UDon2: a library for manipulating Universal Dependencies trees

Dmytro Kalpakchi and Johan Boye
Division of Speech, Music and Hearing
KTH Royal Institute of Technology
Stockholm, Sweden
dmytroka@kth.se, jboye@kth.se

Abstract

UDon2 is an open-source library for manipulating dependency trees represented in the CoNLL-U format. The library is compatible with the Universal Dependencies. UDon2 is aimed at developers of downstream Natural Language Processing applications that require manipulating dependency trees on the sentence level (to complement other available tools geared towards working with treebanks).

1 Introduction

Universal Dependencies (UD) is a framework unifying ways of annotating grammar for different human languages (Nivre et al., 2020). To date, the UD community has produced more than 150 treebanks in 90 languages and a number of UD-compatible tools for processing data. Most of the available tools focus on working with treebanks, e.g. annotating textual data, validating existing treebanks or making simple edits. However, many downstream Natural Language Processing (NLP) applications require researchers to manipulate individual dependency trees. For instance, finding all subordinate clauses in the sentence might help in performing text simplification, finding all objects connected to a verb in the passive form might be useful for creating a list of candidate referents for co-reference resolution, and being able to remove certain subtrees might assist in generating reading-comprehension questions.

Some of those tasks are easy to achieve with some simple scripting, but such ad-hoc solutions become difficult to maintain over time. Furthermore, they tend to lack speed and hinder large-scale experimentation, since they are typically written in high-level programming languages in presence of time pressure. To aid the community in solving these tasks, we present UDon2 - a library for manipulating UD dependency trees optimized for querying. UDon2 has a user-friendly API allowing to perform routine tasks with only a couple of lines of code. For instance, finding all nominal objects in singular requires only a code snippet below.

```
import udon2
nodes = udon2.ConllReader.read_file("example.conll")
sing = [obj for node in nodes for obj in node.select_by("deprel", "obj")
        if obj.has("feats", "Number", "Sing")]
```

UDon2 is an open-source library written in C++ with Python bindings, combining the speed of C++ and the flexibility and ease-of-use of Python. UDon2 is hosted on Github (the source code is available at `https://github.com/udon2/udon2`), and everyone is welcome to contribute.

2 Example use cases

UDon2 operates on dependency trees for individual sentences. Preparing a raw text for downstream applications requires segmenting it into sentences and then parsing every sentence to get its dependency tree stored in CoNLL-U format. The result of reading a CoNLL-U file is an instance of the `Node` class

Proceedings of the Fourth Workshop on Universal Dependencies (UDW 2020), pages 120–125
Barcelona, Spain (Online), December 13, 2020

representing a 'root' pseudonode of the dependency tree. The dependent of the 'root' pseudonode will be later referred to as *a root word*. In the section below, we present possible use cases along with the manipulations available for a generic `Node` instance `n`, and exemplify using the dependency tree in Figure 1 with its root word *study* being denoted as `r`.

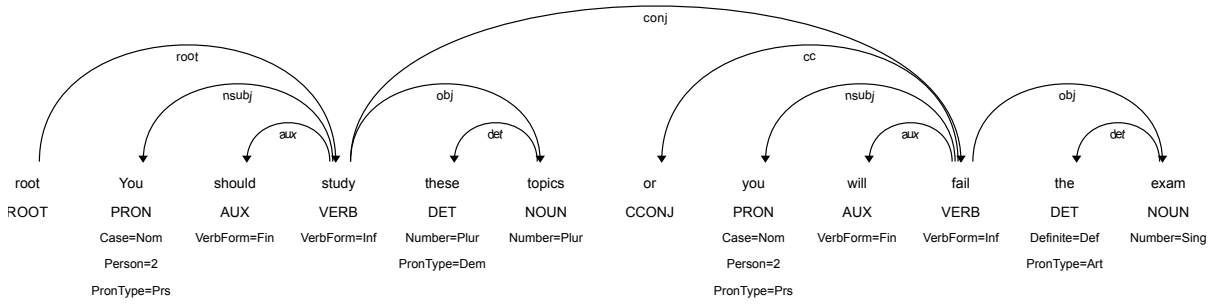

Figure 1: A dependency tree for the sentence "You should study these topics or you will fail the exam", obtained using the ewt-model of Stanza package (Qi et al., 2020) and visualized using UDon2.

