NLTK Tutorial: Parsing

Steven Bird

Edward Loper

Table of Contents

1. Introduction

Note: This section is to be written. For now, please see other published discussions of parsing, such as Jurafsky and Martin, Chapter 10

Warning

Some of the material in this tutorial is out of date. We plan to re-write the parsing tutorials in the near future (this tutorial will probably be split into two tutorials: a basic parsing tutorial, and a chart parsing tutorial). Until then, see the reference documentation for `nltk.parser` for more up-to-date information.

2. Grammars and Lexicons

A *grammar* is a formal specification for the structure of well-formed sentences in some language. At present, only context-free grammars (CFGs) can be represented in NLTK. A CFG consists of a set of context-free rules. Context-free rules have the form `X -> Y` where `X` is a non-terminal, and `Y` is a list of terminals and non-terminals. `X` and `Y` are known as the left-hand side and right-hand side respectively.

In the simplest case, non-terminals and terminals are just Python strings. However, it is possible to use any immutable Python object as a non-terminal or terminal.

Note: The NLTK chart parser makes two additional assumptions about rules. First, if a terminal symbol occurs on the right hand side of a rule, it must be the only element on the right hand side. Second, terminal symbols must be Python strings, for they are matched against the `type` attribute of a token. Thus, the form of "lexical rules" must be `N -> 'cat'`. Using rules like `NP -> 'the' N` will produce unintended results (either `the` will be treated as the name of a non-terminal, of `N` will be ignored). In general, this limitation on the form of lexical rules does not pose any problems.

A grammar can be represented as a tuple of rules, while a lexicon can be represented as a tuple of lexical rules. The only class we need to define then is `Rule`.

2.1. Rules

The `nltk.rule` module defines the `Rule` class, which is used to represent context free rules. A `Rule` consists of a left-hand side and a right-hand side.

- The left-hand side is a single non-terminal, which may be any Python object. In the simplest case it is just a string (e.g. "NP" or "VP").
- The right-hand side is a tuple of non-terminals and terminals, which may be any Python object. In the simplest case these are strings (e.g. "Det", "the").

> **Note:** For the NLTK chart parser, the right-hand side of a rule must be either a tuple of non-terminals, or a tuple consisting of exactly one terminal (e.g. ("the",))

`Rule`s are created with the `Rule constructor`, which takes a left-hand side and a right-hand side:

```
# A typical grammar rule S -> NP VP:
>>> rule1 = Rule('S', ('NP', 'VP'))
S -> NP VP

# A typical lexical rule Det -> 'the':
>>> rule2 = Rule('Det', ('the',))
Det -> the
```

A `Rule`'s left-hand side and right-hand side are accessed with the `lhs` and `rhs` methods:

```
>>> rule1.lhs()
'S'

>>> rule2.rhs()
('the',)
```

A `Rule`'s right-hand side can also be accessed with standard sequence operators:

```
>>> rule1[0]
'NP'

>>> rule2[1]
IndexError: tuple index out of range

>>> len(rule1)
2

>>> 'the' in rule2
1

>>> for cat in rule1:
...     print cat
NP
VP
```

2.2. Building Grammars and Lexicons from Rules

Grammars and Lexicons can easily be built up from `Rule`s as shown in the following examples:

```
# A simple grammar:

grammar = (
    Rule('S',('NP','VP')),
    Rule('NP',('Det','N')),
    Rule('NP',('Det','N', 'PP')),
    Rule('VP',('V','NP')),
    Rule('VP',('V','PP')),
    Rule('VP',('V','NP', 'PP')),
    Rule('VP',('V','NP', 'PP', 'PP')),
    Rule('PP',('P','NP'))
)


# A simple lexicon:

lexicon = (
    Rule('NP',('I',)),
    Rule('Det',('the',)),
    Rule('Det',('a',)),
    Rule('N',('man',)),
    Rule('V',('saw',)),
    Rule('P',('in',)),
    Rule('P',('with',)),
    Rule('N',('park',)),
    Rule('N',('telescope',))
)
```

2.3. Dotted Rules

The `nltk.rule` module defines the `DottedRule` class, which is used to represent the dotted rules used by a chart parser. A `DottedRule` consists of a left-hand side, a right-hand side, and the dot position.

