Separating Identification and Quantification of Linguistic Structures to Enable Reusable Resources

Torsten Zesch[1], Jeanette Bewersdorff[1], Marie Bexte[1], Josef Ruppenhofer[1], Yuning Ding[2,3], Julian F. Lohmann[3], Andrea Horbach[2,3],
[1]CATALPA – Center of Advanced Technology for Assisted Learning and Predictive Analytics, FernUniversität in Hagen, Germany
[2] Kiel University, Germany
[3] Leibniz Institute for Science and Mathematics Education, Kiel, Germany
Correspondence: torsten.zesch@fernuni-hagen.de

Abstract

Detecting linguistic structures in textual data is a prerequisite for computational linguistics research and specialized applications, for example in education. Existing libraries have shortcomings with respect to reusability, multilinguality, and robustness. We argue that a core reason is the missing separation of detection (where in the text?) from quantification (how often?). We identify a set of design principles that linguistic structure extraction should follow and review existing libraries accordingly. The identified gap is filled with a fully open-source and open-resource prototypical implementation called LIFT.

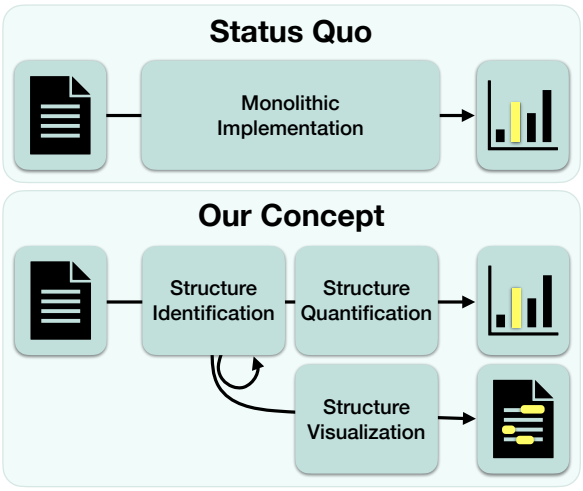


Figure 1: Separating identification and quantification of linguistic structures

1 Introduction

Detecting and quantifying linguistic structures in textual data is a prerequisite for many flavors of computational linguistics research in general, but also for a variety of applications, most prominently in the educational domain, with tasks such as student feedback on errors (Ng et al., 2014), CEFR level classification (Pilán and Volodina, 2018; Kerz et al., 2021), essay scoring (Uto et al., 2020), lexical sophistication classification (Kyle et al., 2018), or coherence and cohesion measurement (Crossley and McNamara, 2011; Ding et al., 2024). However, linguistic structures also play an important role in building explainable systems in other domains such as fake news detection (Pérez-Rosas et al., 2018), authorship attribution (Stamatatos, 2009), or dementia detection (Peled-Cohen and Reichart, 2025). Even in the era of LLMs, linguistic structures remain important. As has been pointed out by Opitz et al. (2024), NLP relies on linguistics in many ways, for example, one important way to judge LLM capabilities is to measure how well they capture linguistic structures.

So as we are convinced that linguistic structures are still relevant, our focus lies in enabling practical *usability* (in the form of a concrete implementation) and to increase *reusability* by separating the **detection** of linguistic structures from their **quantification**, which are often called *features*. This distinction can be seen in Figure 1.

As our (deliberately simple) running example, consider a method for finding the number of finite verbs per sentence. One might be tempted to quickly implement it without separating identification from quantification, e.g. by iterating over all tokens and counting how often we see a POS tag from the set {VBD,VBP,VBZ}. This implicitly assumes that: (i) we are dealing with English and (ii) the underlying text is POS-tagged in PTB-style. Such an implementation impedes, among other aspects, reusability and a multi-lingual design. We argue that a better design principle is to separate identification and quantification as described in the next section. Taking secondary design principles outlined later in the paper into account as well leads us to a prototypical architecture for extracting linguistic structures from text in (almost) arbitrary languages in a structured and reusable way. As we

will show in a comparison with existing libraries this fills a gap in the landscape.

2 Separating Identification and Quantification

Our main argument is that a library for detecting linguistic structures should separate (as much as possible) *identification* of structures from their *quantification* in the form of features.

Core 1: Separation of Concerns

Identification and quantification should be separated.

We define *linguistic structures* as any concepts that are realized in the text, while *features* refer to the quantification of these structures. Following this distinction, the first step in our 'finite verbs' running example would be to find all finite verbs and anchor them in the text. The second step would be quantification, e.g. counting the number of finite verbs in each sentence and averaging this over all sentences in a given text. Once a linguistic structure is detected and anchored it may be reused, as finite verbs play a role in detecting other linguistic structures like argument structure (subject/object). Separating structures and features also helps following the DRY (Don't repeat yourself) principle (Hunt and Thomas, 2000), because every classical feature extractor has to re-implement some kind of counting logic, while this should be generalized in quantification code which is being reused across structures. For example, counting the number of finite verbs and counting the number of tokens should use the same code for counting, i.e. code that receives as parameter a certain structure type and then counts how often it occurs.

