nlppln Documentation

Release 0.3.3

Janneke van der Zwaan

Jan 08, 2019

Contents

1 Flexible and reproducible text processing workflows **3**

2 Contents **5**

2.1 Installation 5
2.2 Creating workflows 6
2.3 Running workflows 10
2.4 Tools 10

nlppln is a python package for creating NLP pipelines using Common Workflow Language (CWL). It provides steps for (generic) NLP functionality, such as tokenization, lemmatization, and part of speech tagging, and helps users to construct workflows from these steps.

A text processing step consist of a (Python) command line tool and a CWL specification to use this tool. Most tools provided by nppln wrap existing NLP functionality. The command line tools are made with Click, a Python package for creating command line interfaces.

To create a workflow, you have to write a Python script:

```python
from nlppln import WorkflowGenerator

with WorkflowGenerator() as wf:
  txt_dir = wf.add_input(txt_dir='Directory')

  frogout = wf.frog_dir(in_dir=txt_dir)
  saf = wf.frog_to_saf(in_files=frogout)
  ner_stats = wf.save_ner_data(in_files=saf)
  new_saf = wf.replace_ner(metadata=ner_stats, in_files=saf)
  txt = wf.saf_to_txt(in_files=new_saf)

  wf.add_outputs(ner_stats=ner_stats, txt=txt)

  wf.save('anonymize.cwl')
```

The resulting workflow can be run using a CWL runner, such as cwltool:

```
cwltool anonymize.cwl --txt_dir /path/to/directory/with/txt/files/
```

For creating new (e.g., project specific) NLP functionality, you can use nlppln-gen to generate boilerplate (i.e., empty) command line tools and CWL specifications.

CHAPTER 1

Flexible and reproducible text processing workflows

One of the problems with existing NLP software is that to combine functionality from different software packages, researchers have to write custom scripts. Generally, these scripts duplicate at least some text processing tasks (e.g., tokenization), and need to be adapted when used for new datasets or in other software or hardware environments. This has a negative impact on research reproducibility and reuse of existing software.

The main advantages of nlppln are:

- Flexibility: it is easy to combine NLP tools written in different programming languages
- Reproducibility and portability: the resulting workflows can be run on a wide variety of hardware environments without changing them

CHAPTER 2

Contents

2.1 Installation

2.1.1 Requirements

Before installing nlppln, please install:

- Python 2 or 3
- node.js
- Docker

For more details about CWL on Windows, see the Windows documentation.

2.1.2 Installing nlppln

Install nlppln using pip:

```
pip install nlppln
```

For development:

```
git clone git@github.com:nlppln/nlppln.git
cd nlppln
pip install -r requirements.txt
python setup.py develop
```

Run tests (including coverage) with:

```
python setup.py test
```

2.2 Creating workflows

Pipelines or workflows can be created by writing a Python script:

```
from nlppln import WorkflowGenerator

with WorkflowGenerator() as wf:
        txt_dir = wf.add_input(txt_dir='Directory')

        frogout = wf.frog_dir(dir_in=txt_dir)
        saf = wf.frog_to_saf(in_files=frogout)
        ner_stats = wf.save_ner_data(in_files=saf)
        new_saf = wf.replace_ner(metadata=ner_stats, in_files=saf)
        txt = wf.saf_to_txt(in_files=new_saf)

        wf.add_outputs(ner_stats=ner_stats, txt=txt)

        wf.save('anonymize.cwl')
```

This workflow finds named entities in all Dutch text files in a directory. Named entities are replaced with their type (PER, LOC, ORG). The output consists of text files and a csv file that contains the named entities that have been replaced.

The workflow creation functionality in `nlppln` is provided by a library called scriptcwl. For a more detailed explanation of how to create workflows, have a look at the scriptcwl documentation.

2.2.1 Setting workflow inputs

Wokflow inputs can be added by calling `add_input()`:

```
txt_dir = wf.add_input(txt_dir='Directory')
```

The `add_input()` method expects a `name=type` pair as input parameter. The pair connects an input name (`txt_dir` in the example) to a CWL type (`'Directory'`). An overview of CWL types can be found in the specification.

Check the scriptcwl documentation to find out how to add optional workflow inputs and default values.

2.2.2 Adding processing steps

To add a processing step to the workflow, you have to call its method on the `WorkflowGenerator` object. The method expects a list of (key, value) pairs as input parameters. (To find out what inputs a step needs call `wf.inputs(<step name>)`. This method prints all the inputs and their types.) The method returns a list of strings containing output names that can be used as input for later steps, or that can be connected to workflow outputs.

