UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing

Milan Straka, Jan Hajič, Jana Straková
Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics
`{straka,hajic,strakova}@ufal.mff.cuni.cz`

Abstract

Automatic natural language processing of large texts often presents recurring challenges in multiple languages: even for most advanced tasks, the texts are first processed by basic processing steps – from tokenization to parsing. We present an extremely simple-to-use tool consisting of one binary and one model (per language), which performs these tasks for multiple languages without the need for any other external data. *UDPipe*, a pipeline processing CoNLL-U-formatted files, performs tokenization, morphological analysis, part-of-speech tagging, lemmatization and dependency parsing for nearly all treebanks of Universal Dependencies 1.2 (namely, the whole pipeline is currently available for 32 out of 37 treebanks). In addition, the pipeline is easily trainable with training data in CoNLL-U format (and in some cases also with additional raw corpora) and requires minimal linguistic knowledge on the users' part. The training code is also released.

Keywords: Universal Dependencies; dependency parsing; part-of-speech tagging

1. Introduction

The Universal Dependencies project (Nivre et al., 2016) seeks to develop cross-linguistically consistent treebank annotation for many languages. The annotation scheme is based on the universal Stanford dependencies (de Marneffe et al., 2014), the Google universal part-of-speech tags (Petrov et al., 2012), and the Interset interlingua for morphosyntactic features (Zeman, 2008). The latest version of UD (Univeral Dependencies Treebanks version 1.2, 2015)[1] consists of 37 dependency treebanks.

Our aim is to create a simple-to-use, one-binary and a single-model tool (per language), to easily process raw text to CoNLL-U-formatted tagged and/or parsed dependency trees (with morphological features if available in UD).

In particular, our goals are

- state-of-the-art tools for tokenization, morphological analysis, part-of-speech tagging and dependency parsing,
- free C++ tools available under the Mozilla Public License (MPL) 2.0 license (code) and CC BY-NC-SA 4.0 license (models),
- simple tool with a single model (per language), easily trainable with custom data (CoNLL-U format),
- trained models provided for as many UD treebanks as possible,
- no feature engineering, no external morphological dictionary, no language specific knowledge,
- efficient programming design in terms of RAM and disc usage.

Following these requirements, we developed a morphological dictionary tool and tagger software, which is described in Section 4. and in detail in (Straková et al., 2014), and a dependency parser described in Section 5. and in detail in (Straka et al., 2015). We wrapped these tools along with a trainable tokenizer based on artificial neural networks (Section 3., which we describe in a little more detail since it has not been published previously) in a single tool called *UDPipe*, described in Section 6. We also compare our set of tools contained in the *UDPipe* with other results, expecially other dependency parsers.

[1] `http://hdl.handle.net/11234/1-1548`

2. Related Work

A considerable number of natural language processing pipelines are available, e.g. OpenNLP[2] or Natural Language Processing Toolkit (NLTK), (Bird et al., 2009)[3]; however, our aim was to develop an extremely simple tool to be easily used by users with no language specific knowledge and little interest in programming, a tool with clear licensing, no feature engineering, no morphosyntactic dictionary or other additional resources necessary, with a trainable tokenizer (which is, surprisingly, an underestimated problem) and above- or near state-of-the-art results.

3. Tokenization

Tokenization is usually considered a trivial task for many languages, notably for those that use separators between words. Nevertheless, treebanks in many languages use language-specific or even treebank-specific rules to partition unseparated words or replace contractions, for example, in English, `isn't` is usually tokenized as `is n't`.

Obviously, it is crucial to perform exactly the same tokenization on a text to be processed by a tool trained on such a (idiosyncratically tokenized) treebank. The requirement for identical tokenization applies also for additional language resources used for additional or different methods (e.g., when using word embeddings).

Although the tokenization of every treebank can usually be described using only a limited set of rules, developing a rule-based tokenizer for several dozens of treebanks appears to be a very demanding task.

[2] `https://opennlp.apache.org`
[3] `http://nltk.org`

We describe a tokenizer which is trained solely by comparing original (raw) and tokenized text, without any additional knowledge about the language. The tokenizer learns both where to split words even without space separators, and also where to split sentences.

If the original raw text is not available, which is quite common, we suggest a method of reconstructing plain text of given tokenized corpus by utilizing an additional raw text corpus.