2.1 Accessing basic properties

Each node `n` has a number of accessors and mutators for its word index, universal part-of-speech (POS) tag, language-specific POS tag, lemma, form, dependency relation with its head node, universal morphological features (FEATS) or any other annotation (MISC). Each accessor can be called as `n.<prop>` substituting `<prop>` for `id`, `upos`, `xpos`, `lemma`, `form`, `deprel`, `feats` and `misc` respectively. The last two will be referred to as key-value properties. Each mutator can be called as `n.<prop> = val` with the same values of `<prop>`. The parent node of `n` can be accessed by calling `n.parent`, and the children of `n` can be accessed by calling `n.children`. While mutator for a parent is available (by calling `n.parent = n1`), no direct mutator for children is. Instead, calling `n.add_child` or `n.remove_child` is required to modify the list of children.

Let $T_{\mathtt{n}}$ denote a subtree rooted at `n`. Calling `n.get_subtree_text()` will return a textual representation of $T_{\mathtt{n}}$. For instance, calling `r.get_subtree_text()` will return the whole sentence.

Comments and enhanced dependency relations are currently not supported, since those are typically not provided by existing dependency parsers.

2.2 Multiword and empty nodes

Multiword tokens are supported and can be accessed by calling `n.multi_word`. If `n` belongs to any multiword token, an instance of `udon2.MultiWordNode` will be returned, otherwise the accessor will return None. Mutators for multi-word nodes are currently not available. Getting a textual representation of a subtree (by calling `n.get_subtree_text()`) accounts for the multiword nodes. Empty nodes[1] are currently ignored while reading CoNLL-U files.

2.3 Querying

Querying a dependency tree for a specific type of node is useful, for instance, for finding all relative clauses of a sentence, or finding the subject of a sentence. UDon2 allows issuing a variety of queries for selecting the nodes in $T_{\mathtt{n}}$:

- having a property with a specified value, by calling `n.select_by(<prop>, <val>)`. Here, `<prop>` could be substituted for the same values as in the previous section, except key-value properties. `<val>` should be substituted for the desired value of the respective property. For instance, `r.select_by("upos", "VERB")` will return a list of `Nodes` corresponding to the verbs *study* and *fail*;

[1] `https://universaldependencies.org/format.html#words-tokens-and-empty-nodes`

- having specified key-value properties in the universal feature format[2] `<key-val-str>`, by calling `n.select_having(<prop>, <key-val-str>)`, where `<prop>` is one of `feats` or `misc`. For instance, the nodes for words *You* and *you* will be returned after calling `r.select_having("feats", "Case=Nom|Person=2|PronType=Prs")`;

- being *direct* children of `n` and having a specified non key-value property, by calling `n.get_by(<prop>, <val>)`.

- having a specified chain of dependency relations, by calling `n.select_by_deprel_chain` or `n.get_by_deprel_chain` (if the requirement of being a *direct* child is added). For instance, `r.select_by_deprel_chain("obj.det")` will return a list of `Nodes` corresponding to the determiners *these* and *the*, whereas `r.get_by_deprel_chain("obj.det")` will return only the `Node` corresponding to the determiner *these*;

- being identical to another node `n'`, by calling `n.select_identical(n')`;

- being identical to another node `n'` except for properties `props`, by calling `n.select_identical_except(n', props)` with `props` being a comma-separated string of property names (later referred to as *a prop-string*), e.g. `pos, rel`;

A number of simpler indicator queries to check whether a specified property is present are also available and described in our online documentation[3].

2.4 Pruning

Suppose we want, as a step in text simplification, to split all coordinate clauses in a sentence into separate sentences. This requires identifying the nodes corresponding to the roots of coordinate clauses, by using the querying functionality from the previous section. Each clause should then be converted to a separate dependency tree, and all coordinate conjunctions should be removed. UDon2 makes this possible via its `n.prune(<rel>)` and `n.make_root()` functions, where `rel` corresponds to the chain of dependency relations pointing at the node to be pruned. To exemplify the pruning operation, `r.prune("conj")` will result in a subtree corresponding to the sentence "You should study these topics". `r.make_root()` function will create a root pseudonode and assign it to be a parent of `r`.

If the same tree is going to be used multiple times, destructive pruning might not be a viable option. In order to avoid copying trees, which might be a time-intensive (currently not implemented) operation, UDon2 allows ignoring individual nodes or subtrees by calling `n.ignore(<label>)` (`n.ignore_subtree(<label>)`), which assigns an ignore label `label` to `n` (all nodes in a subtree induced by `n`). All ignored nodes (no matter the label) will be excluded for all the queries presented in the previous section and during calling `n.get_subtree_text()`.

