50.040 Natural Language Processing (Summer 2024) Homework 2

Due October,28,2024

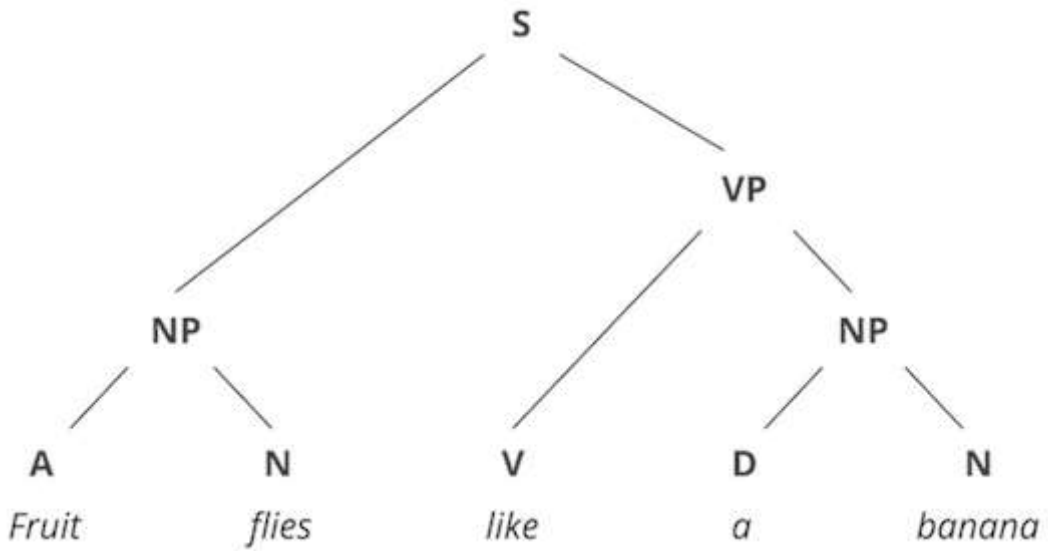
STUDENT ID: 1006184

Name: Atul Parida

Students with whom you have discussed (if any): Anutham Mukunthan, Ansh Oswal

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.

PCFGs

A probabilistic context-free grammar consists of:

- 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.
- 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: $A \rightarrow B\ C$ 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:

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

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

In [1]:
```
import copy
from collections import Counter
from nltk.tree import Tree
from nltk import Nonterminal
from nltk.corpus import LazyCorpusLoader, BracketParseCorpusReader
from collections import defaultdict
import time
```

In [2]:
```
st = time.time()
```

In [3]:
```
import nltk
nltk.download('treebank')
```

```
[nltk_data] Downloading package treebank to
[nltk_data]     C:\Users\atulp\AppData\Roaming\nltk_data...
[nltk_data]   Package treebank is already up-to-date!
```

Out[3]: True

In [4]:
```
def set_leave_lower(tree_string):
    if isinstance(tree_string, Tree):
        tree = tree_string
    else:
```

```
        tree = Tree.fromstring(tree_string)
    for idx, _ in enumerate(tree.leaves()):
        tree_location = tree.leaf_treeposition(idx)
        non_terminal = tree[tree_location[:-1]]
        non_terminal[0] = non_terminal[0].lower()
    return tree

def get_train_test_data():
    '''
    Load training and test set from nltk corpora
    '''
    train_num = 3900
    test_index = range(10)
    treebank = LazyCorpusLoader('treebank/combined', BracketParseCorpusReader, r'ws
    cnf_train = treebank.parsed_sents()[:train_num]
    cnf_test = [treebank.parsed_sents()[i+train_num] for i in test_index]
    #Convert to Chomsky norm form, remove auxiliary labels
    cnf_train = [convert2cnf(t) for t in cnf_train]
    cnf_test = [convert2cnf(t) for t in cnf_test]
    return cnf_train, cnf_test
def convert2cnf(original_tree):
    '''
    Chomsky norm form
    '''
    tree = copy.deepcopy(original_tree)

    #Remove cases like NP->DT, VP->NP
    tree.collapse_unary(collapsePOS=True, collapseRoot=True)
    #Convert to Chomsky
    tree.chomsky_normal_form()

    tree = set_leave_lower(tree)
    return tree
```

