Chapter 5

Chunking

5.1 Introduction

Chunking is an efficient and robust method for identifying short phrases in text, or "chunks". Chunks are *non-overlapping spans of text*, usually consisting of a head word (such as a noun) and the adjacent modifiers and function words (such as adjectives and determiners). For example, here is some Wall Street Journal text with noun phrase chunks marked using brackets (this data is distributed with NLTK):

> [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 ./.

There are two motivations for chunking: to locate information, and to ignore information. In the former case, we may want to extract all noun phrases so they can be indexed. A text retrieval system could use such an index to support efficient retrieval for queries involving terminological expressions.

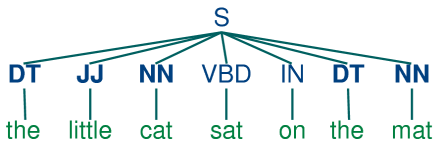
The reverse side of the coin is to *ignore* information. Suppose that we want to study syntactic patterns, finding particular verbs in a corpus and displaying their arguments. For instance, here are some uses of the verb `gave` in the Wall Street Journal (in the Penn Treebank corpus sample). After doing NP-chunking, the internal details of each noun phrase have been suppessed, allowing us to see some higher-level patterns:

```
gave NP
gave up NP in NP
gave NP up
gave NP NP
gave NP to NP
```

In this way we can acquire information about the complementation patterns of a verb like `gave`, for use in the development of a grammar (see Chapter 7).

Chunking in NLTK begins with tagged tokens, converted into a tree. (We will learn all about trees in Part II; for now its enough to know that they are hierarchical structures built over sequences of tagged tokens.)

```
>>> from nltk_lite.parse import Tree
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("cat", "NN"),
...      ("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> Tree('S', tagged_tokens).draw()
```

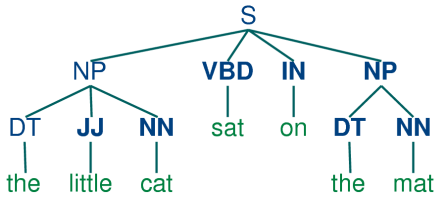


Next, we write regular expressions over tag sequences. The following example identifies noun phrases that consist of an optional determiner, followed by any number of adjectives, then a noun.

```
>>> from nltk_lite import chunk
>>> cp = chunk.Regexp("NP: {<DT>?<JJ>*<NN>}")
```

We create a chunker `cp` which can then be used repeatedly to parse tagged input. The result of chunking is also a tree, but with some extra structure:

```
>>> cp.parse(tagged_tokens).draw()
```



In this chapter we explore chunking in 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.

5.2 Defining and Representing Chunks

5.2.1 An Analogy

Two of the most common operations in language processing are *segmentation* and *labeling*. Recall that in tokenization, we *segment* a sequence of characters into tokens, while in tagging we *label* each of these tokens. Moreover, these two operations of segmentation and labeling go hand in hand. We break up a stream of characters into linguistically meaningful segments (e.g. words) so that we can classify those segments with their part-of-speech categories. The result of such classification is represented by adding a label to the segment in question.

In this chapter we do this segmentation and labeling at the phrase level, as illustrated in Figure 5.1. The solid boxes show word-level segmentation and labeling, while the dashed boxes show a higher-level segmentation and labeling. These larger pieces are called **chunks**, and the process of identifying them is called **chunking**.

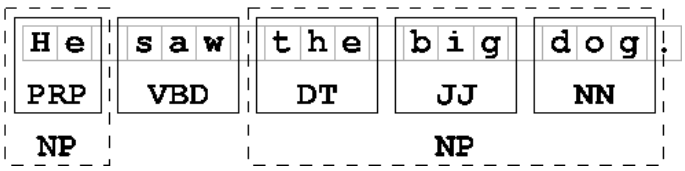


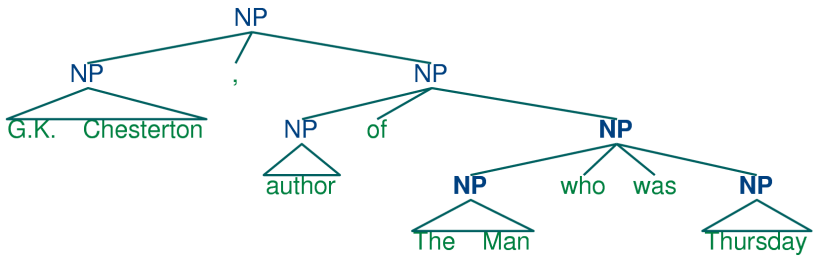
Figure 5.1: Segmentation and Labeling at both the Token and Chunk Levels

Like tokenization, chunking can skip over material in the input. Tokenization omits white space and punctuation characters. Chunking uses only a subset of the tokens and leaves others out.

