Udapi: Universal API for Universal Dependencies

Martin Popel, Zdeněk Žabokrtský, Martin Vojtek
Charles University, Faculty of Mathematics and Physics
Malostranské náměstí 25, Prague
{popel,zabokrtsky}@ufal.mff.cuni.cz, martin.vojtek@hotmail.com

Abstract

Udapi is an open-source framework providing an application programming interface (API) for processing Universal Dependencies data. Udapi is available in Python, Perl and Java. It is suitable both for full-fledged applications and fast prototyping: visualization of dependency trees, format conversions, querying, editing and transformations, validity tests, dependency parsing, evaluation etc.

1 Introduction

Universal Dependencies (UD)[1] is a project that seeks to develop cross-linguistically consistent treebank annotation, by both providing annotation guidelines and releasing freely available treebanks. Two years after the first release, UD version 2 (UDv2) of the guidelines was published, accompanied by the UDv2.0 release of the data: 70 treebanks for 50 languages, with 12M words in total, contributed by 145 treebank developers.[2]

The steady growth of the UD popularity results in an increased need for tools compatible with UD and its native data format CoNLL-U. Such tools are needed by both the treebank developers and users of the treebanks. Thanks to the simplicity of CoNLL-U, simple tasks can be performed with ad-hoc scripts or even standard Unix tools (`sed`, `cut`, `grep` etc.). However, there are several disadvantages of these ad-hoc solutions:

- They tend to be suboptimal regarding speed and memory, thus discouraging more frequent large-scale experiments.
- The code is less readable because the main logic is mixed with boilerplate.
- It is easy to forget handling edge cases.[3]
- Ad-hoc solutions are difficult to maintain once they outgrow the original simple task.

We present Udapi – a framework providing an API for processing UD, which should solve the above-mentioned problems. Udapi implementation is available in Python, Perl and Java. In this paper, we focus on the Python implementation because it currently has the best support and largest user community. The Perl and Java implementations are kept harmonized with the Python implementation as much as the differences between these programming languages allow.

The API is object-oriented and covers both processing units (§3.1) and data representation (§3.2). The development of Udapi is hosted at GitHub.[4] Anyone is welcome to contribute.

2 Example use cases

Udapi can be used both as Python library and via the command-line interface `udapy`. This section gives examples of the latter.

2.1 Parsing

```
echo "John loves Mary." | udapy \
  read.Sentences  tokenize.Simple \
  udpipe.En tokenize=0 write.Conllu
```

In this example, `udapy` executes a pipeline (called *scenario* in Udapi) with four processing units (called *blocks*): `read.Sentences` reads plain text from the standard input, one sentence per line; `tokenize.Simple` does a naïve tokenization; `udpipe.En` applies a UDPipe (Straka et al., 2016) model for English tagging (filling

[1] http://universaldependencies.org

[2] http://hdl.handle.net/11234/1-1983

[3] For example, when deleting a node in a middle of a sentence, we must reindex the ID, HEAD and DEPS columns, delete enhanced dependencies referring to the node, re-attach or delete possible dependents of the node and if it was part of a multi-word token, make sure the token is still valid.

[4] See http://udapi.github.io for further info, documentation and a hands-on tutorial.

Proceedings of the NoDaLiDa 2017 Workshop on Universal Dependencies (UDW 2017), pages 96–101,
Gothenburg, Sweden, 22 May 2017.

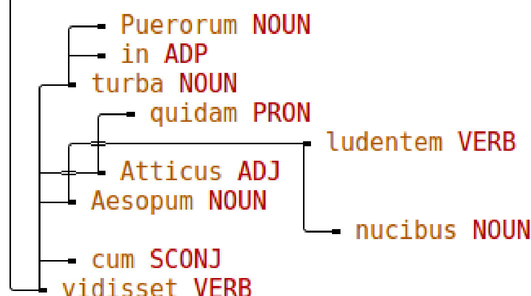
attributes upos, lemma and feats) and parsing (deprel and head). The parameter `tokenize=0` instructs UDPipe to skip tokenization. Finally, `write.Conllu` writes the parsed sentences to the standard output in the CoNLL-U format.

