CHAPTER 3

# Processing and Understanding Text

In the previous chapters, we saw a glimpse of the entire natural language processing and text analytics landscape with essential terminology and concepts. Besides this, we were also introduced to the Python programming language, essential constructs, syntax, and learned how to work with strings to manage textual data. To perform complex operations on text with machine learning or deep learning algorithms, you need to process and parse textual data into more easy-to-interpret formats. All machine learning algorithms, be they supervised or unsupervised techniques, work with input features, which are numeric in nature. While this is a separate topic under feature engineering, which we shall explore in detail in the next chapter, to get to that step, you will need to clean, normalize, and preprocess the initial textual data.

The text corpora and other textual data in their native raw formats are normally not well formatted and standardized and of course we should expect this, after all, text data is highly unstructured! Text processing or, to be more specific preprocessing, involves a wide variety of techniques that convert raw text into well-defined sequences of linguistic components that have standard structure and notation. Additional metadata is often also present in the form of annotations to give more meaning to the text components like tags. The following list gives us an idea of some of the most popular text preprocessing and understanding techniques, which we explore in this chapter.

- Removing HTML tags
- Tokenization
- Removing unnecessary tokens and stopwords
- Handling contractions
- Correcting spelling errors




D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_3

- Stemming
- Lemmatization
- Tagging
- Chunking
- Parsing

Besides these techniques, you also need to perform some basic operations, like case conversion, dealing with irrelevant components, and removing noise based on the problem to be solved. An important point to remember is that a robust text preprocessing system is always an essential part of any application on NLP and text analytics. The primary reason for that is because all the textual components obtained after preprocessing—be they words, phrases, sentences, or any other tokens—form the basic building blocks of input that's fed into the further stages of the application that perform more complex analyses including learning patterns and extracting information. Hence, the saying "Garbage in Garbage out!" is relevant here because if we do not process the text properly, we will end up getting unwanted and irrelevant results from our applications and systems.

Besides this, text processing helps in cleaning and standardization of the text which not only helps in analytical systems like increasing the accuracy of classifiers but we also get additional information and metadata in the form of annotations. They are very useful in giving more information about the text. We will touch upon normalizing text using various techniques including cleaning, removing unnecessary tokens, stems, and lemmas in this chapter.

Another important aspect is understanding the textual data after processing and normalizing it. This will involve revisiting some of the concepts surrounding language syntax and structure from Chapter 1, where we talked about sentences, phrases, parts of speech, shallow parsing, and grammar. In this chapter, we look at ways to implement these concepts and use them on real data. We follow a structured and definite path in this chapter, starting from text processing, and gradually exploring the various concepts and techniques associated with it and then moving on to understanding text structure and syntax. Since this book is specifically aimed at practitioners, various code snippets and practical examples also equip you with the right tools and frameworks for implementing these concepts in solving practical problems. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

# Text Preprocessing and Wrangling

Text *wrangling* (also called *preprocessing* or *normalization*) is a process that consists of a series of steps to wrangle, clean, and standardize textual data into a form that could be consumed by other NLP and intelligent systems powered by machine learning and deep learning. Common techniques for preprocessing include cleaning text, tokenizing text, removing special characters, case conversion, correcting spellings, removing stopwords and other unnecessary terms, stemming, and lemmatization. In this section, we discuss various techniques that are commonly used for text wrangling based on the list we mentioned at the beginning of this chapter. The key idea is to remove unnecessary content from one or more text documents in a corpus (or corpora) and get clean text documents.

## Removing HTML Tags

Often, unstructured text contains a lot of noise, especially if you use techniques like web scraping or screen scraping to retrieve data from web pages, blogs, and online repositories. HTML tags, JavaScript, and Iframe tags typically don't add much value to understanding and analyzing text. Our main intent is to extract meaningful textual content from the data extracted from the web. Let's look at a section of a web page showing the King James version of the Bible, freely available thanks to Project Gutenberg, depicted in Figure 3-1.

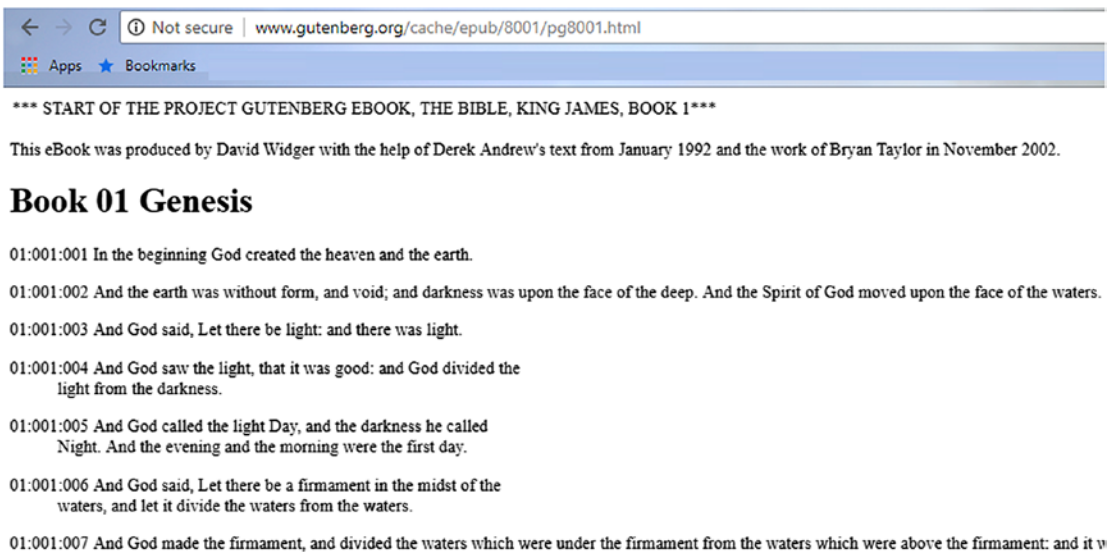


***Figure 3-1.*** *Section of a web page showing a chapter from the Bible*

We will now leverage requests and retrieve the contents of this web page in Python. This is known as web scraping and the following code helps us achieve this.

```
import requests

data = requests.get('http://www.gutenberg.org/cache/epub/8001/pg8001.html')
content = data.content
print(content[1163:2200])
```

```
b'content="Ebookmaker 0.4.0a5 by Marcello Perathoner <webmaster@gutenberg.
org>" name="generator"/>\r\n</head>\r\n  <body><p id="id00000">Project
Gutenberg EBook The Bible, King James, Book 1: Genesis</p>\r\n\r\n<p
id="id00001">Copyright laws are changing all over the world. Be sure
to check the\r\ncopyright laws for your country before downloading or
redistributing\r\nthis or any other Project Gutenberg eBook.</p>\r\n\r\n<p
id="id00002">This header should be the first thing seen when viewing this
Project\r\nGutenberg file.  Please do not remove it.  Do not change or edit
the\r\nheader without written permission.</p>\r\n\r\n<p id="id00003">Please
read the "legal small print," and other information about the\r\neBook and
Project Gutenberg at the bottom of this file.  Included is\r\nimportant
information about your specific rights and restrictions in\r\nhow the
file may be used.  You can also find out about how to make a\r\ndonation
to Project Gutenberg, and how to get involved.</p>\r\n\r\n<p id="id00004"
style="margin-top: 2em">**Welcome To The World of F'
```

We can clearly see from the preceding output that it is extremely difficult to decipher the actual textual content in the web page, due to all the unnecessary HTML tags. We need to remove those tags. The BeautifulSoup library provides us with some handy functions that help us remove these unnecessary tags with ease.

```
import re
from bs4 import BeautifulSoup

def strip_html_tags(text):
    soup = BeautifulSoup(text, "html.parser")
    [s.extract() for s in soup(['iframe', 'script'])]
    stripped_text = soup.get_text()
    stripped_text = re.sub(r'[\r|\n|\r\n]+', '\n', stripped_text)
```

```
    return stripped_text

clean_content = strip_html_tags(content)
print(clean_content[1163:2045])
```

```
*** START OF THE PROJECT GUTENBERG EBOOK, THE BIBLE, KING JAMES, BOOK 1***
This eBook was produced by David Widger
with the help of Derek Andrew's text from January 1992
and the work of Bryan Taylor in November 2002.
Book 01        Genesis
01:001:001 In the beginning God created the heaven and the earth.
01:001:002 And the earth was without form, and void; and darkness was
           upon the face of the deep. And the Spirit of God moved upon
           the face of the waters.
01:001:003 And God said, Let there be light: and there was light.
01:001:004 And God saw the light, that it was good: and God divided the
           light from the darkness.
01:001:005 And God called the light Day, and the darkness he called
           Night. And the evening and the morning were the first day.
01:001:006 And God said, Let there be a firmament in the midst of the
           waters,
```

You can compare this output with the raw web page content and see that we have successfully removed the unnecessary HTML tags. We now have a clean body of text that's easier to interpret and understand.

## Text Tokenization

Chapter 1 explained textual structure, its components, and tokens. Tokens are independent and minimal textual components that have some definite syntax and semantics. A paragraph of text or a text document has several components, including sentences, which can be further broken down into clauses, phrases, and words. The most popular tokenization techniques include sentence and word tokenization, which are used to break down a text document (or corpus) into sentences and each sentence into words. Thus, tokenization can be defined as the process of breaking down or splitting textual data into smaller and more meaningful components called *tokens*. In the following sections, we look at some ways to tokenize text into sentences and words.

## Sentence Tokenization

*Sentence tokenization* is the process of splitting a text corpus into sentences that act as the first level of tokens the corpus is comprised of. This is also known as sentence segmentation, since we try to segment the text into meaningful sentences. Any text corpus is a body of text where each paragraph comprises several sentences. There are various ways to perform sentence tokenization. Basic techniques include looking for specific delimiters between sentences like a period (`.`) or a newline character (`\n`) and sometimes even a semicolon (`;`). We will use the NLTK framework, which provides various interfaces for performing sentence tokenization. We primarily focus on the following sentence tokenizers:

- `sent_tokenize`
- Pretrained sentence tokenization models
- `PunktSentenceTokenizer`
- `RegexpTokenizer`

Before we can tokenize sentences, we need some text on which we can try these operations. We load some sample text and part of the Gutenberg corpus available in NLTK. We load the necessary dependencies using the following snippet.

```
import nltk
from nltk.corpus import gutenberg
from pprint import pprint
import numpy as np

# loading text corpora
alice = gutenberg.raw(fileids='carroll-alice.txt')
sample_text = ("US unveils world's most powerful supercomputer, beats China. "
               "The US has unveiled the world's most powerful supercomputer
               called 'Summit', "
               "beating the previous record-holder China's Sunway
               TaihuLight. With a peak performance "
               "of 200,000 trillion calculations per second, it is over
               twice as fast as Sunway TaihuLight, "
```

```
                "which is capable of 93,000 trillion calculations per
                second. Summit has 4,608 servers, "
                "which reportedly take up the size of two tennis courts.")
sample_text
```

```
"US unveils world's most powerful supercomputer, beats China. The US
has unveiled the world's most powerful supercomputer called 'Summit',
beating the previous record-holder China's Sunway TaihuLight. With a
peak performance of 200,000 trillion calculations per second, it is over
twice as fast as Sunway TaihuLight, which is capable of 93,000 trillion
calculations per second. Summit has 4,608 servers, which reportedly take up
the size of two tennis courts."
```

We can check the length of the "Alice in Wonderland" corpus and the first few lines in it using the following snippet.

```
# Total characters in Alice in Wonderland
len(alice)
```

```
144395
```

```
# First 100 characters in the corpus
alice[0:100]
```

```
"[Alice's Adventures in Wonderland by Lewis Carroll 1865]\n\nCHAPTER
I. Down the Rabbit-Hole\n\nAlice was"
```

## Default Sentence Tokenizer

The `nltk.sent_tokenize(...)` function is the default sentence tokenization function that NLTK recommends and it uses an instance of the `PunktSentenceTokenizer` class internally. However, this is not just a normal object or instance of that class. It has been pretrained on several language models and works really well on many popular languages besides English. The following snippet shows the basic usage of this function on our text samples.

```
default_st = nltk.sent_tokenize
alice_sentences = default_st(text=alice)
sample_sentences = default_st(text=sample_text)

print('Total sentences in sample_text:', len(sample_sentences))
print('Sample text sentences :-')
print(np.array(sample_sentences))

print('\nTotal sentences in alice:', len(alice_sentences))
print('First 5 sentences in alice:-')
print(np.array(alice_sentences[0:5]))
```

Upon running this snippet, you get the following output depicting the total number of sentences and what those sentences look like in the text corpora.

```
Total sentences in sample_text: 4
Sample text sentences :-
["US unveils world's most powerful supercomputer, beats China."
 "The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight."
 'With a peak performance of 200,000 trillion calculations per second, it
is over twice as fast as Sunway TaihuLight, which is capable of 93,000
trillion calculations per second.'
 'Summit has 4,608 servers, which reportedly take up the size of two tennis
courts.']

Total sentences in alice: 1625
First 5 sentences in alice:-
["[Alice's Adventures in Wonderland by Lewis Carroll 1865]\n\nCHAPTER I."
 "Down the Rabbit-Hole\n\nAlice was beginning to get very tired of sitting
by her sister on the\nbank, and of having nothing to do: once or twice she
had peeped into the\nbook her sister was reading, but it had no pictures
or conversations in\nit, 'and what is the use of a book,' thought Alice
'without pictures or\nconversation?'"
 'So she was considering in her own mind (as well as she could, for the\nhot
day made her feel very sleepy and stupid), whether the pleasure\nof making
a daisy-chain would be worth the trouble of getting up and\npicking the
daisies, when suddenly a White Rabbit with pink eyes ran\nclose by her.'
```

```
 "There was nothing so VERY remarkable in that; nor did Alice think it so\
nVERY much out of the way to hear the Rabbit say to itself, 'Oh dear!"
 'Oh dear!']
```

Now, as you can see, the tokenizer is quite intelligent. It doesn't just use periods to delimit sentences, but also considers other punctuation and capitalization of words. We can also tokenize text of other languages using some pretrained models present in NLTK.