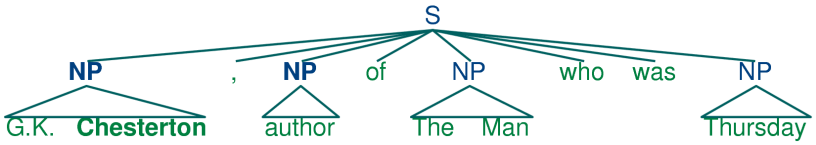
5.2.2 Chunking vs Parsing

Chunking is akin to parsing in the sense that it can be used to build hierarchical structure over text. There are several important differences, however. First, as noted above, chunking is not exhaustive, and typically omits items in the surface string. Second, where parsing constructs deeply nested structures, chunking creates structures of fixed depth, (typically depth 2). These chunks often correspond to the lowest level of grouping identified in the full parse tree, as illustrated in the parsing and chunking examples in (1) below:

(1a)



(1b)



A significant motivation for chunking is its robustness and efficiency relative to parsing. Parsing uses recursive phrase structure grammars and arbitrary-depth trees. Parsing has problems with robustness, given the difficulty in getting broad coverage and in resolving ambiguity. Parsing is also relatively inefficient: the time taken to parse a sentence grows with the cube of the length of the sentence, while the time taken to chunk a sentence only grows linearly.

5.2.3 Representing Chunks: Tags vs Trees

As befits its intermediate status between tagging and parsing, chunk structures can be represented using either tags or trees. The most widespread file representation uses so-called **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 Figure 5.2.

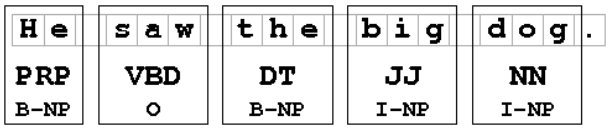


Figure 5.2: Tag Representation of Chunk Structures

IOB tags have become the standard way to represent chunk structures in files, and we will also be using this format. Here is an example of the file representation of the information in Figure 5.2:

```
He PRP B-NP
saw VBD O
the DT B-NP
big JJ I-NP
dog NN I-NP
```

In this representation, there is one token per line, each with its part-of-speech tag and its chunk tag. We will see later that this format permits us to represent more than one chunk type, so long as the chunks do not overlap. This file format was developed for NP chunking by [Ramshaw & Marcus, 1995], and was used for the shared NP bracketing task run by the *Conference on Natural Language Learning* (CoNLL) in 1999. It has come to be called the **IOB Format** (or sometimes **BIO Format**). The same format was adopted by CoNLL 2000 for annotating a section of Wall Street Journal text as part of a shared task on NP chunking.

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 Figure 5.3:

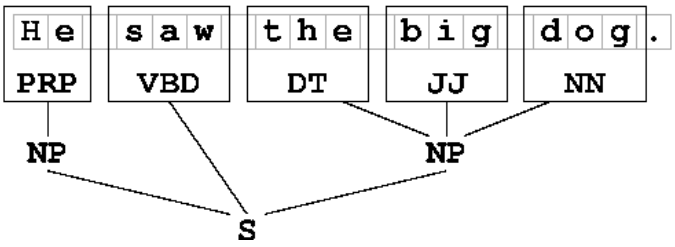


Figure 5.3: Tree Representation of Chunk Structures

NLTK uses trees for its internal representation of chunks, and provides methods for reading and writing such trees to the IOB format. By now you should understand what chunks are, and how they are represented. In the next section you will see how to build a simple chunker.

5.3 Chunking

A **chunker** finds contiguous, non-overlapping spans of related tokens and groups them together into *chunks*. Chunkers often operate on tagged texts, and use the tags to make chunking decisions. In this section we will see how to write a special type of regular expression over part-of-speech tags, and then how to combine these into a chunk grammar. Then we will set up a chunker to chunk some tagged text according to the grammar.

5.3.1 Tag Patterns

A **tag pattern** is a sequence of part-of-speech tags delimited using angle brackets, e.g. `<DT><JJ><NN>`. Tag patterns are the same as the regular expression patterns we have already seen, except for two differences which make them easier to use for chunking. First, angle brackets group their contents into atomic units, so "`<NN>+`" matches one or more repetitions of the tag `NN`; and "`<NN|JJ>`" matches the `NN` or `JJ`. Second, the period wildcard operator is constrained not to cross tag delimiters, so that "`<N.*>`" matches any single tag starting with `N`, e.g. `NN`, `NNS`.

