ESTNLTK - NLP Toolkit for Estonian

Siim Orasmaa, Timo Petmanson, Alexander Tkachenko, Sven Laur, Heiki-Jaan Kaalep
Institute of Computer Science, University of Tartu
Liivi 2, 50409 Tartu, Estonia
siim.orasmaa@ut.ee,tpetmanson@gmail.com,aleksandr.tkatsenko@ut.ee,swen@ut.ee,heiki-jaan.kaalep@ut.ee

Abstract
Although there are many tools for natural language processing tasks in Estonian, these tools are very loosely interoperable, and it is not easy to build practical applications on top of them. In this paper, we introduce a new Python library for natural language processing in Estonian, which provides a unified programming interface for various NLP components. The ESTNLTK toolkit provides utilities for basic NLP tasks including tokenization, morphological analysis, lemmatisation and named entity recognition as well as offers more advanced features such as a clause segmentation, temporal expression extraction and normalization, verb chain detection, Estonian Wordnet integration and rule-based information extraction. Accompanied by a detailed API documentation and comprehensive tutorials, ESTNLTK is suitable for a wide range of audience. We believe ESTNLTK is mature enough to be used for developing NLP-backed systems both in industry and research. ESTNLTK is freely available under the GNU GPL version 2+ license, which is standard for academic software.

Keywords: natural language processing, Python, Estonian language

1. Introduction

Estonian scientific community has recently enjoyed an active period, when a number of major NLP components have been developed under free open source licenses. Unfortunately, these tools are very loosely interoperable. They use different programming languages, data formats and have specific hardware and software requirements. Such situation complicates their usage in both software development and educational setting. In this paper, we introduce ESTNLTK, a Python library for natural language processing in Estonian, which addresses this issue.

The ESTNLTK toolkit glues together existing software components and makes them easily accessible via a unified programming interface. It provides utilities for basic NLP tasks including tokenization, morphological analysis, lemmatisation and named entity recognition as well as offers more advanced features such as a clause segmentation, temporal expression extraction, verb chain detection, Estonian Wordnet integration and grammar-based information extraction.

The ESTNLTK toolkit is written in Python programming language and borrows design ideas from popular NLP toolkits TextBlob and NLTK. Users familiar with these tools can easily get started with ESTNLTK. Accompanied by a detailed API documentation and comprehensive tutorials, ESTNLTK is suitable for a wide range of audience. We believe ESTNLTK is mature enough to be used for developing NLP-backed systems both in industry and research. Additionally, ESTNLTK is a good environment for teaching NLP for students.

Although ESTNLTK is explicitly targeted for the Estonian language, the architecture is quite generic and the toolkit can be used as a template for building analogous toolkits for other languages. The only limiting factor is the availability of external NLP components that must be replaced.

The ESTNLTK toolkit is available under the GNU GPL version 2+ license from `https://github.com/estnltk/estnltk`. The library works on Linux, Windows and Mac OS X and supports Python versions 2.7 and 3.4.

2. Related Work

There is a great variety of available tools for natural language processing. Typically, they come in the form of reusable software libraries, which can be embedded into user applications. Toolkits like Stanford CoreNLP (Manning et al., 2014) and NTLK (Bird and Klein, 2009) provide all-in-one solution for the most common NLP tasks, such as tokenization, lemmatisation, part-of-speech tagging, named entity extraction, chunking, parsing and sentiment analysis. Others, like gensim (Řehůřek and Sojka, 2010), a topic modelling framework in Python, and Apache Lucene (Cutting et al., 2004), a Java library for document indexing and search, are designed for specific tasks. In contrast, projects GATE (Cunningham et al., 2011) and Apache UIMA (Apache, 2010), represent a comprehensive family of tools for text analytics. In addition to software modules, they provide tools to manage complex text processing workflows, annotate corpora and support large-scale distributed computing.

The design of ESTNLTK is strongly influenced by TextBlob (Loria, 2014), a Python library that is built on top of the NLTK toolkit. It provides a simplified API for common NLP tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, classification and translation. The central class throughout the framework is `Text`, which encodes information about natural language text. Being passed through a text-processing pipeline, a `Text` instance accumulates information provided by each processing task. This approach differs from the typical pipeline architecture, where each processing layer modifies specially formatted input text.

