Natural Language Processing, Problem Set 2

In this programming assignment, you will train a probabilistic context-free grammar (PCFG) for syntactic parsing. In class we defined a PCFG as a tuple

$$G = (N, \Sigma, S, R, q)$$

where N is the set of non-terminals, Σ is the vocabulary, S is a root symbol, R is a set of rules, and q is the rule parameters. We will assume that the grammar is in Chomsky normal form (CNF). Recall from class that this means all rules in R will be either *binary rules* $X \rightarrow Y_1\ Y_2$ where $X, Y_1, Y_2 \in N$ or *unary rules* $X \rightarrow Z$ where $X \in N$ and $Z \in \Sigma$.

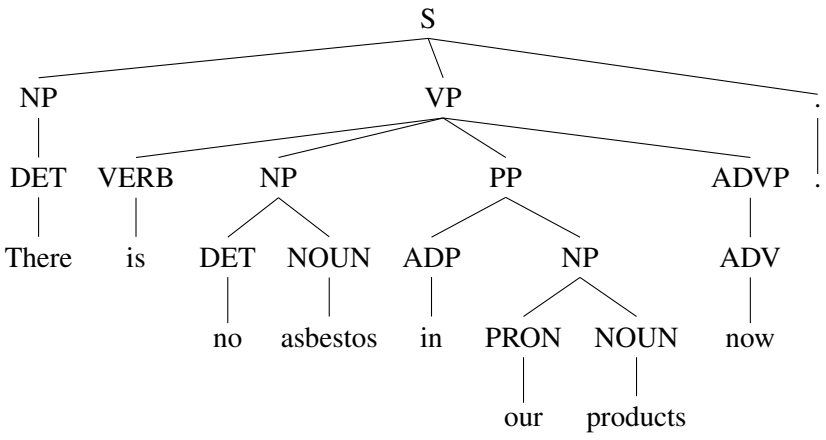
You will estimate the parameters of the grammar from a corpus of annotated questions from the QuestionBank [Judge et al., 2006]. Before diving into the details of the corpus format, we will discuss the rule structure of the grammar. You will not have to implement this section but it will be important to know when building your parser.

Rule Structure

Consider an example parse tree from the Wall Street Journal corpus (on which QuestionBank is modeled)

```
There/DET is/VERB no/DET asbestos/NOUN in/ADP our/PRON products/NOUN now/ADV ./.
```

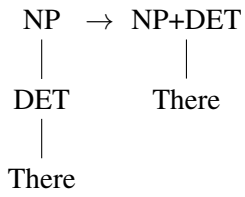
The correct parse tree for the sentence (as annotated by linguists) is as follows



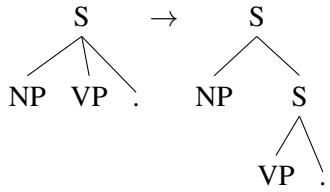
Note that nodes at the fringe of the tree are words, the nodes one level up are part-of-speech tags, and the internal nodes are syntactic tags. For the definition of the syntactic tags used in this corpus, see the work of Taylor et al. [2003]. The part-of-speech tags should be self-explanatory, for further reference see the work of Petrov et al. [2011].

Unfortunately the annotated parse tree is not in Chomsky normal form. For instance, the rule S $\rightarrow$ NP VP . has three children and the rule NP $\rightarrow$ DET has a single non-terminal child. We will need to work around this problem by applying a transfomation to the grammar.

