**EstNLTK - NLP Toolkit for Estonian**

Siim Orasama, Timo Petmanson, Alexander Tkachenko, Sven Laur, Heiki-Jaan Kaalep  
Institute of Computer Science, University of Tartu  
Liivi 2, 50409 Tartu, Estonia  
siim.orasama@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 NLTK (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 (Rehurek 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
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.

1. paragraph tokenization
2. sentence tokenization
3. word tokenization
4. morphological analysis
5. named entity recognition
6. verb chain detection
5. clause detection
5. temporal expressions
5. grammars

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 flows, it is brittle and restricting. A small change in an output format may cause subtle errors and parsing routines.

**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 tokenizer 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 [Knalep 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_words carries individual token labels in BIO format.\textsuperscript{1}

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. [Tü istus nurgalauda ja] [tellis kohvi.]) ["She sat on the corner table and [ordered a coffee.]") and clauses embedded within other clauses (e.g. [Meesi, keda seal nõige,] lühkas kirjutades.) ["The man [who we saw there] left in a hurry."]\textsuperscript{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 kilometrit '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.

<table>
<thead>
<tr>
<th>NLP task</th>
<th>Affected layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenize_paragraphs()</td>
<td>paragraphs</td>
</tr>
<tr>
<td>tokenize_sentences()</td>
<td>sentences</td>
</tr>
<tr>
<td>tokenize_words()</td>
<td>words</td>
</tr>
<tr>
<td>tag_analysis()</td>
<td>words, clauses</td>
</tr>
<tr>
<td>tag_clauses()</td>
<td>words, named_entities</td>
</tr>
<tr>
<td>tag_named_entities()</td>
<td>timexes</td>
</tr>
<tr>
<td>tag_timexes()</td>
<td>verb_chains</td>
</tr>
<tr>
<td>tag_verb_chains()</td>
<td>verb_chains</td>
</tr>
</tbody>
</table>

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.
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 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 otis 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.

**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 \([\text{Mikolov et al., 2013}]\), trained on the Estonian Reference Corpus \([\text{Kahusk, 2010}]\). 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 \([\text{https://code.google.com/p/word2vec/}]\). Resulting models can be used with the word2vec command line tools or a Python library gensim \([\text{https://radimrehurek.com/gensim/}]\).

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 ElasticSearch \([\text{https://www.elastic.co/products/elasticsearch}]\) 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

---

In many information retrieval and text classification tasks one must first recognize 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 terms 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. The sample contains 167,180 tokens. We run benchmarks using Python 3.4 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.

<table>
<thead>
<tr>
<th>NLP task</th>
<th>tokens/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokenization</td>
<td>97,095</td>
</tr>
<tr>
<td>Morphological analysis</td>
<td>3,913</td>
</tr>
<tr>
<td>NER</td>
<td>2,550</td>
</tr>
<tr>
<td>Temporal expression tagging</td>
<td>1,905</td>
</tr>
<tr>
<td>Clause boundary detection</td>
<td>5,271</td>
</tr>
<tr>
<td>Verb chain detection</td>
<td>11,658</td>
</tr>
</tbody>
</table>

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.
  
- **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.
  
- **Ajavt**, a Java-based temporal expression tagger for Estonian, licensed under GPL version 2.

The source code is hosted online on [github.com](http://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 Koelehtehnooogia 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 Kahuusk, Karl-Oskar Masing, Andres Matsin, Siim Orasmaa, Timo Petmanson, Annett Saarik, Alexander Tkachenko, Tarmo Vaino and Karl Valliste.

10. References


We introduce ESTNLTK, a Python library for natural language processing in Estonian, which provides a unified programming interface for various NLP components. EstNLTK provides common natural language processing functionality such as paragraph, sentence and word tokenization, morphological analysis, named entity recognition, etc. for the Estonian language. The project is funded by EKT (Eesti Keeletehnoloogia Riiklik Programm). Currently, there are two branches of EstNLTK EstNLTK, the Estonian Natural Language Toolkit, brings together previously developed Estonian NLP tools and resources in a common environment, making them easily accessible. The toolkit is a set of Python libraries that has been created following the example of NLTK. EstNLTK can also be used as a docker image, which allows researchers to skip the installation process and access the toolchain directly from a web browser in the Jupyter notebook (and copy any tutorials to work with). In addition to Python libraries, parts of EstNLTK can also be accessed as a webservice, or a WebLicht service. Figure 1: Using EstNLTK in a WebLicht workflow for morphological analysis.