Our choice to implement ESTNLTK as a Python library similar to TextBlob was motivated by the following reasons. First, Python is widely adopted in NLP community, due to its powerful text processing capabilities and good

support for NLP and machine learning. Second, Python has a good support for external libraries written in other compiled languages such as C or C++. This greatly simplifies integration of existing tools. Furthermore, the most performance-critical code sections can be moved to C/C++ extension modules. Thirdly, Python is a popular programming language for teaching entry-level computer science courses in many universities including Estonian. Hence, no extra skills are needed to use ESTNLTK.

Initially, we also considered Java as a programming language, since many tools provide libraries written in Java. As a compiled, statically-typed, object-oriented language, Java is well-suited for large software projects. However, it also has a verbose inflexible syntax, what makes it highly unproductive for prototyping and experimentation and it cannot be used for scripting in interactive environments.

As an alternative to building ESTNLTK from scratch, we also considered customising an existing toolkit such as NLTK or Textblob for the Estonian language. This would provide a benefit of using a common interface for text analysis in both Estonian and English. However, we found it difficult to achieve, since neither NLTK nor Textblob are designed to internally handle the morphological ambiguity and attributes such as cases, forms and clitics found in Estonian. Ignoring this information would mean incomplete representation of Estonian morphology. On the other hand, modifying existing framework internals would require significant rewrites and create compatibility issues. Given the benefits and tradeoffs, we decided to implement ESTNLTK as a specialised package for Estonian. That said, we do not rule our integrating ESTNLTK with other toolkits in the future.

3. Design Principles

The ESTNLTK toolkit exposes its NLP utilities through a single wrapper class `Text`. To perform a text-processing operation, the user needs to access the corresponding property of an initialised `Text` object. After that ESTNLTK will carry out all necessary pre-processing behind the scenes. Hence, an analysis pipeline can be specified dynamically through an interactive scripting session without thinking of implementation details.

For example, named entities can be accessed via the property `named_entities`. To extract named entities, the ESTNLTK toolkit will complete five separate operations in succession: (1) segment paragraphs; (2) segment sentences; (3) tokenise words; (4) perform morphological analysis; (5) identify named entities. For each operation in the pipeline, ESTNLTK comes with a sane default implementation. However, a user can provide an alternative implementation through the constructor of the class `Text`.

Text class. The `Text` is a subclass of a standard Python dictionary with additional methods and properties designed for NLP operations. A new instance can be created simply by passing the plaintext string as an argument to the constructor, which initiates a dictionary with a single attribute `text`. Using a dictionary as a base data format has several of advantages: it is simple to inspect and debug, extendible, can store meta-data and can be serialized to a JSON format.

Annotation layers. The outcome of most NLP operations can be seen as different annotation layers in the original text. ESTNLTK stores each layer as a list of non-overlapping regions, defined by start and end positions. The lists are stored as dictionary elements identified by unique layer names.

There are two types of annotations. A *simple* annotation has only a single start-end position pair and is used to denote sentences, words, named entities and other annotations made up of a single continuous area. *Multi-region* annotations can have several start-end position pairs. They are used to denote clauses and verb chains, which in Estonian can allocate several non-adjacent regions in a sentence.

Each annotation is a dictionary, which in addition to compulsory `start` and `end` attributes can store any kind of relevant information. For example, named entity layer annotations have `label` attribute denoting whether the annotation marks a location, organisation or a person. Words layer annotations store a list of morphological analysis variants, where each one is a dictionary containing a lemma, form, part-of-speech tag and other morphological attributes of Estonian words.

Users can create custom annotation layers simply by defining new `Text` dictionary elements. Of course, using reserved layer names used by ESTNLTK is not allowed. Users can also extend existing layers and add new attributes as long as the attribute names are unique.

Dependency management. The ESTNLTK NLP tools require specific preconditions to be satisfied before the desired operations can be performed. These dependencies are depicted in Figure 1. Whether a dependency is satisfied or not, can be answered by looking directly at the dictionary contents of the `Text` class. This is exactly, what the code does, when the user requests a certain resource. In case the resource is not available, the code will execute the operation that can provide the resource. This operation in turn checks if all of its dependencies are available and recursively executes necessary operations to provide the missing resources.

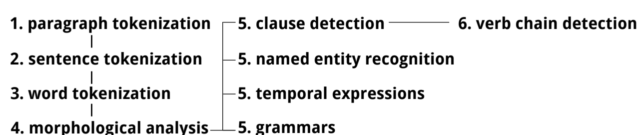

Figure 1: Text-processing utilities in ESTNLTK. Numbers and lines denote the order and dependencies between the operations.

