50.040 Natural Language Processing (Summer 2020) Homework 2

Due

STUDENT ID: 1003056

Name: Ivan Christian

Students with whom you have discussed (if any): Tee Zhi Yao, Eda Tan

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]:

```
from nltk import grammar
```

In [3]:

```
st = time.time()
```

In [4]:

```
import nltk
nltk.download('treebank')
```

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

Out[4]:

```
True
```

In [5]:

```
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'wsj_.*\.mrg')
    cnf_train = treebank.parsed_sents()[:train_num]
    cnf_test = [treebank.parsed_sents()[i+train_num] for i in test_index]
```

```python
    #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 [6]:

```python
### GET TRAIN/TEST DATA
cnf_train, cnf_test = get_train_test_data()
```

In [7]:

```python
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 writedown the roles of.productions(), .rhs(), .lhs(), .leaves()in the ipynb notebook.

In [8]:

```python
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 -> 'pierre', NNP -> 'vin
ken', 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 -> 'as', 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.grammar.Production'>
```

In [9]:

```
rules[0].rhs(), type(rules[0].rhs()[0])
```

Out[9]:

```
((NP-SBJ, S|<VP-.>), nltk.grammar.Nonterminal)
```

In [10]:

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

Out[10]:

```
(('61',), str)
```

In [11]:

```
rules[0].lhs(), type(rules[0].lhs())
```

Out[11]:

```
(S, nltk.grammar.Nonterminal)
```

In [12]:

```
print(cnf_train[0].leaves())
```

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

ANSWER HERE

- productions() :
- rhs(): rhs will check for the next word rule
- lhs(): lhs will check for the previous word rule
- leaves(): The leaves for the tree, when joined form the whole sentence.

Question 2

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

In [13]:

```
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.Nonterminal) to its
number of
                                        occurences (int) in train data.
    '''
    rules = list()
    rules_counts = Counter()
    ### YOUR CODE HERE (~ 2 lines)
    for i in range(len(train_data)):
        rule = train_data[i].productions()
        for j in rule:
            rules.append(j)
            if j in rules_counts:
                rules_counts[j] += 1
            else:
                rules_counts[j] = 0
    ### YOUR CODE HERE
```

```
    return rules, rules_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
    '''
    nonterminals = list()
    ### YOUR CODE HERE (at least one line)
    for rule in rules:
        if nltk.grammar.is_nonterminal(rule.rhs()[0]):
            nonterminals.append(rule.rhs()[1])
    ### END OF YOUR CODE
    return set(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
    '''
    terminals = list()
    ### YOUR CODE HERE (at least one line)
    for rule in rules:
        if type(rule.rhs()[0]) == str:
            terminals.append(rule.rhs()[0])
    ### END OF YOUR CODE
    return set(terminals)
```

In [14]:

```
train_rules, train_rules_counts = collect_rules(cnf_train)
nonterminals = collect_nonterminals(train_rules)
terminals = collect_terminals(train_rules)
```

In [15]:

```
### CORRECT ANSWER (19xxxx, 3xxxx, 1xxxx, 7xxx)
len(train_rules), len(set(train_rules)), len(terminals), len(nonterminals)
```

Out[15]:

```
(196646, 31656, 11367, 7130)
```

In [16]:

```
print(train_rules_counts.most_common(5))
```

```
[(, -> ',', 4875), (DT -> 'the', 4725), (. -> '.', 3813), (PP -> IN NP, 3272), (S|<VP-.> -> VP .,
3002)]
```

Question 3

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

In [17]:

```
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 number of
occurences in train data.
    return:
        rules_dict: dict(dict(dict)) --- a dictionary has a form like:
                    rules_dict = {'terminals':{'NP':{'the':1000,'an':500}, 'ADJ':
```

```
                    rules_dict = {'terminals':{'NP':{'the':1000,'an':500},'ADJ':
{'nice':500,'good':100}},
                                  'nonterminals':{'S':{'NP@VP':1000},'NP':{'NP@NP':540}}}
    When building "rules_dict", you need to use "lhs()", "rhs()" funtion and convert Nonterminal t
o str.
    All the keys in the dictionary are of type str.
    '@' is used as a special symbol to split left and right nonterminal strings.
    '''

    rules_dict = dict()
    ### rules_dict['terminals'] contains rules like "NP->'the'"
    ### rules_dict['nonterminals'] contains rules like "S->NP@VP"
    rules_dict['terminals'] = defaultdict(dict)
    rules_dict['nonterminals'] = defaultdict(dict)

    ### YOUR CODE HERE

    for rule,count in rules_counts.items():

        lhs = str(rule.lhs()) #Always nonterminal
        rhs = rule.rhs() #can be both str or Nonterminal

        check_type = type(rule.rhs()[0]) == nltk.grammar.Nonterminal
        rhs_at = '@'.join([str(x) for x in rhs])

        if check_type: #Nonterminal
            rules_dict['nonterminals'].setdefault(lhs, {})
            rules_dict['nonterminals'][lhs].setdefault(rhs_at, 1)
            rules_dict['nonterminals'][lhs][rhs_at] += count
        else:# terminal
            rules_dict['terminals'].setdefault(lhs, {})
            rules_dict['terminals'][lhs].setdefault(rhs_at, 1)
            rules_dict['terminals'][lhs][rhs_at] += count

    return rules_dict
```

In [18]:

```
train_rules_dict = build_pcfg(train_rules_counts)
```

Question 4

Estimate the probability of rule $NP\rightarrow NNP@NNP$

In [19]:

```
numerator = train_rules_dict['nonterminals']['NP']['NNP@NNP'] #numerator
denominator = sum(train_rules_dict['nonterminals']['NP'].values())


probability = numerator/denominator
probability
```

