Treebanks and PCFGs

Linguistics 165, Professor Roger Levy

6 March 2015

Some code I use just to make the handout look prettier:

```
>>> def printList(x):
...     print("\n".join([str(y) for y in x]))
...
```

1. Thus far we have been looking only at small FRAGMENTS of a natural language (English) grammar

2. These fragments had very limited COVERAGE, and minimal AMBIGUITY

3. We can see what grammars covering a larger and larger fragment of English would have to look like, by looking at samples of naturalistic English hand-annotated for syntactic structure

4. Part of the Penn English Treebank is available within `nltk`; the rest is on our instructional servers. Exploring it within `nltk`, where trees are represented with the `nltk.tree.Tree` class:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> t = treebank.parsed_sents('wsj_0015.mrg')[0]   # pull out one tree
>>> print(t) # textual visualization; for graphical visualization, use t.draw()
(S
  (NP-SBJ-1 (NNP Commonwealth) (NNP Edison) (NNP Co.))
  (VP
    (VBD was)
    (VP
      (VBN ordered)
      (NP-2 (-NONE- *-1))
      (S
        (NP-SBJ (-NONE- *-2))
        (VP
          (TO to)
          (VP
            (VB refund)
            (NP
              (QP (IN about) ($ $) (CD 250) (CD million))
              (-NONE- *U*))
            (PP-DIR-CLR
              (TO to)
              (NP
                (PRP$ its)
                (JJ current)
                (CC and)
                (JJ former)
                (NNS ratepayers)))
            (PP-PRP
              (IN for)
              (NP
                (NP (JJ illegal) (NNS rates))
                (VP
                  (VBN collected)
                  (NP (-NONE- *))
                  (PP-PRP
                    (IN for)
                    (NP
                      (NP (NN cost) (NNS overruns))
                      (PP
                        (IN on)
                        (NP
                          (DT a)
                          (JJ nuclear)
                          (NN power)
                          (NN plant))))))))))))
  (. .))
```

Using the `nltk` `Tree` class: each instance of the class is a NODE in a tree. The `label()` method returns the SYNTACTIC CATEGORY of the node. A `Tree` object functions as a list in terms of how its DAUGHTER nodes are stored. These daughters can be retrieved via ordinary Python indexing, and themselves are `Tree` objects, except for terminals, which are simply strings. For example:

```
>>> t.label() # the root node of the tree is of category S
'S'
>>> len(t) # S has three daughters
3
>>> print(t[0])
```

```
(NP-SBJ-1 (NNP Commonwealth) (NNP Edison) (NNP Co.))
>>> t[0][0]
Tree('NNP', ['Commonwealth'])
>>> len(t[0][0]) # every part of speech node has exactly one daughter
1
>>> t[0][0][0] # this is just a string
'Commonwealth'
```

In the Penn Treebank, anything following a hyphen `-` or an equals sign `=` in a tree node label is a FUNCTIONAL ANNOTATION of the tree node that is not part of the BASE CATEGORY of the node. Here's a function that returns the base category of a tree node, as a string—note that this will work either as

```
>>> import re
>>> def baseCategory(t):
...     if isinstance(t,nltk.tree.Tree):
...         m = re.match("^(-[^-]+-|[^-=]+)",t.label())
...         if m == None:
...             print(t.label())
...         return(m.group(1))
...     else:
...         return(t)
...
```

Task: write a function that takes a `Tree` node and returns the context-free grammar rule such that the base category of node is the left-hand side of the rule and the base categories of the daughters constitute the right-hand side of the rule.

The `nltk.tree.Tree.subtrees()` method gives a list of all the non-terminal subtrees. **Task:** use this new function to build a frequency distribution (you can use `nltk.probability.FreqDist`) over all rule productions in the part of the Penn Treebank provided in `nltk`. **Hint:** if `x` is a list of lists, then `list(itertools.chain(*x))` is the concatenation of all the lists in `x`.

5. Let's look at some of the rare rules (assume that your `FreqDist` object is called `fd`):

```
>>> printList([x for x in fd.items() if x[1]<3][0:20])
('VBZ -> teaches', 1)
('CD -> 319.75', 1)
('VBD -> dominated', 1)
('VBG -> enabling', 2)
('VBN -> overcome', 1)
('VBP -> assume', 2)
('S -> CC ADVP NP ADVP VP .', 1)
('VB -> suggest', 1)
```