3.1. Tokenization for Universal Dependencies

The CoNLL-U format, used by UD treebanks, allows reconstruction of the original pre-tokenized text using the `SpaceAfter=No` feature: it signals that a given token was not followed by a space separator in the original text. In UD 1.2, there are five treebanks using the `SpaceAfter` feature.[4]

Additionally, the so-called *multi-word tokens* in the CoNLL-U format allow to capture various contractions – for example, one can describe that the input token `im` in German represents `in dem`. Note that recognizing multi-word tokens is quite different from tokenization – the multi-word token is usually not just a simple string concatenation. Therefore, the described tokenizer does not handle multi-word tokens.[5]

3.2. The Tokenizer (and Segmentation) Model

Given a plain text, the tokenizer locates both token and sentence boundaries at the same time. We assume that a (white)space character always acts as a token separator, and we train the tokenizer to locate only the token boundaries which arise between two non-space characters.

The tokenizer is based on a bidirectional LSTM artificial neural network (Graves and Schmidhuber, 2005). Long short-term memory (LSTM) unit computes the state sequence $h_1, \ldots, h_n$ given the input vector $x_1, \ldots, x_n$. It was designed to be able to capture non-linear and non-local dynamics in sequences (Hochreiter and Schmidhuber, 1997) and has been used to obtain several state-of-the-art results in sequence classification, for example part-of-speech tagging (Ling et al., 2015). Recently, a gated linear unit (GRU) was proposed by Cho et al. (2014) as an alternative to LSTM, and was shown to have similar performance, while being less computationally demanding.

The inputs of the proposed tokenizer are fixed-length *segments* $c_1, \ldots, c_S \in A$, where A is the alphabet of Unicode characters and S is the length of the input segments. The goal of the tokenizer is to divide input characters in tokens and sentences. The classification task is therefore for each character to be classified into three classes: token boundary follows, sentence boundary follows and no boundary.

We represent each character $c_i \in A$ as an embedding $\mathbf{e}_{c_i} \in \mathbb{R}^d$, a vector of length d, an equivalent of vector word embedding (Collobert et al., 2011) for characters.

When processing the fixed-length segment of characters $c_1, \ldots, c_S$, we use GRU on the sequence $\mathbf{e}_{c_1}, \ldots, \mathbf{e}_{c_S}$ to obtain $\mathbf{h}^f_1, \ldots, \mathbf{h}^f_S$, and another GRU on the reversed sequence to obtain $\mathbf{h}^r_1, \ldots, \mathbf{h}^r_S$, and classify every character c_i (into either no boundary, token boundary or sentence boundary) using a softmax on catenation of vectors $\mathbf{h}^f_i$ and $\mathbf{h}^r_{S+1-i}$. We employ dropout (Srivastava et al., 2014) on the softmax input.

When tokenizing a text, we start by processing the first S characters. If any sentence boundaries were found, we remove as many complete sentences as possible and proceed by processing the rest of the characters. If there are no sentence boundaries in the segment, we continue with the processing from the first word boundary in the second half of the segment, or last word boundary in the second half of the segment if the former was not found. In the cast case when no token boundaries were found in the segment, we continue the processing from the middle of the segment.

We train the tokenizer using the Adam stochastic optimization method (Kingma and Ba, 2014). We randomly shuffle input sentences in each training epoch and we concatenate them (using a space character) into one long string. Using the described method, we locate the positions of the S-character segments covering this string, and perform the stochastic optimization using minibatches of size 100.

3.3. Generating Detokenized Text of a Tokenized Corpus

In case the `SpaceAfter=No` feature is not present in the treebank and the language uses spaces between words,[6] we can use additional raw corpus to *detokenize* the treebank, i.e., to fill the missing `SpaceAfter=No` features according to the raw corpus. This allows us to train the tokenizer even when the `SpaceAfter=No` feature is missing in the treebank.

We suggest the following heuristic when deciding if a `SpaceAfter=No` feature should be added between words v and w. We start by counting the number of occurrences of lowercased sequence $\llcorner v \llcorner w \llcorner$ in the raw corpus, and compare it to the number of corpus occurrences of lowercased sequence $\llcorner vw \llcorner$. If one of the figures is greater than the other, we add or do not add the `SpaceAfter=No` feature accordingly. If both counts are the same, then if both words v and w contain a letter, or both contain a digit, we do not add the `SpaceAfter=No`. Otherwise, we continue counting number of occurrences of additional sequences, until we either find a differing number, or we process the whole sequence (in which case we do not add the `SpaceAfter=No` feature). The additional sequences are:

- lowercased $v \llcorner w$ and vw,
- lowercased $v \llcorner w$ and vw where each character is replaced by its Unicode General category,
- lowercased $\llcorner v \llcorner w \llcorner$ and $\llcorner vw \llcorner$ where each character is replaced by its Unicode General category.

Although the method may seem as an unfounded heuristic, it has straightforward motivation and we show in the next Section that it performs well.

