CHAPTER 7

Extracting Information from Text

For any given question, it's likely that someone has written the answer down somewhere. The amount of natural language text that is available in electronic form is truly staggering, and is increasing every day. However, the complexity of natural language can make it very difficult to access the information in that text. The state of the art in NLP is still a long way from being able to build general-purpose representations of meaning from unrestricted text. If we instead focus our efforts on a limited set of questions or "entity relations," such as "where are different facilities located" or "who is employed by what company," we can make significant progress. The goal of this chapter is to answer the following questions:

1. How can we build a system that extracts structured data from unstructured text?
2. What are some robust methods for identifying the entities and relationships described in a text?
3. Which corpora are appropriate for this work, and how do we use them for training and evaluating our models?

Along the way, we'll apply techniques from the last two chapters to the problems of chunking and named entity recognition.

7.1 Information Extraction

Information comes in many shapes and sizes. One important form is **structured data**, where there is a regular and predictable organization of entities and relationships. For example, we might be interested in the relation between companies and locations. Given a particular company, we would like to be able to identify the locations where it does business; conversely, given a location, we would like to discover which companies do business in that location. If our data is in tabular form, such as the example in Table 7-1, then answering these queries is straightforward.

Table 7-1. Locations data

OrgName	LocationName
Omnicom	New York
DDB Needham	New York
Kaplan Thaler Group	New York
BBDO South	Atlanta
Georgia-Pacific	Atlanta

If this location data was stored in Python as a list of tuples (*entity, relation, entity*), then the question "Which organizations operate in Atlanta?" could be translated as follows:

```
>>> print [org for (e1, rel, e2) if rel=='IN' and e2=='Atlanta']
['BBDO South', 'Georgia-Pacific']
```

Things are more tricky if we try to get similar information out of text. For example, consider the following snippet (from `nltk.corpus.ieer`, for fileid `NYT19980315.0085`).

(1) The fourth Wells account moving to another agency is the packaged paper-products division of Georgia-Pacific Corp., which arrived at Wells only last fall. Like Hertz and the History Channel, it is also leaving for an Omnicom-owned agency, the BBDO South unit of BBDO Worldwide. BBDO South in Atlanta, which handles corporate advertising for Georgia-Pacific, will assume additional duties for brands like Angel Soft toilet tissue and Sparkle paper towels, said Ken Haldin, a spokesman for Georgia-Pacific in Atlanta.

If you read through (1), you will glean the information required to answer the example question. But how do we get a machine to understand enough about (1) to return the list ['BBDO South', 'Georgia-Pacific'] as an answer? This is obviously a much harder task. Unlike Table 7-1, (1) contains no structure that links organization names with location names.

One approach to this problem involves building a very general representation of meaning (Chapter 10). In this chapter we take a different approach, deciding in advance that we will only look for very specific kinds of information in text, such as the relation between organizations and locations. Rather than trying to use text like (1) to answer the question directly, we first convert the **unstructured data** of natural language sentences into the structured data of Table 7-1. Then we reap the benefits of powerful query tools such as SQL. This method of getting meaning from text is called **Information Extraction**.

Information Extraction has many applications, including business intelligence, resume harvesting, media analysis, sentiment detection, patent search, and email scanning. A particularly important area of current research involves the attempt to extract

structured data out of electronically available scientific literature, especially in the domain of biology and medicine.

Information Extraction Architecture

Figure 7-1 shows the architecture for a simple information extraction system. It begins by processing a document using several of the procedures discussed in Chapters 3 and 5: first, the raw text of the document is split into sentences using a sentence segmenter, and each sentence is further subdivided into words using a tokenizer. Next, each sentence is tagged with part-of-speech tags, which will prove very helpful in the next step, **named entity recognition**. In this step, we search for mentions of potentially interesting entities in each sentence. Finally, we use **relation recognition** to search for likely relations between different entities in the text.

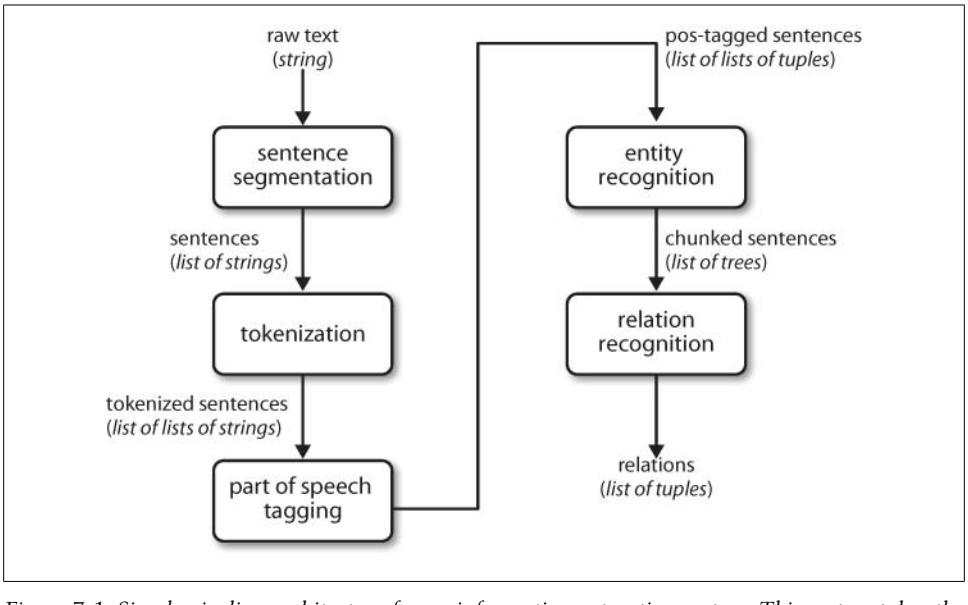


Figure 7-1. Simple pipeline architecture for an information extraction system. This system takes the raw text of a document as its input, and generates a list of (`entity`, `relation`, `entity`) tuples as its output. For example, given a document that indicates that the company Georgia-Pacific is located in Atlanta, it might generate the tuple `([ORG: 'Georgia-Pacific'] 'in' [LOC: 'Atlanta'])`.

