Stringalign: Moving beyond summary statistics with a transparent Unicode-aware tool for evaluating automatic transcription models

Yngve Mardal Moe*† Marie Roald*‡

June 14, 2026

Abstract

Comparing text strings is crucial when evaluating and understanding the performance of various text processing tasks such as document recognition and audio transcription. With an increasingly complex landscape of AI-based handwritten text recognition (HTR), optical character recognition (OCR) and automatic speech recognition (ASR) models, there is a need for tools that facilitate evaluation in a flexible and reproducible way. This paper presents Stringalign, a Python library designed to simplify the evaluation process for automatic transcription projects and facilitate transparent evaluation. Stringalign's tools to examine and visualise both the rate of errors and the types of errors a model makes, give insights into possible improvements and help inform model selection for a particular task. Widely used string comparison metrics, such as the character and word error rates (CER and WER), although useful, can be ambiguous due to varying definitions of what constitutes a character and a word. Stringalign addresses this challenge by ensuring all preprocessing (i.e. normalisation and tokenisation) is transparent and easily replicable, and by providing tools to move beyond summary statistics and analyse common model errors. Moreover, Stringalign adheres to FAIR (Findable, Accessible, Interoperable, and Reusable) principles for research software while staying lightweight and easy to adapt into researchers existing workflows. In this paper, we discuss challenges with character and word level string comparisons and show through examples that where existing tools can yield opaque and sometimes confusing results, Stringalign provides an easy-to-use and unambiguous alternative.

1 Introduction

Machine-learning-based approaches have become increasingly important for document recognition [4]. Therefore, all aspects of document recognition require quality evaluation of model performance. End users need evaluation methods to make an informed choice on which model is right for their use case and researchers need evaluation methods to guide their work in developing new models. Even the data annotation phase of a project may want to evaluate and compare annotations to ensure quality data is produced [24]. Tools that enable such evaluation are therefore essential for the entire document recognition stack.

Similarly, the emergence of deep learning has cemented the importance of data when creating transcription models. Previous work has shown that increasing the size and quality of data is an

*The authors contributed equally
†Independent researcher
‡The National Library of Norway

effective strategy for improving model performance compared to algorithmic optimisations [2, 24]. Still, collecting data can be expensive and time-consuming. Particularly for domains where data is scarce or when the annotation requires domain experts who have little time to devote to annotation, e.g. for low-resource languages [3, 22]. For such cases, precise knowledge about what a model struggles with can give insights into what data is lacking so that the data gathering can be planned strategically.

A critical challenge with evaluating automatic text recognition (ATR) and automatic speech recognition (ASR) models is the ambiguities inherent for comparing strings [11]. Normalisation and tokenisation are applied before comparing strings and there are multiple strategies to choose from [7, 15, 17, 19]. These decisions affect metrics such as the character error rate (CER) and the word error rate (WER), yet they are not always reported together with the metrics, making the evaluations hard compare across projects [11]. Thus, there is a need for open-source and easy-to-use tools that promotes reproducible evaluation practices.

In later years, the value of research software has been increasingly highlighted. In particular, through Findable Accessible Interoperable and Reusable for Research Software (FAIR4RS) principles which extend open science practices to cover research software [1]. Similarly to the FAIR principles for research data [23], the intent of the FAIR4RS principles is to make sure research software is discoverable, easy to access, compatible with other projects and organised in a way that is easy to reuse. Software aiming to facilitate reproducible and transparent transcription evaluation for research should adhere to these principles.

This work contributes an updated review of the existing transcription evaluation landscape expanding on [11] and introduces Stringalign which provides:

- Well documented and extensively tested code for transcription evaluation that is both in line with FAIR4RS and easy to adapt into existing workflows.
- Transparent and Unicode-aware tokenisation.
- Tools to explore and visualise model errors.

The paper is structured such: Section 2 discusses challenges of string comparisons, gives an overview of transcription evaluation tools and motivations for a new tool. Section 3 covers Stringalign's design principles and implementation details. Section 4 contains examples comparing existing tooling and demonstrating Stringalign's features. Finally, Section 5 contains conclusions and future work.

2 Background

2.1 Alignment

A common approach for comparing strings is through *optimal string alignment*, where two strings, which we denote as the *reference string* and the *predicted string*, are aligned to minimise the number of *edits* required to turn the reference string into the predicted string. Consider the two strings

```
Reference: string
Predicted: align
```

Using the edit operations `Insert`, `Delete` and `Replace`, we can transform the reference into the predicted string: 1. `Delete(s)`, 2. `Replace(t, a)`, 3. `Replace(r, l)`, 4. `Keep(i)`, 5. `Delete(n)`, 6. `Keep(g)`, 7. `Insert(n)`; or, visualised:

```
Reference: string-
Predicted: -ali-gn
```

Fortunately, previous work has defined several efficient algorithms to find such optimal alignments [8, 9, 16].

2.2 Comparison metrics

2.2.1 Levenshtein distance

One metric for comparing two strings is the *Levenshtein distance* [5, 6] (as referenced in [8]), also called the *edit distance*[1]. The Levenshtein distance, L, between the reference and predicted strings, r and p, is

$$L(r,p) = N_I(r,p) + N_D(r,p) + N_R(r,p), \tag{1}$$

where $N_I(r,p)$, $N_D(r,p)$ and $N_R(r,p)$ is the number of insertions, deletions and replacements in the optimal alignment between r and p. Thus, $L(r,p)$ is low for similar strings. If we consider the example above, we see that the Levenshtein distance between `string` and `align` is equal to 5.