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 using a slight refinement of the first tag pattern above: `<DT>?<JJ.*>*<NN.*>+`. This can be used to chunk any sequence of tokens beginning with an optional determiner `DT`, followed by zero or more adjectives of any type `JJ.*` (including relative adjectives like `earlier/JJR`), followed by one or more nouns of any type `NN.*`. It is easy to find many more difficult examples:

```
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
```

Your challenge will be to come up with tag patterns to cover these and other examples.

5.3.2 Chunking with Regular Expressions

The chunker begins with a flat structure in which no tokens are chunked. Patterns are applied in turn, successively updating the chunk structure. Once all of the patterns have been applied, the resulting chunk structure is returned. Listing 5.1 shows a simple chunk grammar consisting of two patterns. The first pattern matches an optional determiner, zero or more adjectives, then a noun. We also define some tagged tokens to be chunked, and run the chunker on this input.

Listing 1 Simple Noun Phrase Chunker

```
grammar = r"""
NP:
  {<DT>?<JJ>*<NN>}     # chunk determiners, adjectives and nouns
  {<NNP>+}             # chunk sequences of proper nouns
"""
cp = chunk.Regexp(grammar)
tagged_tokens = [("the", "DT"), ("little", "JJ"), ("cat", "NN"),
     ("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

>>> cp.parse(tagged_tokens)
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
```

If a tag pattern matches at overlapping locations, the first 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 = chunk.Regexp(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>+}`.

5.3.3 Developing Chunkers

Creating a good chunker usually requires several rounds of development and testing, during which existing rules are refined and new rules are added. In order to diagnose any problems, it often helps to trace the execution of a chunker, using its `trace` argument. The tracing output shows the rules that are applied, and uses braces to show the chunks that are created at each stage of processing. In Listing 5.2, two chunk patterns are applied to the input sentence. The first rule finds all sequences of three tokens whose tags are `DT`, `JJ`, and `NN`, and the second rule finds any sequence of tokens whose tags are either `DT` or `NN`. We set up two chunkers one for each rule ordering, and test them on the same input.

Observe that when we chunk material that is already partially chunked, the chunker will only create chunks that do not partially overlap existing chunks. In the case of `cp2`, the second rule did not find any chunks, since all chunks that matched its tag pattern overlapped with existing chunks. Therefore it is necessary to be careful to put chunk rules in the right order.

5.3.4 Exercises

1. ☼ **Chunking Demonstration:** Run the chunking demonstration:

   ```
   from nltk_lite import chunk
   chunk.demo()  # the chunker
   ```

2. ☼ **IOB Tags:** 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?

3. ☼ Write a tag pattern to match noun phrases containing plural head nouns, e.g. "many/JJ researchers/NNS", "two/CD weeks/NNS", "both/DT new/JJ positions/NNS". Try to do this by generalizing the tag pattern that handled singular noun phrases.

4. ◑ Write tag pattern to cover noun phrases that contain gerunds, e.g. "the/DT receiving/VBG end/NN", "assistant/NN managing/VBG editor/NN". Add these patterns to the grammar, one per line. Test your work using some tagged sentences of your own devising.

5. ◑ Write one or more tag patterns to handle coordinated noun phrases, e.g. "July/NNP and/CC August/NNP", "all/DT your/PRP$ managers/NNS and/CC supervisors/NNS", "company/NN courts/NNS and/CC adjudicators/NNS".

6. ◑ Sometimes a word is incorrectly tagged, e.g. the head noun in "12/CD or/CC so/RB cases/VBZ". Instead of requiring manual correction of tagger output, good chunkers are able to work with the erroneous output of taggers. Look for other examples of correctly chunked noun phrases with incorrect tags.

Listing 2 Two Noun Phrase Chunkers Having Identical Rules in Different Orders

```
cp1 = chunk.Regexp(r"""
NP: {<DT><JJ><NN>}      # Chunk det+adj+noun
    {<DT|NN>+}          # Chunk sequences of NN and DT
""")
cp2 = chunk.Regexp(r"""
NP: {<DT|NN>+}          # Chunk sequences of NN and DT
    {<DT><JJ><NN>}      # Chunk det+adj+noun
""")

>>> print cp1.parse(tagged_tokens, trace=1)
# Input:
 <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
# Chunk det+adj+noun:
{<DT>  <JJ>  <NN>} <VBD>  <IN>  <DT>  <NN>
# Chunk sequences of NN and DT:
{<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
>>> print cp2.parse(tagged_tokens, trace=1)
# Input:
 <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
# Chunk sequences of NN and DT:
{<DT>} <JJ> {<NN>} <VBD>  <IN> {<DT>  <NN>}
# Chunk det+adj+noun:
{<DT>} <JJ> {<NN>} <VBD>  <IN> {<DT>  <NN>}
(S:
  (NP: ('the', 'DT'))
  ('little', 'JJ')
  (NP: ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
```