The issue of not separating identification and quantification is not just a theoretical possibility, but happens in practice. For example, the Flesch reading ease measure (Flesch, 1948) internally relies on syllables (a relevant linguistic structure). Most implementations violate the identification/quantification distinction, i.e. they internally count syllables, but from the outside there is no way to retrieve those syllables. This makes these implementations hard to transfer to a different language and less useful, as the syllable knowledge cannot be reused for other purposes.

However, there are also counterexamples where the separation might not be practical. Consider e.g. a lexical diversity measure like type-token-ratio (TTR) that is defined as the number of unique tokens divided by the number of all tokens in a text. It is debatable whether 'unique tokens' is a linguistic structure and where it should be anchored in the text. We argue that deduplication / set conversion is an operation on the quantification layer.

Span Granularity Being able to localize structures like spelling errors or syllables requires span identification with (at least) character-level granularity, while some libraries only output token offsets.

Core 2: Span Granularity

A linguistic structure should be identified by offsets into the source document, and the offset unit (e.g. Unicode code points, UTF-16 code units, or extended grapheme clusters) must be specified.

However, what users might perceive as characters, i.e. grapheme clusters rendered on a screen, is not always identical to what is represented as Unicode code points. For example, the sequence 'fi' may be encoded as two code points f (U+0066) followed by i (U+0069), while the font may render them as a single ligature glyph. Less commonly, the same visual result may be encoded as the single compatibility code point U+FB01. In that case, it is impossible to address the f as a separate code point unless the text is first normalized

The situation is further complicated by the fact that many user-perceived characters, e.g. letters with combining marks, emoji with skin-tone modifiers, or zero-width joiner emoji sequences, are multi-code-point grapheme clusters. Character-offset spans defined over code points may therefore split what users perceive as a single character, while spans defined over grapheme clusters are not directly equivalent to code-point indexing.

The picture is complicated even more because different programming languages and APIs define 'string offsets' differently; some use Unicode code points (Python), many web APIs use UTF-16 code units, and others use UTF-8 byte offsets. Thus, the same numeric offset may refer to different positions unless the offset scheme is explicitly defined.

Identification vs. Preprocessing Structure identification is closely related to what is often called linguistic preprocessing, i.e. enriching the text with linguistic information. The exact dividing line is blurry, but we consider as pre-processing any step where detecting structures is canonically in-

tegrated in existing tools, such as SPACY[1] (Honnibal et al., 2020) or STANZA (Qi et al., 2020) for Python, CORENLP (Manning et al., 2014) or DKPROCORE (Eckart de Castilho and Gurevych, 2014)) for Java, and TIDYTEXT (Silge and Robinson, 2016) or QUANTEDA for R. For most languages, this preprocessing includes the annotation of basic linguistic structures like Token, Sentence, POS, or Dependency annotations.

2.1 Related Libraries

Most libraries only output quantifications in form of features, especially for measuring linguistic complexity, without annotating any corresponding structures. Examples include Coh-Metrics (McNamara et al., 2014), L2 Syntactic Complexity Analyzer - L2SCA (Lu, 2010) and its fork NEOSCA (Tan, 2024), and several tools from the Suite of Automatic Linguistic Analysis Tools[2] that only create features, e.g. TAALES (Kyle et al., 2018) for lexical sophistication or TAASSC (Crossley et al., 2017) for syntactic sophistication and complexity. A special form of linguistic complexity measures are readability formulas in implementations like TEXTSTAT [3] in Python. STYLO (Eder et al., 2016) provides stylometric analysis in R. ELFEN (Maurer, 2026) focuses on efficiency and outputs quantification for a wide range of features in multiple languages, but has no separation of structures.

There are other libraries that focus on creating structures e.g. as annotation-by-query like CSNIPER (Eckart de Castilho et al., 2012) or using automatic annotation functionality within tools like INCEPTION (Klie et al., 2018).

We are not aware of any libraries fully following through with the separation of identification and quantification. LFTK (Lee and Lee, 2023) only provides implementations for English and only a weak separation of structures and quantification, which would almost require a full re-implementation to support other languages. CTAP (Chen and Meurers, 2016) measures text complexity but also has quite weak separation.

UIMA-based (Ferrucci and Lally, 2003) libraries like CTAP, POLKE, and PALME internally all work with span prediction even if they might not explicitly expose those to users.