2.2.2 Character Error Rate

The CER is a common metric for comparing strings, and is given by the number of inserted, deleted and replaced characters divided by the number of characters in the reference string. Specifically, the CER between the reference string, r, and predicted string, p, is given by

$$\mathrm{CER}(r,p) = \frac{N_I^{(\mathrm{c})}(r,p) + N_D^{(\mathrm{c})}(r,p) + N_R^{(\mathrm{c})}(r,p)}{N_D^{(\mathrm{c})}(r,p) + N_R^{(\mathrm{c})}(r,p) + N_K^{(\mathrm{c})}(r,p)}, \tag{2}$$

where $N_I^{(\mathrm{c})}(r,p)$, $N_D^{(\mathrm{c})}(r,p)$, $N_R^{(\mathrm{c})}(r,p)$ and $N_K^{(\mathrm{c})}(r,p)$ is the number of inserted, deleted, replaced and kept characters in the optimal character-alignment for r and p. If we again consider the example above, we see that the CER is $\frac{5}{6} = 0.83$.

2.2.3 Word Error Rate

Another common metric for comparing strings is the WER, which is computed from an optimal alignment of *words*:

$$\mathrm{WER}(r,p) = \frac{N_I^{(\mathrm{w})}(r,p) + N_D^{(\mathrm{w})}(r,p) + N_R^{(\mathrm{w})}(r,p)}{N_D^{(\mathrm{w})}(r,p) + N_R^{(\mathrm{w})}(r,p) + N_K^{(\mathrm{w})}(r,p)}, \tag{3}$$

where $N_I^{(\mathrm{w})}(r,p)$, $N_D^{(\mathrm{w})}(r,p)$, $N_R^{(\mathrm{w})}(r,p)$ and $N_K^{(\mathrm{w})}(r,p)$ are the number of insertions, deletions, replacements and keeps in an optimal word-alignment for r and p. Thus, the WER measures the amount of word errors in the predicted string compared to the reference. With the example above, we see that the WER is 1 as both strings contain just one word, which differs.

[1] More precisely, the Levenshtein distance is the edit distance when the edit operations are insertions, deletions and replacements.

Table 1: Token-specific metrics

Metric	Definition	Synonyms
True positive rate	$\mathrm{TPR}(t) = \frac{\mathtt{TP}(t)}{\mathtt{TP}(t)+\mathtt{FN}(t)}$	Sensitivity, Recall
Positive Predictive Value	$\mathrm{PPV}(t) = \frac{\mathtt{TP}(t)}{\mathtt{TP}(t)+\mathtt{FP}(t)}$	Precision
False Discovery Rate	$\mathrm{FDR}(t) = \frac{\mathtt{FP}(t)}{\mathtt{TP}(t)+\mathtt{FP}(t)}$	
f_1-score	$f_1(t) = 2\frac{\mathrm{TPR}(t)\mathrm{PPV}(t)}{\mathrm{TPR}(t)+\mathrm{PPV}(t)}$	Dice score

2.2.4 Token error rate

From Eqs. (2) and (3), we see that the WER and CER are similar and the only difference is whether they are computed based on characters or words. A natural generalisation of these concepts is the *token error rate* (TER). To compute the TER, we split the reference and predicted strings into *tokens* (e.g. characters or words) before alignment. Thus, the TER is defined by

$$\mathrm{TER}(r,p) = \frac{N_I^{(\mathrm{t})}(r,p) + N_D^{(\mathrm{t})}(r,p) + N_R^{(\mathrm{t})}(r,p)}{N_D^{(\mathrm{t})}(r,p) + N_R^{(\mathrm{t})}(r,p) + N_K^{(\mathrm{t})}(r,p)}, \tag{4}$$

where $N_I^{(\mathrm{t})}(r,p)$, $N_D^{(\mathrm{t})}(r,p)$, $N_R^{(\mathrm{t})}(r,p)$ and $N_k^{(\mathrm{t})}(r,p)$ are the number of inserted, deleted, replaced and kept tokens in an optimal alignment between the tokens of r and p. A benefit of the token concept is that characters and words are (as we will discuss in Section 2.4) ambiguously defined. Moreover, by considering tokens, we can define metrics that are valid for both words and characters.

2.2.5 Token-specific metrics

For some applications it can be useful to look at performance metrics for specific tokens [3, 14]. Based on the optimal alignment, we can, e.g. compute the true positive count (`TP`), false positive count (`FP`) and false negative count (`FN`) for each token in the predicted and reference strings. In particular, we count the number of times it is kept (`TP`), the number of times it is inserted or replaces another token (`FP`) and the number of times it is deleted or replaced with another token (`FN`). From these numbers, we can compute a variety of token-specific statistics, which are defined in Table 1.

2.3 Alignment uniqueness

The optimal alignment is not *unique* as two strings may have more than one alignment with the same number of edit operations. Consider the earlier example:

```
Reference: string-
Predicted: -ali-gn
```

Here, the Levenshtein distance is five, but multiple alignments have five edits. In particular, the first deletion has three possible locations (delete `s`, `t`, or `r`), the transposition of `n` and `g` can be resolved in three ways (aligning the `n`s, `g`s or neither), and the first deletion can be omitted, aligning `stri` with `alig`, keeping the `n` and deleting the `g`. In total, we have ten different optimal alignments where the TP, FP and FN for each token can vary, while the TER and Levenshtein distance are constant. Moreover, the number of possible optimal alignments can grow quickly with string length. Still, character confusions and token-specific statistics can

provide useful information on what tokens a transcription model struggles with, and the non-uniqueness challenge can be alleviated by averaging the `TP`, `FP` and `FN` for several different optimal alignments.

Another way to mitigate this non-uniqueness challenge is merging subsequent edits. Then, the above example would get the multi-token edit `str -> al`, reducing the number of possible alignments to four. This strategy makes counting common edit operations more stable, but makes `TP`, `FP` or `FN` ill-defined.

