# Sentence splitters benchmark

A.P. Zavyalova, P.A. Martynyuk, R.S. Samarev

Bauman Moscow State Technical University, Moscow, Russia

**Abstract.** There are multiple implementations of text into sentences splitters including open source libraries and tools. But the quality of segmentation and the performance of each segmentation tool are very different. Moreover, it is convenient for NLP developers to have all libraries written in the same programming language, except when using some kind of integration programming language. This paper considers two aspects - building a uniform framework and estimating language features of the modern and popular programming language Julia from one side. And the performance estimation of sentence splitting libraries as is. The paper contains detailed performance results, samples of texts after splitting, and a list of some typical issues related to sentence splitting.

**Keywords:** *segmentation, sentence, splitting, NLP, Julia language, benchmark, text analysis.*
**DOI:** 10.14357/20790279230119

## Introduction

Segmentation of text or splitting linear text into fragments is one of the fundamental operations required for most applied NLP tasks. It refers to one of the stages of natural language text processing - the pre-syntactic analysis of the text[4].

Despite a tendency to replace traditional methods of NLP with neural-net-based ones in the last years, where document analysis might be done without intermediate stages, traditional methods are still used in some cases. E.g. when we need to calculate statistics of words, parts of speech in a document, do some specific markup, etc. There are multiple implementations of text into sentences splitters including open source libraries and tools. But the quality of segmentation and the performance of each segmentation tool are very different.

Also, we need to take into account a technological stack that is used by developers of NLP applications. In most cases, developers prefer to use libraries with API in the same language which is used in their applications development.

Let's take a look at some of the most commonly used segmentation tools.

## 1. Segmentation tools review

### 1.1. NLTK

NLTK is a set of natural language analysis tools written in Python language. It includes a set of libraries for classification, tokenization, word stem finding, tagging, parsing, and semantic text analysis. NLTK supports many languages depending on the specific task. This tool implements the segmentation method suggested by S. Bird, etc. [3] using unsupervised learning (learning algorithm "without a teacher") to build a model of abbreviated words, phrases, and words that begin sentences and then find the supply boundary.

### 1.2. Stanford CoreNLP

CoreNLP is a set of NLP analysis tools written in the Java language. CoreNLP allows users to obtain linguistic annotations for text, including tokens, sentence boundaries, and many others. Currently supports 6 languages: Arabic, Chinese, English, French, German and Spanish. This software implements a combined segmentation method for text segmentation. A combined method means the combination of machine learning and rule-based methods. Text processing suggested by C. Manning, etc. [11] executes in the form of a pipeline, at each stage in which the user receives linguistic annotations. Each stage fulfills its function. For example, a pipeline can consist of token boundaries, parts of speech, named entities, and sentence segmentation. Each of these stages can be implemented in different ways.

### 1.3. SpaCy

SpaCy is a set of advanced word processing tools written in Python and Cython programming languages. Unlike NLTK, which is widely used for teaching and research, SpaCy focuses on providing software for production use. With its internal machine learn-

ing library "Thinc", SpaCy supports the connection of statistical models trained by popular machine learning libraries: TensorFlow, PyTorch, and MXNet. SpaCy provides models for part-of-speech tagging, dependency analysis, text segmentation, and named entity recognition. Out-of-the-box statistical models for these tasks are available in 17 languages, including English, Portuguese, Spanish, Russian, and Chinese. There is also a multilingual NER model. Additional tokenization support for over 65 languages allows users to train models on their own datasets. This tool implements two segmentation methods: a method based on heuristic rules and a method based on machine learning, namely "decision trees". The heuristic rules code is in the SpaCy documentation [9]. The Dependency parser uses a variant of the non-monotonic arc-eager transition system suggested by M. Honnibal, etc. [8], with the addition of a "break" transition to performing the sentence segmentation. The pseudo-projective dependency transformation suggested by J. Nivre, etc. [12] is used so that the parser can predict non-projective parsing.