[1] https://spacy.io/

[2] https://www.linguisticanalysistools.org/

[3] https://textstat.org/

LLMs LLMs have been shown to have limited meta-linguistic knowledge (Beguvs et al., 2023) and limited abilities to identify more complex linguistic concepts in texts, like argument structure (Wilson et al., 2023). However, capabilities are rapidly evolving and their zero-shot and few-shot abilities make them interesting for the identification of structures. Most frameworks (including LLM-based extraction in general) do not explicitly predict spans of identified structures in the text, although there are works discussing such capabilities (Semin et al., 2026).

Coverage It is very hard to compare the scope of existing libraries. Many libraries list as one of their core properties the number of implemented features (i.e. a structure identification together with its quantification) with figures in the hundreds or even thousands. However, due to the multiplicative nature of the (structures × quantification) feature space, such numbers tend to be rather meaningless. For example,

```
average_number_of_[POS]_per_sentence
```

already yields 45 features for an English tagset and almost 300 for Chinese.[4] Adding another derivation

```
total_number_of_[POS]
```

doubles that number.

A more meaningful discussion of coverage should focus on languages, breadth and depth of structure identification, and flexibility of quantification.

3 Supporting Design Principles

While core principle 1 (separation of identification and quantification) and core principle 2 (span granularity) are central to enable fine-grained reusable extraction, we now identify and discuss supporting design principles that are downstream of those core design decisions or apply to research software in general.

3.1 Configuration Over Code

A key concept how the envisioned separation of identification and quantification can be facilitated is to rely on configuration as much as possible.

[4] Tagset size numbers according to (Petrov et al., 2012). Your tagset version might vary.

Supporting: Configuration Over Code

Wherever possible use configuration files instead of code.

A straightforward way to implement the check for POS tags encoding finite verbs in English would be:

```
is_finite = pos in {"VBD", "VBP", "VBZ"}
```

However, directly encoding this logic limits reusability and is hard to maintain. Alternatively, we may rely on configuration files that can be edited more easily. Another advantage is that multiple languages can be easily supported by providing a configuration file for each language.

For more complex structures, simple lists as in our finite verb example are insufficient. A possible solution are rule languages (or pattern-languages) like UIMA RUTA (Kluegl et al., 2014) or spaCy Matcher.[5] The corresponding scripts can be stored in configuration files that are loaded and executed by a generic rule execution component.

For example, POLKE[6] (Pedagogically Oriented Language Knowledge Extractor) (Chen and Sagirov, 2025) encodes the rules for annotating grammatical constructs according to the English Grammar Profile (O'Keeffe and Mark, 2017) in RUTA scripts (Kluegl et al., 2014) stored as plain text files. Löfflad et al. (2025) use the same design for a system called PALME making steps towards a German Grammar Profile.

Programming Language Most libraries are written in one specific programming language only, often Python, Java, or R, some older or efficiency-oriented libraries also in C. To our knowledge, there are no cross-programming-language libraries available and there is generally very little exchange across programming language boundaries. In contrast, accessing a configuration file from one programming language or another is simple, so relying on configuration over code decreases the dependence on a specific programming language.

3.2 Open Source & Open Resource

To be really reusable and to allow for replicable research a library should be open source and open resource, i.e. not only the source code should be open, but also the linguistic resources and models it relies on. This is especially important as with relying heavily on configuration, the focus is shifting from the 'glue code' to the resources.

Supporting: Open Science

A library should make both code and resources openly available.

An example for the distinction is READERBENCH (Dascalu et al., 2013) which is open source, but ships without resources and could not be made functional by us. Another example is the often used LIWC[7] that relies on a proprietary lexicon.

For libraries like the mostly Spanish UMUTEXTSTATS (García-Díaz et al., 2022) that are not open source, we are not even able to judge whether they might follow the design principles outlined in this paper.

3.3 Integrated Preprocessing

While it would be ideal to make preprocessing configurable, we consider the resulting increase in complexity unmanageable and the potential increase in performance marginal. We thus take the opinionated stance that:

Supporting: Integrated Preprocessing

Preprocessing should be integrated to reflect a tight coupling between core and derived linguistic structures.

A main reason is that identifying structures often depends on preprocessing assumptions, like in our running example that lists finite verb POS tags using the English PTB tagset that matches the preprocessing.

3.4 Multi-Linguality

As with many fields, most libraries are focused on English and not designed with multi-linguality in mind (e.g. hard-coded paths to English resources, implicit assumptions about linguistic phenomena, etc). Consequently, there are very few truly multilingual libraries.

Beyond English, other mono-lingual ones exist, e.g. PALME (Löfflad et al., 2025) for German or bi-lingual libraries like TRUNAJOD (Palma et al., 2020) for English and Spanish.

