UDPipe

Version 1.2.0

Contents

1 Introduction

UDPipe is a trainable pipeline for tokenization, tagging, lemmatization and dependency parsing of CoNLL-U files. UDPipe is language-agnostic and can be trained given annotated data in CoNLL-U format. Trained models are provided for nearly all UD treebanks. UDPipe is available as a binary for Linux/Windows/OS X, as a library for C++, Python, Perl, Java, C#, and as a web service.

UDPipe is a free software distributed under the Mozilla Public License 2.0 and the linguistic models are free for non-commercial use and distributed under the CC BY-NC-SA license, although for some models the original data used to create the model may impose additional licensing conditions. UDPipe is versioned using Semantic Versioning.

Copyright 2017 by Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University, Czech Republic.

2 Online Web Application and Web Service

UDPipe Web Application is available at http://lindat.mff.cuni.cz/services/udpipe/ using LINDAT/CLARIN infrastructure.

UDPipe REST Web Service is also available, with the API documentation available at http://lindat.mff.cuni.cz/services/udpipe/api-reference.php.

3 Release

3.1 Download

UDPipe releases are available on GitHub, both as source code and as a pre-compiled binary package. The binary package contains Linux, Windows and OS X binaries, Java bindings binary, C# bindings binary, and source code of UDPipe 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 UDPipe, a language model is needed. The language models are available from LINDAT/CLARIN infrastructure and described further in the UDPipe User's Manual. Currently, the following language models are available:

- Universal Dependencies 2.0 Models: udpipe-ud2.0-170801 (documentation)
- CoNLL17 Shared Task Baseline UD 2.0 Models: udpipe-ud2.0-conll17-170315 (documentation)
- Universal Dependencies 1.2 Models: udpipe-ud1.2-160523 (documentation)

3.2 License

UDPipe is an open-source project and is freely available for non-commercial purposes. The library is distributed under Mozilla Public License 2.0 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 Straka et al. 2016 and the UDPipe website.

4 UDPipe Installation

UDPipe 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 UDPipe 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 UDPipe, a language model is needed. Here is a list of available language models.

If you want to compile UDPipe 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 3.0.8` or newer for language bindings other than `C++`

4.2 Compilation

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

Make targets and options:
- `exe`: compile the binaries (default)
- `server`: compile the REST server
- `lib`: compile the static library
- `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

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

4.3 Other language bindings

4.3.1 C#

Binary C# bindings are available in UDPipe binary packages.

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

4.3.2 Java

Binary Java bindings are available in UDPipe binary packages.

To compile Java bindings manually, run `make` in the `bindings/java` directory, optionally with the options described in UDPipe 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
```

4.3.3 Perl

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

To compile Perl bindings manually, run `make` in the `bindings/perl` directory, optionally with the options described in UDPipe 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
```

4.3.4 Python

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

To compile Python bindings manually, run `make` in the `bindings/python` directory, optionally with options described in UDPipe 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 UDPipe User's Manual

Like any supervised machine-learning tool, UDPipe needs a trained linguistic model. This section describes the available language models and also the command line tools and interfaces.

5.1 Running UDPipe

Probably the most common usage of UDPipe is to tokenize, tag and parse the input using

```
udpipe --tokenize --tag --parse udpipe_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 will be tokenized and segmented automatically.

Any number of input files can be specified after the `udpipe_model` and if no file is given, the standard input is used. The output is by default saved to the standard output, but if `--outfile=name` is used, it is saved to the given file name. The output file name can contain a {}, which is replaced by a base name of the processed file (i.e., without directories and an extension).

The full command syntax of running UDPipe is

```
Usage: udpipe [running_opts] udpipe_model [input_files]
       udpipe --train [training_opts] udpipe_model [input_files]
       udpipe --detokenize [detokenize_opts] raw_text_file [input_files]
Running opts: --accuracy (measure accuracy only)
              --input=[conllu|generic_tokenizer|horizontal|vertical]
              --immediate (process sentences immediately during loading)
              --outfile=output file template
              --output=[conllu|matxin|horizontal|plaintext|vertical]
              --tokenize (perform tokenization)
              --tokenizer=tokenizer options, implies --tokenize
              --tag (perform tagging)
              --tagger=tagger options, implies --tag
              --parse (perform parsing)
              --parser=parser options, implies --parse
Training opts: --method=[morphodita_parsito] which method to use
               --heldout=heldout data file name
               --tokenizer=tokenizer options
               --tagger=tagger options
               --parser=parser options
Detokenize opts: --outfile=output file template
Generic opts: --version
              --help
```

5.1.1 Immediate Mode

By default UDPipe loads the whole input file into memory before starting to process it. That allows to store the space markup (see the following Tokenizer section) in most consistent way, i.e., store all spaces following a sentence in the last token of that sentence.

However, sometimes it is desirable to process the input as soon as possible, which can be achieved by specifying the `--immediate` option. In immediate mode, the input is processed and printed as soon as a block of input guaranteed to contain whole sentences is loaded. Specifically, for most input formats the input is processed after loading an empty line (with the exception of `horizontal` input format and `presegmented` tokenizer, where the input is processed after each line).

5.1.2 Loading Model On Demand

Although a model for UDPipe always has to be specified, the model is loaded only if really needed. It is therefore possible to use for example `none` as the model in case it is not required for performing the requested operation (e.g., converting between formats or using a generic tokenizer).

5.1.3 Tokenizer

If the `--tokenize` option is supplied, the input is assumed to be plain text and is tokenized using model tokenizer. Additional arguments to the tokenizer might be specified using `--tokenizer=data` option (which implies `--tokenize`), where `data` is a semicolon-separated list of the following options:

- `normalized_spaces`: by default, UDPipe uses custom MISC fields to exactly encode spaces in the original document (as described below). If the `normalized_spaces` option is given, only the standard CoNLL-U

v2 markup (`SpaceAfter=No` and `# newpar`) is used.
- `presegmented`: the input file is assumed to be already segmented, with each sentence on a separate line, and is only tokenized (respecting sentence breaks)
- `ranges`: for each token, a range in the original document is stored in the format described below.
- `joint_with_parsing`: an experimental mode performing sentence segmentation jointly using the tokenizer and the parser (see *Milan Straka and Jana Straková: Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe* paper for details). The following options are utilized:
  - `joint_max_sentence_len` (default 20): maximum sentence length
  - `joint_change_boundary_logprob` (default -0.5): logprob of using sentence boundary not generated by the tokenizer
  - `joint_sentence_logprob` (default -0.5): additional logprob of every sentence

  The logprob of a sentence is computed using logprob of its best dependency parsing tree, together with `joint_sentence_logprob` and also `joint_change_boundary_logprob` for every sentence boundary not returned by the tokenizer (i.e., either 0, 1 or 2 times). The joint sentence segmentation chooses such a segmentation, where every sentence has length at most `joint_max_sentence_len` and the sum of logprobs of all sentences is as large as possible.

Preserving Original Spaces

By default, UDPipe uses custom MISC fields to store all spaces in the original document. This markup is backward compatible with CoNLL-U v2 `SpaceAfter=No` feature. This markup can be utilized by the `plaintext` output format, which allows reconstructing the original document.

Note that in theory not only spaces, but also other original content can be saved in this way (for example XML tags if the input was encoded in a XML file).

The markup uses the following MISC fields on *tokens* (not words in multi-word tokens):
- `SpacesBefore=content` (by default empty): spaces/other content preceding the token
- `SpacesAfter=content` (by default a space if `SpaceAfter=No` feature is not present, empty otherwise): spaces/other content following the token
- `SpacesInToken=content` (by default equal to the FORM of the token): FORM of the token including original spaces (this is needed only if tokens are allowed to contain spaces and a token contains a tab or newline characters)

The `content` of all the three fields must be escaped to allow storing tabs and newlines. The following C-like schema is used:
- `\s`: space
- `\t`: tab
- `\r`: CR character
- `\n`: LF character
- `\p`: | (pipe character)
- `\\`: \ (backslash character)

Preserving Token Ranges

When the `ranges` tokenizer option is used, the range of each token in the original document is stored in the `TokenRange` MISC field.

The format of the `TokenRange` field (inspired by Python) is `TokenRange=start:end`, where `start` is a zero-based document-level index of the start of the token (counted in Unicode characters) and `end` is a zero-based document-level index of the first character following the token (i.e., the length of the token is `end-start`).

5.1.4 Input Formats

If the tokenizer is not used, the input format can be specified using the `--input` option. The individual input formats can be parametrized in the same way a tokenizer is, by using `format=data` syntax. Currently supported input formats are:

- `conllu` (default): the CoNLL-U format. Supported options:
  - `v2` (default): use CoNLL-U v2
  - `v1`: allow loading only CoNLL-U v1 (i.e., no empty nodes and no spaces in forms and lemmas)
- `generic_tokenizer`: generic tokenizer for English-like languages (with spaces separating tokens and English-like punctuation). The tokenizer is rule-based and needs no trained model. It supports the same options as a model tokenizer, i.e., `normalized_spaces`, `presegmented` and `ranges`.
- `horizontal`: each sentence on a separate line, with tokens separated by spaces. In order to allow spaces in tokens, Unicode character 'NO-BREAK SPACE' (U+00A0) is considered part of token and converted to a space during loading.
- `vertical`: each token on a separate line, with an empty line denoting end of sentence; only the first tab-separated word is used as a token, the rest of the line is ignored.

Note that a model tokenizer can be specified using the `--input` option too, by using the `tokenizer` input format, for example using `--input tokenizer=ranges`.

5.1.5 Tagger

If the `--tag` option is supplied, the input is POS tagged and lemmatized using the model tagger. Additional arguments to the tagger might be specified using the `--tagger=data` option (which implies `--tag`).

5.1.6 Dependency Parsing

If the `--parse` option is supplied, the input is parsed using the model dependency parser. Additional arguments to the parser might be specified using the `--parser=data` option (which implies `--parse`).

5.1.7 Output Formats

The output format is specified using the `--output` option. The individual output formats can be parametrized in the same way as input formats, by using the `format=data` syntax. Currently supported output formats are:

- `conllu` (default): the CoNLL-U format Supported options:
  - `v2` (default): use CoNLL-U v2
  - `v1`: produce output in CoNLL-U v1 format. Note that this is a lossy process, as empty nodes are ignored and spaces in forms and lemmas are converted to underscores.
- `matxin`: the Matxin format
- `horizontal`: writes the *words* (in the UD sense) in horizontal format, that is, each sentence is on a separate line, with words separated by a single space. Because words can contain spaces in CoNLL-U v2, the spaces in words are converted to Unicode character 'NO-BREAK SPACE' (U+00A0). Supported options:
  - `paragraphs`: an empty line is printed after the end of a paragraph or a document (recognized by `# newpar` or `# newdoc` comments)
- `plaintext`: writes the *tokens* (in the UD sense) using original spacing. By default, UDPipe's custom MISC features (`SpacesBefore`, `SpacesAfter` and `SpacesInToken`, see the description in the Tokenizer section) are used to reconstruct the exact original spaces. However, if the document does not contain these features or if you want only normalized spacing, you can use the following option:
  - `normalized_spaces`: write one sentence on a line, and either one or no space between tokens according to the `SpaceAfter=No` feature
- `vertical`: each word on a separate line, with an empty line denoting the end of sentence. Supported options:
  - `paragraphs`: an empty line is printed after the end of a paragraph or a document (recognized by `# newpar` or `# newdoc` comments)

5.2 Running the UDPipe REST Server

UDPipe also provides a REST server binary called `udpipe_server`. The binary uses MicroRestD as a REST server implementation and provides UDPipe REST API.

The full command syntax of `udpipe_server` is