In [5]:
```
### GET TRAIN/TEST DATA
cnf_train, cnf_test = get_train_test_data()
```

In [6]:
```
cnf_train[0].pprint()
```

```
(S
  (NP-SBJ
    (NP (NNP pierre) (NNP vinken))
    (NP-SBJ|<,-ADJP-,>
      (, ,)
      (NP-SBJ|<ADJP-,>
        (ADJP (NP (CD 61) (NNS years)) (JJ old))
        (, ,))))
  (S|<VP-.>
    (VP
      (MD will)
      (VP
        (VB join)
        (VP|<NP-PP-CLR-NP-TMP>
          (NP (DT the) (NN board))
          (VP|<PP-CLR-NP-TMP>
            (PP-CLR
              (IN as)
              (NP
                (DT a)
                (NP|<JJ-NN> (JJ nonexecutive) (NN director))))
            (NP-TMP (NNP nov.) (CD 29))))))
    (. .)))
```

Question 1

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.

In [7]:
```
rules = cnf_train[0].productions()
print(rules, type(rules[0]))
```

```
[S -> NP-SBJ S|<VP-.>, NP-SBJ -> NP NP-SBJ|<,-ADJP-,>, NP -> NNP NNP, NNP -> 'pierr
e', NNP -> 'vinken', NP-SBJ|<,-ADJP-,> -> , NP-SBJ|<ADJP-,>, , -> ',', NP-SBJ|<ADJP
-,> -> ADJP ,, ADJP -> NP JJ, NP -> CD NNS, CD -> '61', NNS -> 'years', JJ -> 'old',
, -> ',', S|<VP-.> -> VP ., VP -> MD VP, MD -> 'will', VP -> VB VP|<NP-PP-CLR-NP-TMP
>, VB -> 'join', VP|<NP-PP-CLR-NP-TMP> -> NP VP|<PP-CLR-NP-TMP>, NP -> DT NN, DT ->
'the', NN -> 'board', VP|<PP-CLR-NP-TMP> -> PP-CLR NP-TMP, PP-CLR -> IN NP, IN -> 'a
s', NP -> DT NP|<JJ-NN>, DT -> 'a', NP|<JJ-NN> -> JJ NN, JJ -> 'nonexecutive', NN ->
'director', NP-TMP -> NNP CD, NNP -> 'nov.', CD -> '29', . -> '.'] <class 'nltk.gram
mar.Production'>
```

In [8]:
```
rules[0].rhs(), type(rules[0].rhs()[0])
```

Out[8]:
```
((NP-SBJ, S|<VP-.>), nltk.grammar.Nonterminal)
```

In [9]:
```
rules[10].rhs(), type(rules[10].rhs()[0])
```

Out[9]:
```
(('61',), str)
```

In [10]:
```
rules[0].lhs(), type(rules[0].lhs())
```

Out[10]: (S, nltk.grammar.Nonterminal)

In [11]:
```
print(cnf_train[0].leaves())
```

```
['pierre', 'vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board',
'as', 'a', 'nonexecutive', 'director', 'nov.', '29', '.']
```

ANSWER HERE

- .productions(): The productions() method returns a list of the productions that correspond to the non-terminal symbol of the tree. These are the production rules when compacted according to Chomsky Normal Form.
- .rhs(): The rhs() method returns the right-hand side of the production rule. This is the part of the production rule that is on the right side of the arrow i.e. either a terminal or non-terminal or both. So for example, if the production rule is A -> B C, then the right-hand side would be B C.
- .lhs(): The lhs() method returns the left-hand side of the production rule. This is the part of the production rule that is on the left side of the arrow i.e. it must be a non-terminal symbol. So for example, if the production rule is A -> B C, then the left-hand side would be A.
- .leaves(): The leaves() method returns a list of the terminal symbols in the tree. These are the symbols that do not have any children in the tree. So for example, if the tree is (A (B C)), then the leaves would be B and C.