An exception are linguistic complexity or readability measures that often only use rather shallow linguistic knowledge and can thus be provided more easily for many languages. For example, TEXTSTAT provides a Python implementation of

[5] https://spacy.io/api/matcher
[6] https://github.com/chxiaobin/polke
[7] https://www.liwc.app

the Flesch reading ease measure (Flesch, 1948) for 7 languages (de, en, es, fr, it, nl, ru).

Supporting: Multi-Linguality

A library should support multiple languages and be designed with that in mind.

3.5 Test-Driven Development

In the context of research software for identifying and quantifying linguistic structures, test-driven development (TDD) ensures that each component of the system is rigorously specified and validated against expected linguistic data and outcomes.

Supporting: Test-Driven Development

Identification and quantification needs tests to ensure robustness and maintainability.

By defining test cases that capture both typical and edge-case linguistic phenomena, developers can iteratively refine algorithms and data processing routines, leading to more robust, reliable, and reproducible research results. TDD also facilitates easier maintenance and extension of the software as new linguistic features or structures are incorporated.

4 Prototypical Implementation

In the following, we present a prototypical implementation called LIFT (short for Linguistic Features in Text) that demonstrates how adhering to the design principles leads to a maintainable and reusable library.

4.1 Separating Identification and Quantification

LIFT comes with a number of structure annotators that showcase the design principles outlined above. Custom ones for structures that are not yet supported can easily be added.

Structure Identification In Figure 2 we give an example for an implementation of a component identifying 'easy words' as defined in a list of simple words that language learners should know. This is a complete example with the necessary imports in lines 1–5. Line 7 is the decorator declaring that this structure extractor (SE) only works for English. In the constructor, the SE declares where to find the list with the English easy words and calls the base classes (e.g. reusing code for reading the list, registering the languages, etc.), and declares an ad-hoc type for this structure (so that we can retrieve it later). The _process() method is called for every document and reads the easy words from the resource. Note that of course it would be more efficient to only load the list once in the constructor, but SEs that support more than one language decide which language (and consequently which list) to use for each document (from its meta-data). Caching of loaded lists and other efficiency optimization is delegated to the ListReader base class. Note that the entries in the easy words lists are lemmatized, thus the SE depends on preprocessing and that lemma annotations are available in the document. The structure extractor iterates over these lemmas and when the lemma string is in the list of easy words it creates an annotation to be stored in the text representation with the same span as the lemma, but marked as an 'easy word'.

Structure Quantification To quantify the annotated structures, LIFT can be used as a feature extractor. Most feature extractors are **generic** in that they count the number of occurrences of a certain structure, optionally normalized by the number of another structure, such as *tokens per sentence*. For example, a quantification like easy words per tokens would look like this:

```
class FE_EasyWordRatio(
    FEL_AnnotationRatio):
  def __init__(self):
    super().__init__(
      FEL_AnnotationCounter('EasyWord'),
      FEL_AnnotationCounter('Token')
    )
```

For our example of identifying the number of finite verbs per sentence, we can define a *ratio annotator* that compares the values of two *count annotators*:

```
class FE_FiniteVerbsPerSentence(
    FEL_AnnotationRatio):
  def __init__(self):
    super().__init__(
      FEL_AnnotationCounter('FiniteVerb'),
      FEL_AnnotationCounter('Sentence')
    )
```

The two custom feature extractors can then be applied to any text:

```
FE_FiniteVerbsPerSentence().extract(text)
FE_EasyWordRatio().extract(text)
```

Span Granularity LIFT internally represents structures in UIMA CAS format (Götz and Suhre, 2004), i.e. as stand-off annotations (Bird and Liberman, 2001) with character-level span offsets over immutable source texts. Note that features (i.e.

```python
from cassis import Cas
from anon_lib.decorators import supported_languages
from anon_lib.types import T_LEMMA
from anon_lib.annotators.api import SEL_BaseAnnotator, SEL_ListReader
from pathlib import Path

@supported_languages('en')
class SE_EasyWordAnnotator(SEL_BaseAnnotator, SEL_ListReader):
    def __init__(self, language, ts=None):
        filename = Path(__file__).parent / "resources" / "en" / "BNC_easy_words.txt"
        SEL_ListReader.__init__(self, filename)
        SEL_BaseAnnotator.__init__(self, language)

        self.EW = self.get_type("org.anon_lib.type.EasyWord")

    def _process(self, cas: Cas) -> bool:
        easy_words = self.read_list()
        for lemma in cas.select(T_LEMMA):
            l_str = lemma.value
            if l_str in easy_words:
                easy_word = self.EW(begin=lemma.get('begin'), end=lemma.get('end'))
                cas.add(easy_word)
        return True
```

Figure 2: Example code for identifying the concept 'easy word' in a text.

quantifications over structures) are eventually also represented as UIMA CAS annotations, but without offsets (as they belong to the whole document, but have no referent within a text).

