NameTag: Name Entity Recognizer

Version 1.1.2

Contents

1 Introduction

NameTag is an open-source tool for named entity recognition (NER). NameTag identifies proper names in text and classifies them into predefined categories, such as names of persons, locations, organizations, etc. NameTag is distributed as a standalone tool or a library, along with trained linguistic models. In the Czech language, NameTag achieves state-of-the-art performance (Straková et al. 2013). NameTag is a free software under Mozilla Public License 2.0 license and the linguistic models are free for non-commercial use and distributed under CC BY-NC-SA license, although for some models the original data used to create the model may impose additional licensing conditions. NameTag is versioned using Semantic Versioning.

2 Online

2.1 Online Demo

LINDAT/CLARIN hosts NameTag Online Demo.

2.2 Web Service

LINDAT/CLARIN also hosts NameTag Web Service.

3 Release

3.1 Download

NameTag releases are available on GitHub, either as a pre-compiled binary package, or source code only. The binary package contains Linux, Windows and OS X binaries, Java bindings binary, C# bindings binary and source code of NameTag and all language bindings. While the binary packages do not contain compiled Python or Perl bindings, packages for those languages are available in standard package repositories, i.e. on PyPI and CPAN.

- Latest release
- All releases, Changelog

3.1.1 Language Models

To use NameTag, a language model is needed. The language models are available from LINDAT/CLARIN infrastructure and described further in the NameTag User's Manual. Currently the following language models are available:

- Czech: czech-cnec-140304 (documentation)
- English: in development

3.2 License

NameTag is an open-source project and is freely available for non-commercial purposes. The source code is distributed under Mozilla Public License 2.0 and the pre-compiled binaries and the associated models and data under CC BY-NC-SA, although for some models the original data used to create the model may impose additional licensing conditions.

If you use this tool for scientific work, please give credit to us by referencing NameTag website and Straková et al. 2014.

3.3 Platforms and Requirements

NameTag is available as a standalone tool and as a library for Linux/Windows/OS X. It does not require any additional libraries. As any supervised machine learning tool, it needs trained linguistic models to perform named entity recognition. The models for the Czech language are available with the tool.

4 NameTag Installation

NameTag releases are available on GitHub, either as a pre-compiled binary package, or source code only. The binary package contains Linux, Windows and OS X binaries, Java bindings binary, C# bindings binary and source code of NameTag and all language bindings. While the binary packages do not contain compiled Python or Perl bindings, packages for those languages are available in standard package repositories, i.e. on PyPI and CPAN.

To use NameTag, a language model is needed. Here is a list of available language models.

If you want to compile NameTag manually, sources are available on on GitHub, both in the pre-compiled binary package releases and in the repository itself.

4.1 Requirements

- `G++ 4.7` or newer, `clang 3.2` or newer, Visual C++ 2015 or newer
- `make`
- `SWIG` for language bindings other than `C++`

4.2 Compilation

To compile NameTag, run `make` in the `src` directory.

Make targets and options:

- `exe`: compile the binaries (default)
- `server`: compile the REST server
- `lib`: compile NameTag library (decoding only)
- `BITS=32` or `BITS=64`: compile for specified 32-bit or 64-bit architecture instead of the default one
- `MODE=release`: create release build which statically links the C++ runtime and uses LTO
- `MODE=debug`: create debug build
- `MODE=profile`: create profile build

4.2.1 Platforms

Platform can be selected using one of the following options:

- `PLATFORM=linux`, `PLATFORM=linux-gcc`: gcc compiler on Linux operating system, default on Linux
- `PLATFORM=linux-clang`: clang compiler on Linux, must be selected manually
- `PLATFORM=osx`, `PLATFORM=osx-clang`: clang compiler on OS X, default on OS X; `BITS=32+64` enables multiarch build
- `PLATFORM=win`, `PLATFORM=win-gcc`: gcc compiler on Windows (TDM-GCC is well tested), default on Windows
- `PLATFORM=win-vs`: Visual C++ 2015 compiler on Windows, must be selected manually; note that the `cl.exe` compiler must be already present in `PATH` and corresponding `BITS=32` or `BITS=64` must be specified

Either POSIX shell or Windows CMD can be used as shell, it is detected automatically.

