2017), tensor2tensor (Vaswani et al., 2018), AllenNLP (Gardner et al., 2018), fairseq (Ott et al., 2019), spaCy (Neumann et al., 2019), Stanza (Qi et al., 2020), Transformers (Wolf et al., 2020), and Torch-Struct (Rush, 2020), among many others.

3 Overview of Algorithms

The `string2string` library offers a rich collection of algorithmic solutions to tackle a wide range of string-to-string problems and tasks. We have clustered these algorithms into four categories: pairwise alignment, distance measurement, similarity analysis, and search.[3] Each category contains a suite of efficient algorithms that are tailored to address specific problems within their respective domain. In what follows, we provide a brief overview of these algorithms, along with the associated problems or tasks they are designed to solve.

3.1 Pairwise Alignment

Pairwise string alignment is the problem of identifying an optimal alignment between two strings, such as nucleotide sequences in DNA or paragraphs in a text. This task involves aligning them in a way that maximizes the number of matching symbols while allowing for gaps or mismatches where necessary. Pairwise string alignment is a widely-used technique that plays a crucial role in tasks such as DNA sequence alignment, database searching, and phylogenetic analysis.

As exhibited in Table 1, the library, in its current state, provides efficient solutions to local alignment, global alignment, longest common substring

[3] By duality, distance measurement methods can naturally be used for string similarity analysis, and vice versa.

Pairwise Alignment
Local alignment : The best possible matching substring or subsequence alignment between two strings—based on a substitution matrix and gap penalty function—allowing for gaps and mismatches within a specified region of the input sequences. ⋆ (a) Dynamic programming solution (Smith and Waterman, 1981): $\mathcal{O}(nm)$ in terms of space and time.
Global alignment : The best possible alignment between two strings over their entire length. ⋆ (a) Dynamic programming solution (Needleman and Wunsch, 1970): $\mathcal{O}(nm)$ in terms of space and time. ⋆ (b) Divide-and-conquer + dynamic programming solution (Hirschberg, 1975): $\mathcal{O}(m)$ in terms of space and $\mathcal{O}(nm)$ time.
Longest common substring : The longest *contiguous* substring that appears in both strings. ⋆ (a) Dynamic programming solution: $\mathcal{O}(nm)$ in terms of space and time.
Longest common subsequence : The longest possible sequence of symbols that appears in the same order in both strings. ⋆ (a) Dynamic programming solution: $\mathcal{O}(nm)$ in terms of space and time.
Dynamic time warping (DTW) : The optimal warp path that minimizes the distance between sequences of varying length. ⋆ (a) Dynamic programming solution (Sakoe and Chiba, 1978): $\mathcal{O}(nm)$ in terms of space and time. ⋆ (b) Space-time improved version of (a) via Hirschberg (1975)'s algorithm: Reduces space complexity to $\mathcal{O}(m)$.
Distance
Levenshtein edit distance : The minimum number of insertions, deletions, and substitutions needed to convert S into T. ⋆ (a) Dynamic programming solution (Wagner and Fischer, 1974): $\mathcal{O}(nm)$ in terms of space and time. ⋆ (b) Space-improved version of (a): Reduces space complexity to $\mathcal{O}(m)$ by storing only two rows.
Hamming distance : The total number of indices at which strings, S and T, of equal length differ. ⋆ (a) Naive solution: $\mathcal{O}(n)$ in terms of space and time.
Damerau–Levenshtein distance : The minimum number of insertions, deletions, substitutions, and adjacent transpositions needed to convert S into T. ⋆ (a) Dynamic programming solution (simple extension of the Wagner-Fisher algorithm): $\mathcal{O}(nm)$ in terms of space and time. ⋆ (b) Space-improved version of (a): Reduces space complexity to $\mathcal{O}(m)$ by storing only two rows.
Jaccard distance : The inverse of Jaccard similarity (that is, 1.0 - Jaccard similarity coefficient). ⋆ (a) Naive solution: $\mathcal{O}(n)$ in terms of space and time.

Table 1: Overview of the pairwise string alignment and distance problems addressed by the library, along with the algorithmic approaches employed to solve them. In all instances, we assume that we are given two strings, S and T, over a finite alphabet Σ, where where $n = |S|$, $m = |T|$, and $k = |\Sigma|$, with $m \leq n$. Also, whenever possible, we include the brute-force and memoized solutions to these problems (as in the case of edit distance, for instance).