To perform the first three tasks, we can define a function that simply connects together NLTK's default sentence segmenter ❶, word tokenizer ❷, and part-of-speech tagger ❸:

```
>>> def ie_preprocess(document):
...     sentences = nltk.sent_tokenize(document) ❶
...     sentences = [nltk.word_tokenize(sent) for sent in sentences] ❷
...     sentences = [nltk.pos_tag(sent) for sent in sentences] ❸
```

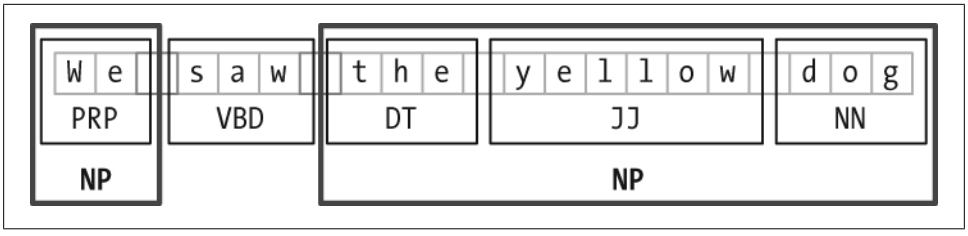


Figure 7-2. Segmentation and labeling at both the Token and Chunk levels.

Remember that our program samples assume you begin your interactive session or your program with `import nltk, re, pprint`.

Next, in named entity recognition, we segment and label the entities that might participate in interesting relations with one another. Typically, these will be definite noun phrases such as *the knights who say "ni"*, or proper names such as *Monty Python*. In some tasks it is useful to also consider indefinite nouns or noun chunks, such as *every student* or *cats*, and these do not necessarily refer to entities in the same way as definite `NP`s and proper names.

Finally, in relation extraction, we search for specific patterns between pairs of entities that occur near one another in the text, and use those patterns to build tuples recording the relationships between the entities.

7.2 Chunking

The basic technique we will use for entity recognition is **chunking**, which segments and labels multitoken sequences as illustrated in Figure 7-2. The smaller boxes show the word-level tokenization and part-of-speech tagging, while the large boxes show higher-level chunking. Each of these larger boxes is called a **chunk**. Like tokenization, which omits whitespace, chunking usually selects a subset of the tokens. Also like tokenization, the pieces produced by a chunker do not overlap in the source text.

In this section, we will explore chunking in some depth, beginning with the definition and representation of chunks. We will see regular expression and n-gram approaches to chunking, and will develop and evaluate chunkers using the CoNLL-2000 Chunking Corpus. We will then return in Sections 7.5 and 7.6 to the tasks of named entity recognition and relation extraction.

Noun Phrase Chunking

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

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

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

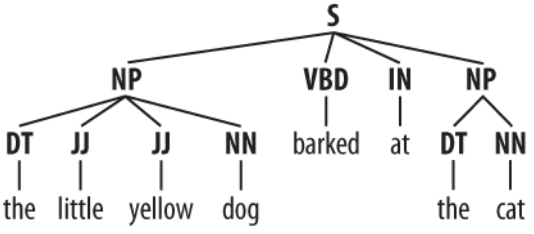
One of the most useful sources of information for NP-chunking is part-of-speech tags. This is one of the motivations for performing part-of-speech tagging in our information extraction system. We demonstrate this approach using an example sentence that has been part-of-speech tagged in Example 7-1. In order to create an NP-chunker, we will first define a **chunk grammar**, consisting of rules that indicate how sentences should be chunked. In this case, we will define a simple grammar with a single regular expression rule ❷. This rule says that an NP chunk should be formed whenever the chunker finds an optional determiner (DT) followed by any number of adjectives (JJ) and then a noun (NN). Using this grammar, we create a chunk parser ❸, and test it on our example sentence ❹. The result is a tree, which we can either print ❺, or display graphically ❻.

Example 7-1. Example of a simple regular expression–based NP chunker.

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

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

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



Tag Patterns

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

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

We can match these noun phrases using a slight refinement of the first tag pattern above, i.e., `<DT>?<JJ.*>*<NN.*>+`. This will chunk any sequence of tokens beginning with an optional determiner, followed by zero or more adjectives of any type (including relative adjectives like `earlier/JJR`), followed by one or more nouns of any type. However, it is easy to find many more complicated examples which this rule will not cover:

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

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

Chunking with Regular Expressions

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

Example 7-2 shows a simple chunk grammar consisting of two rules. The first rule matches an optional determiner or possessive pronoun, zero or more adjectives, then

a noun. The second rule matches one or more proper nouns. We also define an example sentence to be chunked ❶, and run the chunker on this input ❶.

Example 7-2. Simple noun phrase chunker.