## Pretrained Sentence Tokenizer Models

Suppose we were dealing with German text. We can use `sent_tokenize`, which is already trained, or load a pretrained tokenization model on German text into a `PunktSentenceTokenizer` instance and perform the same operation. The following snippet shows this. We start by loading a German text corpus and inspecting it.

```
from nltk.corpus import europarl_raw

german_text = europarl_raw.german.raw(fileids='ep-00-01-17.de')
# Total characters in the corpus
print(len(german_text))
# First 100 characters in the corpus
print(german_text[0:100])
```

```
157171

Wiederaufnahme der Sitzungsperiode Ich erkläre die am Freitag , dem 17.
Dezember unterbrochene Sit
```

Next, we tokenize the text corpus into sentences using the default `sent_tokenize(...)` tokenizer and a pretrained German language tokenizer by loading it from the NLTK resources.

```
# default sentence tokenizer
german_sentences_def = default_st(text=german_text, language='german')

# loading german text tokenizer into a PunktSentenceTokenizer instance
german_tokenizer = nltk.data.load(resource_url='tokenizers/punkt/german.
pickle')
german_sentences = german_tokenizer.tokenize(german_text)
```

We can now verify the time of our German tokenizer and check if the results obtained by using the two tokenizers match!

```
# verify the type of german_tokenizer
# should be PunktSentenceTokenizer
print(type(german_tokenizer))
```

```
<class 'nltk.tokenize.punkt.PunktSentenceTokenizer'>
```

```
# check if results of both tokenizers match
# should be True
print(german_sentences_def == german_sentences)
```

```
True
```

Thus we see that indeed the `german_tokenizer` is an instance of `PunktSentenceTokenizer`, which specializes in dealing with the German language. We also checked if the sentences obtained from the default tokenizer are the same as the sentences obtained by this pretrained tokenizer. As expected, they are the same (`true`). We also print some sample tokenized sentences from the output.

```
# print first 5 sentences of the corpus
print(np.array(german_sentences[:5]))
```

```
[' \nWiederaufnahme der Sitzungsperiode Ich erkläre die am Freitag , dem
17. Dezember unterbrochene Sitzungsperiode des Europäischen Parlaments für
wiederaufgenommen , wünsche Ihnen nochmals alles Gute zum Jahreswechsel und
hoffe , daß Sie schöne Ferien hatten .'
 'Wie Sie feststellen konnten , ist der gefürchtete " Millenium-Bug " nicht
eingetreten .'
 'Doch sind Bürger einiger unserer Mitgliedstaaten Opfer von schrecklichen
Naturkatastrophen geworden .'
 'Im Parlament besteht der Wunsch nach einer Aussprache im Verlauf dieser
Sitzungsperiode in den nächsten Tagen .'
 'Heute möchte ich Sie bitten - das ist auch der Wunsch einiger
Kolleginnen und Kollegen - , allen Opfern der Stürme , insbesondere in den
verschiedenen Ländern der Europäischen Union , in einer Schweigeminute zu
gedenken .']
```

Thus we see that our assumption was indeed correct and you can tokenize sentences belonging to different languages in two different ways.

## PunktSentenceTokenizer

Using the default `PunktSentenceTokenizer` class is also pretty straightforward, as the following snippet shows.

```
punkt_st = nltk.tokenize.PunktSentenceTokenizer()
sample_sentences = punkt_st.tokenize(sample_text)
print(np.array(sample_sentences))
```

```
["US unveils world's most powerful supercomputer, beats China."
 "The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight."
 'With a peak performance of 200,000 trillion calculations per second, it
is over twice as fast as Sunway TaihuLight, which is capable of 93,000
trillion calculations per second.'
 'Summit has 4,608 servers, which reportedly take up the size of two tennis
courts.']
```

## RegexpTokenizer

The last tokenizer we cover in sentence tokenization is using an instance of the `RegexpTokenizer` class to tokenize text into sentences, where we will use specific regular expression-based patterns to segment sentences. Recall the regular expressions from the previous chapter if you want to refresh your memory. The following snippet shows how to use a regex pattern to tokenize sentences.

```
SENTENCE_TOKENS_PATTERN = r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<![A-Z]\.)
(?<=\.|\?|\!)\s'
regex_st = nltk.tokenize.RegexpTokenizer(
            pattern=SENTENCE_TOKENS_PATTERN,
            gaps=True)
sample_sentences = regex_st.tokenize(sample_text)
print(np.array(sample_sentences))
```

```
["US unveils world's most powerful supercomputer, beats China."
 "The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight."
 'With a peak performance of 200,000 trillion calculations per second, it
is over twice as fast as Sunway TaihuLight, which is capable of 93,000
trillion calculations per second.'
 'Summit has 4,608 servers, which reportedly take up the size of two tennis
courts.']
```

This output shows that we obtained the same sentences as we had obtained using the other tokenizers. This gives us an idea of tokenizing text into sentences using different NLTK interfaces. In the following section, we look at tokenizing these sentences into words using several techniques.

## Word Tokenization

*Word tokenization* is the process of splitting or segmenting sentences into their constituent words. A sentence is a collection of words and with tokenization we essentially split a sentence into a list of words that can be used to reconstruct the sentence. Word tokenization is really important in many processes, especially in cleaning and normalizing text where operations like stemming and lemmatization work on each individual word based on its respective stems and lemma. Similar to sentence tokenization, NLTK provides various useful interfaces for word tokenization. We will touch up on the following main interfaces:

- `word_tokenize`
- `TreebankWordTokenizer`
- `TokTokTokenizer`
- `RegexpTokenizer`
- Inherited tokenizers from `RegexpTokenizer`

We leverage our sample text data from the previous section to demonstrate hands-on examples.

## Default Word Tokenizer

The `nltk.word_tokenize(...)` function is the default and recommended word tokenizer, as specified by NLTK. This tokenizer is an instance or object of the `TreebankWordTokenizer` class in its internal implementation and acts as a wrapper to that core class. The following snippet illustrates its usage.

```
default_wt = nltk.word_tokenize
words = default_wt(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'Summit", "'", ',', 'beating', 'the',
       'previous', 'record-holder', 'China', "'s", 'Sunway', 'TaihuLight',
       '.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second', '.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

## TreebankWordTokenizer

The `TreebankWordTokenizer` is based on the Penn Treebank and uses various regular expressions to tokenize the text. Of course, one primary assumption here is that we have already performed sentence tokenization beforehand. The original tokenizer used in the Penn Treebank is available as a `sed` script and you can check it out at http://www.cis.upenn.edu/~treebank/tokenizer.sed to get an idea of the patterns used to tokenize the sentences into words. Some of the main features of this tokenizer are mentioned here:

- Splits and separates out periods that appear at the end of a sentence
- Splits and separates commas and single quotes when followed by whitespace

- Most punctuation characters are split and separated into independent tokens
- Splits words with standard contractions, such as don't to do and n't

The following snippet shows the usage of the `TreebankWordTokenizer` for word tokenization.

```
treebank_wt = nltk.TreebankWordTokenizer()
words = treebank_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'Summit", "'", ',', 'beating', 'the',
       'previous', 'record-holder', 'China', "'s", 'Sunway',
       'TaihuLight.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

As expected, the output is similar to `word_tokenize()`, since they use the same tokenizing mechanism.

## TokTokTokenizer

TokTokTokenizer is one of the newer tokenizers introduced by NLTK present in the `nltk.tokenize.toktok` module. In general, the tok-tok tokenizer is a general tokenizer, where it assumes that the input has one sentence per line. Hence, only the final period is tokenized. However, as needed, we can remove the other periods from the words using regular expressions. Tok-tok has been tested on, and gives reasonably good results for, English, Persian, Russian, Czech, French, German, Vietnamese, and many other languages. It is in fact a Python port of https://github.com/jonsafari/tok-tok, where there is also a Perl implementation. The following code shows a tokenization operation using the TokTokTokenizer.

```
from nltk.tokenize.toktok import ToktokTokenizer
tokenizer = ToktokTokenizer()
words = tokenizer.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', 'called', "'", 'Summit', "'", ',', 'beating',
       'the', 'previous', 'record-holder', 'China', "'", 's', 'Sunway',
       'TaihuLight.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

## RegexpTokenizer

We now look at how to use regular expressions and the `RegexpTokenizer` class to tokenize sentences into words. Remember that there are two main parameters that are useful in tokenization—the regex `pattern` for building the tokenizer and the `gaps` parameter, which, if set to true, is used to find the gaps between the tokens. Otherwise, it is used to find the tokens themselves. The following code snippet shows some examples of using regular expressions to perform word tokenization.

```
# pattern to identify tokens themselves
TOKEN_PATTERN = r'\w+'
regex_wt = nltk.RegexpTokenizer(pattern=TOKEN_PATTERN,
                                gaps=False)
words = regex_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', 's', 'most', 'powerful', 'supercomputer',
       'beats', 'China', 'The', 'US', 'has', 'unveiled', 'the', 'world',
       's', 'most', 'powerful', 'supercomputer', 'called', 'Summit',
```

```
       'beating', 'the', 'previous', 'record', 'holder', 'China', 's',
       'Sunway', 'TaihuLight', 'With', 'a', 'peak', 'performance', 'of',
       '200', '000', 'trillion', 'calculations', 'per', 'second', 'it',
       'is', 'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
       'which', 'is', 'capable', 'of', '93', '000', 'trillion',
       'calculations', 'per', 'second', 'Summit', 'has', '4', '608',
       'servers', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts'], dtype='<U13')
```

```
# pattern to identify tokens by using gaps between tokens
GAP_PATTERN = r'\s+'
regex_wt = nltk.RegexpTokenizer(pattern=GAP_PATTERN,
                                gaps=True)
words = regex_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', "world's", 'most', 'powerful', 'supercomputer,',
       'beats', 'China.', 'The', 'US', 'has', 'unveiled', 'the',
       "world's", 'most', 'powerful', 'supercomputer', 'called',
       "'Summit',", 'beating', 'the', 'previous', 'record-holder',
       "China's", 'Sunway', 'TaihuLight.', 'With', 'a', 'peak',
       'performance', 'of', '200,000', 'trillion', 'calculations', 'per',
       'second,', 'it', 'is', 'over', 'twice', 'as', 'fast', 'as',
       'Sunway', 'TaihuLight,', 'which', 'is', 'capable', 'of', '93,000',
       'trillion', 'calculations', 'per', 'second.', 'Summit', 'has',
       '4,608', 'servers,', 'which', 'reportedly', 'take', 'up', 'the',
       'size', 'of', 'two', 'tennis', 'courts.'], dtype='<U14')
```

Thus, you can see that there are multiple ways of obtaining the same results leveraging token patterns themselves or gap patterns. The following code shows us how to obtain the token boundaries for each token during the tokenize operation.

```
word_indices = list(regex_wt.span_tokenize(sample_text))
print(word_indices)
print(np.array([sample_text[start:end] for start, end in word_indices]))
```

```
[(0, 2), (3, 10), (11, 18), (19, 23), (24, 32), (33, 47), (48, 53), (54,
60), (61, 64), (65, 67), (68, 71), (72, 80), (81, 84), (85, 92), (93, 97),
```

```
(98, 106), (107, 120), (121, 127), (128, 137), (138, 145), (146, 149),
(150, 158), (159, 172), (173, 180), (181, 187), (188, 199), (200, 204),
(205, 206), (207, 211), (212, 223), (224, 226), (227, 234), (235, 243),
(244, 256), (257, 260), (261, 268), (269, 271), (272, 274), (275, 279),
(280, 285), (286, 288), (289, 293), (294, 296), (297, 303), (304, 315),
(316, 321), (322, 324), (325, 332), (333, 335), (336, 342), (343, 351),
(352, 364), (365, 368), (369, 376), (377, 383), (384, 387), (388, 393),
(394, 402), (403, 408), (409, 419), (420, 424), (425, 427), (428, 431),
(432, 436), (437, 439), (440, 443), (444, 450), (451, 458)]
['US' 'unveils' "world's" 'most' 'powerful' 'supercomputer,' 'beats'
 'China.' 'The' 'US' 'has' 'unveiled' 'the' "world's" 'most' 'powerful'
 'supercomputer' 'called' "'Summit'," 'beating' 'the' 'previous'
 'record-holder' "China's" 'Sunway' 'TaihuLight.' 'With' 'a' 'peak'
 'performance' 'of' '200,000' 'trillion' 'calculations' 'per' 'second,'
 'it' 'is' 'over' 'twice' 'as' 'fast' 'as' 'Sunway' 'TaihuLight,' 'which'
 'is' 'capable' 'of' '93,000' 'trillion' 'calculations' 'per' 'second.'
 'Summit' 'has' '4,608' 'servers,' 'which' 'reportedly' 'take' 'up' 'the'
 'size' 'of' 'two' 'tennis' 'courts.']
```

## Inherited Tokenizers from RegexpTokenizer

Besides the base `RegexpTokenizer` class, there are several derived classes that perform different types of word tokenization. The `WordPunktTokenizer` uses the pattern `r'\w+|[^\w\s]+'` to tokenize sentences into independent alphabetic and non-alphabetic tokens.