Tool Integration For more complex cases, LIFT directly calls external libraries that take care of extraction. For example for time expressions we rely on HeidelTime (Strötgen and Gertz, 2015), for generic spellchecking on LanguageTool (Naber, 2003), and for real-word spelling errors on rwsechecker (Zesch et al., 2025). In our integration of these tools, the key parameters are exposed to maintain configurability for e.g. different languages:

```
SE_RWSE_Annotator(language="de",
    model_name="bert-base-multilingual-
    uncased", confusion_sets=
    confusion_sets)
```

If a tool does not return span offsets, LIFT provides a generic module that matches found structures to the text.

LLM Integration Using LLMs for finding linguistic structures is a special case of tool integration and supported by a generic component that handles LLM access and links results back to the text. The prompt and all parameters are provided via configuration files.

Visualization Examples A direct consequence of identifying structures as precise text spans is that they can easily be visualized for example as feedback or diagnostic tools for language learners and teachers. Figure 3 provides examples. Span visualization supports multiple structures at the same offsets (see Figure 3d for an example) as wells as overlapping structures. As a code example for visualizing finite verbs, we instantiate a *SpacySpanVisualizer* and customize it to display finite verbs in a specific style:

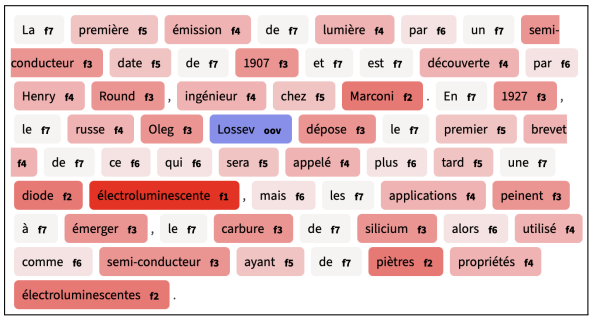
```
span_vis = SpacySpanVisualizer(ts)
span_vis.selected_span_type = highlight
span_vis.add_type(T_FINITE_VERB, label=
    "FiniteVerb", color="#F4C7C3")
html = span_vis.visualize(cas)
```

Apart from spans, it is also possible to visualize more complex annotation types such as dependencies (see example in Figure 3e).

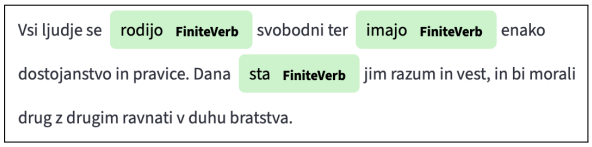
4.2 Configuration over Code

To showcase the versatility of configuration, we implement prototypical variants e.g. based on lists or dictionaries, our running example of the finite verb extractor being one example. Using the same principle we also include examples for modal verbs, discourse connectives, out-of-vocabulary tokens, easy words, word frequencies, etc.

For more complex structures, we also include examples of RUTA patterns, e.g. for finding nominalizations in German:

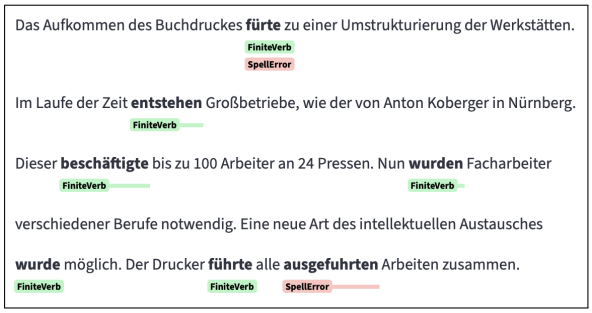


(a) French token frequency

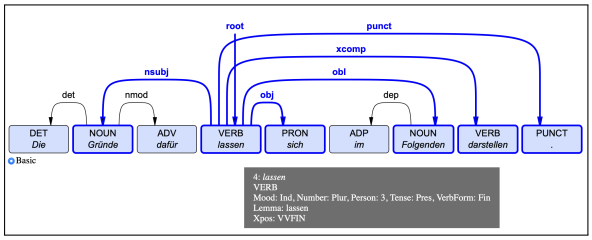


(b) Slovene finite verbs

(c) POS tags with input field and automatic language detection.



(d) German finite verbs and spelling errors in the same visualization.



(e) Visualization of German dependency parsing.