[4]Czech, English, Finnish, Slovenian and Tamil; although Finnish-FTB treebank also contains `SpaceAfter=No` feature, it is used incorrectly.

[5]Currently, UDPipe solves the problem by inserting a module consisting of a trivial lookup in the dictionary of multi-word tokens which is generated from training data.

[6]For languages that do not use spaces between words, we can utilize the tokenizer directly without any modification.

Language	Unit	UDPipe + SpaceAfter			UDPipe + detokenization			Rule-based tokenizer			Baseline tokenizer		
		Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
Czech	Tokens	**100.0**	**100.0**	**100.0**	99.9	99.9	99.9	***100.0***	***100.0***	***100.0***	*85.1*	*72.8*	*78.4*
	Sentences	**89.3**	**81.7**	**85.3**	88.2	80.7	84.3	*88.6*	*72.7*	*79.9*	*73.9*	*68.1*	*70.9*
English	Tokens	**98.6**	**98.7**	**98.7**	**98.6**	98.1	98.4	*91.3*	*96.1*	*93.7*	*85.6*	*73.4*	*79.0*
	Sentences	**76.9**	**66.2**	**71.2**	76.5	64.2	69.9	*74.4*	*42.7*	*54.3*	*72.3*	*51.8*	*60.4*
Finnish	Tokens	**99.8**	**99.8**	**99.8**	99.7	99.5	99.6	*53.7*	*55.2*	*54.4*	*84.3*	*72.2*	*77.8*
	Sentences	91.7	**89.2**	90.4	**92.8**	88.3	**90.5**	*48.1*	*40.0*	*43.6*	*89.5*	*82.9*	*86.1*
Slovenian	Tokens	99.9	**99.9**	**99.9**	**100.0**	**99.9**	**99.9**	*99.5*	*99.6*	*99.6*	*85.3*	*74.1*	*79.3*
	Sentences	**98.4**	**98.4**	**98.4**	97.2	97.5	97.3	*95.1*	*90.6*	*92.8*	*95.0*	*96.3*	*95.7*
Tamil	Tokens	**99.6**	**99.7**	**99.6**	99.3	99.4	99.4	*95.6*	*98.5*	*97.0*	*90.1*	*81.7*	*85.7*
	Sentences	**98.3**	**99.1**	**98.7**	98.0	**99.1**	**98.7**	*16.7*	*0.8*	*1.6*	*73.6*	*85.8*	*79.2*

Table 1: Tokenizer precision, recall and F1-score on treebanks of Universal Dependencies 1.2 which include the original raw text. The UDPipe tokenizer results are presented both when using the original plain text (through the `SpaceAfter=No` feature) and when reconstructing the original plain text using additional raw text corpus. For comparison, we also provide results for a baseline tokenizer (which splits tokens on spaces and sentences on spaces following dot, question mark or exclamation mark) and a rule-based tokenizer from MorphoDiTa (Straková et al., 2014), developed initially for Czech. Best results in each category (Precision/Recall/F1-score) are shown in bold font.

3.4. Tokenization Results

Our goal was not only to achieve high tokenizing accuracy (score), but also reasonable runtime performance. We therefore utilized GRUs instead of LSTMs and preferred short character embeddings during hyperparameter selection. We decided to select single hyperparameters for all treebanks, so that we avoid the need for development set for treebanks which do not contain the `SpaceAfter=No` feature. We utilize character embeddings of size 24, dropout value of 0.3 and segment size 50.

We present the tokenization results on those UD 1.2 treebanks containing the original texts in Table 1, as F1-score compared to gold tokenization of the treebanks. For comparison, we also provide a baseline tokenizer (which splits tokens on spaces and sentences on spaces following dot, question mark or exclamation mark) and a rule-based tokenizer from MorphoDiTa (Straková et al., 2014) developed initially for Czech. The performance of the rule-based tokenizer shows some interesting (even if not unexpected) properties – while the tokenizer reaches very high performance for Czech and Slovak (which are two closely related languages and mother tongues of the authors), it fails for sentence boundary detection in Tamil. This happens because the rule-based tokenizer heavily depends on casing, while Tamil does not have cased characters.

The trainable tokenizer performs remarkably well, notably in token boundary detection. The lower performance in sentence boundary detection for some treebanks is caused among others by the fact that the original texts are segmented using some additional structural information (e.g., headings, titles), the information about which is not retained in the corpus.