Question 2

To count the number of unique rules, nonterminals and terminals, please implement functions **collect_rules**, **collect_nonterminals**, **collect_terminals**

In [12]:
```
def collect_rules(train_data):
    '''
    Collect the rules that appear in data.
    params:
        train_data: list[Tree] --- list of Tree objects
    return:
        rules: list[nltk.grammar.Production] --- list of rules (Production objects)
        rules_counts: Counter object --- a dictionary that maps one rule (nltk.Nont
                                         occurences (int) in train data.
    '''
    rules = list()
    rule_counts = Counter()
    ### YOUR CODE HERE
    # Include rules that appear in the training data
    for tree in train_data:
        rules += tree.productions()
    rule_counts = Counter(rules)
    ### YOUR CODE HERE
    return rules, rule_counts
```

```
def collect_nonterminals(rules):
    '''
    collect nonterminals that appear in the rules
    params:
        rules: list[nltk.grammar.Production] --- list of rules (Production objects)
    return:
        nonterminals: set(nltk.Nonterminal) --- set of nonterminals
    '''
    ### YOUR CODE HERE
    nonterminals = set()
    for rule in rules:
        nonterminals.add(rule.lhs())
        for nonterminal in rule.rhs():
            if isinstance(nonterminal, Nonterminal):
                nonterminals.add(nonterminal)
    ### END OF YOUR CODE
    return nonterminals

def collect_terminals(rules):
    '''
    collect terminals that appear in the rules
    params:
        rules: list[nltk.grammar.Production] --- list of rules (Production objects)
    return:
        terminals: set of strings --- set of terminals
    '''
    ### YOUR CODE HERE
    terminals = set()
    for rule in rules:
        for terminal in rule.rhs():
            if not isinstance(terminal, Nonterminal):
                terminals.add(terminal)
    ### END OF YOUR CODE
    return terminals
```

In [13]:
```
train_rules, train_rules_counts = collect_rules(cnf_train)
nonterminals = collect_nonterminals(train_rules)
terminals = collect_terminals(train_rules)
```

In [14]:
```
### CORRECT ANSWER (19xxxx, 3xxxx, 1xxxx, 7xxx)
len(train_rules), len(set(train_rules)), len(terminals), len(nonterminals)
```

Out[14]: (196646, 31656, 11367, 7869)

In [15]:
```
print(train_rules_counts.most_common(5))
```

```
[(, -> ',', 4876), (DT -> 'the', 4726), (. -> '.', 3814), (PP -> IN NP, 3273), (S|<V
P-.> -> VP ., 3003)]
```

Question 3

Implement the function **build_pcfg** which builds a dictionary that stores the terminal rules and nonterminal rules.

In [18]:
```
def build_pcfg(rules_counts):
    '''
    Build a dictionary that stores the terminal rules and nonterminal rules.
    param:
        rules_counts: Counter object --- a dictionary that maps one rule to its num
    return:
        rules_dict: dict(dict(dict)) --- a dictionary has a form like:
                    rules_dict = {'terminals':{'NP':{'the':1000,'an':500}, 'ADJ':{'
                                  'nonterminals':{'S':{'NP@VP':1000},'NP':{'NP@NP':
    When building "rules_dict", you need to use "lhs()", "rhs()" funtion and conver
    **All the keys in the dictionary are of type str**.
    '@' is used as a special symbol to split left and right nonterminal strings.
    '''
    ### YOUR CODE HERE
    rules_dict = dict()
    rules_dict['terminals'], rules_dict['nonterminals'] = defaultdict(dict), defaul
    for rule in rules_counts:
        tmp = []
        rhs = rule.rhs()
        lhs = str(rule.lhs())
        if len(rhs) == 1:
            if type(rhs[0]) == str: # Assume that the terminal is a string
                rules_dict['terminals'][lhs][rhs[0]] = rules_counts[rule]
        else:
            for r in rhs:
                tmp.append(str(r))
            at_str = '@'.join(tmp)
            rules_dict['nonterminals'][lhs][at_str] = rules_counts[rule]
    ### END OF YOUR CODE
    return rules_dict
```

In [19]:
```
train_rules_dict = build_pcfg(train_rules_counts)
```

Question 4

Find the terminal symbols in "cnf_test[0]" that never appeared in the PCFG we built.

In [20]:
```
### YOUR CODE HERE
test_terminals = collect_terminals(cnf_test[0].productions())
for terminal in test_terminals:
    if terminal not in train_rules_dict['terminals']:
        print(terminal)
```

```
does
exercise
to
leading
the
line-item
have
a
two
legal
said
*
experts
bush
president
0
veto
constitutional-law
authority
n't
```

Question 5

We can use smoothing techniques to handle these cases. A simple smoothing method is as follows. We first create a new "unknown" terminal symbol unk.