### 1.4. Apache OpenNLP

Apache OpenNLP is a machine learning-based toolkit for natural language processing. OpenNLP supports the most common NLP tasks, such as tokenization, sentence segmentation, and others. OpenNLP does not support languages out of the box. The framework can be used to train a model for any language. However, there are adapted models for Danish, German, English, Spanish, Dutch, Portuguese, and Sami. This tool implements a sentence segmentation method based on machine learning, namely unsupervised learning [2].

### 1.5. WordTokenizers.jl

Apache OpenNLP is a machine learning-based toolkit for natural language processing. OpenNLP supports the most common NLP tasks, such as tokenization, sentence segmentation, and others. OpenNLP does not support languages out of the box. The framework can be used to train a model for any language. However, there are adapted models for Danish, German, English, Spanish, Dutch, Portuguese, and Sami. This tool implements a sentence segmentation method based on machine learning, namely unsupervised learning [14].

### 1.6. Sentencize.jl

This package is also written in Julia and re-implements the Python-based package sentence-splitter [1]. The sentence-splitter package is a Python implementation of the Lingua :: Sentence module [13] – a Perl-based extension for breaking text paragraphs into sentences. Sentencize.jl supports the following languages: Catalan, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hungarian, Icelandic, Italian, Latvian, Lithuanian, Norwegian (Bokmål), Polish, Portuguese, Romanian, Russian, Slovak, Slovenian, Spanish, Swedish, Turkish. This tool implements a sentence segmentation method based on heuristic rules suggested by P. Koehn, etc. [10].

### 1.7. Outcomes of the review

Heuristic methods are still widely used in text processing. They are easy to use and do not require significant memory resources. The advantage of these methods is also the stability and predictability of their work. The supply boundary is determined by matching against a set of rules. However, in texts with a special arrangement of punctuation marks (e-mail and physical addresses), mistakes might occur. The disadvantage of these methods is difficult to modify the rules in case of significant changes. Machine learning methods are more delicate than rule-based methods. They are able to find places with a special arrangement of punctuation marks, as well as various mistakes, and avoid them when splitting the text into sentences. However, when using the wrong training sample, the behavior of the sentence splitter in a given text is poorly predictable. The advantage of modern pre-trained neural networks over all the above is context dependence. However, as with statistical methods, the behavior depends on the training sample and is poorly predictable, and the training costs may not be justified in relation to the quality of the text splitting. Combined methods should take into account the shortcomings of machine learning and rule-based methods. But these combined methods require additional RAM resources, which can slow down the process. Let's test all the above segmentation tools to find the best one.

## 2. Benchmark

### 2.1. Julia as an integration platform

Julia language is available for production use since 2018. Then, the community is continuously growing up [7]. Julia can still be called a new language, but it already has impressive functionality. For example, even a new implementation of old libraries and algorithms to solve the problem of text segmentation is mentioned above.

Julia offers packages designed for high performance from the beginning. In machine learning, and natural language processing an immense advantage of Julia is that most packages with similar functionality were developed after the creation of popular Python, Java, and R libraries. But, at the same time, Julia developed much later compared with other popular language stacks.

For many years, a common approach for development libraries that are applicable for multiple lan-

guages use was the way of building C/C++ based binary compiled dynamic libraries with C-style exported names. These libraries might be used in scripting languages with appropriate wrapper code prepared manually or with tools like SWIG.

In the case of Java, there is an interface JNI for calling this kind of library. And there is JSR 223 Java Scripting API with multiple 3-rd party execution modules for exact languages to run a script or code snippet. At the same time, a programmer has to do a lot of additional operations to even activate non-Java code and transfer data to the code e.g. in Python. In the case of Python, as a result of the low performance of baseline implementation of CPython, there is the fragmentation of the language dialects with Cython, PyPy, NIM, etc. And, there are the same issues with additional programmers' work to integrate any other language into a Python project with something other than a binary dynamic library. And, even after 30 years of development, the main issue of Python is low performance and a ponderous toolkit to force a Python code to be a production-applicable application.

Key developers of Julia paid attention to integration features and implemented a way to execute other language scripts with their libraries rather than just giving a binary interface. And, moreover, even Julia code might be integrated into other applications with Julia Embedded API and a system image built on a Julia code.