```
wordpunkt_wt = nltk.WordPunctTokenizer()
words = wordpunkt_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', 'called', "'", 'Summit', "',", 'beating', 'the',
       'previous', 'record', '-', 'holder', 'China', "'", 's', 'Sunway',
       'TaihuLight', '.', 'With', 'a', 'peak', 'performance', 'of', '200',
```

```
       ',', '000', 'trillion', 'calculations', 'per', 'second', ',', 'it',
       'is', 'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
       ',', 'which', 'is', 'capable', 'of', '93', ',', '000', 'trillion',
       'calculations', 'per', 'second', '.', 'Summit', 'has', '4', ',',
       '608', 'servers', ',', 'which', 'reportedly', 'take', 'up', 'the',
       'size', 'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

The `WhitespaceTokenizer` tokenizes sentences into words based on whitespace, like tabs, newlines, and spaces. The following snippet shows demonstrations of these tokenizers.

```
whitespace_wt = nltk.WhitespaceTokenizer()
words = whitespace_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', "world's", 'most', 'powerful', 'supercomputer,',
       'beats', 'China.', 'The', 'US', 'has', 'unveiled', 'the',
       "world's", 'most', 'powerful', 'supercomputer', 'called',
       "'Summit',", 'beating', 'the', 'previous', 'record-holder',
       "China's", 'Sunway', 'TaihuLight.', 'With', 'a', 'peak',
       'performance', 'of', '200,000', 'trillion', 'calculations', 'per',
       'second,', 'it', 'is', 'over', 'twice', 'as', 'fast', 'as',
       'Sunway', 'TaihuLight,', 'which', 'is', 'capable', 'of', '93,000',
       'trillion', 'calculations', 'per', 'second.', 'Summit', 'has',
       '4,608', 'servers,', 'which', 'reportedly', 'take', 'up', 'the',
       'size', 'of', 'two', 'tennis', 'courts.'], dtype='<U14')
```

## Building Robust Tokenizers with NLTK and spaCy

For a typical NLP pipeline, I recommend leveraging state-of-the-art libraries like NLTK and spaCy and using some of their robust utilities to build a custom function to perform both sentence- and word-level tokenization. A simple example is depicted in the following snippets. We start with looking at how we can leverage NLTK.

```
def tokenize_text(text):
    sentences = nltk.sent_tokenize(text)
    word_tokens = [nltk.word_tokenize(sentence) for sentence in sentences]
    return word_tokens
```

```
sents = tokenize_text(sample_text)
np.array(sents)
```

```
array([list(['US', 'unveils', 'world', "'s", 'most', 'powerful',
             'supercomputer', ',', 'beats', 'China', '.']),
       list(['The', 'US', 'has', 'unveiled', 'the', 'world', "'s", 'most',
             'powerful', 'supercomputer', 'called', "'Summit", "'", ',',
             'beating', 'the', 'previous', 'record-holder', 'China', "'s",
             'Sunway', 'TaihuLight', '.']),
       list(['With', 'a', 'peak', 'performance', 'of', '200,000',
             'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
             'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
             ',', 'which', 'is', 'capable', 'of', '93,000', 'trillion',
             'calculations', 'per', 'second', '.']),
       list(['Summit', 'has', '4,608', 'servers', ',', 'which',
             'reportedly', 'take', 'up', 'the', 'size', 'of', 'two',
             'tennis', 'courts', '.'])], dtype=object)
```

We can also get to the level of word-level tokenization by leveraging list comprehensions, as depicted in the following code.

```
words = [word for sentence in sents for word in sentence]
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'Summit", "'", ',', 'beating', 'the',
       'previous', 'record-holder', 'China', "'s", 'Sunway', 'TaihuLight',
       '.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second', '.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

In a similar way, we can leverage spaCy to perform sentence- and word-level tokenizations really quickly, as depicted in the following snippets.

```
import spacy
nlp = spacy.load('en_core', parse = True, tag=True, entity=True)
text_spacy = nlp(sample_text)
```

```
sents = np.array(list(text_spacy.sents))
sents
```

```
array([US unveils world's most powerful supercomputer, beats China.,
       The US has unveiled the world's most powerful supercomputer called
       'Summit', beating the previous record-holder China's Sunway TaihuLight.,
       With a peak performance of 200,000 trillion calculations per second,
       it is over twice as fast as Sunway TaihuLight, which is capable of
       93,000 trillion calculations per second.,
       Summit has 4,608 servers, which reportedly take up the size of two
       tennis courts.],
      dtype=object)
```

```
sent_words = [[word.text for word in sent] for sent in sents]
np.array(sent_words)
```

```
array([list(['US', 'unveils', 'world', "'s", 'most', 'powerful',
             'supercomputer', ',', 'beats', 'China', '.']),
       list(['The', 'US', 'has', 'unveiled', 'the', 'world', "'s", 'most',
             'powerful', 'supercomputer', 'called', "'", 'Summit', "'",
             ',', 'beating', 'the', 'previous', 'record', '-', 'holder',
             'China', "'s", 'Sunway', 'TaihuLight', '.']),
       list(['With', 'a', 'peak', 'performance', 'of', '200,000',
             'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
             'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
             ',', 'which', 'is', 'capable', 'of', '93,000', 'trillion',
             'calculations', 'per', 'second', '.']),
       list(['Summit', 'has', '4,608', 'servers', ',', 'which',
             'reportedly', 'take', 'up', 'the', 'size', 'of', 'two',
             'tennis', 'courts', '.'])],
      dtype=object)
```

```
words = [word.text for word in text_spacy]
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'", 'Summit', "'", ',', 'beating',
       'the', 'previous', 'record', '-', 'holder', 'China', "'s",
       'Sunway', 'TaihuLight', '.', 'With', 'a', 'peak', 'performance',
       'of', '200,000', 'trillion', 'calculations', 'per', 'second', ',',
       'it', 'is', 'over', 'twice', 'as', 'fast', 'as', 'Sunway',
       'TaihuLight', ',', 'which', 'is', 'capable', 'of', '93,000',
       'trillion', 'calculations', 'per', 'second', '.', 'Summit', 'has',
       '4,608', 'servers', ',', 'which', 'reportedly', 'take', 'up',
       'the', 'size', 'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

This should be more than enough to get you started with text tokenization. We encourage you to play around with more text data and see if you can make it even better!

## Removing Accented Characters

Usually in any text corpus, you might be dealing with accented characters/letters, especially if you only want to analyze the English language. Hence, we need to make sure that these characters are converted and standardized into ASCII characters. This shows a simple example—converting **é** to **e**. The following function is a simple way of tackling this task.

```
import unicodedata

def remove_accented_chars(text):
    text = unicodedata.normalize('NFKD', text).encode('ascii', 'ignore').
    decode('utf-8', 'ignore')
    return text

remove_accented_chars('Sómě Áccěntěd těxt')
```

```
'Some Accented text'
```

The preceding function shows us how we can easily convert accented characters to normal English characters, which helps standardize the words in our corpus.

## Expanding Contractions

Contractions are shortened versions of words or syllables. These exist in written and spoken forms. Shortened versions of existing words are created by removing specific letters and sounds. In the case of English contractions, they are often created by removing one of the vowels from the word. Examples include "is not" to "isn't" and "will not" to "won't", where you can notice the apostrophe being used to denote the contraction and some of the vowels and other letters being removed.

Contractions are often avoided when in formal writing, but are used quite extensively in informal communication. Various forms of contractions exist and they are tied to the type of auxiliary verbs, which give us normal contractions, negated contractions, and other special colloquial contractions, some of which may not involve auxiliaries.

By nature, contractions pose a problem for NLP and text analytics because, to start with, we have a special apostrophe character in the word. Besides this, we also have two or more words represented by a contraction and this opens a whole new can of worms when we try to tokenize them or standardize the words. Hence, there should be some definite process for dealing with contractions when processing text.

Ideally, you can have a proper mapping for contractions and their corresponding expansions and then use that to expand all the contractions in your text. I have created a vocabulary for contractions and their corresponding expanded forms, which you can access in the file named `contractions.py` in a Python dictionary (available along with the code files for this chapter). A part of the contractions dictionary is shown in the following snippet.

```
CONTRACTION_MAP = {
    "ain't": "is not",
    "aren't": "are not",
    "can't": "cannot",
    "can't've": "cannot have",
     .
     .
     .
    "you'll've": "you will have",
    "you're": "you are",
    "you've": "you have"
}
```

Remember, however, that some of the contractions can have multiple forms, such the contraction "you'll" which can be either "you will" or "you shall". To make things simple here, we use only one of these expanded forms for each contraction. For our next step, to expand contractions, we use the following code snippet.

```
from contractions import CONTRACTION_MAP
import re

def expand_contractions(text, contraction_mapping=CONTRACTION_MAP):

    contractions_pattern = re.compile('({})'.format('|'.join(contraction_
    mapping.keys())), flags=re.IGNORECASE|re.DOTALL)
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match)\
                               if contraction_mapping.get(match)\
                               else contraction_mapping.get(match.lower())
        expanded_contraction = first_char+expanded_contraction[1:]
        return expanded_contraction

    expanded_text = contractions_pattern.sub(expand_match, text)
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text
```

In this snippet, we use the `expanded_match` function inside the main `expand_contractions` function to find each contraction that matches the regex pattern we create out of all the contractions in our `CONTRACTION_MAP` dictionary. On matching any contraction, we substitute it with its corresponding expanded version and retain the correct case of the word. Let's see this process in action now!

```
expand_contractions("Y'all can't expand contractions I'd think")
```

```
'You all cannot expand contractions I would think'
```

We can see how our function helps expand the contractions from the preceding output. Are there better ways of doing this? Definitely! If we have enough examples, we can even train a deep learning model for better performance.

## Removing Special Characters

Special characters and symbols are usually non-alphanumeric characters or even occasionally numeric characters (depending on the problem), which add to the extra noise in unstructured text. Usually, simple regular expressions (regexes) can be used to remove them. The following code helps us remove special characters.

```
def remove_special_characters(text, remove_digits=False):
    pattern = r'[^a-zA-z0-9\s]' if not remove_digits else r'[^a-zA-z\s]'
    text = re.sub(pattern, '', text)
    return text

remove_special_characters("Well this was fun! What do you think? 123#@!",
                          remove_digits=True)
```

```
'Well this was fun What do you think '
```

I've kept removing digits optional, because often we might need to keep them in the preprocessed text.

## Case Conversions

Often you might want to modify the case of words or sentences to make things easier, like matching specific words or tokens. Usually, there are two types of case conversion operations that are used a lot. These are lower- and uppercase conversions, where a body of text is converted completely to lowercase or uppercase. There are other forms also like sentence case or title case. Lowercase is a form where all the letters of the text are small letters and in uppercase they are all capitalized. Title case will capitalize the first letter of each word in the sentence. The following snippet illustrates these concepts.

```
# lowercase
text = 'The quick brown fox jumped over The Big Dog'
text.lower()
```

```
'the quick brown fox jumped over the big dog'
```

```
# uppercase
text.upper()
```

```
'THE QUICK BROWN FOX JUMPED OVER THE BIG DOG'
```

```
# title case
text.title()
```

```
'The Quick Brown Fox Jumped Over The Big Dog'
```

# Text Correction

One of the main challenges faced in text wrangling is the presence of incorrect words in the text. The definition of *incorrect* here covers words that have spelling mistakes as well as words with several letters repeated that do not contribute much to its overall significance. To illustrate some examples, the word "finally" could be mistakenly written as "fianlly" or someone expressing intense emotion could write it as "finallllyyyyyy". The main objective here is to standardize different forms of these words to the correct form so that we do not end up losing vital information from different tokens in the text. We cover dealing with repeated characters as well as correcting spellings in this section.

## Correcting Repeating Characters

We just mentioned words that often contain several repeating characters that could be due to incorrect spellings, slang language, or even people wanting to express strong emotions. We show a method here that uses a combination of syntax and semantics to correct these words. We start by correcting the syntax of these words and then move on to semantics.

The first step in our algorithm is to identify repeated characters in a word using a regex pattern and then use a substitution to remove the characters one by one. Let's consider the word "finalllyyy" from the earlier example. The pattern `r'(\w*)(\w)\2(\w*)'` can be used to identify characters that occur twice among other characters in the word. In each step, we try to eliminate one of the repeated characters using a substitution for the match by utilizing the regex match groups (groups 1, 2, and 3) using the pattern `r'\1\2\3'`. Then we keep iterating through this process until no repeated characters remain. The following snippet illustrates this process.

```
old_word = 'finalllyyy'
repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
match_substitution = r'\1\2\3'
step = 1
```

```
while True:
    # remove one repeated character
    new_word = repeat_pattern.sub(match_substitution,
                                  old_word)
    if new_word != old_word:
         print('Step: {} Word: {}'.format(step, new_word))
         step += 1 # update step
         # update old word to last substituted state
         old_word = new_word
         continue
    else:
         print("Final word:", new_word)
         break
```

```
Step: 1 Word: finalllyy
Step: 2 Word: finallly
Step: 3 Word: finally
Step: 4 Word: finaly
Final word: finaly
```

This snippet shows us how one repeated character is removed at each stage and we end up with the word "finaly" in the end. However, this word is incorrect and the correct word was "finally," which we had obtained in Step 3. We will now utilize the WordNet corpus to check for valid words at each stage and terminate the loop once it is obtained. This introduces the semantic correction needed for our algorithm, as illustrated in the following snippet.

```
from nltk.corpus import wordnet
old_word = 'finalllyyy'
repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
match_substitution = r'\1\2\3'
step = 1
```

```
while True:
    # check for semantically correct word
    if wordnet.synsets(old_word):
        print("Final correct word:", old_word)
        break
```

```
    # remove one repeated character
    new_word = repeat_pattern.sub(match_substitution,
                                  old_word)
    if new_word != old_word:
        print('Step: {} Word: {}'.format(step, new_word))
        step += 1 # update step
        # update old word to last substituted state
        old_word = new_word
        continue
    else:
        print("Final word:", new_word)
        break
```

```
Step: 1 Word: finalllyy
Step: 2 Word: finallly
Step: 3 Word: finally
Final correct word: finally
```

Thus, we see from this snippet that the code correctly terminated after the third step and we obtained the correct word adhering to both syntax and semantics. We can build a better version of this code by writing the logic in a function, as depicted here, to make it more generic to deal with incorrect tokens from a list of tokens.

```
from nltk.corpus import wordnet

def remove_repeated_characters(tokens):
    repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
    match_substitution = r'\1\2\3'
    def replace(old_word):
        if wordnet.synsets(old_word):
            return old_word
        new_word = repeat_pattern.sub(match_substitution, old_word)
        return replace(new_word) if new_word != old_word else new_word

    correct_tokens = [replace(word) for word in tokens]
    return correct_tokens
```

In this snippet, we use the inner function `replace()` to basically emulate the behavior of our algorithm that we illustrated earlier and then call it repeatedly on each token in a sentence in the outer function `remove_repeated_characters()`. We can see the code in action in the following snippet with an example sentence.

```
sample_sentence = 'My schooool is reallllllyyy amaaazingggg'
correct_tokens = remove_repeated_characters(nltk.word_tokenize(sample_
sentence))
' '.join(correct_tokens)
```

```
'My school is really amazing'
```

We can see from this output that our function performs as intended and replaces the repeating characters in each token, giving us correct tokens as desired.