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

>>> print cp.parse(sentence) ❶
(S
  (NP Rapunzel/NNP)
  let/VBD
  down/RP
  (NP her/PP$ long/JJ golden/JJ hair/NN))
```

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

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

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

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

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

Exploring Text Corpora

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

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

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

Chinking

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

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

Chinking is the process of removing a sequence of tokens from a chunk. If the matching sequence of tokens spans an entire chunk, then the whole chunk is removed; if the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before. If the sequence is at the periphery of the chunk, these tokens are removed, and a smaller chunk remains. These three possibilities are illustrated in Table 7-2.

Table 7-2. Three chinking rules applied to the same chunk

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

In Example 7-3, we put the entire sentence into a single chunk, then excise the chinks.

Example 7-3. Simple chinker.

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

```
>>> print cp.parse(sentence)
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
```

Representing Chunks: Tags Versus Trees

As befits their intermediate status between tagging and parsing (Chapter 8), chunk structures can be represented using either tags or trees. The most widespread file representation uses **IOB tags**. In this scheme, each token is tagged with one of three special chunk tags, `I` (inside), `O` (outside), or `B` (begin). A token is tagged as `B` if it marks the beginning of a chunk. Subsequent tokens within the chunk are tagged `I`. All other tokens are tagged `O`. The `B` and `I` tags are suffixed with the chunk type, e.g., `B-NP`, `I-NP`. Of course, it is not necessary to specify a chunk type for tokens that appear outside a chunk, so these are just labeled `O`. An example of this scheme is shown in Figure 7-3.

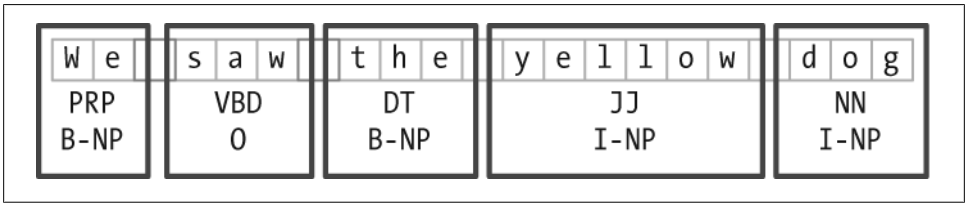


Figure 7-3. Tag representation of chunk structures.

IOB tags have become the standard way to represent chunk structures in files, and we will also be using this format. Here is how the information in Figure 7-3 would appear in a file:

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

In this representation there is one token per line, each with its part-of-speech tag and chunk tag. This format permits us to represent more than one chunk type, so long as the chunks do not overlap. As we saw earlier, chunk structures can also be represented using trees. These have the benefit that each chunk is a constituent that can be manipulated directly. An example is shown in Figure 7-4.

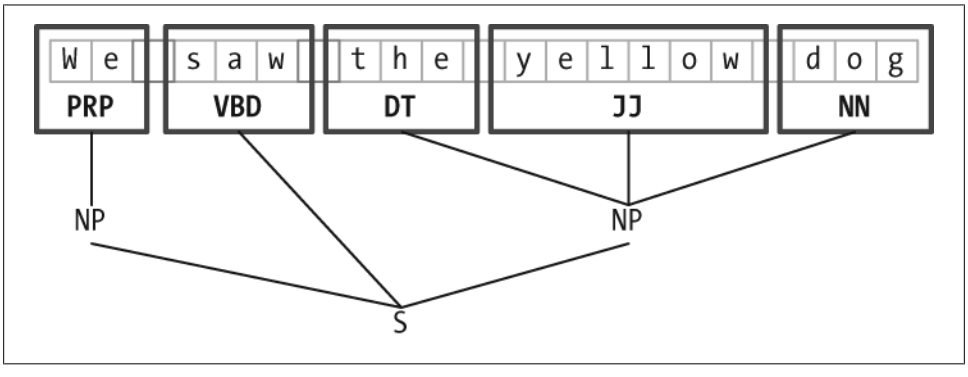


Figure 7-4. Tree representation of chunk structures.

NLTK uses trees for its internal representation of chunks, but provides methods for converting between such trees and the IOB format.

7.3 Developing and Evaluating Chunkers

Now you have a taste of what chunking does, but we haven't explained how to evaluate chunkers. As usual, this requires a suitably annotated corpus. We 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 a chunked corpus. We will see how to score the accuracy of a chunker relative to a corpus, then look at some more data-driven ways to search for NP chunks. Our focus throughout will be on expanding the coverage of a chunker.

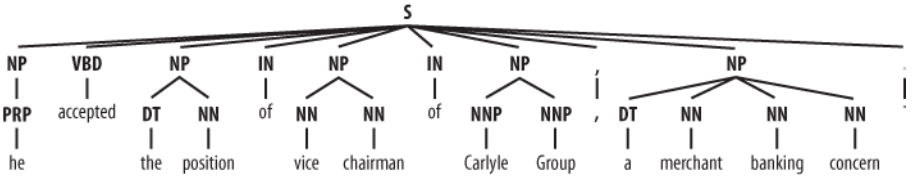
Reading IOB Format and the CoNLL-2000 Chunking Corpus

Using the `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 here:

```
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 multiline strings. Moreover, it permits us to choose any subset of the three chunk types to use, here just for `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
... '''
>>> nltk.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 Chunking Corpus contains 270k words of *Wall Street Journal* text, divided into "train" and "test" portions, annotated with part-of-speech tags and chunk tags in the IOB format. We can access the data using `nltk.corpus.conll2000`. Here is an example that reads the 100th sentence of the "train" portion of the corpus:

```
>>> from nltk.corpus import conll2000
>>> print conll2000.chunked_sents('train.txt')[99]
(S
  (PP Over/IN)
  (NP a/DT cup/NN)
  (PP of/IN)
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  (VP told/VBD)
  (NP his/PRP$ story/NN)
  ./.)
```