This allowed us to speak about integration possibilities of other languages and libraries into a Julia code, and take into account all the advantages and disadvantages of all the mentioned above packages, as well as use ready-made solutions or their parts written in other languages. Let's look at some of its integration features.

Julia can initially (without any "glue" code, code generation, or compilation) [6] directly call the C and Fortran libraries (fig. 1).

Also, there are special packages to call Python, R, Java code. The following sample uses the package PyCall (fig. 2). It allows us to call Python functions and even to write Python code inside Julia programmers.

JavaCall allows only to call Java packages from within Julia code (fig. 3). First of all, we need to initialize the Java Virtual Machine before we can call any other functions in this module. After that, we get access to all the functions of this package, along with the Java program.

In this benchmark, we take into account all the Julia possibilities making Julia a perfect tool for comparison libraries in different languages under equal conditions.

### 2.2. Benchmark details

The idea of the benchmark is to compare text splitting into sentences with the segmentation tools mentioned above using the same set of text and a reference markup of sentences. We illustrate the principle of operation in figure 4. In the input section, we have a marked-up dataset, on the basis of which we form plain text with and without markup. Next, we split the text without markup into sentences using each segmentation tool. After that, we make a comparison of the obtained sentences with its reference. In the result

```
# Example of calling C library
julia> ccall((:sqrt, "libm"), Float64, (Float64,), 49.0)
7.0
```

**Fig. 1.** Example of calling C library

```
# Adding Python dependencies
using Conda
using PyCall
Conda.add("spacy")

Conda.add("nltk")
py"""
import nltk
nltk.download('punkt')
"""
# Example of calling Python library
using PyCall
local spacy = pyimport("spacy")
Dict(:nlp => spacy.load(spacy_model, disable = ["tagger", "ner"]))
```

**Fig. 2.** Example of calling Python library

```
# Custom Java package initialization
using JavaCall
function activate_java_deps()
    local jar_dir = joinpath(project_root,
        "javaNLP", "build", "libs",
        "sentence_splitter_wrapper-0.1-SNAPSHOT-all.jar")

    JavaCall.init(["-Djava.class.path=" * normpath(jar_dir)])
end
# Importing a custom Java class
opennlp_benchmark =
    jvm_benchmark("opennlp", "OpenNLP", @jimport(ru.bmstu.sentencespliter.
         benchmark.OpenNLP))
# Calling Java function inside the jvm_benchmark() function
benchmark = (data) -> begin
        local jwrapper = data[:jWrapperClass](())
        Dict(
            :jvm_res =>
                jcall(jwrapper,
                    "splitSentences",
                    JString,
                    (JString, JString),
                    data[:input_fn],
                    data[:output_fn])
        )
end
```

**Fig. 3.** Example of calling Java library

section, we calculate f-measures and the splitting performance time.

We illustrate the benchmark architecture as the package diagram in figure 6. Here you can see how Julia, Python, and Java packages from its libraries connect with each other. SentenceSplitterBenchmark.jl is our package written in Julia language only. We highlight other packages belonging to different programming languages.

The benchmark framework developed in this work gives a simple way to describe a frame for a new library or algorithm for testing. Any specific frame is stored in the structure like this: (fig. 5)

For each segmentation tool, its own object with the described structure is created. That object describes a function for doing some work before starting the benchmark, a function for running the benchmark, and a function for collecting results if these are not available directly. That description is looking like a declarative form.

It was decided to measure the performance (speed of each tool) using BenchmarkTools.jl [5]. This solution will solve problems with launch heterogeneity by averaging measurements.

In the example above (fig. 8), let's pay attention to the fact that the teardown is set in such a way that running the JVM does not affect the result.