## Correcting Spellings

The second problem we face with words is incorrect or wrong spellings that occur due to human error and even machine based errors, which you might have seen with features like auto-correcting text. There are various ways to deal with incorrect spellings where the final objective is to have tokens of text with the correct spelling. We will talk about one of the famous algorithms developed by Peter Norvig, the director of research at Google. You can find the complete detailed post explaining his algorithm and findings at http://norvig.com/spell-correct.html, which we will be exploring in this section.

The main objective of this exercise is that given a problematic word, we need to find the most likely correct form of that word. The approach we follow is to generate a set of candidate words that are near to our input word and select the most likely word from this set as the correct word. We use a corpus of correct English words in this context to identify the correct word based on its frequency in the corpus from our final set of candidates with the nearest distance to our input word. This distance measures how near or far a word is from our input word and is also called the *edit distance*.

The input corpus we use is a file with several books from the Gutenberg corpus and a list of most frequent words from Wiktionary and the British National Corpus. You can find the file under the name `big.txt` in this chapter's code resources or you can download it from Norvig's direct link at http://norvig.com/big.txt and use it. We use the following code snippet to generate a map of frequently occurring words in the English language and their counts.

```
import re, collections

def tokens(text):
    """
    Get all words from the corpus
    """
    return re.findall('[a-z]+', text.lower())

WORDS = tokens(open('big.txt').read())
WORD_COUNTS = collections.Counter(WORDS)
# top 10 words in corpus
WORD_COUNTS.most_common(10)
```

```
[('the', 80030), ('of', 40025), ('and', 38313), ('to', 28766), ('in', 22050),
 ('a', 21155), ('that', 12512), ('he', 12401), ('was', 11410), ('it', 10681)]
```

Once we have our vocabulary, we define three functions that compute sets of words that are zero, one, and two edits away from our input word. These edits can be made by the means of insertions, deletions, additions, and transpositions. The following code defines the functions.

```
def edits0(word):
    """
    Return all strings that are zero edits away
    from the input word (i.e., the word itself).
    """
    return {word}

def edits1(word):
    """
    Return all strings that are one edit away
    from the input word.
    """
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    def splits(word):
        """
        Return a list of all possible (first, rest) pairs
        that the input word is made of.
        """
```

```
        return [(word[:i], word[i:])
                for i in range(len(word)+1)]

    pairs      = splits(word)
    deletes    = [a+b[1:]           for (a, b) in pairs if b]
    transposes = [a+b[1]+b[0]+b[2:] for (a, b) in pairs if len(b) > 1]
    replaces   = [a+c+b[1:]         for (a, b) in pairs for c in alphabet if b]
    inserts    = [a+c+b             for (a, b) in pairs for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def edits2(word):
    """Return all strings that are two edits away
    from the input word.
    """
    return {e2 for e1 in edits1(word) for e2 in edits1(e1)}
```

We also define a function called `known()`, which returns a subset of words from our candidate set of words obtained from the edit functions based on whether they occur in our vocabulary dictionary `WORD_COUNTS`. This gives us a list of valid words from our set of candidate words.

```
def known(words):
    """
    Return the subset of words that are actually
    in our WORD_COUNTS dictionary.
    """
    return {w for w in words if w in WORD_COUNTS}
```

We can see these functions in action on our test input word in the following code snippet, which shows lists of possible candidate words based on edit distances from the input word.

```
# input word
word = 'fianlly'
# zero edit distance from input word
edits0(word)
```

```
{'fianlly'}
```

```
# returns null set since it is not a valid word
known(edits0(word))
```

```
set()
```

```
# one edit distance from input word
edits1(word)
```

```
{'afianlly',
 'aianlly',
 .
 .
'yianlly',
'zfianlly',
'zianlly'}
```

```
# get correct words from above set
known(edits1(word))
```

```
{'finally'}
```

```
# two edit distances from input word
edits2(word)
```

```
{'fchnlly',
 'fianjlys',
  .
  .
 'fiapgnlly',
 'finanlqly'}
```

```
# get correct words from above set
known(edits2(word))
```

```
{'faintly', 'finally', 'finely', 'frankly'}
```

This output shows a set of valid candidate words that could be potential replacements for the incorrect input word. We select our candidate words from the list by giving higher priority to words whose edit distances are the smallest from the input word. The following code snippet illustrates this.

```
candidates = (known(edits0(word)) or
              known(edits1(word)) or
              known(edits2(word)) or
              [word])
candidates
```

```
{'finally'}
```

In case there is a tie in the candidates, we resolve it by taking the highest occurring word from our vocabulary dictionary `WORD_COUNTS` using the `max(candidates, key=WORD_COUNTS.get)` function. Thus, we now define our function to correct words using this logic.

```
def correct(word):
    """
    Get the best correct spelling for the input word
    """
    # Priority is for edit distance 0, then 1, then 2
    # else defaults to the input word itself.
    candidates = (known(edits0(word)) or
                  known(edits1(word)) or
                  known(edits2(word)) or
                  [word])
    return max(candidates, key=WORD_COUNTS.get)
```

We can use the function on incorrect words directly to correct them, as illustrated in the following snippet.

```
correct('fianlly')
```

```
'finally'
```

```
correct('FIANLLY')
```

```
'FIANLLY'
```

We see that this function is case sensitive and fails to correct words that are not lowercase, hence we write the following functions to make this generic to the case of words and correct their spelling regardless. The logic here is to preserve the original case of the word, convert it to lowercase, correct its spelling, and finally convert it back to its original case using the `case_of` function.

```
def correct_match(match):
    """
    Spell-correct word in match,
    and preserve proper upper/lower/title case.
    """
    word = match.group()
    def case_of(text):
        """
        Return the case-function appropriate
        for text: upper, lower, title, or just str.:
            """
        return (str.upper if text.isupper() else
                str.lower if text.islower() else
                str.title if text.istitle() else
                str)
    return case_of(word)(correct(word.lower()))

def correct_text_generic(text):
    """
    Correct all the words within a text,
    returning the corrected text.
    """
    return re.sub('[a-zA-Z]+', correct_match, text)
```

We can now use the function to correct words irrespective of their case, as illustrated in the following snippet.

```
correct_text_generic('fianlly')
```

```
'finally'
```

```
correct_text_generic('FIANLLY')
```

```
'FINALLY'
```

Of course this method is not always completely accurate and there might be words that are not corrected if they do not occur in our vocabulary dictionary. Using more data would help in this case as long as we cover different words with correct spellings in our vocabulary. This same algorithm is available to be used out-of-the-box in the TextBlob library. This is depicted in the following snippet.

```
from textblob import Word
w = Word('fianlly')
w.correct()
```

```
'finally'
```

```
# check suggestions
w.spellcheck()
```

```
[('finally', 1.0)]
```

```
# another example
w = Word('flaot')
w.spellcheck()
```

```
[('flat', 0.85), ('float', 0.15)]
```

Besides this, there are several robust libraries available in Python, including `PyEnchant` based on the `enchant` library (`http://pythonhosted.org/pyenchant/`), autocorrect, which is available at `https://github.com/phatpiglet/autocorrect/`, and `aspell-python`, which is a Python wrapper around the popular GNU Aspell. With the advent of deep learning, sequential models like RNNs and LSTMs coupled with word embeddings often out-perform these traditional methods. I also recommend readers take a look at DeepSpell, which is available at `https://github.com/MajorTal/DeepSpell`. It leverages deep learning to build a spelling corrector. Feel free to check them out and use them for correcting word spellings!

## Stemming

To understand the process of stemming, we need to understand what word stems represent. In Chapter 1, we talked about morphemes, which are the smallest independent unit in any natural language. Morphemes consist of units that are stems and affixes. Affixes are units like prefixes, suffixes, and so on, which are attached to word

stems to change their meaning or create a new word altogether. Word stems are also often known as the base form of a word and we can create new words by attaching affixes to them. This process is known as *inflection*. The reverse of this is obtaining the base form of a word from its inflected form and this is known as *stemming*.

Consider the word "JUMP", you can add affixes to it and form several new words like "JUMPS", "JUMPED", and "JUMPING". In this case, the base word is "JUMP" and this is the word stem. If we were to carry out stemming on any of its three inflected forms, we would get the base form. This is depicted more clearly in Figure 3-2.

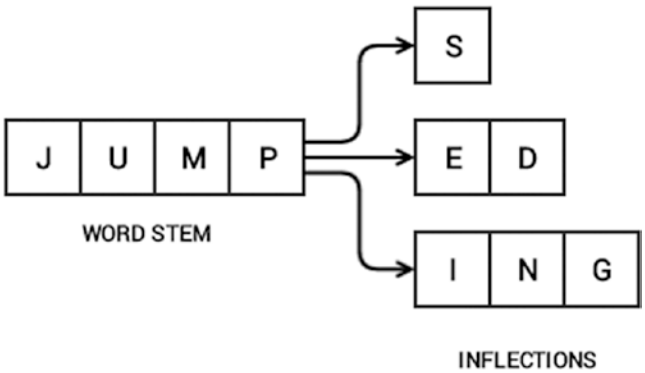


***Figure 3-2.*** *Word stem and inflections*

Figure 3-2 depicts how the word stem is present in all its inflections since it forms the base on which each inflection is built upon using affixes. Stemming helps us standardize words to their base stem irrespective of their inflections, which helps many applications like classifying or clustering text or even in information retrieval. Search engines use such techniques extensively to give better accurate results irrespective of the word form. The NLTK package has several implementations for stemmers. These stemmers are implemented in the `stem` module, which inherits the `StemmerI` interface in the `nltk.stem.api` module. You can even create your own stemmer by using this class (technically it is an interface) as your base class. One of the most popular stemmers is the Porter stemmer, which is based on the algorithm developed by its inventor, Martin Porter. Originally, the algorithm is said to have a total of five different phases for reduction of inflections to their stems, where each phase has its own set of rules. There also exists a Porter2 algorithm, which was the original stemming algorithm with some improvements suggested by Dr. Martin Porter. You can see the Porter stemmer in action in the following code snippet.

```
# Porter Stemmer
In [458]: from nltk.stem import PorterStemmer
     ...: ps = PorterStemmer()
     ...: ps.stem('jumping'), ps.stem('jumps'), ps.stem('jumped')
(jump, jump, jump)

In [460]: ps.stem('lying')
'lie'

In [461]: ps.stem('strange')
'strang'
```

The Lancaster stemmer is based on the Lancaster stemming algorithm, also often known as the Paice/Husk stemmer, which was invented by Chris D. Paice. This stemmer is an iterative stemmer with over 120 rules, which specify specific removal or replacement for affixes to obtain the word stems. The following snippet shows the Lancaster stemmer in action.

```
# Lancaster Stemmer
In [465]: from nltk.stem import LancasterStemmer
     ...: ls = LancasterStemmer()
     ...:  print ls.stem('jumping'), ls.stem('jumps'), ls.stem('jumped')
(jump, jump, jump)

In [467]: ls.stem('lying')
'lying'

In [468]: ls.stem('strange')
'strange'
```

You can see the behavior of this stemmer is different from the previous Porter stemmer. Besides these two, there are several other stemmers, including `RegexpStemmer`, where you can build your own stemmer based on user-defined rules and `SnowballStemmer`, which supports stemming in 13 different languages besides English. The following code snippet shows some ways of using them for performing stemming. The `RegexpStemmer` uses regular expressions to identify the morphological affixes in words and any part of the string matching them is removed.

```
# Regex based stemmer
In [471]: from nltk.stem import RegexpStemmer
     ...: rs = RegexpStemmer('ing$|s$|ed$', min=4)
     ...:  rs.stem('jumping'), rs.stem('jumps'), rs.stem('jumped')
(jump, jump, jump)

In [473]: rs.stem('lying')
'ly'

In [474]: rs.stem('strange')
'strange'
```

You can see how the stemming results are different from the previous stemmers and is based completely on our custom defined rules based on regular expressions. The following snippet shows how we can use the `SnowballStemmer` to stem words in other languages. You can find more details about the Snowball Project at http://snowballstem.org/.

```
# Snowball Stemmer
In [486]: from nltk.stem import SnowballStemmer
     ...: ss = SnowballStemmer("german")
     ...: print('Supported Languages:', SnowballStemmer.languages)
Supported Languages: (u'danish', u'dutch', u'english', u'finnish',
u'french', u'german', u'hungarian', u'italian', u'norwegian', u'porter',
u'portuguese', u'romanian', u'russian', u'spanish', u'swedish')

# stemming on German words
# autobahnen -> cars
# autobahn -> car
In [488]: ss.stem('autobahnen')
'autobahn'

# springen -> jumping
# spring -> jump
In [489]: ss.stem('springen')
'spring'
```

The Porter stemmer is used most frequently, but you should choose your stemmer based on your problem and after trial and error. The following is a basic function that can be used for stemming text.

```
def simple_stemmer(text):
    ps = nltk.porter.PorterStemmer()
    text = ' '.join([ps.stem(word) for word in text.split()])
    return text

simple_stemmer("My system keeps crashing his crashed yesterday, ours
crashes daily")

'My system keep crash hi crash yesterday, our crash daili'
```

Feel free to leverage this function for your own stemming needs. Also, if needed, you can even build your own stemmer with your own defined rules!