Figure 3: Examples of visualization outputs (a–d).

```
Lemma{REGEXP(Lemma.value, ".+(heit|keit|
    ment|mus|nis|schaft|tät|tion|tum|ung
    )$")
-> CREATE(Structure, \"name\"=\"
    Nominalization\" )};
```

Other (more complex) examples included in LIFT are scripts for subordinating clauses, passive constructions, or indirect speech.

Programming Languages A positive side effect of having most of the heavy lifting in the configuration is that one can rather easily switch to another implementation. With LIFT it is even possible to mix programming languages. For example, if a hard-to-replace module for finding some linguistic structure in a low-resource language is only available in Java, one can identify the structure in Java and export the corresponding document file (in UIMA CAS format). Then another Python-based pipeline can add more structures and compute the feature quantification.

4.3 Integrated Preprocessing

For preprocessing, we mainly rely on spaCy. LIFT takes care of mapping tagsets or converting between the spaCy representation and the UIMA-based representation used in LIFT.

Basic linguistic categories such as tokens, sentences, POS tags, lemmata, dependencies and such are covered by the linguistic preprocessing components. With our integration of spaCy preprocessing this is as easy as calling:

```
lang = detect_language(text)
spacy = Spacy_Preprocessor(language=lang)
cas = spacy.run(text)
```

The returned CAS is then ready for annotation with linguistic structures that build on the preprocessing. In terms of our running example pertaining to finite verbs, the text would now have the POS tags that are required to identify finite verbs.

For cases that require specific preprocessing tools not covered in the standard pipelines, it remains possible to provide one's own preprocessing results and feed them into an LIFT pipeline, but this may break compatibility with existing components. For example, a externally provided POS tagger might return a POS tagset that is incompatible with the list of finite verb POS tags as discussed above.

4.4 Multi-Linguality

LIFT is designed with a multi-lingual perspective. It does not make any assumption that English is

the default. Each document to be processed must declare its language scope (in the form of ISO 639 codes). However, we also provide a language identification process that can be used for automation.

Most features (counting, ratios, etc.) are implemented language-agnostic, but challenges arise when it comes to structures. Some structures are not as important in some languages (e.g. morphology in English vs. German). Some structures might not exist in a language, e.g. there is no clear finite/infinite distinction in Chinese verbs. Thus, in LIFT, a structure detector declares, in the form of decorators, for which languages it is defined.

As we rely on configuration, a multi-lingual structure detector relies on language-specific configuration files. For example in German, finite verbs are coded as VAFIN, VVFIN, or VMFIN in STTS tagset, while in English it is VBD, VBP, or VBZ in the PTB tagset.

4.5 Test-Driven Development

LIFT relies heavily on test-driven development, i.e. there is at least one unit test for each structure or feature extractor (usually multiple ones). These tests are also part of the documentation, providing examples of how to use the framework.

In a test, we try to cover edge cases ('what happens if a structure is not present in a text?'), document design decisions related to language ('is *aujourd'hui* counted as one token or two?' or 'is punctuation a token?').

4.6 Open-Source & Open-Resource

All the code in LIFT is made open-source under Apache license, version 2.[8] However, as we have explained above, we consider the code the less interesting part of the library, as we heavily rely on configuration files and language resources in LIFT. So it is very important that everything else is also open-resource, i.e. models, configuration files, rule sets, etc. are also openly distributed under CC BY 4.0 (unless the underlying resource imposes a different license).

Currently, LIFT is actively used in research projects at different universities and institutes. Growing the developer and user base is an important goal, but continued maintenance of research libraries remains an unsolved problem. This publication is part of the public outreach.

[8] https://github.com/zesch/linguistic-features-in-text

5 Conclusion

Starting from the observation that there is little reusability of existing frameworks for detecting linguistics structures in text, we argue that this is largely due to a missing separation between identifying where a structure occurs and quantifying how often it occurs. We therefore propose a set of core and supporting design principles, most importantly separation of concerns and character-level span anchoring, to make linguistic structure extraction reusable. We instantiate these principles in LIFT, a prototypical open-source and open-resource framework that represents identified structures as stand-off span annotations and builds most quantifications via generic, composable feature extractors. By emphasizing configuration over code and integrating preprocessing, LIFT aims to lower the cost of extending coverage to new languages and phenomena while keeping components maintainable and comparable. We expect that making structures explicitly retrievable, rather than only exposing derived numbers, will also improve transparency, enable richer visualizations, and support downstream applications such as educational feedback and explainable NLP. Future work will focus on broadening the inventory of structure annotators and resources, strengthening evaluation and testing across languages, and refining interfaces for tool and LLM integration while preserving the identification/quantification split.

Limitations

Separating identification and quantification comes with trade-offs in terms of computational efficiency. If one really only needs quantifications for a very large set of documents, an implementation disregarding this design principle might be faster.