4.2.2 Further Details

MorphoDiTa uses C++ BuilTem system, please refer to its manual if interested in all supported options.

4.3 Other language bindings

4.4 C#

Binary C# bindings are available in NameTag binary packages.

To compile C# bindings manually, run `make` in the `bindings/csharp` directory, optionally with the options descriged in NameTag Installation.

4.4.1 Java

Binary Java bindings are available in NameTag binary packages.

To compile Java bindings manually, run `make` in the `bindings/java` directory, optionally with the options descriged in NameTag Installation. Java 6 and newer is supported.

The Java installation specified in the environment variable `JAVA_HOME` is used. If the environment variable does not exist, the `JAVA_HOME` can be specified using
`make JAVA_HOME=path_to_Java_installation`

4.4.2 Perl

The Perl bindings are available as `Ufal-NameTag` package on CPAN.

To compile Perl bindings manually, run `make` in the `bindings/perl` directory, optionally with the options descriged in NameTag Installation. Perl 5.10 and later is supported.

Path to the include headers of the required Perl version must be specified in the `PERL_INCLUDE` variable using
`make PERL_INCLUDE=path_to_Perl_includes`

4.4.3 Python

The Python bindings are available as `ufal.nametag` package on PyPI.

To compile Python bindings manually, run `make` in the `bindings/python` directory, optionally with options descriged in NameTag Installation. Both Python 2.6+ and Python 3+ are supported.

Path to the include headers of the required Python version must be specified in the `PYTHON_INCLUDE` variable using
`make PYTHON_INCLUDE=path_to_Python_includes`

5 NameTag User's Manual

In a natural language text, the task of named entity recognition (NER) is to identify proper names such as names of persons, organizations and locations. NameTag recognizes named entities in an unprocessed text using MorphoDiTa. MorphoDiTa library tokenizes the text and performs morphological analysis and tagging and NameTag identifies and classifies named entities by an algorithm described in Straková et al. 2013. NameTag can also performs NER in custom tokenized and morphologically analyzed and tagged texts.

Like any supervised machine learning tool, NameTag needs a trained linguistic model. This section describes the available language models and also the commandline tools. The C++ library is described in NameTag API Reference.

5.1 Czech NameTag Models

Czech models are distributed under the CC BY-NC-SA licence. They are trained on Czech Named Entity Corpus 1.1 and 2.0 and internally use MorphoDiTa as a tagger and lemmatizer. Czech models work in NameTag version 1.0 or later.

Czech models are versioned according to the date when released, the version format is `YYMMDD`, where `YY`, `MM` and `DD` are two-digit representation of year, month and day, respectively. The latest version is 140304.

5.1.1 Download

The latest version 140304 of the Czech NameTag models can be downloaded from LINDAT/CLARIN repository.

5.1.2 Acknowledgements

This work has been using language resources developed and/or stored and/or distributed by the LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project *LM2010013*).

Czech models are trained on Czech Named Entity Corpus, which was created by Magda Ševčíková, Zdeněk Žabokrtský, Jana Straková and Milan Straka.

The recognizer research was supported by the projects *MSM0021620838* and *LC536* of Ministry of Education, Youth and Sports of the Czech Republic, *1ET101120503* of Academy of Sciences of the Czech Republic, LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project *LM2010013*), and partially by SVV project number 267 314. The research was performed by Jana Straková, Zdeněk Žabokrtský and Milan Straka.

Czech models use MorphoDiTa as a tagger and lemmatizer, therefore MorphoDiTa Acknowledgements and Czech MorphoDiTa Model Acknowledgements apply.