We also evaluate the effect of detokenization described in Section 3.3. in Table 1. We used the W2C corpora collection[7] as the source of additional raw corpora.[8] Performance of the tokenizer trained using automatically generated `SpaceAfter=No` feature is highly competitive when compared to the performance of the tokenizer trained using gold annotations. Due to these positive results, we expect the tokenizer trained on automatically detokenized treebanks using raw texts to perform well even on those treebanks which do not use the `SpaceAfter=No` feature (32 out of 37 treebanks of UD 1.2).

4. Morphological Analysis and POS Tagging

There are several morphological fields used in the CoNLL-U format:

- universal part-of-speech tag (Petrov et al., 2012),
- list of morphological features (Zeman, 2008),
- language-specific part-of-speech tag,
- lemma or stem.

UDPipe is able to fill all these fields, depending on which of them are available in the training data.
To perform the POS tagging and lemmatization, UDPipe uses MorphoDiTa (Straková et al., 2014). The MorphoDiTa POS tagger is based on part of (Spoustová et al., 2009) and is implemented as a supervised, rich feature averaged perceptron (Collins, 2002), employing dynamic programming at runtime (Viterbi decoder).

4.1. Morphological Analysis

In order to use the averaged perceptron tagger, morphological analyses for every input form must be provided. Performing such analysis for a language with small tag set may

[7] `http://hdl.handle.net/11858/00-097C-0000-0022-6133-9`

[8] Surprisingly, we obtained best tokenization results by using only the first 500kB of every raw corpus.

be trivial (by considering all possible tags), but if the tag set is richer (which is the case of many languages using morphological features in CoNLL-U) or if lemmatization is performed, either a morphological dictionary or a specialized analyzer is usually required.

In order to avoid the need for additional language resource or language-specific code, we have developed the following morphological "guesser."

For every suffix of fixed length,[9] we establish G most frequent analyses according to the training data. If there are not enough forms with the given suffix, we consider shorter and shorter suffixes, until we obtain the required number of analyses.Each analysis consists of a universal part-of-speech tag, language-specific part-of-speech tag if available, list of (possible empty) morphological features and a lemma rule. The lemma rule is the shortest formula for generating a lemma from a given form, using any combination of "remove a specific prefix", "remove a specific suffix", "append a prefix" and "append a suffix" operations.

We then generate a morphological dictionary of full forms from the UD data. Such a dictionary, even if it covers most frequent analyses, is almost surely not complete. We therefore *enrich* the dictionary by adding E additional analyses to every form of the dictionary (using the analyses associated with the form suffix, which we described above).

To analyze a given form, we try locating it in the enriched dictionary. If it is contained in the dictionary, we return the dictionary analyses (including the E added ones); otherwise, we return all G analyses associated with the form suffix. Usually E can be considerably smaller than G, because the most common form analyses are likely present in the dictionary itself. Consequently, most forms having E instead of $G \gg E$ analyses speeds up the decoding substantially, by a factor of $(G/E)^3$, because Viterbi decoder of order 3 is used (each state consist of a pair of tags).

Morphological information is, in some languages, also indicated by a prefix. For example, negations in Czech are frequently formed by prefixing a word with *ne-*, and superlatives are formed by prefixing comparatives with *nej-* prefix. If we detect such situation during analyzer construction, we create the above described suffix guesser individually for every such prefix.[10]

4.2. Classification Features

Internally, UDPipe uses in fact two models, one disambiguating all available morphological fields (a *POS tagger*), and the other one performing lemmatization (a *lemmatizer*), because a combination of two taggers improves overall accuracy. Both POS tagger and lemmatizer employ their own morphological guesser and utilize different classification feature sets.

The POS tagger disambiguates all available morphological fields (and can jointly disambiguate a lemma too, if it helps tagging accuracy). It uses a feature set adopted from Spoustová et al. (2009), who described classification features developed for Czech, a morphologically rich language. The lemmatizer disambiguates the lemma (and jointly also universal part-of-speech tag in order to improve lemmatization accuracy). Both feature sets are in the UDPipe repository.

4.3. Tagging Results

One of the most important hyperparameters of tagger training is the number of analyses G returned by the guesser, and the number of analyses E added to the dictionary for every word form. These hyperparameters were tuned[11] separately for every treebank, so that the tagging performance on the development portion of the treebank would be as high as possible.

The success rate (whether the correct analysis is one of those returned by the analyzer) of the morphological guesser is presented in Table 2, together with the average number of analyses returned by the guesser. The same characteristics is returned for the lemmatization guesser.

The part-of-speech tagging and lemmatization accuracy is presented also in Table 2. We do not compare our tagger to any related work, because we could not find any part-of-speech tagging and lemmatization results on Universal Dependencies 1.2. However, the used POS tagger algorithm and the used feature set achieve state-of-the-art results on Czech and very good results on English (Spoustová et al., 2009).