```
('NNP -> Henderson', 1)
('NNS -> manipulators', 1)
('NN -> taste', 1)
('-NONE- -> *T*-92', 2)
('NNP -> Terrace', 1)
('JJ -> government-certified', 1)
('VP -> VB PRT NP PP S', 1)
('NP -> DT NNP CD NNS', 2)
('VB -> launch', 2)
('VBZ -> resists', 1)
('VBN -> tracked', 1)
('NP -> DT CD CD NNS', 1)
```

There's all sorts of surprising stuff in here! The context-free grammar we get straight off the Penn Treebank is *huge and complex.*

6. **Question:** with all sorts of surprising bits of rule structure available, what kinds of unexpected structures are possible for innocuous-looking sentences? Example:

   (1) the old harbor burns

   - What should its structure be?
   - What kinds of other parses might be possible?

7. **Tree search:** because `nltk` only has part of the Penn Treebank, and because we may want to probe all sorts of interesting structure, it is convenient to use TREE SEARCH software to explore it. The `Tregex` software package we looked at last week can help us with that. On the instructional servers I have installed it in the directory

   ```
   /home/linux/ieng6/ln165w/public/stanford-tregex
   ```

   If you log on to the servers and run

   ```
   cd /home/linux/ieng6/ln165w/public/stanford-tregex
   ```

   you can then run

   ```
   ./stanford-tregex.sh
   ```

   to call the program. The entire Wall Street Journal section of the Penn Treebank is linked to in the file

   ```
   /home/linux/ieng6/ln165w/public/stanford-tregex/wsj-all.mrg
   ```

8. **A fragment of the search syntax for Tregex:**

`A < B`	A immediately dominates B
`A > B`	A is immediately dominated by B
`A << B`	A dominates B
`A <i B`	A's *i*-th daughter (left to right, starting at 1) is B
`@A`	matches node X if the BASE CATEGORY of X is `A`
`A ! op B`	it is NOT the case that `op B` holds of A
`__`	matches ANY node

Some of the command-line options for Tregex: (compare with that of `egrep`)

`-C`	Return the number of matches, not the content of the matches
`-u`	Print only the label of the matching node, not the rest of the tree
`-w`	Print the entire tree in which the match was found
`-s`	Print the match on a single line, instead of pretty-printing it
`-t`	Print only the YIELD (leaves) of the match, not the tree part
`-v`	Print every tree in which no match was found, and don't print the trees in which matches are found

There's lots more to read about (e.g., at the TregexPattern javadoc page), but this will get you started. For example, try