5.4 Scaling Up

Now that you have a taste of what chunking can do, you are ready to look at a chunked corpus, and use it in developing and testing more complex chunkers. We will begin by looking at the mechanics of converting IOB format into an NLTK tree, then at how this is done on a larger scale using the corpus directly. We will see how to use the corpus to score the accuracy of a chunker, then look some more flexible ways to manipulate chunks. Throughout our focus will be on scaling up the coverage of a chunker.

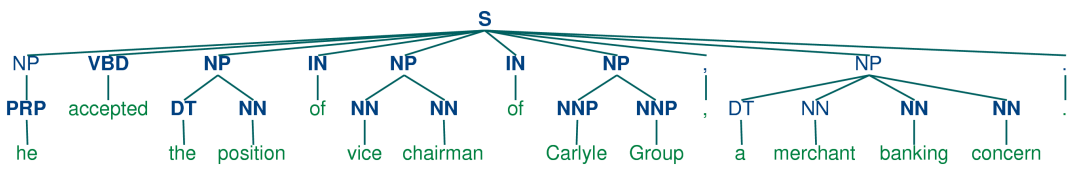
5.4.1 Reading IOB Format and the CoNLL 2000 Corpus

Using the `nltk_lite.corpora` module we can load Wall Street Journal text that has been tagged, then chunked using the IOB notation. The chunk categories provided in this corpus are NP, VP and PP. As we have seen, each sentence is represented using multiple lines, as shown below:

```
he PRP B-NP
accepted VBD B-VP
the DT B-NP
position NN I-NP
...
```

A conversion function `chunk.conllstr2tree()` builds a tree representation from one of these multi-line strings. Moreover, it permits us to choose any subset of the three chunk types to use. The example below produces only `NP` chunks:

```
>>> text = '''
... he PRP B-NP
... accepted VBD B-VP
... the DT B-NP
... position NN I-NP
... of IN B-PP
... vice NN B-NP
... chairman NN I-NP
... of IN B-PP
... Carlyle NNP B-NP
... Group NNP I-NP
... , , O
... a DT B-NP
... merchant NN I-NP
... banking NN I-NP
... concern NN I-NP
... . . O
... '''
>>> chunk.conllstr2tree(text, chunk_types=('NP',)).draw()
```



We can use the NLTK corpus module to access a larger amount of chunked text. The CoNLL 2000 corpus contains 270k words of Wall Street Journal text, with part-of-speech tags and chunk tags in the

IOB format. We can access this data using an NLTK corpus reader called `conll2000`. Here is an example:

```
>>> from nltk_lite.corpora import conll2000, extract
>>> print extract(2000, conll2000.chunked())
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  (VP: ('should', 'MD') ('get', 'VB'))
  ('healthier', 'JJR')
  (PP: ('in', 'IN'))
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
```

This just showed three chunk types, for NP, VP and PP. We can also select which chunk types to read:

```
>>> from nltk_lite.corpora import conll2000, extract
>>> print extract(2000, conll2000.chunked(chunk_types=('NP',)))
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  ('should', 'MD')
  ('get', 'VB')
  ('healthier', 'JJR')
  ('in', 'IN')
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
```

5.4.2 Simple Evaluation and Baselines

Armed with a corpus, it is now possible to do some simple evaluation. The first evaluation is to establish a baseline for the case where nothing is chunked:

```
>>> cp = chunk.Regexp("")
>>> print chunk.accuracy(cp, conll2000.chunked(chunk_types=('NP',)))
0.440845995079
```

Now let's try a naive regular expression chunker that looks for tags beginning with letters that are typical of noun phrase tags:

```
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = chunk.Regexp(grammar)
>>> print chunk.accuracy(cp, conll2000.chunked(chunk_types=('NP',)))
0.874479872666
```

We can extend this approach, and create a function `chunked_tags()` that takes some chunked data, and sets up a conditional frequency distribution. For each tag, it counts up the number of times the tag occurs inside an NP chunk (the `True` case), or outside a chunk (the `False` case). It returns a list of those tags that occur inside chunks more often than outside chunks.

```
>>> def chunked_tags(train):
...     """Generate a list of tags that tend to appear inside chunks"""
...     from nltk_lite.probability import ConditionalFreqDist
...     cfdist = ConditionalFreqDist()
...     for t in train:
...         for word, tag, chtag in chunk.tree2conlltags(t):
```

```
...             if chtag == "O":
...                 cfdist[tag].inc(False)
...             else:
...                 cfdist[tag].inc(True)
...     return [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]
```

