NLTK Tutorial: Chunking

Steven Bird

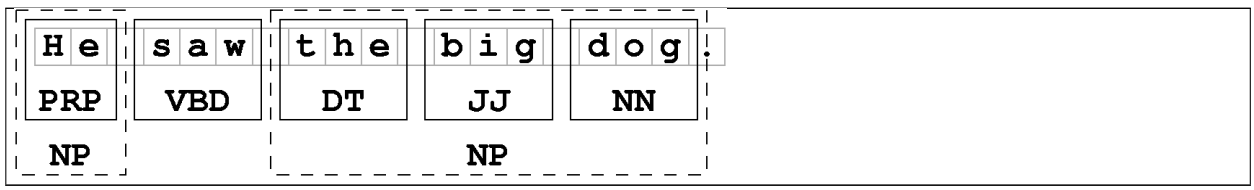
Edward Loper

Revision 1.43, 6 Apr 2005

1. Introduction

Two of the most common operations in language processing are *segmentation* and *labelling*. For example, tokenization *segments* a sequence of characters into tokens, while tagging *labels* each of these tokens. Moreover, these two operations go hand in hand. We segment a stream of characters into linguistically meaningful pieces (e.g. as words) only so that we can classify those pieces (e.g. with their part-of-speech categories) and then identify higher-level structures. The result of such classification is usually stored by adding a label to the piece in question. Now that we have mapped characters to tagged-tokens, we will carry on with segmentation and labelling at a higher level, as illustrated in Figure 1. This process is called *chunking*, and is also known as *chunk parsing*, *partial parsing*, or *light parsing*.

Figure 1. Segmentation and Labelling at both the Token and Chunk Levels



Chunking is like tokenization and tagging in other respects. First, chunking can skip over material in the input. Observe in Figure 1 that only some of the tagged tokens have been chunked, while others are left out. Compare this with the way that tokenization has omitted spaces and punctuation characters. Second, chunking typically uses finite state methods to identify material of interest. For example, the chunk in Figure 1 could have been found by the expression `<DT>?<JJ>*<NN>` which matches an optional determiner, followed by zero or more adjectives, followed by a noun.

Compare this with the way that tokenization and tagging both make use of regular expressions. Third, chunking, like tokenization and tagging, is very application-specific. For example, while linguistic analysis and information extraction both need to identify and label salient extents of text, they typically require quite different definitions of those extents.

There are two chief motivations for chunking: to locate information, or to ignore information. In the former case, we may want to extract all noun phrases so that they can be indexed. A text retrieval system could the use the index to support efficient retrieval for queries involving terminological expressions. In the latter case we may want to study syntactic patterns, finding particular verbs in a corpus and displaying their arguments. For instance, here are uses of the verb `gave` in the first 100 files of the Penn Treebank corpus. NP-chunking has been used so that the internal details of each noun phrase can be replaced with `NP`. In this way we can discover significant grammatical properties even before a grammar has been developed.

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

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 nested deeply structures, chunking creates structures of fixed depth (typically depth 2), and as fine-grained as possible, as shown below:

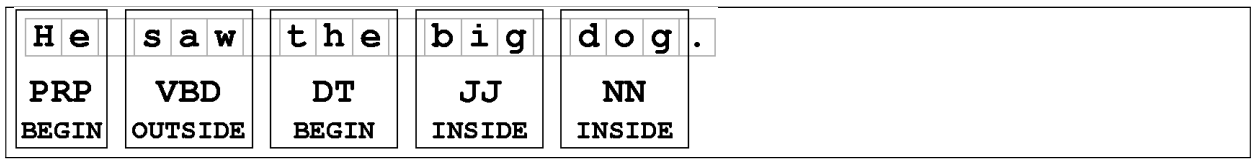
1. $[_{NP}$ $[_{NP}$ G.K. Chesterton], $[_{NP}$ $[_{NP}$ author] of $[_{NP}$ $[_{NP}$ The Man] who was $[_{NP}$ Thursday]]]]
2. $[_{NP}$ G.K. Chesterton], $[_{NP}$ author] of $[_{NP}$ The Man] who was $[_{NP}$ Thursday]

In this chapter we explore the representation of chunks, and show how chunks can be recognized using a *chunk parser*. We describe a chunk parsing method based on regular expressions, then show how chunk parsers can be cascaded to create more deeply nested structures.