Next, for each original non-terminal symbol $A \in N$, we add one new rule $A \rightarrow unk$ to the original PCFG.

The smoothed probabilities for all rules can then be estimated as:

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

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

where $|V|$ is the count of unique terminal symbols.

Implement the function **smooth_rules_prob** which returns the smoothed rule probabilities

In [21]:
```
def smooth_rules_prob(rules_counts):
    '''
    params:
        rules_counts: dict(dict(dict)) --- a dictionary has a form like:
                      rules_counts = {'terminals':{'NP':{'the':1000,'an':500}, 'ADJ
                                      'nonterminals':{'S':{'NP@VP':1000},'NP':{'NP@

    return:
        rules_prob: dict(dict(dict)) --- a dictionary that has a form like:
                                rules_prob = {'terminals':{'NP':{'the':0.6,'an':0.3,
                                                           'ADJ':{'nice':0.6,'good':
                                                           'S':{'<unk>':0.01}}}
```

```
                                                'nonterminals':{'S':{'NP@VP':0.99}}
    '''
    rules_prob = copy.deepcopy(rules_counts)
    unk = '<unk>'
    ### YOUR CODE HERE
    # if a nonterminal is queried from the terminal dictionary, return the probabil
    for keys in rules_prob['terminals'].values():
        sum = 0
        for key_count in keys.values():
            sum += key_count
        for word, key_count in keys.items():
            keys[word] = key_count / (sum + 1)
        keys[unk] = 1 / (sum + 1)
    for key, keys in rules_prob['nonterminals'].items():
        sum = 0
        for key_count in keys.values():
            sum += key_count
        for word, key_count in keys.items():
            keys[word] = key_count / (sum + 1)
        if key not in rules_prob['terminals']:
            rules_prob['terminals'][key] = {unk:(1 / (sum + 1))}
    ### END OF YOUR CODE
    return rules_prob
```

In [22]:
```
s_rules_prob = smooth_rules_prob(train_rules_dict)
terminals.add('<unk>')
```

In [23]:
```
print(s_rules_prob['nonterminals']['S']['NP-SBJ@S|<VP-.>'])
print(s_rules_prob['nonterminals']['S']['NP-SBJ-1@S|<VP-.>'])
print(s_rules_prob['nonterminals']['NP']['NNP@NNP'])
print(s_rules_prob['terminals']['NP'])
```

```
0.1300172371337109
0.025240088648116228
0.039506305917861376
{'<unk>': 5.389673385792821e-05}
```

In [24]:
```
len(terminals)
```

Out[24]:
```
11368
```

Question 6

Estimate the probability of "cnf_train[0]", which is the first parse tree in the training data "cnf_train" by using "s_rules_prob".

In [25]:
```
probability = 1

# Probability of cnf_train[0] using s_rules_prob
for rule in cnf_train[0].productions():
    lhs = str(rule.lhs())
    rhs = [str(r) for r in rule.rhs()]
    if len(rhs) == 1:
```

```
        if rhs[0] in s_rules_prob['terminals'][lhs]:
            probability *= s_rules_prob['terminals'][lhs][rhs[0]]
        else:
            probability *= s_rules_prob['terminals'][lhs]['<unk>']
    else:
        at_str = '@'.join(rhs)
        if at_str in s_rules_prob['nonterminals'][lhs]:
            probability *= s_rules_prob['nonterminals'][lhs][at_str]
        else:
            probability *= s_rules_prob['terminals'][lhs]['<unk>']

print(probability)
```

```
1.3824366894669981e-52
```

CKY Algorithm

Similar to the Viterbi algorithm, the CKY algorithm is a dynamic-programming algorithm. Given a PCFG $G = (N,\ \Sigma,\ S,\ R,\ q)$, 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 and the Numpy package.

The output should be two dictionaries π and bp, which store the optimal probability and backpointer information respectively.

Given a sentence $w_0, w_1, \ldots, w_{n-1}$, $\pi(i, k, X)$, $bp(i, k, X)$ 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 $w_i, \ldots, w_{k-1}$, 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

Implement **CKY** function.

