COMP 150-06 Natural Language Processing
Spring 2016

Problem Set 4: Statistical parsing

Build a PCFG parser using the template `pset4_template.py` and NLTK utilities for representing trees and learning PCFGs.

Your training and test set has been heavily preprocessed, by

1. skipping any tree whose root is not the nonterminal S,
2. removing all functional tags attached to nonterminals (e.g., NP-SBJ replaced with NP),
3. removing trace nodes (these are the preterminal nodes that are labeled with -NONE-),
4. collapsing unary productions using the + symbol (e.g., the subtree `(NP (NN book))` would become `(NP+NN book)`),
5. transforming the trees into Chomsky normal form (CNF) so that all non-preterminal branches are binarized.

We provide the functions that implement these steps as examples for how to traverse and manipulate NLTK's tree objects. NLTK provides a lot of helpful documentation, however, just a couple of life pro tips:

- NLTK trees can be constructed by hand like this:

```
>>> tree = Tree("VP", [Tree("VB", ["Book"]), Tree("NP", [Tree("DT", ["that"]),
Tree("NN", ["flight"])])])
>>> print tree
(VP (VB Book) (NP (DT that) (NN flight)))
```

- A tree is like a list whose elements are its children (its length is its number of children).

```
>>> for child in tree: print child
...
(VB Book)
(NP (DT that) (NN flight))
```

- Tree nodes have heights, leaf nodes are at height=1, preterminal nodes are at height=2, etc.

```
>>> print tree.height(), tree[0].height(), tree[1].height()
4 2 3
```

- Trees nodes have labels:

```
>>> print tree.label(), tree[0].label(), tree[1].label()
VP VB NP
```

- Extracting the sentence of a tree:

```
>>> print tree.leaves()
['Book', 'that', 'flight']
```

- Trees have a helpful function called `subtrees` that give you all tree nodes:

```
>>> for subtree in tree.subtrees():
...     print subtree
...
(VP (VB Book) (NP (DT that) (NN flight)))
(VB Book)
(NP (DT that) (NN flight))
(DT that)
(NN flight)
```

- And another helpful function called `productions` that gives you all the productions (see NLTK's grammar module for the definition of the `Production` object):

```
>>> for production in tree.productions():
...     print production
...
VP -> VB NP
VB -> 'Book'
NP -> DT NN
DT -> 'that'
NN -> 'flight'
```

Problems

1. **Preprocessing:** As usual, build a static vocabulary from your training set, treating every word that occurs not more than once as an unknown token. Transform both of your training and test sets by introducing unknown tokens at the appropriate preterminal nodes (the nodes that are sitting immediately above a leaf node). We provide a function `PreterminalNodes(tree)` that yields all such nodes, which can be used to obtain word counts for vocabulary building, as well as for replacing the appropriate words with unknown tokens. For example:

```
for node in PreterminalNodes(tree)
    if node[0] not in vocabulary:
        node[0] = unknown_token
```

   After applying your transformation, print the first trees of both data sets in your write-up.
2. **Training:** Using your preprocessed training set, learn a PCFG using the function `induce_pcfg(start, productions)` provided to you by NLTK's grammar module. Use `Nonterminal("S")` as your start symbol. You can obtain all the productions that are in your training set using the `productions` method of `nltk.tree.Tree`.
   2.1. How many productions are there for the NP nonterminal?
   2.2. Print the most probable 10 productions for the NP nonterminal in your write-up.
3. **Testing:** Implement the probabilistic CKY algorithm for parsing a test sentence using your learned PCFG.
   3.1. Implement the `BuildIndex` method of `InvertedGrammar` class: This will build an inverted index of your grammar that maps *right hand sides of all productions to their left hands sides*. You will need this to avoid looping over the rules of the PCFG grammar. Since you grammar is going to be quite large, you cannot loop over it efficiently for every span of the sentence. Instead, as we fill our dynamic programming table cell for the span (i, j) and a split-point k, we will consider a rule **A** → **B C** if and only if we have a non-zero probability of **B** covering the span (i, k), and a non-zero probability of **C** covering the span (k, j). Using the PrintIndex function provided, print this index to a file named "index" and submit it.
   3.2. Implement the `Parse` method of `InvertedGrammar`: This will implement the CKY algorithm for PCFGs, populating the dynamic programming table with *log probabilities* of every constituent spanning a sub-span of a given test sentence (i, j) and storing the appropriate back-pointers. What is the log probability of the

nonterminal S for the 5-token sentence "*Terms were n't disclosed* ."? Report in your write-up.

3.3. Implement the `BuildTree` method of `InvertedGrammar`: This will build tree objects by following the back-pointers starting from the largest span `(0, len(sent))` and recursing from larger spans (i, j) to smaller sub-spans (i, k), (k, j) and eventually bottoming out at the preterminal level (i, i+1). Return `None` If the nonterminal S is not found in the cell corresponding to the span `(0, len(sent))` before even starting the recursion, signalling a parsing error. What is the parse tree for the 5-token sentence "*Terms were n't disclosed* ."? Report in your write-up.

3.4. Bucket your test set sentences into the following 5 buckets

```
Bucket1: 0 < len(sent) < 10
Bucket2: 10 <= len(sent) < 20
Bucket3: 20 <= len(sent) < 30
Bucket4: 30 <= len(sent) < 40
Bucket5: 40 <= len(sent)
```

How many sentences fall in each bucket? Report in your write-up.

3.5. For each bucket, unbinarize and print your predicted trees and gold standard trees, one tree per line, into separate files (e.g., test_1, gold_1, test_2, gold_2, etc). Unbinarize your trees using the `tree.un_chomsky_normal_form()` method. Use the function `PrintTree` that we provide for printing one tree per line. Print empty lines for parsing errors (when `BuildTree` returns `None`). Evaluate your parsing performance using EVALB and `COLLINS.prm` parameter file using a command such as `evalb -p COLLINS.prm gold_1 test_1` (See the README for how to make and run `evalb`). **We strongly recommend doing developmental testing on Bucket1, and moving on to the other buckets only as your parser is accurate and efficient enough.** Report your *Bracketing FMeasure* and *Average crossing* for each bucket and the same for the overall combined test set in your write-up. Provide your `test_*` and `gold_*` files.

4. **Extra credit:** Improve your parsing performance by adding parent annotation as we discussed in class.

Deliver the following by Wednesday Night 4/21/2016 02:59 AM.

1. A PDF write-up. All questions 1-4 attempted should have at least a paragraph describing your reasoning and final solution, explaining your decisions in detail. The write-up should include code blurbs if necessary.
2. Your code. This is a .py file that compiles and runs. It should demo the answers that you produced in your work.
3. Any supplementary files you create index, test_*, gold_*, etc.

How to submit your homework:

1. Log onto the Tufts server:

```
ssh your_username@homework.cs.tufts.edu
```

2. Navigate to the directory of your submission files and type the following command to submit all of your files together:
```
provide comp150nlp pset4 [file1 file2 ...]
```