Natural Language Processing HW5: Syntax and Parsing

Jesse Weller, Sheekha Jariwala, Prachi Rahurkar

In this assignment you will

1. train a probabilistic context-free grammar (PCFG) from a treebank (need binarization);
2. build a simple CKY parser (which handles unary rules), and report its parsing accuracy on test data.

Download `http://classes.engr.oregonstate.edu/eecs/fall2019/cs539-001/hw5/hw5-data.tgz`:

corpora:

`train.trees`	training data, one tree per line
`test.txt`	test data, one sentence per line
`test.trees`	gold-standard trees for the test data (for evaluation)

scripts:

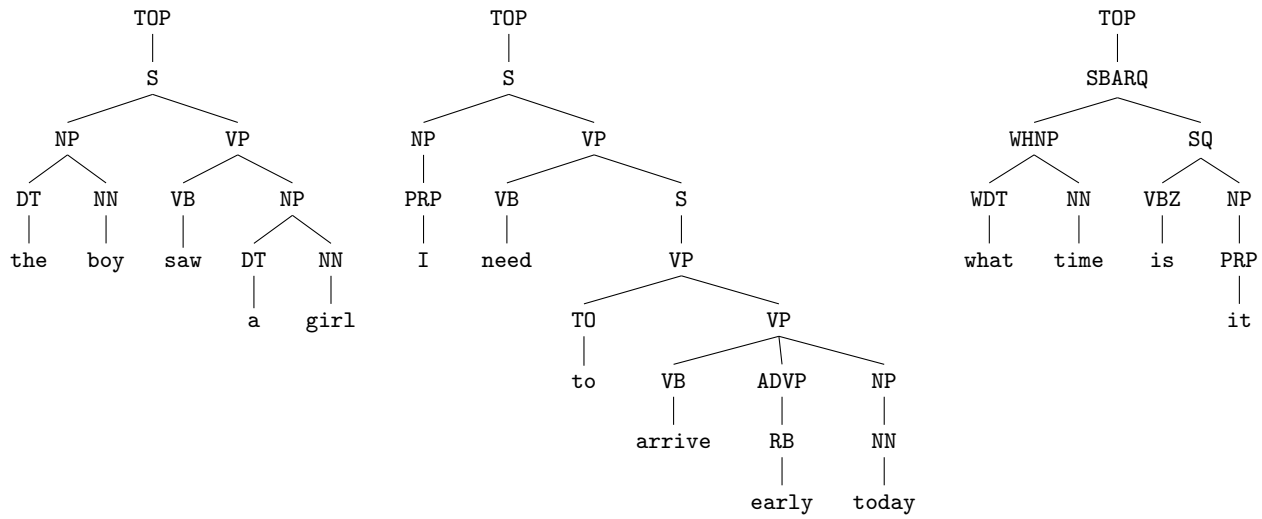
`tree.py`	`Tree` class and reading tool
`evalb.py`	evaluation script

1 Learning a grammar (50 pts)

The file `train.trees` contains a sequence of trees, one per line, each in the following format:

```
(TOP (S (NP (DT the) (NN boy)) (VP (VB saw) (NP (DT a) (NN girl)))))
(TOP (S (NP (PRP I)) (VP (VB need) (S (VP (TO to) (VP (VB arrive) (ADVP (RB early)) (NP (NN today))))))))
(TOP (SBARQ (WHNP (WDT what) (NN time)) (SQ (VBZ is) (NP (PRP it)))))
```

N.B. `tree.py` provides tools for you to read in these trees from file (building a `Tree` object from a string).



Note that these examples are interesting because they demonstrate

- unary rules on preterminals (POS tags) like `NP -> PRP` and `NP -> NN`;

- unary rules on nonterminals (constituents) like `S -> VP` and `TOP -> S`;
- ternary rules like `VP -> VB ADVP NP` (so that you need binarization).
- `NP` can have (at least) three roles: subject (`the boy`), object (`a girl`), and temporal-adverbial (`today`). The Penn Treebank does annotate these roles (e.g., `NP-SBJ`, `NP-TMP`), but for simplicity reasons we removed them for you.
- a sentence can be a declarative sentence (`S`) or a wh-question (`SBARQ`), among others.

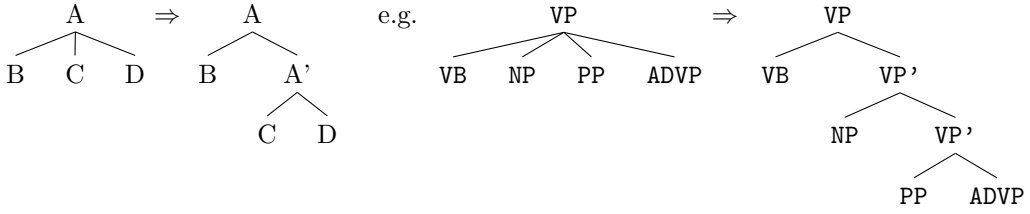
Your job is to learn a probabilistic context-free grammar (PCFG) from these trees.