## Lemmatization

The process of lemmatization is very similar to stemming, where we remove word affixes to get to a base form of the word. However in this case, this base form is also known as the *root* word but not the root stem. The difference between the two is that the root stem may not always be a lexicographically correct word, i.e., it may not be present in the dictionary but the root word, also known as the lemma, will always be present in the dictionary.

The lemmatization process is considerably slower than stemming because an additional step is involved where the root form or lemma is formed by removing the affix from the word if and only if the lemma is present in the dictionary. The NLTK package has a robust lemmatization module where it uses WordNet and the word's syntax and semantics like part of speech and context to get the root word or lemma. Remember from Chapter 1 when we discussed parts of speech? There were three entities of nouns, verbs, and adjectives that occur most frequently in natural language. The following code snippet depicts how to use lemmatization for words belonging to each of those types.

```
In [514]: from nltk.stem import WordNetLemmatizer
     ...: wnl = WordNetLemmatizer()

# lemmatize nouns
In [515]: print(wnl.lemmatize('cars', 'n'))
     ...: print(wnl.lemmatize('men', 'n'))
car
men

# lemmatize verbs
In [516]: print(wnl.lemmatize('running', 'v'))
     ...: print(wnl.lemmatize('ate', 'v'))
run
eat

# lemmatize adjectives
In [517]: print(wnl.lemmatize('saddest', 'a'))
     ...: print(wnl.lemmatize('fancier', 'a'))
sad
fancy
```

This snippet shows us how each word is converted to its base form using lemmatization. This helps us standardize words. This code leverages the `WordNetLemmatizer` class, which internally uses the `morphy()` function belonging to the `WordNetCorpusReader` class. This function basically finds the base form or lemma for a given word using the word and its part of speech by checking the WordNet corpus and uses a recursive technique for removing affixes from the word until a match is found in WordNet. If no match is found, the input word is returned unchanged. The part of speech is extremely important because if that is wrong, the lemmatization will not be effective, as you can see in the following snippet.

```
# ineffective lemmatization
In [518]: print wnl.lemmatize('ate', 'n')
     ...: print wnl.lemmatize('fancier', 'v')
ate
fancier
```

SpaCy makes things a lot easier since it performs parts of speech tagging and effective lemmatization for each token in a text document without you worrying about if you are using lemmatization effectively. The following function can be leveraged for performing effective lemmatization, thanks to spaCy!

```
import spacy
nlp = spacy.load('en_core', parse=True, tag=True, entity=True)
text = 'My system keeps crashing his crashed yesterday, ours crashes daily'

def lemmatize_text(text):
    text = nlp(text)
    text = ' '.join([word.lemma_ if word.lemma_ != '-PRON-' else word.text
    for word in text])
    return text

lemmatize_text("My system keeps crashing! his crashed yesterday, ours
crashes daily")

'My system keep crash ! his crash yesterday , ours crash daily'
```

You can leverage NLTK or spaCy to build your own lemmatizers. Feel free to experiment with these functions on your own data.

## Removing Stopwords

*Stopwords* are words that have little or no significance and are usually removed from text when processing it so as to retain words having maximum significance and context. Stopwords usually occur most frequently if you aggregate a corpus of text based on singular tokens and checked their frequencies. Words like "a," "the," "and," and so on are stopwords. There is no universal or exhaustive list of stopwords and often each domain or language has its own set of stopwords. We depict a method to filter out and remove stopwords for English in the following code snippet.

```
from nltk.tokenize.toktok import ToktokTokenizer
tokenizer = ToktokTokenizer()
stopword_list = nltk.corpus.stopwords.words('english')
def remove_stopwords(text, is_lower_case=False):
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
```

```
    if is_lower_case:
        filtered_tokens = [token for token in tokens if token not in
        stopword_list]
    else:
        filtered_tokens = [token for token in tokens if token.lower() not
        in stopword_list]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text

remove_stopwords("The, and, if are stopwords, computer is not")
```

```
', , stopwords , computer'
```

There is no universal stopword list, but we use a standard English language stopwords list from NLTK. You can also add your own domain-specific stopwords as needed. In the previous function, we leverage the use of NLTK, which has a list of stopwords for English, and use it to filter out all tokens that correspond to stopwords. This output shows us a reduced number of tokens compared to what we had earlier and you can compare and check the tokens that were removed as stopwords. To see the list of all English stopwords in NLTK's vocabulary, you can print the contents of `nltk.corpus.stopwords.words('english')` to get an idea of the various stopwords. One important thing to remember is that negations like "not" and "no" are removed in this case (in the first sentence) and often it is essential to preserve them so as the actual meaning of the sentence is not lost in applications like sentiment analysis. So you would need to make sure you do not remove these words in those scenarios.

## Bringing It All Together—Building a Text Normalizer

Let's now bring everything we learned together and chain these operations to build a text normalizer to preprocess text data. We focus on including the major components often used for text wrangling in our custom function.

```
def normalize_corpus(corpus, html_stripping=True, contraction_expansion=True,
                     accented_char_removal=True, text_lower_case=True,
                     text_lemmatization=True, special_char_removal=True,
                     stopword_removal=True, remove_digits=True):
```

```
    normalized_corpus = []
    # normalize each document in the corpus
    for doc in corpus:
        # strip HTML
        if html_stripping:
            doc = strip_html_tags(doc)
        # remove accented characters
        if accented_char_removal:
            doc = remove_accented_chars(doc)
        # expand contractions
        if contraction_expansion:
            doc = expand_contractions(doc)
        # lowercase the text
        if text_lower_case:
            doc = doc.lower()
        # remove extra newlines
        doc = re.sub(r'[\r|\n|\r\n]+', ' ',doc)
        # lemmatize text
        if text_lemmatization:
            doc = lemmatize_text(doc)
        # remove special characters and\or digits
        if special_char_removal:
            # insert spaces between special characters to isolate them
            special_char_pattern = re.compile(r'([{.(-)!}])')
            doc = special_char_pattern.sub(" \\1 ", doc)
            doc = remove_special_characters(doc, remove_digits=remove_digits)
        # remove extra whitespace
        doc = re.sub(' +', ' ', doc)
        # remove stopwords
        if stopword_removal:
            doc = remove_stopwords(doc, is_lower_case=text_lower_case)

        normalized_corpus.append(doc)

    return normalized_corpus
```

Let's now put this function in action! We will leverage our sample text from the previous sections as the input document, which we will preprocess using the preceding function.

```
{'Original': sample_text,
 'Processed': normalize_corpus([sample_text])[0]}
```

```
{'Original': "US unveils world's most powerful supercomputer, beats
China. The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight.
With a peak performance of 200,000 trillion calculations per second,
it is over twice as fast as Sunway TaihuLight, which is capable of
93,000 trillion calculations per second. Summit has 4,608 servers, which
reportedly take up the size of two tennis courts.",

 'Processed': 'us unveil world powerful supercomputer beat china us unveil
world powerful supercomputer call summit beat previous record holder chinas
sunway taihulight peak performance trillion calculation per second twice
fast sunway taihulight capable trillion calculation per second summit
server reportedly take size two tennis court'}
```

Thus, you can see how our text preprocessor helps in preprocessing our sample news article! In the next section, we look at ways of analyzing and understanding various facets of textual data with regard to its syntactic properties and structure.

# Understanding Text Syntax and Structure

We talked about language syntax and structure in detail in Chapter 1. If you don't remember the basics, head over to the section titled "Language Syntax and Structure" in Chapter 1 and skim through it quickly to get an idea of the various ways of analyzing and understanding the syntax and structure of textual data. To refresh your memory, let's briefly cover the importance of text syntax and structure.

For any language, syntax and structure usually go hand in hand, where a set of specific rules, conventions, and principles govern the way words are combined into phrases; phrases are combined into clauses; and clauses are combined into sentences. We will be talking specifically about the English language syntax and structure in this section. In English, words usually combine to form other constituent units. These constituents

include words, phrases, clauses, and sentences. The sentence "The brown fox is quick and he is jumping over the lazy dog" is made of a bunch of words. Just looking at the words by themselves doesn't tell us much (see Figure 3-3).

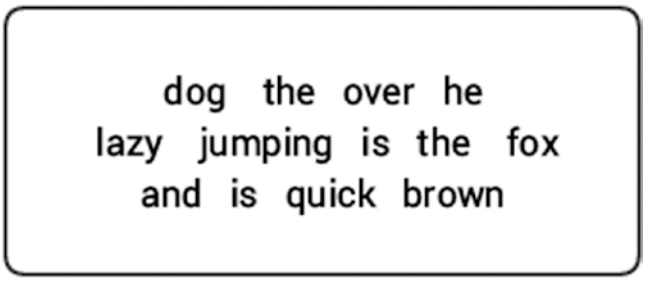


***Figure 3-3.*** *A bunch of unordered words doesn't convey much information*

Knowledge about the structure and syntax of language is helpful in many areas like text processing, annotation, and parsing for further operations such as text classification or summarization. In this section, we implement some of the concepts and techniques used to understand text syntax and structure. This is extremely useful in natural language processing and is usually done after text processing and wrangling. We focus on implementing the following techniques:

- Parts of speech (POS) tagging
- Shallow parsing or chunking
- Dependency parsing
- Constituency parsing

This book is targeted toward practitioners and enforces and emphasizes on best approaches for implementing and using techniques and algorithms in real-world problems. Hence in the following sections, we look at the best possible ways of leveraging libraries like NLTK and spaCy to implement some of these techniques. Besides this, since you might be interested in the internals and implementing some of these techniques on your own, we also look at ways to accomplish this. Before jumping into the details, we look at the necessary dependencies and installation details for the required libraries, since some of them are not very straightforward.

## Installing Necessary Dependencies

We leverage several libraries and dependencies:

- The `nltk` library
- The `spacy` library
- The Stanford Parser
- Graphviz and necessary libraries for visualization

We touched upon installing NLTK in Chapter 1. You can install it directly by going to your terminal or command prompt and typing `pip install nltk`, which will download and install it. Remember to install the library preferably equal to or higher than version 3.2.4. After downloading and installing NLTK, remember to download the corpora, which we also discussed in Chapter 1. For more details on downloading and installing NLTK, you can follow the information at `http://www.nltk.org/install.html` and `http://www.nltk.org/data.html`, which tells you how to install the data dependencies. Start the Python interpreter and use the following snippet.

```
import nltk
# download all dependencies and corpora
nltk.download('all', halt_on_error=False)
# OR use a GUI based downloader and select dependencies
nltk.download()
```

To install spaCy, type `pip install spacy` from the terminal or conda install spaCy. Once it's done, download the English language model using the command, `python -m spacy.en.download` from the terminal, which will download around 500MB of data in the directory of the spaCy package. For more details, you can refer to the link `https://spacy.io/docs/#getting-started`, which tells you how to get started with using spaCy. We will use spaCy for tagging and depicting dependency based parsing. However, in case you face issues loading spaCy's language models, feel free to follow the steps highlighted here to resolve this issue. (I faced this issue in one of my systems but it doesn't always occur.)

```
# OPTIONAL: ONLY USE IF SPACY FAILS TO LOAD LANGUAGE MODEL
# Use the following command to install spaCy
> pip install -U spacy
OR
> conda install -c conda-forge spacy

# Download the following language model and store it in disk
https://github.com/explosion/spacy-models/releases/tag/en_core_web_md-2.0.0

# Link the same to spacy
> python -m spacy link ./spacymodels/en_core_web_md-2.0.0/en_core_web_md
en_core Linking successful
    ./spacymodels/en_core_web_md-2.0.0/en_core_web_md --> ./Anaconda3/lib/
site-packages/spacy/data/en_core
You can now load the model via spacy.load('en_core')
```

The Stanford Parser is a Java-based implementation for a language parser developed at Stanford, which helps parse sentences to understand their underlying structure. We perform both dependency and constituency grammar based parsing using the Stanford Parser and NLTK, which provides an excellent wrapper to leverage and use the parser from Python itself without the need to write code in Java. You can refer to the official installation guide at https://github.com/nltk/nltk/wiki/Installing-Third-Party-Software, which tells us how to download and install the Stanford Parser and integrate it with NLTK. Personally, I faced several issues especially in Windows based systems; hence, I will provide one of the best known methods for installation of the Stanford Parser and its necessary dependencies.

To start with, make sure you download and install the Java Development Kit (not just JRE also known as Java Runtime Environment) by going to http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=otnjp. Use any version typically on or after Java SE 8u101 / 8u102. I used 8u102 since I haven't upgraded Java in a while. After installing, make sure that you have set the path for Java by adding it to the `path` system environment variable. You can also create a `JAVA_HOME` environment variable pointing to the `java.exe` file belonging to the JDK.

In my experience neither worked for me when running the code from Python and I had to explicitly use the Python `os` library to set the environment variable, which I will be

showing when we dive into the implementation details. Once Java is installed, download the official Stanford Parser from http://nlp.stanford.edu/software/stanford-parser-full-2015-04-20.zip, which seems to work quite well. You can try a later version by going to http://nlp.stanford.edu/software/lex-parser.shtml#Download and checking the "Release History" section. After downloading, unzip it to a known location in your filesystem. Once you're done, you are now ready to use the parser from NLTK, which we will be exploring soon!