Reverting to the original state, possible by calling `n.reset(<label>)` (`n.reset_subtree(<label>)`), will unignore only nodes with a matching ignore label. The `<label>` argument defaults to 0 for all mentioned methods. If all nodes should be reset (no matter the label), `n.hard_reset()` or `n.hard_reset_subtree()` should be used.

2.5 Visualization

UDon2 is capable of visualizing the dependency tree and storing it as an SVG file. An example of such visualization is shown in Figure 1 and the code for visualizing a tree with a root `node` is presented below.

```
from udon2.visual import render_dep_tree
render_dep_tree(node, "tree.svg") # node is an instance of udon2.Node
```

Providing support for other image formats is an ongoing work.

[2] https://universaldependencies.org/u/overview/morphology.html#features
[3] https://udon2.github.io

2.6 Transformations and convolution tree kernels

It is non-trivial to represent dependency trees as features to use in machine learning contexts. One option was proposed by Moschitti (2006) in the form of convolution partial tree kernels that can be used with Support Vector Machines (Cortes and Vapnik, 1995). In a nutshell, a partial tree kernel calculates the number of common tree structures (not only full subtrees) between two trees. Unfortunately, tree kernels cannot handle trees with labeled edges, which is why Moschitti (2006) applied kernels to dependency tree containing only lexicals. An alternative solution, proposed by Croce et al. (2011) and implemented in UDon2, is to re-format dependency trees to include the edge labels as separate nodes. Three possible formats were proposed, depending on the order of inclusion:

- POS-tag Centered Tree (PCT) - each grammatical relation is added as the father of the POS-tag and a lexical as a child (transformation is possible by calling `udon2.transform.to_pct(node)`);
- Grammatical Relation Centered Tree (GRCT) - each POS-tag is a child of a grammatical relation and a father of a lexical (transformation is possible by calling `udon2.transform.to_grct(node)`);
- Lexical Centered Tree (LCT) - both a POS-tag and a grammatical relation are children of a lexical (transformation is possible by calling `udon2.transform.to_lct()`).

In UDon2, a partial tree kernel can be calculated in any of the aforementioned formats by substituting a string `tree_format` with any of `PCT`, `GRCT` or `LCT` in the code snippet below.

```
from udon2.kernels import ConvPartialTreeKernel
# ptk_lambda and ptk_mu are decay factors as defined by Moschitti (2006)
kernel = ConvPartialTreeKernel(tree_format, ptk_lambda, ptk_mu)
# prints a number of common tree fragments between trees rooted at root1 and root2
print(kernel(root1, root2)) # root1 and root2 are udon2.Node instances
```

3 Related work

Currently available UD processing tools for Python are geared towards working with treebanks and making batch manipulations and edits. UDPipe (Straka and Straková, 2017) is a library written in C++ with bindings to other programming languages. UDPipe provides a trainable pipeline which performs sentence segmentation, tokenization, POS-tagging, lemmatization and dependency parsing. The library provides no built-in support for manipulations on dependency trees. A similar functionality is also provided by the Stanza package (Qi et al., 2020).

DepEdit (Peng and Zeldes, 2018) is a configurable tool for manipulating dependency trees in the CoNLL-U format. The manipulations are specified in the configuration file using regular expressions for selecting nodes of interest, and a custom syntax for specifying relations between the nodes, and actions to perform on the matched nodes. The tool is geared towards performing batch operations and thus operations like querying to get a list of matching nodes for performing further manipulations, getting a text of the subtree induced by the node or implementing convolution tree kernels are impossible to achieve, to the best of our knowledge.

Udapi (Popel et al., 2017) is one such framework providing the ability to parse dependency trees, visualize them, convert between different representation formats (CoNLL-U, SDParse and VISL-cg), applying batch queries and edits to treebanks, and validate the format and contents of treebanks. Udapi is available as a command line tool, and has APIs for Java, Python and Perl. One of the reviewers has brought to our attention that Udapi is capable of performing directly (or gives a possibility to implement) the same transformations as UDon2.

Two smaller packages, pyconll[4] and conllu[5], provide an interface to the CoNLL-U annotation scheme without the possibility of visualization, but with a possibility to reimplement the same transformations as in UDon2.

