# Extracting Information from Text

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.

1. How can we build a system that extracts structured data, such as tables, 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 will apply techniques from the last two chapters to the problems of chunking and named-entity recognition.

**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 [1.1](https://www.nltk.org/book/ch07.html" \l "tab-db-locations), then answering these queries is straightforward.

***Table 1.1****:*

Locations data

| **Org Name** | **Location Name** |
| --- | --- |
| 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:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> locs = [('Omnicom', 'IN', 'New York'),**  **... ('DDB Needham', 'IN', 'New York'),**  **... ('Kaplan Thaler Group', 'IN', 'New York'),**  **... ('BBDO South', 'IN', 'Atlanta'),**  **... ('Georgia-Pacific', 'IN', 'Atlanta')]**  **>>> query = [e1 for (e1, rel, e2) in locs if e2=='Atlanta']**  **>>> print(query)**  **['BBDO South', 'Georgia-Pacific']** | |

***Table 1.2****:*

Companies that operate in Atlanta

| **Org Name** |
| --- |
| 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. |

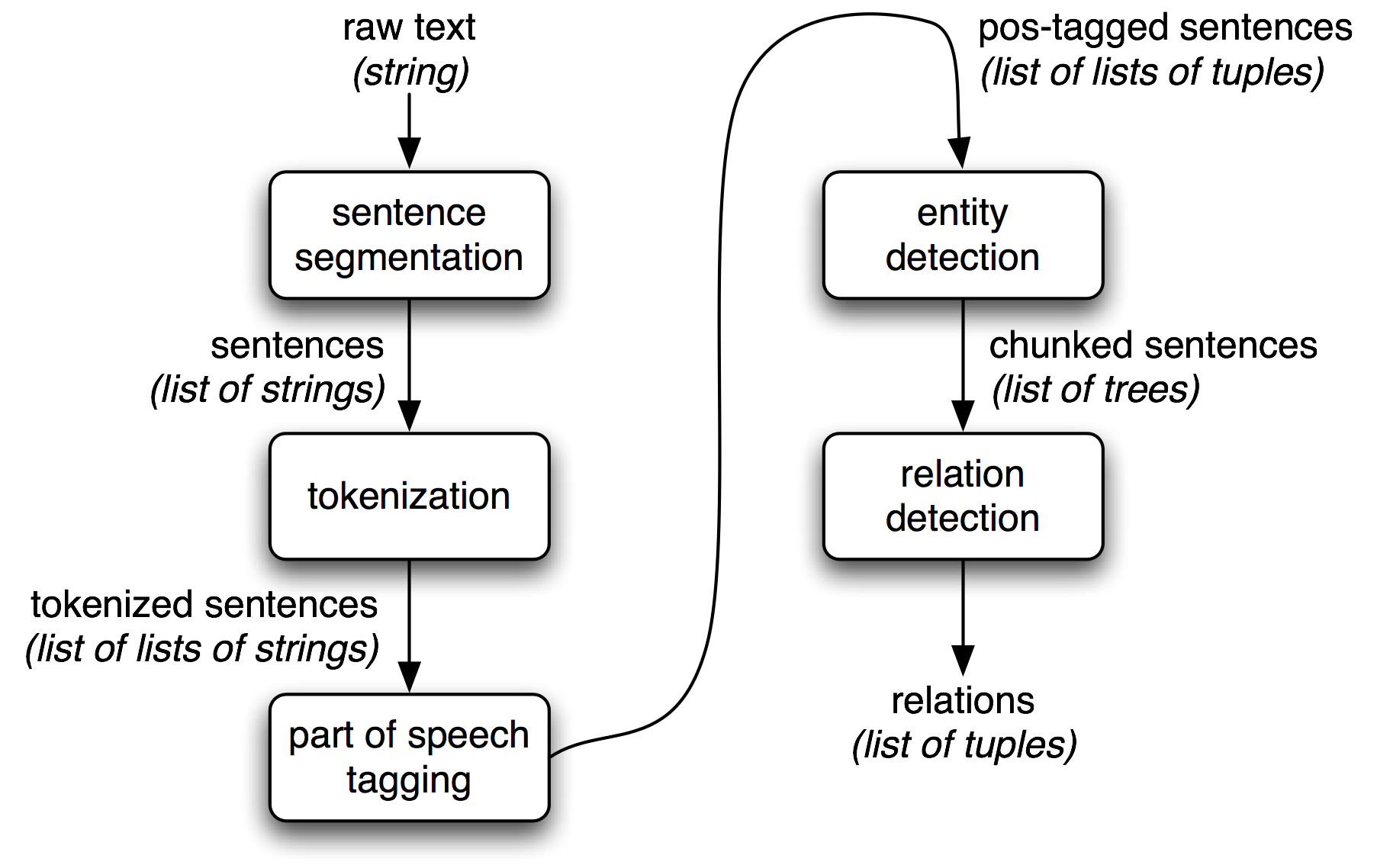
If you read through [(1)](https://www.nltk.org/book/ch08.html#ex-ie4), you will glean the information required to answer the example question. But how do we get a machine to understand enough about [(1)](https://www.nltk.org/book/ch07.html#ex-ie4) to return the answers in [1.2](https://www.nltk.org/book/ch07.html#tab-db-answers)? This is obviously a much harder task. Unlike [1.1](https://www.nltk.org/book/ch05.html#tab-db-locations), [(1)](https://www.nltk.org/book/ch07.html#ex-ie4) contains no structure that links organization names with location names.