As you can see, the CoNLL-2000 Chunking Corpus contains three chunk types: `NP` chunks, which we have already seen; `VP` chunks, such as *has already delivered*; and `PP`

chunks, such as *because of*. Since we are only interested in the NP chunks right now, we can use the chunk_types argument to select them:

```
>>> print conll2000.chunked_sents('train.txt', chunk_types=['NP'])[99]
(S
  Over/IN
  (NP a/DT cup/NN)
  of/IN
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  told/VBD
  (NP his/PRP$ story/NN)
  ./.)
```

Simple Evaluation and Baselines

Now that we can access a chunked corpus, we can evaluate chunkers. We start off by establishing a baseline for the trivial chunk parser cp that creates no chunks:

```
>>> from nltk.corpus import conll2000
>>> cp = nltk.RegexpParser("")
>>> test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
>>> print cp.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  43.4%
    Precision:      0.0%
    Recall:         0.0%
    F-Measure:      0.0%
```

The IOB tag accuracy indicates that more than a third of the words are tagged with O, i.e., not in an NP chunk. However, since our tagger did not find *any* chunks, its precision, recall, and F-measure are all zero. Now let's try a naive regular expression chunker that looks for tags beginning with letters that are characteristic of noun phrase tags (e.g., CD, DT, and JJ).

```
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> print cp.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  87.7%
    Precision:     70.6%
    Recall:        67.8%
    F-Measure:     69.2%
```

As you can see, this approach achieves decent results. However, we can improve on it by adopting a more data-driven approach, where we use the training corpus to find the chunk tag (I, O, or B) that is most likely for each part-of-speech tag. In other words, we can build a chunker using a *unigram tagger* (Section 5.4). But rather than trying to determine the correct part-of-speech tag for each word, we are trying to determine the correct chunk tag, given each word's part-of-speech tag.

In Example 7-4, we define the `UnigramChunker` class, which uses a unigram tagger to label sentences with chunk tags. Most of the code in this class is simply used to convert back and forth between the chunk tree representation used by NLTK's `ChunkParserI` interface, and the IOB representation used by the embedded tagger. The class defines two methods: a constructor ❶, which is called when we build a new UnigramChunker; and the `parse` method ❸, which is used to chunk new sentences.

Example 7-4. Noun phrase chunking with a unigram tagger.

```
class UnigramChunker(nltk.ChunkParserI):
    def __init__(self, train_sents): ❶
        train_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(sent)]
                      for sent in train_sents]
        self.tagger = nltk.UnigramTagger(train_data) ❷

    def parse(self, sentence): ❸
        pos_tags = [pos for (word,pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
        conlltags = [(word, pos, chunktag) for ((word,pos),chunktag)
                     in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)
```

The constructor ❶ expects a list of training sentences, which will be in the form of chunk trees. It first converts training data to a form that's suitable for training the tagger, using `tree2conlltags` to map each chunk tree to a list of `word,tag,chunk` triples. It then uses that converted training data to train a unigram tagger, and stores it in `self.tagger` for later use.

The `parse` method ❸ takes a tagged sentence as its input, and begins by extracting the part-of-speech tags from that sentence. It then tags the part-of-speech tags with IOB chunk tags, using the tagger `self.tagger` that was trained in the constructor. Next, it extracts the chunk tags, and combines them with the original sentence, to yield `conlltags`. Finally, it uses `conlltags2tree` to convert the result back into a chunk tree.

Now that we have `UnigramChunker`, we can train it using the CoNLL-2000 Chunking Corpus, and test its resulting performance:

```
>>> test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
>>> train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
>>> unigram_chunker = UnigramChunker(train_sents)
>>> print unigram_chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  92.9%
    Precision:     79.9%
    Recall:        86.8%
    F-Measure:     83.2%
```

This chunker does reasonably well, achieving an overall F-measure score of 83%. Let's take a look at what it's learned, by using its unigram tagger to assign a tag to each of the part-of-speech tags that appear in the corpus:

```
>>> postags = sorted(set(pos for sent in train_sents
...                      for (word,pos) in sent.leaves()))
>>> print unigram_chunker.tagger.tag(postags)
[('#', 'B-NP'), ('$', 'B-NP'), ("''", 'O'), ('(', 'O'), (')', 'O'),
 (',', 'O'), ('.', 'O'), (':', 'O'), ('CC', 'O'), ('CD', 'I-NP'),
 ('DT', 'B-NP'), ('EX', 'B-NP'), ('FW', 'I-NP'), ('IN', 'O'),
 ('JJ', 'I-NP'), ('JJR', 'B-NP'), ('JJS', 'I-NP'), ('MD', 'O'),
 ('NN', 'I-NP'), ('NNP', 'I-NP'), ('NNPS', 'I-NP'), ('NNS', 'I-NP'),
 ('PDT', 'B-NP'), ('POS', 'B-NP'), ('PRP', 'B-NP'), ('PRP$', 'B-NP'),
 ('RB', 'O'), ('RBR', 'O'), ('RBS', 'B-NP'), ('RP', 'O'), ('SYM', 'O'),
 ('TO', 'O'), ('UH', 'O'), ('VB', 'O'), ('VBD', 'O'), ('VBG', 'O'),
 ('VBN', 'O'), ('VBP', 'O'), ('VBZ', 'O'), ('WDT', 'B-NP'),
 ('WP', 'B-NP'), ('WP$', 'B-NP'), ('WRB', 'O'), ('``', 'O')]
```

It has discovered that most punctuation marks occur outside of NP chunks, with the exception of `#` and `$`, both of which are used as currency markers. It has also found that determiners (`DT`) and possessives (`PRP$` and `WP$`) occur at the beginnings of NP chunks, while noun types (`NN`, `NNP`, `NNPS`, `NNS`) mostly occur inside of NP chunks.