5. Dependency Parsing

Parsito is a transition-based, non-projective dependency parser described in (Straka et al., 2015), capable of parsing both projective and non-projective sentences. The parser, inspired by (Chen and Manning, 2014), uses a neural network classifier for prediction and requires no feature engineering. In (Straka et al., 2015) a new *search-based oracle* is proposed, which improves parsing accuracy similarly to a dynamic oracle, but is applicable to any transition system, such as the fully non-projective `swap` system. The parser has excellent parsing speed, compact models, and achieves high accuracy.

UDPipe employs the Parsito parser nearly unmodified. Compared to (Straka et al., 2015), only an optional beam-search decoding along the lines of (Zhang and Nivre, 2011) was added. The beam-search decoding improves accuracy, but decreases runtime performance. By default, beam search of size 5 is used.

5.1. Parsing Results

We trained the parser as described in (Straka et al., 2015), choosing the hyperparameters, transition system and an oracle which maximize the parser performance on the development portion of the data.

We report both unlabelled (UAS) and labelled attachment score (LAS) of the parser in Table 3.

We consider both the situations when the part-of-speech tags are automatically generated and when the gold tags are used, because while most users will parse data with automatically generated POS tags, system comparison is usually performed using gold tags.

[9]We use suffixes of length 4, thereby constructing morphological guesser with reasonable high accuracy and small size.

[10]We currently use four such most widely used prefixes, if they are used by at least 10 lemmas.

[11]We considered $G \in \{8, 10, 12\}$ and $E \in \{4, 5, 6\}$.

Language	Size		UDPipe morphological analyzer							UDPipe tagger				
	Words	Sentences	TA/F	UPOS	XPOS	Feats	Tags	LA/F	Lemma	UPOS	XPOS	Feats	Tags	Lemma
Ancient Greek	244 993	16 221	7.5	99.8	98.4	98.8	98.4	4.6	90.4	90.9	77.3	88.5	77.2	86.7
Ancient Greek–PROIEL	206 966	16 633	7.9	99.7	99.7	98.7	98.5	5.0	95.0	96.6	96.3	88.9	88.0	93.4
Arabic	282 384	7 664	6.7	99.9	99.3	99.3	99.3	6.7	—	98.7	97.5	97.5	97.3	—
Basque	121 443	8 993	6.2	98.5	—	97.2	95.9	4.5	97.1	93.1	—	87.3	85.6	93.5
Bulgarian	156 319	11 138	7.0	99.6	98.6	98.8	98.2	6.1	98.3	97.8	94.3	93.9	92.7	94.5
Croatian	87 765	3 957	6.9	99.2	—	96.0	95.4	3.8	97.6	95.1	—	85.7	85.1	92.4