2. Representing Chunks: Tags vs Trees

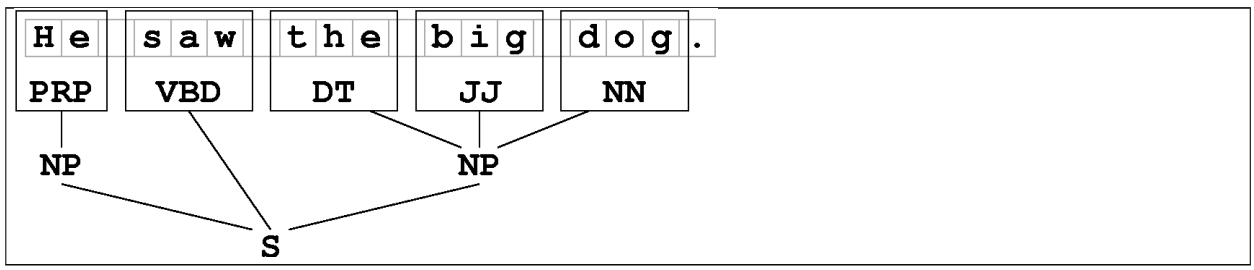
As befitting its intermediate status between tagging and parsing, chunk structures can be represented using tags or trees. The most widespread file representation uses so-called “BIO” tags. In this scheme, each token is tagged with one of three special chunk tags, `BEGIN`, `INSIDE` or `OUTSIDE`. A token is tagged as `BEGIN` if it is at the beginning of a chunk, and contained within that chunk. Subsequent tokens within the chunk are tagged `INSIDE`. All other tokens are tagged `OUTSIDE`. An example of this scheme is shown in Figure 2.

Figure 2. Tag Representation of Chunk Structures



The other logical representation for chunk structures is to use trees, as shown in Figure 3. These have the benefit that each chunk is a constituent that can be manipulated directly. NLTK uses this for its internal representation of chunks.

Figure 3. Tree Representation of Chunk Structures



A *chunk parser* finds contiguous, non-overlapping spans of related tokens and groups them together into *chunks*. The following chunk represents a simple noun phrase: `(NP: <the> <big> <dog>)`

The chunk parser combines these individual chunks together, along with the intervening tokens, to form a chunk structure. A *chunk structure* is a two-level tree that spans the entire text, and contains both chunks and un-chunked tokens. For example, the following chunk structure captures the sample noun phrases in a sentence:

```
(S: (NP: <I>)
    <saw>
    (NP: <the> <big> <dog>)
    <on>
    (NP: <the> <hill>))
```

3. Reading Chunked Text

Chunk parsers often operate on tagged texts, and use the tags to help make chunking decisions. A common string representation for chunked tagged text is illustrated below.

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

This can be parsed using the `ChunkedTaggedTokenReader` as shown.

```
>>> from nltk.tokenreader.tagged import ChunkedTaggedTokenReader
>>> chunked_string = "[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]"
>>> reader = ChunkedTaggedTokenReader(chunk_node='NP', SUBTOKENS='WORDS')
>>> sent_token = reader.read_token(chunked_string)
>>> print sent_token['TREE']
(S:
  (NP: <the/DT> <little/JJ> <cat/NN>)
  <sat/VBD>
  <on/IN>
  (NP: <the/DT> <mat/NN>))
```

We usually want to read structured data from corpora, not strings. In the remainder of this section we show how this is done for two popular chunked corpus formats.

3.1. Reading chunk structures from bracketed text

We can obtain a larger quantity of chunked text from the tagged Wall Street Journal in the Penn Treebank corpus. NLTK includes a sample of this corpus, and it can be accessed as follows:

```
>>> from nltk.corpus import treebank
>>> print treebank.groups()
('raw', 'tagged', 'parsed', 'merged')
>>> treebank.items('tagged')
('tagged/wsj_0001.pos', 'tagged/wsj_0002.pos', ...)
```

Each file in the `tagged` section of the corpus consists of chunked, tagged text. We can read in a file as follows:

```
>>> item = treebank.items('tagged')[10]
>>> tree = treebank.read(item)
<SENTS=[<TREE=(S: (NP_CHUNK: <South/NNP> <Korea/NNP>) <registered/VBD> ...
```

The result is a token with a `SENTS` property that contains a list of chunk structures. We can access the third sentence as shown below:

```
>>> tree = treebank.read(item)['SENTS'][2]['TREE']
>>> print tree
```

```
(S:
  (NP_CHUNK: <Exports/NNS>)
  <in/IN>
  (NP_CHUNK: <October/NNP>)
  <stood/VBD>
  <at/IN>
  (NP_CHUNK: <$/$> <5.29/CD> <billion/CD>)
  <,/,>
  ...
```

We can display this tree graphically using the `nltk.draw.tree` module:

```
>>> from nltk.draw.tree import *
>>> tree.draw()
```

3.2. Reading chunked text from BIO-tagged data

Using the `nltk.corpus` module we can load files that have been chunked using the BIO (`BEGIN/INSIDE/OUTSIDE`) notation, such as that provided by the evaluation competitions run by CoNLL, the *Conference on Natural Language Learning*. In the CoNLL format, each sentence is represented in a file as a multiline string, as shown below:

```
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
```

Each line consists of a word, its part-of-speech, the chunk category `B` (begin), `I` (inside) or `O` (outside), and the chunk type `NP`, `VP` or `PP`. The `ConllTokenReader` parses this information into a chunk structure. Moreover, it permits us to choose any subset of the three chunk types to use (by default it includes all three). The example in Figure 4 produces only `NP` chunks.