`DottedRule`s are created with the `DottedRule constructor`, which takes a left-hand side, a right-hand side, and a position:

```
# A dotted rule with position 0 (default value omitted):
>>> dr1 = DottedRule('S', ('NP', 'VP'))
S -> * NP VP

# A dotted rule with position 0 (default value supplied):
```

```
>>> dr1 = DottedRule('S', ('NP', 'VP'), 0)
S -> * NP VP

# A dotted rule with position 1:
>>> dr2 = DottedRule('S', ('NP', 'VP'), 1)
S -> NP * VP

# A dotted rule with position 2:
>>> dr3 = DottedRule('S', ('NP', 'VP'), 2)
S -> NP VP *
```

Another way to construct a `DottedRule` is from a `Rule`. The `Rule` class has a `drule` member function which returns the dotted version of the rule (with position 0).

```
>>> rule1 = Rule('S', ('NP', 'VP'))
>>> rule1.drule()
S ->  * NP VP
```

`DottedRule` inherits the member functions of `Rule`. In addition, it defines `pos` which returns the dot position, `next` which returns the next right-hand side element following the dot, and `shift` which returns a new rule with the dot shifted one position to the right.

```
>>> dr1.pos()
0

>>> dr1.next()
'NP'

>>> dr1.shift()
S -> NP * VP

>>> dr3.shift()
IndexError: Attempt to move dot position past end of rule
```

`DottedRule` defines a function `complete` which tests to see if the dotted rule is complete (i.e. the dot is in the rightmost position).

```
>>> dr1.complete()
0

>>> dr3.complete()
0
```

3. Encoding Syntax Trees

> **Note:** The `nltk.tree` module currently has only minimal support for representing movement, traces, and co-indexing. We plan to extend the class to support these features more fully in the future.

3.1. Trees

The `nltk.tree` module defines the `Tree` class, which is used to represent syntax trees. A `Tree` consists of a *node value*, and one or more *children*.

- The node value is a string containing the tree's constituant type (e.g., "NP" or "VP").
- The children encode the hierarchical contents of the tree. Each child is either a *leaf* or a *subtree*.

> **Note:** Although the `Tree` class is usually used for encoding syntax trees, it can be used to encode *any* homogenous hierarchical structure that spans a text (such as morphological structure or discourse structure). In the general case, leaves and node values do not have to be strings.

A `Tree` with node value n and children $c_1, c_2, \ldots c_n$ is written $(n: c_1, c_2, \ldots c_n)$. `Tree`s are created with the `Tree constructor`, which takes a node value and zero or more children:

```
# A tree with one child, a leaf:
>>> tree1 = Tree('NP', 'John')
('NP': 'John')

# A tree with two children, both of which are leaves:
>>> tree2 = Tree('NP', 'the', 'man')
('NP': 'the' 'man')

# A tree with two children, one leaf and one subtree:
>>> tree3 = Tree('VP', 'saw', tree2)
('VP': 'saw' ('NP': 'the' 'man'))
```

A `Tree`'s node value is accessed with the `node` method:

```
>>> tree1.node()
'NP'
```

A `Tree`'s children are accessed with standard sequence operators:

```
>>> tree3[0]
'saw'
>>> tree3[1]
('NP': 'the' 'man')
>>> len(tree3)
2
>>> 'saw' in tree3
1
>>> for child in tree3:
...     print child
saw
('NP': 'the' 'man')
>>> [child.upper() for child in tree2]
['THE', 'MAN']
>>> tree3[:]
('saw', ('NP': 'the' 'man'))
```

The printed representation for complex `Tree`s can be difficult to read. In these cases, the `draw` method can be very useful. This method opens a new window, containing a graphical representation of the tree.

```
>>> tree3.draw()
```

The tree display window allows you to zoom in and out; to collapse and expand subtrees; and to print the graphical representation to a postscript file.