```
udpipe_server [options] port default_model (rest_id model_file acknowledgements)*
Options: --concurrent_models=maximum concurrently loaded models (default 10)
         --daemon (daemonize after start)
         --no_check_models_loadable (do not check models are loadable)
         --no_preload_default (do not preload default model)
```

The `udpipe_server` can run either in foreground or in background (when `--daemon` is used).

Since UDPipe 1.1.1, the models are loaded on demand, so that at most `concurrent_models` (default 10) are kept in memory at the same time. The model files are opened during start and never closed until the server stops. Unless `no_check_models_loadable` is specified, the model files are also checked to be loadable during start. Note that the default model is preloaded and never released, unless `no_preload_default` is given. (Before UDPipe 1.1.1, specified model files were loaded during start and kept in memory all the time.)

5.3 Training UDPipe Models

Custom UDPipe models can be trained using the following syntax:

```
udpipe --train model.output [--heldout=heldout_data] training_file ...
```

The training data should be in the CoNLL-U format.

By default, three model components are trained – tokenizer, tagger and parser. Any subset of the model components can be trained and a model component may be copied from an existing model.

The training options are specified for each model component separately using the `--tokenizer`, `--tagger` and `--parser` options. If a model component should not be trained, value `none` should be used (e.g., `--tagger=none`).

The options are `name=value` pairs separated by a semicolon. The value can be either a simple string value (ending by a semicolon), file content specified as `name=file:filename`, or an arbitrary string value specified as `name=data:length:value`, where the value is exactly `length` bytes long.

5.3.1 Reusing Components from Existing Models

The model components (tagger, parser or tagger) can be reused from existing models, by specifying the `from_model=file:filename` option.

5.3.2 Random Hyperparameter Search

The default values of hyperparameters are set to the values which were used the most during UD 1.2 models training, but if you want to reach best performance, the hyperparameters must be tuned.

Apart from manual grid search, UDPipe can perform a simple random search. You can perform the random search by repeatedly training UDPipe (preferably in parallel, most likely on different computers) while specifying different training run number – some of the hyperparameters (chosen by us; you can of course override their value by specifying it on the command line) change their values in different training runs. The pseudorandom sequences of hyperparameters are of course deterministic.

The training run can be specified by providing the `run=number` option to a model component. The run number 1 is the default one (with the best hyperparameters for the UD 1.2 models), run numbers 2 and more randomize the hyperparameters.

5.3.3 Tokenizer

The tokenizer is trained using the `SpaceAfter=No` features in the CoNLL-U files. If the feature is not present, a *detokenizer* can be used to guess the `SpaceAfter=No` features according to a supplied plain text (which typically does not overlap with the texts in the CoNLL-U files).

In order to use the detokenizer, use the `detokenizer=file:filename_with_plaintext` option. In UD 1.2 models, the optimal performance is achieved with very small plain texts – only 500kB.

The tokenizer recognizes the following options:

- `tokenize_url` (default 1): tokenize URLs and emails using a manually implemented recognizer
- `allow_spaces` (default 1 if any token contains a space, 0 otherwise): allow tokens to contain spaces
- `dimension` (default 24): dimension of character embeddings and of the per-character bidirectional GRU. Note that inference time is quadratic in this parameter. Supported values are only 16, 24 and 64, with 64 needed only for languages with complicated tokenization like Japanese, Chinese or Vietnamese.
- `epochs` (default 100): the number of epochs to train the tokenizer for
- `batch_size` (default 50): batch size used during tokenizer training
- `learning_rate` (default 0.005): the learning rate used during tokenizer training
- `dropout` (default 0.1): dropout used during tokenizer training
- `early_stopping` (default 1 if heldout is given, 0 otherwise): perform early stopping, choosing training iteration maximizing sentences F1 score plus tokens F1 score on heldout data

During random hyperparameter search, `batch_size` is chosen uniformly from {*50,100*} and `learning_rate` logarithmically from <*0.0005, 0.01)*.

Detokenizing CoNLL-U Files

The `--detokenizer` option allows generating the `SpaceAfter=No` features automatically from a given plain text. Even if the current algorithm is very simple and makes quite a lot of mistakes, the tokenizer trained on generated features is very close to a tokenizer trained on gold `SpaceAfter=No` features (the difference in token F1 score is usually one or two tenths of percent).

The generated `SpaceAfter=No` features are only used during tokenizer training, not printed. However, if you would like to obtain the CoNLL-U files with automatic detokenization (generated `SpaceAfter=No` features), you can run UDPipe with the `--detokenize` option. In this case, you have to supply plain text in the given language (usually the best results are achieved with just 500kB or 1MB of text) and UDPipe then detokenizes all the given CoNLL-U files.

The complete usage of the `--detokenize` option is:

```
udpipe --detokenize [detokenize_opts] raw_text_file [input_files]
Detokenize opts: --outfile=output file template
```

5.3.4 Tagger

The tagging is currently performed using MorphoDiTa. The UDPipe tagger consists of possibly several MorphoDiTa models, each tagging some of the POS tags and/or lemmas.

By default, only one model is constructed, which generates all available tags (UPOS, XPOS, Feats and Lemma). However, we found out during the UD 1.2 models training that performance improves if one model tags the UPOS, XPOS and Feats tags, while the other is performing lemmatization. Therefore, if you utilize two MorphoDiTa models, by default the first one generates all tags (except lemmas) and the second one performs lemmatization.

The number of MorphoDiTa models can be specified using the `models=number` parameter. All other parameters may be either generic for all models (`guesser_suffix_rules=5`), or specific for a given model (`guesser_suffix_rules_2=6`), including the `from_model` option (therefore, MorphoDiTa models can be trained separately and then combined together into one UDPipe model).

Every model utilizes UPOS for disambiguation and the first model is the one producing the UPOS tags on output.

The tagger recognizes the following options:

- `use_lemma` (default for the second model and also if there is only one model): use the lemma field internally to perform disambiguation; the lemma may be not outputted
- `provide_lemma` (default for the second model and also if there is only one model): produce the disambiguated lemma on output
- `use_xpostag` (default for the first model): use the XPOS tags internally to perform disambiguation; it may not be outputted
- `provide_xpostag` (default for the first model): produce the disambiguated XPOS tag on output
- `use_feats` (default for the first model): use the Feats internally to perform disambiguation; it may not be outputted
- `provide_feats` (default for the first model): produce the disambiguated Feats field on output
- `dictionary_max_form_analyses` (default 0 - unlimited): the maximum number of (most frequent) form analyses from UD training data that are to be kept in the morphological dictionary
- `dictionary_file` (default empty): use a given custom morphological dictionary, where each line contains 5 tab-separated fields FORM, LEMMA, UPOSTAG, XPOSTAG and FEATS. Note that this dictionary data is appended to the dictionary created from the UD training data, not replacing it.
- `guesser_suffix_rules` (default 8): number of rules generated for every suffix
- `guesser_prefixes_max` (default 4 if "provide_lemma', 0 otherwise): maximum number of form-generating prefixes to use in the guesser
- `guesser_prefix_min_count` (default 10): minimum number of occurrences of form-generating prefix to consider using it in the guesser
- `guesser_enrich_dictionary` (default 6 if no `dictionary_file` is passed, 0 otherwise): number of rules generated for forms present in training data (assuming that the analyses from the training data may not be all)
- `iterations` (default 20): number of training iterations to perform
- `early_stopping` (default 1 if heldout is given, 0 otherwise): perform early stopping, choosing training iteration maximizing tagging accuracy on the heldout data
- `templates` (default `lemmatizer` for second model, `tagger` otherwise): MorphoDiTa feature templates to use, either `lemmatizer` which focuses more on lemmas, or `tagger` which focuses more on UPOS/XPOS/FEATS

During random hyperparameter search, `guesser_suffix_rules` is chosen uniformly from $\{5,6,7,8,9,10,11,12\}$ and `guesser_enrich_dictionary` is chosen uniformly from $\{3,4,5,6,7,8,9,10\}$.

5.3.5 Parser

The parsing is performed using Parsito, which is a transition-based parser using a neural-network classifier.

The transition-based systems can be configured by the following options:

- `transition_system` (default projective): which transition system to use for parsing (language dependent, you can choose according to language properties or try all and choose the best one)
  - `projective`: projective stack-based arc standard system with `shift`, `left_arc` and `right_arc` transitions
  - `swap`: fully non-projective system which extends `projective` system by adding the `swap` transition
  - `link2`: partially non-projective system which extends `projective` system by adding `left_arc2` and `right_arc2` transitions
- `transition_oracle` (default dynamic/static_lazy_static whichever first is applicable): which transition oracle to use for the chosen `transition_system`:
  - `transition_system=projective`: available oracles are `static` and `dynamic` (`dynamic` usually gives better results, but training time is slower)
  - `transition_system=swap`: available oracles are `static_eager` and `static_lazy` (`static_lazy` almost always gives better results)
  - `transition_system=link2`: only available oracle is `static`
- `structured_interval` (default 8): use search-based oracle in addition to the `translation_oracle` specified. This almost always gives better results, but makes training 2-3 times slower. For details, see the paper *Straka et al. 2015: Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle*

- `single_root` (default 1): allow only single root when parsing, and make sure only the root node has the `root` deprel (note that training data are checked to be in this format)

The Lemmas/UPOS/XPOS/FEATS used by the parser are configured by:
- `use_gold_tags` (default 0): if false and a tagger exists, the Lemmas/UPOS/XPOS/FEATS for both the training and heldout data are generated by the tagger, otherwise they are taken from the gold data

The embeddings used by the parser can be specified as follows:
- `embedding_upostag` (default 20): the dimension of the UPos embedding used in the parser
- `embedding_feats` (default 20): the dimension of the Feats embedding used in the parser
- `embedding_xpostag` (default 0): the dimension of the XPos embedding used in the parser
- `embedding_form` (default 50): the dimension of the Form embedding used in the parser
- `embedding_lemma` (default 0): the dimension of the Lemma embedding used in the parser
- `embedding_deprel` (default 20): the dimension of the Deprel embedding used in the parser
- `embedding_form_file`: pre-trained word embeddings in `word2vec` textual format
- `embedding_lemma_file`: pre-trained lemma embeddings in `word2vec` textual format
- `embedding_form_mincount` (default 2): for forms not present in the pre-trained embeddings, generate random embeddings if the form appears at least this number of times in the trainig data (forms not present in the pre-trained embeddings and appearing less number of times are considered OOV)
- `embedding_lemma_mincount` (default 2): for lemmas not present in the pre-trained embeddings, generate random embeddings if the lemma appears at least this number of times in the trainig data (lemmas not present in the pre-trained embeddings and appearing less number of times are considered OOV)

The neural-network training options:
- `iterations` (default 10): number of training iterations to use
- `hidden_layer` (default 200): the size of the hidden layer
- `batch_size` (default 10): batch size used during neural-network training
- `learning_rate` (default 0.02): the learning rate used during neural-network training
- `learning_rate_final` (0.001): the final learning rate used during neural-network training
- `l2` (0.5): the L2 regularization used during neural-network training
- `early_stopping` (default 1 if heldout is given, 0 otherwise): perform early stopping, choosing training iteration maximizing LAS on heldout data

During random hyperparameter search, `structured_interval` is chosen uniformly from {*0,8,10*}, `learning_rate` is chosen logarithmically from <`0.005,0.04)` and `l2` is chosen uniformly from <`0.2,0.6)`.

Pre-trained Word Embeddings

The pre-trained word embeddings for forms and lemmas can be specified in the `word2vec` textual format using the `embedding_form_file` and `embedding_lemma_file` options.

Note that pre-training word embeddings even on the UD data itself improves the accuracy (we use `word2vec` with `-cbow 0 -size 50 -window 10 -negative 5 -hs 0 -sample 1e-1 -threads 12 -binary 0 -iter 15 -min-count 2` options to pre-train on the UD data after converting it to the horizontal format using `udpipe --output=horizontal`).

Forms and lemmas can contain spaces in CoNLL-U v2, so these spaces are converted to a Unicode character 'NO-BREAK SPACE' (U+00A0) before performing the embedding lookup, because spaces are usually used to delimit tokens in word embedding generating software (both `word2vec` and `glove` use spaces to separate words on input and on output). When using UDPipe to generate plain texts from CoNLL-U format using `--output=horizontal`, this space replacing happens automatically.

When looking up an embedding for a given word, the following possibilities are tried in the following order until a match is found (or an embedding for unknown word is returned):
- original word
- all but the first character lowercased
- all characters lowercased
- if the word contains only digits, just the first digit is tried

5.3.6 Measuring Model Accuracy

Measuring custom model accuracy can be performed by running:
```
udpipe --accuracy [udpipe_options] udpipe_model file ...
```

The command syntax is similar to the regular UDPipe operation, only the input must be always in the CoNLL-U format and the `--input` and `--output` options are ignored.

Three different settings (depending on `--tokenize(r)`, `--tag(ger)` and `--parse(r)`) can be evaluated:

- `--tokenize(r) [--tag(ger) [--parse(r)]]`: Tokenizer is used to segment and tokenize plain text (obtained by `SpaceAfter=No` features and `# newdoc` and `# newpar` comments in the input file). Optionally, a tagger is used on the resulting data to obtain Lemma/UPOS/XPOS/Feats columns and eventually a parser can be used to parse the results.

  The tokenizer is evaluated using F1-score on tokens, multi-word tokens, sentences and words. The words are aligned using a word-alignment algorithm described in the CoNLL 2017 Shared Task in UD Parsing. The tagger and parser are evaluated on aligned words, resulting in F1 scores of Lemmas/UPOS/XPOS/Feats/UAS/LAS.
- `--tag(ger) [--parse(r)]`: The gold segmented and tokenized input is tagged (and then optionally parsed using the tagger outputs) and then evaluated.
- `--parse(r)`: The gold segmented and tokenized input is parsed using gold morphology (Lemmas/UPOS/XPOS/Feats) and evaluated.

5.4 Universal Dependencies 2.0 Models

Universal Dependencies 2.0 Models are distributed under the CC BY-NC-SA licence. The models are based solely on Universal Dependencies 2.0 treebanks. The models work in UDPipe version 1.2 and later.

Universal Dependencies 2.0 Models are versioned according to the date released in the format `YYMMDD`, where `YY`, `MM` and `DD` are two-digit representation of year, month and day, respectively. The latest version is 170801.

5.4.1 Download

The latest version 170801 of the Universal Dependencies 2.0 models can be downloaded from LINDAT/CLARIN repository.

5.4.2 Acknowledgements