Publications

- Ševčíková Magda, Žabokrtský Zdeněk, Krůza Ondřej: *Named Entities in Czech: Annotating Data and Developing NE Tagger*. In: Matoušek, V., Mautner, P. (eds.) TSD 2007. LNCS (LNAI), vol. 4629, pp. 188–195. Springer, Heidelberg (2007).
- Kravalová Jana, Žabokrtský Zdeněk: *Czech Named Entity Corpus and SVM-based Recognizer*. In: Proceedings of the 2009 Named Entities Workshop: Shared Task on Transliteration. NEWS 2009, pp. 194–201. Association for Computational Linguistics (2009).
- Straková Jana, Straka Milan, Hajič Jan: *A New State-of-The-Art Czech Named Entity Recognizer*. In: Lecture Notes in Computer Science, Vol. 8082, Text, Speech and Dialogue: 16th International Conference, TSD 2013. Proceedings, Copyright © Springer Verlag, Berlin / Heidelberg, ISBN 978-3-642-40584-6, ISSN 0302-9743, pp. 68-75, 2013
- Straková Jana, Straka Milan and Hajič Jan. *Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition.* In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 13-18, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

5.1.3 Czech Named Entity Corpus 2.0 Model

The model is trained on the training portion of the Czech Named Entity Corpus 2.0. The corpus uses a detailed two-level named entity hierarchy, whose detailed description is available in the documentation of the Czech Named Entity Corpus 2.0. This hierarchy is an updated version of CNEC 1.1 hierarchy and is more suitable for automatic named entity recognition.

`czech-cnec2.0-<version>.ner`
: Czech named entity recognizer trained on training portion of CNEC 2.0.

  The latest version `czech-cnec2.0-140304.ner` reaches 75.38% F1-measure on two-level hierarchy and 79.16% F1-measure on top-level hierarchy of CNEC 2.0 etest data. Model speed: ~40k words/s, model size: ~8MB.

`czech-cnec2.0-<version>-no_numbers.ner`
: Czech named entity recognizer trained on training portion of CNEC 2.0, except for the supertype `n` (Number expressions), which is not included.

  The latest version `czech-cnec2.0-140304-no_numbers.ner` reaches 77.35% F1-measure on two-level hierarchy and 80.59% F1-measure on top-level hierarchy of CNEC 2.0 etest data without `n` (Number expressions) supertype. Model speed: ~45k words/s, model size: ~8MB.

5.1.4 Czech Named Entity Corpus 1.1 Model

The model is trained on the training portion of the Czech Named Entity Corpus 1.1. The corpus uses a detailed two-level named entity hierarchy, whose detailed description is available in the documentation of the Czech Named Entity Corpus 1.1.

`czech-cnec1.1-<version>.ner`
: Czech named entity recognizer trained on training portion of CNEC 1.1.

  The latest version `czech-cnec1.1-140304.ner` reaches 75.47% F1-measure on two-level hierarchy and 78.51% F1-measure on top-level hierarchy of CNEC 1.1 etest data. Model speed: ~35k words/s, model size: ~7MB.

`czech-cnec1.1-<version>-no_numbers.ner`
: Czech named entity recognizer trained on training portion of CNEC 1.1, except for the supertypes `c` (Bibliographic items), `n` (Number expressions) and `q` (Quantitative expressions), which are not included.

  The latest version `czech-cnec1.1-140304-no_numbers.ner` reaches 77.48% F1-measure on two-level hierarchy and 80.72% F1-measure on top-level hierarchy of CNEC 1.1 etest data without `c` (Bibliographic items), `n` (Number expressions) and `q` (Quantitative expressions) supertypes. Model speed: ~40k words/s, model size: ~7MB.

5.2 Running the Recognizer

The NameTag Recognizer can be executed using the following command:
`run_ner recognizer_model`

The input is assumed to be in UTF-8 encoding and can be either already tokenized and segmented, or it can be a plain text which is tokenized and segmented automatically.

Any number of files can be specified after the `recognizer_model`. If an argument `input_file:output_file` is used, the given `input_file` is processed and the result is saved to `output_file`. If only `input_file` is used, the result is saved to standard output. If no argument is given, input is read from standard input and written to standard output.

The full command syntax of `run_ner` is

```
Usage: run_ner [options] recognizer_model [file[:output_file]]...
Options: --input=untokenized|vertical
         --output=vertical|xml
```

5.2.1 Input Formats

The input format is specified using the `--input` option. Currently supported input formats are:

- `untokenized` (default): the input is tokenized and segmented using a tokenizer defined by the model,
- `vertical`: the input is in vertical format, every line is considered a word, with empty line denoting end of sentence.

