5. Chunk Parsing

5.1 Introduction

Chunk parsing is an efficient and robust approach to parsing natural language, and a popular alternative to the full parsing that we will see in later chapters. Chunks are non-overlapping regions of text, usually consisting of a head word (such as a noun) and the adjacent modifiers and function words (such as adjectives and determiners).

5.1.1 Motivation

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.

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 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`**:

```
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`**. This information can then be used in the development of a grammar.

5.1.2 Chunking: Analogy with Tokenization and Tagging

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 the following diagram. The solid boxes show word-level segmentation and labelling, while the dashed boxes show a higher-level segmentation and labelling. These larger pieces are called *chunks*, and the process of identifying them is called *chunking*, *chunk parsing*, *partial parsing*, or *light parsing*.

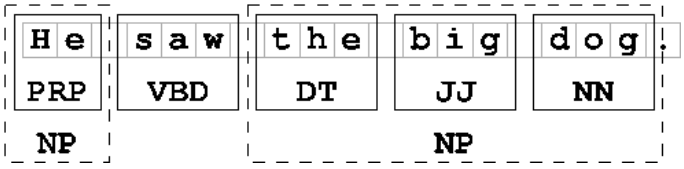


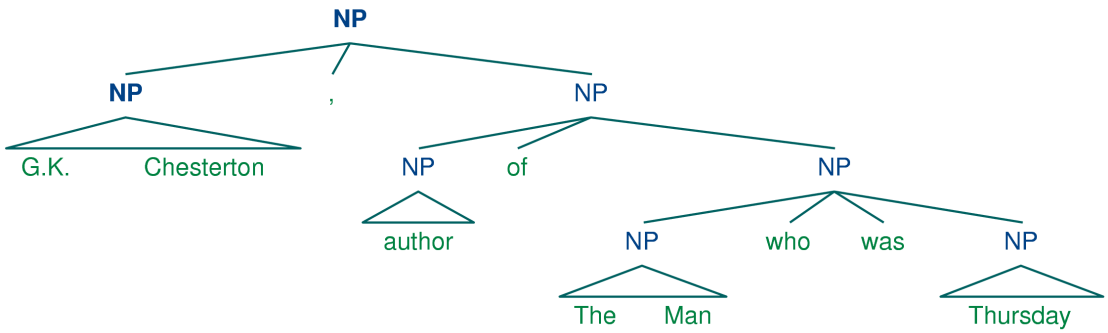
Figure 1: Segmentation and Labelling at both the Token and Chunk Levels

Chunking is like tokenization and tagging in other respects. First, chunking can skip over material in the input. Observe 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 regular-expression based methods (also known as *finite-state methods*) to identify material of interest. For example, the chunk in the above diagram 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.

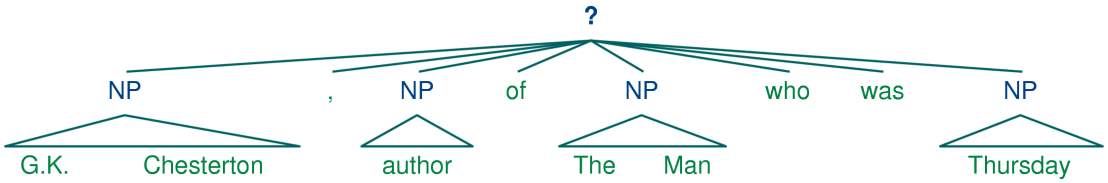
5.1.3 Chunking vs Parsing

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). Typically, these chunks correspond to the lowest level of grouping identified in the full parse tree. These differences are illustrated in (1) below:

(1a)



(1b)



Another significant motivation for chunk parsing is its robustness and efficiency relative to so-called full parsing. The latter approach is built around recursive phrase structure grammars, parsing strategies, and arbitrary-depth trees. Full parsing has problems with robustness, given the difficulty in

getting broad coverage and in resolving ambiguity. Full 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 is linear in the length of the sentence (i.e. full parsing is an O(n^3) problem, while chunking is only an O(n) problem.)

Like tagging, chunking is an example of lightweight methodology in natural language processing: how far can we get with identifying linguistic structures (such as phrases, verb arguments, etc) with recourse only to local, surface context. Also like tagging, chunking cannot be done perfectly. For example, as pointed out by Abney (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, or whether *off* is part of the verb-particle construction *turn off*. This structural ambiguity is shown in (2).

(2a) Prepositional phrase: [I] [turned] [off the spectroroute]

(2b) Verb-particle construction: [I] [turned off] [the spectroroute]

However, we continue undeterred, to find out just how far we can get with this lightweight parsing method. We begin by considering the representation of chunks and the available annotated corpora. We then show how chunks can be recognized using a chunk parser based on matching regular expressions over sequences of part-of-speech tags.

5.2 Accessing Chunked Corpora

5.2.1 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, **INSIDE**, **OUTSIDE**, or **BEGIN**. 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 below:

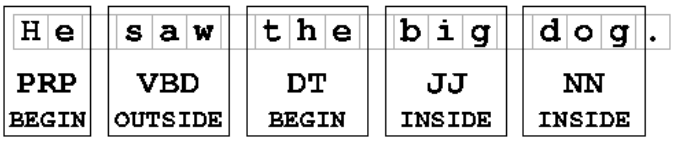


Figure 2: Tag Representation of Chunk Structures

The other obvious representation for chunk structures is to use trees, as shown below. These have the benefit that each chunk is a constituent that can be manipulated directly. NLTK-Lite uses this latter method for its internal representation of chunks:

A *chunk parser* finds contiguous, non-overlapping spans of related tokens and groups them together into *chunks*. 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 noun phrases in a sentence:

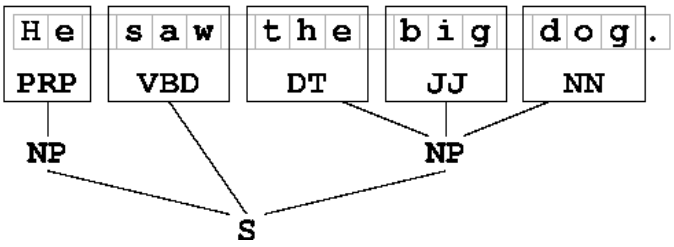


Figure 3: Tree Representation of Chunk Structures

(S: (NP: 'I')
'saw'
(NP: 'the' 'big' 'dog')
'on'
(NP: 'the' 'hill'))

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 **`tree.chunk()`** function as shown.