(LCSubstring), longest common subsequence (LCSubsequence), and dynamic time warping (DTW) problems. It is worth noting that all of the problems and tasks covered in this suite can be solved using standard dynamic programming-based solutions. Alternative approaches to long sequence or string alignment problems, such as FASTA (Lipman and Pearson, 1985) and BLAST (Altschul et al., 1990), also exist; they offer improved efficiency through the use of probabilistic or heuristic methods, but they do not always guarantee optimal solutions and may sacrifice accuracy for speed. Due to their ability to handle large datasets quickly and provide reasonably accurate results, BLAST and FASTA are still widely used in bioinformatics, and for that reason, we plan on including them in the `string2string` library in the future.

3.2 Distance

String distance refers to the problem of quantifying the extent to which two given strings are dissimilar based on a distance function. The Levenshtein edit distance metric, for instance, corresponds to the minimum number of insertion, deletion, or substitution operations required to transform one string into another. It has a famous dynamic programming solution, which is often referred to as the Wagner-Fischer algorithm (Wagner and Fischer, 1974). In this library, we provide an implementation of the Wagner-Fischer algorithm, which has a quadratic time and space complexity, as well as an improved version of it, which reduces the overall space complexity to linear.[4] We further cover and provide efficient solutions to the Hamming dis-

[4]Incidentally, we highlight an important discovery by Backurs and Indyk (2015) that the edit distance between two strings cannot be computed in strongly subquadratic time, unless the strong exponential time hypothesis is false.

Similarity
Jaccard similarity : The size of the set of unique symbols that appear in both strings (i.e., in the intersection) divided by the size of the set of the union of the symbols in both strings.
⋆ (a) Naive solution: $\mathcal{O}(n)$ in terms of space and time.
Jaro(-Winkler) similarity : A measure of similarity based on matching symbols and transpositions in two strings.
⋆ (a) Naive solution: $\mathcal{O}(nm)$ in terms of time and $\mathcal{O}(n)$ in terms of space.
LCSubsequence similarity : A degree of similarity between two strings based on the length of their longest common subsequence.
⋆ (a) Based on the efficient solution to the longest common subsequence problem.♣
Cosine similarity : The similarity between two strings based on the angle between their corresponding vector representations.
⋆ (a) Utilizes `numpy` and `torch` functions: $\mathcal{O}(E)$ in terms of time and $\mathcal{O}(E)$ in terms of space.♢
BERTScore (Zhang et al., 2020): A measure of semantic similarity that employs contextualized embeddings derived from the pre-trained BERT model (Devlin et al., 2019) to estimate the semantic closeness between two pieces of text.
⋆ (a) Adaptation of the original BERTScore implementation: $\mathcal{O}(nm)$ in terms of time and $\mathcal{O}(nm \cdot E)$ in terms of space.
BARTScore (Yuan et al., 2021): A measure of semantic similarity that utilizes the pre-trained BART model (Lewis et al., 2020) and that achieves high correlation with human judgements.
⋆ (a) Adaptation of the original BARTScore implementation: $\mathcal{O}(nm)$ in terms of time and $\mathcal{O}(nm \cdot E)$ in terms of space.
Search
Lexical search :
⋆ (a) Naive (brute-force) search: $\mathcal{O}(mn)$ in terms of match time and $\mathcal{O}(1)$ in terms of space.
⋆ (b) Rabin-Karp algorithm (Karp and Rabin, 1987): $\mathcal{O}(mn)$ in terms of match time and $\mathcal{O}(1)$ in terms of space.
⋆ (c) Boyer-Moore algorithm (Boyer and Moore, 1977): $\mathcal{O}(mn)$ in terms of match time and $\mathcal{O}(\lvert\Sigma\rvert)$ in terms of space..
⋆ (d) Knuth-Morris-Pratt algorithm (Knuth et al., 1977): $\mathcal{O}(n)$ in terms of match time and $\mathcal{O}(m)$ in terms of space.
Semantic search :
⋆ (a) FAISS (Johnson et al., 2019): $\mathcal{O}(\log^2 n)$ in terms of match time and $\mathcal{O}(n \cdot E)$ in terms of space.♠