The next step is to convert this list of tags into a tag pattern. To do this we need to "escape" all non-word characters, by preceding them with a backslash. Then we need to join them into a disjunction. This process would convert a tag list `['NN', 'NN$']` into the tag pattern `<NN|NN\$>`. The following function does this work, and returns a regular expression chunker:

```
>>> def baseline_chunker(train):
...     import re
...     chunk_tags = [re.sub(r'(\W)', r'\\\1', tag)
...                   for tag in chunked_tags(train)]
...     grammar = 'NP: {<' + '|'.join(chunk_tags) + '>+}'
...     return chunk.Regexp(grammar)
```

The final step is to train this chunker and test its accuracy (this time on data not seen during training):

```
>>> cp = baseline_chunker(conll2000.chunked(files='train', chunk_types=('NP',)))
>>> print chunk.accuracy(cp, conll2000.chunked(files='test', chunk_types=('NP',)))
0.914262194736
```

5.4.3 Splitting and Merging (incomplete)

[Notes: the above approach creates chunks that are too large, e.g. *the cat the dog chased* would be given a single NP chunk because it does not detect that determiners introduce new chunks. For this we would need a rule to split an NP chunk prior to any determiner, using a pattern like: `"NP: <.*>}{<DT>"`. We can also merge chunks, e.g. `"NP: <NN>{}<NN>"`.]

5.4.4 Chinking

Sometimes it is easier to define what we *don't* want to include in a chunk than it is to define what we *do* want to include. In these cases, it may be easier to build a chunker using a method called **chinking**.

The word **chink** initially meant a sequence of stopwords, according to a 1975 paper by Ross and Tukey [Church, Young, & Bloothooft, 1996]. Following Abney, we define a *chink* is a sequence of tokens that is not included in a chunk. In the following example, `sat/VBD on/IN` is a chink:

```
[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]
```

Chinking is the process of removing a sequence of tokens from a chunk. If the 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 beginning or end of the chunk, these tokens are removed, and a smaller chunk remains. These three possibilities are illustrated in Table 5.1.

	Entire chunk	Middle of a chunk	End of a chunk
Input	[a/DT big/JJ cat/NN]	[a/DT big/JJ cat/NN]	[a/DT big/JJ cat/NN]
Operation	Chink "DT JJ NN"	Chink "JJ"	Chink "NN"
Pattern	"}DT JJ NN{"	"}JJ{"	"}NN{"

Output	a/DT big/JJ cat/NN	[a/DT] big/JJ [cat/NN]	[a/DT big/JJ] cat/NN

Table 5.1: Three chinking rules applied to the same chunk

In the following grammar, we put the entire sentence into a single chunk, then excise the chink:

```
>>> grammar = r"""
... NP:
...   {<.*>+}          # Chunk everything
...   }<VBD|IN>+{      # Chink sequences of VBD and IN
... """
>>> cp = chunk.Regexp(grammar)
>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
>>> print chunk.accuracy(cp, conll2000.chunked(files='test', chunk_types=('NP',)))
0.581041433607
```

A chunk grammar can use any number of chunking and chinking patterns in any order.

5.4.5 Multiple Chunk Types (incomplete)

So far we have only developed NP chunkers. However, as we saw earlier in the chapter, the CoNLL chunking data is also annotated for PP and VP chunks. Here is an example, to show the structure we get from the corpus and the flattened version that will be used as input to the parser.

```
>>> example = extract(2000, conll2000.chunked())
>>> print example
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  (VP: ('should', 'MD') ('get', 'VB'))
  ('healthier', 'JJR')
  (PP: ('in', 'IN'))
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
>>> print example.flatten()
(S:
  ('Health-care', 'JJ')
  ('companies', 'NNS')
  ('should', 'MD')
  ('get', 'VB')
  ('healthier', 'JJR')
  ('in', 'IN')
  ('the', 'DT')
  ('third', 'JJ')
  ('quarter', 'NN')
  ('.', '.'))
```

Now we can set up a multi-stage chunk grammar, as shown in Listing 5.3. It has a stage for each of the chunk types.