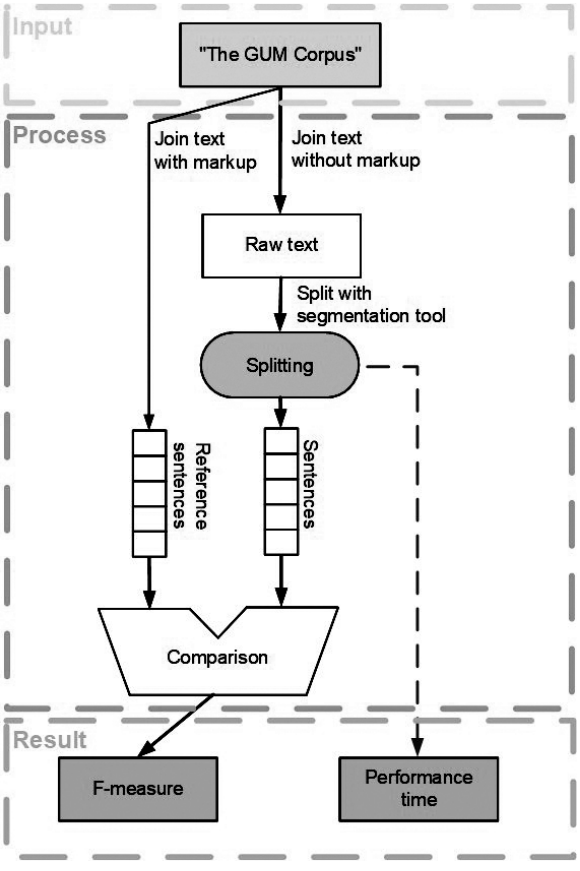


**Fig. 4.** Principle of benchmark operation

```
struct BenchmarkFrame
    setup::Function
    benchmark::Function
    teardown::Function

    data::Dict
    ...
end
```

**Fig. 5.** BrenchmarkFrame structure

Then we can call all benchmark objects of segmentation tools sequentially via evaluate_list() with specifying the number of samples and the benchmark duration after that the trial will be terminated(fig.7). Parameters samples = 100, seconds = 40 passed directly into the BenchmarkTools.jl module which is providing running of tests and collecting a stable execution statistics.

### 2.3. Setup

Let's compare the segmentation tools for the problem of splitting the text into sentences according to the following criteria:

1. Execution performance and required resources.
2. Quality of text segmentation.
3. Errors of text segmentation.

It was decided to measure the performance (speed of each tool) using the BenchmarkTools.jl [5]. Testing will be performed on 5840 sentences from "The GUM Corpus" [16].

GUM stands for Georgetown University Multilayer Corpus, a corpus of English texts with different text types. This corpus consists of interviews, news, travel guides, how-to guides, academic writing, biographies, fiction, online forum discussions, spontaneous face-to-face conversations, political speeches, textbooks, and vlogs. Such a variety of texts allows us to test segmentation close to natural conditions when we don't know what is the input text. For example, one sentence from the GUM Corpus is presented in fig.9

Each token (word, punctuation mark) has its annotation. Our task was to test sentence segmentation, so we use GUM annotation only as a reference for sentence boundaries. Thus, we will combine all the sentences of the corpus into plain text without line breaks (punctuation will remain) and compare the splitting using each of the tools with the reference markup.

We will take average values of the time to obtain the execution performance of tools.

The task of text segmentation is the task of classification (hyphenation might be present or not). The effectiveness of the text segmentation tool can be numerically assessed by the quality of predictions for the