In case of a newly initiated `Text` instance, requesting any non-trivial resource executes large portions of the pipeline. However, consequent calls to the same resource only require few dictionary lookups later.

Overriding standard operations. ESTNLTK comes with reasonable default behaviour for all built-in operations, but it may be desirable to override this functionality. For example, the user might require a custom sentence tokenizer or an optimised named entity recogniser for some task. How a specific component can be replaced, is discussed more thoroughly in Section 4. In simple cases, it is

sufficient to provide replacement components as keyword arguments to the `Text` constructor. However, miscellaneous use cases may benefit from subclassing the `Text` class and develop the custom behaviour directly into it.

Text segmentation. It is often better to process texts in smaller chunks if smaller pieces represent the problem we are solving better or the text is just too big to process as a whole. This can be achieved with the `split_by` method. The method takes in a name of an annotation layer and creates a list of `Text` instances based on the regions defined in the annotations. The number of the resulting texts is equal to the total number regions in the original annotation layer. As multi-layer annotations can define more than a single region, this number can be greater than the number of annotations.

All annotation layers are preserved in this process, but the annotations themselves are divided between the resulting `Text` instances. As annotations may define one or more regions, they can end up in more than a single piece. In such cases, only the regions belonging to the piece will appear in the annotation, although other attributes are preserved.

Comparison with alternatives. Note that layers can be embedded directly to the textual output format. This has been the traditional way for Estonian NLP tools. Although it makes it easy to write shell programs for analysis workflows, it is brittle and restricting. A small change in an output format may cause subtle errors and parsing routines are needed to do unexpected analysis steps. The second alternative is to store the layer information in a separate index object that is shared by many documents. This can significantly speed up certain search operations, as linear scan over all documents can be replaced with simple index lookup. However, the right structure of the index object depends on a particular task and is wasteful for online processing of documents. Hence, ESTNLTK uses this alternative only for handling large document collections where the creation index objects justified and the potential list of search terms are known upfront.

4. Standard NLP Tasks

In this section, we will discuss the rationale and the design tradeoffs of each standard NLP task. The ESTNLTK standard tasks are paragraph, sentence and word tokenization, morphological analysis, clause detection, named entity recognition and temporal time expression detection. These tools depend on each other and form a dependency graph, which can be seen in Figure 1.

Tokenization. Text tokenization tasks are the most basic steps of any NLP pipeline. Paragraph tokenization is useful when texts are longer and contain more than a single paragraph, such as news articles. By default, ESTNLTK assumes the paragraphs are separated by a single empty line (two newline characters). Sentence tokenization uses a pre-trained NLTK `punkt` tokenizer for Estonian, which is trained on a corpus of news articles. For word tokenization, we use a modified NLTK `WordPunctTokenizer` that makes a tradeoff between traditional word tokenization practices and compatibility with other NLP tools.

Note that word tokenization depends on sentence tokenization, which in turn depends on paragraph tokenization. Although any of the tokenizers can be actually executed on raw plain text, they might not be consistent. Thus, the `Text` class enforces consistency by performing sentence tokenization on each individual paragraph and word tokenization on each individual sentence.

To customise tokenisation, one needs to implement an interface defined by NLTK's `StringTokenizer` class and pass the tokeniser as `paragraph_tokenizer`, `sentence_tokenizer` or `word_tokenizer` argument to the constructor of the `Text` class.

Morphological analysis. Morphological analysis is a core part of any text-processing pipeline, as it serves needs of many higher level tasks. ESTNLTK provides a wrapper API built on top of the VABAMORF, a C++ library for morphological analysis for Estonian (Kaalep and Vaino, 2001). Full text analysis can be performed using the function `tag_analysis` of the class `Text`. It identifies word lemmas, suffixes, endings, parts of speech, forms (e.g. the case of a noun, the tense and the voice of a verb). These bits of information can be accessed via the corresponding properties of the class `Text`. When a word has multiple morphological interpretations, VABAMORF will try to perform disambiguation. If disambiguation fails, multiple analysis variants will be reported.