In practice, we recommend to use UDPipe's internal tokenization and `udapy -s` as a shortcut for appending `write.Conllu` to the scenario:

```
echo "John loves Mary." | udapy -s \
  read.Sentences  udpipe.En
```

2.2 Visualization

```
cat latin-sample.conllu | udapy \
 write.TextModeTrees attributes=form,upos
```



If no reader block is provided, `read.Conllu` is used by default. Block `write.TextModeTrees` is very useful for fast visualization of dependency trees in terminal and tracking changes in `vimdiff`. As the example above shows, it can render non-projectivities, it has a parameter for specifying the node attributes to be printed and it uses color highlighting in the terminal.

For longer documents, it is handy to use a pager (`less -R`)[5] so one can search attributes of all nodes with regular expressions. The shortcut `udapy -T` stands for `write.TextModeTrees color=1` and printing the default set of attributes *form*, *upos* and *deprel*:

```
udapy -T < latin-sample.conllu | less -R
```

Similarly, `udapy -H` is a shortcut for a (static) HTML version of this writer (see Figure 1) and `-A` is a shortcut for printing all attributes. Run `udapy --help` to learn more shortcuts.

Block `write.Html` also generates a HTML file, but it uses JavaScript for traditional-style tree rendering, tooltips, SVG export button, highlighting alignments between the sentence and nodes and possibly also between nodes of word-aligned parallel sentences, see Figure 2.

Block `write.Tikz` generates a LaTeX code, suitable for inclusion in papers, see Figure 3.

[5] The `-R` flag ensures the ANSI colors are displayed.

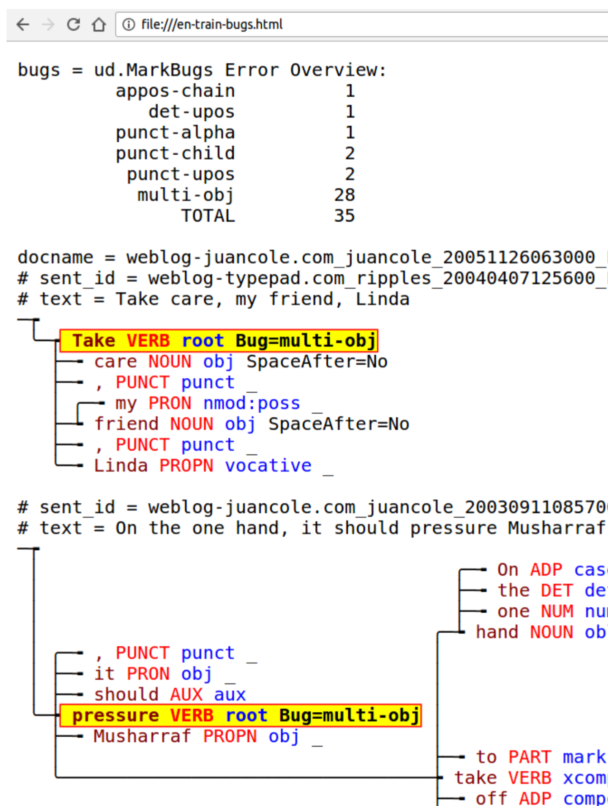


Figure 1: `write.TextModeTreesHtml` example output with a sample of annotation errors (and overall statistics) found by `ud.MarkBugs`.

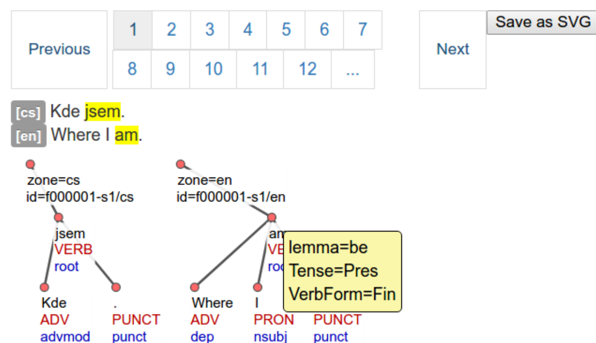


Figure 2: `write.Html` example output with a sample of Czech-English parallel treebank CzEng (Bojar et al., 2016) converted to the UD style.

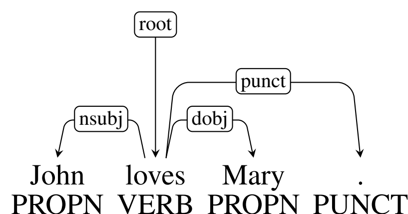


Figure 3: `write.Tikz` example output.