Acknowledgments

This work was partially conducted at "CATALPA - Center of Advanced Technology for Assisted Learning and Predictive Analytics" of the FernUniversität in Hagen, Germany.

We thank Richard Eckart de Castilho for inspiration and support.

References

Gavsper Beguvs, Maksymilian Dkabkowski, and Ryan Rhodes. 2023. Large linguistic models: Investigating

LLMs' metalinguistic abilities. *IEEE Transactions on Artificial Intelligence*.

Steven Bird and Mark Liberman. 2001. A formal framework for linguistic annotation. *Speech Commun.*, 33(1–2):23–60.

Xiaobin Chen and Detmar Meurers. 2016. CTAP: A web-based tool supporting automatic complexity analysis. In *Proceedings of the Workshop on Computational Linguistics for Linguistic Complexity (CL4LC)*, pages 113–119, Osaka, Japan. The COLING 2016 Organizing Committee.

Xiaobin Chen and Nelly Sagirov. 2025. POLKE: A system for comprehensively annotating pedagogically-oriented grammatical structure use in language production.

Scott A. Crossley, Kristopher Kyle, and Danielle S. McNamara. 2017. Sentiment Analysis and Social Cognition Engine (SEANCE): An automatic tool for sentiment, social cognition, and social-order analysis. *Behavior Research Methods*, 49(3):803–821.

Scott A. Crossley and Danielle S. McNamara. 2011. Understanding expert ratings of essay quality: CohMetrix analyses of first and second language writing. *International Journal of Continuing Engineering Education and Life-Long Learning*, 21(2-3):170–191.

Mihai Dascalu, Philippe Dessus, Ştefan Trausan-Matu, Maryse Bianco, and Aurélie Nardy. 2013. Readerbench, an environment for analyzing text complexity and reading strategies. In *Artificial Intelligence in Education*, pages 379–388, Berlin, Heidelberg. Springer Berlin Heidelberg.

Yuning Ding, Omid Kashefi, Swapna Somasundaran, and Andrea Horbach. 2024. When argumentation meets cohesion: Enhancing automatic feedback in student writing. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 17513–17524, Torino, Italia. ELRA and ICCL.

Richard Eckart de Castilho, Sabine Bartsch, and Iryna Gurevych. 2012. CSNIPER - Annotation-by-query for Non-canonical Constructions in Large Corpora. In *Proceedings of the ACL 2012 System Demonstrations*, pages 85–90, Jeju Island, Korea. Association for Computational Linguistics.

Richard Eckart de Castilho and Iryna Gurevych. 2014. A broad-coverage collection of portable NLP components for building shareable analysis pipelines. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pages 1–11, Dublin, Ireland. Association for Computational Linguistics and Dublin City University.

Maciej Eder, Jan Rybicki, and Mike Kestemont. 2016. Stylometry with R: A Package for Computational Text Analysis. *The R Journal*, 8(1):107–121.

David Ferrucci and Adam Lally. 2003. Accelerating corporate research in the development, application and deployment of human language technologies. In *SEALTS '03: Proceedings of the HLT-NAACL 2003 workshop on Software engineering and architecture of language technology systems*, pages 67–74, Morristown, NJ, USA. Association for Computational Linguistics.

Rudolph Flesch. 1948. A new readability yardstick. *Journal of Applied Psychology*, 32(3):p221 – 233.

José Antonio García-Díaz, Pedro José Vivancos-Vicente, Ángela Almela, and Rafael Valencia-García. 2022. UMUTextStats: A linguistic feature extraction tool for Spanish. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 6035–6044, Marseille, France. European Language Resources Association.

T. Götz and O. Suhre. 2004. Design and implementation of the uima common analysis system. *IBM Syst. J.*, 43(3):476–489.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python.

Andrew Hunt and David Thomas. 2000. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., USA.

Elma Kerz, Daniel Wiechmann, Yu Qiao, Emma Tseng, and Marcus Ströbel. 2021. Automated Classification of Written Proficiency Levels on the CEFR-Scale through Complexity Contours and RNNs. In *Proceedings of the 16th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 199–209, Online. Association for Computational Linguistics.

Jan-Christoph Klie, Michael Bugert, Beto Boullosa, Richard Eckart de Castilho, and Iryna Gurevych. 2018. The INCEpTION Platform: Machine-Assisted and Knowledge-Oriented Interactive Annotation. In *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*, pages 5–9. Association for Computational Linguistics. Event Title: The 27th International Conference on Computational Linguistics (COLING 2018).

Peter Kluegl, Martin Toepfer, Philip-Daniel Beck, Georg Fette, and Frank Puppe. 2014. UIMA Ruta Workbench: Rule-based Text Annotation. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: System Demonstrations*, pages 29–33, Dublin, Ireland. Dublin City University and Association for Computational Linguistics.