Figure 4. Reading NP Chunks from the CoNLL Corpus

```
>>> from nltk.tokenreader.conll import *
>>> text = """he PRP B-NP
... accepted VBD B-VP
# ...
... . . O"""
>>> reader = ConllTokenReader(chunk_types=['NP'])
>>> text_tok = reader.read_token(text)
>>> print text_tok['SENTS'][0]['TREE']
(S:
  (NP: <he/PRP>)
  <accepted/VBD>
  (NP: <the/DT> <position/NN>)
  <of/IN>
  (NP: <vice/NN> <chairman/NN>)
  <of/IN>
  (NP: <Carlyle/NNP> <Group/NNP>)
  <,/,>
  (NP: <a/DT> <merchant/NN> <banking/NN> <concern/NN>)
  <./.>)
```

This concludes our discussion of loading chunked data. In the rest of this chapter we will see how chunked data can be created from tokenized text, chunked using a chunk parser, then evaluated against the so-called gold-standard data.

4. Chunking with Regular Expressions

Earlier we noted that chunking builds flat (or non-nested) structures. In practice, the extents of text to be chunked are identified using regular expressions over sequences of part-of-speech tags. In NLTK provides a regular expression chunk parser, `RegexpChunkParser` to define the kinds of chunk we are interested in, and then to chunk a tagged text. This discussion presumes that readers are already familiar with regular expressions.

`RegexpChunkParser` works by manipulating a *chunk structure*, which represents a particular chunking of the text. The chunk parser begins with a structure in which no tokens are chunked. Each regular-expression pattern (or *chunk rule*) is applied in turn, successively updating the chunk structure. Once all of the rules have been applied, the resulting chunk structure is stored in the specified property of the token.

Chunk rules are defined in terms of regular expression patterns over "tag strings." A *tag string* is a string consisting of tags, delimited with angle-brackets, e.g.: `<DT><JJ><NN><VBD><DT><NN>`. (Note that tag strings do not contain any whitespace.) Chunk rules are defined using a special kind of regular expression pattern, called a *tag pattern*. Tag patterns are identical to the regular

expression patterns we have already seen, except for three differences which make them easier to use for chunk parsing. First, The angle brackets group their contents into atomic units, so "`<NN>+`" matches one or more repetitions of "`<NN>`"; and "`<NN|JJ>`" matches "`<NN>`" or "`<JJ>`." Second, the period wildcard operator is constrained not to cross tag boundaries, so that "`<NN.*>`" matches any single tag starting with "`NN`." Finally, whitespace is ignored in tag patterns, so "`<DT> | <NN>`" is equivalent to "`<DT>|<NN>`". This allows tag patterns to be formatted to improve readability.

Now that we can define tag patterns, it is a straightforward matter to set up an NLTK chunk parser. The simplest type of rule is `ChunkRule`. This chunks anything that matches a given tag pattern. The `ChunkRule` constructor takes a tag pattern and a description string. Here is a rule which chunks sequences consisting of one or more words tagged as `DT` or `NN` (i.e. determiners and nouns).

```
>>> from nltk.parser.chunk import *
>>> rule = ChunkRule('<DT|NN>+',
...                  'Chunk sequences of DT and NN')
```

Now we can define a `RegexpChunkParser` based on this rule as follows:

```
# Construct a new noun-phrase chunk parser
>>> chunkparser = RegexpChunkParser([rule], chunk_node='NP', top_node='S')
```

Note that the constructor has optional second and third arguments that specify the node labels for chunks and for the top-level node, respectively. Now we can use this to chunk a tagged sentence, as illustrated by the complete program in Figure 5. [1]

Figure 5. Simple Chunking in NLTK

```
>>> from nltk.tokenreader.tagged import *
>>> from nltk.parser.chunk import *

>>> sent = "the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN"
>>> reader = TaggedTokenReader(SUBTOKENS='WORDS')
>>> sent_token = reader.read_token(sent)

>>> rule = ChunkRule('<NN|DT>+',
...                  'Chunk sequences of NN and DT')
>>> parser = RegexpChunkParser([rule], SUBTOKENS='WORDS',
...                            chunk_node='NP', top_node='S')
>>> parser.parse(sent_token)
>>> print sent_token['TREE']
(S:
  (NP: <the/DT>)
  <little/JJ>
```

1. Each rule must have a "description" associated with it, which provides a short explanation of the purpose or the effect of the rule. This description is accessed via the `descr()` method.

```
  (NP: <cat/NN>)
  <sat/VBD>
  <on/IN>
  (NP: <the/DT> <mat/NN>))
```

We can also use more complex tag patterns, such as `<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.*`, followed by a single noun of any type `NN.*`.