2.4 Ambiguities in string normalisation and segmentation

So far, we have glossed over some questions critical for comparing strings in practice: What constitutes a "character"? And when are two characters "different"? This section discusses how characters are represented in a computer and how ambiguities in the representation can lead to ambiguities in evaluation metrics.

2.4.1 Unicode

Computers represent characters through enumeration. Typically, `a` is represented by the number 97, or `0x61` in hexadecimal, `b` is 98, or `0x62`, etc. The mapping from numbers (or *code points*) to characters is a *string encoding*, and the most common string encoding is Unicode [15, 20]. Unicode code points are often represented as hexadecimal numbers prepended with `U+`, so `a` is `U+0061`.

2.4.2 Normalisation

Some characters have more than one Unicode representation [17]. Such ambiguities can make it unclear how to count or compare strings. Therefore, Unicode defines two equivalence classes for characters: *canonical equivalence* and *compatibility equivalence* [17, 20]. Canonically equivalent characters are code points, or code point sequences, that represent the same abstract character and should always look the same. Meanwhile, compatibility equivalent characters do not need to look the same and can be both visually and semantically different.

Unicode also defines composed and decomposed forms for each equivalence class. The decomposed form expands characters into as many constituent code points as possible and sorts them, while the composed form then iteratively merges compatible code points to get a compact representation. In total, we get the four Unicode normalisation forms: canonical decomposed (NFD), canonical composed (NFC), compatibility decomposed (NFKD) and compatibility composed (NFKC). These normalisation forms are illustrated in Table 2. Since NFC-normalised text is as compact as possible while not modifying the visual appearance, it is a normalisation likely to fit most research applications.

2.4.3 Confusable characters

Unicode also contains *confusable characters* or *confusables* (also called homoglyphs or homographs) [7, 18]. Such characters look the same or similar but are different and cannot be normalised to the same code point. Addressing this, Unicode provides two lists: `confusables.txt`, containing visual confusables useful for detecting security problems and `intentional.txt`, containing characters often identical in typefaces with a harmonised design [18]. During evaluation, you should consider how you expect models to handle confusables as this can depend on the type of model and application.

Table 2: Various characters and their different normalisation forms

Character	NFD	NFC	NFKD	NFKC
Å	A ◦	Å	A ◦	Å
U+00C5	U+0041 U+030A	U+00C5	U+0041 U+030A	U+00C5
𝔗	𝔗	𝔗	T	T
U+1D517	U+1D517	U+1D517	U+0054	U+0054
ſ̣	ſ̣	ſ̣	s̩	s̩
U+017F U+0323	U+017F U+0323	U+017F U+0323	U+0073 U+0323	U+1E63

2.4.4 Character segmentation

A simple strategy for character segmentation is segmenting on code points, but, as we saw in Section 2.4.2, a character can consist of multiple code points. Some such cases can be normalised to one code point, but this is not always possible (e.g. ą) [7, 19]. Thus, segmenting on code points can give unexpected effects when counting tokens, which can affect the CER (or any metric described in Section 2.2). As such, Unicode defines a method for detecting so-called *grapheme cluster boundaries*[2] [19]. Character-level metrics should therefore segment on these boundaries to ensure a standardised result.

2.4.5 Word segmentation

As it is not clear how to treat aspects like different whitespace and punctuation types, word segmentation is also ambiguous. One approach is to split on `' '`, however, this does not account for other whitespace types (e.g. tabluation). A more robust approach is to split at any Unicode whitespace or any Unicode space, segment or paragraph separator. Alternatively, Unicode defines word boundaries for extracting words without punctuation. These strategies work well on many languages using Latin, Arabic or Devanagari scripts, but they do not work sufficiently e.g. for many Asian languages using other scripts [19].

2.5 Related work

A variety of tools for string comparison exists, targeting different needs. Still, each tool has limitations hindering transparent and reproducible research. Here, we examine six such tools and what limitations in the current landscape motivates a new tool. Specifically, we compare whether tools support token-specific metrics, are libraries or applications, have a transparent documentation (explaining how a tool works rather than just how to use it), have Unicode-aware character and word-segmentation, support custom tokenisation and provide alignment visualisation. As a proxy for how easy the tools can be integrated in other workflows, we also compare their disk footprint. Table 3 provides a brief overview.

The IMPACT centre's ocrevalUAtion tool[3] and the Information Science Research Insititue (ISRI) analytic tools for optical character recognition (OCR) evaluation [14] are applications that create text reports for CER, WER and some character-specific metrics. Neither of these tools address the optimal alignment's non-uniqueness, and both segment characters as code points. While the ISRI analytic tools have been extended with Unicode word segmentation [15],

[2]Note that establishing a completely unambiguous definition of user perceived characters is not always possible [19].

[3]`https://github.com/impactcentre/ocrevalUAtion`

Table 3: Comparison of transcription model evaluation tools

	Calamari	Dinglehopper	ISRI	Jiwer	Meeteval	ocrevalUAtion	Stringalign
Token-specific metrics[+]	Yes[*]	No	Yes[*]	No	No	Yes[*]	Yes
Library or executable	Exe.	Exe.	Exe.	Both	Both	Exe.	Lib.
Transparent docs.	No	No	Yes	No	No	No	Yes
Installed size	2.3 G	0.41 G	0.73 M	2 M	0.24 G	13 M	83 M
Unicode-aware	No	Yes	Words	No	No	No	Yes
Custom tokenisation	No	No	No	Yes	No	No	Yes
Alignment vis.	No	Yes	No	Yes	Yes	Yes	Yes

[+] The different tools support slightly different token-specific metrics
[*] Does not address the non-uniqueness challenge of optimal alignments

ocrevalUAtion use an undocumented regular expression. Moreover, as both tools produce text reports, they can be difficult to integrate in a unified evaluation pipeline, and out of the two, only the ISRI analytic tools support aggregating statistics across samples.