Morphological synthesis and spell checking. To synthesise a particular form of a word, ESTNLTK provides a function `synthesize`. Given a word and some criteria, it generates all possible inflections that satisfy these criteria. The spell-corrector allows to identify misspelled words and provides suggestions for correction through the properties `spelling` and `spelling_suggestions` of the class `Text`. As the synthesis and spell check functions are built on top of VABAMORF, they inherit strengths and shortcomings of VABAMORF counterpart functions.

Named entity recognition. Named entity recognition (NER) is often used as an important subtask in information retrieval, opinion mining and semantic indexing. The class `Text` provides default algorithms for recognising persons, organisations and locations. These algorithms can be invoked by calling `tag_named_entities`. The call will trigger the whole text-processing pipeline, including tokenization, morphological analysis if these analyses have not been performed before. The first invocation of `tag_named_entities` will additionally load statistical models and store them in the global scope for future use.

NER algorithms in ESTNLTK are reimplementations of the methods described by Tkachenko et al (Tkachenko et al., 2013). The main difference lies in the implementation details. The original NER tool uses conditional random fields training algorithm (Lafferty et al., 2001) implemented in a Java package MALLET (McCallum, 2002), whereas the algorithms in ESTNLTK use the C++ CRFSUITE library (Okazaki, 2007). Both algorithms achieve reasonable accuracy for Estonian text. The C++ version just provides a better integration with Python.

Extracted named entities, their categories and locations in the text can be accessed using proper-

ties named_entites, named_entity_labels and named_entity_spans. The same information is also accessible through the layers named_entities and words. The layer named_entities stores information on entity categories and their positions in text. The layer word carries individual token labels in BIO format (Tjong Kim Sang and De Meulder, 2003).

The default models have been pre-trained to recognise a fixed set of entity-types in generic news articles. It is also possible to customise NER algorithms by providing a new training corpus consisting of annotated sentences. For this purpose, the class estnltk.ner.NerTrainer implements necessary feature extraction logic and provides a training interface. To train a new CRFSUITE model, a user must provide a training corpus and a custom configuration module, which lists feature extractors and feature templates used for detecting named entities. The resulting model can be used with a class estnltk.ner.NerTagger to extract entities from text.

Temporal expression tagging. In many information extraction applications, recognition and normalisation of time-referring expressions (*timexes*) can significantly improve accuracy. We can essentially discard sentences that describe events from irrelevant time-periods. Sometimes, the exact time of an event is the desired information.

The ESTNLTK package integrates AJAVT, a rule-based temporal expression tagger for Estonian (Orasmaa, 2012). The tagger recognises timexes and normalises semantics of these expressions into a standard format based on TimeML's TIMEX3 (Pustejovsky et al., 2003). Results of the tagging can be accessed via Text object's property timexes.

By default, expressions with relative semantics (such as *eile* 'yesterday' or *reedel* 'on Friday') are resolved with respect to the execution time of the program. This default can also be overridden by providing creation_date argument on initialisation of the Text object, after which relative expressions are resolved with respect to the argument date.

Although the current version of the tagger is tuned for processing news texts, our evaluation has shown that the current configuration of rules obtains high accuracy levels also on other types of formal written language texts, such as on law texts, and on parliament transcripts.

Details on the implementation of the tagger and its evaluation are provided in (Orasmaa, 2012).

Clause boundary detection. Clause boundary detection can be used to split long and complex sentences into smaller segments (*clauses*). As such, it can be useful in machine learning experiments offering a linguistically motivated "window" for feature extraction. In linguistic analysis, clause boundary detection offers a context form which phrase boundaries or word collocations can be detected.

The clause tagging in ESTNLTK is a re-implementation of the clause detector module introduced by Kaalep and Muischnek (Kaalep and Muischnek, 2012). It recognises consecutively ordered clauses (e.g. *[Ta istus nurgalauda ja] [tellis kohvi.]* '[She sat on the corner table and] [ordered a coffee.]') and clauses embedded within other clauses (e.g. *[Mees[, keda seal nägime,] lahkus kiirustades.]* '[The man [who we saw there] left in a hurry.]').[1] Calling the property clause_texts performs clause tagging along with all the dependent computations such as paragraph, sentence, word tokenization and morphological analysis. Clauses are stored as multi-region annotation in the clauses layer.

Because commas are important clause delimiters in Estonian, the quality of the clause segmentation may suffer due to missing commas. We have improved the original clause boundary detection algorithm, adding a special mode in which the program is more robust to missing commas.