Listing 3

```
cp = chunk.Regexp(r"""
NP: {<DT>?<JJ>*<NN.*>+} # noun phrase chunks
VP: {<TO>?<VB.*>}       # verb phrase chunks
PP: {<IN>}              # prepositional phrase chunks
""")

>>> example = extract(2000, conll2000.chunked())
>>> print cp.parse(example.flatten(), trace=1)
# Input:
 <JJ>  <NNS>  <MD>  <VB>  <JJR>  <IN>  <DT>  <JJ>  <NN>  <.>
# noun phrase chunks:
{<JJ>  <NNS>} <MD>  <VB>  <JJR>  <IN> {<DT>  <JJ>  <NN>} <.>
# Input:
 <NP>  <MD>  <VB>  <JJR>  <IN>  <NP>  <.>
# verb phrase chunks:
 <NP>  <MD> {<VB>} <JJR>  <IN>  <NP>  <.>
# Input:
 <NP>  <MD>  <VP>  <JJR>  <IN>  <NP>  <.>
# prepositional phrase chunks:
 <NP>  <MD>  <VP>  <JJR> {<IN>} <NP>  <.>
(S:
  (NP: ('Health-care', 'JJ') ('companies', 'NNS'))
  ('should', 'MD')
  (VP: ('get', 'VB'))
  ('healthier', 'JJR')
  (PP: ('in', 'IN'))
  (NP: ('the', 'DT') ('third', 'JJ') ('quarter', 'NN'))
  ('.', '.'))
```

5.4.6 Exercises

1. ☼ Pick one of the three chunk types in the CoNLL corpus. Inspect the CoNLL corpus and try to observe any patterns in the POS tag sequences that make up this kind of chunk. Develop a simple chunker using the regular expression chunker `chunk.Regexp`. Discuss any tag sequences that are difficult to chunk reliably.

2. ☼ An early definition of *chunk* was the material that occurs between chinks. Develop a chunker which starts by putting the whole sentence in a single chunk, and then does the rest of its work solely by chinking. Determine which tags (or tag sequences) are most likely to make up chinks with the help of your own utility program. Compare the performance and simplicity of this approach relative to a chunker based entirely on chunk rules.

3. ◑ Develop a chunker for one of the chunk types in the CoNLL corpus using a regular-expression based chunk grammar `RegexpChunk`. Use any combination of rules for chunking, chinking, merging or splitting.

4. ★ We saw in the tagging chapter that it is possible to establish an upper limit to tagging performance by looking for ambiguous n-grams, n-grams that are tagged in more than one possible way in the training data. Apply the same method to determine an upper bound on the performance of an n-gram chunker.

5. ★ Pick one of the three chunk types in the CoNLL corpus. Write functions to do the following tasks for your chosen type:

   a) List all the tag sequences that occur with each instance of this chunk type.

   b) Count the frequency of each tag sequence, and produce a ranked list in order of decreasing frequency; each line should consist of an integer (the frequency) and the tag sequence.

   c) Inspect the high-frequency tag sequences. Use these as the basis for developing a better chunker.

6. ★ The baseline chunker presented in the evaluation section tends to create larger chunks than it should. For example, the phrase: `[every/DT time/NN] [she/PRP] sees /VBZ [a/DT newspaper/NN]` contains two consecutive chunks, and our baseline chunker will incorrectly combine the first two: `[every/DT time/NN she/PRP]`. Write a program that finds which of these chunk-internal tags typically occur at the start of a chunk, then devise one or more rules that will split up these chunks. Combine these with the existing baseline chunker and re-evaluate it, to see if you have discovered an improved baseline.

7. ★ Develop an NP chunker which converts POS-tagged text into a list of tuples, where each tuple consists of a verb followed by a sequence of noun phrases and prepositions, e.g. `the little cat sat on the mat` becomes `('sat', 'on', 'NP')`...

8. ★ The Penn Treebank contains a section of tagged Wall Street Journal text which has been chunked into noun phrases. The format uses square brackets, and we have encountered it several times during this chapter. It can be accessed by importing the Treebank corpus reader (`from nltk_lite.corpora import treebank`), then iterating over its

chunked items (`for sent in treebank.chunked():`). These items are flat trees, just as we got using `conll2000.chunked()`.

a) Consult the documentation for the NLTK chunk package to find out how to generate Treebank and IOB strings from a tree. Write functions `chunk2brackets()` and `chunk2iob()` which take a single chunk tree as their sole argument, and return the required multi-line string representation.

b) Write command-line conversion utilities `bracket2iob.py` and `iob2bracket.py` that take a file in Treebank or CoNLL format (resp) and convert it to the other format. (Obtain some raw Treebank or CoNLL data from the NLTK Corpora, save it to a file, and then use `for line in open(filename)` to access it from Python.)

