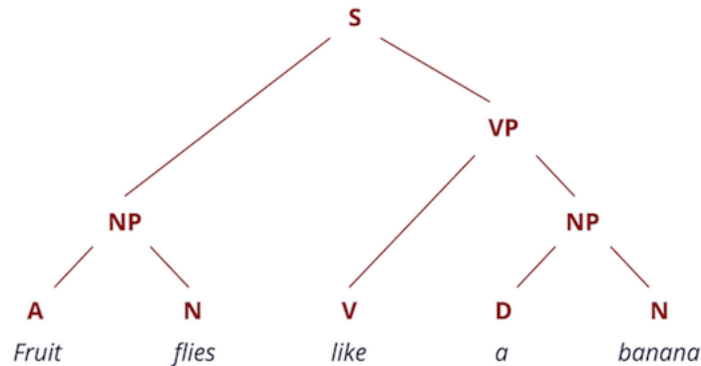


50.040 Natural Language Processing, Summer 2022
Homework 2

Due 1 July 2022, 5pm

Homework 2 will be graded by Ming Shan

Introduction Constituency parsing aims to extract a constituency-based parse tree from a sentence that represents its syntactic structure according to a phrase structure grammar. A typical constituency parse tree is shown below:



S is a distinguished start symbol, node labels such as NP (noun phrase), VP (verb phrase) are non-terminal symbols, leaf labels such as “a”, “banana” are terminal symbols.

In this homework, we will implement a constituency parser based on probabilistic context-free grammars (PCFGs) and evaluate its performance.

Dataset We will be using a version of the “Penn Treebank” released in NLTK corpora to induce PCFGs and evaluate our algorithm. The preprocessing code has been provided, do not make any changes to the text and code unless you are requested to do so. Run the code we provide to load the training and test sets as Python lists, it will take 1 minute. Since we will not tune hyper-parameters in this homework, there will be no need for a development set.

Requirements: nltk

PCFGs A probabilistic context-free grammar consists of:

1. A context-free grammar $G = (N, \Sigma, S, R)$ where N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, R is a finite set of rules (e.g., $NP \rightarrow NP PP$), $S \in N$ is the start symbol.
2. One parameter $q(A \rightarrow \beta)$ for each rule $A \rightarrow \beta$ in R . Since the grammar is in Chomsky normal form, there are only two types of rules: $A \rightarrow BC$ and $A \rightarrow \alpha$, where $A, B, C \in N$, $\alpha \in \Sigma$.

We can estimate the parameter $q(A \rightarrow \beta)$ using maximum likelihood estimation:

$$q_{MLE}(A \rightarrow \beta) = \frac{\text{count}(A \rightarrow \beta)}{\text{count}(A)} \quad (1)$$

where $\text{count}(A \rightarrow \beta)$ refers to the number of occurrences of the rule $A \rightarrow \beta$ in all the parse trees in the training set, and $\text{count}(A)$ refers to the number of occurrences of the non-terminal symbol A .

(50 points)

Question 1 [written] (3 points) To better understand PCFG, let's consider the first parse tree in the training data "cnf_train" as an example. Run the code we have provided for you and then write down the roles of `.productions()`, `.rhs()`, `.lhs()`, `.leaves()` in the ipynb notebook.

Question 2 [code] (5 points) To count the number of unique rules, nonterminals and terminals, please implement functions `collect_rules`, `collect_nonterminals`, `collect_terminals`.

Question 3 [code] (4 points) Implement the function `build_pcfg` which builds a dictionary that stores the terminal rules and nonterminal rules.

Question 4 [code] (1 point) Count and list all the terminal symbols in "cnf_test" that never appeared in the PCFG we built.