```
>>> from nltk_lite.parse import tree
>>> tree.chunk("[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]")
(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 following sections we show how this is done for two popular chunked corpus formats, namely bracketed text and IOB-tagged data.

5.2.2 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-Lite includes a sample of this corpus. The tagged section of the corpus consists of chunked, tagged text, such as the annotated sentence shown below:

```
| In/IN
| [ happier/JJR news/NN ]
| ,/,
| [ South/NNP Korea/NNP ]
| ,/, in/IN establishing/VBG
| [ diplomatic/JJ ties/NNS ]
| with/IN
| [ Poland/NNP yesterday/NN ]
| ,/, announced/VBD
| [ $/$ 450/CD million/CD ]
```

```
| in/IN
| [ loans/NNS ]
| to/TO
| [ the/DT ]
| financially/RB strapped/VBN
| [ Warsaw/NNP government/NN ]
| ./.
```

We can read in a chunked sentence as follows:

```
>>> from nltk_lite.corpora import treebank, extract
>>> chunk_tree = extract(603, treebank.chunked())
>>> print chunk_tree
(S:
  ('In', 'IN')
  (NP: ('happier', 'JJR') ('news', 'NN'))
  (',', ',')
  (NP: ('South', 'NNP') ('Korea', 'NNP'))
  (',', ',')
  ('in', 'IN')
  ('establishing', 'VBG')
  (NP: ('diplomatic', 'JJ') ('ties', 'NNS'))
  ('with', 'IN')
  (NP: ('Poland', 'NNP') ('yesterday', 'NN'))
  (',', ',')
  ('announced', 'VBD')
  (NP: ('$', '$') ('450', 'CD') ('million', 'CD'))
  ('in', 'IN')
  (NP: ('loans', 'NNS'))
  ('to', 'TO')
  (NP: ('the', 'DT'))
  ('financially', 'RB')
  ('strapped', 'VBN')
  (NP: ('Warsaw', 'NNP') ('government', 'NN'))
  ('.', '.'))
```