Graphviz is not a necessity and we will only be using it to view the dependency parse tree generated by the Stanford Parser. You can download Graphviz from its official website at http://www.graphviz.org/Download_windows.php and install it. Next you need to install `pygraphviz`, which you can get by downloading the wheel file from http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygraphviz based on your system architecture and Python version. Then install it using the `pip install pygraphviz-1.3.1-cp34-none-win_amd64.whl` command for a 64-bit system running Python 3.x. Once it's installed, `pygraphviz` should be ready to work. Some people reported running into additional issues and you might need to install `pydot-ng` and `graphviz` in the same order using the following snippet in the terminal.

```
pip install pydot-ng
pip install graphviz
```

We also leverage NLTK's plotting capabilities to visualize parse trees in Jupyter notebooks. To enable this, you might need to install `ghostscript` in case NLTK throws an error. Instructions for installation and setup are depicted as follows.

```
## download and install ghostscript from https://www.ghostscript.com/
download/gsdnld.html

# often need to add to the path manually (for windows)
os.environ['PATH'] = os.environ['PATH']+";C:\\Program Files\\gs\\gs9.09\\bin\\"
```

With this, we are done with installing our necessary dependencies and can start implementing and looking at practical examples. However, we are not ready just yet. We need to go through a few basic concepts of machine learning before we dive into the code and examples.

## Important Machine Learning Concepts

We will be implementing and training some of our own taggers in the following section using corpora and leverage existing taggers. There are some important concepts related to analytics and machine learning, which you must know to understand the implementations more clearly.

- **Data preparation:** Usually consists of preprocessing the data before extracting features and training
- **Feature extraction:** The process of extracting useful features from raw data that are used to train machine learning models
- **Features:** Various useful attributes of the data (examples could be age, weight, and so on for personal data)
- **Training data:** A set of data points used to train a model
- **Testing/validation data:** A set of data points on which a pretrained model is tested and evaluated to see how well it performs
- **Model:** This is built using a combination of data/features and a machine learning algorithm that could be supervised or unsupervised
- **Accuracy:** How well the model predicts something (also has other detailed evaluation metrics like precision, recall, and F1-score)

These terms should be enough to get you started. Going into details is beyond the current scope; however, you will find a lot of resources on the web on machine learning if you are interested in exploring some of them further. We recommend checking out *Practical Machine Learning with Python*, Apress 2018, if you are interested in learning machine learning using a hands-on approach. Besides this, we cover supervised and unsupervised learning with regards to textual data in subsequent chapters.

## Parts of Speech Tagging

Parts of speech (POS) are specific lexical categories to which words are assigned based on their syntactic context and role. If you remember from Chapter 1, we covered some ground on POS, where we mentioned the main POS being nouns, verbs, adjectives, and adverbs. The process of classifying and labeling POS tags for words is defined as parts of speech tagging (POS tagging).

POS tags are used to annotate words and depict their POS, which is really helpful when we need to use the same annotated text later in NLP-based applications because we can filter by specific parts of speech an-d utilize that information to perform specific analysis. We can narrow down nouns and determine which ones are the most prominent. Considering our previous example sentence, "The brown fox is quick and he is jumping over the lazy dog", if we were to annotate it using basic POS tags, it would look like Figure 3-4.

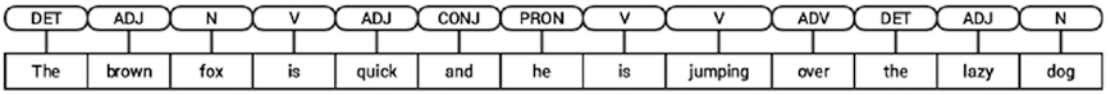


***Figure 3-4.*** *POS tagging for a sentence*

Thus, a sentence typically follows a hierarchical structure consisting of the following components: sentence → clauses → phrases → words.

We will be using the Penn Treebank notation for POS tagging and most of the recommended POS taggers also leverage it. You can find out more information about various POS tags and their notation at `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/Penn-Treebank-Tagset.pdf`, which contains detailed documentation explaining each tag with examples. The Penn Treebank project is a part of the University of Pennsylvania and their web page can be found at `https://catalog.ldc.upenn.edu/docs/LDC95T7/treebank2.index.html`, which gives more information about the project. Remember there are various tags, such as POS tags for parts of speech assigned to words, chunk tags, which are usually assigned to phrases, and some tags are secondary tags, which are used to depict relations.

Table 3-1 provides a detailed overview of different tags with examples in case you do not want to go through the detailed documentation for Penn Treebank tags. You can use this as a reference anytime to understand POS tags and parse trees in a better way.

***Table 3-1.*** *Parts of Speech Tags*

| Sl No. | TAG | DESCRIPTION | EXAMPLE(S) |
|---|---|---|---|
| 1 | CC | Coordinating conjunction | and, or |
| 2 | CD | Cardinal number | five, one, 2 |
| 3 | DT | Determiner | a, the |
| 4 | EX | Existential *there* | there were two cars |
| 5 | FW | Foreign word | d'hoevre, mais |
| 6 | IN | Preposition/subordinating conjunction | of, in, on, that |
| 7 | JJ | Adjective | quick, lazy |
| 8 | JJR | Adjective, comparative | quicker, lazier |
| 9 | JJS | Adjective, superlative | quickest, laziest |
| 10 | LS | List item marker | 2) |
| 11 | MD | Verb, modal | could, should |
| 12 | NN | Noun, singular or mass | fox, dog |
| 13 | NNS | Noun, plural | foxes, dogs |
| 14 | NNP | Noun, proper singular | John, Alice |
| 15 | NNPS | Noun, proper plural | Vikings, Indians, Germans |
| 16 | PDT | Predeterminer | both cats |
| 17 | POS | Possessive ending | boss's |
| 18 | PRP | Pronoun, personal | me, you |
| 19 | PRP$ | Pronoun, possessive | our, my, your |
| 20 | RB | Adverb | naturally, extremely, hardly |
| 21 | RBR | Adverb, comparative | better |
| 22 | RBS | Adverb, superlative | best |
| 23 | RP | Adverb, particle | about, up |
| 24 | SYM | Symbol | %, $ |
| 25 | TO | Infinitival to | how to, what to do |

(*continued*)

***Table 3-1.*** (*continued*)

| Sl No. | TAG | DESCRIPTION | EXAMPLE(S) |
|---|---|---|---|
| 26 | UH | Interjection | oh, gosh, wow |
| 27 | VB | Verb, base form | run, give |
| 28 | VBD | Verb, past tense | ran, gave |
| 29 | VBG | Verb, gerund/present participle | running, giving |
| 30 | VBN | Verb, past participle | given |
| 31 | VBP | Verb, non-third person singular present | I think, I take |
| 32 | VBZ | Verb, third person singular present | he thinks, he takes |
| 33 | WDT | Wh-determiner | which, whatever |
| 34 | WP | Wh-pronoun, personal | who, what |
| 35 | WP$ | Wh-pronoun, possessive | whose |
| 36 | WRB | Wh-adverb | where, when |
| 37 | NP | Noun phrase | the brown fox |
| 38 | PP | Prepositional phrase | in between, over the dog |
| 39 | VP | Verb phrase | was jumping |
| 40 | ADJP | Adjective phrase | warm and snug |
| 41 | ADVP | Adverb phrase | also |
| 42 | SBAR | Subordinating conjunction | whether or not |
| 43 | PRT | Particle | up |
| 44 | INTJ | Interjection | hello |
| 45 | PNP | Prepositional noun phrase | over the dog, as of today |
| 46 | -SBJ | Sentence subject | the fox jumped over the dog |
| 47 | -OBJ | Sentence object | the fox jumped over the dog |

This table shows us the main POS tag set used in the Penn Treebank and is the most widely used POS tag set in various text analytics and NLP applications. In the following sections, we look at some hands-on implementations of POS tagging.

## Building POS Taggers

We will be leveraging NLTK and spaCy, which use the *Penn Treebank notation* for POS tagging. To demonstrate how things work, we will leverage a news headline from our sample news article from the previous sections. Let's look at how POS tagging can be implemented using spaCy. See Figure 3-5.

```
sentence = "US unveils world's most powerful supercomputer, beats China."
```

```
import pandas as pd
import spacy
nlp = spacy.load('en_core', parse=True, tag=True, entity=True)
sentence_nlp = nlp(sentence)
# POS tagging with Spacy
spacy_pos_tagged = [(word, word.tag_, word.pos_) for word in sentence_nlp]
pd.DataFrame(spacy_pos_tagged, columns=['Word', 'POS tag', 'Tag type']).T
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Word** | US | unveils | world | 's | most | powerful | supercomputer | , | beats | China | . |
| **POS tag** | NNP | VBZ | NN | POS | RBS | JJ | NN | , | VBZ | NNP | . |
| **Tag type** | PROPN | VERB | NOUN | PART | ADV | ADJ | NOUN | PUNCT | VERB | PROPN | PUNCT |

***Figure 3-5.*** *POS tagging for our news headline using spaCy*

Thus, we can clearly see in Figure 3-5 the POS tag for each token in our sample news headline, as defined using spaCy, and they make perfect sense. Let's try to perform the same task using NLTK (see Figure 3-6).

```
# POS tagging with nltk
import nltk
nltk_pos_tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
pd.DataFrame(nltk_pos_tagged, columns=['Word', 'POS tag']).T
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Word** | US | unveils | world | 's | most | powerful | supercomputer | , | beats | China | . |
| **POS tag** | NNP | JJ | NN | POS | RBS | JJ | NN | , | VBZ | NNP | . |

***Figure 3-6.*** *POS tagging for our news headline using NLTK*

The output in Figure 3-6 gives us tags that purely follow the Penn Treebank format specifying the specific form of adjective, noun, or verbs in more detail.

We will now explore some techniques to build our own POS taggers! We leverage some classes provided by NLTK. To evaluate the performance of our taggers, we use some test data from the `treebank` corpus in NLTK. We will also be using some training data for training some of our taggers. To start with, we will first get the necessary data for training and evaluating the taggers by reading in the tagged `treebank` corpus.

```
from nltk.corpus import treebank
data = treebank.tagged_sents()
train_data = data[:3500]
test_data = data[3500:]
print(train_data[0])
```

```
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years',
'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the',
'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'),
('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')]
```

We will use the test data to evaluate our taggers and see how they work on our sample sentence by using its tokens as input. All the taggers we will be leveraging from NLTK are a part of the `nltk.tag` package. Each tagger is a child class of the base `TaggerI` class and each tagger implements a `tag()` function, which takes a list of sentence tokens as input and returns the same list of words with their POS tags as output. Besides tagging, there is also an `evaluate()` function, which is used to evaluate the performance of the tagger. This is done by tagging each input test sentence and then comparing the result with the actual tags of the sentence. We will be using the same function to test the performance of our taggers on `test_data`.

We will first look at the `DefaultTagger`, which inherits from the `SequentialBackoffTagger` base class and assigns the same user input POS tag to each word. This might seem to be really naïve but it is an excellent way to form a baseline POS tagger and improve upon it.

```
# default tagger
from nltk.tag import DefaultTagger
dt = DefaultTagger('NN')
```

```
# accuracy on test data
dt.evaluate(test_data)
```

```
0.1454158195372253
```

```
# tagging our sample headline
dt.tag(nltk.word_tokenize(sentence))
```

```
[('US', 'NN'), ('unveils', 'NN'), ('world', 'NN'), ("'s", 'NN'), ('most', 'NN'),
 ('powerful', 'NN'), ('supercomputer', 'NN'), (',', 'NN'), ('beats', 'NN'),
 ('China', 'NN'), ('.', 'NN')]
```

We can see from this output we have obtained 14% accuracy in correctly tagging words from the treebank test dataset, which is not great. The output tags on our sample sentence are all nouns, just like we expected since we fed the tagger with the same tag. We will now use regular expressions and the `RegexpTagger` to see if we can build a better performing tagger.

```
# regex tagger
from nltk.tag import RegexpTagger
# define regex tag patterns
patterns = [
        (r'.*ing$', 'VBG'),               # gerunds
        (r'.*ed$', 'VBD'),                # simple past
        (r'.*es$', 'VBZ'),                # 3rd singular present
        (r'.*ould$', 'MD'),               # modals
        (r'.*\'s$', 'NN$'),               # possessive nouns
        (r'.*s$', 'NNS'),                 # plural nouns
        (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
        (r'.*', 'NN')                     # nouns (default) ...
]
rt = RegexpTagger(patterns)
```

```
# accuracy on test data
rt.evaluate(test_data)
```

```
0.24039113176493368
```

```
# tagging our sample headline
```

```
rt.tag(nltk.word_tokenize(sentence))
```

```
[('US', 'NN'), ('unveils', 'NNS'), ('world', 'NN'), ("'s", 'NN$'), ('most', 'NN'),
 ('powerful', 'NN'), ('supercomputer', 'NN'), (',', 'NN'), ('beats', 'NNS'),
 ('China', 'NN'), ('.', 'NN')]
```

This output shows us that the accuracy has now increased to 24%, but can we do better? We will now train some n-gram taggers. If you don't know already, n-grams are contiguous sequences of **n** items from a sequence of text or speech. These items could consist of words, phonemes, letters, characters, or syllables. Shingles are n-grams where the items only consist of words. We will use n-grams of size 1, 2, and 3, which are also known as unigram, bigram, and trigram, respectively. The `UnigramTagger`, `BigramTagger`, and `TrigramTagger` are classes that inherit from the base class `NGramTagger`, which itself inherits from the `ContextTagger` class, which inherits from the `SequentialBackoffTagger` class. We will use the `train_data` as training data to train the n-gram taggers based on sentence tokens and their POS tags. Then we will evaluate the trained taggers on `test_data` and see the result upon tagging our sample sentence.

```
## N gram taggers
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger

ut = UnigramTagger(train_data)
bt = BigramTagger(train_data)
tt = TrigramTagger(train_data)

# testing performance of unigram tagger
print(ut.evaluate(test_data))
print(ut.tag(nltk.word_tokenize(sentence)))
```

```
0.8619421047536063
[('US', 'NNP'), ('unveils', None), ('world', 'NN'), ("'s", 'POS'), ('most',
'JJS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','), ('beats',
None), ('China', 'NNP'), ('.', '.')]
```

```
# testing performance of bigram tagger
print(bt.evaluate(test_data))
print(bt.tag(nltk.word_tokenize(sentence)))
```

```
0.1359279697937845
[('US', None), ('unveils', None), ('world', None), ("'s", None), ('most',
None), ('powerful', None), ('supercomputer', None), (',', None), ('beats',
None), ('China', None), ('.', None)]
```

```
# testing performance of trigram tagger
print(tt.evaluate(test_data))
print(tt.tag(nltk.word_tokenize(sentence)))
```

```
0.08142124116565011
[('US', None), ('unveils', None), ('world', None), ("'s", None), ('most',
None), ('powerful', None), ('supercomputer', None), (',', None), ('beats',
None), ('China', None), ('.', None)]
```

This output clearly shows us that we obtain 86% accuracy on the test set using unigram tagger alone, which is really good compared to our last tagger. The None tag indicates the tagger was unable to tag that word and the reason for that would be that it was unable to get a similar token in the training data. Accuracies of the bigram and trigram models are far lower because the same bigrams and trigrams observed in the training data aren't always present in the same way in the testing data.

