Grammars_Solution

May 3, 2022

1 Syntax and Grammars

1.1 1. Classical CFG

You might know context free grammars (CFG) from theoretical comouter sciebce, formal languages or compiler construction. Origiannly CFGs were developed to describe natural languages.

On the one hand side CFG is a generative model. Starting with a start symbol we can meake context free derivations and thus count all grammatical words (or sentences) of a language (a possible infinite set of words/strings). On the other hand side, for CFGs efficient ($O(n^{2.7})$) parsing algorithms (like Earley, Cock-Kasamy-Younger) exist that can produce parse trees, representing an analysis of a given input.

Stanfords NLTK provides a CFG parser taht easily can be used.

[ ]:
```
import nltk

grammar = nltk.CFG.fromstring("""
S -> NP VP
PP -> P NP
NP -> Det N | Det N PP | 'I'
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
""")
```

NLTK offers a number of different parser generators. See http://www.nltk.org/book/ch08.html for more options.

[ ]:
```
parser = nltk.ChartParser(grammar)
testlist = []
sent = nltk.word_tokenize('I shot an elephant in my pajamas')
# print(sent)

parse_trees = list(parser.parse(sent))
# print(parse_trees)
for tree in parse_trees:
    print(tree)
```

```
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant) (PP (P in) (NP (Det my) (N pajamas))))))
```

If Ghostscript is installed, you can draw beautifull trees:

[ ]:
```
parse_trees[0].draw()
```

[ ]:
```
parse_trees[1].draw()
```

An example with recursion

[ ]:
```
grammar2 = nltk.CFG.fromstring("""
  S  -> NP VP
  NP -> Det Nom | PropN
  Nom -> Adj Nom | N
  VP -> V Adj | V NP | V S | V NP PP
  PP -> P NP
  PropN -> 'Buster' | 'Chatterer' | 'Joe'
  Det -> 'the' | 'a'
  N -> 'bear' | 'squirrel' | 'tree' | 'fish' | 'log'
  Adj  -> 'angry' | 'frightened' |  'little' | 'tall'
  V ->  'chased'  | 'saw' | 'said' | 'thought' | 'was' | 'put'
  P -> 'on'
  """)
```

[ ]:
```
parser = nltk.ChartParser(grammar2)

sent1 = nltk.word_tokenize('the angry bear chased the frightened little␣
 ↪squirrel')
sent2 = nltk.word_tokenize('Chatterer said Buster thought the tree was tall')
print(type(parser.parse(sent1)))
parse_trees = list(parser.parse(sent1))
# parse_trees1 = list(parser.parse(sent1))
# parse_trees2 = list(parser.parse(sent2))
# parse_trees = parse_trees1 + parse_trees2
for tree in parse_trees:
    print(tree)
    # tree.draw() # if you have ghostscript installed
```

```
<class 'generator'>
(S
```

```
  (NP (Det the) (Nom (Adj angry) (Nom (N bear))))
  (VP
    (V chased)
    (NP
      (Det the)
      (Nom (Adj frightened) (Nom (Adj little) (Nom (N squirrel)))))))
```

We also can generate sentences, of course. This is not intended to be a usefull mechanism for language generation, but it helps to check to see whether you grammar is complete and not too much overgenerating.