One approach to this problem involvement of conversion of the **unstructured data** of natural language sentences into the structured data of [1.1](https://www.nltk.org/book/ch07.html" \l "tab-db-locations). 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.

**1.1   Information Extraction Architecture**

[1.1](https://www.nltk.org/book/ch07.html" \l "fig-ie-architecture) shows the architecture for a simple information extraction system. It begins by processing a document using several of the pre-processing stages, 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 detection**. In this step, we search for mentions of potentially interesting entities in each sentence. Finally, we use **relation detection** to search for likely relations between different entities in the text.



***Figure 1.1****: 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 simple function that simply connects together NLTK's default sentence segmenter[[1]](https://www.nltk.org/book/ch07.html#ie-segment), word tokenizer [[2]](https://www.nltk.org/book/ch07.html#ie-tokenize), and part-of-speech tagger [[3]](https://www.nltk.org/book/ch07.html#ie-postag):

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> def ie\_preprocess(document):**  **... sentences = nltk.sent\_tokenize(document)** **[[1]](https://www.nltk.org/book/ch07.html#ref-ie-segment)**  **... sentences = [nltk.word\_tokenize(sent) for sent in sentences]** **[[2]](https://www.nltk.org/book/ch07.html#ref-ie-tokenize)**  **... sentences = [nltk.pos\_tag(sent) for sent in sentences]** **[[3]](https://www.nltk.org/book/ch07.html#ref-ie-postag)** | |

**Note**

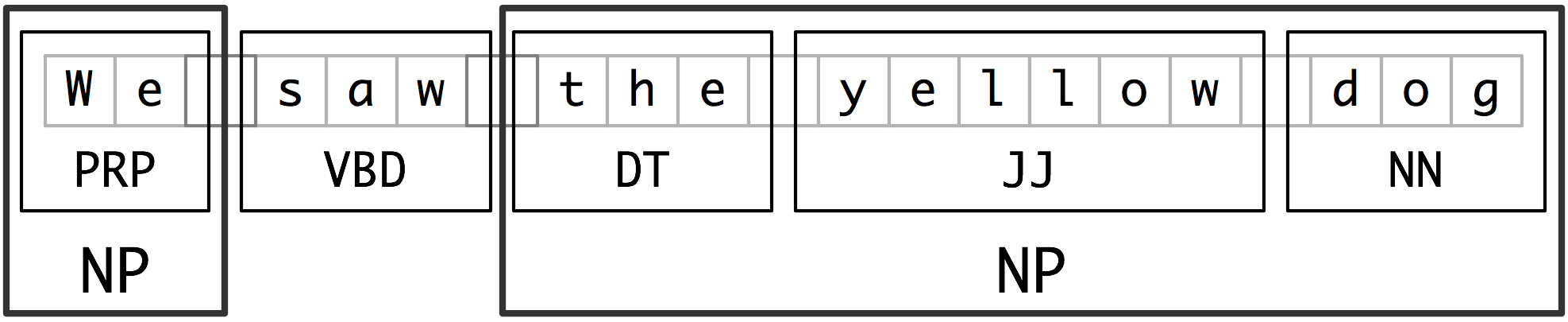
Remember that our program samples assume you begin your interactive session or your program with: import nltk, re, pprint