The `Tree` class implements a number of other useful methods. See the `Tree` reference documentation for more information about these methods.

```
>>> tree3.leaves()
('saw', 'the', 'man')
>>> tree3.height()
3
>>> tree3.nodes()
('VP': ('NP':))
```

3.2. Tree Tokens

NLTK makes a distinction between *tree type*s and *tree token*s, that is analogous to the distinction between word types and word tokens. In particular, a tree token is an individual occurance of a tree type in a text; and a tree type is an abstract syntax tree, without context.

Tree tokens are represented with the `TreeToken` class. `TreeToken`s behave very much like `Tree`s, except that the leaves of a `TreeToken` are word tokens, not word types.

4. The Parser Interface

The `parser` module defines the `ParserI` class, which is the standard interface which all parsers should support. This class should only be used in the definition of other parser classes, which should inherit from the `ParserI` class.

The `ParserI` class defines `parse`, which takes a list of tokens as its argument and returns a list of `TreeTokens`. The `parse` function has an optional second argument which specifies the maximum number of parses to return. No program should call the `ParserI.parse` method. Derived classes must implement their own `parse` method.

A parser returns an empty list of trees in the situation where it was unable to assign a tree to the list of tokens. This happens in situations where the list of tokens forms an ungrammatical sentence, or when the grammar is deficient, or when the search strategy of the parser does not explore the section of the search space where the parse tree(s) are found.

A parser returns more than one tree in the case of syntactic ambiguity. The sentence being parsed may be genuinly ambiguous, or the grammar may be deficient. Statistical parsers usually return the most likely parse (based on training data), or the set of *n* most likely parses. The same interface applies to these cases. (Where the probability of the parse tree needs to be returned, we assume this is stored in the root node of the tree.) When asked to return *n* parses, we assume such a parser will rank parses in order of decreasing likelihood, and return the *n*-best parses.

Parsers may be used anywhere where there is need for processing sequences with a grammar. In the most common case, the sequence consists of a string of words forming a sentence. However, other cases are possible, e.g. the string of characters forming a syllable, the string of morphemes forming a word, the string of sentences forming a text. We could also have grammars over subsequences. For example, the collection of XML elements in a document forms a subsequence of the document, and we could specify a grammar over those elements alone, ignoring document content.

The `ParserI` class also defines a `parse_types` method, which takes a list of token types (e.g. strings) and returns a list of `Trees`. This is useful in the case where the sentence to be parsed did not come from a tokenized text. This method works by converting the list of types into a list of tokens, then calling `parse` as before.

5. Simple Parsers

Simple top-down and bottom-up parsers can easily be defined using classes which inherit from `ParserI`. Here is some pseudocode to use as the basis of two simple modules. Writing these modules is left as an exercise for the reader. More work is required in order to return trees.

```
# elementary top-down parser
def tdparse(goal, sent):
```

```
    if goal == sent == empty:
        pass # we're finished
    else:
        if goal[0] == sent[0]:
    tdparse(goal[1:], sent[1:])
        else:
            for rule in grammar:
                if rule.lhs() == goal[0]:
                    make a local copy of the goal list
                    goal[:1] = rule.rhs()
                    tdparse(goal, sent)

# left-corner parser
def lcparse(goal, sent):
    # this is like tdparse, except the step which iterates over the rules
    # of the grammar only considers rules whose "left corner" matches the next
    # word of the input stream.  The left corner of a lexical rule is the
    # content of the rule.  The left corner of a non-lexical rule R is the left
    # corner of R[0], the first element on the RHS of the rule.

# elementary bottom-up parser
def buparse(sent):
    if sent == [S]:
        pass # we're finished
    else:
        for rule in grammar:
    does rule.rhs() match any sublist of sent?
            if so, replace the substring with the LHS of the rule
            buparse(sent)
```

The `srparser_template` module defines the `SRParser` class, which is a template for a shift-reduce parser.

6. Chart Parsing

The elementary parsers discussed above suffer from a major efficiency problem. During processing, they typically build and discard the same structure many times over. The standard solution to this general kind of problem involves dynamic programming in which intermediate results are kept on a blackboard (the chart) for later reuse.