If a tag pattern matches at multiple 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 the first two nouns will be chunked:

```
>>> from nltk.tagger import *
>>> from nltk.tokenreader import *
>>> text = "dog/NN cat/NN mouse/NN"
>>> nouns = TaggedTokenReader(SUBTOKENS='WORDS').read_token(text)
>>> rule = ChunkRule('<NN><NN>',
...                  'Chunk two consecutive nouns')
>>> parser = RegexpChunkParser([rule], chunk_node='NP',
...            top_node='S', SUBTOKENS='WORDS')
>>> parser.parse(nouns)
>>> print nouns['TREE']
(S: (NP: <dog> <cat>) <mouse>)
```

5. Developing Chunk Parsers

Creating a good chunk parser usually requires several develop and test cycles, during which existing rules are refined and new rules are added. In a later section we will describe an automatic evaluation process. Here we show how to trace the execution of a chunk parser, to help the developer diagnose any problems.

The `RegexpChunkParser` constructor takes an optional "`trace`" argument, which specifies whether debugging output should be shown during parsing. This output shows the rules that are applied, and shows the chunking hypothesis at each stage of processing. [2] In the execution trace, chunks are indicated by braces. In the following example, two chunking rules 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`.

```
>>> sent = "the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN"
>>> reader = TaggedTokenReader(SUBTOKENS='WORDS')
```

2. Note that the `parse` method can also be given a trace value; this overrides the value given to the constructor.

```
>>> sent_token = reader.read_token(sent)

>>> rule1 = ChunkRule('<DT><JJ><NN>', 'Chunk det+adj+noun')
>>> rule2 = ChunkRule('<DT|NN>+', 'Chunk sequences of NN and DT')
>>> chunkparser = RegexpChunkParser([rule1, rule2], chunk_node='NP',
...                 top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(sent_token, 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>}
>>> print sent_token['TREE']
(S: (NP: <the/DT> <little/JJ> <cat/NN>)
      <sat/VBD> <on/IN>
      (NP: <the/DT> <mat/NN>))
```

When a `ChunkRule` is applied to a chunking hypothesis, it will only create chunks that do not partially overlap with chunks already in the hypothesis. Thus, if we apply these two rules in reverse order, we will get a different result:

```
>>> chunkparser = RegexpChunkParser([rule2, rule1], chunk_node='NP',
...                 top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(sent_token, 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>}
>>> print sent_token['TREE']
(S: (NP: <the/DT>)
      <little/JJ>
      (NP: <cat/NN>)
      <sat/VBD> <on/IN>
      (NP: <the/DT> <mat/NN>))
```

Here, rule 2 ("chunk det+adj+noun") did not find any chunks, since all chunks that matched its tag pattern overlapped with chunks that were already in the hypothesis.

6. More Chunking Rules

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 chunk parser using `UnChunkRule` and `ChinkRule`.

6.1. The Chink Rule

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 the following table:

	Chink an entire chunk	**Chink the middle of a chunk**	**Chink the 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 "a/DT big/JJ cat/NN"	Chink "big/JJ"	Chink "cat/DT"
Output	a/DT big/JJ cat/NN	[a/DT] big/JJ [cat/NN]	[a/DT big/JJ] cat/NN

A `ChinkRule` chinks anything that matches a given tag pattern. `ChinkRules` are created with the `ChinkRule` constructor, which takes a tag pattern and a description string. For example, the following rule will chink any sequence of tokens whose tags are all "`VBD`" or "`IN`":

```
>>> chink_rule = ChinkRule('<VBD|IN>+',
...                        'Chink sequences of VBD and IN')
```

Remember that `RegexpChunkParser` begins with a chunking hypothesis where nothing is chunked. So before we apply our chink rule, we'll apply another rule that puts the entire sentence in a single chunk:

```
>>> chunkall_rule = ChunkRule('<.*>+',
...                           'Chunk everything')
```

Finally, we can combine these two rules to create a chunk parser:

```
>>> chunkparser = RegexpChunkParser([chunkall_rule, chink_rule],
...                 chunk_node='NP', top_node='S', SUBTOKENS='WORDS')
```

```
>>> chunkparser.parse(sent_token, trace=1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk everything:
               {<DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>}
Chink sequences of VBD and IN:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}

>>> print sent_token['TREE']
(S: (NP: <the/DT> <little/JJ> <cat/NN>)
      <sat/VBD> <on/IN>
      (NP: <the/DT> <mat/NN>))
```

If a tag pattern matches at multiple overlapping locations, the first match takes precedence.