Kristopher Kyle, Scott Crossley, and Cynthia Berger. 2018. The tool for the automatic analysis of lexical sophistication (TAALES): version 2.0. *Behavior Research Methods*, 50(3):1030–1046.

Bruce W. Lee and Jason Lee. 2023. LFTK: Handcrafted features in computational linguistics. In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pages 1–19, Toronto, Canada. Association for Computational Linguistics.

Denise Löfflad, Benedikt Beuttler, and Detmar Meurers. 2025. German Grammar Profile for Learners: Pedagogical Feature Definition and Automated Extraction. In *Proceedings of the 21st Conference on Natural Language Processing (KONVENS 2025): Workshops*, pages 212–223, Hannover, Germany. HsH Applied Academics.

Xiaofei Lu. 2010. Automatic Analysis of Syntactic Complexity in Second Language Writing. *International Journal of Corpus Linguistics*, 15(4):474–496.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

Maximilian Maurer. 2026. elfen: A Python Package for Efficient Linguistic Feature Extraction for Natural Language Datasets. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 61–74, Rabat, Marocco. Association for Computational Linguistics.

Danielle S. McNamara, Arthur C. Graesser, Philip M. McCarthy, and Zhiqiang Cai. 2014. *Automated Evaluation of Text and Discourse with Coh-Metrix*. Cambridge University Press.

Daniel Naber. 2003. A Rule-Based Style and Grammar Checker. Master's thesis, Bielefeld University. Catalog number: V108379.

Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. 2014. The CoNLL-2014 shared task on grammatical error correction. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–14, Baltimore, Maryland. Association for Computational Linguistics.

Anne O'Keeffe and Graham Mark. 2017. The English Grammar Profile of learner competence: Methodology and key findings. *International Journal of Corpus Linguistics*, 22(4):457–489.

Juri Opitz, Shira Wein, and Nathan Schneider. 2024. Natural language processing relies on linguistics. *ArXiv*, abs/2405.05966.

Diego Palma, Christian Soto, Mónica Veliz, Bernardo Riffo, and Antonio Gutiérrez. 2020. A Data-Driven Methodology to Assess Text Complexity Based on Syntactic and Semantic Measurements. In *Human Interaction and Emerging Technologies*, pages 509–515, Cham. Springer International Publishing.

Lotem Peled-Cohen and Roi Reichart. 2025. A Systematic Review of NLP for Dementia – Tasks, Datasets and Opportunities. *Preprint*, arXiv:2409.19737.

Verónica Pérez-Rosas, Bennett Kleinberg, Alexandra Lefevre, and Rada Mihalcea. 2018. Automatic Detection of Fake News. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3391–3401, Santa Fe, New Mexico, USA. Association for Computational Linguistics.

Slav Petrov, Dipanjan Das, and Ryan McDonald. 2012. A Universal Part-of-Speech Tagset. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 2089–2096, Istanbul, Turkey. European Language Resources Association (ELRA).

Ildikó Pilán and Elena Volodina. 2018. Investigating the Importance of Linguistic Complexity Features across Different Datasets Related to Language Learning. In *Proceedings of the Workshop on Linguistic Complexity and Natural Language Processing*, pages 49–58, Santa Fe, New-Mexico. 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*.

Danil Semin, Ondřej Dušek, and Zdeněk Kasner. 2026. Strategies for Span Labeling with Large Language Models. *Preprint*, arXiv:2601.16946.

Julia Silge and David Robinson. 2016. tidytext: Text Mining and Analysis Using Tidy Data Principles in R. *JOSS*, 1(3).

Efstathios Stamatatos. 2009. A Survey of Modern Authorship Attribution Methods. *Journal of the American Society for Information Science and Technology*, 60(3):538–556.

Jannik Strötgen and Michael Gertz. 2015. A Baseline Temporal Tagger for all Languages. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 541–547, Lisbon, Portugal. Association for Computational Linguistics.

Long Tan. 2024. Neosca: A fork of l2 syntactic complexity analyzer, version 0.1.4. `https://github.com/tanloong/neosca`.

Masaki Uto, Yikuan Xie, and Maomi Ueno. 2020. Neural Automated Essay Scoring Incorporating Handcrafted Features. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6077–6088, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Michael Wilson, Jackson Petty, and Robert Frank. 2023. How Abstract Is Linguistic Generalization in Large Language Models? Experiments with Argument

Structure. *Transactions of the Association for Computational Linguistics*, 11:1377–1395.

Torsten Zesch, Dominic Gardner, and Marie Bexte. 2025. Transformer-Based Real-Word Spelling Error Feedback with Configurable Confusion Sets. In *Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025)*, pages 375–383, Vienna, Austria. Association for Computational Linguistics.