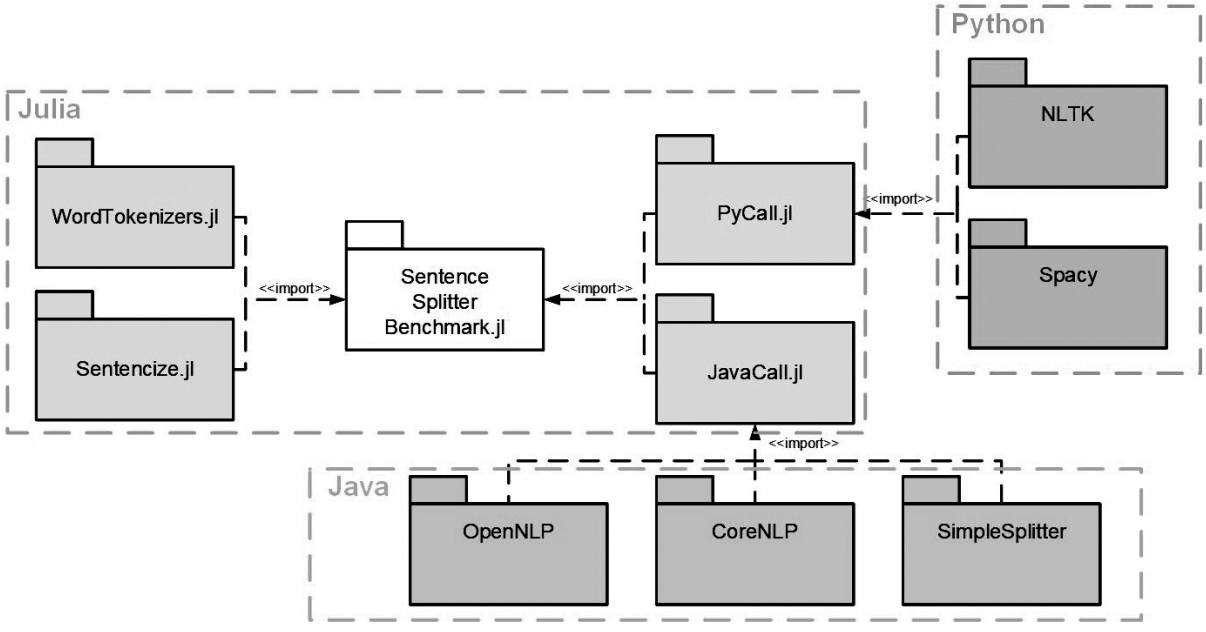


**Fig. 6.** Package diagram

```
evaluate_splitters() = evaluate_list(() -> [
        nltk_benchmark(samples = 100, seconds = 40),
        opennlp_benchmark(samples = 100, seconds = 40),
        corenlp_benchmark(samples = 100, seconds = 40),
        julia_wt_benchmark(samples = 100, seconds = 40),
    ]
)

evaluate_all() = evaluate_splitters()
```

**Fig. 7.** Example of calling all benchmark objects

```
# BenchmarkFrame structure instance for NLTK
nltk_benchmark = BenchmarkFrame(
    "nltk", "NLTK",
    setup = (data) ->
        Dict(:nlt => pyimport("nltk")),
    benchmark = (data) ->
        Dict(:output_sentences => data[:nlt].sent_tokenize(data[:input_text]))
)

# BenchmarkFrame structure instance for Java segmentation tools
# can be called for OpenNLP and CoreNLP
jvm_benchmark(name, description, jWrapperClass) = BenchmarkFrame(name,
     description;
    setup = (data) -> begin
        local output_fn = joinpath(out_dir, "jvm_output", "output_" * data[:name
            ] * ".txt")
        Dict(:jWrapperClass => jWrapperClass, :output_fn => output_fn)
    end,
    benchmark = (data) -> begin
            local jwrapper = data[:jWrapperClass](())
            Dict(
                :jvm_res =>
                    jcall(jwrapper,
                        "splitSentences",
                        JString,
                        (JString, JString),
                        data[:input_fn],
                        data[:output_fn])
            )
    end,
    teardown = (data) -> begin
        local text = open(f->read(f, String), data[:output_fn])
        local sentenses = split(text," \n ")
        pop!(sentenses)
        Dict(:output_sentences => sentenses)
    end
)

# BenchmarkFrame structure instance for WordTokenizers.jl
julia_wt_benchmark = BenchmarkFrame(
    "julia_wt", "WordTokenizers.jl",
    benchmark = (data) ->
        Dict(:output_sentences => split_sentences(data[:input_text]))
)
```