5.2.2 Output Formats

The output format is specified using the `--output` option. Currently supported output formats are:

- `xml` (default): Simple XML format without a root element, using <`sentence`> element to mark sentences and <`token`> element to mark tokens. The recognized named entities are encoded using <`ne type="..."`> element.

  Example input:

  ```
  Václav Havel byl český dramatik, esejista, kritik komunistického režimu a později
      politik.
  ```

  A NameTag identifies a first name (`pf`), a surname (`ps`) and a person name container (`P`) in the input (line breaks added):

  ```
  <sentence><ne type="P"><ne type="pf"><token>Václav</token></ne> <ne
      type="ps"><token>Havel</token></ne></ne>
  <token>byl</token> <token>český</token> <token>dramatik</token><token>,</token>
      <token>esejista</token><token>,</token>
  <token>kritik</token> <token>komunistického</token> <token>režimu</token>
      <token>a</token> <token>později</token>
  <token>politik</token><token>.</token></sentence>
  ```

- `vertical`: Every found named entity is on a separate line. Each line contains three tab-separated fields: *entity_range*, *entity_type* and *entity_text*. The *entity_range* is composed of token identifiers (counting from 1 and including end-of-sentence; if the input is also `vertical`, token identifiers correspond exactly to line numbers) of tokens forming the named entity and *entity_type* represents its type. The *entity_text* is not strictly necessary and contains space separated words of this named entity.

  Example input:

  ```
  Václav Havel byl český dramatik, esejista, kritik komunistického režimu a později
      politik.
  ```

  Example output:

  ```
  1,2     P       Václav Havel
  1       pf      Václav
  2       ps      Havel
  ```

5.3 Running the Tokenizer

Using the `run_tokenizer` executable it is possible to perform only tokenization and segmentation used in a specified model.

The input is a UTF-8 encoded plain text and the input files are specified same as with the `run_ner` command.

The full command syntax of `run_tokenizer` is

```
run_tokenizer [options] recognizer_model [file[:output_file]]...
Options: --output=vertical|xml
```

5.3.1 Output Formats

The output format is specified using the `--output` option. Currently supported output formats are:

- `xml` (default): Simple XML format without a root element, using <`sentence`> element to mark sentences and <`token`> element to mark tokens.

  Example output for input `Děti pojedou k babičce.  Už se těší.` (line breaks added):

  ```
  <sentence><token>Děti</token> <token>pojedou</token> <token>k</token>
  <token>babičce</token><token>.</token></sentence> <sentence><token>Už</token>
  <token>se</token> <token>těší</token><token>.</token></sentence>
  ```

- `vertical`: Each token is on a separate line, every sentence is ended by a blank line.

  Example output for input `Děti pojedou k babičce.  Už se těší.`:

  ```
  Děti
  pojedou
  k
  babičce
  .

  Už
  se
  těší
  .
  ```

5.4 Running REST Server

NameTag also provides REST server binary `nametag_server`. The binary uses MicroRestD as a REST server implementation and provides NameTag REST API.

The full command syntax of `nametag_server` is

```
nametag_server [options] port (model_name model_file acknowledgements)*
Options: --daemon
         --log=path to log file
```

The `nametag_server` can run either in foreground or in background (when `--daemon` is used). The specified model files are loaded during start and kept in memory all the time. This behaviour might change in future to load the models on demand.

5.5 Training of Custom Models

Training of custom models is possible using the `train_ner` binary.

5.5.1 Training data

To train a named entity recognizer model, training data is needed. The training data must be tokenized and contain annotated name entities. The name entities are non-overlapping, consist of a sequence of words and have a specified type.

The training data must be encoded in UTF-8 encoding. The lines correspond to individual words and an empty line denotes an end of sentence. Each non-empty line contains exactly two tab-separated columns, the first is the word form and the second is the annotation. The format of the annotation is taken from CoNLL-2003: the annotation `O` (or _) denotes no named entity, annotations `I-type` and `B-type` denote named entity of specified type. The `I-type` and `B-type` annotations are equivalent except for one case – if the previous word is also a named entity of same type, then

- if the current word is annotated as `I-type`, it is part of the same named entity as the previous word,

- if the current word is annotated as `B-type`, it is in a different name entity than the previous word (albeit with the same type).