This work has been partially supported and has been using language resources and tools developed, stored and distributed by the LINDAT/CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic (project *LM2015071*). The wark was also partially supported by OP VVV projects CZ.02.1.01/0.0/0.0/16_013/0001781 and CZ.02.2.69/0.0/0.0/16_018/0002373, and by SVV project number 260 453.

The models were trained on Universal Dependencies 2.0 treebanks.

For the UD treebanks which do not contain original plain text version, raw text is used to train the tokenizer instead. The plain texts were taken from the W2C – Web to Corpus.

Publications

- (Straka et al. 2017) Milan Straka and Jana Straková. *Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe*. In Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, Vancouver, Canada, August 2017.

- (Straka et al. 2016) Straka Milan, Hajič Jan, Straková Jana. *UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing*. In Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016), Portorož, Slovenia, May 2016.

5.4.3 Model Description

The Universal Dependencies 2.0 models contain 68 models of 50 languages, each consisting of a tokenizer, tagger, lemmatizer and dependency parser, all trained using the UD data. Note that we use custom train-dev split, by moving sentences from the beginning of dev data to the end of train data, until the training data is at least 9 times the dev data.

The tokenizer is trained using the `SpaceAfter=No` features. If the features are not present in the data, they can be filled in using raw text in the language in question.

The tagger, lemmatizer and parser are trained using gold UD data.

Details about model architecture and training process can be found in the (Straka et al. 2017) paper.

Reproducible Training

In case you want to train the same models, scripts for downloading and resplitting UD 2.0 data, precomputed word embedding, raw texts for tokenizers, all hyperparameter values and training scripts are available in the second archive on the model download page.

5.4.4 Model Performance

We present the tagger, lemmatizer and parser performance, measured on the testing portion of the data, evaluated in three different settings: using raw text only, using gold tokenization only, and using gold tokenization plus gold morphology (UPOS, XPOS, FEATS and Lemma).

Treebank	Mode	Words	Sents	UPOS	XPOS	Feats	AllTags	Lemma	UA
Ancient Greek	Raw text	100.0%	98.7%	82.4%	72.3%	85.8%	72.3%	82.6%	64.4
Ancient Greek	Gold tok	-	-	82.4%	72.4%	85.8%	72.3%	82.7%	64.6
Ancient Greek	Gold tok+morph	-	-	-	-	-	-	-	69.2
Ancient Greek-PROIEL	Raw text	100.0%	47.2%	95.8%	96.0%	88.6%	87.2%	92.6%	71.8
Ancient Greek-PROIEL	Gold tok	-	-	95.8%	96.1%	88.7%	87.2%	92.8%	77.2
Ancient Greek-PROIEL	Gold tok+morph	-	-	-	-	-	-	-	79.7
Arabic	Raw text	93.8%	83.1%	88.4%	83.4%	83.5%	82.3%	87.5%	71.7
Arabic	Gold tok	-	-	94.4%	89.5%	89.6%	88.3%	92.6%	81.3
Arabic	Gold tok+morph	-	-	-	-	-	-	-	82.9
Basque	Raw text	100.0%	99.5%	93.2%	-	87.6%	-	93.8%	75.8
Basque	Gold tok	-	-	93.3%	-	87.7%	-	93.9%	75.9
Basque	Gold tok+morph	-	-	-	-	-	-	-	82.3
Belarusian	Raw text	99.4%	76.8%	88.2%	85.6%	71.7%	68.6%	81.3%	68.0
Belarusian	Gold tok	-	-	88.7%	85.7%	72.4%	69.2%	81.5%	69.4
Belarusian	Gold tok+morph	-	-	-	-	-	-	-	76.8
Bulgarian	Raw text	99.9%	93.9%	97.6%	94.6%	95.6%	94.0%	94.6%	88.8
Bulgarian	Gold tok	-	-	97.7%	94.7%	95.6%	94.1%	94.7%	89.5
Bulgarian	Gold tok+morph	-	-	-	-	-	-	-	92.0
Catalan	Raw text	100.0%	99.2%	98.0%	98.0%	97.1%	96.5%	97.9%	88.8
Catalan	Gold tok	-	-	98.0%	98.0%	97.2%	96.5%	97.9%	88.8
Catalan	Gold tok+morph	-	-	-	-	-	-	-	91.1
Chinese	Raw text	90.2%	98.8%	84.0%	83.8%	89.0%	82.7%	90.2%	62.9
Chinese	Gold tok	-	-	92.2%	92.0%	98.7%	90.8%	100.0%	75.0
Chinese	Gold tok+morph	-	-	-	-	-	-	-	84.1
Coptic	Raw text	65.8%	35.7%	62.6%	62.1%	65.7%	62.1%	64.6%	41.1
Coptic	Gold tok	-	-	95.1%	94.3%	99.7%	94.2%	96.2%	83.2
Coptic	Gold tok+morph	-	-	-	-	-	-	-	88.1
Croatian	Raw text	99.9%	97.0%	95.9%	-	84.3%	-	94.4%	83.0
Croatian	Gold tok	-	-	96.0%	-	84.4%	-	94.4%	83.9
Croatian	Gold tok+morph	-	-	-	-	-	-	-	87.1
Czech	Raw text	99.9%	91.6%	98.3%	92.8%	92.1%	91.7%	97.8%	86.8
Czech	Gold tok	-	-	98.4%	92.9%	92.2%	91.9%	97.9%	87.7
Czech	Gold tok+morph	-	-	-	-	-	-	-	90.2
Czech-CAC	Raw text	100.0%	99.8%	98.1%	90.6%	89.4%	89.1%	97.0%	86.9
Czech-CAC	Gold tok	-	-	98.1%	90.7%	89.5%	89.1%	97.1%	87.0
Czech-CAC	Gold tok+morph	-	-	-	-	-	-	-	89.7
Czech-CLTT	Raw text	99.5%	92.3%	96.5%	87.5%	87.8%	87.3%	96.8%	80.2
Czech-CLTT	Gold tok	-	-	97.0%	87.9%	88.3%	87.7%	97.2%	81.0
Czech-CLTT	Gold tok+morph	-	-	-	-	-	-	-	83.8
Danish	Raw text	99.8%	77.9%	95.2%	-	94.2%	-	94.9%	78.4
Danish	Gold tok	-	-	95.5%	-	94.5%	-	95.0%	80.4
Danish	Gold tok+morph	-	-	-	-	-	-	-	85.0
Dutch	Raw text	99.8%	77.6%	91.4%	88.1%	89.3%	87.0%	89.9%	75.4
Dutch	Gold tok	-	-	91.8%	88.8%	89.9%	87.7%	90.1%	77.0
Dutch	Gold tok+morph	-	-	-	-	-	-	-	82.9
Dutch-LassySmall	Raw text	100.0%	80.4%	97.6%	-	97.2%	-	98.1%	84.4
Dutch-LassySmall	Gold tok	-	-	97.7%	-	97.4%	-	98.2%	87.5
Dutch-LassySmall	Gold tok+morph	-	-	-	-	-	-	-	89.7
English	Raw text	99.0%	76.6%	93.5%	92.9%	94.4%	91.5%	96.0%	80.2
English	Gold tok	-	-	94.5%	93.9%	95.4%	92.5%	96.9%	84.3
English	Gold tok+morph	-	-	-	-	-	-	-	87.8
English-LinES	Raw text	99.9%	86.2%	95.0%	92.7%	-	-	-	78.0
English-LinES	Gold tok	-	-	95.1%	92.8%	-	-	-	79.5
English-LinES	Gold tok+morph	-	-	-	-	-	-	-	84.1
English-ParTUT	Raw text	99.6%	97.5%	94.2%	94.0%	93.3%	92.0%	96.9%	81.0
English-ParTUT	Gold tok	-	-	94.6%	94.4%	93.6%	92.3%	97.3%	82.1
English-ParTUT	Gold tok+morph	-	-	-	-	-	-	-	86.4
Estonian	Raw text	99.9%	94.2%	91.2%	93.2%	85.0%	83.2%	84.5%	72.4
Estonian	Gold tok	-	-	91.3%	93.2%	85.0%	83.3%	84.5%	72.8
Estonian	Gold tok+morph	-	-	-	-	-	-	-	83.1
Finnish	Raw text	99.7%	86.7%	94.5%	95.7%	91.5%	90.3%	86.5%	80.5
Finnish	Gold tok	-	-	94.9%	96.0%	91.8%	90.7%	86.8%	82.0

5.5 CoNLL17 Shared Task Baseline UD 2.0 Models

As part of CoNLL 2017 Shared Task in UD Parsing, baseline models for UDPipe were released. The CoNLL 2017 Shared Task models were trained on most of UD 2.0 treebanks (64 of them) and are distributed under the CC BY-NC-SA licence.

Note that the models were released when the test set of UD 2.0 was unknown. Therefore, the models were trained on a subset of training data only, to allow fair comparison on the development data (which were unused during training and hyperparameter settings). Consequently, the performance of the models is not directly comparable to other models. Details about the concrete data split, hyperparameter values and model performance are available in the model archive.

5.5.1 Download

The CoNLL17 Shared Task Baseline UD 2.0 Models can be downloaded from LINDAT/CLARIN repository.

5.5.2 Acknowledgements

This work has been partially supported and has been using language resources and tools developed, stored and distributed by the LINDAT/CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic (project *LM2015071*).

The models were trained on a Universal Dependencies 2.0 treebanks.

5.6 Universal Dependencies 1.2 Models

Universal Dependencies 1.2 Models are distributed under the CC BY-NC-SA licence. The models are based solely on Universal Dependencies 1.2 treebanks. The models work in UDPipe version 1.0.

Universal Dependencies 1.2 Models are versioned according to the date released in the format `YYMMDD`, where `YY`, `MM` and `DD` are two-digit representation of year, month and day, respectively. The latest version is 160523.

5.6.1 Download

The latest version 160523 of the Universal Dependencies 1.2 models can be downloaded from LINDAT/CLARIN repository.

5.6.2 Acknowledgements

This work has been partially supported and has been using language resources and tools developed, stored and distributed by the LINDAT/CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic (project *LM2015071*).

The models were trained on Universal Dependencies 1.2 treebanks.

For the UD treebanks which do not contain original plain text version, raw text is used to train the tokenizer instead. The plain texts were taken from the W2C – Web to Corpus.

Publications

- (Straka et al. 2016) Straka Milan, Hajič Jan, Straková Jana. *UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing.* LREC 2016, Portorož, Slovenia, May 2016.

5.6.3 Model Description

The Universal Dependencies 1.2 models contain 36 models, each consisting of a tokenizer, tagger, lemmatizer and dependency parser, all trained using the UD data. The model for Japanese is missing, because we do not have the license for the required corpus of Mainichi Shinbun 1995.

The tokenizer is trained using the `SpaceAfter=No` features. If the features are not present in the data, they can be filled in using raw text in the language in question (surprisingly, quite little data suffices, we use 500kB).

The tagger, lemmatizer and parser are trained using gold UD data.

Details about model architecture and training process can be found in the (Straka et al. 2016) paper.

5.6.4 Model Performance

We present the tagger, lemmatizer and parser performance, measured on the testing portion of the data. Only the segmentation and the tokenization of the testing data is retained before evaluation. Therefore, the dependency parser is evaluated without gold POS tags.

Treebank	UPOS	XPOS	Feats	All Tags	Lemma	UAS	LAS
Ancient Greek	91.1%	77.8%	88.7%	77.7%	86.9%	68.1%	61.6%
Ancient Greek-PROIEL	96.7%	96.4%	89.3%	88.4%	93.4%	75.8%	69.6%
Arabic	98.8%	97.7%	97.8%	97.6%	-	80.4%	75.6%
Basque	93.3%	-	87.2%	85.4%	93.5%	74.8%	69.5%
Bulgarian	97.8%	94.8%	94.4%	93.1%	94.6%	89.0%	84.2%
Croatian	94.9%	-	85.5%	85.0%	93.1%	78.6%	71.0%
Czech	98.4%	93.2%	92.6%	92.2%	97.8%	86.9%	83.0%
Danish	95.8%	-	94.8%	93.6%	95.2%	78.6%	74.8%
Dutch	89.7%	88.7%	91.2%	86.4%	88.9%	78.1%	70.7%
English	94.5%	93.8%	95.4%	92.5%	97.0%	84.2%	80.6%
Estonian	88.0%	73.7%	80.0%	73.6%	77.0%	79.9%	71.5%
Finnish	94.9%	96.0%	93.2%	92.1%	86.8%	81.0%	76.5%
Finnish-FTB	94.0%	91.6%	93.3%	91.2%	89.1%	81.5%	76.9%
French	95.8%	-	-	95.8%	-	82.8%	78.4%
German	90.5%	-	-	90.5%	-	78.2%	72.2%
Gothic	95.5%	95.7%	88.0%	86.3%	93.4%	76.4%	68.2%
Greek	97.3%	97.3%	92.8%	91.7%	94.8%	80.3%	76.5%
Hebrew	94.9%	94.9%	91.3%	90.5%	-	82.6%	76.8%
Hindi	95.8%	94.8%	90.2%	87.7%	98.0%	91.7%	87.5%
Hungarian	92.6%	-	89.9%	88.9%	86.9%	77.0%	70.6%
Indonesian	93.5%	-	-	93.5%	-	79.9%	73.3%
Irish	91.8%	90.3%	79.4%	76.6%	87.3%	74.4%	66.1%
Italian	97.2%	97.0%	97.1%	96.2%	97.7%	88.6%	85.8%
Latin	91.2%	75.8%	79.3%	75.6%	79.9%	57.1%	46.7%
Latin-ITT	98.8%	94.0%	94.6%	93.8%	98.3%	79.9%	76.4%
Latin-PROIEL	96.4%	96.0%	88.9%	88.2%	95.3%	75.3%	68.3%
Norwegian	97.2%	-	95.5%	94.7%	96.9%	86.7%	84.1%
Old Church Slavonic	95.3%	95.1%	89.1%	88.2%	92.9%	80.6%	73.4%
Persian	97.0%	96.3%	96.5%	96.2%	-	83.8%	79.4%
Polish	95.8%	84.0%	84.1%	83.8%	92.8%	86.3%	79.6%
Portuguese	97.6%	92.3%	95.3%	92.0%	97.8%	85.8%	81.9%
Romanian	89.0%	81.0%	82.3%	81.0%	75.3%	68.6%	56.9%
Slovenian	95.7%	88.2%	88.6%	87.5%	95.0%	84.1%	80.3%
Spanish	95.3%	-	95.9%	93.4%	96.3%	84.2%	80.3%
Swedish	95.8%	93.9%	94.8%	93.2%	95.5%	81.4%	77.1%
Tamil	85.9%	80.8%	84.3%	80.2%	88.0%	67.2%	58.8%

6 UDPipe API Reference

The UDPipe API is defined in header `udpipe.h` and resides in `ufal::udpipe` namespace. The API allows only using existing models, for custom model creation you have to use the `train_parser` binary.

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

6.1 UDPipe Versioning

UDPipe 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 UDPipe version with the same *major*.
- If two versions differ in *major*, their API may differ in any way.

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