For example, to add a step called `frog-dir` to the workflow, the following method must be called:

```
frogout = wf.frog_dir(dir_in=txt_dir)
```

In a next step, `frogout` can be used as input:

```
saf = wf.frog_to_saf(in_files=frogout)
txt = wf.saf_to_txt(in_files=saf)
```

Etcetera.

2.2.3 Listing steps

To find out what steps are available in `nlppln` and to get copy/paste-able specification of what needs to be typed to add a step to the workflow you can type:

```
print(wf.list_steps())
```

The result is:

```
Steps
apachetika............... out_files = wf.apachetika(in_files[, tika_server])
basic-text-statistics.... metadata_out = wf.basic_text_statistics(in_files, out_file)
chunk-list-of-files...... file_list = wf.chunk_list_of_files(chunk_size, in_files)
clear-xml-elements....... out_file = wf.clear_xml_elements(element, xml_file)
copy-and-rename.......... copy = wf.copy_and_rename(in_file[, rename])
docx2txt................. out_files = wf.docx2txt(in_files)
download................. out_files = wf.download(urls)
freqs.................... freqs = wf.freqs(in_files)
frog-dir................. frogout = wf.frog_dir(in_files[, skip])
frog-filter-nes.......... filtered_nerstats = wf.frog_filter_nes(nerstats[, name])
frog-single-text......... frogout = wf.frog_single_text(in_file)
frog-to-saf.............. saf = wf.frog_to_saf(in_files)
ixa-pipe-tok............. out_file = wf.ixa_pipe_tok(language, in_file)
language................. language_csv = wf.language(dir_in)
liwc-tokenized........... liwc = wf.liwc_tokenized(in_dir, liwc_dict[, encoding])
lowercase................ out_files = wf.lowercase(in_file)
ls....................... out_files = wf.ls(in_dir[, recursive])
merge-csv................ merged = wf.merge_csv(in_files[, name])
normalize-whitespace-punctuation metadata_out = wf.normalize_whitespace_
↪punctuation(meta_in)
pattern-nl............... out_files = wf.pattern_nl(in_files)
rename-and-copy-files.... out_files = wf.rename_and_copy_files(in_files)
replace-ner.............. out_files = wf.replace_ner(metadata, in_files[, mode])
saf-to-freqs............. freqs = wf.saf_to_freqs(in_files[, mode])
saf-to-txt............... out_files = wf.saf_to_txt(in_files)
save-dir-to-subdir....... out = wf.save_dir_to_subdir(inner_dir, outer_dir)
save-files-to-dir........ out = wf.save_files_to_dir(dir_name, in_files)
save-ner-data............ ner_statistics = wf.save_ner_data(in_files)
textDNA-generate......... json = wf.textDNA_generate(dir_in, mode[, folder_sequences,␣
↪name_prefix, output_dir])
xml-to-text.............. out_files = wf.xml_to_text(in_files[, tag])

Workflows
anonymize................ ner_stats, out_files = wf.anonymize(in_files[, mode])
```

2.2.4 Setting workflow outputs

When all steps of the workflow have been added, you can specify workflow outputs by calling `wf.add_outputs()`:

```
wf.add_outputs(ner_stats=ner_stats, txt=txt)
```

In this case the workflow has two outputs, one called `ner_stats`, which is a csv file and one called `txt`, which is a list of text files.

2.2.5 Saving workflows

To save a workflow call the `WorkflowGenerator.save()` method:

```
wf.save('anonymize.cwl')
```

Other options when saving workflows are described in the scriptcwl documentation. By default, `nlppln` saves workflows with embedded steps (`inline=True`).

2.2.6 Adding documentation

To add documentation to your workflow, use the `set_documentation()` method:

```
doc = """Workflow that replaces named entities in text files.

Input:
        txt_dir: directory containing text files

Output:
        ner_stats: csv-file containing statistics about named entities in the text␣
↪files
        txt: text files with named enities replaced
"""
wf.set_documentation(doc)
```

2.2.7 Loading processing steps

`nlppln` comes with nlp functionality pre-loaded. If you need custom processing steps, you can create them using nlppln-gen. To be able to add these custom processing steps to you workflow, you have to load them into the `WorkflowGenerator`. To load a single CWL file, do:

```
wf.load(step_file='/path/to/step_or_workflow.cwl')
```

The `step_file` can also be a url.