```
./stanford-tregex.sh '@NP <1 DT <2 JJ ! <3 __' wsj-all.mrg
```

to find instances of the rule NP $\rightarrow$ DT JJ.

9. **Task:** use `Tregex` to determine whether the Penn Treebank contains evidence for context-free rules that would license the weird structure I proposed for Sentence (1).

10. **Motivation for probabilistic context-free grammars.** Although the weird structure is just as *possible* as the first structure, there is clearly a sense in which the weird structure is *dispreferred* (by use, the native English speakers) in comparison with the first structure. There are many reasons for this dispreference. **Task:** list the reasons you can think of for this dispreference. Some of these reasons it is not realistic for us to expect computers to be able to learn given contemporary methods, but other reasons are readily easily recoverable from the content of the Treebank. These reasons generally can be stated in the form: "in the face of ambiguity, prefer that which better explains the data." This is the same approach we used to document classification, and we can bring it into syntactic parsing by adding probabilities to our context-free grammars.

11. **Definition of a probabilistic context-free grammar.** A PROBABILISTIC CONTEXT-FREE GRAMMAR (PCFG) is exactly like a context-free grammar, but it has a *probability* component associated with it. The definition: a PCFG consists of:

    - A set N of NON-TERMINAL symbols;
    - A set Σ of TERMINAL symbols;

- A START SYMBOL $\mathsf{S} \in N$;
- A set R of RULES, such that each rule $r \in R$ has the form $X \to \alpha$, where $X \in N$ and $\alpha \in (N \cup \Sigma)^*$—that is, the LEFT-HAND SIDE of r is a single symbol in N, and the RIGHT-HAND SIDE of r is a (possibly empty) sequence of symbols drawn from N and/or Σ (the * is the Kleene star that we saw in regular expressions)
- A PROBABILITY FUNCTION P that maps each rule $r \in R$ to a probability ranging from $[0, 1]$, such that for every non-terminal $X \in N$,

  $$\sum_{\alpha:\ (X\to\alpha)\in R} P(X \to \alpha) = 1$$

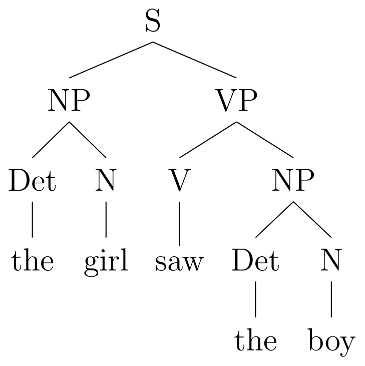
  That is, for every non-terminal X in the grammar, the probabilities of the rules REWRITING X sum to 1. You can think of $P(X \to \alpha)$ as indicating the CONDITIONAL PROBABILITY of daughters α appearing in the tree, given that X is the mother.

Example PCFG:

S	→ NP VP	1	Det	→ the	1.0
NP	→ Det N	0.7	N	→ girl	0.4
NP	→ NP PP	0.3	N	→ boy	0.4
VP	→ V NP	0.6	N	→ telescope	0.2
VP	→ V NP PP	0.4	V	→ saw	1.0
PP	→ P NP	1	P	→ with	1.0

12. **Probability of a tree in a PCFG.** With context-free grammars we linked DERIVATIONS (sequences of rule applications) to the TREES they derived. With probabilistic context-free grammars, we can score derivations (and thus trees) with their probabilities. The probability of a tree is the product of the probabilities of the rules used to derive the tree. If we enumerate the derivation steps in a tree as the rule probabilities, we get the probability of the tree in the grammar.

    Consider the tree



The rule sequence (top-down, left to right) and the associated product of probabilities in the grammar from above:

S	→	NP VP		1
NP	→	Det N	×	0.7
Det	→	the	×	1
N	→	girl	×	0.4
VP	→	V NP	×	0.6
V	→	saw	×	1
Det	→	the	×	1
N	→	boy	×	0.4
			=	0.0672

13. **Disambiguating among parses with a PCFG.** As with text classification, we can use Bayesian inference to disambiguate among CFG parses with PCFG probabilities. If s is our sentence and t ranges over possible trees, the definition of joint and conditional probabilities allows us to say

$$P(t|s) = \frac{P(s|t)P(t)}{P(s)}$$

As is typically the case in Bayesian inference, in parsing the denominator of the right-hand side is fixed because we have our sentence and we don't consider alternative sentences in the process of parsing a given sentence. The likelihood term $P(s|t)$ is 1 for trees whose yield is s—that is, for actual parses of s—and 0 for all other trees. This means that we can rank parses of s according to their probability $P(t)$ given to us by our PCFG.

Example: consider the sentence

the girl saw the boy with the telescope

What are the two possible parses in our PCFG? What is $P(t)$ for each parse? Which parse is preferred?

14. **Estimating PCFG probabilities from a treebank.** The best way to estimate PCFG probabilities from a treebank is a major open problem in computational linguistics, and there's a huge amount of work on it. The simplest, or VANILLA estimation procedure, is using relative frequency estimation:

$$\widehat{P}_{RFE}(X \to \alpha) = \frac{\text{Count}(X \to \alpha)}{\text{Count}(X)}$$

Example: we can use the two following calls to `Tregex` to estimate the probability of the rule NP → DT JJ from the Penn Treebank:

```
./stanford-tregex.sh -c '@NP <1 DT <2 JJ ! <3 __' wsj-all.mrg
./stanford-tregex.sh -c '@NP' wsj-all.mrg
```

Task: Use `Tregex` to estimate the ratio of probabilities of the two trees corresponding to the correct and alternative, weird parses of Sentence (1). Does a vanilla PCFG from the Penn Treebank select the correct parse?