Having built a unigram chunker, it is quite easy to build a bigram chunker: we simply change the class name to `BigramChunker`, and modify line ❷ in Example 7-4 to construct a `BigramTagger` rather than a `UnigramTagger`. The resulting chunker has slightly higher performance than the unigram chunker:

```
>>> bigram_chunker = BigramChunker(train_sents)
>>> print bigram_chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  93.3%
    Precision:     82.3%
    Recall:        86.8%
    F-Measure:     84.5%
```

Training Classifier-Based Chunkers

Both the regular expression–based chunkers and the n-gram chunkers decide what chunks to create entirely based on part-of-speech tags. However, sometimes part-of-speech tags are insufficient to determine how a sentence should be chunked. For example, consider the following two statements:

(3) a. Joey/NN sold/VBD the/DT farmer/NN rice/NN ./.
    b. Nick/NN broke/VBD my/DT computer/NN monitor/NN ./.

These two sentences have the same part-of-speech tags, yet they are chunked differently. In the first sentence, *the farmer* and *rice* are separate chunks, while the corresponding material in the second sentence, *the computer monitor*, is a single chunk. Clearly, we need to make use of information about the content of the words, in addition to just their part-of-speech tags, if we wish to maximize chunking performance.

One way that we can incorporate information about the content of words is to use a classifier-based tagger to chunk the sentence. Like the n-gram chunker considered in the previous section, this classifier-based chunker will work by assigning IOB tags to

the words in a sentence, and then converting those tags to chunks. For the classifier-based tagger itself, we will use the same approach that we used in Section 6.1 to build a part-of-speech tagger.

The basic code for the classifier-based NP chunker is shown in Example 7-5. It consists of two classes. The first class ❶ is almost identical to the `ConsecutivePosTagger` class from Example 6-5. The only two differences are that it calls a different feature extractor ❷ and that it uses a `MaxentClassifier` rather than a `NaiveBayesClassifier` ❸. The second class ❹ is basically a wrapper around the tagger class that turns it into a chunker. During training, this second class maps the chunk trees in the training corpus into tag sequences; in the `parse()` method, it converts the tag sequence provided by the tagger back into a chunk tree.

Example 7-5. Noun phrase chunking with a consecutive classifier.

```
class ConsecutiveNPChunkTagger(nltk.TaggerI): ❶

    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = npchunk_features(untagged_sent, i, history) ❷
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.MaxentClassifier.train( ❸
            train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)

class ConsecutiveNPChunker(nltk.ChunkParserI): ❹
    def __init__(self, train_sents):
        tagged_sents = [[((w,t),c) for (w,t,c) in
                         nltk.chunk.tree2conlltags(sent)]
                        for sent in train_sents]
        self.tagger = ConsecutiveNPChunkTagger(tagged_sents)

    def parse(self, sentence):
        tagged_sents = self.tagger.tag(sentence)
        conlltags = [(w,t,c) for ((w,t),c) in tagged_sents]
        return nltk.chunk.conlltags2tree(conlltags)
```

The only piece left to fill in is the feature extractor. We begin by defining a simple feature extractor, which just provides the part-of-speech tag of the current token. Using

this feature extractor, our classifier-based chunker is very similar to the unigram chunker, as is reflected in its performance:

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     return {"pos": pos}
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  92.9%
    Precision:     79.9%
    Recall:        86.7%
    F-Measure:     83.2%
```

We can also add a feature for the previous part-of-speech tag. Adding this feature allows the classifier to model interactions between adjacent tags, and results in a chunker that is closely related to the bigram chunker.

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         prevword, prevpos = "<START>", "<START>"
...     else:
...         prevword, prevpos = sentence[i-1]
...     return {"pos": pos, "prevpos": prevpos}
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  93.6%
    Precision:     81.9%
    Recall:        87.1%
    F-Measure:     84.4%
```

Next, we'll try adding a feature for the current word, since we hypothesized that word content should be useful for chunking. We find that this feature does indeed improve the chunker's performance, by about 1.5 percentage points (which corresponds to about a 10% reduction in the error rate).

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         prevword, prevpos = "<START>", "<START>"
...     else:
...         prevword, prevpos = sentence[i-1]
...     return {"pos": pos, "word": word, "prevpos": prevpos}
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  94.2%
    Precision:     83.4%
    Recall:        88.6%
    F-Measure:     85.9%
```

Finally, we can try extending the feature extractor with a variety of additional features, such as lookahead features ❶, paired features ❷, and complex contextual features ❸. This last feature, called `tags-since-dt`, creates a string describing the set of all part-of-speech tags that have been encountered since the most recent determiner.

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         prevword, prevpos = "<START>", "<START>"
...     else:
...         prevword, prevpos = sentence[i-1]
...     if i == len(sentence)-1:
...         nextword, nextpos = "<END>", "<END>"
...     else:
...         nextword, nextpos = sentence[i+1]
...     return {"pos": pos,
...             "word": word,
...             "prevpos": prevpos,
...             "nextpos": nextpos, ❶
...             "prevpos+pos": "%s+%s" % (prevpos, pos),  ❷
...             "pos+nextpos": "%s+%s" % (pos, nextpos),
...             "tags-since-dt": tags_since_dt(sentence, i)}  ❸

>>> def tags_since_dt(sentence, i):
...     tags = set()
...     for word, pos in sentence[:i]:
...         if pos == 'DT':
...             tags = set()
...         else:
...             tags.add(pos)
...     return '+'.join(sorted(tags))

>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  95.9%
    Precision:     88.3%
    Recall:        90.7%
    F-Measure:     89.5%
```

Your Turn: Try adding different features to the feature extractor function `npchunk_features`, and see if you can further improve the performance of the NP chunker.

