nlgen Documentation

Release 0.1

Yusuke Tsutsumi

May 12, 2017

Contents

Warning: nlgen is currently in a very early alpha. There is currently no guarantee about the stability of any API it exposes.

You can inquire more about the status of it's APIs by filing an issue: https://github.com/toumorokoshi/nlgen/issues

nlgen is a set of tools for natural language generation.

nlgen currently includes:

- a featured context-free grammar

Contents:

CHAPTER 1

FCFG Tutorial

A step by step walk-through around the features of nlgen's context free grammar.

This tutorial shows creating a grammar using the markup language. However, the API is also available for programmatic generation of grammars.

Basic Markup Syntax

The markup language for FCFGs is a list of productions, in the format of:

```
<PRODUCTION_NAME> -> <PRODUCTION>;
```

- production name on the left-hand-side
- the symbol "->"
- the production itself on the right-hand-side
- a semicolon at the end.

comments should be denoted with a leading semicolon:

```
# this is a comment
```

Warning: the current grammar parse is not very informative on incorrect grammars. Thus, please check your syntax carefully.

Basic Production Types

Terminal

Terminal's represent an actual string, token. For example, let's add a pronoun, a verb, and a noun:

```
PRONOUN -> "I";
VERB -> "play";
NOUN -> "blackjack";
```

A token is parsed as a terminal when:

- double quotes wrap the string

References

A production can reference another production. For example, if wanted to alias "VI" to "VERB":

```
VI -> VERB;
```

A token is parsed as an alias when:

- no double quotes wrap a string

List

A production list exists in the case where you want to compose multiple productions together. For example, we can define a basic sentence as:

```
SENTENCE -> PRONOUN VI NOUN;
```

A production is parsed as list when:

- multiple productions tokens occur before a semicolon

Unions

Unions allow multiple expressions to match a single rule. Let's expand the nouns a bit with this:

```
NOUN -> "blackjack" | "poker"
```

A production is parsed as a union when:

- multiple productions occur before a semicolon, separated by a "|"

Unions are also generated automatically, if multiple identical production names are used:

```
# this works too.
NOUN -> "blackjack"
NOUN -> "poker"
```

Note: unions take operator precedence over lists. Thus, "FOO BAR | BAZ" resolves as a union of the production list "FOO BAR", or "BAZ", not a production list with a union.

At this point, a nlcfg file may look like:

```
# example.nlcfg
SENTENCE -> PRONOUN VI NOUN;
VI -> VERB;
PRONOUN -> "I";
VERB -> "play";
NOUN -> "blackjack" | "poker";
```

You can now load and use this to generate a sentence! The simple example is:

```
from nlgen.cfg import read_cfg

EXAMPLE_NO_FEATURE = """
# example.nlcfg
SENTENCE -> PRONOUN VI NOUN;
VI -> VERB;
PRONOUN -> "I";
VERB -> "play";
NOUN -> "blackjack" | "poker";
""".strip()


def test_no_feature_example():
    cfg = read_cfg(EXAMPLE_NO_FEATURE)
    # permutations returns a generator.
    # we use sets for comparisons as there's
    # no guaranteed order for generated
    # values.
    assert set(cfg.permutation_values("SENTENCE")) == set([
        ("I", "play", "blackjack"),
        ("I", "play", "poker"),
    ])
```

Features

We now know how to generate basic sentences, but our use case is very limited. For example, how do we allow for sentences that are for more than the first person? If we want to generate sentences for the second person, we could just add the "you": terminal:

```
SENTENCE -> PRONOUN VI NOUN;
VI -> VERB;
PRONOUN -> "I" | "you";
VERB -> "play";
NOUN -> "blackjack" | "poker";

# this will generate:
- I play blackjack
- You play blackjack
- I play poker
- You play poker
```

But how about the third person, e.g. "she"?:

```
SENTENCE -> PRONOUN VI NOUN;
VI -> VERB;
```

```
PRONOUN -> "I" | "you" | "she" ;
VERB -> "play";
NOUN -> "blackjack" | "poker";

# this will generate:
- I play blackjack
- you play blackjack
- she play blackjack
- I play poker
- you play poker
- she play poker
```

Unfortunately, this results in some improper english sentences:

- she play poker
- she play blackjack

The lack of verb-subject agreement makes this sentence improper. Really, we would want to replace "play" with "plays", but we need to make sure it's specifically for the third person. Thus, some terminals should only be used for other specific terminals.

Enter features. Features can attach attributes to a terminal. the CFG in nlgen requires that the features match: mismatches will be thrown out.

Feature Syntax

To define features, add a json dictionary after the terminal, of string -> string_or_list pairs:

```
# the value can be a single string, or a list of strings.
VERB -> "play" {"person": ["1", "2"]} |
        "plays": {"person": 3};

# all values are coerced to a string, such as the 1 here.
PRONOUN -> "I" {"person": 1} |
           "you" {"person": "2"} |
           "she" {"person" 3}
```

When generating sentences, combinations of "you" and "plays" will be invalid, as they don't agree on their person feature.

- features with a list of values will match as long as any of the values match.

Final Example

So our final example looks like:

```
# example.nlcfg
SENTENCE -> PRONOUN VI NOUN;
VI -> VERB;
PRONOUN -> "I" {"person": 1} |
           "you" {"person": "2"} |
           "she" {"person": 3};
VERB -> "play"  {"person": ["1", "2"]} |
        "plays" {"person": 3};
NOUN -> "blackjack" | "poker";
```

and generates the set:

```
("I", "play", "blackjack"),
("you", "play", "blackjack"),
("she", "plays", "blackjack"),
("I", "play", "poker"),
("you", "play", "poker"),
("she", "plays", "poker"),
```

CHAPTER 2

API

Due to the heavy flux of the apis, there is currently no explicit documentation.

However, there is a fairly extensive unit testing suite. It's recommended to look at those for now:

- CFG example: https://github.com/toumorokoshi/nlgen/blob/master/nlgen/tests/cfg/test_full.py
- Using the text representation of a CFG: https://github.com/toumorokoshi/nlgen/blob/master/nlgen/tests/cfg/test_markup_parser.py

CHAPTER 3

Indices and tables

- genindex
- modindex
- search