> **Note:** Cascading Rules: `RegexpChunkParsers` can use any number of `ChunkRules` and `ChinkRules`, in any order. As was discussed in Section 4, `ChunkRules` only create chunks that do not partially overlap with chunks already in the chunking hypothesis. Similarly, `ChinkRules` only create chinks that do not partially overlap with chinks that are already in the hypothesis.

6.2. The Unchunk Rule

An `UnChunkRule` removes any chunk that matches a given tag pattern. `UnChunkRule` is very similar to `ChinkRule`; but it will only remove a chunk if the tag pattern matches the entire chunk. In contrast, `ChinkRule` can remove sequences of tokens from the middle of a chunk.

`UnChunkRules` are created with the `UnChunkRule` constructor, which takes a tag pattern and a description string. For example, the following rule will unchunk any sequence of tokens whose tags are all "`NN`" or "`DT`":

```
>>> unchunk_rule = UnChunkRule('<NN|DT>+',
...                            'Unchunk sequences of NN and DT')
```

We can combine this rule with a chunking rule to form a chunk parser:

```
>>> chunk_rule = ChunkRule('<NN|DT|JJ>+',
...                        'Chunk sequences of NN, JJ, and DT')
>>> chunkparser = RegexpChunkParser([chunk_rule, unchunk_rule],
...                 chunk_node='NP', top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(sent_token, trace=1)
Input:
               <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
```

```
Chunk sequences of NN, JJ, and DT:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
Unchunk sequences of NN and DT:
           {<DT>  <JJ>  <NN>} <VBD>  <IN>  <DT>  <NN>
>>> print sent_token['TREE']
(S: (NP: <the/DT> <little/JJ> <cat/NN>)
    <sat/VBD>
    <on/IN>
    <the/DT>
    <mat/NN>)
```

Note that we would get a different result if we used a `ChinkRule` with the same tag pattern (instead of an `UnChunkRule`), since `ChinkRule`s can remove pieces of a chunk:

```
>>> unchink_rule = UnChunkRule('<NN|DT>+',
...                            'Chink sequences of NN and DT')
>>> chunk_rule = ChunkRule('<NN|DT|JJ>+',
...                        'Chunk sequences of NN, JJ, and DT')
>>> chunkparser = RegexpChunkParser([chunk_rule, chink_rule],
...                 chunk_node='NP', top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(sent_token, trace=1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk sequences of NN, JJ, and DT:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
Chink sequences of NN and DT:
                <DT> {<JJ>} <NN>  <VBD>  <IN>  <DT>  <NN>
>>> print sent_token['TREE']
(S: <the/DT>
    (NP: <little/JJ>)
    <cat/NN>
    <sat/VBD>
    <on/IN>
    <the/DT>
    <mat/NN>)
```

6.3. The Merge Rule

When constructing complex chunk parsers, it is often convenient to perform operations other than chunking, chinking, and unchunking. In this section and the next, we discuss two more complex rules which can be used to merge and split chunks.

`MergeRule`s are used to merge two contiguous chunks. Each `MergeRule` is parameterized by two tag patterns: a *left pattern* and a *right pattern*. A `MergeRule` will merge two contiguous chunks C_1

and C_2 if the end of C_1 matches the left pattern, and the beginning of C_2 matches the right pattern. For example, consider the following chunking hypothesis:

```
[the/DT little/JJ] [cat/NN]
```

Where C_1 is `[the/DT little/JJ]` and C_2 is `[cat/NN]`. If the left pattern is "`JJ`", and the right pattern is "`NN`", then C_1 and C_2 will be merged to form a single chunk:

```
[the/DT little/JJ cat/NN]
```

`MergeRule`s are created with the `MergeRule` constructor, which takes a left tag pattern, a right tag pattern, and a description string. For example, the following rule will merge two contiguous chunks if the first one ends in a determiner, noun or adjective; and the second one begins in a determiner, noun, or adjective:

```
>>> merge_rule = MergeRule('<NN|DT|JJ>', '<NN|DT|JJ>',
...                        'Merge NNs + DTs + JJs')
```

