50.040 Natural Language Processing (Summer 2020) Homework 2

Due

STUDNET ID: 1002934

Name: chloe zheng

Students with whom you have discussed (if any): yi xuan

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\chloe\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', BracketParseCorpusRea
```

Processing math: 100%

```
der, 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
```

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

In [7]:
```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 -> '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-T
MP> -> 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 -> 'dir
ector', NP-TMP -> NNP CD, NNP -> 'nov.', CD -> '29', . -> '.'] <class 'nlt
k.grammar.Production'>
```

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

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

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

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

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

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

In [11]:
```python
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 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(): Return the right-hand side of this Production.
- lhs(): Return the left-hand side of this Production.
- leaves(): Return the leaves of the tree.

Question 2

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

In [12]:
```python
from nltk import Tree
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.
    '''
    # instantiate a list
    rules = list()
    rules_counts = Counter()
    ### YOUR CODE HERE (~ 2 lines)

    for tree in train_data:
        for subtree in tree.productions():
            rules.append(subtree)

    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
    '''
    nonterminals = list()
    ### YOUR CODE HERE (at least one line)
#   print(rules)
    for r in rules:
        for sym in list(r.rhs()) + [r.lhs()]:
            if type(sym)== nltk.Nonterminal:
                nonterminals.append(sym)

    ### 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)

    # terminal, rhs == str
    for r in rules:
#       print(type(r.rhs())) -> tuple
```

```
        for sym in list(r.rhs()):
            if type(sym) == str:
                terminals.append(sym)

    ### END OF YOUR CODE

    return set(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, 32
73), (S|<VP-.> -> VP ., 3003)]
```

Question 3

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

In [16]:
```
def build_pcfg(rules_counts):
    '''
    Build a dictionary that stores the terminal rules and nonterminal rule
s.
    param:
        rules_counts: Counter object --- a dictionary that maps one rule t
o 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 a
nd 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 st
rings.
    '''

    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
```

```
    # rules: A -> BC (nonterminal), A ->  α (terminal, str)

    # get key value pair from rules_counts
    for k, v in rules_counts.items():
        if type(k.rhs()[0]) == str:
            sym = 'terminals'
        else:
            # if not str
            sym = 'nonterminals'

        if sym == 'nonterminals':
            # iterate through B and C
            r_s = list(str(r) for r in k.rhs())
            rhs = '@'.join(r_s)

        else:
            # terminal α, get strings on rhs
            rhs = k.rhs()[0]

        # key of inner dict
        lhs = str(k.lhs())

        # build inner dict to insert into rules_dict
        # insert inner dict with lhs as key into rules_dict
        if lhs not in rules_dict[sym].keys():
            rules_dict[sym][lhs]={}

        if rhs not in rules_dict[sym][lhs].keys():
            rules_dict[sym][lhs][rhs] = v

        else:
            # if rhs is inside, + count to val
            rules_dict[sym][lhs][rhs] += v

    ### END OF YOUR CODE
    return rules_dict
```

In [17]:

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

Question 4

Estimate the probability of rule NP → NNP@NNP

In [18]:

```
# get value of NP from terminals
term_val = sum(train_rules_dict['terminals']['NP'].values())

# get value of NP from nonterminals
nonterm_val = sum(train_rules_dict['nonterminals']['NP'].values())
denom = term_val + nonterm_val

# get value of NP -> NNP@NNP
num = train_rules_dict['nonterminals']['NP']['NNP@NNP']
```

```
prob = num/denom

print(prob)
```

0.03950843529348353

Question 5

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

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

# terminal symbols dict{'terminals': {'NP':100, 'ADJ': 12....}}
# cnf_test - train => get dict value
storesymbols=[]
storeleaves=cnf_test[0].leaves()

for out in train_rules_dict['terminals'].values():
    for inner in out.keys():
        storesymbols.append(inner)