Next, in named entity detection, 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 NPs 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.

**2   Chunking**

The basic technique we will use for entity detection is **chunking**, which segments and labels multi-token sequences as illustrated in [2.1](https://www.nltk.org/book/ch07.html" \l "fig-chunk-segmentation). The smaller boxes show the word-level tokenization and part-of-speech tagging, while the large boxes show higher-level chunking. Each of these larger boxes is called a **chunk**. Like tokenization, which omits whitespace, chunking usually selects a subset of the tokens. Also like tokenization, the pieces produced by a chunker do not overlap in the source text.



***Figure 2.1****: Segmentation and Labeling at both the Token and Chunk Levels*

In this section, we will explore chunking in some depth, beginning with the definition and representation of chunks. We will see regular expression and n-gram approaches to chunking, and will develop and evaluate chunkers using the CoNLL-2000 chunking corpus. We will then return in [(5)](https://www.nltk.org/book/ch07.html" \l "sec-ner) and [6](https://www.nltk.org/book/ch07.html#sec-relextract) to the tasks of named entity recognition and relation extraction.

**2.1   Noun Phrase Chunking**

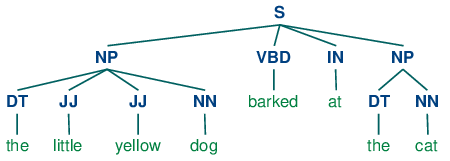
We will begin by considering the task of **noun phrase chunking**, or **NP chunking**, where we search for chunks corresponding to individual noun phrases. For example, here is some Wall Street Journal text with NP-chunks marked using brackets:

|  |  |  |
| --- | --- | --- |
| (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 [2.2](https://www.nltk.org/book/ch07.html" \l "code-chunkex). 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 [[2]](https://www.nltk.org/book/pylisting/code_unigram_chunker.py#chunkex-grammar). 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 [[3]](https://www.nltk.org/book/ch07.html#chunkex-cp), and test it on our example sentence [[4]](https://www.nltk.org/book/ch07.html#chunkex-test). The result is a tree, which we can either print [[5]](https://www.nltk.org/book/ch07.html#chunkex-print), or display graphically [[6]](https://www.nltk.org/book/ch07.html#chunkex-draw).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),** **[[1]](https://www.nltk.org/book/ch07.html#ref-chunkex-sent)**  **... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]**  **>>> grammar = "NP: {<DT>?<JJ>\*<NN>}"** **[[2]](https://www.nltk.org/book/ch07.html#ref-chunkex-grammar)**  **>>> cp = nltk.RegexpParser(grammar)** **[[3]](https://www.nltk.org/book/ch07.html#ref-chunkex-cp)**  **>>> result = cp.parse(sentence)** **[[4]](https://www.nltk.org/book/ch07.html#ref-chunkex-test)**  **>>> print(result)** **[[5]](https://www.nltk.org/book/ch07.html#ref-chunkex-print)**  **(S**  **(NP the/DT little/JJ yellow/JJ dog/NN)**  **barked/VBD**  **at/IN**  **(NP the/DT cat/NN))**  **>>> result.draw()** **[[6]](https://www.nltk.org/book/ch07.html#ref-chunkex-draw)** | |
| **Figure 2.2**: Example of a Simple Regular Expression Based NP Chunker. |