To illustrate this rule, we will use a combine it with a chunking rule that chunks each individual token, and an unchunking rule that unchunks verbs and prepositions:

```
>>> chunk_rule = ChunkRule('<.*>',
...                        'Chunk all individual tokens')
>>> unchunk_rule = UnChunkRule('<IN|VB.*>',
...                            'Unchunk VBs and INs')
>>> rules = [chunk_rule, unchunk_rule, merge_rule]
>>> chunkparser = RegexpChunkParser(rules, chunk_node='NP',
...                   top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(sent_token, trace=1)
Input:
                 <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk all individual tokens:
               {<DT>}{<JJ>}{<NN>}{<VBD>}{<IN>}{<DT>}{<NN>}
Unchunk VBs and INs:
               {<DT>}{<JJ>}{<NN>} <VBD>  <IN> {<DT>}{<NN>}
Merge NNs + DTs + JJs:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
>>> print sent_token['TREE']
(S: (NP: <the/DT> <little/JJ> <cat/NN>)
    <sat/VBD> <on/IN>
    (NP: <the/DT> <mat/NN>))
```

6.4. The Split Rule

`SplitRule`s are used to split a single chunk into two smaller chunks. Each `SplitRule` is parameterized by two tag patterns: a *left pattern* and a *right pattern*. A `SplitRule` will split a chunk at any point p, where the left pattern matches the chunk to the left of p, and the right pattern matches the chunk to the right of p. For example, consider the following chunking hypothesis:

```
[the/DT little/JJ cat/NN the/DT dog/NN]
```

If the left pattern is "`NN`", and the right pattern is "`DT`", then the chunk will be split in two between "`cat`" and "`dog`", to form two smaller chunks:

```
[the/DT little/JJ cat/NN] [the/DT dog/NN]
```

`SplitRule`s are created with the `SplitRule` constructor, which takes a left tag pattern, a right tag pattern, and a description string. For example, the following rule will split any chunk at a location that has "`NN`" to the left and "`DT`" to the right:

```
>>> split_rule = SplitRule('<NN>', '<DT>',
...                        'Split NN followed by DT')
```

To illustrate this rule, we will use a combine it with a chunking rule that chunks sequences of noun phrases, adjectives, and determiners:

```
# Tokenize a new test text
>>> from nltk.tokenreader import *
>>> text = 'Bob/NNP saw/VBD the/DT man/NN the/DT cat/NN chased/VBD'
>>> sent_token = TaggedTokenReader(SUBTOKENS='WORDS').read_token(text)

# Create the chunk parser
>>> chunk_rule = ChunkRule('<NN.*|DT|JJ>+',
...                           'Chunk sequences of NN, JJ, and DT')
>>> chunkparser = RegexpChunkParser([chunk_rule, split_rule],
...                    chunk_node='NP', top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(sent_token, trace=1)
Input:
                <NNP>  <VBD>  <DT>  <NN>  <DT>  <NN>  <VBD>
Chunk sequences of NN, JJ, and DT:
               {<NNP>} <VBD> {<DT>  <NN>  <DT>  <NN>} <VBD>
Split NN followed by DT:
               {<NNP>} <VBD> {<DT>  <NN>}{<DT>  <NN>} <VBD>
>>> print sent_token['TREE']
(S: (NP: <Bob/NNP>)
    <saw/VBD>
    (NP: <the/DT> <man/NN>)
    (NP: <the/DT> <cat/NN>)
    <chased/VBD>)
```

7. Evaluating Chunk Parsers

An easy way to evaluate a chunk parser is to take some already chunked text, strip off the chunks, rechunk it, and compare the result with the original chunked text. The `ChunkScore.score()` function takes the correctly chunked sentence as its first argument, and the newly chunked version as its second argument, and compares them. It reports the fraction of actual chunks that were found (recall), the fraction of hypothesized chunks that were correct (precision), and a combined score, the F-measure (the harmonic mean of precision and recall).

A number of different metrics can be used to evaluate chunk parsers. We will concentrate on a class of metrics that can be derived from two sets:

- *guessed*: The set of chunks returned by the chunk parser.
- *correct*: The correct set of chunks, as defined in the test corpus.

From these two sets, we can define five useful metrics:

- *Precision*: What percentage of guessed chunks were correct?
- *Recall*: What percentage of correct chunks were guessed?
- *F Measure*: the harmonic mean of precision and recall.
- *Missed Chunks*: What correct chunks were not guessed?
- *Incorrect Chunks*: What guessed chunks were not correct?

> **Note:** Note that these metrics do not assign any credit for chunks that are "almost" right (e.g., chunks that extend one word too long). It would be possible to design metrics that do assign partial credit for such cases, they would be more complex. We decided to keep our metrics simple, so that it is easy to understand what a given result means.