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

```
from nltk_lite.draw.tree import *
chunk_tree.draw()
```

5.2.3 Reading chunked text from IOB-tagged data

Using the **`nltk_lite.corpora`** module we can load files that have been chunked using the IOB (**`INSIDE/OUTSIDE/BEGIN`**) 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 multi-line string, as shown below:

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

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 CoNLL chunk reader **parse.conll_chunk()** 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 below produces only **NP** chunks:

```
>>> from nltk_lite import parse
>>> 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
... '''
>>> print parse.conll_chunk(text)
(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'))
  ('.', '.'))
```

We can load the CoNLL 2000 corpus using the CoNLL corpus reader **corpora.conll2000.chunked()**.

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.

5.2.4 Exercises

1. **IOB 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?

2. **CoNLL 2000 Corpus:** In this section we saw how chunked data could be read from the Treebank corpus. Write a similar program to access the first sentence of the CoNLL 2000 corpus. You will need to import the `conll2000` module from the `nltk_lite.corpora` package.

3. **Format Conversion:** We have seen two file formats for chunk data, and NLTK-Lite provides corpus readers for both.

   a) Write functions `chunk2brackets()` and `chunk2iob()` which take a single chunk structure 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 `open(filename).readlines()` to access it from Python.)

5.3 Chunk Parsing

5.3.1 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. NLTK-Lite provides a regular expression chunk parser, `parse.RegexpChunk` to define the kinds of chunk we are interested in, and then to chunk a tagged text.

`RegexpChunk` 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 returned.

In order to define chunk rules, we first need to introduce the notion of "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.) We can now create a special kind of regular expression pattern over tag strings, called a *tag pattern*. An example of a tag pattern is `<DT><JJ>?<NN>`, which matches a determiner followed by an optional adjective, followed by a noun. Tag patterns are similar 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 the tag string "`<NN>`"; and "`<NN|JJ>`" matches the tag strings "`<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`."

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

```
>>> rule = parse.ChunkRule('<DT|NN>+',
...                        'Chunk sequences of DT and NN')
```

Now we can define a regular expression chunk parser based on this rule as follows:

```
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
```

Note that **RegexpChunk** 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 below.

Observe that the object being parsed is actually a tree. It consists of an **S** root node which dominates all the leaf nodes.

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:

```
>>> ttoks = string2tags("dog/NN cat/NN mouse/NN")
>>> sent = Tree('S', ttoks)
>>> rule = parse.ChunkRule('<NN><NN>', 'Chunk two consecutive nouns')
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> print chunkparser.parse(sent)
(S: (NP: ('dog', 'NN') ('cat', 'NN')) ('mouse', 'NN'))
```

Note

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

5.3.2 Developing Chunk Parsers

Creating a good chunk parser usually requires several iterations of development and testing, during which existing rules are refined and new rules are added. In a later section we will describe an automatic evaluation method that can be used to support this development process. Here we show how to trace the execution of a chunk parser, to help the developer diagnose any problems.

RegexpChunk has 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. 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**.

```
>>> ttoks = string2tags("the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN")
>>> sent = Tree('S', ttoks)
>>> rule1 = parse.ChunkRule('<DT><JJ><NN>', 'Chunk det+adj+noun')
>>> rule2 = parse.ChunkRule('<DT|NN>+', 'Chunk sequences of NN and DT')
>>> chunkparser = parse.RegexpChunk([rule1, rule2], chunk_node='NP')
>>> chunk_tree = chunkparser.parse(sent, 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>}
```