In this mode, we apply the regular segmentation rules augmented with the following general heuristic: if a conjunction word (such as *et* 'that, for', *sest* 'because', *kuid* 'although', *millal* 'when', *kus* 'where', *kuni* 'as long as') is in the context where it is preceded and followed by a verbal centre of a clause, but it is not preceded by a comma (although it should be, according to Estonian orthographic conventions), we mark a clause boundary at the location of missing comma. The heuristic has a number of exceptions, which account for ambiguous usages of the specific conjunction words.

For example, *kuni* also means 'to', and so we have to exclude its usage in range-denoting phrases (such as in *5 kuni 7 kilomeetrit* '5 to 7 kilometres') from being marked as a clause boundary. We have also written exceptions to exclude frequently used verbal idiomatic expressions, such as *vaat et* 'almost, nearly', lit. 'look that (until)', from being detected as clause boundaries.

We believe that this new mode can be useful for improving clause segmentation quality on non-standard language (such as the Internet language), where comma usage often does not follow the orthographic rules.

NLP task	Affected layers
tokenize_paragraphs()	paragraphs
tokenize_sentences()	sentences
tokenize_words()	words
tag_analysis()	words
tag_clauses()	words, clauses
tag_named_entities()	words, named_entities
tag_timexes()	timexes
tag_verb_chains()	verb_chains

Table 1: Annotation layers affected by ESTNLTK's standard NLP tasks.

5. Additional Analysis Tools

Besides standard NLP components ESTNLTK contains additional tools which are less commonly used or are less mature compared to the standard components.

WordNet. The ESTNLTK package comes together with the Estonian Wordnet developed under the EuroWordNet project (Ellman, 2003). Provided interface is mostly NLTK-compatible and enables to query for synsets, relationships and compute similarities between the synsets. To

[1]Clause boundaries in the examples above are indicated by square brackets surrounding the clauses.

```
>> text = Text('Londoni lend, mis pidi täna hommikul kell 4:30 Tallinna saabuma,
   hilineb mootori starteri rikke tõttu ning peaks Tallinna jõudma ööl vastu
   homset kell 02:20.')
>>> text.named_entities
[u'London', u'Tallinn', u'Tallinn']
>>> text.clause_texts
[u'Londoni lend hilineb mootori starteri rikke tõttu ning',
 u', mis pidi täna hommikul kell 4:30 Tallinna saabuma,',
 u'peaks Tallinna jõudma ööl vastu homset kell 02:20.']
>>> text.timex_values
[u'2016-02-28T04:30', u'2016-02-29T02:20']
>>> text.verb_chain_texts
[u'hilineb', u'pidi saabuma', u'peaks jõudma']
```

Figure 2: Example of standard NLP tasks in Estnltk applied to a sentence "Londoni lend, mis pidi täna hommikul kell 4:30 Tallinna saabuma, hilineb mootori starteri rikke tõttu ning peaks Tallinna jõudma ööl vastu homset kell 02:20." (eng. "The flight from London, which was scheduled to land to Tallinn today morning at 4:30, is late due to an engine starter malfunction and is about to arrive to Tallinn tomorrow night at 02:00."). The example has been run on date 2016-02-28 (so timex values have been calculated with respect to that date).

access EuroWordNet database files, we use a Python module Eurown (Kahusk, 2010).

Verb chain detection. Estonian language is rich in verb chain constructions where the meaning of the content verb is significantly altered by other parts of the chain. Hence, proper detection and semantic normalisation of such constructions is essential in many practical applications. It allows us to detect negated clauses in sentiment analysis, to distinguish actual events from events marked with uncertainty in event factuality analysis, and to handle verb chains as a single translation unit in machine translation experiments.

The corresponding module in ESTNLTK addresses single-word main verbs and the following verb chain constructions: regular grammatical constructions (negation and compound tenses), catenative verb constructions (such as modal verb + infinite verb, e.g. *Ta **võib** meid homme **külastada*** 'He **might visit** us tomorrow', and finite verb + infinite verb constructions in general, e.g. *Ta **unustas** meid **külastada*** 'He **forgot to visit** us'), and verb+nominal constructions which subcategorise for infinite verbs (e.g. *Ta **otsis võimalust** meid **külastada*** 'He **sought for opportunity to visit** us.'). Detected verb chains are available via property `verb_chains` of the `Text` object. Grammatical features of the finite verb (polarity, tense, mood and voice) are provided as attributes of a verb chain object.