We now look at an approach to combine all the taggers by creating a combined tagger with a list of taggers and use a backoff tagger. Essentially, we would create a chain of taggers and each tagger would fall back on a backoff tagger if it cannot tag the input tokens.

```
def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff
```

```
ct = combined_tagger(train_data=train_data,
                     taggers=[UnigramTagger, BigramTagger, TrigramTagger],
                     backoff=rt)
```

```
# evaluating the new combined tagger with backoff taggers
print(ct.evaluate(test_data))
print(ct.tag(nltk.word_tokenize(sentence)))
```

```
0.9108335753703166
[('US', 'NNP'), ('unveils', 'NNS'), ('world', 'NN'), ("'s", 'POS'),
('most', 'JJS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','),
('beats', 'NNS'), ('China', 'NNP'), ('.', '.')]
```

We now obtain an accuracy of 91% on the test data, which is excellent. Also we see that this new tagger can successfully tag all the tokens in our sample sentence (even though a couple of them are not correct, like beats should be a verb).

For our final tagger, we will use a supervised classification algorithm to train our tagger. The `ClassifierBasedPOSTagger` class enables us train a tagger by using a supervised learning algorithm in the `classifier_builder` parameter. This class is inherited from the `ClassifierBasedTagger` and it has a `feature_detector()` function that forms the core of the training process. This function is used to generate various features from the training data like word, previous word, tag, previous tag, case, and so on. In fact, you can even build your own feature detector function and pass it to the `feature_detector` parameter when instantiating an object of the `ClassifierBasedPOSTagger` class.

The classifier we will be using is the `NaiveBayesClassifier`. It uses the Bayes' theorem to build a probabilistic classifier assuming the features are independent. You can read more about it at https://en.wikipedia.org/wiki/Naive_Bayes_classifier since going into details about the algorithm is out of our current scope. The following code snippet shows a classification based approach to building and evaluating a POS tagger.

```
from nltk.classify import NaiveBayesClassifier, MaxentClassifier
from nltk.tag.sequential import ClassifierBasedPOSTagger

nbt = ClassifierBasedPOSTagger(train=train_data,
                               classifier_builder=NaiveBayesClassifier.train)

# evaluate tagger on test data and sample sentence
print(nbt.evaluate(test_data))
print(nbt.tag(nltk.word_tokenize(sentence)))
```

```
0.9306806079969019
[('US', 'PRP'), ('unveils', 'VBZ'), ('world', 'VBN'), ("'s", 'POS'),
('most', 'JJS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','),
('beats', 'VBZ'), ('China', 'NNP'), ('.', '.')]
```

Using this tagger, we get an accuracy of 93% on our test data, which is the highest out of all our taggers. Also if you observe the output tags for the sample sentence, you will see they are correct and make perfect sense. This gives us an idea of how powerful and effective classifier based POS taggers can be! Feel free to use a different classifier like `MaxentClassifier` and compare the performance with this tagger. We have included the code in the notebook to make things easier.

There are also several other ways to build and use POS taggers using NLTK and other packages. Even though it is not necessary and this should be enough to cover your POS tagging needs, you can go ahead and explore other methods to compare with these methods and satisfy your curiosity.

## Shallow Parsing or Chunking

*Shallow parsing,* also known as *light parsing* or *chunking*, is a technique of analyzing the structure of a sentence to break it down into its smallest constituents, which are tokens like words, and group them together into higher-level phrases. In shallow parsing, there is more focus on identifying these phrases or chunks rather than diving into further details of the internal syntax and relations inside each chunk, like we see in grammar based parse trees obtained from deep parsing. The main objective of shallow parsing is to obtain semantically meaningful phrases and observe relations among them.

You can look at the "Language Syntax and Structure" section from Chapter 1 just to refresh your memory regarding how words and phrases give structure to a sentence consisting of a bunch of words. Based on the hierarchy we depicted earlier, groups of words make up phrases. There are five major categories of phrases:

- **Noun phrase (NP):** These are phrases where a noun acts as the head word. Noun phrases act as a subject or object to a verb.
- **Verb phrase (VP):** These phrases are lexical units that have a verb acting as the head word. Usually, there are two forms of verb phrases. One form has the verb components as well as other entities such as nouns, adjectives, or adverbs as parts of the object.
- **Adjective phrase (ADJP):** These are phrases with an adjective as the head word. Their main role is to describe or qualify nouns and pronouns in a sentence, and they will be placed before or after the noun or pronoun.

- **Adverb phrase (ADVP):** These phrases act like adverbs since the adverb acts as the head word in the phrase. Adverb phrases are used as modifiers for nouns, verbs, or adverbs by providing further details that describe or qualify them.
- **Prepositional phrase (PP):** These phrases usually contain a preposition as the head word and other lexical components like nouns, pronouns, and so on. These act like an adjective or adverb, describing other words or phrases.

A shallow parsed tree is depicted in Figure 3-7 for a sample sentence just to refresh your memory on its structure.

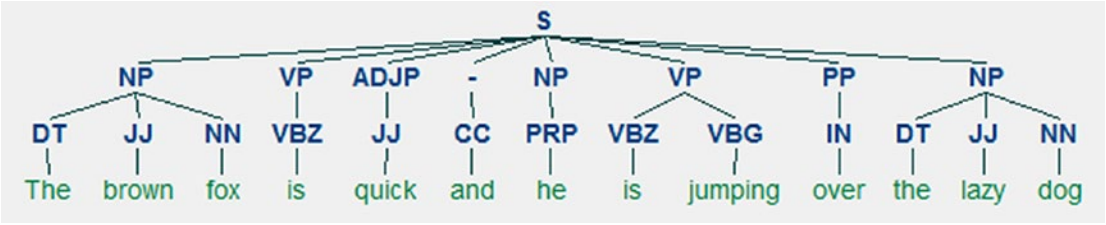


***Figure 3-7.*** *An example of shallow parsing depicting higher level phrase annotations*

We will now look at ways in which we can implement shallow parsing on text data using a wide variety of techniques, including regular expressions, chunking, chinking, and tag based training.

## Building Shallow Parsers

We use several techniques like regular expressions and tagging based learners to build our own shallow parsers. Just like POS tagging, we use some training data to train our parsers if needed and evaluate all our parsers on some test data and on our sample sentence. The `treebank` corpus is available in NLTK with chunk annotations. We load it and then prepare our training and testing datasets using the following code snippet.

```
from nltk.corpus import treebank_chunk
data = treebank_chunk.chunked_sents()

train_data = data[:3500]
test_data = data[3500:]
```

```
# view sample data
print(train_data[7])
```

```
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

From this output, you can see that our data points are sentences and are already annotated with phrase and POS tags metadata, which will be useful in training shallow parsers. We start by using regular expressions for shallow parsing using concepts of chunking and chinking. Using the process of *chunking*, we can use and specify specific patterns to identify what we would want to chunk or segment in a sentence, such as phrases based on specific metadata. *Chinking* is the reverse of chunking, where we specify which specific tokens we do not want to be a part of any chunk and then form the necessary chunks excluding these tokens. Let's consider a simple sentence (our news headline) and use regular expressions. We leverage the `RegexpParser` class to create shallow parsers to illustrate chunking and chinking for noun phrases.

```
from nltk.chunk import RegexpParser

# get POS tagged sentence
tagged_simple_sent = nltk.pos_tag(nltk.word_tokenize(sentence))
print('POS Tags:', tagged_simple_sent)

# illustrate NP chunking based on explicit chunk patterns
chunk_grammar = """
NP: {<DT>?<JJ>*<NN.*>}
"""
rc = RegexpParser(chunk_grammar)
c = rc.parse(tagged_simple_sent)

# print and view chunked sentence using chunking
```

```
print(c)
c
```

```
(S
  (NP US/NNP)
  (NP unveils/JJ world/NN)
  's/POS
  most/RBS
  (NP powerful/JJ supercomputer/NN)
  ,/,
  beats/VBZ
  (NP China/NNP)
  ./.)
```

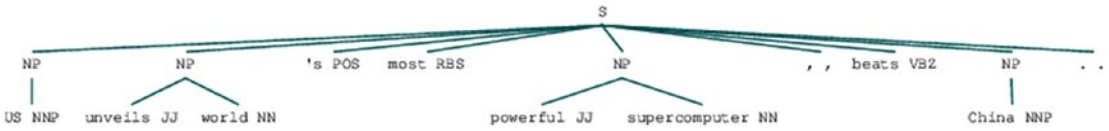


***Figure 3-8.*** *Shallow parsing using chunking*

We can see how the shallow parse tree looks in Figure 3-8 with only NP chunks using chunking. Let's look at building this using chinking now.

```
# illustrate NP chunking based on explicit chink patterns
chink_grammar = """
NP:
    {<.*>+}             # Chunk everything as NP
    }<VBZ|VBD|JJ|IN>+{  # Chink sequences of VBD\VBZ\JJ\IN
"""
rc = RegexpParser(chink_grammar)
c = rc.parse(tagged_simple_sent)

# print and view chunked sentence using chinking
print(c)
c
```

```
(S
  (NP US/NNP)
```

```
  unveils/JJ
  (NP world/NN 's/POS most/RBS)
  powerful/JJ
  (NP supercomputer/NN ,/,)
  beats/VBZ
  (NP China/NNP ./.))
```


***Figure 3-9.*** *Shallow parsing using chinking*

We can see how the shallow parse tree looks in Figure 3-9, with verbs and adjectives forming chinks and separating out noun phrases. Remember that *chunks* are sequences of tokens that are included in a collective group (chunk) and *chinks* are tokens or sequences of tokens that are excluded from chunks. We will now train a more generic regular expression-based shallow parser and test its performance on our test `treebank` data. Internally there are several steps that are executed to perform this parsing. The `Tree` structures used to represent parsed sentences in NLTK are converted to `ChunkString` objects. We create an object of `RegexpParser` using defined chunking and chinking rules. Objects of the `ChunkRule` and `ChinkRule` classes help create the complete shallow parsed tree with the necessary chunks based on specified patterns. The following code snippet represents a shallow parser using regular expression based patterns.

```
# create a more generic shallow parser
grammar = """
NP: {<DT>?<JJ>?<NN.*>}
ADJP: {<JJ>}
ADVP: {<RB.*>}
PP: {<IN>}
VP: {<MD>?<VB.*>+}
"""
rc = RegexpParser(grammar)
c = rc.parse(tagged_simple_sent)
```

```
# print and view shallow parsed sample sentence
print(c)
c
```

```
(S
  (NP US/NNP)
  (NP unveils/JJ world/NN)
  's/POS
  (ADVP most/RBS)
  (NP powerful/JJ supercomputer/NN)
  ,/,
  (VP beats/VBZ)
  (NP China/NNP)
  ./.)
```

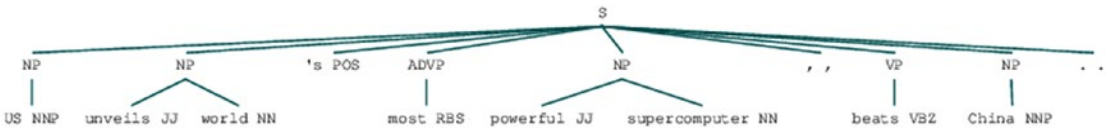


***Figure 3-10.*** *Shallow parsing using more specific rules*

We can see how the shallow parse tree looks in Figure 3-10, with more specific rules for specific phrases. Let's take a look at how this parser performs on the test dataset we built earlier.

```
# Evaluate parser performance on test data
print(rc.evaluate(test_data))
```

```
ChunkParse score:
    IOB Accuracy:  46.1%%
    Precision:     19.9%%
    Recall:        43.3%%
    F-Measure:     27.3%%
```

From the output, we can see that the parse tree for our sample sentence is very similar to the one we obtained from the out-of-the-box parser in the previous section. Also the accuracy of the overall test data is 54.5%, which is quite decent for a start. To

get more details as to what each performance metric signifies, you can refer to the "Evaluating Classification Models" section in Chapter 5.

Remember when we said annotated tagged metadata for text is useful in many ways? We use the chunked and tagged `treebank` training data now to build a shallow parser. We leverage two chunking utility functions—`tree2conlltags` to get triples of word, tag, and chunk tags for each token and `conlltags2tree` to generate a parse tree from these token triples.

We use these functions to train our parser later. First, let's see how these two functions work. Remember the chunk tags use a popular format, known as the IOB format. In this format, you will notice some new notations with `I`, `O`, and `B` prefixes, which is the popular IOB notation used in chunking. It depicts Inside, Outside, and Beginning. The `B-` prefix before a tag indicates it is the beginning of a chunk; the `I-` prefix indicates that it is inside a chunk. The `O` tag indicates that the token does not belong to any chunk. The `B-` tag is always used when there are subsequent tags following it of the same type without the presence of `O` tags between them.