During evaluation of a chunk parser, it is useful to flatten a chunk structure into a list of tokens. Since chunk structures are just trees, we use the `leaves()` method to extract this flattened list. Note that we must always include location information when evaluating a chunk parser.

```
>>> from nltk.token import *
>>> from nltk.tokenreader.tagged import ChunkedTaggedTokenReader

>>> chunked_string = "[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]"
>>> reader = ChunkedTaggedTokenReader(chunk_node='NP', SUBTOKENS='WORDS')
>>> correct = reader.read_token(chunked_string, add_locs=True)
>>> guess = Token(WORDS=correct['TREE'].leaves())
>>> print guess
[<the/DT>@[2:5c], <little/JJ>@[9:15c], <cat/NN>@[19:22c],
<sat/VBD>@[28:31c], <on/IN>@[36:38c], <the/DT>@[44:47c], <mat/NN>@[51:54c]]
```

Now we run a chunker, and then compare the resulting chunked sentences with the originals, as follows:

```
>>> from nltk.parser.chunk import *
>>> chunkscore = ChunkScore()

>>> rule = ChunkRule('<PRP|DT|POS|JJ|CD|N.*>+', "Chunk items that often occur in NPs
>>> chunkparser = RegexpChunkParser([rule], chunk_node='NP',
...                 top_node='S', SUBTOKENS='WORDS')
>>> chunkparser.parse(guess)
>>> chunkscore.score(correct['TREE'], guess['TREE'])
>>> print chunkscore
ChunkParser score:
    Precision: 100.0%
    Recall:    100.0%
    F-Measure: 100.0%
```

`ChunkScore` is a class for scoring chunk parsers. It can be used to evaluate the output of a chunk parser, using precision, recall, f-measure, missed chunks, and incorrect chunks. It can also be used to combine the scores from the parsing of multiple texts. This is quite useful if we are parsing a text one sentence at a time. The following program listing shows a typical use of the `ChunkScore` class. In this example, `chunkparser` is being tested on each sentence from the Wall Street Journal tagged files.

```
>>> from nltk.parser.chunk import *
>>> from nltk.corpus import treebank

>>> rule = ChunkRule('<DT|JJ|NN>+', "Chunk sequences of DT, JJ, and NN")
>>> chunkparser = RegexpChunkParser([rule], chunk_node='NP',
...                 top_node='S', TAG='POS', SUBTOKENS='WORDS')

>>> chunkscore = ChunkScore()

>>> for item in treebank.items('tagged')[:10]:
...     for sent in treebank.read(item, add_locs=True)['SENTS']:
...         test_sent = Token(WORDS = sent['TREE'].leaves())
...         chunkparser.parse(test_sent)
...         chunkscore.score(sent['TREE'], test_sent['TREE'])

>>> print chunkscore
ChunkParser score:
    Precision:  43.5%
    Recall:     30.5%
    F-Measure:  35.8%
```

> **Note:** It is important to add locations to tokens (using `add_locs=True`) so that the scorer doesn't confuse tokens taken from different regions of the data.

The overall results of the evaluation can be viewed by printing the `ChunkScore`. Each evaluation metric is also returned by an accessor method: `precision()`, `recall`, `f_measure`, `missed`, and `incorrect`. The `missed` and `incorrect` methods can be especially useful when trying to improve the performance of a chunk parser:

```
from random import randint

print 'Chunks missed by the chunk parser:'
missed = chunkscore.missed()
for i in range(15):
    print missed[randint(0,len(missed)-1)].type()

print 'Incorrect chunks returned by the chunk parser:'
incorrect = chunkscore.incorrect()
for i in range(15):
    print incorrect[randint(0,len(incorrect)-1)].type()
```

When evaluating a chunker it is important to score it against the baseline performance of a very simple chunker. Perhaps the most naive chunking method is to classify every tag in the training data as to whether it occurs inside or outside a chunk more often. We can do this easily using a chunked corpus and a conditional frequency distribution as shown in Figure 6.

Figure 6. Computing Baseline Performance for Chunking

```
>>> from nltk.probability import ConditionalFreqDist
>>> from nltk.parser.chunk import *
>>> from nltk.corpus import treebank

>>> cfdist = ConditionalFreqDist()
>>> items = treebank.items('tagged')
>>> split = len(items)*9/10
>>> train, test = items[:split], items[split:]

>>> for item in train:
...     for sent in treebank.read(item)['SENTS']:
...         for t in sent['TREE']:
...             if isinstance(t, Tree):
...                 for tok in t.leaves():
...                     cfdist[tok['POS']].inc(True)
...             else:
...                 cfdist[t['POS']].inc(False)

>>> chunk_tags = [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]
```

```
>>> tag_pattern = '<' + '|'.join(chunk_tags) + '>+'
>>> print 'Chunking:', tag_pattern
Chunking: <PRP$|POS|WDT|JJ|WP|DT|$|NN|PRP|NNS|NNP|PDT|LS|CD|EX|WP$|NNPS|JJS|JJR>+

# Now, in the evaluation phase we chunk any sequence of those tags:

>>> rule = ChunkRule(tag_pattern, 'Chunk any sequence involving commonly chunked tag
>>> chunkparser = RegexpChunkParser([rule], chunk_node='NP',
...                 top_node='S', TAG='POS', SUBTOKENS='WORDS')
>>> chunkscore = ChunkScore()

>>> for item in test:
...     for sent in treebank.read(item, add_locs=True)['SENTS']:
...         test_sent = Token(WORDS = sent['TREE'].leaves())
...         chunkparser.parse(test_sent)
...         chunkscore.score(sent['TREE'], test_sent['TREE'])

>>> print chunkscore
ChunkParser score:
    Precision:  80.2%
    Recall:     81.1%
    F-Measure:  80.7%
```