Table 2: Overview of the string similarity and search solutions used in the library. As in Table 1, we assume that we are provided with two strings, S and T, over a finite alphabet Σ, where $n = |S|$, $m = |T|$, and $k = |\Sigma|$, with $m \leq n$. Furthermore, we use E to denote the size of the embedding space (or token), whenever applicable. Both the Rabin-Karp and Knuth-Morris-Pratt algorithms require $\Theta(m)$ time for pre-processing and $\Theta(n)$ time for searching, whereas the Boyer-Moore algorithm has a pre-processing time complexity of $\Theta(m+k)$. In terms of space, the Rabin-Karp, Boyer-Moore, and Knuth-Morris-Pratt algorithms require $\Theta(1)$, $\Theta(k)$. and $\Theta(m)$, respectively. Footnote ♣: Please refer to Eqn. (11) in (Suzgun et al., 2022a) for a mathematical formulation of LCSubsequence similarity. Note that the authors call this similarity measure "`Sim-LCS`." Footnote ♢: We assume that the dimension (size) of the two vectors are both E. Footnote ♠: We invite our readers to look at the blogpost by Feinberg (2019) for a detailed analysis of Facebook AI Research's Faiss algorithm.

tance, Damerau-Levenshtein distance, and Jaccard distance problems, as shown in Table 1.

One noteworthy feature of the library is that it allows the user to specify the weight of string operations (insertions, deletions, substitutions, and transpositions) depending on the distance function of choice. Furthermore, it can compute the distance between not only string pairs but also pairs of lists of strings, thereby not limiting the users to make comparisons only at the character or symbol level.

3.3 Similarity

String similarity refers to the problem of measuring the degree to which two given strings are similar to each other based on a similarity function—which can be defined on various criteria, such as character matching, longest common substring or subsequence comparison, or structural alignment. There is a natural duality between string similarity measures and string distance measures, which means that it is possible to convert one into the other with ease; hence, it is often the case that one uses string similarity and distance measures interchangeably.

Jaccard similarity, Jaro similarity, Jaro-Winkler similarity, LCSubsequence similarity, cosine similarity, BERTScore, and BARTScore are among the similarity measures that are covered in the library. The first four can be seen as lexical similarity measures, as they assess surface or structural closeness, whereas the remaining three can be regarded as semantic similarity measures, as they take the implied meaning of the constituents of the given strings into account.

The present library provides users with the ability to calculate cosine similarity not only between individual words—via pre-trained GloVe (Pennington et al., 2014) or fastText (Joulin et al., 2016) word embeddings—but also for longer pieces of text such as sentences, paragraphs, or even documents—via averaged or last-token embed-

dings obtained from a neural language model such as BERT (Devlin et al., 2019). As we also mention in Section 4, this feature enables users to compare the semantic similarity of longer segments of text in only a few lines of code, providing greater flexibility in text analysis tasks.

3.4 Search

String search, also known as string matching, refers to the problem of determining whether a given pattern string exists inside a longer string. The brute-force approach to string search would involve examining each position of the longer string to determine if it matches the pattern string; however, this method can be inefficient, particularly when dealing with large strings. In the library, we therefore include the Rabin-Karp (Karp and Rabin, 1987), Boyer-Moore (Boyer and Moore, 1977), and Knuth-Morris-Pratt (Knuth et al., 1977) algorithms for identical string matching as well.

The library additionally provides support for semantic search via Facebook AI Research's Faiss library (Johnson et al., 2019), which, in essence, allows efficient similarity search and clustering of dense vectors. In contrast to the previous setup for identical string matching, Faiss initially requires the user to provide a *list* of strings (texts) as a corpus and creates a fixed-vector representation of each string using a neural language model.[5] Once the initialization of the corpus is done, one can perform "queries" on the corpus. Given a new query, one can automatically get the embedding of that query, map it onto the embedding space of the corpus, and return the nearest neighbours of the query on the embedding space, thereby finding the texts that are semantically closest to the query.

As one might imagine, string search algorithms are highly practical tools that have a wide range of applications across different fields. These algorithms allow users to locate and retrieve specific patterns within a long text or a large corpus. For instance, the string search algorithms covered in the `string2string` library—as shown in Table 2—can be used for pattern recognition, DNA matching, plagiarism detection, and data mining, among many other downstream applications and tasks.

[5]The user is provided with the flexibility to determine how to obtain a fixed embedding for each text. Specifically, the user has the option to choose between different embedding methods, such as averaging the token embeddings or selecting the embedding of the final token in the sequence.