5.5.2 Tagger

Most named entity recognizer models utilize part of speech tags and lemmas. NameTag can utilize several taggers to obtain the tags and lemmas:

`trivial`
: Do not use any tagger. The lemma is the same as the given form and there is no part of speech tag.

`external`
: Use some external tagger. The input "forms" can contain multiple space-separated values, first being the *form*, second the *lemma* and the rest is *part of speech tag*. The part of speech tag is optional. The lemma is also optional and if missing, the form itself is used as a lemma.

`morphodita:model`
: Use MorphoDiTa as a tagger with the specified model. This tagger model is embedded in resulting named entity recognizer model. The *lemmatizer* model of MorphoDiTa is recommended, because it is very fast, small and detailed part of speech tags do not improve the performance of the named entity recognizer significantly.

Lemma Structure

The lemmas used by the recognizer can be structured and consist of three parts:
- *raw lemma* is the textual form of the lemma, possibly ambiguous
- *lemma id* is the unique lemma identification (for example a raw lemma plus a numeric identifier)
- *lemma comment* is additional information about a lemma occurrence, not used to identify the lemma. If used, it usually contains information which is not possible to encode in part of speech tags.

Currently, all these parts are filled only when `morphodita` tagger is used. If `external` tagger is used, raw lemma and lemma id are the same and lemma comment is empty.

5.5.3 Feature Templates

The recognizer utilizes feature templates to generate features which are used as the input to the named entity classifier. The feature templates are specified in a file, one feature template on a line. Empty lines and lines starting with `#` are ignored.

The first space-separated column on a line is the name of the feature template, optionally followed by a slash and a window size. The window size specifies how many adjacent words can observe the feature template value of a given word, with default value of 0 denoting only the word in question.

List of commonly used feature templates follows. Note that it is probably not exhaustive (see the sources in the `features` directory).
- `BrownClusters file [prefix_lengths]` – use Brown clusters found in the specified file. An optional list of lengths of cluster prefixes to be used in addition to the full Brown cluster can be specified. Each line of the Brown clusters file must contain two tab-separated columns, the first of which is the Brown cluster label and the second is a raw lemma.
- `CzechLemmaTerm` – feature template specific for Czech morphological system by Jan Hajič (Hajič 2004). The term information (personal name, geographic name, ...) specified in lemma comment are used as features.
- `Form` – use forms as features
- `Gazetteers [files]` – use given files as gazetteers. Each file is one gazetteers list independent of the others and must contain a set of lemma sequences, each on a line, represented as raw lemmas separated by spaces.
- `Lemma` – use lemma ids as a feature

- `NumericTimeValue` – recognize numbers which could represent hours, minutes, hour:minute time, days, months or years
- `PreviousStage` – use named entities predicted by previous stage as features
- `RawLemma` – use raw lemmas as features
- `RawLemmaCapitalization` – use capitalization of raw lemma as features
- `Tag` – use tags as features
- `URLEmailDetector url_type email_type` – detect URLs and emails. If an URL or an email is detected, it is immediately marked with specified named entity type and not used in further processing.

For inspiration, we present feature file used for Czech NER model. This feature file is a simplified version of feature templates described in the paper Straková et al. 2013: Straková Jana, Straka Milan, Hajič Jan, *A New State-of-The-Art Czech Named Entity Recognizer.* In: Lecture Notes in Computer Science, Vol. 8082, Text, Speech and Dialogue: 16th International Conference, TSD 2013. Proceedings, Copyright © Springer Verlag, Berlin / Heidelberg, ISBN 978-3-642-40584-6, ISSN 0302-9743, pp. 68-75, 2013.

```
# Sentence processors
Form/2
Lemma/2
RawLemma/2
RawLemmaCapitalization/2
Tag/2
NumericTimeValue/1

CzechLemmaTerm/2

BrownClusters/2 clusters/wiki-1000-3

Gazetteers/2 gazetteers/cities.txt gazetteers/clubs.txt gazetteers/countries.txt
    gazetteers/feasts.txt gazetteers/institutions.txt gazetteers/months.txt
    gazetteers/objects.txt gazetteers/psc.txt gazetteers/streets.txt

PreviousStage/5

# Detectors
URLEmailDetector mi me

# Entity processors
CzechAddContainers
```