In general, a parser hypothesizes constituents based on the the grammar and on the current state of its knowledge about the constituents which are already attested. Any given constituent may be proposed during the search of a blind alley; there is no way of knowing at the time whether the constituent will be used in any of the final parses which are reported. Locally, what we know about a constituent is the C{Rule} which licenses its node and immediate daughter nodes (i.e. the "local tree"). Also, in the

case of chart parsing (which is related to bottom-up parsing), we know the whole subtree of this node, how it connects to the tokens of the sentence being parsed, and its span within the sentence (i.e. location). In a chart parser, these these three things: rule, subtree and location, are stored in a special structure called an edge.

6.1. Edges

In order to understand how edges work, it is first necessary to recall NLTK's notion of `Location`. A location in a list of tokens is analgous to a Python slice. The third token has the location `@[2:3]`. It is helpful to think of the numbers as referring not to tokens but to imaginary points between the tokens (the "interstices"). Each location is then like an edge connecting two adjacent points. This is where the name "edge" comes from.

It is easy to generalize the idea of edges. The location which spans the third and fourth tokens `@[2:4]`, is essentially a longer edge which connects non-adjacent points.

The notion of an edge is best communicated graphically. The following code sample creates some edges for the phrase `the park`, displaying them in text on the screen, and also using tkinter.

```
from nltk.chartparser import *

tok_sent = [Token('the', 0), Token('park', 1)]
loc = Location(0,2)
chart = Chart(loc)

dr0 = DottedRule('Det', ('the',), 1)
tt0 = TreeToken(dr0.lhs(), tok_sent[0])
edge0 = Edge(dr0, tt0, Location(0))
chart.insert(edge0)

dr1 = DottedRule('N', ('park',), 1)
tt1 = TreeToken(dr1.lhs(), tok_sent[1])
edge1 = Edge(dr1, tt1, Location(1))
chart.insert(edge1)

dr2 = DottedRule('NP', ('Det','N'), 2)
tt2 = TreeToken(dr2.lhs(), tt0, tt1)
edge2 = Edge(dr2, tt2, Location(0,2))
chart.insert(edge2)

chart.draw()

from nltk.draw.chart import *
ChartView(Tkinter.Tk(), chart, tok_sent)
Tkinter.mainloop()
```

The purpose of an edge is to store a *hypothesis* about the syntactic structure of the sentence. Such hypotheses may be either complete or partial. A complete hypothesis is represented as an edge decorated with a `DottedRule` where the dot position is on the right edge of the rule. This means that the entire right-hand side has been processed and that the associated `TreeToken` is complete. A partial hypothesis is represented as an edge decorated with a rule where the dot position is in an internal position, e.g. `NP -> Det * N`. This means that only part of the right-hand side has been processed and that the associated tree is incomplete. The symbol immediately to the right of the dot position is the next symbol (terminal or non-terminal) to be processed for this edge.

There is a special case where edges have zero width. Such an edge is actually a self-loop, starting and ending at the same index position. Zero-width edges represent the hypothesis that a syntactic constituent *begins* at this location.

In the process of chart parsing, edges are combined with other edges to form new edges which are then added to the chart. Note that nothing is ever removed from the chart, and nothing is ever modified once it is in the chart.

6.2. Charts

A chart is a set of edges and a location. The edge set represents the state of a chart parser during processing, and new edges can be inserted into the set at any time. Once entered, an edge cannot be modified or removed. The chart also stores a location. This represents the combined span of the list of input tokens. The chart uses this information in order to return edges which span the entire sentence.

In NLTK, a chart is implemented by the Chart class. Full details of the methods are available online. The key methods to note are `insert` and `parses`.

6.3. Chart Rules and Parsing Strategies

The ChartParserStrategy class encodes the list of rules used by a chart parser to specify the conditions under which new edges should be added to the chart. There are two kinds of chart rules:

- Static Rules: these search the chart for specific contexts where edges should be added. The parse method will repeatedly invoke each static rule, until it produces no new edges.
- Edge Triggered Rules: these add new edges to the chart whenever certain kinds of edges are added by any chart rule.