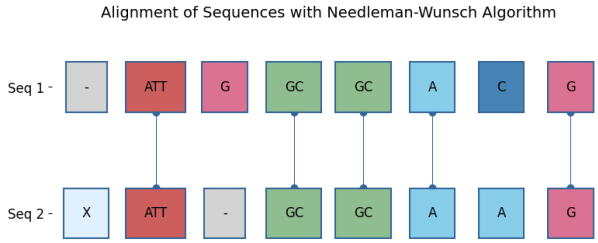

Figure 2: Alignment of two sequences of strings, `[ATT G GC GC A C G]` and `[X ATT GC GC A A G]`, as obtained by the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970). Our library allows users to visualize the pairwise alignment between strings (or lists of strings).

4 Additional Features

One noteworthy feature of the library is its ability to simplify the use of the GloVe (Pennington et al., 2014) or fastText (Joulin et al., 2016) word embeddings by enabling users to download and use them with just one line of code. This streamlined process not only saves users time and effort but also eliminates the need for additional installations or complex configurations. By providing this feature, we have sought to make the library more accessible to users and encourage the use of pre-trained word embeddings in various string-to-string tasks and applications, such as measuring the cosine similarity between two words.[6]

Similarly, users can seamlessly get the averaged or last token embeddings of a piece of text from a pre-trained language model that is hosted on Hugging Face Models (Wolf et al., 2020) or on their local path in a few lines of code, and we provide both CPU and GPU support for these computations.

Finally, we note that the library offers various visualization capabilities that allow users to visually inspect the alignment between two strings or the score matrix of the distance or similarity between them. This functionality facilitates the understanding and interpretation of the output of various algorithms and can aid in the selection of the most suitable algorithm for a given task. By incorporating this feature into the library, we aim to enhance the user experience and provide intuitive means of interpreting the outputs.[7] Figure 2 shows a simple alignment between two lists of strings, as generated by our library.

[6]Notably, fastText offers pre-trained word embeddings for 157 languages—trained on Common Crawl and Wikipedia—that one can easily download and use with our library.

[7]For instance, our library provides a practical, hands-on tutorial focused on the HUPD (Suzgun et al., 2022b), a large patent corpus. This tutorial showcases the efficient use of our library's functionalities and features for performing semantic search and visualizing the textual content of patent documents.

5 Library Design Principles

We have endeavoured to build a comprehensive and easy-to-use platform for numerous string-to-string processing, comparison, manipulation, and search algorithms. We have purposefully structured and organized the library to allow easy customization, functional extension, and modular integration.

Completeness. The library offers a comprehensive set of classical algorithms as well as neural approaches to tackle a wide range of string-to-string problems. We have intentionally included both efficient and simpler solutions, such as brute-force and memoization-based approaches, where appropriate. By providing multiple solutions to the same problem, the library allows users to compare and contrast the performance of different algorithmic methods. This approach enhances users' understanding of the trade-offs between different algorithmic solutions and helps them appreciate the strengths and limitations of each approach.

Modularity. To improve the efficiency and maintainability of the codebase, we have adopted a modular design approach that breaks down the code into smaller and self-contained modules. This approach simplifies the process of adding new features and functionalities and modifying existing ones for developers, while also enabling efficient testing, debugging, and overall maintenance of the library. The modular design has allowed us to quickly locate and fix any errors, without disrupting the entire codebase, during development. Moreover, this modular approach ensures that the library is scalable and adaptable to future updates and changes, which should enable us to easily improve the library's functionality and expand its use in various tasks and applications moving forward.

Efficiency. We have taken great care to ensure that the algorithm implementations are efficient both computationally and memory-wise so that they could easily handle large datasets and complex tasks. We provide basic support for process-based parallelism via Python's inherent `multiprocessing` package, as well as `joblib`. Additionally, we provide GPU support for neural-based approaches, whenever applicable. While we strive to balance efficiency and clarity, we acknowledge that in some cases, trade-offs may exist between the two. In such cases, we have placed greater emphasis on clarity, ensuring that the algorithms are transparent and easy to understand, even at the cost of some efficiency. Nonetheless, we believe that the library's overall efficiency, combined with its transparency and comprehensibility, makes it a valuable resource for the community.

Support for List of Strings. The library has been designed to support not only individual strings but also lists of strings—whenever possible, enabling users to align or compare strings at the subword or token level. This feature provides greater flexibility in the library's use cases, as it allows users to analyze and compare more complex data structures beyond only individual strings. By supporting lists of strings, the library can handle a wider range of textual input types and structures.