5.5.4 Running train_ner

The `train_ner` binary has the following arguments (which has to be specified in this order):

1. *ner_identifier* – identifier of the named entity recognizer type. This affects the tokenizer used in this model, and in theory any other aspect of the recognizer. Supported values:
   - *czech*
   - *english*
   - *generic*
2. *tagger* – the tagger identifier as described in the Tagger section
3. *feature_templates_file* – file with feature templates as described in the Feature Templates section.
4. *stages* – the number of stages performed during recognition. Common values are either *1* or *2*. With more stages, the model is larger and recognition is slower, but more accurate.
5. *iterations* – the number of iterations performed when training each stage of the recognizer. With more iterations, training take longer (the recognition time is unaffected), but the model gets over-trained when too many iterations are used. Values from *10* to *30* or *50* are commonly used.

6. *missing_weight* – default value of missing weights in the log-linear model. Common values are small negative real numbers like *-0.2*.
7. *initial_learning_rage* – learning rate used in the first iteration of SGD training method of the log-linear model. Common value is *0.1*.
8. *final_learning_rage* – learning rate used in the last iteration of SGD training method of the log-linear model. Common values are in range from *0.1* to *0.001*, with *0.01* working reasonably well.
9. *gaussian* – the value of Gaussian prior imposed on the weights. In other words, value of L2-norm regularizer. Common value is either *0* for no regularization, or small real number like *0.5*.
10. *hidden_layer* – experimental support for hidden layer in the artificial neural network classifier. To not use the hidden layer (recommended), use *0*. Otherwise, specify the number of neurons in the hidden layer. Please note that non-zero values will create enormous models, slower recognition and are *not* guaranteed to create models with better accuracy.
11. *heldout_data* – optional parameter with heldout data in the described format. If the heldout data is present, the accuracy of the heldout data classification is printed during training. The heldout data is not used in any other way.

The training data in the described format is read from the standard input and the trained model is written to the standard output if the training is successful.

6 NameTag API Reference

The NameTag API is defined in header `nametag.h` and resides in `ufal::nametag` namespace.

The strings used in the NameTag API are always UTF-8 encoded (except from file paths, whose encoding is system dependent).

6.1 NameTag Versioning

NameTag is versioned using Semantic Versioning. Therefore, a version consists of three numbers *major.minor.patch*, optionally followed by a hyphen and pre-release version info, with the following semantics:

- Stable versions have no pre-release version info, development have non-empty pre-release version info.
- Two versions with the same *major.minor* have the same API with the same behaviour, apart from bugs. Therefore, if only *patch* is increased, the new version is only a bug-fix release.
- If two versions v and u have the same *major*, but *minor(v)* is greater than *minor(u)*, version v contains only additions to the API. In other words, the API of u is all present in v with the same behaviour (once again apart from bugs). It is therefore safe to upgrade to a newer NameTag version with the same *major*.
- If two versions differ in *major*, their API may differ in any way.

Models created by NameTag have the same behaviour in all NameTag versions with same *major*, apart from obvious bugfixes. On the other hand, models created from the same data by different *major.minor* NameTag versions may have different behaviour.

6.2 Struct string_piece

```
struct string_piece {
  const char* str;
  size_t len;

  string_piece();
  string_piece(const char* str);
  string_piece(const char* str, size_t len);
  string_piece(const std::string& str);
```

```
}
```

The string_piece is used for efficient string passing. The string referenced in string_piece is not owned by it, so users have to make sure the referenced string exists as long as the string_piece.

6.3 Struct token_range

```
struct token_range {
  size_t start;
  size_t length;
};
```

The token_range represent a range of a token as returned by a tokenizer. The `start` and `length` fields specify the token position in Unicode characters, not in bytes of UTF-8 encoding.

6.4 Struct named_entity

```
struct named_entity {
  size_t start;
  size_t length;
  std::string type;

  named_entity();
  named_entity(size_t start, size_t length, const std::string& type);
};
```

The named_entity is used to represend a named entity. The `start` and `length` fields represent the entity range in either tokens, unicode characters or bytes, depending on the usage. The `type` represents the entity type.

