Random Sentence Generation with Grammars v2.0🐙

McMaster University | Computing & Software

CS 4TB3 Winter 2020, Dr Sekerinski
Group 8: Kael Boseland, Hamid Yuksel, Chen Chen

Proposal

What:
After being given a grammar G, be able to create a random sentence s, or a set of sentences S.

However there are two extensions to be added:
- add probabilities for alternatives in productions
- context information to make the output from the grammar G well-typed

Why:
To create random sentences to be used for testing. Specifically, testing a parser or a natural language processor. As well as artistic purposes, such as generating 'poetry'.

How:
Using Python 3.5, we decided to create a library that allows a user to create a custom context free grammar (CFG), or a context sensitive grammar[1] (CSG) . Of course, we would add our own extensions to account for probability.

e.g. a production, S -> A | B, becomes, S -> A @ 60% | B @ 40%

Solution

To build this library, we needed to a way to represent the following:
- a grammar G
- a production P
- repetition (with a min and max set by the user)
- optional (0 or 1 of a production, [...])
- grouping (for context, (...))
- alternation (A | B)
- probability in alternation (A @ 70% | B @ 20% | C @ 10%)
- the end output list of randonmly generated sentences

Assumptions:
- only production rules with alternation have the probability weights extension
- the sum of the probability weights total ~100% for any single production
- typing is specified by the user, thereby preserving it by never changing it

Grammar G

To create a ***context free grammar***:

```
grammar = CFGrammar(non_terminals, terminals, start_symbol, productions)

non_terminals = [“S”, “A”, “B”]
terminals = [“dog”, 3.3, “3.4”, 400, “100”]
start_symbol = “S”
productions = [Production(nonterminal, [rules])]
```

Rules are the sequences of production for a given nonterminal. The user provides a list of objects which can be of any type (including their own). Since we never alter the typing, typing is thereby preserved as what the user specifies.

To create a ***context sensitive grammar:***

```
grammar = CSGrammar(non_terminals, terminals, start_symbol, productions)

non_terminals = [“S”, “A”, “B”]
terminals = [“dog”, 3.3, “3.4”, 400, “100”]
start_symbol = [“S”]
productions = [SensitiveProduction([lhs_rules], [rhs_rules])]
```

To model a CSG, we can use the SensitiveProduction object. This object is specifically for CSG's as it takes a list to allow the user to specify a sequence of terminals and nonterminals for the LHS. Once again, users specify type, allowing typing to be preserved throughout generation. Also, take note that the start symbol for CSG is now wrapped in a list. For all grammars, the nonterminals must be specified as a string only.

Production P

For CFG: to create a single production, A -> "abc" 100 "3.3" 4.5

```
A = Production(“A”, [“abc”, 100, “3.3”, 4.5])
```

For CSG: to create a single production, "3.3" A 100 -> "abc" 4.5

```
3_3A100 = SensitiveProduction([“3.3”,“A”,100], [“abc”, 100, “3.3”, 4.5])
```

Notice, the difficulty that comes from naming sensitiveProduction objects with float terminals.
Luckily, the names do not affect the logic, only readability. Of course, the LHS for sensitiveProduction objects can consist of nonterminals as well:

For CSG: to create a similar single production, B A "a" -> "abc" 4.5

```
BAa = SensitiveProduction([“B”, “A”, “a”], [“abc”, 100, “3.3”, 4.5])
```

The following are some functionalities which can be added to a CFG or CSG. Since the RHS input can have other objects or even nested lists , it will will not invoke the object or go through the nested lists, treating them as items. Below are the grammar specific objects which will be invoked during generation.

Repetition:

```
R = Repeat([“a”],min,max) #min and max number of repeats desired
```

```
A = Production(“A”, R) #models A = {‘a’}
```

Optional:

```
B = Production(“B”, [Optional(“b”)]) #models B = [‘b’]
```

Alternation:

```
C = Production(“C”, [Alternation([Alternate(“c1”, 0.2), Alternate(“c2”, 0.8)])])
#models C = c1 @ 20% | c2 @ 80%
```

```
C = SensitiveProduction([“C”,”c”], [Alternation([Alternate(“c1”, 0.2), Alternate(“c2”, 0.8)])])
#models Cc = c1 @ 20% | c2 @ 80%
```

Notice for alternation, it is a list of alternate objects. This was done so that the probabilities could be quickly accessed for random sentence generation.

Of course, we can also nest these functionalities to create complex productions or sensitiveProductions.

```
C = Production(“C”, [Optional(Repeat(Alternation([Alternate(“c”, 1.0)])))]
#models C = [{‘c’ @ 100%}]
```

```
Cc = SensitiveProduction([“C”,”c”], [Optional(Repeat(Alternation([Alternate(“c”, 1.0)])))]
#models Cc = [{‘c’ @ 100%}]
```

Generating random sentences

After we construct our grammar and productions, we can finally randomly generate our sentences.

```
grammar.genRandSentence(start_symbol)
```

During generation, we apply type checks to see if we need to convert the string terminals into the appropriate float or integer type. This is then all stored in a list you can access as:

`grammar.output` and printed through `Output(grammar.output).returnString()`

Testing with Poetry

We made sure to test out our functionalities and their probabilities. One fun way we did this was through modeling poetry, by modeling a basic English grammar[2] with our library.

🌺
```
a lone flower
smiles against
  the tsunami
```

```
a bright volcano
 resides before
the fiery ocean
```
🌊

Conclusion, Challenges, and the Future

To wrap up, our GrammarTools.py library is capable of generating random sentences with context free or sensitive grammars.

The biggest challenge was in implementing probability, as well as creating an easy to use framework for the user to model their grammars.

Future goals include improving efficieny as for CSG we must match all possible RHS's to the sequence before randomly choosing a valid production, which runs in O(n^2) time.

Endnotes
1 Peter Linz, page 291, An Introduction to Formal Languages and Automata
2 Harry Schwartz, https://github.com/hrs/grammar-based-sentence-generator/blob/master/grammar.lisp