[ ]:
```python
from nltk.parse.generate import generate

for sentence in generate(grammar2, depth=4):
    print(' '.join(sentence))
```

```
Buster chased angry
Buster chased frightened
Buster chased little
Buster chased tall
Buster saw angry
Buster saw frightened
Buster saw little
Buster saw tall
Buster said angry
Buster said frightened
Buster said little
Buster said tall
Buster thought angry
Buster thought frightened
Buster thought little
Buster thought tall
Buster was angry
Buster was frightened
Buster was little
Buster was tall
Buster put angry
Buster put frightened
Buster put little
Buster put tall
Chatterer chased angry
Chatterer chased frightened
Chatterer chased little
Chatterer chased tall
Chatterer saw angry
Chatterer saw frightened
Chatterer saw little
Chatterer saw tall
```

```
Chatterer said angry
Chatterer said frightened
Chatterer said little
Chatterer said tall
Chatterer thought angry
Chatterer thought frightened
Chatterer thought little
Chatterer thought tall
Chatterer was angry
Chatterer was frightened
Chatterer was little
Chatterer was tall
Chatterer put angry
Chatterer put frightened
Chatterer put little
Chatterer put tall
Joe chased angry
Joe chased frightened
Joe chased little
Joe chased tall
Joe saw angry
Joe saw frightened
Joe saw little
Joe saw tall
Joe said angry
Joe said frightened
Joe said little
Joe said tall
Joe thought angry
Joe thought frightened
Joe thought little
Joe thought tall
Joe was angry
Joe was frightened
Joe was little
Joe was tall
Joe put angry
Joe put frightened
Joe put little
Joe put tall
```

1.2 Feature Grammars

Theoretically it can be discussed whether a CFG for natural languages can be written or not. Using the simple classical notation that we also used above this is however practicaly unfeasible. E.g. yo need to distinguis between singular and plural nouns, because we can combine singular nouns with *this* and plural nouns with *these*. Thsu we would have to duplicate all rules in which a Noun occurs.

A more practical way to handle this is represented by feature grammars. Feature grammars (as presenetd below) are still (weakly) equivalent to CFGs. That means that we could automatically expand all the rules of the grmmar to obtain a CFG generating exactly the same sentences.

In a feature grammar the symbols are extended wit a list of features and values. Values iof features also can be feature lists (we should not allow however recursive structures here as that would mean that we would leave the context free world very soon!). I a rule, we cal also use variables for the values. Using the variables we can e.g. enforce that the value of a feature on two symbols has to be the same, e.g. singular in both cases or plural in both cases. Filling in the values for the variables now becomes a central issue in parsing. This process of filling in the variables is called *unification*.

Let us consider some simple examples:

[ ]:
```
from nltk import grammar, parse

g = """
% start DP
DP[AGR=?a] -> D[AGR=?a] N[AGR=?a]
D[AGR=[NUM=sg, PERS=3]] -> 'this' | 'that'
D[AGR=[NUM=pl, PERS=3]] -> 'these' | 'those'
D[AGR=[NUM='pl', PERS=1]] -> 'we'
D[AGR=[PERS=2]] -> 'you'
N[AGR=[NUM=sg, GND='m']] -> 'boy'
N[AGR=[NUM='pl', GND='m']] -> 'boys'
N[AGR=[NUM='sg', GND='f']] -> 'girl'
N[AGR=[NUM='pl', GND='f']] -> 'girls'
N[AGR=[NUM='sg']] -> 'student'
N[AGR=[NUM='pl']] -> 'students'
"""

grammar = grammar.FeatureGrammar.fromstring(g)
tokens = 'these girls'.split()
parser = parse.FeatureEarleyChartParser(grammar)
trees = parser.parse(tokens)
for tree in trees:
    tree.draw()
    print(tree)
```

```
(DP[AGR=[GND='f', NUM='pl', PERS=3]]
  (D[AGR=[NUM='pl', PERS=3]] these)
  (N[AGR=[GND='f', NUM='pl']] girls))
```

[ ]:
```
tokens = 'these girl'.split()
trees = parser.parse(tokens)
for tree in trees:
    tree.draw()
    print(tree)
```

1.3 Chunking

Even a feature grammar is very hard to write. Moreover, parsing with CFGs is very problematic if we have non-ideal input (typos, tokenization errors, ungrammatical sentences) or if the grammar does not cover everything. For many purposes we do not need the complete parse trees. Here chunking grammars can provide an alternative. Like regular expressions, a cuncking grammar describes patterns that are searched in the string. NLTH offers a ckunking grammar with the following properties:

- Sentence is not requires to be covered by one tree. The algorithm will find partial structures ('chunks') in the sentence.
- No recursion possible
- Depth of parse trees is delimited
- Right hand side of a rule is a regular expression
- Also regex can be used for the names of the grammar symbols
- The parser requires a list of pairs (word,POS) as input

1.3.1 POS Tagging

First we have to find the contextual correct POS for each word. For English we can use the POS tagges from NLTK. Note that there are many POS tagges availabel on the internet. There are considerable differences in the quality of POS Taggers. So inform and test before using a POS Tagger!