```
class token {
 public:
  string form;
  string misc;

  token(string_piece form = string_piece(), string_piece misc = string_piece());

  // CoNLL-U defined SpaceAfter=No feature
  bool get_space_after() const;
  void set_space_after(bool space_after);

  // UDPipe-specific all-spaces-preserving SpacesBefore and SpacesAfter features
  void get_spaces_before(string& spaces_before) const;
  void set_spaces_before(string_piece spaces_before);
  void get_spaces_after(string& spaces_after) const;
  void set_spaces_after(string_piece spaces_after);
  void get_spaces_in_token(string& spaces_in_token) const;
  void set_spaces_in_token(string_piece spaces_in_token);

  // UDPipe-specific TokenRange feature
  bool get_token_range(size_t& start, size_t& end) const;
  void set_token_range(size_t start, size_t end);
};
```

The `token` class represents a sentence token, with `form` and `misc` fields corresponding to CoNLL-U fields. The `token` class acts mostly as a parent to `word` and `multiword_token` classes.

The class also offers several methods for manipulating features in the `misc` field. Notably, UDPipe uses custom `misc` fields to store all spaces in the original document. This markup is backward compatible with CoNLL-U v2 `SpaceAfter=No` feature. This markup can be utilized by `plaintext` output format, which allows reconstructing the original document.

The markup uses the following `misc` fields:

- `SpacesBefore=content` (by default empty): spaces/other content preceding the token
- `SpacesAfter=content` (by default a space if `SpaceAfter=No` feature is not present, empty otherwise): spaces/other content following the token
- `SpacesInToken=content` (by default equal to the FORM of the token): FORM of the token including original spaces (this is needed only if tokens are allowed to contain spaces and a token contains a tab or newline characters)

The `content` of all above three fields must be escaped to allow storing tabs and newlines. The following C-like schema is used:

- `\s`: space
- `\t`: tab
- `\r`: CR character
- `\n`: LF character
- `\p`: | (pipe character)
- `\\`: \ (backslash character)

6.3.1 token::get_space_after()

```
bool get_space_after() const;
```

Returns `true` if the token should be followed by a spaces, `false` if not, according to the absence or presence of the `SpaceAfter=No` feature in the `misc` field.

6.3.2 token::set_space_after()

```
void set_space_after(bool space_after);
```

Adds or removes the `SpaceAfter=No` feature in the `misc` field.

6.3.3 token::get_spaces_before()

```
void get_spaces_before(string& spaces_before) const;
```

Return spaces preceding current token, stored in the `SpacesBefore` feature in the `misc` field. If `SpacesBefore` is not present, empty string is returned.

6.3.4 token::set_spaces_before()

```
void set_spaces_before(string_piece spaces_before);
```

Set the `SpacesBefore` feature in the `misc` field.

6.3.5 token::get_spaces_after()

```
void get_spaces_after(string& spaces_after) const;
```

Return spaces after current token, stored in the `SpacesAfter` feature in the `misc` field.

If `SpacesAfter` is not present and `SpaceAfter=No` is present, return an empty string; if neither feature is present, one space is returned.

6.3.6 token::set_spaces_after()

```
void set_spaces_after(string_piece spaces_after);
```

Set the `SpacesAfter` and `SpaceAfter=No` features in the `misc` field.

6.3.7 token::get_spaces_in_token()

```
void get_spaces_in_token(string& spaces_in_token) const;
```

Return the value of the `SpacesInToken` feature, if present. Otherwise, empty string is returned.

6.3.8 token::set_spaces_in_token()

```
void set_spaces_in_token(string_piece spaces_in_token);
```

Set the `SpacesInToken` feature in the `misc` field.

6.3.9 token::get_token_range()

```
bool get_token_range(size_t& start, size_t& end) const;
```

If present, return the value of the `TokenRange` feature in the `misc` field. The format of the feature (inspired by Python) is `TokenRange=start:end`, where `start` is zero-based document-level index of the start of the token (counted in Unicode characters) and `end` is zero-based document-level index of the first character following the token (i.e., the length of the token is `end-start`).

6.3.10 token::set_token_range()

```
void set_token_range(size_t start, size_t end);
```

Set the `TokenRange` feature in the `misc` field. If `string::npos` is passed in the `start` argument, `TokenRange` feature is removed from the `misc` field.

6.4 Class word

```
class word : public token {
 public:
  // form and misc are inherited from token
  int id;         // 0 is root, >0 is sentence word, <0 is undefined
  string lemma;   // lemma
  string upostag; // universal part-of-speech tag
  string xpostag; // language-specific part-of-speech tag
  string feats;   // list of morphological features
  int head;       // head, 0 is root, <0 is undefined
  string deprel;  // dependency relation to the head
  string deps;    // secondary dependencies

  vector<int> children;

  word(int id = -1, string_piece form = string_piece());
};
```

The word class represents a sentence word. The word fields correspond to CoNLL-U fields, with the children field representing the opposite direction of head links (the elements of the children array are in ascending order).

6.5 Class multiword_token

```
class multiword_token : public token {
 public:
  // form and misc are inherited from token
  int id_first, id_last;

  multiword_token(int id_first = -1, int id_last = -1, string_piece form = string_piece(),
      string_piece misc = string_piece());
};
```

The multiword_token represents a multi-word token described in CoNLL-U format. The multi-word token has a form and a misc field, other CoNLL-U word fields are guaranteed to be empty.

6.6 Class empty_node

```
class empty_node {
 public:
  int id;          // 0 is root, >0 is sentence word, <0 is undefined
  int index;       // index for the current id, should be numbered from 1, 0=undefined
  string form;     // form
  string lemma;    // lemma
  string upostag; // universal part-of-speech tag
  string xpostag; // language-specific part-of-speech tag
  string feats;    // list of morphological features
  string deps;     // secondary dependencies
  string misc;     // miscellaneous information

  empty_node(int id = -1, int index = 0) : id(id), index(index) {}
};
```

The empty_node class represents an empty node from CoNLL-U 2.0, with the fields corresponding to CoNLL-U fields. For a specified id, the index are numbered sequentially from 1.

6.7 Class sentence