In [26]:
```
def CKY(sent, rules_prob):
    '''
    params:
        sent: list[str] --- a list of strings
        rules_prob: dict(dict(dict)) --- a dictionary that has a form like:
                                            rules_prob = {'terminals':{'NP':{'the':0
                                                                             'ADJ':{'nice'
                                                                             'S':{'<unk>':
                                                                   'nonterminals':{'S':{'NP@V
    return:
        score: dict(dict) --- score[(i,i+span)][root] represents the highest score
                           across words w_i, w_{i+1},..., w_{i+span-1}.
        back: dict(dict) --- back[(i,i+span)][root] = (split , left_child, right_ch
                          left_child: str; right_child: str.
```

```
    '''
    score = defaultdict(dict)
    back = defaultdict(dict)
    sent_len = len(sent)
    ### YOUR CODE HERE
    # CKY algorithm
    for i in range(0, sent_len):
        term = sent[i]
        for A, dicts in rules_prob['terminals'].items():
            if term in dicts:
                score[(i, i+1)][A] = dicts[term]
                back[(i, i+1)][A] = (-1, term, '')

    for span in range(2, sent_len + 1):
        for begin in range(0, sent_len - span + 1):
            end = begin + span

            for split in range(begin + 1, end):
                for A in rules_prob['nonterminals'].keys():
                    for key in rules_prob['nonterminals'][A].keys():
                        B = key.split('@')[0]
                        C = key.split('@')[1]

                        if (B in score[(begin, split)].keys()) and (C in score[(spl
                            updated_score = rules_prob['nonterminals'][A][key] *\
                                            score[(begin, split)][B] *\
                                            score[(split, end)][C]

                            if (A not in score[(begin, end)]) or (updated_score > s
                                score[(begin, end)][A] = updated_score
                                back[(begin, end)][A] = (split, B, C)

    ### END OF YOUR CODE
    return score, back
```

Question 8

What is the time complexity of CKY function (in worst case), please verify.

The time complexity of the CKY Algorithm (in the worst case) is O(n^3 * |N|^2), where n is the length of the sentence and |N| is the number of non-terminal symbols in the grammar. This is because the algorithm iterates over all possible spans of the sentence and for each span, it iterates over all possible non-terminal symbols in the grammar. The algorithm also iterates over all possible splits of the span to find the best split point. This results in a time complexity of O(n^3 * |N|^2).

Mathematically, this time complexity can be derived as follows:

- The algorithm iterates over all possible spans of the sentence, which results in a time complexity of O(n^2).

- For each span, the algorithm iterates over all possible non-terminal symbols in the grammar, which results in a time complexity of O(|N|).
- For each non-terminal symbol, the algorithm iterates over all possible splits of the span to find the best split point, which results in a time complexity of O(n).
- Combining these complexities, we get a total time complexity of O(n^3 * |N|^2).

Question 9

Implement **build_tree** function to reconstruct the parse tree.

In [27]:
```
def build_tree(back, root):
    '''
    Build the tree recursively.
    params:
        back: dict() --- back[(i,i+span)][X] = (split , left_child, right_child); s
        root: tuple() --- (begin, end, nonterminal_symbol), e.g., (0, 10, 'S
    return:
        tree: nltk.tree.Tree
    '''
    begin = root[0]
    end = root[1]
    root_label = root[2]
    ### YOUR CODE HERE
    split, left, right = back[(begin, end)][root_label]
    if right != '': # Assume that if right is not empty, then left is not empty
        l_tree = build_tree(back, (begin, split, left))
        r_tree = build_tree(back, (split, end, right))

        tree = nltk.tree.Tree(root_label, [l_tree, r_tree])
    else:
        tree = nltk.tree.Tree(root_label, [left])
    ### END OF YOUR CODE
    return tree
```

