## 7.2 Chunking

The basic technique we will use for entity detection is chunking, which segments and labels multi-token sequences as illustrated in 7.2. The smaller boxes show the word-level tokenization and part-of-speech tagging, while the large boxes show higher-level chunking. Each of these larger boxes is called a chunk. Like tokenization, which omits whitespace, chunking usually selects a subset of the tokens. Also like tokenization, the pieces produced by a chunker do not overlap in the source text.

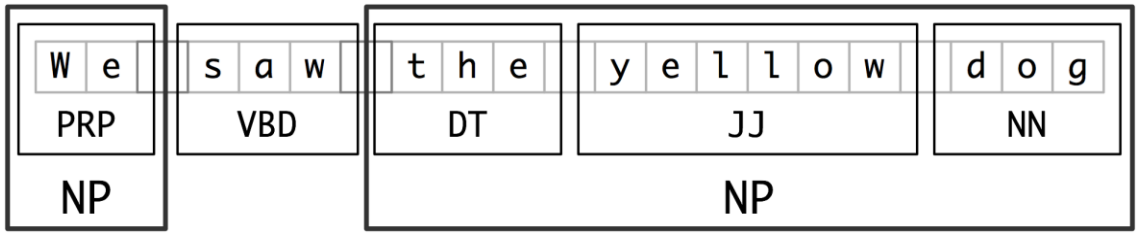



Figure 7.2: Segmentation and Labeling at both the Token and Chunk Levels

In this section, we will explore chunking in some depth, beginning with the definition and representation of chunks. We will see regular expression and n-gram approaches to chunking, and will develop and evaluate chunkers using the CoNLL-2000 chunking corpus. We will then return in (5) and 7.6 to the tasks of named entity recognition and relation extraction.

### Noun Phrase Chunking

We will begin by considering the task of noun phrase chunking, or NP-chunking, where we search for chunks corresponding to individual noun phrases. For example, here is some Wall Street Journal text with NP-chunks marked using brackets:

(2) [ The/DT market/NN ] for/IN [ system-management/NN software/NN ] for/IN [ Digital/NNP ] [ 's/POS hardware/NN ] is/VBZ fragmented/JJ enough/RB that/IN [ a/DT giant/NN ] such/JJ as/IN [ Computer/NNP Associates/NNPS ] should/MD do/VB well/RB there/RB ./.

As we can see, NP-chunks are often smaller pieces than complete noun phrases. For example, the market for system-management software for Digital's hardware is a single noun phrase (containing two nested noun phrases), but it is captured in NP-chunks by the simpler chunk the market. One of the motivations for this difference is that NP-chunks are defined so as not to contain other NP-chunks. Consequently, any prepositional phrases or subordinate clauses that modify a nominal will not be included in the corresponding NP-chunk, since they almost certainly contain further noun phrases.

One of the most useful sources of information for NP-chunking is part-of-speech tags. This is one of the motivations for performing part-of-speech tagging in our information extraction system. We demonstrate this approach using an example sentence that has been part-of-speech tagged in 7.3. In order to create an NP-chunker, we will first define a chunk grammar, consisting of rules that indicate how sentences should be chunked. In this case, we will define a simple grammar with a single regular-expression rule ❷. This rule says that an NP chunk should be formed whenever the chunker finds an

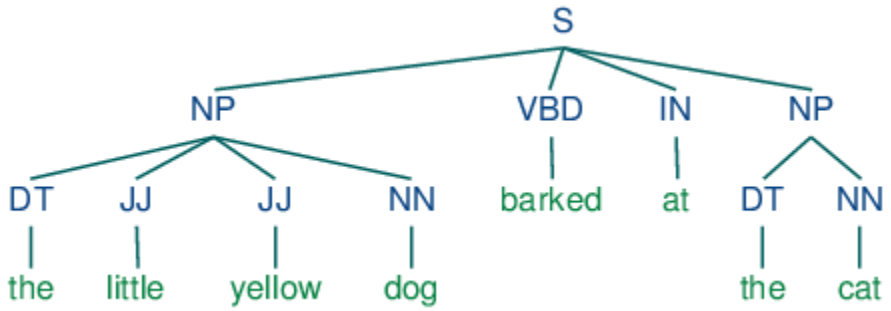
optional determiner (DT) followed by any number of adjectives (JJ) and then a noun (NN). Using this grammar, we create a chunk parser ❸, and test it on our example sentence ❹. The result is a tree, which we can either print ❺, or display graphically ❻.

```
>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ❶
... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat",
"NN")]

>>> grammar = "NP: {<DT>?<JJ>*<NN>}" ❷

>>> cp = nltk.RegexpParser(grammar) ❸
>>> result = cp.parse(sentence) ❹
>>> print result ❺
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
>>> result.draw() ❻
```

Example 7.3 (code_chunkex.py): Figure 7.3: Example of a Simple Regular Expression Based NP Chunker.




