CSCI-GA 2590: Natural Language Processing

Constituent Parsing

Name
NYU ID

Collaborators:
By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

In this homework, you will build an English parser with extended Cocke–Younger–Kasami (CKY) algorithm for given probabilistic context-free grammars (PCFGs). You just need to submit your code. No written part is required for this homework.

1 Data

There are three types of files under `data`.

1. **Grammar file (`*.gr`).** Each line is a grammar rule with three tab-separated columns: rule probability, left hand side (LHS), and right hand side (RHS). Terminal/non-terminal symbols on RHS are separated by white spaces. There are at least one symbols on RHS. Symbols are case sensitive, e.g. `DT` $\rightarrow$ `the` and `DT` $\rightarrow$ `The` are different rules.
2. **Sentence file (`*.sen`).** Each line is a sentence to be parsed and Symbols are separated by white spaces.
3. **Sample parse file (`*.par`).** For each sentence in `*.sen`, your parser should print the best parse tree followed by its log probability (use natural log). If the sentence is invalid (i.e. no valid parse under the given grammar), print `NONE`.

There are two sets of files in `data`.

1. **Toy data (`arith.*`).** This is a toy grammar that generates arithmetic experssions. You should develop the algorithm on this toy data first.
2. **Wallstreet data (`wallstreet.*`).** This is a subset of the WSJ treebank. The longer sentences may take significantly more time to parse.

2 Submission and Evaluation

For now, implement the function in `submission.py` and test using `eval.py`. Additional submission instructions will be announced on Piazza.

3 Implementing the Parser

Dependencies. Your code should run with `python3`. Please use only built-in python packages for this homework. Please do not import additional libraries which will break the auto-grader.

Binarization. First, let's convert the PCFG to Chomsky Normal Form, where the RHS consists of either a terminal symbol or two non-terminal symbols. This has been implemented for you in the `get_cnf` function in the `Grammar` class. (Our implemention is very basic and assumes that there is no epsilon rules whre RHS is empty. In practice, this step is more sophisticated and may involve parent annotation and markovization, e.g. see Accurate Unlexicalized Parsing by Klein and Manning.

1. If the RHS contains more than one symbol and at least one is a terminal, then convert the terminal to a new non-terminal. For example,

PCFG		CNF	
rule	score	rule	score
`A` $\rightarrow$ `B a`	p	`A` $\rightarrow$ `B NT_a`	p
		`NT_a` $\rightarrow$ `a`	1

   After this step, the RHS consists of either a single terminal or one or more non-terminals.

2. If the RHS contains more than one non-terminal, binarize it by creating new non-terminals. For example, After this step, the RHS consists of either a single terminal or two non-terminals.

PCFG		CNF	
rule	score	rule	score
`A` $\rightarrow$ `B C D`	p	`A` $\rightarrow$ `B BIN_A_C,D`	p
		`BIN_A_C,D` $\rightarrow$ `C D`	1

We will use the binarized grammar for parsing. However, the final parse tree should be expressed in the original grammar. Note that given a parse tree in the binarized grammar, the tree in the original grammar can be easily recovered.

Handling unary rules. Unary rules are rules where there is a single non-terminal on the RHS, e.g. `A` *to* `B`. They are quite common and handling unary rules is the most tricky part when implementing a basic CKY parser. The CKY algorithm we covered during the lecture only handles binary rules. Suppose we have added the non-terminal `A` for the span `[i,j]`, we need to additionally add all unary rules where `A` is on the RHS, e.g. `D` $\rightarrow$ `A` (need to further check if there is unary rules that produces `D`!). The modified CKY algorithm is

```
Input: Binarized PCFG $G = (\Sigma, N, R, S)$, a string $x = x_1 \dots x_n$
for all $0 \leq i < j \leq n$, $A \in N$ do
    initialize bestScore[i, j][A] $\leftarrow$ 0;
    initialize backpointer;
end
// Fill terminal rules
for $i \in [1, n]$ do
    for all $A \in N$ where $(p, A \rightarrow x_i) \in R$ do
        bestScore[i-1, i][A] $\leftarrow \max(p,$ bestScore[i-1, i][A]$)$;
        Update backpointer;
    end
    Add unary rules for cell [i-1, i];
end
// l: span length, i: start, j: end, k: split
for $l \in [2, n]$ do
    for $i \in [0, n - l]$ do
        $j = i + l$;
        for $k \in [i + 1, j - 1]$ do
            for $(p, A \rightarrow BC) \in R$ do
                bestScore[i, j][A]
                  $\leftarrow \max(p \times$ bestScore[i, k][B] $\times$ bestScore[k+1, j][C], bestScore[i, j][A]$)$;
                Update backpointer;
            end
        end
        // Add unary rules for non-terminals in cell [i, j]
        found $\leftarrow$ true;
        while found do
            for $A \in$ bestScore[i,j] do
                found $\leftarrow$ false;
                // Find A's unary parents
                for all $B$ where $(p, B \rightarrow A) \in R$ do
                    bestScore[i, j][B] $\leftarrow \max(p \times$ bestScore[i, j][A], bestScore[i, j][B]$)$;
                    Update backpointer;
                    found $\leftarrow$ true;
                end
            end
        end
    end
end
```

Note that

1. When handling unary rules, we need the while loop to handle unary chains, e.g. `D` $\rightarrow$ `C` $\rightarrow$ `B` $\rightarrow$ `A`. **Important**: there may be loops; make sure to break the loop if any seen rules are added to the path.

2. We have left out the pseudocode for updating backpointers, which are needed for backtracking the highest-probability tree. Think about what information you need to remember for reconstruction.

3. If the ROOT symbol is not in bestScore[0, n], then there is no valid parse given the grammar.

4. Remember to do all computation in the log space.

Implement the `parse` function in class `Grammar`, which takes in a string and print its log probability and parse tree to stdout. There are some helper functions but you don't have to use them. For evaluation, we will call the `parse` function in `eval.py`.

Check your implementation on the `arith` grammar. The reference output is in `data/arith.par`. Make sure your parser works on this toy grammar first. You can also check your parse on Stanford CoreNLP: `https://corenlp.run`.

4 Grading

Four points for matching reference outputs on `arith.sen` and seven points for matching hidden reference outputs on `wallstreet.sen`.