Out[19]:

```
0.03950843529348353
```

Question 5

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

In [20]:

```
# terminal symbols
prod = cnf_test[0].productions()
lhs_ls = list()


for rule in prod:
    if type(rule.rhs()[0]) != nltk.grammar.Nonterminal:
        lhs_ls.append(rule.lhs()) #List of terminal symbols in the sentence
```

```
for symb in set(lhs_ls):
    if str(symb) not in list(train_rules_dict['terminals'].keys()): #pcfg we built
        print(str(symb))
```

Question 6

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':
{'nice':500,'good':100}},
                                      'nonterminals':{'S':{'NP@VP':1000},'NP':{'NP@NP':540}}}

    return:
        rules_prob: dict(dict(dict)) --- a dictionary that has a form like:
                               rules_prob = {'terminals':{'NP':{'the':0.6,'an':0.3, '<unk>':0.1},
                                                          'ADJ':{'nice':0.6,'good':0.3,'<unk>':0.1,
                                                          'S':{'<unk>':0.01}}}
                                             'nonterminals':{'S':{'NP@VP':0.99}}
    '''
    rules_prob = copy.deepcopy(rules_counts)
    unk = '<unk>'
    ### Hint: don't forget to consider nonterminal symbols that don't appear in
rules_counts['terminals'].keys()
    ### YOUR CODE HERE

    for term in rules_counts:
        rules_terminal = rules_counts[term]
        for key in rules_terminal:
            denominator = sum(rules_terminal[key].values())

            for word,count in rules_terminal[key].items():
                rules_prob[term][key][word] = count / (denominator + 1)
            rules_prob['terminals'][key][unk] = 1 / (denominator + 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
```

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 $w_0, w_1, ...,w_{n-1}$, $\pi(i, k, X)$, $bp(i, k, X)$ 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 $w_i, ..., w_{k-1}$, where $0 \le i < k \le n$. Note that a backpointer includes both the best grammar rule chosen and the best split point.

Question 7

Implement **CKY** function and run the test code to check your implementation.

In [25]:

```
import numpy as np
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':0.6,'an':0.3, '<u
k>':0.1},
                                                                           'ADJ':
{'nice':0.6,'good':0.3,'<unk>':0.1},
                                                                           'S':{'<unk>':0.01}}}
                                                              'nonterminals':{'S':{'NP@VP':0.99}}
    return:
        score: dict() --- score[(i,i+span)][root] represents the highest score for the parse
(sub)tree that has the root "root"
                          across words w_i, w_{i+1},..., w_{i+span-1}.
        back: dict() --- back[(i,i+span)][root] = (split , left_child, right_child); split: int;
                         left_child: str; right_child: str.
    '''
    score = defaultdict(dict)
    back = defaultdict(dict)
    sent_len = len(sent)
    ### YOUR CODE HERE
    '''
    CKY Algorithm based on the pseudocode given
    '''
    for i in range(sent_len):
        word = sent[i]
        for a,terminal in rules_prob['terminals'].items():
            if word in terminal:
                score[(i, i+1)][a] = terminal[word]
                back[(i, i+1)][a] = (-1, word, '')
    for span in range(2,sent_len+1):
        for begin in range(sent_len+1-span):
            end = begin + span
            for split in range(begin + 1, end):
                for a, bc in rules_prob['nonterminals'].items():
                    for bc_key, bc_val in bc.items():
                        b_key , c_key = bc_key.split('@')
                        c_key = c_key.replace('@', '')
                        if b_key in score[(begin,split)] and c_key in score[(split,end)]:
                            prob = bc_val * score[(begin,split)][b_key] * score[(split,end)][c_key]
                            if a in score[(begin, end)]:
```

```
                            if a in score[(begin, end)]:
                                if prob > score[(begin, end)][a]:
                                    score[(begin,end)][a] = prob
                                    back[(begin,end)][a] = (split, b_key,c_key)
                            else:
                                score[(begin,end)][a] = prob
                                back[(begin,end)][a] = (split, b_key,c_key)

    # probability is not log
    ### END OF YOUR CODE
    return score, back
```

In [26]:

```
sent = cnf_train[0].leaves()
score, back = CKY(sent, s_rules_prob)
```

In [27]:

```
score[(0, len(sent))]['S']
```

Out[27]:

```
9.135335125206641e-52
```

Question 8

Implement **build_tree** function according to algorithm 2 to reconstruct theparse tree

In [28]:

```
def build_tree(back, root):
    '''
    Build the tree recursively.
    params:
        back: dict() --- back[(i,i+span)][X] = (split , left_child, right_child); split:int; 
left_child: str; right_child: str.
        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 != '':
        build_left_tree = build_tree(back, (begin, split, left))
        build_right_tree = build_tree(back, (split, end, right))

        tree = nltk.tree.Tree(root_label, [build_left_tree,build_right_tree])
    else:
        tree = nltk.tree.Tree(root_label, [left])


    ### END OF YOUR CODE
    return tree
```

In [29]:

```
build_tree(back, (0, len(sent), 'S')).pprint()
```

```
(S
  (NP-SBJ
    (NP (NNP pierre) (NNP vinken))
    (NP-SBJ|<,-NP-,>
      (, ,)
      (NP-SBJ|<NP-,>
        (NP (CD 61) (NP|<NNS-JJ> (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 9

In [30]:

```python
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 unknown word to "unk"
    '''
    return w.lower() if w in terminals else '<unk>'
```

In [31]:

```python
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'))
```

```
    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, recall, f1))
```

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

In [33]:

```
print('Overall precision: {:.3f}, recall: {:.3f}, f1: {:.3f}'.format(precision, recall, f1))
```

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

In [34]:

```
et=time.time()
print(et - st)
```

```
667.4941356182098
```