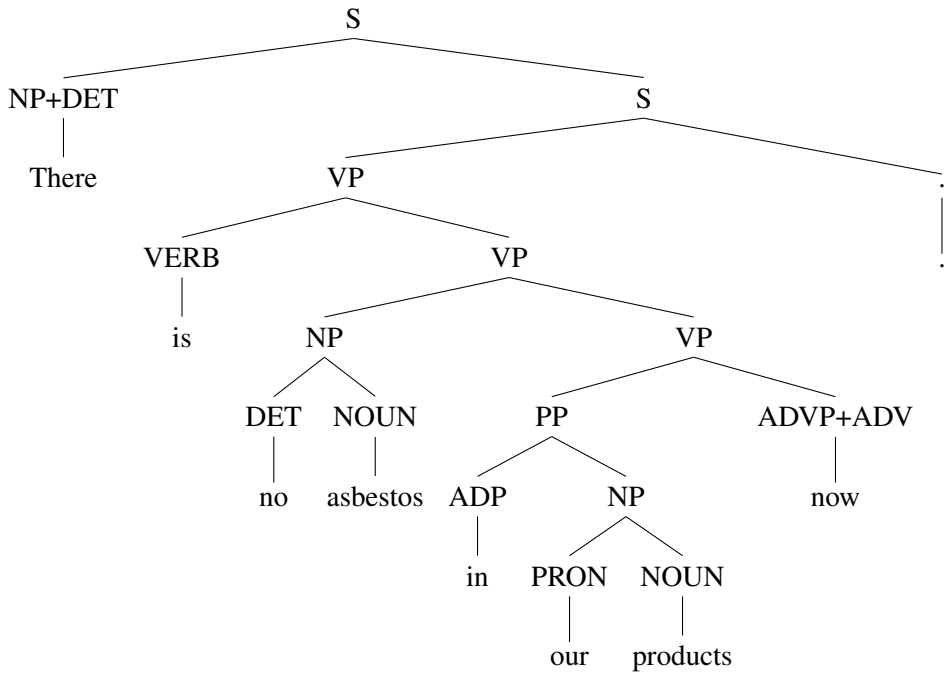
We will apply two transformations to the tree. The first is to collapse illegal unary rules of the form $X \rightarrow Y$ where $X, Y \in N$ into new non-terminals $X + Y$, for example



The second is to split n-ary rules $X \rightarrow Y_1\ Y_2\ Y_3$ where $X, Y_1, Y_2, Y_3 \in N$ into right branching binary rules of the form $X \rightarrow Y_1 X$ and $X \rightarrow Y_2 Y_3$ for example



With these transformation to the grammar, the correct parse tree for this sentence is



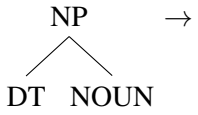
Converting the grammar to CNF will greatly simplify the algorithm for decoding. All the parses we provide for the assignment will be in the transformed format, you will not need to implement this transformation, but you should understand how it works.

Corpus Format

We provide a training data set with parses `parse_train.dat`, a development set of sentences `parse_dev.dat` along with the correct parses `parse_dev.key`, and a test set of sentences `parse_test.dat`. All the data comes for the QuestionBank corpus.

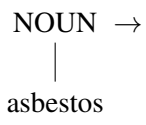
In the training set, each line of the file consists of a single parse tree in CNF. The trees are represented as nested multi-dimensional arrays encoded JSON. JSON is a general data encoding standard similar in spirit to XML. We use JSON because it is well-supported in pretty much every programming language (see `http://www.json.org/`).

Binary rules are represented as arrays of length 3.




```
["NP", ["DT", ...], ["NOUN", ...]]
```

Unary rules are represented as arrays of length 2.