**Fig. 8.** BrenchmarkFrame structure for NLTK and WordTokenizers.jl

```
# text = Insights from Eye-Tracking
1   Insights    insight NOUN    NNS Number=Plur 0   root    0:root  Discourse=
    elaboration-additional:2->1:0|Entity=(3-abstract-new-1-coref
2   from    from    ADP IN  _   5   case    5:case  _
3   Eye eye NOUN    NN  Number=Sing 5   compound    5:compound  Entity=(4-
    abstract-new-3-coref(5-object-new-1-coref)|SpaceAfter=No|XML=<w>
4   -   -   PUNCT   HYPH    _   3   punct   3:punct SpaceAfter=No
5   Tracking    tracking    NOUN    NN  Number=Sing 1   nmod    1:nmod:from
    Entity=4)3)|XML=</w>
```

**Fig.9.** One sentence from the GUM Corpus

test sample. The forecasts made are considered either positive or negative, and the expected judgments are true or false.

Four classes that include all predictions made by the segmentation tool are shown in Table 1. Predictions must be made for each token (word, punctuation mark) in the sentence. So each token goes to one of the four classes (TP, FN, TN, FN) according to prediction.

**Table 1**

Confusion matrix

| Class | Predict | Result | Explanation |
|---|---|---|---|
| TP, True Positive | 1 | 1 | Line break where it should be |
| FP, False Positive | 1 | 0 | Line break NOT where it should be |
| TN, True Negative | 0 | 1 | There is no line break, but there should be |
| FN, False Negative | 0 | 0 | There is no line break, and there shouldn't be |

We use the following general indicators [15]:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Error = 1 - Accuracy = \frac{FP + FN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F_{measure} = 2 * \frac{Precision * Recall}{Precision + Recall}$$

### 2.4. Execution time performance

Table 2 shows the results of measuring the execution time of the text segmentation tools.

The table is sorted in ascending order of average execution time for each tool.

The first three lines are the simplest and the fastest algorithms with the supposedly lowest segmentation quality. Next, the best results are WordTokenizers.jl and OpenNLP. The difference in the average execution time of 100 iterations ranges from 0.1s to 2s at each benchmark run, with WordTokenizers.jl having an advantage. Thus, the WordTokenizers.jl (Julia-based with heuristics) can be said to perform sentence splitting faster than others. At this step, the hypothesis about rule-based methods is only partially confirmed. One of the rule-based tools shows the fastest results, the other two show the slowest. It all depends on the quality of heuristics. The next step is to estimate the quality of the sentence splitting.

**Table 2**

Performance

| Tool Name | Iterations | Time(ms) |
|---|---|---|
| Julia split() | 100 | 0,57 |
| Julia split() with file | 100 | 3,58 |
| SimpleSplitter | 100 | 13,77 |
| WordTokenizers.jl | 100 | 30,20 |
| OpenNLP | 100 | 30,73 |
| NLTK | 100 | 216,07 |
| CoreNLP | 100 | 311.99 |
| Spacy (Rule-based) | 43 | 1036.76 |
| Sentencize.jl | 7 | 6476.84 |
| Spacy (Dependency parser) | 6 | 10690.8 |

### 2.5. Text segmentation quality

As each segmentation tool has its own tokenizer, the number of tokens (predictions) for each tool might be different. Thus, we can compare the quality of segmentation with relative values (accuracy, error, precision, error, f1).

It is worth clarifying that "Julia split () with file" is omitted in this table, since "Julia split ()" and "Julia split () with file" are one splitting algorithm, therefore, the results of the accuracy estimation will be the same.

The quality of text segmentation is shown in Table 3. As we can see, the best quality is shown by Sentencize.jl - port of a rule-based Perl extension for

**Table 3**

Comparison metrics results.