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 = parse.RegexpChunk([rule2, rule1], chunk_node='NP')
>>> chunk_tree = chunkparser.parse(sent, 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>}
```

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.

5.3.3 The Chink Rule

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 **ChinkRule**.

The word *chink* initially meant a sequence of stopwords, according to a 1975 paper by Ross and Tukey (cited by Abney in the recommended reading for this chapter). 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 the following table:

Chinking			
	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 "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. For example, the following rule will chink any sequence of tokens whose tags are all "**VBD**" or "**IN**":

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

Before we apply our chink rule, we'll apply a rule that puts the entire sentence in a single chunk:

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

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

```
>>> chunkparser = parse.RegexpChunk([chunkall_rule, chink_rule], chunk_node='NP')
>>> chunk_tree = chunkparser.parse(sent, 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>}
```

RegexpChunk can use any number of **ChunkRule**s and **ChinkRule**s, in any order. NLTK also provides a method for merging adjacent chunks, called **MergeRule**, and a method for splitting a chunk in two, called **SplitRule**.

5.3.4 Exercises

1. **Simple Chunker:** 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 **ChunkRule** and the regular-expression chunk parser **RegexpChunk**. Discuss any tag sequences that are difficult to chunk reliably.

2. **Automatic Analysis:** 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.

3. **Chinking:** 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 using chink rules. 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.

4. **Complex Chunker:** Develop a chunker for one of the chunk types in the CoNLL corpus using the regular-expression chunk parser **RegexpChunk**. Use any combination of rules (i.e. **ChunkRule**, **ChinkRule**, **MergeRule**, and **SplitRule**).

5.4 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.

The evaluation method we will use comes from the field of information retrieval, and considers the performance of a document retrieval system. We will set up an analogy between the correct set of chunks and a user's so-called "information need", and between the set of returned chunks and a system's returned documents. Consider the following diagram.

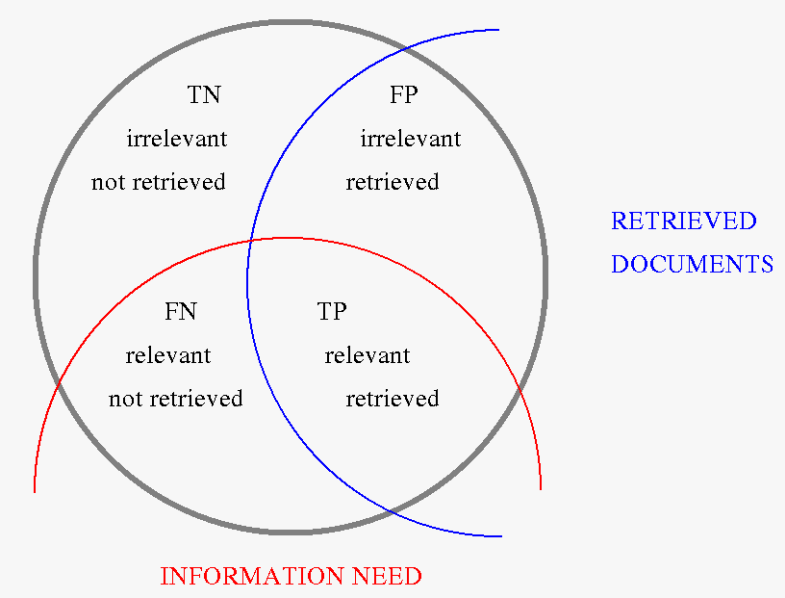


Figure 4: True and False Positives and Negatives

The intersection of these sets defines four regions: the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Two standard measures are *precision*, the fraction of guessed chunks that were correct TP/(TP+FP), and *recall*, the fraction of correct chunks that were identified TP/(TP+FN). A third measure, the *F measure*, is the harmonic mean of precision and recall, i.e. 1/(0.5/Precision + 0.5/Recall).

During evaluation of a chunk parser, it is useful to flatten a chunk structure into a tree consisting only of a root node and leaves:

```
>>> correct = tree.chunk(
...     "[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]")
>>> correct.flatten()
(S: ('the', 'DT') ('little', 'JJ') ('cat', 'NN') ('sat', 'VBD')
('on', 'IN') ('the', 'DT') ('mat', 'NN'))
```

We run a chunker over this flattened data, and compare the resulting chunked sentences with the originals, as follows:

```
>>> from nltk_lite import parse
>>> chunkscore = parse.ChunkScore()
>>> rule = parse.ChunkRule('<PRP|DT|POS|JJ|CD|N.*>+',
...              "Chunk items that often occur in NPs")
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> guess = chunkparser.parse(correct.flatten())
>>> chunkscore.score(correct, guess)
>>> print chunkscore
ChunkParse 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 itertools import islice
>>> rule = parse.ChunkRule('<DT|JJ|NN>+', "Chunk sequences of DT, JJ, and NN")
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> chunkscore = parse.ChunkScore()
>>> for chunk_struct in islice(treebank.chunked(), 10):
...     test_sent = chunkparser.parse(chunk_struct.flatten())
...     chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
    Precision:  48.6%
    Recall:     34.0%
    F-Measure:  40.0%
```

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. Here are the missed chunks:

```
>>> from random import shuffle
>>> missed = chunkscore.missed()
>>> shuffle(missed)
>>> print missed[:10]
[(('A', 'DT'), ('Lorillard', 'NNP'), ('spokeswoman', 'NN')),
 (('even', 'RB'), ('brief', 'JJ'), ('exposures', 'NNS')),
 (('its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS')),
 (('30', 'CD'), ('years', 'NNS')),
 (('workers', 'NNS'),),
 (('preliminary', 'JJ'), ('findings', 'NNS')),
 (('Medicine', 'NNP'),),
 (('Consolidated', 'NNP'), ('Gold', 'NNP'), ('Fields', 'NNP'), ('PLC', 'NNP')),
 (('its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS')),
 (('researchers', 'NNS'),)]
```

Here are the incorrect chunks:

```
>>> incorrect = chunkscore.incorrect()
>>> shuffle(incorrect)
>> print incorrect[:10]
[(('New', 'JJ'), ('York-based', 'JJ')),
 (('Micronite', 'NN'), ('cigarette', 'NN')),
 (('a', 'DT'), ('forum', 'NN'), ('likely', 'JJ')),
 (('later', 'JJ'),),
 (('preliminary', 'JJ'),),
 (('New', 'JJ'), ('York-based', 'JJ')),
 (('resilient', 'JJ'),),
 (('group', 'NN'),),
 (('the', 'DT'),),
 (('Micronite', 'NN'), ('cigarette', 'NN'))]
```

As we saw with tagging, we need to interpret the performance scores for a chunker relative to a baseline. 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 below:

```
>>> from nltk_lite.probability import ConditionalFreqDist
>>> from nltk_lite.parse import Tree
>>> import re
>>> cfdist = ConditionalFreqDist()
>>> chunk_data = list(treebank.chunked())
>>> split = len(chunk_data)*9/10
>>> train, test = chunk_data[:split], chunk_data[split:]
>>> for chunk_struct in train:
...     for constituent in chunk_struct:
...         if isinstance(constituent, Tree):
...             for (word, tag) in constituent.leaves():
...                 cfdist[tag].inc(True)
...         else:
...             (word, tag) = constituent
...             cfdist[tag].inc(False)

>>> chunk_tags = [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]
>>> chunk_tags = [re.sub(r'(\W)', r'\\\1', tag) for tag in chunk_tags]
>>> tag_pattern = '<' + '|'.join(chunk_tags) + '>+'
>>> print 'Chunking:', tag_pattern
Chunking: <PRP\$|VBG\|NN|POS|WDT|JJ|WP|DT|\#|\$|NN|FW|PRP|NNS|NNP|LS|PDT|RBS|CD|EX|
```

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

```
>>> rule = parse.ChunkRule(tag_pattern, 'Chunk any sequence involving commonly chur
>>> chunkparser = parse.RegexpChunk([rule], chunk_node='NP')
>>> chunkscore = parse.ChunkScore()
>>> for chunk_struct in test:
...     test_sent = chunkparser.parse(chunk_struct.flatten())
...     chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
```

```
Precision:  90.7%
Recall:     94.0%
F-Measure:  92.3%
```

5.5 Cascaded Chunking

So far, our chunk structures have been relatively flat: trees consisting of tagged tokens, optionally grouped under a chunk node such as **NP**. As of NLTK-Lite version 0.6.4, it is possible to build chunk structures of arbitrary depth, simply by connecting the output of one chunker to the input of another.

First we define several chunkers, e.g. for noun phrases, prepositional phrases, verb phrases, and sentences.