5.5 N-Gram Chunking

Our approach to chunking has been to try to detect structure based on the part-of-speech tags. We have seen that the IOB format represents this extra structure using another kind of tag. The question arises then, as to whether we could use the same n-gram tagging methods we saw in the last chapter, applied to a different vocabulary.

The first step is to get the `word,tag,chunk` triples from the CoNLL corpus and map these to `tag,chunk` pairs:

```
>>> from nltk_lite import tag
>>> chunk_data = [[(t,c) for w,t,c in chunk.tree2conlltags(chtree)]
...               for chtree in conll2000.chunked()]
```

5.5.1 A Unigram Chunker

Now we can train and score a **unigram chunker** on this data, just as if it was a tagger:

```
>>> unigram_chunker = tag.Unigram()
>>> unigram_chunker.train(chunk_data)
>>> print tag.accuracy(unigram_chunker, chunk_data)
0.781378851068
```

This chunker does reasonably well. Let's look at the errors it makes. Consider the opening phrase of the first sentence of the chunking data, here shown with part of speech tags:

> Confidence/NN in/IN the/DT pound/NN is/VBZ widely/RB expected/VBN to/TO take/VB another/DT sharp/JJ dive/NN

We can try the unigram chunker out on this first sentence by creating some "tokens" using `[t for t,c in chunk_data[0]]`, then running our chunker over them using `list(unigram_chunker.tag(tokens))`. The unigram chunker only looks at the tags, and tries to add chunk tags. Here is what it comes up with:

> NN/I-NP IN/B-PP DT/B-NP NN/I-NP VBZ/B-VP RB/O VBN/I-VP TO/B-PP VB/I-VP DT/B-NP JJ/I-NP NN/I-NP

Notice that it tags all instances of `NN` with `I-NP`, because nouns usually do not appear at the beginning of noun phrases in the training data. Thus, the first noun `Confidence/NN` is tagged incorrectly. However, `pound/NN` and `dive` are correctly tagged as `I-NP`; they are not in the initial position that should be tagged `B-NP`. It incorrectly tags `widely/RB` as outside `O`, and it incorrectly tags the infinitival `to/TO` as `B-PP`, as if it was a preposition starting a prepositional phrase.

5.5.2 A Bigram Chunker (incomplete)

[Why these problems might go away if we look at the previous chunk tag?]

Let's run a bigram chunker:

```
>>> bigram_chunker = tag.Bigram(backoff=unigram_chunker)
>>> bigram_chunker.train(chunk_data)
>>> print tag.accuracy(bigram_chunker, chunk_data)
0.89312652614
```

We can run the bigram chunker over the same sentence as before using `list(bigram_chunker.tag(tokens))`. Here is what it comes up with:

> 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

This is 100% correct.

5.5.3 Exercises

1. ◑ The bigram chunker scores about 90% accuracy. Study its errors and try to work out why it doesn't get 100% accuracy.

2. ◑ Experiment with trigram chunking. Are you able to improve the performance any more?

3. ★ An n-gram chunker can use information other than the current part-of-speech tag and the $n-1$ previous chunk tags. Investigate other models of the context, such as the $n-1$ previous part-of-speech tags, or some combination of previous chunk tags along with previous and following part-of-speech tags.

4. ★ Consider the way an n-gram tagger uses recent tags to inform its tagging choice. Now observe how a chunker may re-use this sequence information. For example, both tasks will make use of the information that nouns tend to follow adjectives (in English). It would appear that the same information is being maintained in two places. Is this likely to become a problem as the size of the rule sets grows? If so, speculate about any ways that this problem might be addressed.

5.6 Cascaded Chunkers

So far, our chunk structures have been relatively flat. Trees consist of tagged tokens, optionally grouped under a chunk node such as `NP`. However, it is possible to build chunk structures of arbitrary depth, simply by creating a multi-stage chunk grammar.

So far, our chunk grammars have consisted of a single stage: a chunk type followed by one or more patterns. However, chunk grammars can have two or more such stages. These stages are processed

in the order that they appear. The patterns in later stages can refer to a mixture of part-of-speech tags and chunk types. Listing 5.4 has patterns for noun phrases, prepositional phrases, verb phrases, and sentences. This is a four-stage chunk grammar, and can be used to create structures having a depth of at most four.