```
class sentence {
 public:
  sentence();

  vector<word> words;
  vector<multiword_token> multiword_tokens;
  vector<empty_node> empty_nodes;
  vector<string> comments;
  static const string root_form;

  // Basic sentence modifications
  bool empty();
  void clear();
  word& add_word(string_piece form = string_piece());
  void set_head(int id, int head, const string& deprel);
  void unlink_all_words();

  // CoNLL-U defined comments
  bool get_new_doc(string* id = nullptr) const;
  void set_new_doc(bool new_doc, string_piece id = string_piece());
  bool get_new_par(string* id = nullptr) const;
  void set_new_par(bool new_par, string_piece id = string_piece());
  bool get_sent_id(string& id) const;
  void set_sent_id(string_piece id);
  bool get_text(string& text) const;
  void set_text(string_piece text);
};
```

The `sentence` class represents a sentence CoNLL-U sentence, which consists of:

- sequence of `word`s stored in ascending order, with the first word (with index 0) always being a technical root with form `root_form`
- sequence of `multiword_token`s also stored in ascending order
- sequence of `empty_node`s also stored in ascending order
- comments

Although you can manipulate the `words` directly, the `sentence` class offers several simple node manipulation methods. There are also several methods manipulating CoNLL-U v2 comments.

6.7.1 sentence::empty()

```
bool empty();
```

Returns `true` if the sentence is empty. i.e., if it contains only a technical root node.

6.7.2 sentence::clear()

```
void clear();
```

Removes all words, multi-word tokens and comments (only the technical root `word` is kept).

6.7.3 sentence::add_word()

```
word& add_word(string_piece form = string_piece());
```

Adds a new word to the sentence. The new word has first unused `id`, specified `form` and is not linked to any other node. Reference to the new word is returned so that other fields can be also filled.

6.7.4 sentence:set_head()

```
void set_head(int id, int head, const std::string& deprel);
```

Link the word `id` to the word `head`, with the specified dependency relation. If the `head` is negative, the word `id` is unlinked from its current head, if any.

6.7.5 sentence::unlink_all_words()

```
void unlink_all_words();
```

Unlink all words.

6.7.6 sentence::get_new_doc()

```
bool get_new_doc(string* id = nullptr) const;
```

Return `true` if `# newdoc` comment is present. Optionally, document id is also returned (in `# newdoc id = ...` format).

6.7.7 sentence::set_new_doc()

```
void set_new_doc(bool new_doc, string_piece id = string_piece());
```

Adds/removes `# newdoc` comment, optionally with a given document id.

6.7.8 sentence::get_new_par()

```
bool get_new_par(string* id = nullptr) const;
```

Return `true` if `# newpar` comment is present. Optionally, paragraph id is also returned (in `# newpar id = ...` format).

6.7.9 sentence::set_new_par()

```
void set_new_par(bool new_par, string_piece id = string_piece());
```

Adds/removes `# newpar` comment, optionally with a given paragraph id.

6.7.10 sentence::get_sent_id()

```
bool get_sent_id(string& id) const;
```

Return `true` if `# sent_id = ...` comment is present, and fill given `id` with sentence id. Otherwise, return `false` and clear `id`.

6.7.11 sentence::set_sent_id()

```
void set_sent_id(string_piece id);
```

Set the `# sent_id = ...` comment using given sentence id; if the sentence id is empty, remove all present `# sent_id` comment.

6.7.12 sentence::get_text()

```
bool get_text(string& text) const;
```

Return `true` if `# text = ...` comment is present, and fill given `text` with sentence text. Otherwise, return `false` and clear `text`.

6.7.13 sentence::set_text()

```
void set_text(string_piece text);
```

Set the `# text = ...` comment using given text; if the given text is empty, remove all present `# text` comment.

6.8 Class input_format

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

  virtual bool read_block(istream& is, string& block) const = 0;
  virtual void reset_document(string_piece id = string_piece()) = 0;
  virtual void set_text(string_piece text, bool make_copy = false) = 0;
  virtual bool next_sentence(sentence& s, string& error) = 0;

  // Static factory methods
  static input_format* new_input_format(const string& name);
  static input_format* new_conllu_input_format(const string& options = std::string());
  static input_format* new_generic_tokenizer_input_format(const string& options =
      std::string());
  static input_format* new_horizontal_input_format(const string& options = std::string());
  static input_format* new_vertical_input_format(const string& options = std::string());

  static input_format* new_presegmented_tokenizer(input_format* tokenizer);

  static const string CONLLU_V1;
  static const string CONLLU_V2;
  static const string GENERIC_TOKENIZER_NORMALIZED_SPACES;
  static const string GENERIC_TOKENIZER_PRESEGMENTED;
  static const string GENERIC_TOKENIZER_RANGES;
};
```

The `input_format` class allows loading sentences in various formats.

Th class instances may store internal state and are not thread-safe.

6.8.1 input_format::read_block()

```
virtual bool read_block(istream& is, string& block) const = 0;
```

Read a portion of input, which is guaranteed to contain only complete sentences. Such portion is usually a paragraph (text followed by an empty line) or a line, but it may be more complex (i.e., in a XML-like format).

6.8.2 input_format::reset_document()

```
virtual void reset_document(string_piece id = string_piece()) = 0;
```

Resets the `input_format` instance state. Such state is needed not only for remembering unprocessed text of the last `set_text` call, but also for correct inter-block state tracking (for example to track document-level ranges or inter-sentence spaces – if you pass only spaces to `set_text`, these spaces has to accumulate and be returned as preceding spaces of the next sentence).

If applicable, first read sentence will have the `# newdoc` comment, optionally with given document id.

6.8.3 input_format::set_text()

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

Set the text from which the sentences will be read.

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 instance is destroyed or `set_text` is called again. If `make_copy` is `true`, a copy of the given text is made and retained until the instance is destroyed or `set_text` is called again.

6.8.4 input_format::next_sentence()

```
virtual bool next_sentence(sentence& s, string& error) = 0;
```

Try reading another sentence from the text specified by `set_text`. Returns `true` if the sentence was read and `false` if the text ended or there was a read error. The latter two conditions can be distinguished by the `error` parameter – if it is empty, the text ended, if it is nonempty, it contains a description of the read error.

6.8.5 input_format::new_input_format()

```
static input_format* new_input_format(const string& name);
```

Create new `input_format` instance, given its name. The individual input formats can be parametrized by using `format=data` syntax. The following input formats are currently supported:

- `conllu`: return the `new_conllu_input_format`
- `generic_tokenizer`: return the `new_generic_tokenizer_input_format`

- `horizontal`: return the `new_horizontal_input_format`
- `vertical`: return the `new_vertical_input_format`

The new instance must be deleted after use.

6.8.6 input_format::new_conllu_input_format()

```
static input_format* new_conllu_input_format(const string() options = std::string());
```

Create `input_format` instance which loads sentences in the CoNLL-U format. The new instance must be deleted after use.

Supported options:
- `v2` (default): use CoNLL-U v2
- `v1`: allow loading only CoNLL-U v1 (i.e., no empty nodes and no spaces in forms and lemmas)

6.8.7 input_format::new_generic_tokenizer_input_format()

```
static input_format* new_generic_tokenizer_input_format(const string() options =
    std::string());
```

Create rule-based generic tokenizer for English-like languages (with spaces separating tokens and English-like punctuation). The new instance must be deleted after use.

Supported options:
- `normalized_spaces`: by default, UDPipe uses custom `misc` fields to exactly encode spaces in the original document. If `normalized_spaces` option is given, only standard CoNLL-U v2 markup (`SpaceAfter=No` and `# newpar`) is used.
- `presegmented`: input is assumed to be already segmented, with every sentence on a line, and is only tokenized (respecting sentence breaks)
- `ranges`: for every token, range in the original document is stored in a format described in `token` class

6.8.8 input_format::new_horizontal_input_format()

```
static input_format* new_horizontal_input_format(const string() options = std::string());
```

Create `input_format` instance which loads forms from a simple horizontal format – each sentence on a line, with word forms separated by spaces. The new instance must be deleted after use.

In order to allow spaces in tokens, Unicode character 'NO-BREAK SPACE' (U+00A0) is considered part of token and converted to a space during loading.

6.8.9 input_format::new_vertical_input_format()

```
static input_format* new_vertical_input_format(const string() options = std::string());
```

Create `input_format` instance which loads forms from a simple vertical format – each word on a line, with empty line denoting end of sentence. The new instance must be deleted after use.

6.8.10 input_format::new_presegmented_tokenizer()

```
static input_format* new_presegmented_tokenizer(input_format* tokenizer);
```

Create `input_format` instance which acts as a tokenizer adapter – given a tokenizer which segments anywhere, it creates a tokenizer which segments on newline characters (by calling the tokenizer on individual lines, and if the tokenizer segments in the middle of the line, it calls it repeatedly and merges the results).

The new instance must be deleted after use. Note that the new instance *takes ownership* of the given tokenizer and *deletes* it during its own deletion.