[4] `https://pyconll.github.io/`
[5] `https://github.com/EmilStenstrom/conllu/`

In order to compare the last three mentioned packages, we provide the benchmark results for UDon2 and Udapi in Table 1 on the same CoNLL-U file[6] as in (Popel et al., 2017) ran on the same machine having Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz on Ubuntu (x86_64) and Windows 10 (win32).

Package	OS	Memory	Load	Save	Read	Write	Text	Relchain
pyconll	Ubuntu	1683.1	12.88*	6.32	0.34	0.23	NA	0.47
	Windows	876.4	10.97	6.23	0.38	0.23	NA	0.54
conllu	Ubuntu	1208.7	16.83	4.28	0.19	0.1	NA	0.25
	Windows	707.2	19.11*	5.23*	0.22	0.09	NA	0.3
Udapi-Python	Ubuntu	756.0	19.88*	6.86	0.19	0.14	0.94	0.16
	Windows	421.6	19.09*	8.51*	0.2	0.11	1.01	0.15
UDon2	Ubuntu	772.0	3.27	3.34	0.75	0.42	0.24	0.14
	Windows	439.7	4.44	5.53	0.83	0.42	0.41	0.15

Table 1: Speed and memory comparison on `cs-ud-train-l.conllu` from UDv1.2 (68 MiB, 41k sentences, 800k words). Memory is in MiB and all other benchmarks provide average time in seconds after 30 runs on the computer with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz. **Load** refers to loading from CoNLL-U file, **Save** - to storing to the CoNLL-U file, **Read** - getting a form and a lemma for every node of every tree, **Write** - changing a deprel for every node of every tree, **Text** - computing a textual representation of a subtree induced by every root node of every tree, **Relchain** - finding nodes at the end of a relchain for every tree. The values with star indicate experiments with a standard deviation of more than 1 second.

4 Discussion and conclusions

Most of the current UD-compatible tools are focused on treebank developers, whereas UDon2 aims at helping researchers explore the use of dependency trees for downstream applications, and hence is optimized mostly for querying and interacting with CoNLL-U files. To the best of our knowledge, UDon2 is the first package providing the possibility to both perform manipulations on dependency trees, perform advanced transformations (such as GRCT, PCT or LCT), and compute convolution tree kernels.

UDon2 provides a superior performance on the majority of the benchmarks, except for Read and Write. The reason is that these two benchmarks require using Python's for-loops for C++ objects, requiring a lot of type conversions between Python and C++. UDon2 tries to avoid this by offering various query methods for common tasks, where looping is done in C++ as well (e.g. Text, Relchain, Load and Save benchmarks), which brings evident performance gains. Optimizing UDon2 for working better with Python's loops is an ongoing work and contributions are welcome. We hope that UDon2 is going to aid researchers in experimenting with dependency trees, and that it will be expanded with the help of the UD community.

Acknowledgements

This work was supported by Vinnova (Sweden's Innovation Agency) within project 2019-02997. We are also sincerely grateful to both reviewers for the incredibly useful comments (especially Reviewer 1 for the most thorough review we have ever seen). We would also like to thank Martin Popel for helpful discussions on the matter of benchmarking.

References

Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning*, 20(3):273–297.

[6] `https://github.com/UniversalDependencies/UD_Czech-PDT/raw/r1.2/cs-ud-train-l.conllu`

Danilo Croce, Alessandro Moschitti, and Roberto Basili. 2011. Structured lexical similarity via convolution kernels on dependency trees. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1034–1046.

Alessandro Moschitti. 2006. Efficient convolution kernels for dependency and constituent syntactic trees. In *European Conference on Machine Learning*, pages 318–329. Springer.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajič, Christopher D Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers, and Daniel Zeman. 2020. Universal dependencies v2: An evergrowing multilingual treebank collection. *arXiv preprint arXiv:2004.10643*.

Siyao Peng and Amir Zeldes. 2018. All roads lead to ud: Converting stanford and penn parses to english universal dependencies with multilayer annotations. In *Proceedings of the Joint Workshop on Linguistic Annotation, Multiword Expressions and Constructions (LAW-MWE-CxG-2018)*, pages 167–177.

Martin Popel, Zdeněk Žabokrtskỳ, and Martin Vojtek. 2017. Udapi: Universal api for universal dependencies. In *Proceedings of the NoDaLiDa 2017 Workshop on Universal Dependencies (UDW 2017)*, pages 96–101.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. *arXiv preprint arXiv:2003.07082*.

Milan Straka and Jana Straková. 2017. 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*, pages 88–99, Vancouver, Canada, August. Association for Computational Linguistics.