[ ]:
```
# sent1 = nltk.word_tokenize('the angry bear chased the frightened little␣
↪squirrel')
tagged_tokens = nltk.pos_tag(sent1)
# print(type(sent1))
print(tagged_tokens)
```

```
[('the', 'DT'), ('angry', 'JJ'), ('bear', 'NN'), ('chased', 'VBD'), ('the',
'DT'), ('frightened', 'JJ'), ('little', 'JJ'), ('squirrel', 'NN')]
```

[ ]:
```
grammar = r"""
   NP: {<DT>?(<JJ>)*<NN.*>}
"""


cp = nltk.RegexpParser(grammar)
```

[ ]:
```
tree = cp.parse(tagged_tokens)
print(tree)
tree.draw()
```

```
(S
  (NP the/DT angry/JJ bear/NN)
  chased/VBD
  (NP the/DT frightened/JJ little/JJ squirrel/NN))
```

Or we extract just the NPs we defined:

```
[ ]: for node in tree:
         if isinstance(node, nltk.tree.Tree):
             if node.label() == 'NP':
                 NP =  node.leaves()
                 print(NP)
```

```
[('the', 'DT'), ('angry', 'JJ'), ('bear', 'NN')]
[('the', 'DT'), ('frightened', 'JJ'), ('little', 'JJ'), ('squirrel', 'NN')]
```

1.3.2 Extension - Limitations

```
[ ]: grammar = r"""
        NP: {<DT|PRP.*>?(<JJ>)*<NN.*><PP>?}
        {<PRP>}
        PP: {<IN><NP>}
     """

     cp = nltk.RegexpParser(grammar)
```

```
[ ]: tagged_tokens = nltk.pos_tag(sent)
     #tagged_tokens = [(w,t.strip('$')) for (w,t) in tagged_tokens]
     print(tagged_tokens)
     tree = cp.parse(tagged_tokens)
     print(tree)
     tree.draw()
```

```
[('I', 'PRP'), ('shot', 'VBP'), ('an', 'DT'), ('elephant', 'NN'), ('in', 'IN'),
('my', 'PRP$'), ('pajamas', 'NN')]
(S
  (NP I/PRP)
  shot/VBP
  (NP an/DT elephant/NN)
  (PP in/IN (NP my/PRP$ pajamas/NN)))
```

1.4 Learn Chunking

writing grammars still is not easy. Chunking also can easily be learned from annotated data. To do so we hav to reformulate the chunkin problem a little bit. Since we do not allow recursion, we could simply annotate each word with the phrase it belongs to. Since e.g. two NPs might be adjacent, we also need to mark the first word of a phrase. Thus for each phrase type we have two tags. Typically B-tag (begin) and I-tag (inside) for the subsequent words of the phrase. So, if we want to find NPs we have B-NP and I-NP. For the remaining words we use O (outside). This tagging scheme is called IOB-tagging.

Of course we need annotated data. A well-known corpus of texts with IOB tags is the CONLL corpus of texts from the Wall Street Journal. This corpus can be downloaded here: http://www.cnts.ua.ac.be/conll2000/chunking/

In Python we can easily use many different classifiers. We now only take very simple classifiers that use only one feature per word. As input we now have a list of characteristics. The result is a list of IOB tags. As training data, we now need lists of pairs consisting of a feature for a word and the IOB tag that goes with it. We write a function that reads the corpus and returns such lists. As features we take the parts of speech that are also present in the corpus.