6.5 Class version

```
class version {
 public:
  unsigned major;
  unsigned minor;
  unsigned patch;

  static version current();
};
```

The version class represents NameTag version. See NameTag Versioning for more information.

6.5.1 version::current

```
static version current();
```

Returns current NameTag version.

6.6 Class tokenizer

```
class tokenizer {
 public:
  virtual ~tokenizer() {}

  virtual void set_text(string_piece text, bool make_copy = false) = 0;
  virtual bool next_sentence(std::vector<string_piece>* forms, std::vector<token_range>*
      tokens) = 0;

  static tokenizer* new_vertical_tokenizer();
};
```

The tokenizer class performs segmentation and tokenization of given text. The class is *not* threadsafe.

The tokenizer instances can be obtained either directly using the static method new_vertical_tokenizer or through instances of ner.

6.6.1 tokenizer::set_text

```
virtual void set_text(string_piece text, bool make_copy = false) = 0;
```

Set the text which is to be tokenized.

If `make_copy` is `false`, only a reference to the given text is stored and the user has to make sure it exists until the tokenizer is released or `set_text` is called again. If `make_copy` is `true`, a copy of the given text is made and retained until the tokenizer is released or `set_text` is called again.

6.6.2 tokenizer::next_sentence

```
virtual bool next_sentence(std::vector<string_piece>* forms, std::vector<token_range>*
    tokens) = 0;
```

Locate and return next sentence of the given text. Returns `true` when successful and `false` when there are no more sentences in the given text. The arguments are filled with found tokens if not `NULL`. The `forms` contain token ranges in bytes of UTF-8 encoding, the `tokens` contain token ranges in Unicode characters.

6.6.3 tokenizer::new_vertical_tokenizer

```
static tokenizer new_vertical_tokenizer();
```

Returns a new instance of a vertical tokenizer, which considers every line to be one token, with empty line denoting end of sentence. The user should delete the instance after use.

6.7 Class ner

```
class ner {
 public:
  virtual ~ner() {}

  static ner* load(const char* fname);
  static ner* load(istream& is);

  virtual void recognize(const std::vector<string_piece>& forms, std::vector<named_entity>&
      entities) const = 0;

  virtual tokenizer* new_tokenizer() const = 0;
};
```

A ner instance represents a named entity recognizer. All methods are thread-safe.

6.7.1 ner::load(const char*)

```
static ner* load(const char* fname);
```

Factory method constructor. Accepts C string with a file name of the model. Returns a pointer to an instance of ner which the user should delete after use.

6.7.2 ner::load(istream&)

```
static ner* load(istream& is);
```

Factory method constructor. Accepts an input stream with the model. Returns a pointer to an instance of ner which the user should delete after use.

6.7.3 ner::recognize

```
virtual void recognize(const std::vector<string_piece>& forms, std::vector<named_entity>&
    entities) const = 0;
```

Perform named entity recognition on a tokenized sentence given in the `forms` argument. The found entities are returned in the `entities` argument. The range of the returned named_entity is represented using form indices.

6.7.4 ner::tokenize_and_recognize

```
void tokenize_and_recognize(string_piece text, std::vector<named_entity>& entities, bool
    unicode_offsets = false) const;
```

Perform named entity recognition on an untokenized text given in the `text` argument. The found entities are returned in the `entities` argument. The range of the returned named_entity is represented either in Unicode characters (when `unicode_offsets == true`), or in UTF-8 bytes (when `unicode_offset == false`).

6.7.5 ner::new_tokenizer

```
virtual tokenizer* new_tokenizer() const = 0;
```

Returns a new instance of a suitable tokenizer or `NULL` if no such tokenizer exists. The user should delete it after use.