| Tool Name | tp | fp | tn | fn | accuracy | error | precision | recall | f1 |
|---|---|---|---|---|---|---|---|---|---|
| Sentencize.jl | 6330 | 254 | 107813 | 1078 | 0,99 | 0,01 | 0,96 | 0,85 | 0,905 |
| NLTK | 6269 | 283 | 107787 | 1139 | 0,99 | 0,01 | 0,96 | 0,85 | 0,898 |
| OpenNLP | 6255 | 276 | 107791 | 1153 | 0,99 | 0,01 | 0,96 | 0,84 | 0,897 |
| CoreNLP | 6278 | 362 | 107786 | 1130 | 0,99 | 0,01 | 0,95 | 0,85 | 0,894 |
| WordTokenizers.jl | 6140 | 264 | 107809 | 1268 | 0,99 | 0,01 | 0,96 | 0,83 | 0,889 |
| Spacy (Dependency parser) | 6631 | 934 | 107268 | 777 | 0,99 | 0,01 | 0,88 | 0,90 | 0,886 |
| Spacy (Rule-based) | 6183 | 994 | 107531 | 1225 | 0,98 | 0,02 | 0,86 | 0,83 | 0,848 |
| SimpleSplitter | 5760 | 772 | 107847 | 1648 | 0,98 | 0,02 | 0,88 | 0,78 | 0,826 |
| Julia split() | 5760 | 878 | 107780 | 1648 | 0,98 | 0,02 | 0,87 | 0,78 | 0,820 |

**Table 4**

Number of 2nd type errors

| Tool Name | Errors |
|---|---|
| Sentencize.jl | 0 |
| NLTK | 3 |
| OpenNLP | 0 |
| CoreNLP | 84 |
| WordTokenizers.jl | 6 |
| Spacy (Dependency parser) | 135 |
| Spacy (Rule-based) | 458 |

sentence splitting. At this step, again the hypothesis is only partially confirmed. The best quality is shown by another heuristic.

### 2.6. Types of segmentation errors

For a more complete understanding of the work of each segmentation tool, it is necessary to take into account the nuances of their work. This can be done by printing out the markup errors and comparing them to the reference markup. During the analysis, we identified two types of errors: errors in setting the line break and errors in recognizing tokens.

Errors of the 1st type are associated with the absence of punctuation marks in the source text, headings, and enumerations. All segmentation tools make similar errors in the same places. There is no point in showing them.

Errors of the 2nd type are more exotic. They depend on the segmentation tool and are not repeated in the analyzed tools. It is associated with a specific segmentation algorithm (method). These errors consist of incorrect recognition of tokens and occur with consecutive punctuation marks. Consider some examples in Table 5.

**Table 5**

Error examples

| **WordTokenizers.jl** | |
|---|---|
| Reference markup | Markup error |
| - It ‘s a little bit like Achilles and the turtle.\vskip 3pt- ... love story and romance and surprises and tragedies and all this but alsothis structure interested me a lot. | - It ‘s a little bit like Achilles and the turtle.... love story and romance andsurprises and tragedies and all this but also this structure interested me a lot. |
| **NLTK** | |
| Reference markup | Markup error |
| - A Connecticut Yankee in King Arthur ‘s Court ( solo )\vskip 3pt- Ed.: See the LibriVox catalog for a full index. | - A Connecticut Yankee in King Arthur ‘s Court ( solo ) Ed.\ vskip 3pt- : See the LibriVox catalog for a full index. |
| **CoreNLP** | |
| Reference markup | Markup error |
| - Had they died fast or were they now suffering a fate far worse..? | - Had they died fast or were they now suffering a fate far worse.\vskip 3pt- .\vskip 3pt- ? |
| **Spacy (Dependency parser)** | |
| Reference markup | Markup error |
| - Finally, our study complements Navarro’s (2016) automatic metrical analyses of Spanish Golden Age sonnets, by covering a wider period and focusing on enjambment. | - Finally, our study complements Navarro\vskip 3pt- s (2016) automatic metrical analyses of Spanish Golden Age sonnets, by covering a wider period and focusing on enjambment. |
| **Spacy (Rule-based)** | |
| Reference markup | Markup error |
| - The severe concerns underpinning the alleged crisis have several dimensions relating to: (a) the (small) amount of published replication research; (b) the (poor) quality of replication research; and (c) the (lack of) reproducibility, which refers to the extent to which findings can (not) be reproduced in replication attempts that have been undertaken. | - The severe concerns underpinning the alleged crisis have several dimensions relating to: (\vskip 3pt- a) the (small) amount of published replication research; (\vskip 3pt- b) the (poor) quality of replication research;\vskip 3pt- and (c) the (lack of) reproducibility, which refers to the extent to which findings can (not) be reproduced in replication attempts that have been undertaken. |

The results in Table 4 shows that CoreNLP and Spacy make the most errors of the 2nd type (in recognizing tokens). Because of this, the share of correct predictions and other indicators were not maximum. It also confirms that Sentencize.jl performs segmentation with the fewest errors (all tokens were recognized correctly).