A `ChartParserStrategy` consists of a list of static rules and a list of edge triggered rules.

Chart rules are defined using functions, which return a list of new edges to add to a chart. These functions should *not* directly modify the chart (e.g., do not add the edges to the chart yourself; C{ChartParser} will add them for you). Static rules are defined with functions of the form:

```
def static_rule(chart, grammar, basecat):
    # Decide which edges to add (if any)
    return edges
```

where `chart` is the chart that the static rule should act upon; `grammar` is the grammar used by the chart parser; and `basecat` is the top-level category of the grammar (e.g. 'S'). The function should return a list of edges. These edges will be added to the chart by the chart parser.

Edge triggered rules are defined with functions of the form:

```
def edge_triggered_rule(chart, grammar, basecat, edge):
    # Decide which edges to add (if any)
    return edges
```

Where the arguments are as before, and where `edge` is the edge that triggered this rule. As before, the function should return a list of edges, and these will be added to the chart by the chart parser.

The main kinds of chart rules are enumerated and discussed below:

6.3.1. Chart Initialization

The first step in parsing a tokenized sentence is to initialize the chart with word edges. For each lexical rule which expands to a given word, a corresponding edge is created. This edge hosts a complete dotted rule (e.g. `Det -> 'the' *`), a simple treetoken containing the syntactic structure, and a location equal to the location of the original token.

6.3.2. Fundamental Rule

The most important way that edges are combined is known as the *Fundamental Rule*. Suppose that an edge e_1 has a dotted rule whose next symbol is X. Suppose that a second edge e_2, immediately to the right of e_1, represents a complete X constituent. Then the Fundamental Rule states that we must add a new edge e_3 spanning both e_1 and e_2, in which the dot is moved one position to the right. In other words, e_1 was looking for an X to its right, which it found on e_2, and we record this fact on e_3.

The `Edge` class has a `FR` method which applies the fundamental rule to a pair of edges. The operation of the rule is accomplished in four lines:

```
def FR(self, edge):
```

```
    loc = self._loc.union(edge.loc())
    dr = self._drule.shift()
    tree = TreeToken(self._tree.node(), *(self._tree.children() + (edge.tree(),)
    return Edge(dr, tree, loc)
```

First we create a new location which is the union of the locations of the two existing edges. Next we create a new dotted rule, just the same as the existing one but with the dot position shifted one to the right. Next we take the existing tree `self._tree`, break it apart, and incorporate the tree from the other edge. Finally, we construct a new edge from the new dotted rule, new tree, and new location.

This rule is called many times by `chartparser.FR()`, as it considers all pairs of adjacent edges and all grammar rules.

```
def FR(chart, grammar, basecat):
    added = []
    for edge in chart.edges(): # try every edge
        if not edge.complete(): # only do incomplete ones
            for edge2 in chart.complete_edges(): # next edge complete?
                if (edge.drule().next() == edge2.drule().lhs() and
                    edge.end() == edge2.start()):
                    new_edge = edge.FR(edge2)
                    added += [new_edge]
    return added
```

6.3.3. Top Down Initialization

Top down parsing is initialized by creating a zero-width (or self-loop) edges at the leftmost position in the chart. For every grammar rule whose left-hand side is the base category of the grammar, we create the corresponding dotted rule with the dot position at the start of the right-hand side.

```
def TD_init(chart, grammar, basecat):
    added = []
    loc = chart.loc().start_loc()
    for rule in grammar:
        if rule.lhs() == basecat:
            drule = rule.drule()
            new_edge = Edge(drule, TreeToken(drule.lhs()), loc)
            added += [new_edge]
    return added
```

