homework2-1002949

July 2, 2020

1 50.040 Natural Language Processing (Summer 2020) Homework 2

Due

1.0.1 STUDNET ID: 1002949

1.0.2 Name: Hong Pengfei

1.0.3 Students with whom you have discussed (if any):NA

[1]:
```python
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
import numpy as np
```

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

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

```
[nltk_data] Downloading package treebank to /home/pengfei/nltk_data…
[nltk_data]   Package treebank is already up-to-date!
```

[3]: True

[4]:
```python
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]
    #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
```

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

[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))))))
  (. .)))
```

1.1 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.

[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 ->
'pierre', 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 -> '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'>
```

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

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

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

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

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

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

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

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

ANSWER HERE - productions(): Generate the production rules that formed the tree

Generate the productions that correspond to the non-terminal nodes of the tree. For each subtree of the form (P: C1 C2 … Cn) this produces a production of the form P -> C1 C2 … Cn. - rhs(): the right hand side elements (the childrens) in the production rule - lhs(): the left hand side elements (the parent) in the production rule - leaves(): a list of strings containing this tree's leaves.

The order reflects the order of the leaves in the tree's hierarchical structure.

1.2 Question 2

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

[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.
↪Nonterminal) to its number of
                                                 occurences (int) in train data.
    '''
    rules = list()
    rules_counts = Counter()
    ### YOUR CODE HERE (~ 2 lines)
    rules = [oo  for o in train_data for oo in o.productions()]
    rules_counts = Counter(rules)
    ### 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)
    nonterminals = [o for r in rules for o in list(r.rhs()) + [r.lhs()] if␣
↪type(o) == nltk.Nonterminal ]
    ### END OF YOUR CODE
```

```python
    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)
    terminals = [o for r in rules for o in list(r.rhs()) if type(o) == str]
    ### END OF YOUR CODE

    return set(terminals)
```

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

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

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

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

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

1.3 Question 3

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

[16]:
```python
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':{'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 to 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 r,n in rules_counts.items():
        terminal = 'terminals' if type(r.rhs()[0]) == str else 'nonterminals'
        if terminal == 'nonterminals':
            rhs = '@'.join(list(str(o) for o in r.rhs()))
        else:
            rhs = r.rhs()[0]
        lhs = str(r.lhs())
        if lhs not in rules_dict[terminal].keys():
            rules_dict[terminal][lhs] = {}

        if rhs not in rules_dict[terminal][lhs].keys():
            rules_dict[terminal][lhs][rhs] = n
        else:
            rules_dict[terminal][lhs][rhs] += n
    ### END OF YOUR CODE
    return rules_dict
```

[17]:
```
train_rules_dict = build_pcfg(train_rules_counts)
```

1.4 Question 4

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

[18]:
```
denom = sum(train_rules_dict['nonterminals']['NP'].values()) +␣
↪sum(train_rules_dict['terminals']['NP'].values())
nume = train_rules_dict['nonterminals']['NP']['NNP@NNP']
print(f'The estimated probability for NP -> NNP@NNP is {nume / denom}')
```

```
The estimated probability for NP -> NNP@NNP is 0.03950843529348353
```

1.5 Question 5

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

```
[19]: set(o for o in cnf_test[0].leaves() if type(o) == str) - set([oo for o in␣
       ↪train_rules_dict['terminals'].values() for oo in o.keys()])
```

```
[19]: {'constitutional-law'}
```

1.6 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 $A \to unk$ to the original PCFG.

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

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

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

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

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

```
[20]: 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 parent in set(list(rules_prob['terminals'].keys()) +␣
       ↪list(rules_prob['nonterminals'].keys())):
```