To load all CWL files in a directory, do:

```
wf.load(steps_dir='/path/to/dir/with/cwl/steps/')
```

2.2.8 Using a working directory

Once you need more functionality than nlppln provides, and start creating your own processing steps, we recommend using a CWL working directory. A CWL working directory is a directory containing all available CWL specifications. To specify a working directory, do:

```
      from nlppln import WorkflowGenerator

with WorkflowGenerator(working_dir='path/to/working_dir') as wf:
  wf.load(steps_dir='some/path/')
  wf.load(steps_dir='some/other/path/')

  # add inputs, steps and outputs
```

If you use a working directory when creating pipelines, nlppln copies all CWL files to the working directory.

To copy these files manually, you can also use the `nlppln_copy_cwl` command on the command line:

```
nlppln_copy_cwl /path/to/cwl/working/dir
```

To copy CWL files from a different directory than the one containing the nlppln CWL files, do:

```
nlppln_copy_cwl --from_dir /path/to/your/dir/with/cwl/files /path/to/cwl/working/dir
```

If you use a working directory, please save your workflow using the `wd=True` option:

```
wf.save('workflow.cwl', wd=True)
```

The workflow is saved in the working directory and then copied to you specified location. Subsequently, the workflow should be run from the working directory.

2.2.9 Tips and tricks

Create workflows you can run for a single file

If you want to create a workflow that should be applied to each (text) file in a directory, create a workflow that performs all the steps to a single file. Then, use this workflow as a subworkflow that is scattered over a list of input files:

```
from nlppln import WorkflowGenerator

with WorkflowGenerator(working_dir='path/to/working_dir') as wf:
  wf.load(steps_dir='some/path/')

  in_dir = wf.add_input(in_dir='Directory')

  in_files = wf.ls(in_dir=in_dir)
  processed_files = wf.some_subworkflow(in_file=in_files, scatter='in_file', scatter_
↪method='dotproduct' [, ...])

  wf.add_outputs(out_files=processed_files)
```

Having a workflow you can run for a single file makes it easier to test the workflow.

Test your workflow by running it for the largest or otherwise most complex file

By running your workflow for the largest or otherwise most complex file, you can identify problems, such as excessive memory usage, early and/or before running it for all files in your dataset. There may, of course, still be problems with other files, but starting analysis with the largest file is easy to do.

Use `create_chunked_list` and `ls_chunk` to run a workflow for a subset of files

Sometimes running a workflow for all files in a directory takes too long, and you'd like to run it for subsets of files. Using `create_chunked_list`, you can create a JSON file containing a division of the files in a directory in chunks. You can then create a workflow that, instead of using `ls` to list all files in a directory, uses `ls_chunk` that runs the workflow for a single chunk of files.

To create a division of the input files, do:

```
python -m nlppln.commands.create_chunked_list [--size 500 --out_name output.json] /
↪path/to/directory/with/input/files
```

The result is a JSON file named `output.json` that contains numbered chunks containing 500 files each.

To run a workflow for a chunk of files, instead of all files in a directory, do:

```
from nlppln import WorkflowGenerator

with WorkflowGenerator(working_dir='path/to/working_dir') as wf:
  wf.load(steps_dir='some/path/')

  in_dir = wf.add_input(in_dir='Directory')
  chunks = wf.add_input(chunks='File')
  chunk_name = wf.add_input(name='string')

  in_files = wf.ls_chunk(in_dir=in_dir, chunks=chunks, name=chunk_name)
  processed_files = wf.some_subworkflow(in_file=in_files, scatter='in_file', scatter_
↪method='dotproduct' [, ...])

  wf.add_outputs(out_files=processed_files)
```

2.3 Running workflows

To run a workflow created with nlppln, you need to use a CWL runner. The default CWL runner cwltool is installed when you install nlppln. To run a tool:

```
cwltool <workflow> <inputs>
```

The scriptcwl documentation contains some tips and tricks for working with CWL files.

2.3.1 Windows

To run a workflow created with nlppln on Windows, use the nlppln Docker container:

```
cwltool --default-container nlppln:nlppln <workflow> <inputs>
```

Also, if you have to refer to file paths with spaces in them, use the `--relax-path-checks` option.

For more details about CWL on Windows, see the Windows documentation.

2.4 Tools

`nlppln` contains the following tools:

2.4.1 anonymize.cwl

Replace named entities in a directory of text files.

Can be used as part of an data anonymization workflow.

2.4.2 apachetika.cwl

Convert Word documents to text using Apache Tika.

2.4.3 archive2dir.cwl

Extract archive.