set(storeleaves) - set(storesymbols)
```

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

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 ∈ N, we add one new rule A → unk to the original PCFG.

The smoothed probabilities for all rules can then be estimated as:
$q_{smooth}(A \rightarrow \beta)$ = count(A → β)count(A) + 1 $q_{smooth}(A \rightarrow unk)$ = 1count(A) + 1 where |V| is the count of unique terminal symbols.

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

In [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':5
00}, '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>':0.0
1}}
                                                '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'].key
s()
    ### YOUR CODE HERE

    # to get 'NP', 'ADJ'... from term
    term_keys = rules_counts['terminals'].keys()

    # to get 'S', 'NP'.. from nonterm
    nonterm_keys = rules_counts['nonterminals'].keys()

    for l in term_keys | nonterm_keys:

        term_lhs_val = rules_counts['terminals'][l].values()
        nonterm_lhs_val = rules_counts['nonterminals'][l].values()

        # count(A) + 1, getting all val produced from A
        total = sum(term_lhs_val) + sum(nonterm_lhs_val) + 1

        # cal smoothing
        for r, sym in rules_counts['terminals'][l].items():
            rules_prob['terminals'][l][r] = sym / total
        for r, sym in rules_counts['nonterminals'][l].items():
            rules_prob['nonterminals'][l][r] = sym / total

        # qsmooth (a -> unk)
        rules_prob['terminals'][l][unk] = 1 / total


    ### END OF YOUR CODE
    return rules_prob
```

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

In [22]:
```
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
0.03950630591786l376
{'<unk>': 5.389673385792821e-05}
```

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

Out[23]:
```
11368
```

CKY Algorithm

Similar to the Viterbi algorithm, the CKY algorithm is a dynamic-programming algorithm. Given a PCFG G = (N, Σ, 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 $w_0, w_1, \ldots, w_{n-1}$, π(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 $0 \le i < k \le n$. 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 [24]:
```
from itertools import chain, product
from math import log
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 s
core 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, rig
ht_child); split: int;
                        left_child: str; right_child: str.
    '''
    score = defaultdict(dict)
```

```
    back = defaultdict(dict)
    sent_len = len(sent)
    ### YOUR CODE HERE
   ##########################WIP#######################
    rules_key = rules_prob['terminals'].keys()

    # for i = 0, .., len - 1
    for begin in range(sent_len):

        # for A in nonterm
        for root in rules_key:

            word = str(sent[begin])
            if word not in terminals:
                word = '<unk>'

            term_root = rules_prob['terminals'][root]
            if sent[begin] in term_root:
                log_term_root = log(rules_prob['terminals'][root][sent[beg
in]])
                score[(begin, begin+1)][root] = log_term_root
                back[(begin, begin+1)][root] = (begin,word,word)

        # for span = 2,.., len
        for span in range(2, sent_len+1):

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

                nonterm_items = rules_prob['nonterminals'].items()


                for root, rhs_dict in nonterm_items:
                   # start from -ve inf
                    max_score = -float('inf')
                    max_prev = (-1, None, None)


                    for split, (rhs, rule_prob) in product(range(begin+1,
end), rhs_dict.items()):
                        left, right = rhs.split('@')
                        if left in score[(begin, split)] and right in scor
e[(split, end)]:

                            split_left = score[(begin, split)][left]
                            split_right = score[(split, end)][right]
                            cur_score =  split_left + split_right + log(ru
le_prob)

                            if cur_score > max_score:
                                max_score = cur_score
                                max_prev = (split, left, right)

                    score[(begin, end)][root] = max_score
                    back[(begin, end)][root] = max_prev
```

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

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

In [26]:
```
score[(0, len(sent))]['S']
```

Out[26]:
```
-117.52227496068694
```

Question 8

Implement **build_tree** function according to algorithm 2 to reconstruct theparse 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); 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
    if end - begin != 1:
        # non-terminal
        split, left_label, right_label = back[(begin, end)][root_label]
        left_child = build_tree(back, (begin, split, left_label))
        right_child = build_tree(back, (split, end, right_label))
        tree = nltk.tree.Tree(root_label, [left_child, right_child])

    else:
        # terminal
        tree = nltk.tree.Tree(root_label, [back[(begin, end)][root_label][
1]])


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

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

Question 9

In [29]:
```
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 [30]:
```
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: 26
####################
```

```
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 [31]:
```
print('Overall precision: {:.3f}, recall: {:.3f}, f1: {:.3f}'.format(preci
sion, recall, f1))
```

```
Overall precision: 0.532, recall: 0.532, f1: 0.532
```

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

```
Overall precision: 0.532, recall: 0.532, f1: 0.532
```

In [33]:
```
et=time.time()
print(et - st)
```

```
812.0147874355316
```