Q1.a Preprocessing: replace all one-count words (those occurring only once, *case-sensitive*) with `<unk>`:

```
cat train.trees | python replace_onecounts.py > train.trees.unk 2> train.dict
```

where `train.trees.unk` is the resulting trees with `<unk>` symbols, and `train.dict` is a file of words which appear more than once (one word per line). **Explain why we do this.**
Replacing one-count words during training is a good strategy to categorize all the rare and unseen (during training) words as <UNK>and assign a minimum probability to those words whenever encountered.

Q1.b Binarization: we binarize a non-branching tree (recursively) in the following way:




```
cat train.trees.unk | python binarize.py > train.trees.unk.bin
```

What are the properties of this binarization scheme?
The binarization scheme described above is the *right binarization* scheme as it always combines the two *right-most* pairs of symbols to form a non-terminal.

Why we would use such a scheme? (What are the alternatives, and why we don't use them?)
Any binarization scheme ensures that the following CKY algorithm used to build the parse tree would run with time-complexity of O(n^3).
Other binarization schemes are *left-binarization head-binarization* and *compact binarization* schemes. Essentially all the binarization schemes give the same parsing accuracy but different parsing efficiency.
If left-binarization is used then an extra constituent will be produced. Using right-binarization would avoid creation of such an extra constituent thus resulting in better parse-efficiency.

Is this binarized CFG in Chomsky Normal Form (CNF)?
Chomsky Normal Form is one in which all rules of grammar are either terminal rules or binary rules. That is exactly what binarization scheme described above results in. So yes, this binarized CFG is in Chomsky Normal Form.

Q1.c Learn a PCFG from trees:

```
cat train.trees.unk.bin | python learn_pcfg.py > grammar.pcfg.bin
```

The output `grammar.pcfg.bin` should have the following format:

```
TOP
TOP -> S # 1.0000
S -> NP VP # 0.8123
S -> VP # 0.1404
VP -> VB NP # 0.3769
VB -> saw # 0.2517
...
```

where the first line (`TOP`) denotes the start symbol of the grammar, and the number after `#` in each rule is its probability, with four digits accuracy.

We group rules into three categories: binary rules (`A -> B C`), unary rules (`A -> B`), and lexical rules (`A -> w`). How many rules are there in each group?
Number of binary rules - 179
Number of unary rules - 48
Number of lexical rules - 287

2 CKY Parser (70 pts)

Now write a CKY parser that takes a PCFG and a test file as input, and outputs, for each sentence in the test file, the best parse tree in the grammar. For example, if the input file is

```
the boy saw a girl
I need to arrive early today
```

the output (according to some PCFG) could be:

```
(TOP (S (NP (DT the) (NN boy)) (VP (VB saw) (NP (DT a) (NN girl)))))
(TOP (S (NP (PRP I)) (VP (VB need) (S (VP (TO to) (VP (VB arrive) (ADVP (RB early)) (NP (NN today))))))))
```

Q2.a Design a toy grammar `toy.pcfg` and its binarized version `toy.pcfg.bin` such that the above two trees are indeed the best parses for the two input sentences, respectively. **How many strings do these two grammars generate? Just 2.**

Q2.b Write a CKY parser. Your code should have the following input-output protocol:

```
cat toy.txt | python cky.py toy.pcfg.bin > toy.parsed
```

Verify that you get the desired output in `toy.parsed`. **Note that your output trees should be debinarized (see examples above).**

Q2.c Now that you passed the toy testcase, apply your CKY parser to the real test set:

```
cat test.txt | python cky.py grammar.pcfg.bin  > test.parsed
```

Your program should handle (any levels of) **unary rules** correctly, even if there are unary cycles.

How many sentences failed to parse? Your CKY code should simply output a line

```
NONE
```

for these cases (so that the number of lines in `test.parsed` should be equal to that of `test.txt`). What are main reasons of parsing failures?
Number of sentences that fail to parse - 17
The main reason for parsing failures is that they contain one count words or no count words from the training data. We need to convert them to ¡unk¿ for successful parsing.

Q2.d Now modify your parser so that it first replaces words that appear once or less in the training data:

```
cat test.txt | python cky.py grammar.pcfg.bin train.dict > test.parsed.new
```

Note that:

- `train.dict` should be treated as an **optional** argument: your CKY code should work in either case! (and please submit only **one version** of `cky.py`).
- your output tree, however, should use the original words rather than the `<unk>` symbol. Like binarization, it should be treated as internal representation only.

Now how many sentences fail to parse? **Number of sentences that fail to parse - 0**
After converting one and zero counts to ¡unk¿, they all parse.

Q2.e Evaluate your parsing accuracy by

```
python evalb.py test.parsed test.trees

test.parsed     244 brackets
test.trees     385 brackets
matching        224 brackets

Precision: 91.8%
Recall: 58.2%
F1: 71.2%

python evalb.py test.parsed.new test.trees

test.parsed.new 372 brackets
test.trees     385 brackets
matching        312 brackets

Precision: 83.87%
Recall: 81.0%
F1: 82.41%
```

Report the labeled precisions, recalls and F-1 scores of the two parsing results. Do their differences make sense?