8. Cascaded Chunking

A chunk parser can be used to add multiple levels of structure to a sequence of tokens. For instance, on the first pass, NP chunks could be identified. On the second pass, these chunks could be treated as atomic units, and VP chunks could be identified. First we create a tree whose leaves are tagged tokens.

> **Note:** There should be a way to use the `TreebankTokenReader` to do this work.

```
>>> from nltk.tree import Tree
>>> from nltk.parser.chunk import *

>>> the    = Token(TEXT='the', TAG='DT')
>>> little = Token(TEXT='little', TAG='JJ')
>>> cat    = Token(TEXT='cat', TAG='NN')
>>> sat    = Token(TEXT='sat', TAG='VBD')
>>> on     = Token(TEXT='on', TAG='IN')
>>> the    = Token(TEXT='the', TAG='DT')
>>> mat    = Token(TEXT='mat', TAG='NN')
```

```
>>> np2 = Tree('NP', [the, mat])
>>> vp = Tree('VP', [sat, on, np2])
>>> np1 = Tree('NP', [the, little, cat])
>>> s = Tree('S', [np1, vp])
>>> print s
(S:
  (NP: <the/DT> <little/JJ> <cat/NN>)
  (VP: <sat/VBD> <on/IN> (NP: <the/DT> <mat/NN>)))
```

Now we can extract the leaves of this tree, and run two chunk parsers over them, first to find NP chunks, then to find VP chunks. Observe that the first chunk parser puts its result tree in a property called `NP-CHUNKS`, while the second chunk parser uses the contents of the `NP-CHUNKS` property as input.

```
# find NP chunks
>>> rule = ChunkRule(r'<DT>?<JJ>*<NN.*>', 'Chunk NPs')
>>> parser = RegexpChunkParser([rule], chunk_node='NP',
...            top_node='S', TREE='NP-CHUNKS', SUBTOKENS='WORDS')
>>> text_tok = Token(WORDS=s.leaves())
>>> parser.parse(text_tok)

# find VP chunks
>>> rule = ChunkRule(r'<VB.*><.*>*', 'Chunk VPs')
>>> parser = RegexpChunkParser([rule], chunk_node='VP',
...            top_node='S', SUBTOKENS='NP-CHUNKS')
>>> parser.parse( text_tok )
>>> print text_tok['TREE']
(S:
  (NP: <the/DT> <little/JJ> <cat/NN>)
  (VP: <sat/VBD> <on/IN> (NP: <the/DT> <mat/NN>)))
```

9. Conclusion

In this chapter we have explored a robust method for identifying structure in text using chunk parsers. There are a surprising number of different ways to chunk a sentence. The chunk rules can add, shift and remove chunk delimiters in many ways, and the chunk rules can be combined 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, or train up a brute-force method using existing chunked text.

- `ChunkRule` chunks anything that matches a given tag pattern.
- `ChinkRule` chinks anything that matches a given tag pattern.
- `UnChunkRule` will un-chunk any chunk that matches a given tag pattern.

- `MergeRule` can be used to merge two contiguous chunks.
- `SplitRule` can be used to split a single chunk into two smaller chunks.

10. Further Reading

11. Exercises

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

   ```
   >>> from nltk.parser.chunk import *
   >>> demo()  # the chunk parser
   ```

2. **BIO Tagging.** A common file representation of chunks uses the tags `BEGIN`, `INSIDE` and `OUTSIDE`. Why are three tags necessary? What problem would be caused if we used `INSIDE` and `OUTSIDE` tags exclusively?
3. **Predicate structure.** Develop an NP chunker which converts POS-tagged text into a list of tuples, where each tuple is a verb followed by its arguments. E.g. `the little cat sat on the mat` becomes `('sat', 'on', 'NP')`...
4. **NP Chunker.** Develop a noun phrase chunker for the CoNLL corpus using the regular-expression chunk parser `RegexpChunkParser`. Use any combination of rules.
   a. Evaluate your chunker on 100 sentences from a chunked corpus, and report the precision, recall and F-measure.
   b. Use the `chunkscore.missed()` and `chunkscore.incorrect()` methods to identify the errors made by your chunker. Discuss.

Index