**2.2   Tag Patterns**

The rules that make up a chunk grammar use **tag patterns** to describe sequences of tagged words. A tag pattern is a sequence of part-of-speech tags delimited using angle brackets, e.g. <DT>?<JJ>\*<NN>. Tag patterns are similar to regular expression patterns ([3.4](https://www.nltk.org/book/ch03.html" \l "sec-regular-expressions-word-patterns)). Now, consider the following noun phrases from the Wall Street Journal:

another/DT sharp/JJ dive/NN

trade/NN figures/NNS

any/DT new/JJ policy/NN measures/NNS

earlier/JJR stages/NNS

Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP

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

his/PRP$ Mansion/NNP House/NNP speech/NN

The/DT price/NN cutting/VBG

3/CD %/NN to/TO 4/CD %/NN

more/JJR than/IN 10/CD %/NN

the/DT fastest/JJS developing/VBG trends/NNS

's/POS skill/NN

**Note**

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

**2.3   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.

[2.3](https://www.nltk.org/book/ch07.html#code-chunker1) 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 [[1]](https://www.nltk.org/book/ch05.html#code-chunker1-ex), and run the chunker on this input [[2]](https://www.nltk.org/book/ch07.html#code-chunker1-run).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **grammar = r"""**  **NP: {<DT|PP\$>?<JJ>\*<NN>} # chunk determiner/possessive, adjectives and noun**  **{<NNP>+} # chunk sequences of proper nouns**  **"""**  **cp = nltk.RegexpParser(grammar)**  **sentence = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"),** **[[1]](https://www.nltk.org/book/ch07.html#ref-code-chunker1-ex)**  **("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]** | |
| |  |  | | --- | --- | |  | **>>> print(cp.parse(sentence))**  **(S**  **(NP Rapunzel/NNP)**  **let/VBD**  **down/RP**  **(NP her/PP$ long/JJ golden/JJ hair/NN))** | |
| **Figure 2.3**: Simple Noun Phrase Chunker |

**Note**

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

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

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

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

**Note**

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

**2.4   Exploring Text Corpora**

In [2](https://www.nltk.org/book/ch07.html#sec-tagged-corpora) 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.label() == 'CHUNK': print(subtree)**  **...**  **(CHUNK combined/VBN to/TO achieve/VB)**  **(CHUNK continue/VB to/TO place/VB)**  **(CHUNK serve/VB to/TO protect/VB)**  **(CHUNK wanted/VBD to/TO wait/VB)**  **(CHUNK allowed/VBN to/TO place/VB)**  **(CHUNK expected/VBN to/TO become/VB)**  **...**  **(CHUNK seems/VBZ to/TO overtake/VB)**  **(CHUNK want/VB to/TO buy/VB)** | |

**Note**

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

**2.5   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 [2.1](https://www.nltk.org/book/ch07.html" \l "tab-chinking-example).