2.3 Format conversions

Udapi can be used for converting between various data formats. In addition to the native CoNLL-U format and to the visualization layouts mentioned in Section 2, Udapi currently supports SDParse (popularized by Stanford dependencies and Brat)[6] and VISL-cg[7] formats as illustrated below:

```
udapy write.Vislcg < x.conllu > x.vislcg
udapy read.Vislcg write.Sdparse \
      < x.vislcg > x.sdparse
```

2.4 Querying and simple edits

There are two online services for querying the released UD treebanks: SETS by the University of Turku[8] and PML-TQ by the Charles University.[9] The SETS querying language is easier to learn, but less expressive than the PML-TQ language.

Udapi offers an alternative where queries are specified in Python and may use all the methods defined in the API, thus being suitable even for complex queries. For example, using the method `is_nonprojective()`, we can find all non-projective trees in a CoNLL-U file, and mark the non-projective edges with a label "nonproj" stored in the MISC column (so the dependent node will be highlighted by `udapy -T`):

```
cat in.conllu | udapy -T \
util.Filter mark=nonproj \
keep_tree_if_node='node.is_nonprojective()'
```

The same can be achieved using `util.Mark` and the `-M` shortcut, which instructs the writer to print only trees that are "marked":

```
cat in.conllu | udapy -TM util.Mark \
 node='node.is_nonprojective()'
```

Block `util.Eval` can execute arbitrary Python code, so it can be used not only for querying, but also for simple (ad-hoc) editing. For example, we can delete the subtypes of dependency relations and keep only the universal part:[10]

```
cat in.conllu | udapy -s util.Eval \
 node='node.deprel = node.udeprel' \
 > out.conllu
```

For better reusability and maintainability, we recommend to store more complex edits in separate Python modules. For instance, module `udapi.block.my.edit` with class `Edit` will be available via `udapy` as `my.Edit`.

[6] http://brat.nlplab.org/
[7] http://visl.sdu.dk/visl/vislcg-doc.html
[8] http://bionlp-www.utu.fi/dep_search/
[9] http://lindat.cz
[10] So e.g. *acl:relcl* is changed to *acl*.

2.5 Validation

UD treebanks are distributed with an official `validate.py` script, which checks the CoNLL-U validity and also treebank-specific restrictions (e.g. a set of allowed deprel subtypes). Udapi currently does not attempt to duplicate this *format validation* because it tries to keep CoNLL-U loading as fast as possible, checking only the most critical properties, such as absence of cycles in dependencies. Udapi can also be used for non-UD treebanks,[11] which use a different set of values for UPOS, DEPREL etc., so a strict non-optional validation is not desired.

UD website features also *content validation* available as an online service.[12] It is basically a special case of querying, with a set of tests formalized as queries. For example, the `multi-obj` test searches for nodes with two or more (direct) objects or clausal complements. This can be implemented in Udapi as follows: `len([n for n in node.children if n.deprel in {'obj', 'ccomp'}]) > 1`.

Although some tests may occasionally bring false alarms (finding a construction which is not forbidden by the UD guidelines), it is worth checking the results of tests with most hits for a given treebank as these often signal real errors or inconsistencies. For example, the two `multi-obj` hits highlighted in Figure 1 are both annotation (or conversion) errors: In the first sentence, *friend* should have deprel *vocative* and *Linda* should depend on it as *appos*. In the second sentence, *it* should be a subject (*nsubj*) instead of object. The tree visualization also sets off the erroneous non-projectivity, where *On the one hand* should depend on *pressure*.

Block `ud.MarkBugs` is an improved version of the online content validation with higher precision and coverage. Treebank developers can apply `ud.MarkBugs` on their data offline (before pushing a new version on GitHub, or on secret test data), so it complements the online validation. It is possible to apply only some tests using parameters `tests` and `skip`:

```
udapy -HAM ud.MarkBugs skip='no-NumType' \
  < in.conllu > bugs.html
```

[11] The reader block `read.Conllu` can load even CoNLL-X and CoNLL-2007 formats, using the optional parameter `attributes` listing the column names.
[12] `http://universaldependencies.org/svalidation.html`

2.6 UDv2 conversion

When the UDv2 guidelines were released, there were many treebanks annotated in the UDv1 style. Luckily, most of the changes could be at least partially automatized. Some of the changes were simple renaming of labels, e.g. CONJ → CCONJ, which is easy to implement in any tool. Some of the changes were more difficult to implement correctly, e.g. conversion of ellipsis from the old *remnant* style to the new *orphan* style.