Czech	1 506 490	87 913	7.4	99.8	98.7	98.3	98.2	6.4	99.6	98.4	93.0	92.4	92.0	97.8
Danish	100 733	5 512	7.2	99.8	—	98.9	98.8	7.2	99.0	95.8	—	94.5	93.3	95.5
Dutch	200 654	13 735	6.6	98.0	96.0	98.2	95.7	5.2	93.4	89.2	88.4	90.8	85.9	88.9
English	254 830	16 622	9.1	99.7	99.2	99.5	99.1	7.5	99.4	94.5	93.4	95.0	92.0	97.0
Estonian	9 491	1 315	7.3	97.1	88.4	93.6	88.3	5.0	85.8	87.9	73.0	79.3	72.9	77.0
Finnish	181 022	13 581	5.4	98.9	98.9	97.5	97.1	4.2	90.5	94.9	95.7	92.7	91.6	86.5
Finnish–FTB	159 829	18 792	6.0	98.4	96.8	97.6	96.8	5.7	93.7	93.6	91.0	92.7	90.5	89.0
French	401 491	16 446	7.5	99.9	—	—	99.9	6.9	—	95.9	—	—	95.9	—
German	298 242	15 894	6.9	99.7	—	—	99.7	7.3	—	90.7	—	—	90.7	—
Gothic	56 128	5 450	7.8	99.5	99.5	97.9	97.8	4.6	95.9	95.4	95.8	88.0	86.2	93.4
Greek	59 156	2 411	6.7	99.7	99.4	99.2	98.3	4.9	97.3	97.4	97.1	92.3	91.0	94.7
Hebrew	158 855	6 216	8.3	99.7	99.4	97.5	96.7	6.6	—	94.8	94.5	91.2	90.1	—
Hindi	351 704	16 647	14.1	99.5	98.9	97.6	96.5	12.6	99.6	95.8	94.6	89.6	87.2	98.0
Hungarian	26 538	1 299	5.3	97.8	—	96.4	95.7	5.2	95.4	92.8	—	89.8	88.8	87.3
Indonesian	121 923	5 593	7.4	99.9	—	—	99.9	7.2	—	93.6	—	—	93.6	—
Irish	23 686	1 020	6.1	97.5	97.0	88.9	86.7	4.9	92.6	90.3	88.7	79.0	75.7	87.3
Italian	271 180	12 677	7.5	99.9	99.6	99.6	99.4	5.0	99.5	97.2	96.9	96.8	96.0	97.7
Japanese–KTC	267 631	9 995	—	—	—	—	—	—	—	—	—	—	—	—
Latin	47 303	3 269	8.5	98.2	92.5	95.0	92.5	6.4	92.2	90.8	76.5	79.7	76.2	79.9
Latin–ITT	259 684	15 295	6.0	99.9	99.2	99.5	99.1	4.1	99.7	98.8	93.7	94.4	93.4	98.4
Latin–PROIEL	165 201	14 982	7.8	99.7	99.7	98.3	98.1	5.4	97.7	96.2	95.8	88.4	87.7	95.2
Norwegian	311 277	20 045	7.6	99.6	—	99.4	99.1	6.7	99.5	97.2	—	95.4	94.6	96.9
Old Church Slavonic	57 507	6 346	7.4	99.5	99.4	97.6	97.2	4.3	94.5	95.7	95.4	88.6	88.0	92.9
Persian	152 871	5 997	6.8	99.9	99.6	99.7	99.6	5.0	—	96.9	96.1	96.3	96.0	—
Polish	83 571	8 227	7.5	99.7	97.0	97.0	97.0	5.5	97.6	96.0	84.7	84.8	84.5	92.7
Portuguese	212 545	9 359	14.3	99.9	98.7	99.2	98.6	4.8	99.4	97.4	91.5	94.5	91.1	97.8
Romanian	12 094	633	6.8	97.9	93.3	94.5	93.3	5.8	93.8	88.3	80.9	82.0	80.9	75.3
Slovenian	140 418	7 996	7.5	99.4	97.3	97.7	97.2	5.5	98.9	95.6	87.6	88.0	86.9	94.8
Spanish	431 587	16 013	8.1	99.7	—	99.1	98.4	5.6	99.4	95.0	—	95.6	92.9	96.2
Swedish	96 819	6 026	6.9	99.7	98.8	99.0	98.7	4.3	98.8	95.7	93.5	94.3	92.8	95.5
Tamil	9 581	600	4.7	95.7	91.8	93.4	91.7	4.7	95.4	85.4	79.4	82.5	78.6	87.7