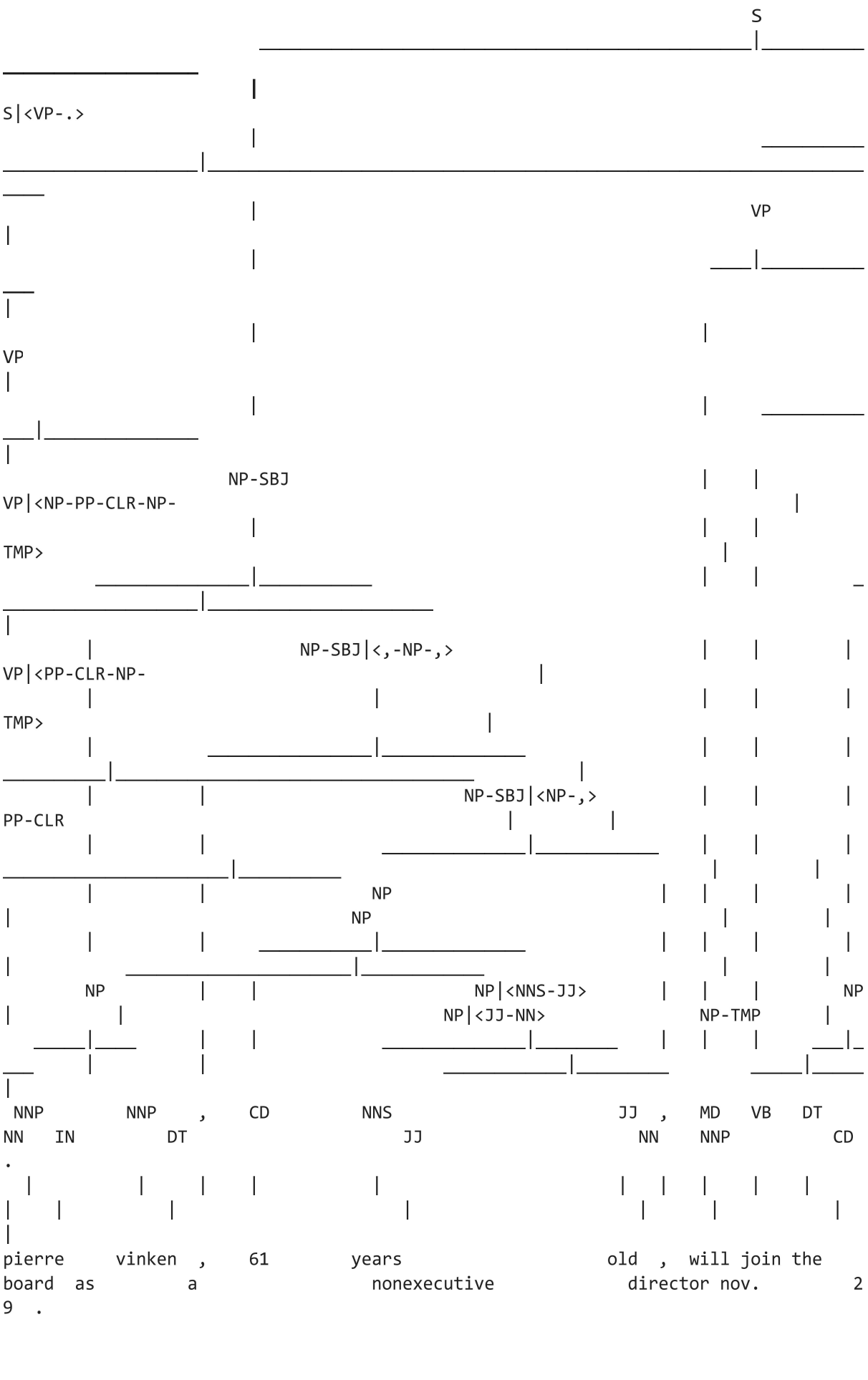
Question 10

- 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 form "cnf_train[0]", which is the first parse tree in the training data "cnf_train".
- Generate and display the parse tree of the sentence.

In [28]:
```
train_rules, train_rules_counts = collect_rules(cnf_train)
train_non, train_term = collect_nonterminals(train_rules), collect_terminals(train_
train_rules_dict = build_pcfg(train_rules_counts)
train_rules_prob = smooth_rules_prob(train_rules_dict)
train_term.add('<unk>')

train_sent = cnf_train[0].leaves()
score, back = CKY(train_sent, train_rules_prob)
```

In [29]:
```
train_tree = build_tree(back, (0, len(train_sent), 'S'))
train_tree.pretty_print()
```


S
S|<VP-.>
VP
VP
VP|<NP-PP-CLR-NP-
TMP>
NP-SBJ
NP-SBJ|<,-NP-,>
VP|<PP-CLR-NP-
TMP>
NP-SBJ|<NP-,>
PP-CLR
NP
NP
NP|<NNS-JJ>
NP|<JJ-NN>
NP-TMP
NNP NNP , CD NNS JJ , MD VB DT NN IN DT JJ NN NNP CD .
pierre vinken , 61 years old , will join the board as a nonexecutive director nov. 29 .