7.4 Recursion in Linguistic Structure

Building Nested Structure with 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 multistage chunk grammar

containing recursive rules. Example 7-6 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.

Example 7-6. A chunker that handles NP, PP, VP, and S.

```
grammar = r"""
  NP: {<DT|JJ|NN.*>+}          # Chunk sequences of DT, JJ, NN
  PP: {<IN><NP>}               # Chunk prepositions followed by NP
  VP: {<VB.*><NP|PP|CLAUSE>+$} # Chunk verbs and their arguments
  CLAUSE: {<NP><VP>}           # Chunk NP, VP
  """
cp = nltk.RegexpParser(grammar)
sentence = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),
    ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

>>> print cp.parse(sentence)
(S
  (NP Mary/NN)
  saw/VBD
  (CLAUSE
    (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. Notice that it fails to identify the VP chunk starting at ❶.

```
>>> sentence = [("John", "NNP"), ("thinks", "VBZ"), ("Mary", "NN"),
...     ("saw", "VBD"), ("the", "DT"), ("cat", "NN"), ("sit", "VB"),
...     ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(sentence)
(S
  (NP John/NNP)
  thinks/VBZ
  (NP Mary/NN)
  saw/VBD ❶
  (CLAUSE
    (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 = nltk.RegexpParser(grammar, loop=2)
>>> print cp.parse(sentence)
(S
  (NP John/NNP)
  thinks/VBZ
  (CLAUSE
    (NP Mary/NN)
    (VP
      saw/VBD
      (CLAUSE
```

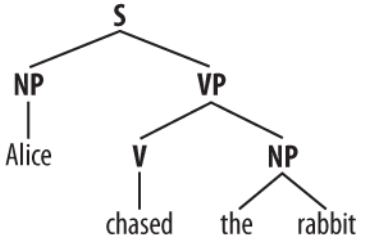
```
(NP the/DT cat/NN)
(VP sit/VB (PP on/IN (NP the/DT mat/NN)))))))
```

This cascading process enables us to create deep structures. However, creating and debugging a cascade is difficult, and there comes a point where it is more effective to do full parsing (see Chapter 8). Also, the cascading process can only produce trees of fixed depth (no deeper than the number of stages in the cascade), and this is insufficient for complete syntactic analysis.

Trees

A **tree** is a set of connected labeled nodes, each reachable by a unique path from a distinguished root node. Here's an example of a tree (note that they are standardly drawn upside-down):

(4)



We use a 'family' metaphor to talk about the relationships of nodes in a tree: for example, `S` is the **parent** of `VP`; conversely `VP` is a **child** of `S`. Also, since `NP` and `VP` are both children of `S`, they are also **siblings**. For convenience, there is also a text format for specifying trees:

```
(S
  (NP Alice)
  (VP
    (V chased)
    (NP
      (Det the)
      (N rabbit))))
```

Although we will focus on syntactic trees, trees can be used to encode *any* homogeneous hierarchical structure that spans a sequence of linguistic forms (e.g., morphological structure, discourse structure). In the general case, leaves and node values do not have to be strings.

In NLTK, we create a tree by giving a node label and a list of children:

```
>>> tree1 = nltk.Tree('NP', ['Alice'])
>>> print tree1
(NP Alice)
>>> tree2 = nltk.Tree('NP', ['the', 'rabbit'])
>>> print tree2
(NP the rabbit)
```

We can incorporate these into successively larger trees as follows:

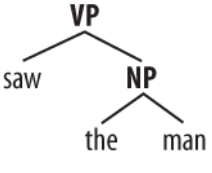
```
>>> tree3 = nltk.Tree('VP', ['chased', tree2])
>>> tree4 = nltk.Tree('S', [tree1, tree3])
>>> print tree4
(S (NP Alice) (VP chased (NP the rabbit)))
```

Here are some of the methods available for tree objects:

```
>>> print tree4[1]
(VP chased (NP the rabbit))
>>> tree4[1].node
'VP'
>>> tree4.leaves()
['Alice', 'chased', 'the', 'rabbit']
>>> tree4[1][1][1]
'rabbit'
```

The bracketed representation for complex trees can be difficult to read. In these cases, the `draw` method can be very useful. It opens a new window, containing a graphical representation of the tree. The tree display window allows you to zoom in and out, to collapse and expand subtrees, and to print the graphical representation to a postscript file (for inclusion in a document).

```
>>> tree3.draw()
```



Tree Traversal

It is standard to use a recursive function to traverse a tree. The listing in Example 7-7 demonstrates this.

Example 7-7. A recursive function to traverse a tree.