Table 2: Morphological analyzer success rate and tagging accuracy for all Universal Dependencies 1.2 treebanks. The tag analyses per form (TA/F) is the average number of morphological tagger analyses per form, while lemma analyses per form (LA/F) is the average number of lemmatizer analyses. We report the success rate, i.e., the ratio of words, for which the analyser produces among others the correct analysis, for the morphological analyser. The column Tags stands for all morphological tags (concatenation of UPOS, XPOS and Feats). For the tagger, accuracy is reported.

	Size	Size	Non-proj.	Non-proj.	Automatic POS tags	Automatic POS tags	Gold POS tags	Gold POS tags	Gold POS tags	Gold POS tags
Language	Words	Sentences	Non-proj. edges	Non-proj. sentences	UDPipe		UDPipe		(Ammar et al., 2016) monolin.	(Ammar et al., 2016) multilin.
					UAS	LAS	UAS	LAS	LAS	LAS
Ancient Greek	244 993	16 221	9.78%	63.22%	69.3	62.9	**72.0**	**67.1**		
Ancient Greek–PROIEL	206 966	16 633	5.95%	39.48%	76.3	70.4	**77.8**	**73.0**		
Arabic	282 384	7 664	0.33%	8.19%	80.8	76.0	**81.0**	**76.6**		
Basque	121 443	8 993	4.95%	33.74%	75.0	69.7	**80.0**	**76.2**		
Bulgarian	156 319	11 138	0.21%	2.83%	89.2	84.7	**91.7**	**87.2**		
Croatian	87 765	3 957	0.46%	7.48%	78.7	71.5	**82.9**	**76.6**		
Czech	1 506 490	87 913	0.93%	12.58%	86.6	82.7	**88.6**	**85.8**		
Danish	100 733	5 512	1.97%	22.84%	78.6	74.8	**83.5**	**80.6**		
Dutch	200 654	13 735	4.10%	30.87%	**78.7**	71.3	78.5	**75.0**		
English	254 830	16 622	0.48%	4.96%	84.0	80.2	**87.5**	85.0	***85.9***	*85.4*
Estonian	9 491	1 315	0.08%	0.61%	81.3	73.4	**87.3**	**84.5**		
Finnish	181 022	13 581	0.74%	7.68%	80.7	76.3	**84.5**	**81.7**		
Finnish–FTB	159 829	18 792	1.09%	6.78%	81.2	76.3	**85.6**	**82.9**		
French	401 491	16 446	0.83%	12.45%	81.9	77.8	**84.5**	81.0	*81.7*	***82.4***
German	298 242	15 894	0.90%	12.08%	77.9	71.8	**82.9**	78.6	***79.3***	*78.9*
Gothic	56 128	5 450	3.86%	23.85%	76.5	68.8	**79.5**	**74.1**		
Greek	59 156	2 411	1.95%	27.87%	80.8	76.7	**82.4**	**79.4**		
Hebrew	158 855	6 216	0.00%	0.00%	82.7	77.1	**85.6**	**81.8**		
Hindi	351 704	16 647	0.76%	13.60%	91.7	87.5	**94.2**	**91.2**		
Hungarian	26 538	1 299	2.09%	25.17%	76.2	69.3	**82.3**	**76.9**		
Indonesian	121 923	5 593	0.13%	1.93%	80.7	73.9	**82.8**	**78.3**		
Irish	23 686	1 020	0.81%	12.84%	72.8	63.4	**75.6**	**69.5**		
Italian	271 180	12 677	0.32%	3.94%	88.7	85.7	**90.2**	88.1	*88.7*	***89.1***
Japanese–KTC	267 631	9 995	0.00%	0.00%	–	–	**85.9**	**76.9**		
Latin	47 303	3 269	7.13%	46.22%	58.1	48.5	**62.7**	**55.7**		
Latin–ITT	259 684	15 295	3.45%	37.20%	79.6	76.2	**81.2**	**78.7**		
Latin–PROIEL	165 201	14 982	5.22%	30.09%	75.2	68.3	**78.3**	**73.3**		
Norwegian	311 277	20 045	0.60%	7.70%	87.0	84.5	**90.2**	**88.3**		
Old Church Slavonic	57 507	6 346	3.71%	21.57%	81.1	74.1	**84.6**	**79.5**		
Persian	152 871	5 997	0.38%	5.14%	84.1	79.7	**86.3**	**83.0**		
Polish	83 571	8 227	0.04%	0.32%	86.2	79.4	**91.3**	**87.8**		
Portuguese	212 545	9 359	1.27%	18.44%	85.0	81.3	**87.2**	84.7	*85.7*	***86.2***
Romanian	12 094	633	0.89%	11.37%	68.4	56.4	**74.1**	**63.2**		
Slovenian	140 418	7 996	1.11%	13.61%	83.8	80.2	**89.5**	**88.1**		
Spanish	431 587	16 013	0.30%	6.05%	83.3	79.7	**87.1**	**84.5**	*83.7*	*84.3*
Swedish	96 819	6 026	0.19%	2.77%	81.2	77.0	**86.2**	83.2	*83.5*	***84.5***
Tamil	9 581	600	0.29%	2.17%	64.8	56.3	**78.1**	**71.5**		

Table 3: Parsing accuracy on all treebanks of Universal Dependencies version 1.2. We characterize each treebank by its size and level of non-projectivity and present our parsing results in terms of unlabeled attachment score UAS and labeled attachment score LAS. We show the results separately for the case when the POS tags are automatically recognized, and when the gold POS tags are used. For comparison, we also show parsing results of (Ammar et al., 2016), both in the monolingual setting and in multilingual setting. Best results in each category (UAS/LAS) are shown in bold font.

We compare our parser results to (Ammar et al., 2016), which present two parsers trained on 7 treebanks from Universal Dependencies 1.2. The first parser is monolingual (it is trained on the training data of the target language only), the second one is trained on all 7 treebanks. Although both these parsers utilize additional raw corpora during training (by employing pretrained word embeddings and Brown clusters embeddings) and use more advanced Stack-LSTM model (Dyer et al., 2015), UDPipe parsing results are quite competitive, as presented in Table 3.