Calamari [21] is a framework for the full OCR pipeline and can report CER, WER and common edits. However, for the latter, it uses a heuristic based on recursively finding the longest matching substrings, which can result in a suboptimal alignment (and is also not unique). Finally, as it is built for Calamari models, it does not support seamless evaluation of non-Calamari models.

Similarly, Dinglehopper[4] is an application for evaluating OCR outputs that integrates with the OCR-D platform [10] and reports CER, WER and a visualisation of the full documents with highlighted edits. Dinglehopper uses grapheme clusters and Unicode word segmentation for characters and words, respectively[5]. However as it is built to integrate with a comprehensive platform and lacks transparent documentation, it can be difficult to integrate into an arbitrary workflow.

Unlike the other tools covered in this section, Meeteval and Jiwer are geared towards ASR-evaluations. Jiwer is lower-level and provides utilities for aligning strings, computing performance metrics like CER and WER, and visualising alignments in the command line. While Jiwer supports arbitrary tokenisation, it does only minimal preprocessing by default, with code points as characters and segmenting words based on `' '`. Meeteval, on the other hand, is higher-level, with visualisations and WER-variants aimed at multi-speaker transcriptions. Meeteval's targeted focus means that it only supports word-tokenisation (based on whitespace-like characters) and it is challenging to customise to other needs.

Earlier work compared several of the tools described above and showed that even standardised metrics like CER and WER cannot be directly compared across implementations [11]. To improve this state, they suggested researchers follow clear OCR evaluation standards and consider provenance, i.e. by keeping track of and reporting evaluation parameters. Analysing the error types, not just the rate has also been shown to give valuable insights [12].

[4] `https://github.com/qurator-spk/dinglehopper`
[5] with the `uniseg` Python Library

3 Design principles and implementation details

3.1 Design principles

As discussed, there is a need for easy-to-use tooling for transparent, reproducible evaluation in automatic transcription research. This section outlines how Stringalign is designed to fill this need. First, we describe how Stringaligns API facilitates transparent comparisons. Then, how its modular and low-dependency architecture enables Stringalign to be used across different setups. Finally, as Stringalign is designed for research, we describe how it adheres to FAIR4RS.

3.1.1 Facilitating reproducible and transparent comparisons

There are, as mentioned in Section 2, many subtle choices that can affect metrics widely used to evaluate automatic transcription models. Motivated by this, Stringalign is, in contrast to other transcription evaluation tools, explicit in its tokenisation and normalisation and base them on standardised Unicode recommendations. Researchers start by defining a tokeniser before comparing strings, or they use utility functions that return the tokeniser and performance metrics. Thus, Stringalign aids in reporting the exact steps needed to reproduce results.

Furthermore, by supporting token-specific metrics, as described in Section 2.3, Stringalign encourages researchers to move beyond summary statistics and get a deeper insight into model performance. Moreover, to tackle the non-uniqueness challenge of optimal string alignment, Stringalign has the option to randomise alignments, or even iterate over all possible optimal alignments, which can be used to estimate averages and standard deviations for the token-specific metrics.

3.1.2 Lightweight architecture

Research code can have, and often has, a very large number of dependencies. These dependencies can, sometimes, be both necessary and beneficial, but each dependency increases the likelihood of conflicts[6].

To aid comparable evaluation across different projects Stringalign is designed to be lightweight and easily fit into various workflows. Effort has been placed into keeping Stringalign's dependencies minimal, without sacrificing usability. Also, Stringalign is scoped to only cover transcription model evaluation. By limiting Stringalign's scope, we make the code more forward-compatible, increasing its long-term maintainability. This focus also enables researchers to include Stringalign in almost any Python project without altering existing environments.

3.1.3 FAIR research software

The past decade, the importance of good practices for research software has been increasingly acknowledged cumulating in, among others, the FAIR4RS [1] principles which state that research software should be:

Findable: Software, and its associated metadata, is easy for both humans and machines to find.

Accessible: Software, and its metadata, is retrievable via standardised protocols.

[6] To illustrate this, consider the Calamari OCR framework, which requires TensorFlow installed with a version between `2.4.0` and `2.15.x` (inclusive). This limits the applications its evaluation pipeline can be integrated with easily.

Interoperable: Software interoperates with other software by exchanging data and/or metadata, and/or through interaction via application programming interfaces (APIs), described through standards.

Reusable: Software is both usable (can be executed) and reusable (can be understood, modified, built upon or incorporated into other software).

As Stringalign is intended for research, following these principles are essential.

Findable To ensure that Stringalign is findable, it is hosted on GitHub[7] with releases posted to PyPI[8] and Zenodo[9]. Moreover, Stringalign follows the latest standards in the Python ecosystem with respect to software metadata[10]. By publishing releases on Zenodo and PyPI, we also ensure that the software is archived with a digital object identifier (DOI). Thus, Stringalign and its associated metadata is easy for both humans and machines to find.

Accessible By following the packaging standards in the Python ecosystem, Stringalign is accessible as both built artifacts (wheels) and source distributions (sdists). Furthermore, the built binaries are available for many platforms and CPU architectures, increasing accessibility. These distributions are available via HTTP, ensuring that Stringalign is retrievable via standardised protocols.