Block `ud.Convert1to2` has been successfully used for converting five UDv2 treebanks: Bulgarian, Romanian, Galician, Russian and Irish. Block `ud.Google2ud` converts data for 15 languages from a pre-UDv1 style used by Google.

2.7 Other use cases

Block `ud.SetSpaceAfter` uses heuristic rules to add the attribute SpaceAfter=No, while `ud.SetSpaceAfterFromText` does the same based on the raw text. Even more advanced is `ud.ComplyWithText`, which can also adapt the annotation so it matches the raw text, e.g. by reverting the normalization of word forms.[13]

Block `ud.AddMwt` splits multi-word tokens into words based on language-specific rules – there are subclasses for several languages, e.g. `ud.cs.AddMwt` for Czech.

Overall statistics (number of words, empty words, multi-word tokens, sentences) can be printed with `util.Wc`. Advanced statistics about nodes matching a given condition (relative to other nodes) can be printed with `util.See`.

For evaluation, `eval.Parsing` computes the standard UAS and LAS, while `eval.F1` computes Precision/Recall/F1 of various attributes based on the longest common subsequence.

Tree projectivization and deprojectivization (Nivre and Nilsson, 2005) can be performed using `transform.Proj` and `transform.Deproj`.

3 Design and Implementation

The primary focus of Udapi is simplicity of use and speed. The amount of effort spent on designing specialized data structures, micro-optimizing the speed and memory critical parts (e.g. loading CoNLL-U, iterating over all nodes sorted by word order while allowing changes of the word order too) and other technical issues was much bigger than the effort spent on the use cases described in Section 2.

system	memory (MiB)	load (s)	save (s)	bench (s)
Treex	18,024	2,501	201	287
PyTreex	3,809	158	8	74
Udapi-Python	879	24	6	16
Udapi-Perl	**748**	**7**	3	11
Udapi-Java	1,323	9	**1**	**5**

Table 1: Memory and speed comparison. We measured performance of individual implementations on loading and saving from/to CoNLL-U, and on a benchmark composed of iterating over all nodes, reading and writing node attributes, changing the dependency structure, adding and removing nodes, and changing word order. We used `cs-ud-train-l.conllu` from UDv1.2 (68 MiB, 41k sentences, 800k words).

For example, to provide access to structured attributes FEATS and MISC (e.g. `node.feats['Case'] = 'Nom'`) while allowing access to the serialized data (e.g. `node.feats = 'Case=Nom|Person=1'`), Udapi maintains both representations (string and dict) and synchronizes them transparently, but lazily.

Table 1 shows a benchmark of 5 frameworks: Treex (Perl), PyTreex (Python 2) and three implementations of Udapi (Python 3, Perl, Java 8).[14]

The full description of the API is available online.[15] The following two sections summarize only the most important classes and methods.

3.1 Classes for data processing

Block. A block is the smallest processing unit that can be applied on UD data. Block classes implement usually some reasonably limited and well-defined tasks, often corresponding to the classical NLP components (tokenization, tagging, parsing...), but there can be blocks for purely technical tasks (such as for feature extraction).

[13] There are several treebanks which use ``TeX-like quotes'' instead of the "quotes" used in the raw text or which normalize numbers by deleting the thousand separators. However, the UDv2 quidelines require word forms to match exactly the raw text.

[14] `https://github.com/ufal/treex`
`https://github.com/ufal/pytreex`
`https://github.com/udapi/udapi-python`
`https://github.com/udapi/udapi-perl`
`https://github.com/udapi/udapi-java`

[15] `http://udapi.readthedocs.io`

Run. The `Run` class instance corresponds to a sequence of blocks (also called scenario) that are to be applied on data one after another. Such scenarios can compose very complex NLP pipelines. This class offers also the support for the command-line interface `udapy`.

3.2 Classes for data representation

Document. A document consists of a sequence of bundles, mirroring a sequence of sentences in a typical natural language text. A document instance can be composed programatically or can be loaded from (or stored to) a CoNLL-U file.

Bundle. A bundle corresponds to a sentence, possibly in more forms or with different representations, such as sentence-tuples from parallel corpora, or paraphrases in the same language or alternative analyses (e.g. parses produced by different parsers). If there are more trees in a bundle, they must be distinguished by a so called *zone* (a label which contains the language code).