```
def traverse(t):
    try:
        t.node
    except AttributeError:
        print t,


    else:
```

```
        # Now we know that t.node is defined
        print '(', t.node,
        for child in t:
            traverse(child)
        print ')',
>>> t = nltk.Tree('(S (NP Alice) (VP chased (NP the rabbit)))')
>>> traverse(t)
( S ( NP Alice ) ( VP chased ( NP the rabbit ) ) )
```

We have used a technique called **duck typing** to detect that `t` is a tree (i.e., `t.node` is defined).

7.5 Named Entity Recognition

At the start of this chapter, we briefly introduced named entities (NEs). Named entities are definite noun phrases that refer to specific types of individuals, such as organizations, persons, dates, and so on. Table 7-3 lists some of the more commonly used types of NEs. These should be self-explanatory, except for "FACILITY": human-made artifacts in the domains of architecture and civil engineering; and "GPE": geo-political entities such as city, state/province, and country.

Table 7-3. Commonly used types of named entity

NE type	Examples
ORGANIZATION	*Georgia-Pacific Corp., WHO*
PERSON	*Eddy Bonte, President Obama*
LOCATION	*Murray River, Mount Everest*
DATE	*June, 2008-06-29*
TIME	*two fifty a m, 1:30 p.m.*
MONEY	*175 million Canadian Dollars, GBP 10.40*
PERCENT	*twenty pct, 18.75 %*
FACILITY	*Washington Monument, Stonehenge*
GPE	*South East Asia, Midlothian*

The goal of a **named entity recognition** (NER) system is to identify all textual mentions of the named entities. This can be broken down into two subtasks: identifying the boundaries of the NE, and identifying its type. While named entity recognition is frequently a prelude to identifying relations in Information Extraction, it can also contribute to other tasks. For example, in Question Answering (QA), we try to improve the precision of Information Retrieval by recovering not whole pages, but just those parts which contain an answer to the user's question. Most QA systems take the

documents returned by standard Information Retrieval, and then attempt to isolate the minimal text snippet in the document containing the answer. Now suppose the question was *Who was the first President of the US?*, and one of the documents that was retrieved contained the following passage:

(5) The Washington Monument is the most prominent structure in Washington, D.C. and one of the city's early attractions. It was built in honor of George Washington, who led the country to independence and then became its first President.

Analysis of the question leads us to expect that an answer should be of the form *X was the first President of the US*, where *X* is not only a noun phrase, but also refers to a named entity of type PER. This should allow us to ignore the first sentence in the passage. Although it contains two occurrences of *Washington*, named entity recognition should tell us that neither of them has the correct type.

How do we go about identifying named entities? One option would be to look up each word in an appropriate list of names. For example, in the case of locations, we could use a **gazetteer**, or geographical dictionary, such as the Alexandria Gazetteer or the Getty Gazetteer. However, doing this blindly runs into problems, as shown in Figure 7-5.

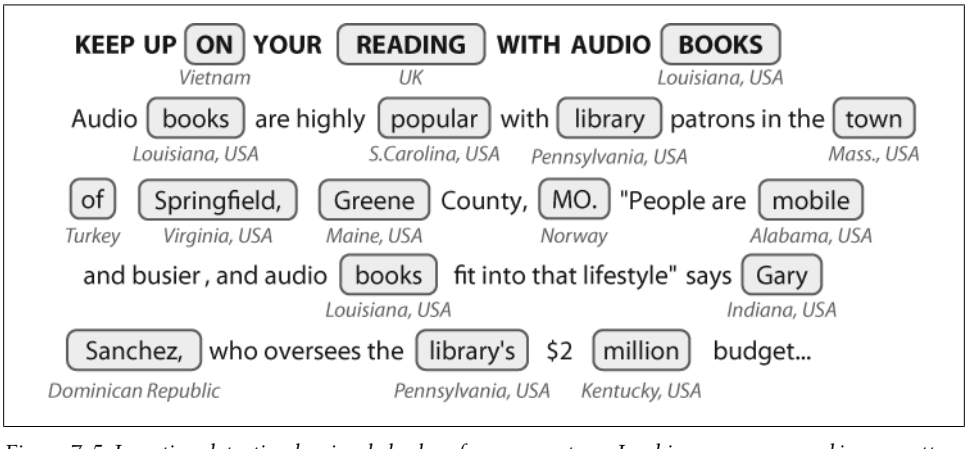


Figure 7-5. Location detection by simple lookup for a news story: Looking up every word in a gazetteer is error-prone; case distinctions may help, but these are not always present.

Observe that the gazetteer has good coverage of locations in many countries, and incorrectly finds locations like Sanchez in the Dominican Republic and On in Vietnam. Of course we could omit such locations from the gazetteer, but then we won't be able to identify them when they do appear in a document.

It gets even harder in the case of names for people or organizations. Any list of such names will probably have poor coverage. New organizations come into existence every

day, so if we are trying to deal with contemporary newswire or blog entries, it is unlikely that we will be able to recognize many of the entities using gazetteer lookup.

Another major source of difficulty is caused by the fact that many named entity terms are ambiguous. Thus *May* and *North* are likely to be parts of named entities for DATE and LOCATION, respectively, but could both be part of a PERSON; conversely *Christian Dior* looks like a PERSON but is more likely to be of type ORGANIZATION. A term like *Yankee* will be an ordinary modifier in some contexts, but will be marked as an entity of type ORGANIZATION in the phrase *Yankee infielders*.

Further challenges are posed by multiword names like *Stanford University*, and by names that contain other names, such as *Cecil H. Green Library* and *Escondido Village Conference Service Center*. In named entity recognition, therefore, we need to be able to identify the beginning and end of multitoken sequences.

Named entity recognition is a task that is well suited to the type of classifier-based approach that we saw for noun phrase chunking. In particular, we can build a tagger that labels each word in a sentence using the IOB format, where chunks are labeled by their appropriate type. Here is part of the CONLL 2002 (`conll2002`) Dutch training data:

```
Eddy N B-PER
Bonte N I-PER
is V O
woordvoerder N O
van Prep O
diezelfde Pron O
Hogeschool N B-ORG
. Punc O
```

In this representation, there is one token per line, each with its part-of-speech tag and its named entity tag. Based on this training corpus, we can construct a tagger that can be used to label new sentences, and use the `nltk.chunk.conlltags2tree()` function to convert the tag sequences into a chunk tree.

NLTK provides a classifier that has already been trained to recognize named entities, accessed with the function `nltk.ne_chunk()`. If we set the parameter `binary=True` ❶, then named entities are just tagged as `NE`; otherwise, the classifier adds category labels such as PERSON, ORGANIZATION, and GPE.