Strong Typing. The use of strong typing requirements is an essential aspect of the library, as it ensures that the inputs are always consistent and accurate, which is crucial for generating reliable results. By carefully annotating all the arguments of the algorithms used in the library, we have sought to increase the robustness and reliability of the codebase. This approach has helped prevent input-related errors, such as incorrect data type or format, from occurring during execution.

Accessibility. The library has been implemented in Python, a programming language which has been the core of many natural-language processing tools and applications in academia and industry. The `string2string` library is "pip"-installable and can be integrated into common machine learning and natural-language processing frameworks such as PyTorch, TensorFlow, and scikit-learn.

Open-Source Effort. The library is—and will remain—free and accessible to all users. We hope that this approach will promote community-driven development and encourage collaboration among researchers and developers, enabling them to contribute to and improve the library.

6 Conclusion

We introduced `string2string`, an open-source library that offers a large collection of algorithms for a broad range of string-to-string problems. The library is implemented in Python, hosted on GitHub, and installable via pip. It contains extensive documentation along with several hands-on tutorials to aid users to explore and utilize the library effectively. With the help of the open-source community, we hope to grow and improve the library. We encourage users to feel free to provide us with feedback, report any issues, and propose new features to expand the functionality and scope of the library.

Acknowledgements

We would like to express our deepest appreciation to Corinna Coupette for providing us with the inspiration to create this open-source library. Furthermore, we would like to thank Sarah DeMott, Sebastian Gehrmann, Elena Glassman, Tayfun Gür, Daniel E. Ho, Şule Kahraman, Deniz Keleş, Scott D. Kominers, Christopher Manning, Luke Melas-Kyriazi, Tolúlọpẹ́ Ògúnrẹ̀mí, Alexander "Sasha" Rush, George Saussy, Kutay Serova, Holger Spamann, Kyle Swanson, and Garrett Tanzer for their valuable comments, useful suggestions, and support. We owe a special debt of gratitude to Federico Bianchi for inspecting our code, documentation, and tutorials many times and providing us with such helpful and constructive feedback.

References

SF Altschul, W Gish, W Miller, EW Myers, and DJ Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Arturs Backurs and Piotr Indyk. 2015. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 51–58, New York, NY, USA. Association for Computing Machinery.

Steven Bird and Edward Loper. 2004. NLTK: The natural language toolkit. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 214–217, Barcelona, Spain. Association for Computational Linguistics.

Robert S. Boyer and J. Strother Moore. 1977. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Vlad Feinberg. 2019. Facebook AI similarity search (FAISS), part II. Accessed on March 13, 2023.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. AllenNLP: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 1–6, Melbourne, Australia. Association for Computational Linguistics.

Daniel S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. FastText.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*.

Richard M Karp and Michael O Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880.

David J Lipman and William R Pearson. 1985. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland. Association for Computational Linguistics.

Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453.

Mark Neumann, Daniel King, Iz Beltagy, and Waleed Ammar. 2019. ScispaCy: Fast and robust models for biomedical natural language processing. In *Proceedings of the 18th BioNLP Workshop and Shared Task*, pages 319–327, Florence, Italy. Association for Computational Linguistics.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for

sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 101–108, Online. Association for Computational Linguistics.

Radim Řehůřek and Petr Sojka. 2010. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA. http://is.muni.cz/publication/884893/en.

Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66, Jeju Island, Korea. Association for Computational Linguistics.

Alexander Rush. 2020. Torch-Struct: Deep structured prediction library. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 335–342, Online. Association for Computational Linguistics.

Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49.

T. F. Smith and Michael S. Waterman. 1981. Identification of common molecular subsequences. *Journal of molecular biology*, 147 1:195–7.

Mirac Suzgun, Luke Melas-Kyriazi, and Dan Jurafsky. 2022a. Follow the wisdom of the crowd: Effective text generation via minimum Bayes risk decoding. *arXiv preprint arXiv:2211.07634*.

Mirac Suzgun, Luke Melas-Kyriazi, Suproteem K Sarkar, Scott Duke Kominers, and Stuart M Shieber. 2022b. The Harvard USPTO Patent Dataset: A Large-Scale, Well-Structured, and Multi-Purpose Corpus of Patent Applications. *arXiv preprint arXiv:2207.04043*.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416.

Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Weizhe Yuan, Graham Neubig, and Pengfei Liu. 2021. BARTScore: Evaluating generated text as text generation. In *Advances in Neural Information Processing Systems*, volume 34, pages 27263–27277. Curran Associates, Inc.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating text generation with BERT. In *International Conference on Learning Representations*.