The module has been implemented in a rule-based manner. It firstly uses morphological information (information about finite verbs) and clause boundary information to detect grammatical main verb constructions, and then relies on subcategorisation information listed in lexicons to further extend the grammatical main verbs into catenative, and/or verb+nominal+infinite verb constructions. Subcategorisation lexicons were derived by semi-automatic analysis of the Balanced Corpus of Estonian[2].

[2] `http://www.cl.ut.ee/korpused/grammatikakorpus/index.php`

Word2Vec models. Recent advances in machine learning provide high dimensional word embeddings which capture distributed semantics of words and phrases. These have been successfully applied in machine translation and synonym detection. Thus, we have included experimental support for word2vec models (Mikolov et al., 2013), trained on the Estonian Reference Corpus[3]. The corpus consists of 16M sentences, 55M words and 3M types. We provide separate models trained on the original or on the lemmatised version of the corpus. For training, we used word2vec software[4]. Resulting models can be used with the word2vec command line tools or a Python library gensim[5].

Efficient document storage. Many modern text analysis methods require a large collection of documents as an input to be bootstrapped. This process is usually very resource intensive. In most cases, the computational complexity can be significantly reduced by providing an efficient search interface. Most storage solutions provide only efficient keyword search for text whereas operations in ESTNLTK require efficient search over layers. Therefore, we have included a `Database` class that uses NoSQL database Elastic Search[6] to store and retrieve `Text` objects.

Visualisation tools. Many NLP applications produce text annotations. Visualisation of these annotations is often the best way convey the analysis results to the end user. The `PrettyPrinter` class can be used to highlight text fragments according to annotations. The class generates valid HTML markup together with CSS files that can be used for developing Web front ends.

There are eight aesthetics that can be used for visualisation: text colour, text background, font, font weight, normal/italics, normal/underline, text size, character tracking. The most simple use case involves mapping a layer to an

[3] `http://www.cl.ut.ee/korpused/segakorpus/`
[4] `https://code.google.com/p/word2vec/`
[5] `https://radimrehurek.com/gensim/`
[6] `https://www.elastic.co/products/elasticsearch`

aesthetic and then rendering it as HTML and CSS. As a result, all regions in the layer will obtain the same markup, for instance have a green background. It is also possible to decorate regions according to annotations, e.g., colour verbs green and nouns red. For that one must specify mapping between annotation values and aesthetic values.

Grammar extractor. In many information retrieval and text classification tasks one must first recognise text fragments with a certain structure, such as measurements in medical text or trigger phrases in sentiment mining. Grammar extractor module allows to specify such phrases in term of layers using finite grammar.

6. Performance

For practical uses, it is important to have idea of the performance characteristics of individual components of the framework. Thus, we benchmarked ESTNLTK using a sample of 500 news articles picked randomly from the Estonian Reference Corpus[7]. The sample contains 167,180 tokens. We run benchmarks using Python3.4 on a Linux Mint desktop with the Intel Core i5 processor and 4GB of RAM. Table 2 illustrates the results. Note that the absolute values should be taken with a grain of salt, since they largely depend on a particular benchmark environment.

NLP task	tokens/second
Tokenization	97,095
Morphological analysis	3,913
NER	2,550
Temporal expression tagging	1,905
Clause boundary detection	5,271
Verb chain detection	11,658

Table 2: Performance of ESTNLTK core components. The reported performance relates to the individual component alone and does not account for execution time of its dependencies.

7. Source Code

ESTNLTK is implemented in Python and supports language versions 2.7 and 3.4. Under the hood, ESTNLTK uses a number of external components:

vabamorf, a C++ library for morphological analysis, disambiguation and synthesis, licensed under LGPL.
`https://github.com/Filosoft/vabamorf`

python-crfsuite, a python binding to CRFSUITE C++ library. python-crfsuite is licensed under MIT license, CRFSUITE library is licensed under BSD license.

osalausestaja, a Java-based clause segmenter for Estonian, licensed under GPL version 2.
`https://github.com/soras/osalausestaja`

Ajavt, a Java-based temporal expression tagger for Estonian, licensed under GPL version 2.
`https://github.com/soras/Ajavt`

[7] `http://www.cl.ut.ee/korpused/segakorpus/index.php`

The source code is hosted online on `github.com`. ESTNLTK can be installed directly from GitHub or, alternatively from the Python Package Index (PyPi) using command `pip install estnltk`.
ESTNLTK source code is licensed under a GNU GPL version 2+. It permits to use ESTNLTK to build proprietary software requiring the entire source code of the derived work to be made available to end users.