Interoperable Stringalign supports all Python versions $\geq$`3.11`. Furthermore, Stringalign has only one runtime dependency: NumPy and supports all NumPy versions $\geq$`1.24.0`. Thus, Stringalign supports all Python- and NumPy-versions that the Scientific Python ecosystem suggests[11]. Also, Stringalign's wheels are built against the stable Python Application Binary Interface (ABI), which makes them forward compatible with respect to future Python and NumPy[12] releases.

Moreover, Stringalign puts minimal assumptions on input format. This choice contrasts with some ATR evaluation tools. By building on pure Python strings instead of specific file and file system structures, we increase interoperability. Still, to support domain-specific formats, Stringalign's documentation shows how to extract text-lines from PAGE and ALTO XML. Consequently, researchers from any domain needing string comparison can easily use Stringalign.

In short, Stringalign is built to interoperate with Python and other libraries through a clearly defined API based on Python and NumPy primitives.

Reusable To make it easy to use, reuse and modify, Stringalign has a clearly designed API and a test suite, covering over 95% of the code. The intuitive API makes Stringalign user-friendly and the test coverage grants users confidence to adapt Stringalign to their needs without inadvertently breaking functionality.

Stringalign's in-depth documentation[13] further aids researchers in both using and understanding the tools provided. Inspired by *Diátaxis*[14], it has four main parts: Concepts, examples,

[7] `https://github.com/yngvem/stringalign/`
[8] `https://pypi.org/project/stringalign/`
[9] `https://zenodo.org/records/18808052`
[10] E.g. PEP 517 and 518 for build system metadata and PEP 621, 639 and 735 for main software metadata, such as license information, authors, etc.
[11] See SPEC-0: `https://scientific-python.org/specs/spec-0000/`
[12] NumPy guarantees a backwards compatible ABI
[13] Available on `https://www.stringalign.com/`
[14] `https://diataxis.fr/`

a tutorial and API documentation. Concepts cover important topics such as alignment, normalisation, and grapheme clusters; examples contain code snippets that are easy to copy-paste and reuse; the tutorial explains a real-world analysis using Stringalign; and the API documentation describes all public-facing functionality. By helping researchers understand every part of the evaluation, the documentation promotes transparent use of the code.

Thus, Stringalign's transparent documentation, extensive test suite, clear design and permissive open-source MIT license ensures that it can easily be understood, modified, built upon or incorporated into other software.

3.2 Implementation details

In this section, we describe the six public-facing modules in Stringalign.

`stringalign.evaluate` is the main module and has two main classes: the `AlignmentAnalyzer` and the `MultiAlignmentAnalyzer`. The former has functionality for analysing a single reference/prediction pair such as computing TER, token-specific metrics and heuristic-based edit classification, listing edit frequencies and visualising alignments. The latter aggregates `AlignmentAnalyzer`s to compute micro-averaged dataset-level metrics. The `MultiAlignmentAnalyzer` also adds indexes to look up and examine reference/prediction string pairs based on edit operations. See Fig. 1 for an overview of these classes' main functionality.

`stringalign.tokenize` contains tokeniser classes for character and word segmentation. The `GraphemeClusterTokenizer`, `UnicodeWordTokenizer` and `SplitAtWordBoundaryTokenizer` use Stringalign's bindings to the Rust crate `unicode_segmentation`[15]. The `SplitAtWhitespaceTokenizer` can also tokenise words. All tokenisers support normalising text before and after tokenisation. NFC normalisation is used by default, but other normalisers can be supplied.

`stringalign.normalize` provides a `StringNormalizer` class, which can apply Unicode normalisation, case-fold, remove or standardise whitespace, remove non-word characters and resolve confusables using either official Unicode lists or custom mappings. The normaliser also ensures transforms are applied in the correct order. For example, for case-insensitive NFKD normalised comparison, the correct order is case-fold→NFKD-normalise→case-fold→NFKD-normalise [20].

`stringalign.statistics` defines `StringConfusionMatrix`. This class computes token-specific statistics based on counts of edit operators, `TP`, `FP` and `FN`. Researchers will typically not need to instantiate this class directly but instead use `AlignmentAnalyzer.confusion_matrix` for single alignments or `MultiAlignmentAnalyzer.confusion_matrix` for aggregates.

`stringalign.visualize` provides alignment visualisation with more customisation compared to just using the simplified convenience method provided by `AlignmentAnalyzer.visualize`. These visualisations can be seamlessly integrated in, e.g., Jupyter notebooks for data exploration (example in Section 4.2).

`stringalign.align` contains functionality for aligning strings and joining alignments. While this module is the backbone of Stringalign, it is lower-level, so we recommend using the `AlignmentAnalyzer` or the `MultiAlignmentAnalyzer`.

4 Illustrative examples

This section contains three examples illustrating Stringalign's functionality. The first two demonstrate the potential for ambiguity when reporting error rates in practice, while the final example shows Stringalign's easy-to-use API and how researchers can use it to transparently evaluate an OCR transcription. For more examples, including demonstrations of token-specific metrics and