[ ]:
```
def read_conll(filepath):
    result = []
    file = open(filepath)
    sentence = []
    for line in file:
        line = line.strip('\n')
        if not line.strip(' '):
            result.append(sentence)
            sentence = []
            continue
        (word,pos,tag) = line.split(' ')
        sentence.append((pos,tag))
    return result


conll_train = read_conll('train.txt')
conll_test = read_conll('test.txt')

print(conll_train[0])
```

```
[('NN', 'B-NP'), ('IN', 'B-PP'), ('DT', 'B-NP'), ('NN', 'I-NP'), ('VBZ',
'B-VP'), ('RB', 'I-VP'), ('VBN', 'I-VP'), ('TO', 'I-VP'), ('VB', 'I-VP'), ('DT',
'B-NP'), ('JJ', 'I-NP'), ('NN', 'I-NP'), ('IN', 'B-SBAR'), ('NN', 'B-NP'),
('NNS', 'I-NP'), ('IN', 'B-PP'), ('NNP', 'B-NP'), (',', 'O'), ('JJ', 'B-ADJP'),
('IN', 'B-PP'), ('NN', 'B-NP'), ('NN', 'B-NP'), (',', 'O'), ('VB', 'B-VP'),
('TO', 'I-VP'), ('VB', 'I-VP'), ('DT', 'B-NP'), ('JJ', 'I-NP'), ('NN', 'I-NP'),
('IN', 'B-PP'), ('NNP', 'B-NP'), ('CC', 'I-NP'), ('NNP', 'I-NP'), ('POS',
'B-NP'), ('JJ', 'I-NP'), ('NNS', 'I-NP'), ('.', 'O')]
```

The simplest tagger just takes the most likely tag for each POS. We call this tagger a unigram tagger, because it bases its decision always on a sequence of 1 tokens.

[ ]:
```
import nltk
import nltk.tag

u_chunker = nltk.tag.UnigramTagger(conll_train)
```

[ ]:
```
from pprint import pprint

sent = u'Less than three days before President Obama turns the keys to the␣
 ↪White House, and the nuclear codes, over to President-elect Donald J. Trump,␣
 ↪Mr. Trump’s transition staff has barely engaged with the National Security␣
 ↪Council below the most senior levels.'
```

```python
tokenized_sent = nltk.tokenize.word_tokenize(sent)
pos_tagged_sent = nltk.pos_tag(tokenized_sent)
#print(pos_tagged_sent)
pos_tags = [t for (w,t) in pos_tagged_sent]
#print(pos_tags)
chunks = u_chunker.tag(pos_tags)
chunked_sent = [(w,p,t) for (w,(p,t)) in zip(tokenized_sent,chunks)]
pprint(chunked_sent)
```

```
[('Less', 'JJR', 'B-NP'),
 ('than', 'IN', 'B-PP'),
 ('three', 'CD', 'I-NP'),
 ('days', 'NNS', 'I-NP'),
 ('before', 'IN', 'B-PP'),
 ('President', 'NNP', 'I-NP'),
 ('Obama', 'NNP', 'I-NP'),
 ('turns', 'VBZ', 'B-VP'),
 ('the', 'DT', 'B-NP'),
 ('keys', 'NNS', 'I-NP'),
 ('to', 'TO', 'B-PP'),
 ('the', 'DT', 'B-NP'),
 ('White', 'NNP', 'I-NP'),
 ('House', 'NNP', 'I-NP'),
 (',', ',', 'O'),
 ('and', 'CC', 'O'),
 ('the', 'DT', 'B-NP'),
 ('nuclear', 'JJ', 'I-NP'),
 ('codes', 'NNS', 'I-NP'),
 (',', ',', 'O'),
 ('over', 'IN', 'B-PP'),
 ('to', 'TO', 'B-PP'),
 ('President-elect', 'JJ', 'I-NP'),
 ('Donald', 'NNP', 'I-NP'),
 ('J.', 'NNP', 'I-NP'),
 ('Trump', 'NNP', 'I-NP'),
 (',', ',', 'O'),
 ('Mr.', 'NNP', 'I-NP'),
 ('Trump', 'NNP', 'I-NP'),
 ('’', 'NNP', 'I-NP'),
 ('s', 'POS', 'B-NP'),
 ('transition', 'NN', 'I-NP'),
 ('staff', 'NN', 'I-NP'),
 ('has', 'VBZ', 'B-VP'),
 ('barely', 'RB', 'B-ADVP'),
 ('engaged', 'VBN', 'I-VP'),
 ('with', 'IN', 'B-PP'),
```

```
('the', 'DT', 'B-NP'),
('National', 'NNP', 'I-NP'),
('Security', 'NNP', 'I-NP'),
('Council', 'NNP', 'I-NP'),
('below', 'IN', 'B-PP'),
('the', 'DT', 'B-NP'),
('most', 'RBS', 'B-NP'),
('senior', 'JJ', 'I-NP'),
('levels', 'NNS', 'I-NP'),
('.', '.', 'O')]
```