***Table 2.1****:*

Three chinking rules applied to the same chunk

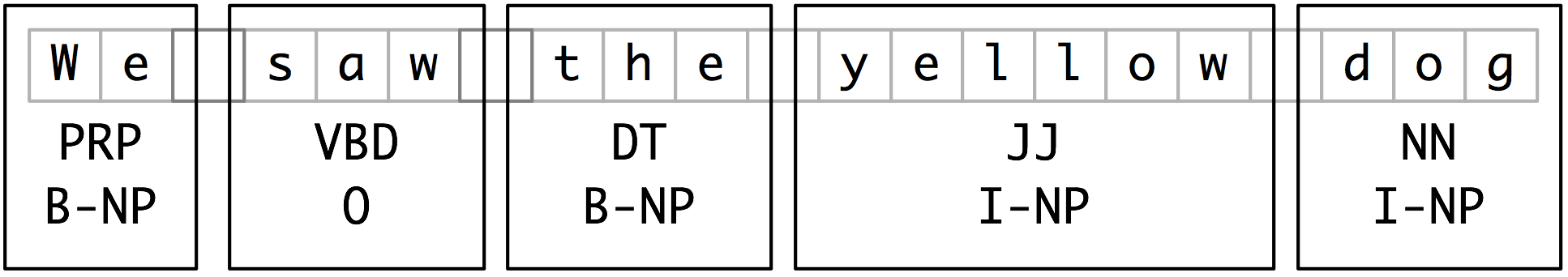
| **` `** | **Entire chunk** | **Middle of a chunk** | **End of a chunk** |
| --- | --- | --- | --- |
| *Input* | [a/DT little/JJ dog/NN] | [a/DT little/JJ dog/NN] | [a/DT little/JJ dog/NN] |
| *Operation* | Chink "DT JJ NN" | Chink "JJ" | Chink "NN" |
| *Pattern* | }DT JJ NN{ | }JJ{ | }NN{ |
| *Output*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

In [2.4](https://www.nltk.org/book/ch07.html#code-chinker), we put the entire sentence into a single chunk, then excise the chinks.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **grammar = r"""**  **NP:**  **{<.\*>+} # Chunk everything**  **}<VBD|IN>+{ # Chink sequences of VBD and IN**  **"""**  **sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),**  **("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]**  **cp = nltk.RegexpParser(grammar)** | |
| |  |  | | --- | --- | |  | **>>> print(cp.parse(sentence))**  **(S**  **(NP the/DT little/JJ yellow/JJ dog/NN)**  **barked/VBD**  **at/IN**  **(NP the/DT cat/NN))** | |
| **Figure 2.4**: Simple Chinker |

**2.6   Representing Chunks: Tags vs Trees**

As befits their intermediate status between tagging and parsing ([8.](https://www.nltk.org/book/ch07.html#chap-parse)), 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 [2.5](https://www.nltk.org/book/ch07.html" \l "fig-chunk-tagrep).



***Figure 2.5****: 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 [2.5](https://www.nltk.org/book/ch07.html#fig-chunk-tagrep) would appear in a file:

We PRP B-NP

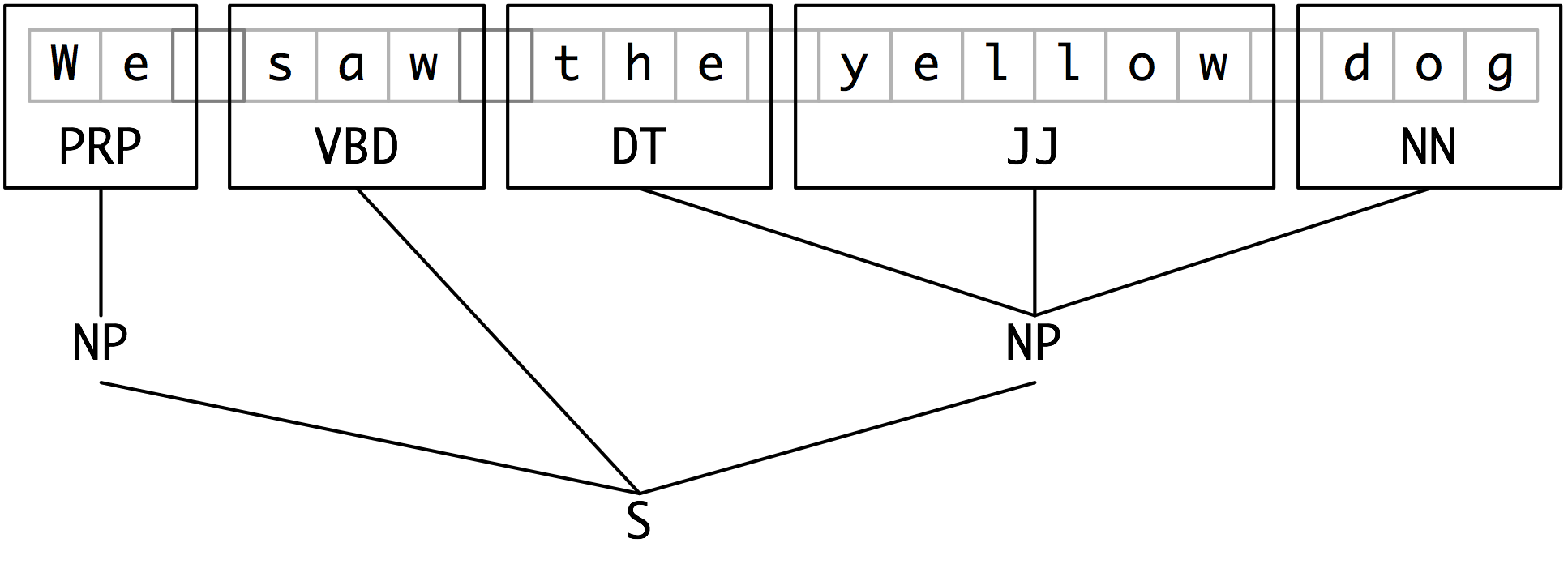
saw VBD O

the DT B-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 [2.6](https://www.nltk.org/book/ch07.html#fig-chunk-treerep).



***Figure 2.6****: Tree Representation of Chunk Structures*

**Note**

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

**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, and 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.

**3.1   Reading IOB Format and the CoNLL 2000 Corpus**

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

he PRP B-NP

accepted VBD B-VP

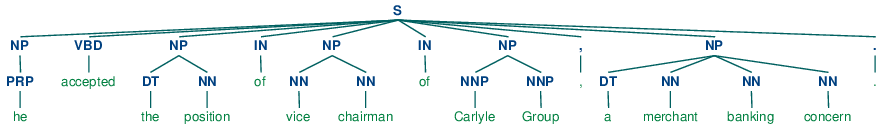
the DT B-NP

position NN I-NP

...

A conversion function chunk.conllstr2tree () builds a tree representation from one of these multi-line strings. Moreover, it permits us to choose any subset of the three chunk types to use, 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 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 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)**  **./.)** | |

## 3.2   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 ([4](https://www.nltk.org/book/ch07.html#sec-automatic-tagging)). 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 [3.1](https://www.nltk.org/book/ch07.html#code-unigram-chunker), 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 [[1]](https://www.nltk.org/book/ch07.html#code-unigram-chunker-constructor) which is called when we build a new UnigramChunker; and the parse method [[3]](https://www.nltk.org/book/ch07.html#code-unigram-chunker-parse) which is used to chunk new sentences.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **class UnigramChunker(nltk.ChunkParserI):**  **def \_\_init\_\_(self, train\_sents):** **[[1]](https://www.nltk.org/book/ch07.html#ref-code-unigram-chunker-constructor)**  **train\_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(sent)]**  **for sent in train\_sents]**  **self.tagger = nltk.UnigramTagger(train\_data)** **[[2]](https://www.nltk.org/book/ch07.html#ref-code-unigram-chunker-buildit)**  **def parse(self, sentence):** **[[3]](https://www.nltk.org/book/ch07.html#ref-code-unigram-chunker-parse)**  **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)** | |
| [**Example 3.1 (code\_unigram\_chunker.py)**](https://www.nltk.org/book/ch07.html): **Figure 3.1**: Noun Phrase Chunking with a Unigram Tagger |

The constructor [[1]](https://www.nltk.org/book/ch07.html#code-unigram-chunker-constructor) expects a list of training sentences, which will be in the form of chunk trees. It first converts training data to a form that is 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 [[3]](https://www.nltk.org/book/ch07.html#code-unigram-chunker-parse) 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 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 [[2]](https://www.nltk.org/book/ch07.html#code-unigram-chunker-buildit) in [3.1](https://www.nltk.org/book/ch07.html#code-unigram-chunker) 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%** | |