[15] `https://docs.rs/unicode-segmentation/latest/unicode_segmentation/`

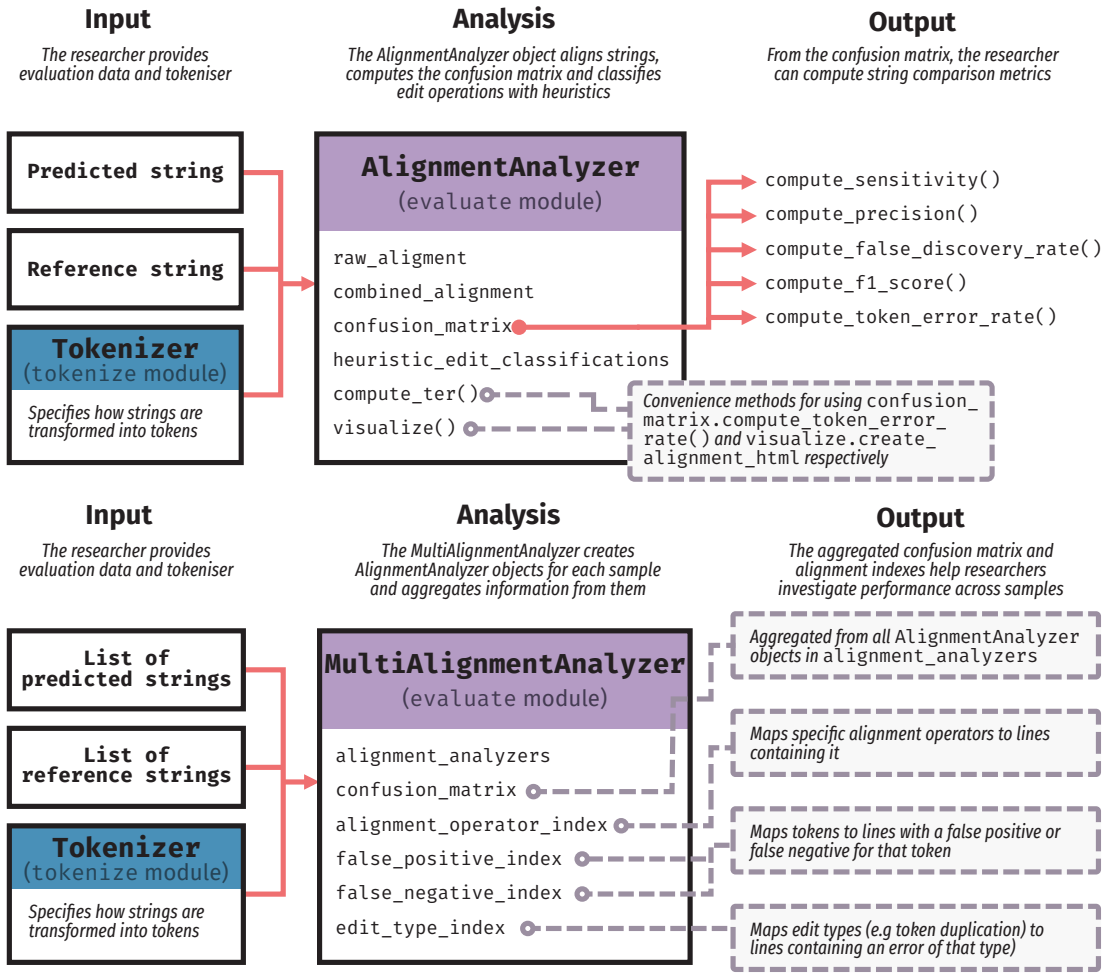

Figure 1: Inputs and most commonly used attributes, methods and properties for the `AlignmentAnalyzer` and `MultiAlignmentAnalyzer` classes

how Stringalign can provide uncertainty estimations for such metrics by randomising alignments, see the Stringalign documentation[13].

4.1 Examples showing ambiguities in CER and WER

Inspired by [11], we use two examples to show where CER and WER calculations differ across tools[16]: A small synthetic example and a real-world example based on the analysis in [11]. For both examples, we apply the six tools discussed in Section 2.5: Calamari, Dinglehopper, the ISRI Analytic Tools, Jiwer, Meeteval and ocrevalUAtion, as well as Stringalign (using a `GraphemeClusterTokenizer` and a `SplitAtWhitespaceTokenizer` with default string normalisation for CER and WER, respectively). The results are rounded to four decimal points (the accuracy reported by the ISRI Analytic Tools and ocrevalUAtion).

[16]These experiments do not consider token-specific metrics as the different tools define various similar, but non-comparable, metrics.

Table 4: Results from the examples in Section 4.1

	Simple example		IMPACT Data	
	CER	WER	CER	WER
Calamari	0.2105	–	0.2133	–
Dinglehopper	0.1176	0.0000	0.1821	0.3445
ISRI	0.2105	0.0000	0.2132	0.4547
Jiwer	0.2105	0.3333	0.2135	0.5117
Meeteval	–	0.2500	–	0.4903
ocrevalUAtion	–*	–*	0.2261+	0.4675+
Stringalign	0.1176	0.2500	0.2120	0.4903

* ocrevalUAtion crashed with the emoji input.
+ Manually computed macro average.

4.1.1 Small synthetic example

We applied the evaluation tools on the text strings `'That was\nclose! 🤔'` and `'That was\nclose!'` (stored as NFC-normalised UTF-8 text files). This example is a variant of a documentation example[17] and has a whitespace character different from space and a multi code point character. Table 4 shows that different tools reported different CER and WER values. Dinglehopper and Stringalign's CER differ from the rest, as other tools count 🤔 as three characters. Meanwhile, for WER, Dinglehopper and ISRI see no error, as they use Unicode word segmentation, which removes emojis. Jiwer only tokenises words based on `' '` and does not split at newlines, while Meeteval and Stringalign (here) splits on any space-like code point.

4.1.2 Real-world example

For a multi-sample example, we applied the tools on the 378 pages from the IMPACT dataset [13] analysed in [11][18], using the same reference and predicted texts (`GT` and `GT4Hist`). Table 4 shows the overall performance, demonstrating again that tokenisation differences causes deviations in CER and WER between tools. ocrevalUAtion cannot aggregate results, so for Table 4, we extracted the CER and WER values from ocrevalUAtion's HTML reports and computed their average. This becomes a macro-average, which has the downside of weighting short and long samples equally. Furthermore Dinglehopper's metrics are notably low, which there are two reasons for. First, it also macro-averages the results, which in this case leads to a slightly lower CER. More importantly, however, is its use of an undocumented list for resolving certain confusables. If we resolve the same list of confusables with Stringalign, then the Stringalign and Dinglehopper's CER coincide for all samples and WER coincide for most values (the overall CER results still differs due to aggregation differences). Thus, consistent with the previous example, these results show that different tools do different processing under the hood (which is not always fully documented), resulting in different CER and WER for the same string pairs.