Root. A root is a special (artificial) node that is added to the top of a CoNLL-U tree in the Udapi model. The root serves as a representant of the whole tree (e.g. it bears the sentence's identifier). The root's functionality partially overlaps with functionality of nodes (e.g., it has methods `children` and `descendants`), but differs in other aspects (its lemma cannot be set, its linear position is always 0, it has methods for creating and accessing *multiword tokens*, computing the sentence text (detokenized), accessing the tree-level CoNLL-U comments, etc.).

Node. The `Node` class corresponds to a node of a dependency tree. It provides access to all the CoNLL-U-defined attributes. There are methods for tree traversal (`parent`, `root`, `children`, `descendants`); word-order traversal (`next_node`, `prev_node`); tree manipulation (`parent` setter) including word-order changes (`shift_after_node(x)`, `shift_before_subtree(x)`, etc.); and utility methods: `is_descendant_of(x)`, `is_nonprojective()`, `precedes(x)`, `is_leaf()`, `is_root()`, `get_attrs([])`, `compute_text()`, `print_subtree()`.

Some methods have optional arguments, e.g., `child = node.create_child(form= "was", lemma="be")` for creating a new node with given attributes or `node.remove( children="rehang")` for removing a node but keeping its children by re-attaching them to the parent of the removed node.[16]

4 Related work

Treex (Popel and Žabokrtský, 2010) is a Perl NLP framework focusing on MT and multi-layer annotation in the PDT (Bejček et al., 2013) style. It is the only framework we are aware of with at least partial support for UD and CoNLL-U. NLTK (Bird et al., 2009) is a popular framework focusing on teaching NLP and Python, with no UD support yet.[17] GATE (Cunningham et al., 2011)[18] is a family of Java tools for NLP and IR. There is a converter from CoNLL-U to GATE documents.[19]

Brat (Stenetorp et al., 2012) is an online editor (without full CoNLL-U support) and `conllu.js`[20] is a related JavaScript library used in the embedded CoNLL-U visualizations on the UD website.

5 Conclusion

Most of the current Udapi applications are focused on treebank developers. In future, we would like to focus also on other users, including NLP students, linguists and other researchers to make UD data more useful for them.

We hope Udapi will also serve as a common repository of interoperable NLP tools.

Acknowledgments

This work has been supported by projects GA15-10472S (Manyla), GAUK 1572314, and SVV 260 451. We thank the two anonymous reviewers for helpful feedback.

References

[Bejček et al.2013] Eduard Bejček, Eva Hajičová, Jan Hajič, Pavlína Jínová, Václava Kettnerová, Veronika Kolářová, Marie Mikulová, Jiří Mírovský, Anna Nedoluzhko, Jarmila Panevová, Lucie Poláková, Magda Ševčíková, Jan Štěpánek, and Šárka Zikánová. 2013. Prague Dependency Treebank 3.0. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague.

[16] See footnote 3.

[17] `github.com/nltk/nltk/issues/875`

[18] `https://gate.ac.uk`

[19] `https://github.com/GateNLP/corpusconversion-universal-dependencies`

[20] `http://spyysalo.github.io/conllu.js/`

[Bird et al.2009] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

[Bojar et al.2016] Ondřej Bojar, Ondřej Dušek, Tom Kocmi, Jindřich Libovický, Michal Novák, Martin Popel, Roman Sudarikov, and Dušan Variš. 2016. CzEng 1.6: Enlarged Czech-English Parallel Corpus with Processing Tools Dockered. In Petr Sojka, Aleš Horák, Ivan Kopeček, and Karel Pala, editors, *Text, Speech, and Dialogue: 19th International Conference, TSD 2016*, number 9924 in Lecture Notes in Computer Science, pages 231–238, Cham / Heidelberg / New York / Dordrecht / London. Masaryk University, Springer International Publishing.

[Cunningham et al.2011] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. 2011. *Text Processing with GATE (Version 6)*.

[Nivre and Nilsson2005] Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 99–106, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Popel and Žabokrtský2010] Martin Popel and Zdeněk Žabokrtský. 2010. TectoMT: modular NLP framework. *Advances in Natural Language Processing*, pages 293–304.

[Stenetorp et al.2012] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. 2012. brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107. Association for Computational Linguistics.

[Straka et al.2016] Milan Straka, Jan Hajič, and Jana Straková. 2016. 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'16)*, Paris, France, May. European Language Resources Association (ELRA).