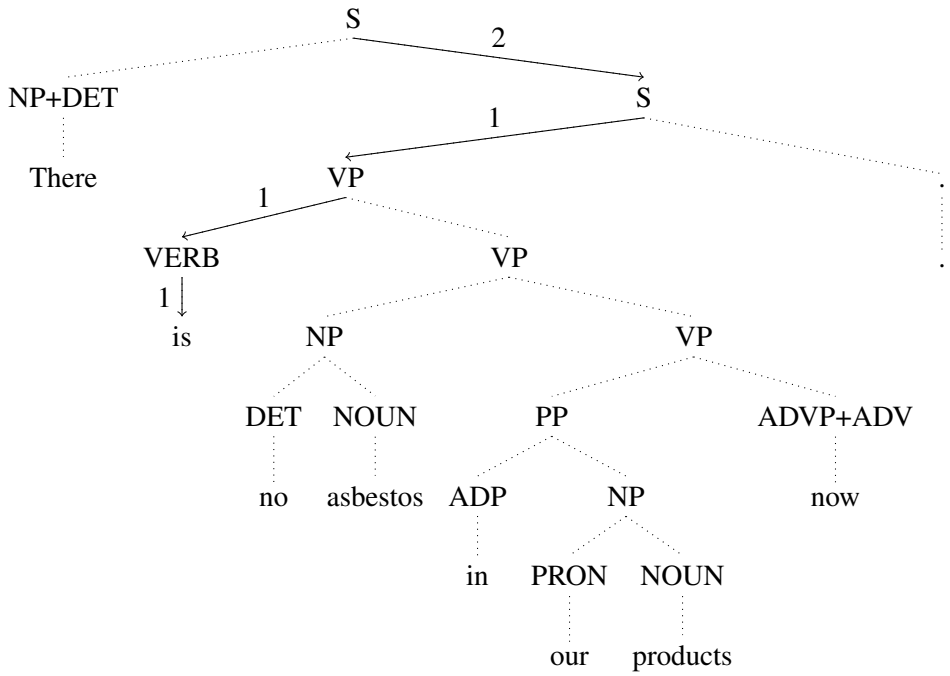

```
["NOUN", "asbestos"]
```

For example, the file `tree.example` has the tree given above

```
["S", ["NP", ["DET", "There"]], ["S", ["VP", ["VERB", "is"], ["VP", ["
   NP", ["DET", "no"], ["NOUN", "asbestos"]], ["VP", ["PP", ["ADP", "in
   "], ["NP", ["PRON", "our"], ["NOUN", "products"]]], ["ADVP", ["ADV",
    "now"]]]]], [".", "."]]]
```

The tree is represented as a recursive, multi-dimensional JSON array. If the array is length 3 it is a binary rule, if it is length 2 it is a unary rule as shown above.

As an example, assume we want to access the node for "is" by walking down the tree as in the following diagram.



In Python, we can read and access this node using the following code

```
import json
tree = json.loads(open("tree.example").readline())
print tree[2][1][1][1]
```

The second line reads the tree into an array, and the third line walks down the tree to the node for "is".

Other languages have similar libraries for reading in these arrays. Consult the forum to find a similar parser and sample code for your favorite language.

Tree Counts

In order to estimate the parameters of the model you will need the counts of the rules used in the corpus. The script `count_cfg_freq.py` reads in a training file and produces these counts. Run the script on the training data and pipe the output into some file:

```
python count_cfg_freq.py parse_train.dat > cfg.counts
```

Each line in the output contains the count for one event. There are three types of counts:

- Lines where the second token is `NONTERMINAL` contain counts of non-terminals $Count(X)$

  ```
  17 NONTERMINAL NP
  ```

  indicates that the non-terminal NP was used 17 times.

- Lines where the second token is `BINARYRULE` contain emission counts $Count(X \rightarrow Y_1\ Y_2)$, for example

```
918 BINARYRULE NP DET NOUN
```

indicates that binary rule NP → DET NOUN was used 918 times in the training data.

- Lines where the second token is `UNARYRULE` contain unigram counts $Count(X \rightarrow Y)$.

```
8 UNARYRULE NP+NOUN place
```

indicates that the rule NP+NOUN → place was seen 8 times in the training data.

Part 1 (20 points)

- As in the tagging model, we need to predict emission probabilities for words in the test data that do not occur in the training data. Recall our approach is to map infrequent words in the training data to a common class and to treat unseen words as members of this class. Replace infrequent words ($Count(x) < 5$) in the original training data file with a common symbol `_RARE_`. Note that this is more difficult than the previous assignment because you will need to read the training file, find the words at the fringe of the tree, and rewrite the tree out in the correct format. We provide a script `python pretty_print_tree.py parse_dev.key` which you can use to view your trees in a more readable format. The script will also throw an error if the output is not in the correct format. Re-run `count_cfg_freq.py` to produce new counts.

  Write the new counts out to a file `parse_train.counts.out` and submit with `submit.py`.

Part 2 (40 points)

- Using the counts produced by `count_cfg_freq.py`, write a function that computes rule parameters

$$q(X \rightarrow Y_1\ Y_2) = \frac{Count(X \rightarrow Y_1\ Y_2)}{Count(X)}$$

$$q(X \rightarrow w) = \frac{Count(X \rightarrow w)}{Count(X)}$$

- Using the maximum-likelihood estimates for the rule parameters, implement the CKY algorithm to compute

$$\arg \max_{t \in \mathcal{T}_G(S)} p(t).$$

Note: Since we are parsing questions, the root symbol $S =$ `SBARQ`. This means you should return the best parse with `SBARQ` as the non-terminal spanning $[1, n]$.

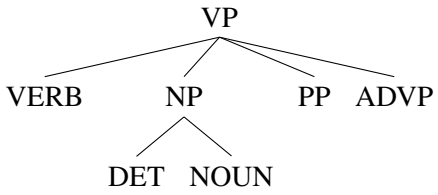
Your parser should read in the counts file (with rare words) and the file `parse_dev.dat` (which is just the sentence from `parse_dev.key` without the parse) and produce output in the following format: the tree for one sentence per line represented in the JSON tree format specified above. Each line should correspond to a parse for a sentence from `parse_dev.dat`.