[17] `https://stringalign.com/auto_examples/plot_emoji_ocr_evaluation.html`
[18] Code to reproduce is on Stringalign's GitHub repo and the IMPACT data is available with the code for [11] (`https://github.com/cneud/hip21_ocrevaluation`)

4.2 Example use-case

Here, we include an abbreviated version of the tutorial example in Stringalign's documentation[19]. In particular, we use Stringalign to investigate the handwritten text recognition (HTR) model `Sprakbanken/TrOCR-norhand-v3`'s performance on the test split of `Teklia/NorHand-v3-line` (both pulled from Huggingface).

Figures 2 and 3 illustrate how Stringalign makes it straightforward to inspect transcription errors. The code is run in a Jupyter Notebook, and consists of five steps: First, we import modules and load data, before creating a `MultiAlignmentAnalyzer`, explicitly stating how to tokenise strings. Next, we compute the full dataset WER and list the most common edits. From these results, we see that the model struggles with the name "Ivar" and replaces "jej" and "eg", which both mean the same as "jeg" ("I" in Norwegian Bokmål). This requires further investigation, but could indicate that the model has overfitted to the Bokmål form of Norwegian and is struggling with other dialects.

Finally, we see how we can use Stringalign's visualisation utilities to inspect errors more closely by displaying alignments next to the corresponding images (this requires the optional dependency `Pillow`). We see that the letter "I" in "Ivar" can resemble an "S". Also, the model appears to struggle with printed text and including more printed text in the training could mitigate this problem.

Thus, Stringalign's API makes it straightforward to investigate errors made by a transcription model. This code is only parts of one example from Stringalign's vast gallery that researchers can copy and learn from.

5 Conclusion and future work

This paper introduces Stringalign, an open-source Python library for comparing strings. Our work shows that where other tools can give ambiguous results, Stringalign improves on the current trancription model evaluation landscape with a transparent API, detailed documentation and tools for exploring the types of mistakes a transcription model makes, which in turn facilitates reproducible evaluation. Moreover, Stringalign has an open-source license, requires few dependencies and is modular in design, which makes it painless to adapt into a variety of existing workflows.

This work focuses on transparent string comparison, one of the two challenges of ATR evaluation highlighted in [11]. As future work, Stringalign could be further extended. For example, while performance bottlenecks are implemented in Rust, algorithmic improvements could increase speed [16]. Furthermore, the author's are only familiar with latin scripts, so this work has not focused on non-Latin scripts. Another valuable extension would, therefore, be adding support and examples for languages requiring custom tokenisation (and in particular languages using non-Latin scripts). As Stringalign is open-source, we invite the research community to provide feedback and code.

Disclosure of Interests

The authors have no competing interests to declare that are relevant to the content of this article

[19] `https://stringalign.com/worked_example.html`

```python
# Import functionality and load transcriptions
import pandas as pd
from IPython.display import HTML, display
import stringalign as sa
from stringalign.evaluate import MultiAlignmentAnalyzer

data = pd.read_json("transcription.json")

# Create the alignment analyzer and compute the WER
analyzer = MultiAlignmentAnalyzer.from_strings(
    references=data["reference"],
    predictions=data["predicted"],
    metadata=[{"im_path": r.img} for r in data.itertuples()],
    tokenizer=sa.tokenize.UnicodeWordTokenizer(),
)
print(f"The WER is: {analyzer.compute_ter():.2%}")

# Print the three most common edit operations
for edit, count in analyzer.edit_counts["raw"].most_common(3):
    print(f"Edit '{edit}' occured {count} times.")

# Ivar is one of the most commonly missed words
for aa in analyzer.false_negative_index["Ivar"]:
    img = sa.visualize.create_html_image(aa.metadata["im_path"])
    alignment_html = aa.visualize(space_alignment_ops=True)
    display(HTML(img + alignment_html))

# Look at the three lines with the lowest WER
alignment_analyzers = sorted(
    analyzer.alignment_analyzers,
    key=lambda aa: -aa.compute_ter(),
)

for aa in alignment_analyzers[:5]:
    img = sa.visualize.create_html_image(aa.metadata["im_path"])
    alignment_html = aa.visualize(space_alignment_ops=True)
    display(HTML(img + alignment_html))
```

```
The WER is: 15.05%
Edit 'REPLACED 'jej' -> 'jeg'' occured 7 times.
Edit 'REPLACED 'Ivar' -> 'Svar'' occured 7 times.
Edit 'REPLACED 'eg' -> 'og'' occured 7 times.
Edit 'REPLACED 'De' -> 'de'' occured 4 times.
Edit 'DELETED '1'' occured 3 times.
```

Figure 2: Code example with text output showing how Stringalign can help in inspecting the performance of a handwriting text recognition model. Visualisations generated by this code are shown in Fig. 3.

(a) (b)

Figure 3: The visual output of Fig. 2 (abbreviated). (a) shows the first three lines with a false positive "Ivar" and (b) shows the three lines with highest WER.

