CSCI-8450

2018 Spring

Yuchen Wang

You will find Chapter 7 posted at:

<http://www.nltk.org/book/ch07.html>

1. The IOB format categorizes tagged tokens as I, O and B. Why are three tags necessary? What problem would be caused if we used I and O tags exclusively?

Provide a sentence to illustrate your point.

Result:

Exercise 1

It's because IOB tags have become the standard way to represent chunk structures in files.

In the IOB representation, there is one token per line, each with its part-of-speech tag and its chunk tag. The IOB format permits us to represent more than one chunk type, so long as the chunks do not overlap. This file format was developed as part of the chunking evaluation task run by the Conference on Natural Language Learning in 2000, and a section of Wall Street Journal text has been annotated in this format.

Chunk structures can also be represented using trees, which have the benefit that each chunk is a constituent that can be manipulated directly. NLTK uses trees for its internal representation of chunks, and provides methods for reading and writing

such trees to the IOB format.

What problem would be caused if we used I and O tags exclusively?

In IOB format, tokens are tagged with one of three special chunk tags, I (inside), O (outside), or B (begin). If we use I and O exclusively, extracted information is no longer accurate. The beginnings of the chunks or sentences are missing.

Provide a sentence to illustrate your point.

Example: saw yellow cat

1. Write tag patterns to match noun phrases containing plural head nouns, e.g. "many/JJ researchers/NNS", "two/CD weeks/NNS", "both/DT new/JJ positions/NNS". Extend the “grammar” defined in Example 3 (Example 2.2 named code\_chunkex in the chapter 7).

by your regular expressions. Redefine the “cp” object from this example to use your new grammar. Use this object to parse sentenceSample defined as follows

sentenceSample = [("Many", "JJ"), ("little", "JJ"), ("dogs", "NNS"), ("barked", "VBD"), ("at", "IN"), ("cats", "NNS")]

Report your outcome.

Code:

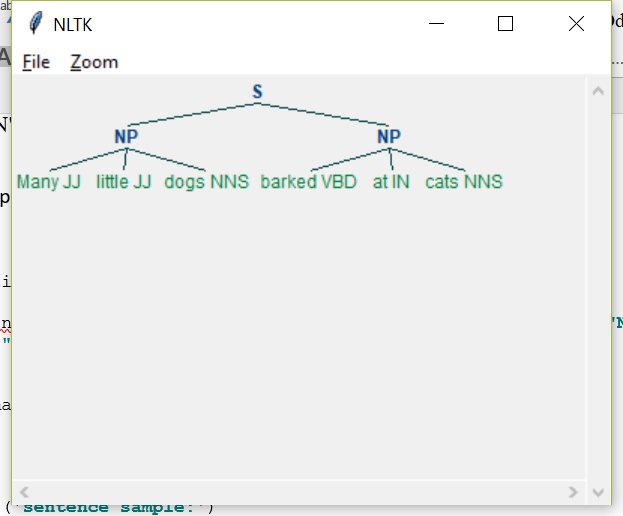
**def** exercise2():  
  
 sentenceSample = [(**"Many"**, **"JJ"**), (**"little"**, **"JJ"**), (**"dogs"**, **"NNS"**), (**"barked"**, **"VBD"**), (**"at"**, **"IN"**),  
 (**"cats"**, **"NNS"**)]  
  
 grammar = **r"""  
 NP: {<DT>?<JJ>\*<NN>}  
 {<VBD>?<IN>?<JJ>\*<NNS>}  
 """** print(**'sentence sample:'**)  
 cp = nltk.RegexpParser(grammar)  
 result2 = cp.parse(sentenceSample)  
 print(result2)  
 result2.draw()

Result:

Exercise 2

sentence sample:

(S (NP Many/JJ little/JJ dogs/NNS) (NP barked/VBD at/IN cats/NNS))



1. Carry out the following evaluation tasks for the chunker you have developed in question 2.
2. Evaluate your chunker on first 100 sentences from a chunked corpus nltk.corpus.conll2000, and report the precision, recall and F-measure.
3. Compare the performance of your chunker to the baseline chunker discussed in the evaluation section of 3 (the very first chunker that does nothing).
4. Extend the “grammar” of your chunker by at least one more regular expression. Give rationally behind your extension. See whether this extension allows you to boost the performance of your chunker. Evaluate your new chunker on 100 sentences from a chunked corpus nltk.corpus.conll2000, and report the precision, recall and F-measure.

Code:

**def** exercise3():  
 *# question a* print(**"part a"**)  
 test\_sents = conll2000.chunked\_sents(**'train.txt'**)[:99]  
 grammar = **r"""  
 NP: {<DT>?<JJ>\*<NN>}  
 {<VBD>?<IN>?<JJ>\*<NNS>}  
 """** cp = nltk.RegexpParser(grammar)  
 print(cp.evaluate(test\_sents))  
  
 *# question b* print(**"part b"**)  
 *# chunk\_types = ['NP', 'NNS','JJ', 'NNS', 'VBD', 'IN']* test\_sents = **"Many little dogs barked at cats"** *# train\_sents = conll2000.chunked\_sents('train.txt', chunk\_types=chunk\_types)  
  
 # establishing a baseline for the trivial chunk parser cp that creates no chunks* cp = nltk.RegexpParser(**""**)  
 test\_sents = conll2000.chunked\_sents(**'test.txt'**, chunk\_types=[**'NP'**])  
 print(**"Baseline with no chunks"**, cp.evaluate(test\_sents))  
  
 grammar = **r"NP: {<[CDJNP].\*>+}"** *# tags beginning with letters that are characteristic of noun phrase tags (e.g. CD, DT, and JJ)* cp = nltk.RegexpParser(grammar)  
 print(**"IOB tag evaluation"**, cp.evaluate(test\_sents))  
  
 *# question c* print(**"part c"**)  
 test\_sents = conll2000.chunked\_sents(**'train.txt'**)[:99]  
 grammar = **r"""  
 NP: {<DT>?<JJ>\*<NN>}  
 {<VBD>?<IN>?<JJ>\*<NNS>}  
 {<[CDJNP].\*>+}  
 """** cp = nltk.RegexpParser(grammar)  
 print(cp.evaluate(test\_sents))

Result:

Exercise 3

part a

ChunkParse score:

IOB Accuracy: 37.3%%

Precision: 38.6%%

Recall: 17.8%%

F-Measure: 24.4%%

part b

Baseline with no chunks ChunkParse score:

IOB Accuracy: 43.4%%

Precision: 0.0%%

Recall: 0.0%%

F-Measure: 0.0%%

IOB tag evaluation ChunkParse score:

IOB Accuracy: 87.7%%

Precision: 70.6%%

Recall: 67.8%%

F-Measure: 69.2%%

For this part, the precision, recall and F-measure of Baseline are 0%, while for IOB tag, we get a decent result. The reason is that the grammar for baseline is null, but the grammar for IOB that I use is grammar = **r"NP: {<[CDJNP].\*>+}".** This means that we can match more tags than baseline, for instance, “JJ, CD, NP”. Therefore, we got more decent result than the baseline.

part c

ChunkParse score:

IOB Accuracy: 52.7%%

Precision: 37.6%%

Recall: 29.8%%

F-Measure: 33.3%%

The difference that I make for the regex is

grammar = **r"""  
 NP: {<DT>?<JJ>\*<NN>}  
 {<VBD>?<IN>?<JJ>\*<NNS>}  
 {<[CDJNP].\*>+}  
 """**

This means that we can match more tags than part a. For example, “CD”. Therefore, we can get more decent result that the part a.