View your output using `pretty_print_parse.py` and check your performance with `eval_parser.py` by running `python eval_parser.py parse_dev.key parse_dev.out`. The evaluator takes two files where each line is a matching parse in JSON format and outputs an evaluation description.

The expected development F1-Score is posted on the assignment page. When you are satisfied with development performance, run your decoder on `parse_test.dat` to produce `parse_test.p2.out`. Submit your output with `submit.py`.

Optional Part 3 (1 point)

Consider the following subtree of the original parse.



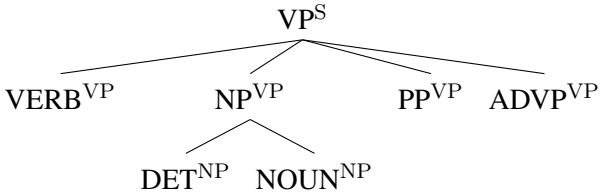
The tree structure makes the implicit independence assumption that the rule NP $\rightarrow$ DT NOUN is generated independently of the parent non-terminal VP, i.e. the probability of generating this subtree in context is

$$p(\text{VP} \rightarrow \text{VERB NP PP ADVP} \mid \text{VP}) \times p(\text{NP} \rightarrow \text{DET NOUN} \mid \text{NP})$$

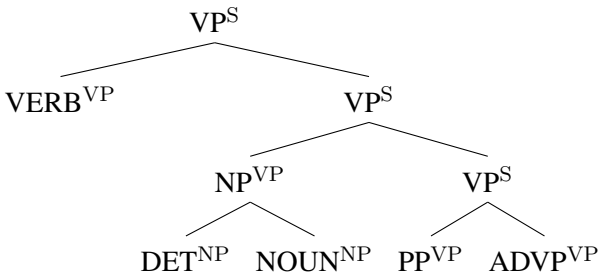
However, often this independence assumption is too strong. It can be informative for the lower rule to condition on the parent non-terminal as well, i.e. model generating this subtree as

$$p(\text{VP} \rightarrow \text{VERB NP PP ADVP} \mid \text{VP}) \times p(\text{NP} \rightarrow \text{DET NOUN} \mid \text{NP, parent=VP})$$

This is a similar idea to using a trigram language model, except with parent non-terminals instead of words. We can implement this model without changing our parsing algorithm by applying a tree transformation known as *vertical markovization*. We simply augment each non-terminal with the symbol of its parent.



We apply vertical markovization before binarization; the new non-terminals are augmented with their original parent. After applying both transformations the final tree will look like



There is a downside of vertical markovization. The transformation greatly increases the number of rules in the grammar potentially causing data sparsity issues when estimating the probability model q. In practice

though, using one level of vertical markovization (conditioning on the parent), has been shown to increase accuracy.

- The file `parse_train_vert.dat` contains the original training sentence with vertical markovization applied to the parse trees. Train with the new file and reparse the development set. Run `python eval_parser.py parse_dev.key parse_dev.out` to evaluate performance. The expected development F1-Score is posted on the assignment page.

  Note: This problem looks straightforward, but the difficulty is that we now have a much larger set of non-terminals N. It is possible that a simple implementation of CKY which worked for the last question will be too slow for this problem. If you find that your code is too slow, you will need to add optimizations to speed things up, while still finding the best parse tree. Specifically, think about how to avoid processing elements in π that already have zero probability and therefore cannot possibly lead to the highest-scoring parse tree.

  The challenge is to improve the basic inefficiencies of the implementation in order to handle the extra non-terminals.

  When you are satisfied with development performance, run your decoder on `parse_test.dat` to produce `parse_test.p3.out`. Submit your output with `submit.py`.

References

John Judge, Aoife Cahill, and Josef Van Genabith. Questionbank: Creating a corpus of parse-annotated questions. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 497–504. Association for Computational Linguistics, 2006.

Slav Petrov, Dipanjan Das, and Ryan McDonald. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086*, 2011.

Ann Taylor, Mitchell Marcus, and Beatrice Santorini. The penn treebank: An overview. *Treebanks*, pages 1–22, 2003.