```
>>> np_chunk = parse.ChunkRule(r'<DT|JJ|NN.*>+', 'Chunk sequences of DT, JJ, NN')
>>> np_parse = parse.RegexpChunk([np_chunk], chunk_node='NP')
>>> pp_chunk = parse.ChunkRule(r'<IN><NP>', 'Chunk prepositions followed by NP')
>>> pp_parse = parse.RegexpChunk([pp_chunk], chunk_node='PP')
>>> vp_chunk = parse.ChunkRule(r'<VB.*><NP|PP|S>+$', 'Chunk verbs and arguments/adj
>>> vp_parse = parse.RegexpChunk([vp_chunk], chunk_node='VP')
>>> s_chunk = parse.ChunkRule(r'<NP><VP>$', 'Chunk NP, VP')
>>> s_parse = parse.RegexpChunk([s_chunk], chunk_node='S')
>>> chunkparsers = [np_parse, pp_parse, vp_parse, s_parse, vp_parse, s_parse]
```

Next, we create some tagged data and chunk it:

```
>>> text = '''John/NNP thinks/VBZ Mary/NN saw/VBD the/DT cat/NN
...           sit/VB on/IN the/DT mat/NN'''
>>> ttoks = string2tags(text)
>>> sent = Tree('S', ttoks)
>>> for chunkparser in chunkparsers:
...     sent = chunkparser.parse(sent)
>>> print sent
(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'))))))))
```

Note

At present there is no systematic way of evaluating these cascading chunkers in NLTK.

5.6 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.

We have seen that the same light-weight methods that were successful in tagging can be applied in the recognition of simple linguistic structure. 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.

A recurring theme of this chapter has been *diagnosis*. The simplest kind is manual, when we inspect the output of a chunker and observe some undesirable behavior that we would like to fix. We have also seen three objective approaches to diagnosis. The first 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. The second approach is to perform error analysis on the missed and incorrect chunks produced by the chunk parser. Sometimes those errors can be fixed. In other cases we may observe shortcomings in the methodology itself, cases where we cannot hope to get the correct answer because the system simply does not have access to the necessary information. The third approach is to evaluate the system against some gold standard data to obtain an overall performance score; we can use this diagnostically by parameterising 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 *tune* the performance of our system. We will see this theme emerge again later in chapters dealing with other topics in natural language processing.

5.7 Further Reading

Abney, Steven (1996). Tagging and Partial Parsing. In: Ken Church, Steve Young, and Gerrit Bloothooft (eds.), *Corpus-Based Methods in Language and Speech.* Kluwer Academic Publishers, Dordrecht. **`http://www.vinartus.net/spa/95a.pdf`**

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

5.8 Exercises

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

   ```
   from nltk_lite.parse import chunk
   chunk.demo()  # the chunk parser
   ```

2. **Chunker Evaluation:** Carry out the following evaluation tasks for any of the chunkers you have developed earlier. (Note that most chunking corpora contain some internal inconsistencies, such that any reasonable rule-based approach will produce errors.)

   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.

c) Compare the performance of your chunker to the baseline chunker discussed in the evaluation section of this chapter.

3. **Baseline NP Chunker:** 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 a **`SplitRule`** that will split up these chunks. Combine this rule with the existing baseline chunker and re-evaluate it, to see if you have discovered an improved baseline.

4. **Predicate structure:** 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')`**...

5. (Advanced) **Transformation-Based Chunking:** Apply the n-gram and Brill tagging methods to IOB chunk tagging. Instead of assigning POS tags to words, here we will assign IOB tags to the POS tags. E.g. if the tag **`DT`** (determiner) often occurs at the start of a chunk, it will be tagged **`B`** (begin). Evaluate the performance of these chunking methods relative to the regular expression chunking methods covered in this chapter.

6. (Advanced) **Modularity:** 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.