The result is as expected: not particularly useful. Nevertheless, many tags are correct. NLTK provides a function to evaluate the chunker with the sentences from the test corpus:

```
[ ]: u_chunker.evaluate(conll_test)
```

```
/tmp/ipykernel_23844/49953250.py:1: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  u_chunker.evaluate(conll_test)
```

```
[ ]: 0.7729066846782194
```

The main problem is that the sequence of I and B tags is hopeless. The BigramTagger also considers the predicted tag of the previous word as a feature. So the probabilities of sequences of two words, so-called bigrams, are optimized.

```
[ ]: b_chunker = nltk.tag.BigramTagger(conll_train)
     b_chunker.evaluate(conll_test)
```

```
/tmp/ipykernel_23844/1041831056.py:2: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  b_chunker.evaluate(conll_test)
```

```
[ ]: 0.8869915781919496
```

With a little trick we can improve the results a bit: if the bigram chunker cannot decide between two possibilities, the unigram chunker should decide:

```
[ ]: bu_chunker = nltk.tag.BigramTagger(conll_train, backoff=u_chunker)
     bu_chunker.evaluate(conll_test)
```

```
/tmp/ipykernel_23844/2110052980.py:2: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  bu_chunker.evaluate(conll_test)
```

```
[ ]: 0.8905164953458429
```

The bigram tagger optimizes the probabilities only locally. A Hidden Markov Model optimizes the probability of the complete sequence of tags.

[ ]:
```
hmm_trainer = nltk.tag.HiddenMarkovModelTrainer()
hmm_model = hmm_trainer.train_supervised(conll_train)
```

[ ]:
```
hmm_model.evaluate(conll_test)
```

```
/tmp/ipykernel_23844/1605633779.py:1: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  hmm_model.evaluate(conll_test)
```

[ ]: 0.9050383097283492

Results can be improved a little bit by using Conditional Random Fields insteatd of a HMM:

[ ]:
```
ct = nltk.tag.CRFTagger()
ct.train(conll_train,'crf.model')
```

[ ]:
```
#ct.set_model_file('crf.model')
ct.evaluate(conll_test)
```

Let us inspect, what the result for our test sentence now looks like

[ ]:
```
chunks = bu_chunker.tag(pos_tags)
chunked_sent = [(w,p,t) for (w,(p,t)) in zip(tokenized_sent,chunks)]
pprint(chunked_sent)
```

```
[('Less', 'JJR', 'B-NP'),
 ('than', 'IN', 'B-PP'),
 ('three', 'CD', 'B-NP'),
 ('days', 'NNS', 'I-NP'),
 ('before', 'IN', 'B-PP'),
 ('President', 'NNP', 'B-NP'),
 ('Obama', 'NNP', 'I-NP'),
 ('turns', 'VBZ', 'B-VP'),
 ('the', 'DT', 'B-NP'),
 ('keys', 'NNS', 'I-NP'),
 ('to', 'TO', 'B-VP'),
 ('the', 'DT', 'B-NP'),
 ('White', 'NNP', 'I-NP'),
 ('House', 'NNP', 'I-NP'),
 (',', ',', 'O'),
 ('and', 'CC', 'O'),
 ('the', 'DT', 'B-NP'),
 ('nuclear', 'JJ', 'I-NP'),
 ('codes', 'NNS', 'I-NP'),
 (',', ',', 'O'),
```

```
('over', 'IN', 'B-PP'),
('to', 'TO', 'B-PP'),
('President-elect', 'JJ', 'B-NP'),
('Donald', 'NNP', 'I-NP'),
('J.', 'NNP', 'I-NP'),
('Trump', 'NNP', 'I-NP'),
(',', ',', 'O'),
('Mr.', 'NNP', 'B-NP'),
('Trump', 'NNP', 'I-NP'),
('’', 'NNP', 'I-NP'),
('s', 'POS', 'B-NP'),
('transition', 'NN', 'I-NP'),
('staff', 'NN', 'I-NP'),
('has', 'VBZ', 'B-VP'),
('barely', 'RB', 'I-VP'),
('engaged', 'VBN', 'I-VP'),
('with', 'IN', 'B-PP'),
('the', 'DT', 'B-NP'),
('National', 'NNP', 'I-NP'),
('Security', 'NNP', 'I-NP'),
('Council', 'NNP', 'I-NP'),
('below', 'IN', 'B-PP'),
('the', 'DT', 'B-NP'),
('most', 'RBS', 'I-NP'),
('senior', 'JJ', 'I-NP'),
('levels', 'NNS', 'I-NP'),
('.', '.', 'O')]
```

Finally, note that NLTK also provides a function to build the tree structure as used above from the IOB format:

[ ]:
```
tree = nltk.chunk.conlltags2tree(chunked_sent)
#tree.draw()
print(tree)
```

```
(S
  (NP Less/JJR)
  (PP than/IN)
  (NP three/CD days/NNS)
  (PP before/IN)
  (NP President/NNP Obama/NNP)
  (VP turns/VBZ)
  (NP the/DT keys/NNS)
  (VP to/TO)
  (NP the/DT White/NNP House/NNP)
  ,/,
  and/CC
  (NP the/DT nuclear/JJ codes/NNS)
```

```
,/,
(PP over/IN)
(PP to/TO)
(NP President-elect/JJ Donald/NNP J./NNP Trump/NNP)
,/,
(NP Mr./NNP Trump/NNP ’/NNP)
(NP s/POS transition/NN staff/NN)
(VP has/VBZ barely/RB engaged/VBN)
(PP with/IN)
(NP the/DT National/NNP Security/NNP Council/NNP)
(PP below/IN)
(NP the/DT most/RBS senior/JJ levels/NNS)
./.)
```

The other way around is also possible (and practical for the excercise!).