Listing 4 A Chunker that Handles NP, PP, VP and S

```
cp = chunk.Regexp(r"""
NP: {<DT|JJ|NN.*>+}          # Chunk sequences of DT, JJ, NN
PP: {<IN><NP>}               # Chunk prepositions followed by NP
VP: {<VB.*><NP|PP|S>+$}      # Chunk rightmost verbs and arguments/adjuncts
S:  {<NP><VP>}               # Chunk NP, VP
""")
tagged_tokens = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),
    ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('Mary', 'NN'))
  ('saw', 'VBD')
  (S:
    (NP: ('the', 'DT') ('cat', 'NN'))
    (VP:
      ('sit', 'VB')
      (PP: ('on', 'IN') (NP: ('the', 'DT') ('mat', 'NN'))))))
```

Unfortunately this result misses the VP headed by *saw*. It has other shortcomings too. Let's see what happens when we apply this chunker to a sentence having deeper nesting.

```
>>> tagged_tokens = [("John", "NNP"), ("thinks", "VBZ"), ("Mary", "NN"),
...     ("saw", "VBD"), ("the", "DT"), ("cat", "NN"), ("sit", "VB"),
...     ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('John', 'NNP'))
  ('thinks', 'VBZ')
  (NP: ('Mary', 'NN'))
  ('saw', 'VBD')
  (S:
    (NP: ('the', 'DT') ('cat', 'NN'))
    (VP:
      ('sit', 'VB')
      (PP: ('on', 'IN') (NP: ('the', 'DT') ('mat', 'NN'))))))
```

The solution to these problems is to get the chunker to loop over its patterns: after trying all of them, it repeats the process. We add an optional second argument `loop` to specify the number of times the set of patterns should be run:

```
>>> cp = chunk.Regexp(grammar, loop=2)
>>> print cp.parse(tagged_tokens)
(S:
  (NP: ('John', 'NNP'))
```

```
('thinks', 'VBZ')
(S:
  (NP: ('Mary', 'NN'))
  (VP:
    ('saw', 'VBD')
    (S:
      (NP: ('the', 'DT') ('cat', 'NN'))
      (VP:
        ('sit', 'VB')
        (PP: ('on', 'IN') (NP: ('the', 'DT') ('mat', 'NN'))))))))
```

This cascading process enables us to create deep structures. However, creating and debugging a cascade is quite difficult, and there comes a point where it is more effective to do full parsing (see Chapter 7).

5.7 Conclusion

In this chapter we have explored efficient and robust methods that can identify linguistic structures in text. Using only part-of-speech information for words in the local context, a "chunker" can successfully identify simple structures such as noun phrases and verb groups. We have seen how chunking methods extend the same lightweight methods that were successful in tagging. The resulting structured information is useful in information extraction tasks and in the description of the syntactic environments of words. The latter will be invaluable as we move to full parsing.

There are a surprising number of ways to chunk a sentence using regular expressions. The patterns can add, shift and remove chunks in many ways, and the patterns can be sequentially ordered in many ways. One can use a small number of very complex rules, or a long sequence of much simpler rules. One can hand-craft a collection of rules, and one can write programs to analyze a chunked corpus to help in the development of such rules. The process is painstaking, but generates very compact chunkers that perform well and that transparently encode linguistic knowledge.

It is also possible to chunk a sentence using the techniques of n-gram tagging. Instead of assigning part-of-speech tags to words, we assign IOB tags to the part-of-speech tags. Bigram tagging turned out to be particularly effective, as it could be sensitive to the chunk tag on the previous word. This statistical approach requires far less effort than rule-based chunking, but creates large models and delivers few linguistic insights.

Like tagging, chunking cannot be done perfectly. For example, as pointed out by [Church, Young, & Bloothooft, 1996], we cannot correctly analyze the structure of the sentence *I turned off the spectroroute* without knowing the meaning of *spectroroute*; is it a kind of road or a type of device? Without knowing this, we cannot tell whether *off* is part of a prepositional phrase indicating direction (tagged `B-PP`), or whether *off* is part of the verb-particle construction *turn off* (tagged `I-VP`).

A recurring theme of this chapter has been *diagnosis*. The simplest kind is manual, when we inspect the tracing output of a chunker and observe some undesirable behavior that we would like to fix. Sometimes we discover cases where we cannot hope to get the correct answer because the part-of-speech tags are too impoverished and do not give us sufficient information about the lexical item. A second approach is to write utility programs to analyze the training data, such as counting the number of times a given part-of-speech tag occurs inside and outside an NP chunk. A third approach is to evaluate the system against some gold standard data to obtain an overall performance score. We can even use this to parameterize the system, specifying which chunk rules are used on a given run, and tabulating performance for different parameter combinations. Careful use of these diagnostic methods

permits us to optimize the performance of our system. We will see this theme emerge again later in chapters dealing with other topics in natural language processing.

5.8 Further Reading

Abney's Cass system: `http://www.vinartus.net/spa/97a.pdf`