6.9 Class output_format

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

  virtual void write_sentence(const sentence& s, ostream& os) = 0;
  virtual void finish_document(ostream& os) {};

  // Static factory methods
  static output_format* new_output_format(const string& name);
  static output_format* new_conllu_output_format(const string() options = std::string());
  static output_format* new_epe_output_format(const string() options = std::string());
  static output_format* new_matxin_output_format(const string() options = std::string());
  static output_format* new_horizontal_output_format(const string() options =
      std::string());
  static output_format* new_plaintext_output_format(const string() options = std::string());
  static output_format* new_vertical_output_format(const string() options = std::string());

  static const string CONLLU_V1;
  static const string CONLLU_V2;
  static const string HORIZONTAL_PARAGRAPHS;
  static const string PLAINTEXT_NORMALIZED_SPACES;
  static const string VERTICAL_PARAGRAPHS;
};
```

The output_format class allows printing sentences in various formats.

The class instances may store internal state and are not thread-safe.

6.9.1 output_format::write_sentence()

```
virtual void write_sentence(const sentence& s, ostream& os) = 0;
```

Write given sentence to the given output stream.

When the output format requires document-level markup, it is written automatically when the first sentence is written using this output_format instance (or after finish_document call).

6.9.2 output_format::finish_document()

```
virtual void finish_document(ostream& os) {};
```

When the output format requires document-level markup, write the end-of-document mark and reset the output_format instance state (i.e., the next write_sentence will write start-of-document mark).

6.9.3 output_format::new_output_format()

```
static output_format* new_output_format(const string& name);
```

Create new output_format instance, given its name. The following output formats are currently supported:

- conllu: return the new_conllu_output_format
- epe: return the new_epe_output_format
- matxin: return the new_matxin_output_format
- horizontal: return the new_horizontal_output_format
- plaintext: return the new_plaintext_output_format
- vertical: return the new_vertical_output_format

The new instance must be deleted after use.

6.9.4 output_format::new_conllu_output_format()

```
static output_format* new_conllu_output_format(const string() options = std::string());
```

Creates output_format instance for writing sentences in the CoNLL-U format. The new instance must be deleted after use.

Supported options:
- v2 (default): use CoNLL-U v2
- v1: produce output in CoNLL-U v1 format. Note that this is a lossy process, as empty nodes are ignored and spaces in forms and lemmas are converted to underscores.

6.9.5 output_format::new_epe_output_format()

```
static output_format* new_epe_output_format(const string() options = std::string());
```

Creates output_format instance for writing sentences in the EPE (Extrinsic Parser Evaluation 2017) interchange format. The new instance must be deleted after use.

6.9.6 output_format::new_matxin_output_format()

```
static output_format* new_matxin_output_format(const string() options = std::string());
```

Creates output_format instance for writing sentences in the Matxin format – UDPipe produces a XML with the following DTD:

```
<!ELEMENT    corpus     (SENTENCE*)>
<!ELEMENT    SENTENCE   (NODE*)>
<!ATTLIST    SENTENCE    ord           CDATA         #REQUIRED
                         alloc         CDATA         #REQUIRED>
<!ELEMENT    NODE    (NODE*)>
<!ATTLIST    NODE        ord           CDATA         #REQUIRED
                         alloc         CDATA         #REQUIRED
                         form          CDATA         #REQUIRED
                         lem           CDATA         #REQUIRED
                         mi            CDATA         #REQUIRED
                         si            CDATA         #REQUIRED
                         sub           CDATA         #REQUIRED>
```

The new instance must be deleted after use.

6.9.7 output_format::new_plaintext_output_format()

```
static output_format* new_plaintext_output_format(const string() options = std::string());
```

Creates output_format instance for writing sentence *tokens* (in the UD sense) using original spacing. By default, UDPipe custom misc features (see description of token class) are used to reconstruct the exact original spaces. However, if the document does not contain these features or if only normalized spacing is wanted, you can use the following option:
- normalized_spaces: write one sentence on a line, and either one or no space between tokens, using the SpaceAfter=No feature

6.9.8 output_format::new_horizontal_output_format()

```
static output_format* new_horizontal_output_format(const string() options = std::string());
```

Creates output_format instance for writing sentences in a simple horizontal format – each sentence on a line, with word forms separated by spaces. The new instance must be deleted after use.

Because words can contain spaces in CoNLL-U v2, the spaces in words are converted to Unicode character 'NO-BREAK SPACE' (U+00A0).

Supported options:
- `paragraphs`: if given, an empty line is printed after the end of a paragraph or a document (recognized by `# newpar` or `# newdoc` comments)

6.9.9 output_format::new_vertical_output_format()

```
static output_format* new_vertical_output_format(const string() options = std::string());
```

Creates output_format instance for writing sentences in a simple vertical format – each word form on a line, with empty line denoting end of sentence. The new instance must be deleted after use.

Supported options:
- `paragraphs`: if given, an empty line is printed after the end of a paragraph or a document (recognized by `# newpar` or `# newdoc` comments)

6.10 Class model

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

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

  virtual input_format* new_tokenizer(const string& options) const = 0;
  virtual bool tag(sentence& s, const string& options, string& error) const = 0;
  virtual bool parse(sentence& s, const string& options, string& error) const = 0;

  static const string DEFAULT;
  static const string TOKENIZER_NORMALIZED_SPACES;
  static const string TOKENIZER_PRESEGMENTED;
  static const string TOKENIZER_RANGES;
};
```

Class representing UDPipe model, allowing to perform tokenization, tagging and parsing.

6.10.1 model::load(const char*)

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

Load a new model from a given file, returning `NULL` on failure. The new instance must be deleted after use.

6.10.2 model::load(istream&)

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

Load a new model from a given input stream, returning `NULL` on failure. The new instance must be deleted after use.

6.10.3 model::new_tokenizer()

```
virtual input_format* new_tokenizer(const string& options) const = 0;
```

Construct a new tokenizer (or `NULL` if no tokenizer is specified by the model). The new instance must be deleted after use.

6.10.4 model::tag()

```
virtual bool tag(sentence& s, const string& options, string& error) const = 0;
```

Tag the given sentence.

6.10.5 model::parse()

```
virtual bool parse(sentence& s, const string& options, string& error) const = 0;
```

Parse the given sentence.

6.11 Class pipeline

```
class pipeline {
 public:
  pipeline(const model* m, const string& input, const string& tagger, const string& parser,
      const string& output);

  void set_model(const model* m);
  void set_input(const string& input);
  void set_tagger(const string& tagger);
  void set_parser(const string& parser);
  void set_output(const string& output);

  void set_immediate(bool immediate);
  void [set_document_id #pipeline_set_document_id[(const string& document_id);

  bool process(istream& is, ostream& os, string& error) const;

  static const string DEFAULT;
  static const string NONE;
};
```

The `pipeline` class allows simple file-to-file processing. A model and input/tagger/parser/output options can be specified in the pipeline.

The input file can be processed either after fully loaded (default), or in immediate mode, in which case is the input processed and printed as soon as a block of input guaranteed to contain whole sentences is loaded. Specifically, for most input formats the input is processed after loading an empty line (with the exception of `horizontal` input format and `presegmented` tokenizer, where the input is processed after loading every line).

6.11.1 pipeline::set_model()

```
void set_model(const model* m);
```

Use the given model.

6.11.2 pipeline::set_input()

```
void set_input(const string& input);
```

Use the given input format. In addition to formats described in `new_input_format`, a special `tokenizer` or `tokenizer=options` format allows using the model tokenizer.

6.11.3 pipeline::set_tagger()

```
void set_tagger(const string& tagger);
```

Use the given tagger options.

6.11.4 pipeline::set_parser()

```
void set_parser(const string& parser);
```

Use the given parser options.

6.11.5 pipeline::set_output()

```
void set_output(const string& output);
```

Use the given output format (see `new_output_format` for a list).

6.11.6 pipeline::set_immediate()

```
void set_immediate(bool immediate);
```

Set or reset the immediate mode (default is `immediate=false`).

6.11.7 pipeline::set_document_id()

```
void set_document_id(const string& document_id);
```

Set document id, which is passed to `input_format::reset_document`).

6.11.8 pipeline::process()

```
bool process(istream& is, ostream& os, string& error) const;
```

Process the given input stream, writing results to the given output stream. If the processing succeeded, `true` is returned; otherwise, `false` is returned with an error stored in the `error` argument.

6.12 Class trainer

```
class trainer {
 public:
  static bool train(const string& method, const vector<sentence>& train, const
      vector<sentence>& heldout,
                    const string& tokenizer, const string& tagger, const string& parser,
                    ostream& os, string& error);

  static const string DEFAULT;
  static const string NONE;
};
```

Class allowing training a UDPipe model.

6.12.1 trainer::train()

```
static bool train(const string& method, const vector<sentence>& train, const
    vector<sentence>& heldout,
                  const string& tokenizer, const string& tagger, const string& parser,
                  ostream& os, string& error);
```

Train a UDPipe model. The only supported method is currently `morphodita_parsito`. Use the supplied train and heldout data, and given tokenizer, tagger and parser options (see the Training UDPipe Models section in the User's Manual).

If the training succeeded, `true` is returned and the model is saved to the given `os` stream; otherwise, `false` is returned with an error stored in the `error` argument.

6.13 Class evaluator

```
class evaluator {
 public:
  evaluator(const model* m, const string& tokenizer, const string& tagger, const string&
      parser);

  void set_model(const model* m);
  void set_tokenizer(const string& tokenizer);
  void set_tagger(const string& tagger);
  void set_parser(const string& parser);

  bool evaluate(istream& is, ostream& os, string& error) const;

  static const string DEFAULT;
  static const string NONE;
};
```

Class evaluating performance of given model on CoNLL-U file.

Three different settings (depending on whether tokenizer, tagger and parser is used) can be evaluated. For details, see Measuring Model Accuracy in User's Manual.