## Conclusion

In this paper, we compared the results of the 8 segmentation tools using comparison metrics and calculating performance. It is worth noting that the best results in terms of performance (WordTokenizers.jl) and quality (Sentencize.jl) belong to Julia tools. The benchmark source code is available at https://bmstu.codes/AnnaZav/sentencesplitterbenchmark.

The novelty from the technical side is developed by our unified benchmarking framework for libraries written in different programming languages which allows connecting the new libraries into a testing pipeline and getting comparison results on both quality and execution performance. And, due to the selected Julia language, these libraries might be written in different languages, including native Julia, Python, R, Java, etc. That work confirms that Julia might be used as an integration platform.

This paper is a part of the research work carried out within the Bauman Deep Analytics project of the Priority 2030 program.

## References

1. Text to sentence splitter. https://github.com/mediacloud/sentence-splitter, 2019. Accessed: 2022-01-20.
2. Apache. Opennlp. http://opennlp.apache.org, 2010. Accessed: 2022-01-20.
3. *Bird, S., Klein, E., and Loper, E.* Natural language processing with Python: analyzing text with the natural language toolkit. " O'Reilly Media, Inc.", 2009.
4. *Bolshakova, E.I., Peskova, O., Klyshinsky, E., Noskov, A.A., Lande, D., and Yagunova, E.V.* Automatic natural language processing and computational linguistics, 2015.
5. *Chen, J., and Revels, J.* Robust benchmarking in noisy environments. arXiv e-prints (Aug 2016).
6. *Community, T.J.* Calling c and fortran code, may 2022.
7. *Community, T.J.* Why we use julia, 10 years later, february 2022.
8. *Honnibal, M., and Johnson, M.* An improved non-monotonic transition system for dependency parsing. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (Lisbon, Portugal, Sept. 2015), Association for Computational Linguistics, pp. 1373–1378.
9. *Honnibal, M., and Montani, I.* spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. 2017.
10. *Koehn, P., et al.* Europarl: A parallel corpus for statistical machine translation. In MT summit (2005), vol. 5, Citeseer, pp. 79–86.
11. *Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., and McClosky, D.* The stanford corenlp natural language processing toolkit. In Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations (2014), pp. 55–60.
12. *Nivre, J., and Nilsson, J.* Pseudo-projective dependency parsing. In Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05) (Ann Arbor, Michigan, June 2005), Association for Computational Linguistics, pp. 99–106.
13. *Ruopp, A.* Lingua sentence. https://metacpan.org/pod/Lingua::Sentence, 2010. Accessed: 2022-01-20.
14. *Sætre, R., Søvik, H., Amble, T., and Tsuruoka, Y.* Genetuc, genia and google: Natural language understanding in molecular biology literature. In Transactions on Computational Systems Biology V (Berlin, Heidelberg, 2006), C. Priami, X. Hu, Y. Pan, and T. Y. Lin, Eds., Springer Berlin Heidelberg, pp. 68–82.
15. *Soricut, R., and Marcu, D.* Sentence level discourse parsing using syntactic and lexical information. In Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (2003), pp. 228–235.
16. *Zeldes, A.* The GUM corpus: Creating multilayer resources in the classroom. Language Resources and Evaluation 51, 3 (2017), 581–612.

**A.P.Zavyalova.** Master student. Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow, 105005, Russia. E-mail: annazav13@gmail.com

**P.A.Martynyuk.** Master student. Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow, 105005, Russia. E-mail: martapauline@yandex.ru

**R.S.Samarev.** Associate Professor. Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow, 105005, Russia. E-mail: samarev@acm.org