Question 11

Run the remaining code to test your model on test data "cnf_test".

In [30]:
```
def set_leave_index(tree):
    '''
    Label the leaves of the tree with indexes
    Arg:
        tree: original tree, nltk.tree.Tree
    Return:
        tree: preprocessed tree, nltk.tree.Tree
    '''
    for idx, _ in enumerate(tree.leaves()):
        tree_location = tree.leaf_treeposition(idx)
        non_terminal = tree[tree_location[:-1]]
        non_terminal[0] = non_terminal[0] + "_" + str(idx)
    return tree

def get_nonterminal_bracket(tree):
    '''
    Obtain the constituent brackets of a tree
    Arg:
        tree: tree, nltk.tree.Tree
    Return:
        nonterminal_brackets: constituent brackets, set
    '''
    nonterminal_brackets = set()
    for tr in tree.subtrees():
        label = tr.label()
        #print(tr.leaves())
        if len(tr.leaves()) == 0:
            continue
        start = tr.leaves()[0].split('_')[-1]
        end = tr.leaves()[-1].split('_')[-1]
        if start != end:
            nonterminal_brackets.add(label+'-('+start+':'+end+')')
    return nonterminal_brackets

def word2lower(w, terminals):
    '''
    Map an unknow word to "unk"
    '''
    return w.lower() if w in terminals else '<unk>'
```

In [31]:
```
correct_count = 0
pred_count = 0
gold_count = 0
for i, t in enumerate(cnf_test):
    #Protect the original tree
    t = copy.deepcopy(t)
    sent = t.leaves()
    #Map the unknow words to "unk"
    sent = [word2lower(w.lower(), terminals) for w in sent]
```

```
    #CKY algorithm
    score, back = CKY(sent, s_rules_prob)
    candidate_tree = build_tree(back, (0, len(sent), 'S'))

    #Extract constituents from the gold tree and predicted tree
    pred_tree = set_leave_index(candidate_tree)
    pred_brackets = get_nonterminal_bracket(pred_tree)

    #Count correct constituents
    pred_count += len(pred_brackets)
    gold_tree = set_leave_index(t)
    gold_brackets = get_nonterminal_bracket(gold_tree)
    gold_count += len(gold_brackets)
    current_correct_num = len(pred_brackets.intersection(gold_brackets))
    correct_count += current_correct_num

    print('#'*20)
    print('Test Tree:', i+1)
    print('Constituent number in the predicted tree:', len(pred_brackets))
    print('Constituent number in the gold tree:', len(gold_brackets))
    print('Correct constituent number:', current_correct_num)

recall = correct_count/gold_count
precision = correct_count/pred_count
f1 = 2*recall*precision/(recall+precision)
```

```
####################
Test Tree: 1
Constituent number in the predicted tree: 20
Constituent number in the gold tree: 20
Correct constituent number: 14
####################
Test Tree: 2
Constituent number in the predicted tree: 54
Constituent number in the gold tree: 54
Correct constituent number: 30
####################
Test Tree: 3
Constituent number in the predicted tree: 30
Constituent number in the gold tree: 30
Correct constituent number: 23
####################
Test Tree: 4
Constituent number in the predicted tree: 17
Constituent number in the gold tree: 17
Correct constituent number: 16
####################
Test Tree: 5
Constituent number in the predicted tree: 32
Constituent number in the gold tree: 32
Correct constituent number: 26
####################
Test Tree: 6
Constituent number in the predicted tree: 40
Constituent number in the gold tree: 40
Correct constituent number: 18
####################
Test Tree: 7
Constituent number in the predicted tree: 22
Constituent number in the gold tree: 22
Correct constituent number: 7
####################
Test Tree: 8
Constituent number in the predicted tree: 18
Constituent number in the gold tree: 18
Correct constituent number: 6
####################
Test Tree: 9
Constituent number in the predicted tree: 28
Constituent number in the gold tree: 28
Correct constituent number: 16
####################
Test Tree: 10
Constituent number in the predicted tree: 40
Constituent number in the gold tree: 40
Correct constituent number: 8
```

In [32]:
```
print('Overall precision: {:.3f}, recall: {:.3f}, f1: {:.3f}'.format(precision, rec
```

```
Overall precision: 0.545, recall: 0.545, f1: 0.545
```