## Tag Patterns

The rules that make up a chunk grammar use tag patterns to describe sequences of tagged words. A tag pattern is a sequence of part-of-speech tags delimited using angle brackets, e.g. <DT>?<JJ>*<NN>. Tag patterns are similar to regular expression patterns (3.4). Now, consider the following noun phrases from the Wall Street Journal:

```
another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
```

We can match these noun phrases using a slight refinement of the first tag pattern above, i.e. <DT>?<JJ.*>*<NN.*>+. This will chunk any sequence of tokens beginning with an optional determiner, followed by zero or more adjectives of any type (including relative adjectives like earlier/JJR), followed

by one or more nouns of any type. However, it is easy to find many more complicated examples which this rule will not cover:

```
his/PRP$ Mansion/NNP House/NNP speech/NN
the/DT price/NN cutting/VBG
3/CD %/NN to/TO 4/CD %/NN
more/JJR than/IN 10/CD %/NN
the/DT fastest/JJS developing/VBG trends/NNS
's/POS skill/NN
```

Note

**Your Turn:** Try to come up with tag patterns to cover these cases. Test them using the graphical interface nltk.app.chunkparser(). Continue to refine your tag patterns with the help of the feedback given by this tool.

## Chunking with Regular Expressions

To find the chunk structure for a given sentence, the RegexpParser chunker begins with a flat structure in which no tokens are chunked. The chunking rules are applied in turn, successively updating the chunk structure. Once all of the rules have been invoked, the resulting chunk structure is returned.

7.4 shows a simple chunk grammar consisting of two rules. The first rule matches an optional determiner or possessive pronoun, zero or more adjectives, then a noun. The second rule matches one or more proper nouns. We also define an example sentence to be chunked ❶, and run the chunker on this input ❷.

```
grammar = r"""
  NP: {<DT|PP\$>?<JJ>*<NN>}   # chunk determiner/possessive, adjectives and
nouns
      {<NNP>+}                # chunk sequences of proper nouns
"""
cp = nltk.RegexpParser(grammar)
sentence = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"),
                ("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair",
"NN")]
>>> print cp.parse(sentence)
(S
  (NP Rapunzel/NNP)
  let/VBD
  down/RP
  (NP her/PP$ long/JJ golden/JJ hair/NN))
```

Example 7.4 (code_chunker1.py): Figure 7.4: Simple Noun Phrase Chunker

Note

The $ symbol is a special character in regular expressions, and must be backslash escaped in order to match the tag PP$.

If a tag pattern matches at overlapping locations, the leftmost match takes precedence. For example, if we apply a rule that matches two consecutive nouns to a text containing three consecutive nouns, then only the first two nouns will be chunked:

```
>>> nouns = [("money", "NN"), ("market", "NN"), ("fund", "NN")]
>>> grammar = "NP: {<NN><NN>}  # Chunk two consecutive nouns"
>>> cp = nltk.RegexpParser(grammar)
>>> print cp.parse(nouns)
(S (NP money/NN market/NN) fund/NN)
```

Once we have created the chunk for money market, we have removed the context that would have permitted fund to be included in a chunk. This issue would have been avoided with a more permissive chunk rule, e.g. NP: {<NN>+}.

Note

We have added a comment to each of our chunk rules. These are optional; when they are present, the chunker prints these comments as part of its tracing output.

## Exploring Text Corpora

In 5.2 we saw how we could interrogate a tagged corpus to extract phrases matching a particular sequence of part-of-speech tags. We can do the same work more easily with a chunker, as follows:

```
>>> cp = nltk.RegexpParser('CHUNK: {<V.*> <TO> <V.*>}')
>>> brown = nltk.corpus.brown
>>> for sent in brown.tagged_sents():
...     tree = cp.parse(sent)
...     for subtree in tree.subtrees():
...         if subtree.node == 'CHUNK': print subtree
...
(CHUNK combined/VBN to/TO achieve/VB)
(CHUNK continue/VB to/TO place/VB)
(CHUNK serve/VB to/TO protect/VB)
(CHUNK wanted/VBD to/TO wait/VB)
(CHUNK allowed/VBN to/TO place/VB)
(CHUNK expected/VBN to/TO become/VB)
...
(CHUNK seems/VBZ to/TO overtake/VB)
(CHUNK want/VB to/TO buy/VB)
```

Note

**Your Turn:** Encapsulate the above example inside a function find_chunks() that takes a chunk string like `"CHUNK: {<V.*> <TO> <V.*>}"` as an argument. Use it to search the corpus for several other patterns, such as four or more nouns in a row, e.g. `"NOUNS: {<N.*>{4,}}"`

## Chinking

Sometimes it is easier to define what we want to exclude from a chunk. We can define a chink to be a sequence of tokens that is not included in a chunk. In the following example, barked/VBD at/IN is a chink:

```
[ the/DT little/JJ yellow/JJ dog/NN ] barked/VBD at/IN [ the/DT cat/NN ]
```

Chinking is the process of removing a sequence of tokens from a chunk. If the matching sequence of tokens spans an entire chunk, then the whole chunk is removed; if the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before. If the sequence is at the periphery of the chunk, these tokens are removed, and a smaller chunk remains. These three possibilities are illustrated in 7.3.