To recursively put all files in the output directory, use the *–remove-dir-structure* option.

Uses Patool for extracting archives.

2.4.4 basic-text-statistics.cwl

Output a csv file with basic text statistics (#tokens, #sentences).

2.4.5 check-utf8.cwl

Convert text files to utf-8 encoding.

Uses BeautifulSoup's Unicode, Dammit module to guess the file encoding if it isn't utf-8.

2.4.6 clear-xml-elements.cwl

Empty (i.e. remove all content from) specified XML elements in the XML file.

2.4.7 copy-and-rename.cwl

Copy a file and optionally rename it.

File renaming options are: `copy` (don't rename), `spaces` (remove spaces, default), and `random` (generate a random file name. The file extension is copied too.) If the renaming option is spaces, this tool must be run with the `--relax-path-checks` option, because it accepts file names with spaces, which CWL normally does not accept.

2.4.8 create-chunked-list.cwl

No documentation

2.4.9 delete-empty-files.cwl

No documentation

2.4.10 filter-nes.cwl

Control which named entities will be removed.

See *replace-ner.cwl*.

2.4.11 flatten-dirs.cwl

Given a list of directories, return a directory that contains all the files in the input directories.

By default the name of the output directory is *flattened*. You can specify a different name using the *–dir_name* option.

2.4.12 flatten-list.cwl

No documentation

2.4.13 freqs.cwl

Return a sorted list of word frequencies in the corpus.

The corpus should consist of files containing space-separated tokens.

2.4.14 frog-dir.cwl

Frog a directory of text files.

2.4.15 frog-single-text.cwl

Frog a single text file.

2.4.16 frog-to-saf.cwl

Convert frog csv output to saf.

2.4.17 gather-dirs.cwl

Given a list of directories, return a directory that contains all the files in the input directories.

By default the name of the output directory is *gathered*. You can specify a different name using the *–dir_name* option.

2.4.18 ixa-pipe-tok.cwl

Tokenize a text using ixa-pipe-tok.

2.4.19 liwc-tokenized.cwl

Apply LIWC to a directory of tokenized text files.

The text files have to contain space separated tokens.

2.4.20 lowercase.cwl

Lowercase a text.

2.4.21 ls.cwl

List files in a directory.

This command can be used to convert a `Directory` into a list of files. This list can be filtered on file name by specifying `--endswith`.

2.4.22 ls_chunk.cwl

No documentation

2.4.23 merge-csv.cwl

Merge csv files (with the same header) into a single csv file.

2.4.24 merge-yaml.cwl

No documentation

2.4.25 mkdir.cwl

Create directory

2.4.26 normalize-whitespace-punctuation.cwl

Normalize whitespace and punctuation.

Replace multiple subsequent occurrences of whitespace characters and punctuation with a single occurrence.

2.4.27 prettify-xml.cwl

Pretty print xml file.

Uses BeautifulSoup pretty printing.

2.4.28 remove-newlines.cwl

Remove newlines from a text.

2.4.29 remove-xml-elements.cwl

Remove specified XML elements from XML file.

2.4.30 replace-ner.cwl

Replace named entities in saf files.

Named entities can be replaced with their type or deleted.

2.4.31 saf-to-freqs.cwl

Return csv file wit a ranked list of (word, pos) pairs.

The list can be of (word, pos) pairs of (lemma, pos) pairs.

2.4.32 saf-to-txt.cwl

Convert saf to space separated tokens.

2.4.33 save-dir-to-subdir.cwl

Save a directory to a subdirectory.

Puts `inner_dir` into the `outer_dir`.

2.4.34 save-files-to-dir.cwl

Save a list of files to a directory.

If the `dir_name` is not specified, it is set to the string before the rightmost - of the `nameroot` of the first input file (e.g., `input-file-1-0000.txt` becomes `input-file-1`). If the file name does not contain a -, the `nameroot` is used (e.g. `input.txt` becomes `input`).

2.4.35 save-ner-data.cwl

Create csv file with statistics about named entities.

By editing the csv file, you can control which named entities are replaced or removed using *replace-ner.cwl*.

2.4.36 tar.cwl

Extract zipped tar archives.

2.4.37 textDNA-generate.cwl

Generate data to vizualize using TextDNA.

2.4.38 xml-to-text.cwl

Extract text from an XML element and save it to a file.

2.4.39 zip-dir-flat.cwl

Compress a directory into a zip archive.

All structure is removed from the input directory. So, if you unzip the archive, you get a flat list of files.