References

[1] Barker, M., Chue Hong, N.P., Katz, D.S., Lamprecht, A.L., Martinez-Ortiz, C., Psomopoulos, F., Harrow, J., Castro, L.J., Gruenpeter, M., Martinez, P.A., et al.: Introducing the FAIR principles for research software. Sci. data **9**(622), 1–6 (2022). https://doi.org/10.1038/s41597-022-01710-x

[2] Domingos, P.: A few useful things to know about machine learning. Commun. ACM **55**(10), 78–87 (Oct 2012). https://doi.org/10.1145/2347736.2347755

[3] Enstad, T., Trosterud, T., Røsok, M.I., Beyer, Y., Roald, M.: Comparative analysis of optical character recognition methods for Sámi texts from the National Library of Norway. In: Proc. Jt. 25th Nord. Conf. Comput. Linguist. 11th Balt. Conf. Hum. Lang. Technol. (NoDaLiDa/Balt.-HLT 2025). pp. 98–108. University of Tartu Library, Tallinn, Estonia (Mar 2025)

[4] Garrido-Munoz, C., Rios-Vila, A., Calvo-Zaragoza, J.: Handwritten text recognition: A survey. IEEE Trans. Pattern Anal. Mach. Intell. pp. 1–20 (2025). https://doi.org/10.1109/TPAMI.2025.3646002

[5] Levenshtein, V.: Binary codes capable of correcting spurious insertions and deletions of ones. Probl. Inf. Transm. **1**, 8–17 (1965)

[6] Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. **10**(8), 707–710 (1966), original in Russian in Dokl. Akad. Nauk SSSR 163, 4, 845–848, 1965

[7] Moran, S., Cysouw, M.: The Unicode Cookbook for Linguists: Managing writing systems using orthography profiles, Translation and Multilingual Natural Language Processing, vol. 10. Language Science Press (2018). https://doi.org/10.5281/zenodo.1300528

[8] Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (Mar 2001). https://doi.org/10.1145/375360.375365

[9] Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970). https://doi.org/10.1016/0022-2836(70)90057-4

[10] Neudecker, C., Baierer, K., Federbusch, M., Boenig, M., Würzner, K.M., Hartmann, V., Herrmann, E.: OCR-D: An end-to-end open source OCR framework for historical printed documents. In: Proc. 3rd Int. Conf. Digit. Access Textual Cult. Herit. pp. 53–58 (2019). https://doi.org/10.1145/3322905.3322917

[11] Neudecker, C., Baierer, K., Gerber, M., Clausner, C., Antonacopoulos, A., Pletschacher, S.: A survey of OCR evaluation tools and metrics. In: Proc. 6th Int. Workshop Hist. Doc. Imaging Process. (HIP '21). pp. 13–18 (2021). https://doi.org/10.1145/3476887.3476888

[12] Nguyen, T.T.H., Jatowt, A., Coustaty, M., Nguyen, N.V., Doucet, A.: Deep statistical analysis of OCR errors for effective post-OCR processing. In: 2019 ACM/IEEE Jt. Conf. Digit. Libr. (JCDL). pp. 29–38 (2019). https://doi.org/10.1109/JCDL.2019.00015

[13] Papadopoulos, C., Pletschacher, S., Clausner, C., Antonacopoulos, A.: The IMPACT dataset of historical document images. In: Proc. 2nd Int. Workshop Hist. Doc. Imaging Process. (HIP '13). p. 123–130. ACM (2013). https://doi.org/10.1145/2501115.2501130

[14] Rice, S.V., Nartker, T.A.: The ISRI analytic tools for OCR evaluation version 5.1. Tech. Rep. TR-96-02, Information Science Research Institute (1996)

[15] Santos, E.A.: OCR evaluation tools for the 21st century. In: Proc. 3rd Workshop Use Comput. Methods Study Endanger. Lang. Vol. 1 (Pap.). pp. 23–27 (2019)

[16] Ukkonen, E.: Algorithms for approximate string matching. Inf. Control **64**(1), 100–118 (1985). https://doi.org/10.1016/S0019-9958(85)80046-2

[17] Unicode Consortium: Unicode normalization forms. Unicode Technical Report UAX #15, Unicode Consortium (Jul 2025), `https://unicode.org/reports/tr15/`

[18] Unicode Consortium: Unicode security mechanisms. Unicode Technical Report UAX #39, Unicode Consortium (Sep 2025), `https://unicode.org/reports/tr39/`

[19] Unicode Consortium: Unicode text segmentation. Unicode Technical Report UAX #29, Unicode Consortium (Aug 2025), `https://unicode.org/reports/tr29/`

[20] Unicode Consortium: The Unicode® standard version 17.0. Core specification, Unicode Consortium (Sep 2025), `https://www.unicode.org/versions/Unicode17.0.0/core-spec/`

[21] Wick, C., Reul, C., Puppe, F.: Calamari - A High-Performance Tensorflow-based Deep Learning Package for Optical Character Recognition. Digit. Humanit. Q. **14**(2) (2020)

[22] Wiechetek, L., Pirinen, F., Kappfjell, M.L., Trosterud, T., Gaup, B., Moshagen, S.N.: The ethical question – use of indigenous corpora for large language models. In: Proc. 2024 Jt. Int. Conf. Comput. Linguist. Lang. Resour. Eval. (LREC-COLING 2024). pp. 15922–15931. ELRA and ICCL, Torino, Italia (May 2024)

[23] Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., da Silva Santos, L.B., Bourne, P.E., et al.: The FAIR guiding principles for scientific data management and stewardship. Sci. data **3**(160018), 1–9 (2016). https://doi.org/10.1038/sdata.2016.18

[24] Zha, D., Bhat, Z.P., Lai, K.H., Yang, F., Hu, X.: Data-centric AI: Perspectives and challenges. In: Proc. 2023 SIAM Int. Conf. Data Min. (SDM). pp. 945–948. SIAM (2023). https://doi.org/10.1137/1.9781611977653.ch106