Table 7.3:

Three chinking rules applied to the same chunk

| `` | **Entire chunk** | **Middle of a chunk** | **End of a chunk** |
|---|---|---|---|
| *Input* | [a/DT little/JJ dog/NN] | [a/DT little/JJ dog/NN] | [a/DT little/JJ dog/NN] |
| *Operation* | Chink "DT JJ NN" | Chink "JJ" | Chink "NN" |
| *Pattern* | }DT JJ NN{ | }JJ{ | }NN{ |
| *Output* | a/DT little/JJ dog/NN | [a/DT] little/JJ [dog/NN] | [a/DT little/JJ] dog/NN |

In 7.5, we put the entire sentence into a single chunk, then excise the chinks.

```
grammar = r"""
  NP:
    {<.*>+}          # Chunk everything
    }<VBD|IN>+{      # Chink sequences of VBD and IN
  """
sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
       ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat",
"NN")]
cp = nltk.RegexpParser(grammar)
>>> print cp.parse(sentence)
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
```

Example 7.5 (code_chinker.py): Figure 7.5: Simple Chinker

## Representing Chunks: Tags vs Trees

As befits their intermediate status between tagging and parsing (8), chunk structures can be represented using either tags or trees. The most widespread file representation uses IOB tags. In this scheme, each token is tagged with one of three special chunk tags, I (inside), O (outside), or B (begin). A token is tagged as B if it marks the beginning of a chunk. Subsequent tokens within the chunk are tagged I. All

other tokens are tagged O. The B and I tags are suffixed with the chunk type, e.g. B-NP, I-NP. Of course, it is not necessary to specify a chunk type for tokens that appear outside a chunk, so these are just labeled O. An example of this scheme is shown in 7.6.

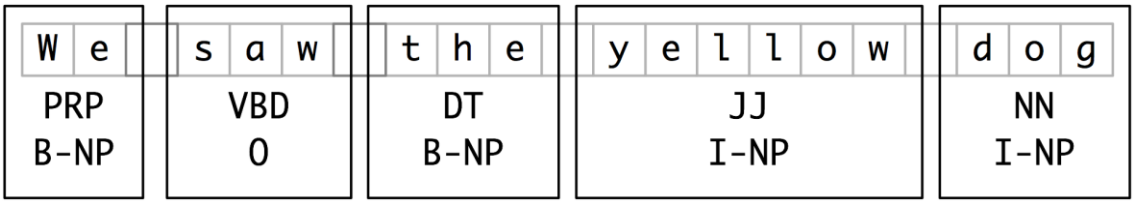



Figure 7.6: Tag Representation of Chunk Structures

IOB tags have become the standard way to represent chunk structures in files, and we will also be using this format. Here is how the information in 7.6 would appear in a file:

```
We PRP B-NP
saw VBD O
the DT B-NP
little JJ I-NP
yellow JJ I-NP
dog NN I-NP
```

In this representation there is one token per line, each with its part-of-speech tag and chunk tag. This format permits us to represent more than one chunk type, so long as the chunks do not overlap. As we saw earlier, chunk structures can also be represented using trees. These have the benefit that each chunk is a constituent that can be manipulated directly. An example is shown in 7.7.

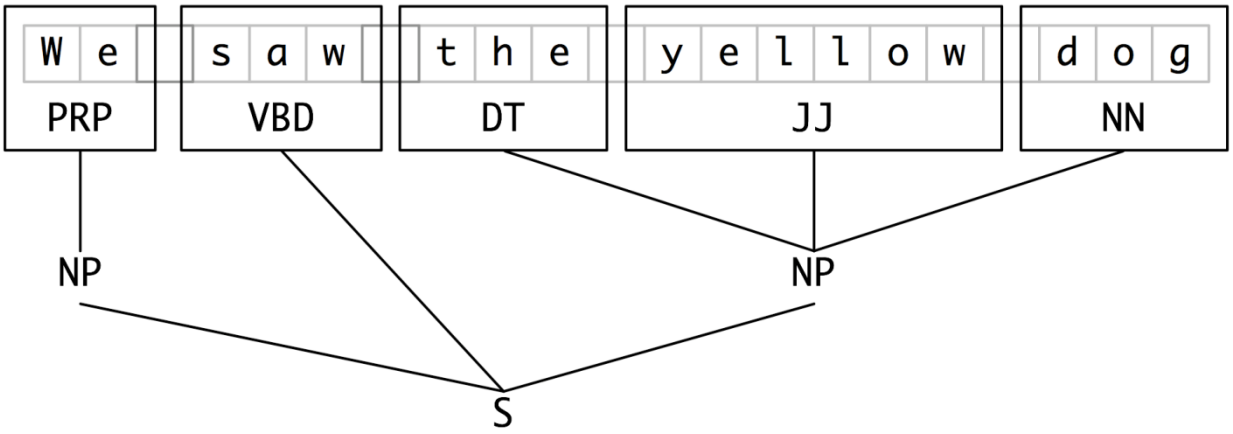



Figure 7.7: Tree Representation of Chunk Structures

Note

NLTK uses trees for its internal representation of chunks, but provides methods for reading and writing such trees to the IOB format.