8. Conclusions

We presented ESTNLTK, a Python library which aims to organise Estonian NLP resources into a single framework. Although ESTNLTK is right in its infancy, it has already seen a number of uses, including online media monitoring, digital book indexing and client complaint categorisation. Recently, ESTNLTK for the first time has been used in teaching a NLP course in Estonian at the University of Tartu. We believe that in the future ESTNLTK will find even more applications.

9. Acknowledgements

ESTNLTK has been funded by Eesti Keeletehnoloogia Riiklik Programm under the project EKT57, and supported by Estonian Ministry of Education and Research (grant IUT 20-56 "Computational models for Estonian"). We are grateful to our code contributors: Heiki-Jaan Kaalep, Neeme Kahusk, Karl-Oskar Masing, Andres Matsin, Siim Orasmaa, Timo Petmanson, Annett Saarik, Alexander Tkachenko, Tarmo Vaino and Karl Valliste.

10. References

Apache, U. (2010). Unstructured information management applications. `http://uima.apache.org`. Online; accessed 20-October-2015.

Bird, Steven, E. L. and Klein, E. (2009). *Natural Language Processing with Python*. O'Reilly Media Inc.

Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrell, G., Funk, A., Roberts, A., Damljanovic, D., Heitz, T., Greenwood, M. A., Saggion, H., Petrak, J., Li, Y., and Peters, W. (2011). *Text Processing with GATE (Version 6)*.

Cutting, D., Busch, M., Cohen, D., Gospodnetic, O., Hatcher, E., Hostetter, C., Ingersoll, G., McCandless, M., Messer, B., Naber, D., et al. (2004). Apache lucene. `https://lucene.apache.org/`. Online; accessed 20-October-2015.

Řehůřek, R. and Sojka, P. (2010). Software framework for topic modelling with large corpora.

Ellman, J. (2003). Eurowordnet: A multilingual database with lexical semantic networks: Edited by piek vossen. kluwer academic publishers. 1998. isbn 0792352955. Kluwer Academic Publishers.

Kaalep, H. J. and Muischnek, K. (2012). Robust clause boundary identification for corpus annotation. In *LREC*, pages 1632–1636.

Kaalep, H.-J. and Vaino, T. (2001). Complete morphological analysis in the linguist's toolbox. *Congressus Nonus Internationalis Fenno-Ugristarum Pars V*, pages 9–16. The corresponding C++ code is available from `https://github.com/Filosoft/vabamorf`.

Kahusk, N. (2010). Eurown: an eurowordnet module for python. In *Principles, Construction and Application of Multilingual Wordnets. Proceeding of the 5th Global Wordnet Conference: The 5th International Conference of the Global WordNet Association (GWC–2010)*, pages 360–364.

Lafferty, J., McCallum, A., and Pereira, F. C. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data.

Loria, S. (2014). Textblob: Simplified text processing. *TextBlob. Np*.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60.

McCallum, A. K. (2002). Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Okazaki, N. (2007). Crfsuite: a fast implementation of conditional random fields (crfs). http://www.chokkan.org/software/crfsuite/. Online; accessed 20-October-2015.

Orasmaa, S. (2012). Automaatne ajaväljendite tuvastamine eestikeelsetes tekstides *(Automatic Recognition and Normalization of Temporal Expressions in Estonian Language Texts). Eesti Rakenduslingvistika Ühingu aastaraamat*, (8):153–169.

Pustejovsky, J., Castano, J. M., Ingria, R., Sauri, R., Gaizauskas, R. J., Setzer, A., Katz, G., and Radev, D. R. (2003). Timeml: Robust specification of event and temporal expressions in text. *New directions in question answering*, 3:28–34.

Tjong Kim Sang, E. F. and De Meulder, F. (2003). Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 142–147. Association for Computational Linguistics.

Tkachenko, A., Petmanson, T., and Laur, S. (2013). Named entity recognition in estonian. In *Proceedings of the Workshop on Balto-Slavic NLP*, page 78. Association for Computational Linguistics.