Note that for some languages there is a noticeable drop in parsing performance when automatically generated POS tags are used instead of gold POS tags. Since we train the tagger and the parser using the same input data, we were concerned whether it is proper to train the parser using generated POS tags that were trained on the same data (because these POS tags are nearly identical to the gold tags, unlike POS tags generated on development and testing portion of the data). However, when the parser is trained using POS tags generated by 10-fold cross-training (i.e., POS tags of each fold of the training data are generated by a tagging model trained on the remaining folds), the resulting attachment scores (both unlabelled and labelled) averaged across all treebanks dropped, demonstrating that this is not the case.

6. UDPipe

UDPipe is a single C++ tool containing a tokenizer, morphological analyzer, POS tagger, lemmatizer and a dependency parser as described in the previous sections. It is released under the very permissive Mozilla Public License (MPL) and can be obtained from the UDPipe homepage `http://ufal.mff.cuni.cz/udpipe` or directly using the permanent ID `http://hdl.handle.net/11234/1-1659`. The training code is part of the release.

The whole pipeline is easily trainable using training data in CoNLL-U format (and optionally additional raw text if the tokenizer is to be trained and the pre-tokenized version of the text is not available).[12] All the trained models of the whole pipeline are stored in a single file. Naturally, it is possible to train only a selected part of the complete pipeline.

We believe UDPipe to be a light-weight, efficient software with low resource usage and high throughput – the complete pipeline usually has throughput of several thousand words per second, with model sizes on the order of megabytes.

In addition to the UDPipe binary we also provide a library with many language bindings – we currently offer Java, Python, Perl and C#. Furthermore, UDPipe is available as a web service with REST API.

Finally, we released models for most of UD 1.2 treebanks – the only missing treebanks are Japanese (because of licensing issues) and four historical treebanks (due to unavailable raw corpus for tokenizer training, which we nevertheless hope to obtain soon).

[12]Note that even if we have set the hyperparameters to reasonable defaults, some hyperparameter tuning still must take place in order to obtain best possible performance results.

7. Conclusions and Future Work

We presented *UDPipe*, a simple, unified tool for tokenization, morphological analysis, POS tagging, lemmatization and dependency parsing. It is distributed as one binary (and as a library) and with models for most of the Universal Dependencies treebanks. UDPipe can also be easily trained for new languages and requires neither additional resources such as morphosyntactic dictionaries, nor feature engineering and no language-specific knowledge.

There are several areas we want to explore in the future. We want to evaluate the effect of employing real morphological dictionary instead of morphological analysis using UD data only described in Section 4.1. Furthermore, we would like to utilize additional raw corpora during training, for example by using word embeddings, character-level embeddings, or improving the morphological guesser.

Acknowledgments

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). This research was also partially supported by SVV project number 260 224.

Ammar, W., Mulcaire, G., Ballesteros, M., Dyer, C., and Smith, N. A. (2016). One parser, many languages. *CoRR*, abs/1602.01595.

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

Chen, D. and Manning, C. (2014). A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar, October. Association for Computational Linguistics.

Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259.

Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8. Association for Computational Linguistics, July.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537.

de Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., and Manning, C. D. (2014). Universal stanford dependencies: A cross-linguistic typology. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. In *Proceedings* *of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China, July. Association for Computational Linguistics.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, pages 5–6.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., and Trancoso, I. (2015). Finding function in form: Compositional character models for open vocabulary word representation. *CoRR*, abs/1508.02096.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., McDonald, R., Petrov, S., Pyyaslo, S., Silveira, N., Tsarfaty, R., and Zeman, D. (2016). Universal dependencies v1: A multilingual treebank collection. Submitted to this LREC.

Petrov, S., Das, D., and McDonald, R. (2012). A universal part-of-speech tagset. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may. European Language Resources Association (ELRA).

Spoustová, D. j., Hajič, J., Raab, J., and Spousta, M. (2009). Semi-Supervised Training for the Averaged Perceptron POS Tagger. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 763–771, Athens, Greece, March. Association for Computational Linguistics.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Straka, M., Hajič, J., Straková, J., and Hajič jr., J. (2015). Parsing universal dependency treebanks using neural networks and search-based oracle. In *Proceedings of Fourteenth International Workshop on Treebanks and Linguistic Theories (TLT 14)*, December.

Straková, J., Straka, M., and Hajič, J. (2014). 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. Association for Computational Linguistics.

Univeral Dependencies Treebanks version 1.2. (2015). Released Nov 15, 2015. Permanent identifier for download `http://hdl.handle.net/11234/1-1548`, documentation at `http://universaldependencies.org/`.

Zeman, D. (2008). Reusable tagset conversion using tagset drivers. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, may. European Language Resources Association (ELRA).

Zhang, Y. and Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Short Papers - Volume 2*, HLT '11, pages 188–193, Stroudsburg, PA, USA. Association for Computational Linguistics.