6.3.4. The Top Down Edge-Triggered Rule

Whenever a chart contains an incomplete edge, with an incomplete rule having *X* as the next symbol (to the right of the dot), we know that the parser is expecting to find

an *X* constituent immediately to the right. The top-down edge-triggered rule looks for all incompleted edges expecting an *X* and, for each grammar rule having *X* on its left-hand side, creates a zero-width edge containing this rule.

```
def TD_edge(chart, grammar, basecat, edge):
    "Top-down init (edge triggered rule)"
    added = []
    for rule in grammar:
        if not edge.complete() and rule.lhs() == edge.drule().next():
            new_edge = edge.self_loop_end(rule)
            added += [new_edge]
    return added
```

6.3.5. Bottom Up Initialization

A bottom up parser builds syntactic structure starting from the words in the input. For each word, it must consider which grammar rules apply. The bottom up initialization step involves inserting zero-width edges for all words which could be at the left corner of a phrase.

```
def BU_init(chart, grammar, basecat):
    added = []
    for edge in chart.edges():
        for rule in grammar:
            if edge.drule().lhs() == rule[0]:
                new_edge = edge.self_loop_start(rule)
                added += [new_edge]
    return added
```

6.3.6. Chart Parser Strategies

Chart parsers use the various rules described above in order to emulate top down, bottom up or hybrid parsers. A policy about what chart rule to apply when is called a *rule-invocation stratagy*. (Note that the rules being invoked here are chart rules, not grammar rules.)

NLTK defines basic stratagies using instances of the ChartParserStrategy class. Chart rules, whether static or edge-triggered, are defined as functions (some are listed above), and then these are passed to the `ChartParserStrategy` class. The initializer takes two arguments: a list of static rules and a list of edge-triggered rules. Here are the definitions of the top down and bottom up strategies:

```
TD_STRATEGY = ChartParserStrategy([TD_init, FR], [TD_edge])
BU_STRATEGY = ChartParserStrategy([BU_init, FR], [])
```

6.4. Creating and Using Chart Parsers

Here is a simple chart parser, which makes use of `grammar` and `lexicon` as defined earlier in this tutorial.

```
cp = ChartParser(grammar, lexicon, 'S', strategy = BU_STRATEGY)
cp.parse(WSTokenizer().tokenize(sent))
for parse in cp.parses():
    print parse.pp()
```

The result of running this parser on the sentence *I saw a man in the park with a telescope* is the following set of trees:

```
('S':
  ('NP': 'I')
  ('VP':
    ('V': 'saw')
    ('NP':
      ('Det': 'a')
      ('N': 'man')
      ('PP':
        ('P': 'in')
        ('NP': ('Det': 'the') ('N': 'park'))))
    ('PP':
      ('P': 'with')
      ('NP': ('Det': 'a') ('N': 'telescope')))))@[0w:10w]

('S':
  ('NP': 'I')
  ('VP':
    ('V': 'saw')
    ('NP': ('Det': 'a') ('N': 'man'))
    ('PP':
      ('P': 'in')
      ('NP':
        ('Det': 'the')
        ('N': 'park')
        ('PP':
          ('P': 'with')
          ('NP': ('Det': 'a') ('N': 'telescope')))))))@[0w:10w]

('S':
  ('NP': 'I')
  ('VP':
    ('V': 'saw')
    ('NP': ('Det': 'a') ('N': 'man'))
    ('PP': ('P': 'in') ('NP': ('Det': 'the') ('N': 'park')))
    ('PP':
      ('P': 'with')
      ('NP': ('Det': 'a') ('N': 'telescope')))))@[0w:10w]
```

```
('S':
  ('NP': 'I')
  ('VP':
    ('V': 'saw')
    ('NP':
      ('Det': 'a')
      ('N': 'man')
      ('PP':
        ('P': 'in')
        ('NP':
          ('Det': 'the')
          ('N': 'park')
          ('PP':
            ('P': 'with')
            ('NP': ('Det': 'a') ('N': 'telescope'))))))))@[0w:10w]
```

To see the corresponding chart, use the function `Chart.chart().draw()`.

> **Note:** Note that some edges are duplicated because they contain different trees. In order for a chart parser to have polynomial complexity the trees corresponding to an edge must be stored in a distributed way...

6.5. The Tk Interface

> **Note:** To be written.

Please see the nltk.draw.chart module.