```python
        total = 1
        if parent in rules_prob['terminals'].keys():
            total += sum(rules_prob['terminals'][parent].values())

        if parent in rules_prob['nonterminals'].keys():
            total += sum(rules_prob['nonterminals'][parent].values())


        if parent in rules_prob['terminals'].keys():
            for child in rules_prob['terminals'][parent].keys():
                rules_prob['terminals'][parent][child] /= total
            rules_prob['terminals'][parent]['<unk>'] = 1/total

        if parent in rules_prob['nonterminals'].keys():
            for child in rules_prob['nonterminals'][parent].keys():
                rules_prob['nonterminals'][parent][child] /= total
    ### END OF YOUR CODE
    return rules_prob
```

[21]:
```python
s_rules_prob = smooth_rules_prob(train_rules_dict)
terminals.add('<unk>')
```

[22]:
```python
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}
```

[23]:
```python
len(terminals)
```

[23]:
```
11368
```

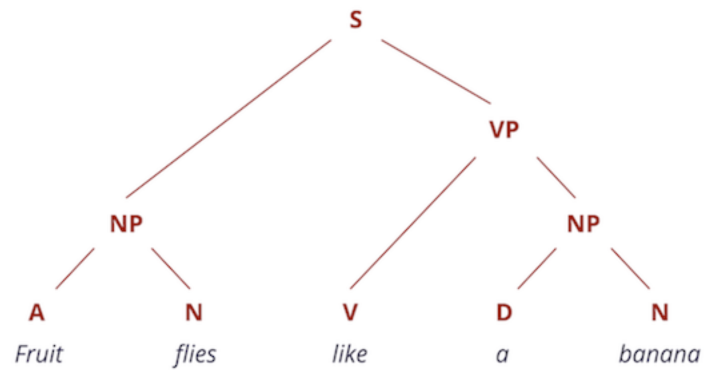
1.7 CKY Algorithm

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 $0 \leq i < k \leq n$. Note that a backpointer includes both the best grammar rule chosen and the best split point.



1.8 Question 7

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

[24]:
```
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, '<unk>':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


    for span in range(1, sent_len+1):
        for i in range(sent_len+1-span):
```

```
            temp_dict = defaultdict(list)

            # A -> alpha
            if span == 1:
                alpha = sent[i]
                split = 0

                for A in rules_prob['terminals'].keys():
                    if alpha in rules_prob['terminals'][A].keys(): temp_dict[A].
↪append((np.log(rules_prob['terminals'][A][alpha]), split, alpha, alpha))

            # A -> B C
            else:

                for split in range(1, span):

                    Bs = back[(i,i+split)].keys()
                    Cs = back[(i+split,i+span)].keys()
                    for A in rules_prob['nonterminals'].keys():
                        for BC, p in rules_prob['nonterminals'][A].items():
                            B, C = BC.split('@')
                            if B in Bs and C in Cs: temp_dict[A].append((np.
↪log(p) + score[(i,i+split)][B] + score[(i+split,i+span)][C], split, B, C))

            # best score
            for A, temp_list in temp_dict.items():
                best_score, split, B, C = max(temp_list, key=lambda x:x[0])
                score[(i, i+span)][A] = best_score
                back[(i, i+span)][A] = (split, B, C)


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

[25]:
```
sent = cnf_train[0].leaves()
score, back = CKY(sent, s_rules_prob)
```

[26]:
```
np.exp(score[(0, len(sent))]['S'])
```

[26]: 9.135335125206607e-52

1.9 Question 8

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

[27]:
```
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_label, right_label = back[(begin, end)][root_label]
    if split == 0:
        tree = Tree(root_label, [left_label])
    else:
        left_child = build_tree(back, (begin, begin+split, left_label))
        right_child = build_tree(back, (begin+split, end, right_label))
        tree = Tree(root_label, [left_child, right_child])
    ### END OF YOUR CODE
    return tree
```

[28]:
```
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))))))
    (. .)))
```

1.10 Question 9

[29]:
```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 unknow word to "unk"
    '''
    return w.lower() if w in terminals else '<unk>'
```

[30]:
```python
correct_count = 0
pred_count = 0
```

```python
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: 20
####################
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: 24
####################
Test Tree: 6
Constituent number in the predicted tree: 40
Constituent number in the gold tree: 40
Correct constituent number: 19
####################
Test Tree: 7
Constituent number in the predicted tree: 22
Constituent number in the gold tree: 22
Correct constituent number: 9
####################
Test Tree: 8
Constituent number in the predicted tree: 18
Constituent number in the gold tree: 18
Correct constituent number: 10
####################
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
```

[31]:

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

```
Overall precision: 0.551, recall: 0.551, f1: 0.551
```

[32]:

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

```
Overall precision: 0.551, recall: 0.551, f1: 0.551
```

[34]:

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

719.4037554264069

[ ]:

```python

```