[ ]:
```
nltk.chunk.tree2conlltags(tree)
```

[ ]:
```
[('Less', 'JJR', 'B-NP'),
 ('than', 'IN', 'B-PP'),
 ('three', 'CD', 'B-NP'),
 ('days', 'NNS', 'I-NP'),
 ('before', 'IN', 'B-PP'),
 ('President', 'NNP', 'B-NP'),
 ('Obama', 'NNP', 'I-NP'),
 ('turns', 'VBZ', 'B-VP'),
 ('the', 'DT', 'B-NP'),
 ('keys', 'NNS', 'I-NP'),
 ('to', 'TO', 'B-VP'),
 ('the', 'DT', 'B-NP'),
 ('White', 'NNP', 'I-NP'),
 ('House', 'NNP', 'I-NP'),
 (',', ',', 'O'),
 ('and', 'CC', 'O'),
 ('the', 'DT', 'B-NP'),
 ('nuclear', 'JJ', 'I-NP'),
 ('codes', 'NNS', 'I-NP'),
 (',', ',', 'O'),
 ('over', 'IN', 'B-PP'),
 ('to', 'TO', 'B-PP'),
 ('President-elect', 'JJ', 'B-NP'),
 ('Donald', 'NNP', 'I-NP'),
 ('J.', 'NNP', 'I-NP'),
 ('Trump', 'NNP', 'I-NP'),
 (',', ',', 'O'),
 ('Mr.', 'NNP', 'B-NP'),
 ('Trump', 'NNP', 'I-NP'),
 ('’', 'NNP', 'I-NP'),
```

```
('s', 'POS', 'B-NP'),
('transition', 'NN', 'I-NP'),
('staff', 'NN', 'I-NP'),
('has', 'VBZ', 'B-VP'),
('barely', 'RB', 'I-VP'),
('engaged', 'VBN', 'I-VP'),
('with', 'IN', 'B-PP'),
('the', 'DT', 'B-NP'),
('National', 'NNP', 'I-NP'),
('Security', 'NNP', 'I-NP'),
('Council', 'NNP', 'I-NP'),
('below', 'IN', 'B-PP'),
('the', 'DT', 'B-NP'),
('most', 'RBS', 'I-NP'),
('senior', 'JJ', 'I-NP'),
('levels', 'NNS', 'I-NP'),
('.', '.', 'O')]
```

2 Exercise

1. Take the CONLL-Train data and extract from each sentence all chunks and the sequence of tags that make up that chunk.
2. For each type of chunk, count how often each pattern occurs.
3. Use these patterns to construct a regex chunking grammar by hand.
4. Evaluate your grammar on the CONLL test data.

Tip: start with just a very small number of simple rules and test your grammar. Then continue adding more rules and making rules more complex. Always check, whether addition improve your grammar.

[ ]:
```
with open('train.txt','r') as f:  # open file for reading purpose
    conlltext = f.read()
traintree = nltk.chunk.conllstr2tree(conlltext, chunk_types=['NP', 'VP', 'PP'])␣
 ↪ # read the file and convert it to tree

# Below is the commented code for the counting the VP, NP, PP chunks, there is␣
 ↪another way of doing the same thing in the next section
# vals = ['VP', 'NP', 'PP'] # list of chunk types
# valsCount = [0,0,0] # list of chunk types count

# for stuff in range(len(vals)):  # loop through the list
#     for subtree in traintree.subtrees():  # loop through the tree
#         if subtree.label() == vals[stuff]: # if the chunk type matches the␣
 ↪chunk type in the list
#             valsCount[stuff] = valsCount[stuff] + 1   # increment the count

# print(vals, ' --> counts are: ', valsCount)
```

[ ]:
```python
NP = [] # list of NP chunks
VP = [] # list of VP chunks
PP = [] # list of PP chunks
valsCount = [0,0,0] # list of chunk types count
for node in traintree: # loop through the tree
    # print(node)
    if isinstance(node, nltk.tree.Tree):
        if node.label() == 'NP':
            NP.append(node.leaves()) # append the NP chunk to the list
            # print(NP)
            valsCount[0] = valsCount[0] + 1
        if node.label() == 'VP':
            VP.append(node.leaves()) # append the VP chunk to the list
            # print(VP)
            valsCount[1] = valsCount[1] + 1
        if node.label() == 'PP':
            PP.append(node.leaves()) # append the PP chunk to the list
            # print(PP)
            valsCount[2] = valsCount[2] + 1
print(' NP, VP, PP', ' --> counts are: ', valsCount) # print the counts of the␣
↪chunks
```

```
NP, VP, PP  --> counts are:  [55081, 21467, 21281]
```