```
>>> sent = nltk.corpus.treebank.tagged_sents()[22]
>>> print nltk.ne_chunk(sent, binary=True) ❶
(S
  The/DT
  (NE U.S./NNP)
  is/VBZ
  one/CD
  ...
  according/VBG
  to/TO
  (NE Brooke/NNP T./NNP Mossman/NNP)
  ...)
```

```
>>> print nltk.ne_chunk(sent)
(S
  The/DT
  (GPE U.S./NNP)
  is/VBZ
  one/CD
  ...
  according/VBG
  to/TO
  (PERSON Brooke/NNP T./NNP Mossman/NNP)
  ...)
```

7.6 Relation Extraction

Once named entities have been identified in a text, we then want to extract the relations that exist between them. As indicated earlier, we will typically be looking for relations between specified types of named entity. One way of approaching this task is to initially look for all triples of the form (*X*, α, *Y*), where *X* and *Y* are named entities of the required types, and α is the string of words that intervenes between *X* and *Y*. We can then use regular expressions to pull out just those instances of α that express the relation that we are looking for. The following example searches for strings that contain the word *in*. The special regular expression `(?!\b.+ing\b)` is a negative lookahead assertion that allows us to disregard strings such as *success in supervising the transition of*, where *in* is followed by a gerund.

```
>>> IN = re.compile(r'.*\bin\b(?!\b.+ing)')
>>> for doc in nltk.corpus.ieer.parsed_docs('NYT_19980315'):
...     for rel in nltk.sem.extract_rels('ORG', 'LOC', doc,
...                                      corpus='ieer', pattern = IN):
...         print nltk.sem.show_raw_rtuple(rel)
[ORG: 'WHYY'] 'in' [LOC: 'Philadelphia']
[ORG: 'McGlashan &AMP; Sarrail'] 'firm in' [LOC: 'San Mateo']
[ORG: 'Freedom Forum'] 'in' [LOC: 'Arlington']
[ORG: 'Brookings Institution'] ', the research group in' [LOC: 'Washington']
[ORG: 'Idealab'] ', a self-described business incubator based in' [LOC: 'Los Angeles']
[ORG: 'Open Text'] ', based in' [LOC: 'Waterloo']
[ORG: 'WGBH'] 'in' [LOC: 'Boston']
[ORG: 'Bastille Opera'] 'in' [LOC: 'Paris']
[ORG: 'Omnicom'] 'in' [LOC: 'New York']
[ORG: 'DDB Needham'] 'in' [LOC: 'New York']
[ORG: 'Kaplan Thaler Group'] 'in' [LOC: 'New York']
[ORG: 'BBDO South'] 'in' [LOC: 'Atlanta']
[ORG: 'Georgia-Pacific'] 'in' [LOC: 'Atlanta']
```

Searching for the keyword *in* works reasonably well, though it will also retrieve false positives such as `[ORG: House Transportation Committee] , secured the most money in the [LOC: New York]`; there is unlikely to be a simple string-based method of excluding filler strings such as this.

As shown earlier, the Dutch section of the CoNLL 2002 Named Entity Corpus contains not just named entity annotation, but also part-of-speech tags. This allows us to devise patterns that are sensitive to these tags, as shown in the next example. The method `show_clause()` prints out the relations in a clausal form, where the binary relation symbol is specified as the value of parameter `relsym` ❶.

```
>>> from nltk.corpus import conll2002
>>> vnv = """
... (
... is/V|    # 3rd sing present and
... was/V|   # past forms of the verb zijn ('be')
... werd/V|  # and also present
... wordt/V  # past of worden ('become')
... )
... .*       # followed by anything
... van/Prep # followed by van ('of')
... """
>>> VAN = re.compile(vnv, re.VERBOSE)
>>> for doc in conll2002.chunked_sents('ned.train'):
...     for r in nltk.sem.extract_rels('PER', 'ORG', doc,
...                                    corpus='conll2002', pattern=VAN):
...         print  nltk.sem.show_clause(r, relsym="VAN") ❶
VAN("cornet_d'elzius", 'buitenlandse_handel')
VAN('johan_rottiers', 'kardinaal_van_roey_instituut')
VAN('annie_lennox', 'eurythmics')
```

Your Turn: Replace the last line ❶ with `print show_raw_rtuple(rel, lcon=True, rcon=True)`. This will show you the actual words that intervene between the two NEs and also their left and right context, within a default 10-word window. With the help of a Dutch dictionary, you might be able to figure out why the result `VAN('annie_lennox', 'euryth mics')` is a false hit.

7.7 Summary

- Information extraction systems search large bodies of unrestricted text for specific types of entities and relations, and use them to populate well-organized databases. These databases can then be used to find answers for specific questions.
- The typical architecture for an information extraction system begins by segmenting, tokenizing, and part-of-speech tagging the text. The resulting data is then searched for specific types of entity. Finally, the information extraction system looks at entities that are mentioned near one another in the text, and tries to determine whether specific relationships hold between those entities.
- Entity recognition is often performed using chunkers, which segment multitoken sequences, and label them with the appropriate entity type. Common entity types include ORGANIZATION, PERSON, LOCATION, DATE, TIME, MONEY, and GPE (geo-political entity).