6.13.1 evaluator::set_model()

```
void set_model(const model* m);
```

Use the given model.

6.13.2 evaluator::set_tokenizer()

```
void set_tokenizer(const string& tokenizer);
```

Use the given tokenizer options; pass `DEFAULT` to use default options or `NONE` not to use a tokenizer.

6.13.3 evaluator::set_tagger()

```
void set_tagger(const string& tagger);
```

Use the given tagger options; pass `DEFAULT` to use default options or `NONE` not to use a tagger.

6.13.4 evaluator::set_parser()

```
void set_parser(const string& parser);
```

Use the given parser options; pass `DEFAULT` to use default options or `NONE` not to use a parser.

6.13.5 evaluator::evaluate()

```
bool evaluate(istream& is, ostream& os, string& error) const;
```

Evaluate the specified model on the given CoNLL-U input read from `is` stream.

If the evaluation succeeded, `true` is returned and the evaluation results are written to the `os` stream in a plain text format; otherwise, `false` is returned with an error stored in the `error` argument.

6.14 Class version

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

  static version current();
};
```

The `version` class represents UDPipe version. See UDPipe Versioning for more information.

6.14.1 version::current

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

Returns current UDPipe version.

6.15 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:

6.15.1 Helper Structures

```
typedef vector<int> Children;

typedef vector<string> Comments;

class ProcessingError {
public:
  bool occurred();
  string message;
};

class Token {
 public:
  string form;
  string misc;

  Token(const string& form = string(), const string& misc = string());

  // CoNLL-U defined SpaceAfter=No feature
  bool getSpaceAfter() const;
  void setSpaceAfter(bool space_after);

  // UDPipe-specific all-spaces-preserving SpacesBefore and SpacesAfter features
  string getSpacesBefore() const;
  void setSpacesBefore(const string& spaces_before);
  string getSpacesAfter() const;
  void setSpacesAfter(const string& spaces_after);
  string getSpacesInToken() const;
  void setSpacesInToken(const string& spaces_in_token);

  // UDPipe-specific TokenRange feature
  bool getTokenRange() const;
  size_t getTokenRangeStart() const;
  size_t getTokenRangeEnd() const;
  void setTokenRange(size_t start, size_t end);
};

class Word : public Token {
 public:
  // form and misc are inherited from token
  int id;         // 0 is root, >0 is sentence word, <0 is undefined
  string lemma;   // lemma
  string upostag; // universal part-of-speech tag
  string xpostag; // language-specific part-of-speech tag
  string feats;   // list of morphological features
  int head;       // head, 0 is root, <0 is undefined
  string deprel;  // dependency relation to the head
  string deps;    // secondary dependencies

  Children children;

  Word(int id = -1, const string& form = string());
};
typedef vector<Word> Words;

class MultiwordToken : public Token {
 public:
  // form and misc are inherited from token
  int idFirst, idLast;

  MultiwordToken(int id_first = -1, int id_last = -1, const string& form = string(), const
      string& misc = string());
};
typedef vector<MultiwordToken> MultiwordTokens;

class EmptyNode {
 public:
  int id;          // 0 is root, >0 is sentence word, <0 is undefined
  int index;       // index for the current id, should be numbered from 1, 0=undefined
  string form;     // form
  string lemma;    // lemma
  string upostag;  // universal part-of-speech tag
  string xpostag;  // language-specific part-of-speech tag
  string feats;    // list of morphological features
  string deps;     // secondary dependencies
  string misc;     // miscellaneous information

  EmptyNode(int id = -1, int index = 0) : id(id), index(index) {}
};
typedef vector<empty_node> EmptyNodes;

class Sentence {
 public:
  Sentence();

  Words words;
  MultiwordTokens multiwordTokens;
  EmptyNodes emptyNodes;
  Comments comments;
  static const string rootForm;

  // Basic sentence modifications
  bool empty();
  void clear();
  virtual Word& addWord(const char* form);
  void setHead(int id, int head, const string& deprel);
  void unlinkAllWords();

  // CoNLL-U defined comments
  bool getNewDoc() const;
  string getNewDocId() const;
  void setNewDoc(bool new_doc, const string& id = string());
  bool getNewPar() const;
  string getNewParId() const;
  void setNewPar(bool new_par, const string& id = string());

  string getSentId() const;
  void setSentId(const string& id);
  string getText() const;
  void setText(const string& id);
};
typedef vector<Sentence> Sentences;
```

6.15.2 Main Classes

```
class InputFormat {
 public:
  virtual void resetDocument(const string& id = string());
  virtual void setText(const char* text);
  virtual bool nextSentence(Sentence& s, ProcessingError* error = nullptr);

  static InputFormat* newInputFormat(const string& name);
  static InputFormat* newConlluInputFormat(const string& id = string());
  static InputFormat* newGenericTokenizerInputFormat(const string& id = string());
  static InputFormat* newHorizontalInputFormat(const string& id = string());
  static InputFormat* newVerticalInputFormat(const string& id = string());

  static InputFormat* newPresegmentedTokenizer(InputFormat tokenizer);

  static const string CONLLU_V1;
  static const string CONLLU_V2;
  static const string GENERIC_TOKENIZER_NORMALIZED_SPACES;
  static const string GENERIC_TOKENIZER_PRESEGMENTED;
  static const string GENERIC_TOKENIZER_RANGES;
};

class OutputFormat {
 public:
  virtual string writeSentence(const Sentence& s);
  virtual string finishDocument();

  static OutputFormat* newOutputFormat(const string& name);
  static OutputFormat* newConlluOutputFormat(const string& options = string());
  static OutputFormat* newEpeOutputFormat(const string& options = string());
  static OutputFormat* newMatxinOutputFormat(const string& options = string());
  static OutputFormat* newHorizontalOutputFormat(const string& options = string());
  static OutputFormat* newPlaintextOutputFormat(const string& options = string());
  static OutputFormat* newVerticalOutputFormat(const string& options = string());

  static const string CONLLU_V1;
  static const string CONLLU_V2;
  static const string HORIZONTAL_PARAGRAPHS;
  static const string PLAINTEXT_NORMALIZED_SPACES;
  static const string VERTICAL_PARAGRAPHS;
};
```

```
class Model {
 public:
  static Model* load(const char* fname);

  virtual InputFormat* newTokenizer(const string& options) const;
  virtual bool tag(Sentence& s, const string& options, ProcessingError* error = nullptr)
      const;
  virtual bool parse(Sentence& s, const string& options, ProcessingError* error) const;

  static const string DEFAULT;
  static const string TOKENIZER_PRESEGMENTED;
};

class Pipeline {
 public:
  Pipeline(const Model* m, const string& input, const string& tagger, const string& parser,
      const string& output);

  void setModel(const Model* m);
  void setInput(const string& input);
  void setTagger(const string& tagger);
  void setParser(const string& parser);
  void setOutput(const string& output);

  void setImmediate(bool immediate);
  void setDocumentId(const string& document_id);

  string process(const string& data, ProcessingError* error = nullptr) const;

  static const string DEFAULT;
  static const string NONE;
};

class Trainer {
 public:

  static string train(const string& method, const Sentences& train, const Sentences&
      heldout,
                      const string& tokenizer, const string& tagger, const string& parser,
                      ProcessingError* error = nullptr);

  static const string DEFAULT;
  static const string NONE;
};

class Evaluator {
 public:
  Evaluator(const Model* m, const string& tokenizer, const string& tagger, const string&
      parser);

  void setModel(const Model* m);
  void setTokenizer(const string& tokenizer);
  void setTagger(const string& tagger);
  void setParser(const string& parser);

  string evaluate(const string& data, ProcessingError* error = nullptr) const;

  static const string DEFAULT;
  static const string NONE;
};
```

```
class Version {
 public:
  unsigned major;
  unsigned minor;
  unsigned patch;
  string prerelease;

  // Returns current version.
  static version current();
};
```

6.16 C# Bindings

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

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

6.17 Java Bindings

UDPipe library bindings is available in the `cz.cuni.mff.ufal.udpipe` package.

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

The bindings require native C++ library `libudpipe_java` (called `udpipe_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 `udpipe_java.setLibraryPath(String path)` call (before the first call inside the C++ library, of course).

6.18 Perl Bindings

UDPipe library bindings is available in the `Ufal::UDPipe` 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::UDPipe::Words` for reference. Static methods and enumerations are available only through the module, not through object instance.

6.19 Python Bindings

UDPipe library bindings is available in the `ufal.udpipe` 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

UDPipe website.

UDPipe 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 UDPipe User's Manual.

8.1 Publications

- (Straka et al. 2017) Milan Straka and Jana Straková. *Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe*. In Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, Vancouver, Canada, August 2017.
- (Straka et al. 2016) Straka Milan, Hajič Jan, Straková Jana. *UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing*. In Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016), Portorož, Slovenia, May 2016.

8.2 Bibtex for Referencing

```
@InProceedings{udpipe:2017,
  author    = {Straka, Milan  and  Strakov\'{a}, Jana},
  title     = {Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe},
  booktitle = {Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw
      Text to Universal Dependencies},
  month     = {August},
  year      = {2017},
  address   = {Vancouver, Canada},
  publisher = {Association for Computational Linguistics},
  pages     = {88--99},
  url       = {http://www.aclweb.org/anthology/K/K17/K17-3009.pdf}
}
```

8.3 Persistent Identifier

If you prefer to reference UDPipe by a persistent identifier (PID), you can use `http://hdl.handle.net/11234/1-1702`.