[ ]:
```python
import re
# Defining grammar and searching in the flattened tree, it has been converted␣
↪to a string and findall function is used to location grammar chunks
grammar = ['(DT)\s(NN)', '(DT)\s(JJ)\s(NN)', 'IN', 'TO', '((VB\w?)\s)+', '(To)?
↪(\s(VB\w?))+', '(TO)?\s(DT)?', '(RB)\s(VB\w?)?', '(VB\w?)?\s(NN)?', '((VB\w?
↪)|(MD))?\s(VB\w?)?' ] # list of grammar chunks
chunksDict = {'NP': NP, 'VP': VP, 'PP': PP}  # dictionary of chunks

for key, value in chunksDict.items(): # loop through the dictionary
    print('Computation for ', key) # print the key
    flat_list = [item for sublist in value for item in sublist] # flatten the␣
↪list
    newlist = [] # list that will contain only the tags of chunks
    for bek in flat_list: # loop through the flattened list
        newlist.append(bek[1]) # append the tags to the new list
    bigString = ' '.join([str(item) for item in newlist]) # convert the list to␣
↪a string
    for subgram in grammar: # loop through the grammar list
        res = re.findall(subgram,bigString) # search for the grammar chunks in␣
↪the string
        print('Occurrence of',subgram,'is', len(res)) # print the count of the␣
↪grammar chunks
    print()
```

```
Computation for  NP
Occurrence of (DT)\s(NN) is 12603
Occurrence of (DT)\s(JJ)\s(NN) is 3105
Occurrence of IN is 634
Occurrence of TO is 90
Occurrence of ((VB\w?)\s)+ is 1574
Occurrence of (To)?(\s(VB\w?))+ is 1574
Occurrence of (TO)?\s(DT)? is 118387
Occurrence of (RB)\s(VB\w?)? is 947
Occurrence of (VB\w?)?\s(NN)? is 118387
Occurrence of ((VB\w?)|(MD))?\s(VB\w?)? is 118387

Computation for  VP
Occurrence of (DT)\s(NN) is 1
Occurrence of (DT)\s(JJ)\s(NN) is 0
Occurrence of IN is 23
Occurrence of TO is 2932
Occurrence of ((VB\w?)\s)+ is 6532
Occurrence of (To)?(\s(VB\w?))+ is 6531
Occurrence of (TO)?\s(DT)? is 33469
Occurrence of (RB)\s(VB\w?)? is 1436
Occurrence of (VB\w?)?\s(NN)? is 33469
Occurrence of ((VB\w?)|(MD))?\s(VB\w?)? is 33469

Computation for  PP
Occurrence of (DT)\s(NN) is 0
Occurrence of (DT)\s(JJ)\s(NN) is 0
Occurrence of IN is 18933
Occurrence of TO is 2018
Occurrence of ((VB\w?)\s)+ is 295
Occurrence of (To)?(\s(VB\w?))+ is 295
Occurrence of (TO)?\s(DT)? is 21571
Occurrence of (RB)\s(VB\w?)? is 81
Occurrence of (VB\w?)?\s(NN)? is 21571
Occurrence of ((VB\w?)|(MD))?\s(VB\w?)? is 21571
```

[ ]:
```
from nltk.corpus import conll2000

# I could not find a way to get my grammar working for the evaluation, so I␣
 ↪used some dummy grammar to get the results.

grammar = r"""
  NP: {<DT|PP\$>?<JJ>*<NN>}
  NP: {<NNP>+}
  VP: {<VB.*>+<RB>?<VB.*>+}
  VP: {<VB.*>+<RB>?<VB.*>+}
```

```
  NP: {<DT|PRP.>?(<JJ>)<NN.*><PP>?}
  {<PRP>}
  PP: {<IN|TO>}
  VP: {<VB.*><NP|PP>}
  PP: {<IN><NP>}
  VP: {<VB.*><NP|PP>}
  VP: {<MD><VP>}
"""

cp = nltk.RegexpParser(grammar) # create a parser
test_data = conll2000.chunked_sents('test.txt', chunk_types=['NP','VP','PP']) #
 ↪read the test data
print(cp.evaluate(test_data)) # evaluate the test data
```

```
/tmp/ipykernel_23844/1977737851.py:21: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(cp.evaluate(test_data))
```

```
ChunkParse score:
    IOB Accuracy:  58.8%%
    Precision:     51.5%%
    Recall:        43.2%%
    F-Measure:     47.0%%
```