6.8 C++ Bindings API

Bindings for other languages than C++ are created using SWIG from the C++ bindings API, which is a slightly modified version of the native C++ API. Main changes are replacement of string_piece type by native strings and removal of methods using `istream`. Here is the C++ bindings API declaration:

6.8.1 Helper Structures

```
typedef vector<string> Forms;

struct TokenRange {
  size_t start;
  size_t length;
};
typedef vector<TokenRange> TokenRanges;

struct NamedEntity {
  size_t start;
  size_t length;
  string type;

  NamedEntity();
  NamedEntity(size_t start, size_t length, const string& type);
};
typedef vector<NamedEntity> NamedEntities;
```

6.8.2 Main Classes

```
class Version {
 public:
  unsigned major;
  unsigned minor;
```

```
  unsigned patch;
  string prerelease;

  static Version current();
};

class Tokenizer {
 public:
  virtual void setText(const char* text);
  virtual bool nextSentence(Forms* forms, TokenRanges* tokens);

  static Tokenizer* newVerticalTokenizer();
};

class Ner {
  static ner* load(const char* fname);

  virtual void recognize(Forms& forms, NamedEntities& entities) const;

  virtual Tokenizer* newTokenizer() const;
};
```

6.9 C# Bindings

NameTag library bindings is available in the `Ufal.NameTag` namespace.

The bindings is a straightforward conversion of the `C++` bindings API. The bindings requires native C++ library `libnametag_csharp` (called `nametag_csharp` on Windows).

6.10 Java Bindings

NameTag library bindings is available in the `cz.cuni.mff.ufal.nametag` package.

The bindings is a straightforward conversion of the `C++` bindings API. Vectors do not have native Java interface, see `cz.cuni.mff.ufal.nametag.Forms` class for reference. Also, class members are accessible and modifiable using using `getField` and `setField` wrappers.

The bindings require native C++ library `libnametag_java` (called `nametag_java` on Windows). If the library is found in the current directory, it is used, otherwise standard library search process is used. The path to the C++ library can also be specified using static `nametag_java.setLibraryPath(String path)` call (before the first call inside the C++ library, of course).

6.11 Perl Bindings

NameTag library bindings is available in the `Ufal::NameTag` package. The classes can be imported into the current namespace using the `:all` export tag.

The bindings is a straightforward conversion of the `C++` bindings API. Vectors do not have native Perl interface, see `Ufal::NameTag::Forms` for reference. Static methods and enumerations are available only through the module, not through object instance.

6.12 Python Bindings

NameTag library bindings is available in the `ufal.nametag` module.

The bindings is a straightforward conversion of the `C++` bindings API. In Python 2, strings can be both `unicode`

and UTF-8 encoded `str`, and the library always produces `unicode`. In Python 3, strings must be only `str`.

7 Contact

Authors:
- Milan Straka, straka@ufal.mff.cuni.cz
- Jana Straková, strakova@ufal.mff.cuni.cz

NameTag website.

NameTag LINDAT/CLARIN entry.

8 Acknowledgements

This work has been using language resources developed and/or stored and/or distributed by the LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project LM2010013).

Acknowledgements for individual language models are listed in NameTag User's Manual.

8.1 Publications

- (Straková et al. 2014) Straková Jana, Straka Milan and Hajič Jan. *Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition.* In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 13-18, Baltimore, Maryland, June 2014. Association for Computational Linguistics.
- (Straková et al. 2013) Straková Jana, Straka Milan, Hajič Jan: *A New State-of-The-Art Czech Named Entity Recognizer.* In: Lecture Notes in Computer Science, Vol. 8082, Text, Speech and Dialogue: 16th International Conference, TSD 2013. Proceedings, Copyright © Springer Verlag, Berlin / Heidelberg, ISBN 978-3-642-40584-6, ISSN 0302-9743, pp. 68-75, 2013

8.2 Bibtex for referencing

```
@InProceedings{strakova14,
  author    = {Strakov\'{a}, Jana  and  Straka, Milan  and  Haji\v{c}, Jan},
  title     = {Open-{S}ource {T}ools for {M}orphology, {L}emmatization, {POS} {T}agging and
      {N}amed {E}ntity {R}ecognition},
  booktitle = {Proceedings of 52nd Annual Meeting of the Association for Computational
      Linguistics: System Demonstrations},
  month     = {June},
  year      = {2014},
  address   = {Baltimore, Maryland},
  publisher = {Association for Computational Linguistics},
  pages     = {13--18},
  url       = {http://www.aclweb.org/anthology/P/P14/P14-5003.pdf}
}
```

8.3 Persistent Identifier

If you prefer to reference NameTag by a persistent identifier (PID), you can use `http://hdl.handle.net/11858/00-097C-0000-0023-43CE-E`.