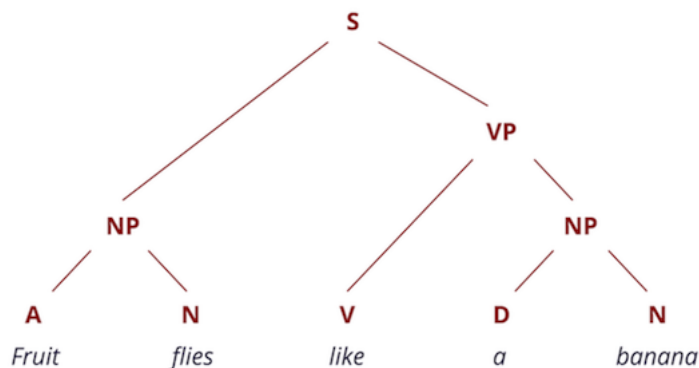
Question 5 [code] (10 points) To handle cases where terminal symbols in test data never appear in the training data, we introduce a simple smoothing technique. We first create a new "unknown" terminal symbol "unk". Next, for each original non-terminal symbol $A \in N$, we add one new rule " $A \rightarrow \text{unk}$ " to the original PCFG. The smoothed probabilities for all rules can then be estimated as:

$$q_{smooth}(A \rightarrow \beta) = \frac{\text{count}(A \rightarrow \beta)}{\text{count}(A) + 1}$$
$$q_{smooth}(A \rightarrow \text{unk}) = \frac{1}{\text{count}(A) + 1}$$

Implement the function `smooth_rules_prob` which returns the smoothed rule probabilities. Then run the code we have provided for you.

Question 6 [code] (2 points) Estimate the probability of “cnf_test[0]”, which is the first parse tree in the testing data “cnf_test” by using *s_rules_prob*.

CKY algorithm Similar to the Viterbi algorithm, the CKY algorithm is a dynamic-programming algorithm. Given a PCFG $G = (N, \Sigma, S, R)$, we can use the CKY algorithm described in class to find the highest scoring parse tree for a sentence. First, let us complete the *CKY* function from scratch using only Python built-in functions. The output should be two dictionaries π and bp , which store the optimal probability and backpointer information respectively. Given a sentence w_0, w_1, \dots, w_{n-1} , $\pi(i, k, X)$, $bp(i, k, X)$ refer to the highest score and backpointer for the (partial) parse tree that has the root X (a non-terminal symbol) and covers the word span w_i, \dots, w_{k-1} , where $0 \leq i < k \leq n$. Note that a backpointer includes both the best grammar rule chosen and the best split point.



Question 7 [code] (15 points) Implement *CKY* function according to Algorithm 1.

Question 8 [code] (7 points) Implement *build_tree* function according to Algorithm 2 to reconstruct the parse tree.

Question 9 [code] (1 points) Given a parse tree, the original sentence can be obtained by arranging the terminal symbols from left to right. Use *CKY* function to compute the max probability for the sentence from “cnf_test[0]”, which is the first parse tree in the testing data “cnf_test”.

Question 10 [code] (1 points) Generate and display the parse tree for the sentence from “cnf_test[0]”.

Question 11 [code] (1 point) Run the remaining code to evaluate your model on the entire “cnf_test”.

Algorithm 1 CKY Algorithm

```
procedure CKY(sent, rules)
  Initialization: score, back, len
  for  $i = 0, \dots, \text{len} - 1$  do
    for  $A$  in nonterminals do
      if  $A \rightarrow \text{sent}[i]$  in rules then
        Update score[(i,i+1)][A], back[(i,i+1)][A]
      end if
    end for
  end for
  for span=2,..., len do
    for begin = 0,..., (len-span) do
      end = begin + span
      for split = begin+1,..., end-1 do
        for A,B,C in nonterminals do
          if  $A \rightarrow B,C$  in rules then
            Update score[begin,end][A], back[(begin, end)][A]
          end if
        end for
      end for
    end for
  end for
end procedure
```

Algorithm 2 Build Tree algorithm

```
procedure BUILD_TREE(back, root)
  Initialization: begin, end, root_label
  Return: tree
  Read split point, left child node and right child node from back
  if right child node exists then
    build left_tree, right_tree
    tree = nltk.tree.Tree(root_label, [left_tree, right_tree])
  else
    tree = nltk.tree.Tree(root_label, [left_child])
  end if
end procedure
```

How to submit

1. Fill up your student ID and name in the Jupyter Notebook.
2. Click the Save button at the top of the Jupyter Notebook.
3. Select Cell - All Output - Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell Run All. This will run all the cells in order, and will take 20-25 minutes.
5. Once you've rerun everything, select File – Download as – PDF via LaTeX
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see! Submit your PDF on eDimension.