```
from nltk.chunk.util import tree2conlltags, conlltags2tree
# look at a sample training tagged sentence
train_sent = train_data[7]
print(train_sent)
```

```
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

```
# get the (word, POS tag, Chunk tag) triples for each token
wtc = tree2conlltags(train_sent)
wtc
```

```
[('A', 'DT', 'B-NP'),
 ('Lorillard', 'NNP', 'I-NP'),
```

```
 ('spokewoman', 'NN', 'I-NP'),
 ('said', 'VBD', 'O'),
 (',', ',', 'O'),
 ('``', '``', 'O'),
 ('This', 'DT', 'B-NP'),
 ('is', 'VBZ', 'O'),
 ('an', 'DT', 'B-NP'),
 ('old', 'JJ', 'I-NP'),
 ('story', 'NN', 'I-NP'),
 ('.', '.', 'O')]

# get shallow parsed tree back from the WTC triples
tree = conlltags2tree(wtc)
print(tree)

(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

Now that we know how these functions work, we define a function called `conll_tag_chunks()` to extract POS and Chunk tags from sentences with chunked annotations and reuse our `combined_taggers()` function from POS tagging to train multiple taggers with backoff taggers, as depicted in the following code snippet.

```
def conll_tag_chunks(chunk_sents):
  tagged_sents = [tree2conlltags(tree) for tree in chunk_sents]
  return [[(t, c) for (w, t, c) in sent] for sent in tagged_sents]

def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff
```

We now define a NGramTagChunker class, which will take in tagged sentences as training input, get their WTC triples (word, POS tag, chunk tag), and train a BigramTagger with a UnigramTagger as the backoff tagger. We also define a parse() function to perform shallow parsing on new sentences.

```
from nltk.tag import UnigramTagger, BigramTagger
from nltk.chunk import ChunkParserI

class NGramTagChunker(ChunkParserI):

  def __init__(self, train_sentences,
               tagger_classes=[UnigramTagger, BigramTagger]):
    train_sent_tags = conll_tag_chunks(train_sentences)
    self.chunk_tagger = combined_tagger(train_sent_tags, tagger_classes)

  def parse(self, tagged_sentence):
    if not tagged_sentence:
        return None
    pos_tags = [tag for word, tag in tagged_sentence]
    chunk_pos_tags = self.chunk_tagger.tag(pos_tags)
    chunk_tags = [chunk_tag for (pos_tag, chunk_tag) in chunk_pos_tags]
    wpc_tags = [(word, pos_tag, chunk_tag) for ((word, pos_tag), chunk_tag)
                    in zip(tagged_sentence, chunk_tags)]
    return conlltags2tree(wpc_tags)
```

In this class, the constructor __init__() function is used to train the shallow parser using n-gram tagging based on the WTC triples for each sentence. Internally, it takes a list of training sentences as input, which is annotated with chunked parse tree metadata. It uses the conll_tag_chunks() function, which we defined earlier, to get a list of WTC triples for each chunked parse tree. Finally, it trains a BigramTagger with a Unigram tagger as a backoff tagger using these triples and stores the training model in self.chunk_tagger.

Remember that you can parse other n-gram based taggers for training by using the tagger_classes parameter. Once trained, the parse() function can be used to evaluate the tagger on test data and shallow parse new sentences. Internally, it takes a POS tagged sentence as input, separates the POS tags from the sentence, and uses our trained self.chunk_tagger to get the IOB chunk tags for the sentence. This is then combined with

the original sentence tokens and we use the `conlltags2tree()` function to get our final shallow parsed tree. The following snippet shows our parser in action. See Figure 3-11.

```
# train the shallow parser
ntc = NGramTagChunker(train_data)

# test parser performance on test data
print(ntc.evaluate(test_data))
```

```
ChunkParse score:
    IOB Accuracy:  97.2%%
    Precision:     91.4%%
    Recall:        94.3%%
    F-Measure:     92.8%%
```

```
# parse our sample sentence
sentence_nlp = nlp(sentence)
tagged_sentence = [(word.text, word.tag_) for word in sentence_nlp]
tree = ntc.parse(tagged_sentence)
print(tree)
tree
```

```
(S
  (NP US/NNP)
  unveils/VBZ
  (NP world/NN 's/POS most/RBS powerful/JJ supercomputer/NN)
  ,/,
  beats/VBZ
  (NP China/NNP)
  ./.)
```

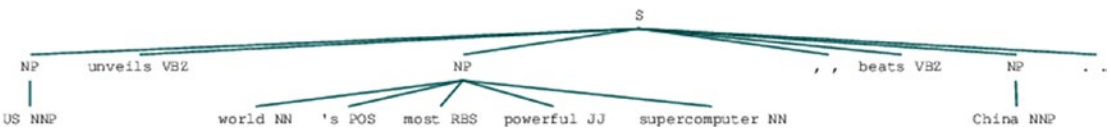


***Figure 3-11.*** *Shallow parsed news headline using n-gram based chunking on treebank data*

This output depicts our parser performance on the treebank test set data, which has an overall accuracy of 99.6%, which is excellent! Figure 3-11 also shows us how the parse tree looks for our sample news headline.

Let's now train and evaluate our parser on the `conll2000` corpus, which contains excerpts from *The Wall Street Journal* and is a much larger corpus. We will train our parser on the first 10,000 sentences and test its performance on the remaining 940+ sentences. The following snippet depicts this process.

```
from nltk.corpus import conll2000
wsj_data = conll2000.chunked_sents()
train_wsj_data = wsj_data[:10000]
test_wsj_data = wsj_data[10000:]

# look at a sample sentence in the corpus
print(train_wsj_data[10])

(S
  (NP He/PRP)
  (VP reckons/VBZ)
  (NP the/DT current/JJ account/NN deficit/NN)
  (VP will/MD narrow/VB)
  (PP to/TO)
  (NP only/RB #/# 1.8/CD billion/CD)
  (PP in/IN)
  (NP September/NNP)
  ./.)

# train the shallow parser
tc = NGramTagChunker(train_wsj_data)

# test performance on the test data
print(tc.evaluate(test_wsj_data))

ChunkParse score:
    IOB Accuracy:  89.1%%
    Precision:     80.3%%
    Recall:        86.1%%
    F-Measure:     83.1%%
```

This output shows that our parser achieved an overall accuracy of around 89%, which is quite good considering this corpus is much larger compared to the `treebank` corpus. Let's look at how it chunks our sample news headline.

```
# parse our sample sentence
tree = tc.parse(tagged_sentence)
print(tree)
tree
```

```
(S
  (NP US/NNP)
  (VP unveils/VBZ)
  (NP world/NN)
  (NP 's/POS most/RBS powerful/JJ supercomputer/NN)
  ,/,
  (VP beats/VBZ)
  (NP China/NNP)
  ./.)
```


***Figure 3-12.*** *Shallow parsed news headline using n-gram based chunking on conll2000 data*

Figure 3-12 shows us how the parse tree looks for our sample news headline with more defined verb phrases as compared to previous parse trees. You can also look at implementing shallow parsers using other techniques, like supervised classifiers, by leveraging the `ClassifierBasedTagger` class.

## Dependency Parsing

In dependency-based parsing, we try to use dependency-based grammars to analyze and infer both structure and semantic dependencies and relationships between tokens in a sentence. Refer to the "Dependency Grammars" subsection under "Grammar" in the "Language Syntax and Structure" section of Chapter 1 to refresh your memory. Dependency grammars help us annotate sentences with dependency tags, which

are one-to-one mappings between tokens signifying dependencies between them. A dependency grammar-based parse tree representation is a labeled and directed tree or graph to be more precise. The nodes are always the lexical tokens and the labeled edges depict dependency relationships between the heads and their dependents. The labels on the edges indicate the grammatical role of the dependent.

The basic principle behind a dependency grammar is that in any sentence in the language, all words except one have some relationship or dependency on other words in the sentence. The word that has no dependency is called the *root* of the sentence. The verb is taken as the root of the sentence in most cases. All the other words are directly or indirectly linked to the root verb using links , which are the dependencies. If we wanted to draw the dependency syntax tree for our sentence, "The brown fox is quick and he is jumping over the lazy dog," we would have the structure depicted in Figure 3-13.

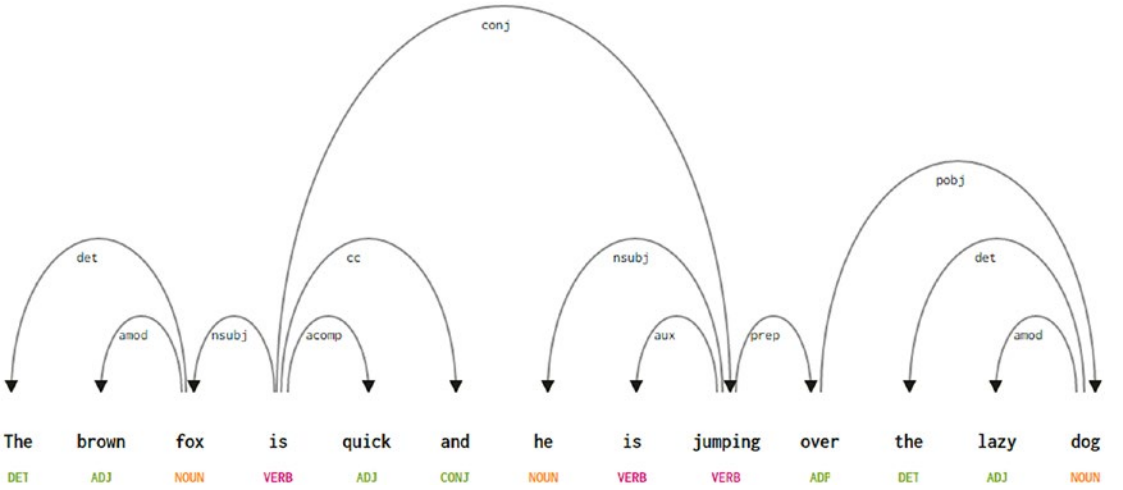


***Figure 3-13.*** *A dependency parse tree for a sample sentence*

These dependency relationships each have their own meanings and are part of a list of universal dependency types. This is discussed in an original paper, entitled "Universal Stanford Dependencies: A Cross-Linguistic Typology," by de Marneffe et al., 2014. You can check out the exhaustive list of dependency types and their meanings at `http://universaldependencies.org/u/dep/index.html`. Just to refresh your memory, if we observe some of these dependencies, it is not too hard to understand them.

- The dependency tag `det` is pretty intuitive—it denotes the determiner relationship between a nominal head and the determiner. Usually, the word with `POS` tag `DET` will also have the `det` dependency tag relation. Examples include fox → the and dog → the.
- The dependency tag `amod` stands for adjectival modifier and stands for any adjective that modifies the meaning of a noun. Examples include fox → brown and dog → lazy.
- The dependency tag `nsubj` stands for an entity that acts as a subject or agent in a clause. Examples include is → fox and jumping → he.
- The dependencies `cc` and `conj` have more to do with linkages related to words connected by coordinating conjunctions. Examples include is → and and is → jumping.
- The dependency tag `aux` indicates the auxiliary or secondary verb in the clause. Example: jumping → is.
- The dependency tag `acomp` stands for adjective complement and acts as the complement or object to a verb in the sentence. Example: is → quick.
- The dependency tag `prep` denotes a prepositional modifier, which usually modifies the meaning of a noun, verb, adjective, or preposition. Usually, this representation is used for prepositions having a noun or noun phrase complement. Example: jumping → over.
- The dependency tag `pobj` is used to denote the object of a preposition. This is usually the head of a noun phrase following a preposition in the sentence. Example: over → dog.

Let's look at some ways in which we can build dependency parsers for parsing unstructured text!

## Building Dependency Parsers

We use a couple of state-of-the-art libraries, including NLTK and spaCy, to generate dependency-based parse trees and test them on our sample news headline. *SpaCy* had two types of English dependency parsers based on what language models you use. You can find more details at `https://spacy.io/api/annotation#section-dependency-parsing`.

Based on language models, you can use the Universal Dependencies Scheme or the CLEAR Style Dependency Scheme, also available in NLP4Jnow. We now leverage spaCy and print the dependencies for each token in our news headline.

```
dependency_pattern = '{left}<---{word}[{w_type}]--->{right}\n--------'
for token in sentence_nlp:
    print(dependency_pattern.format(word=token.orth_,
                                    w_type=token.dep_,
                                    left=[t.orth_
                                              for t
                                              in token.lefts],
                                    right=[t.orth_
                                               for t
                                               in token.rights]))
```

```
[]<---US[nsubj]--->[]
--------
['US']<---unveils[ROOT]--->['supercomputer', ',', 'beats', '.']
--------
[]<---world[poss]--->["'s"]
--------
[]<---'s[case]--->[]
--------
[]<---most[advmod]--->[]
--------
['most']<---powerful[amod]--->[]
--------
['world', 'powerful']<---supercomputer[dobj]--->[]
--------
[]<---,[punct]--->[]
--------
[]<---beats[conj]--->['China']
--------
[]<---China[dobj]--->[]
--------
[]<---.[punct]--->[]
--------
```

This output gives us each token and its dependency type. The left arrow points to the dependencies on its left and the right arrow points to the dependencies on its right. It is evident that the verb "beats" is the root since it doesn't have any other dependencies as compared to the other tokens. To learn more about each annotation, you can always refer to the CLEAR dependency scheme at https://emorynlp.github.io/nlp4j/components/dependency-parsing.html. We can also visualize these dependencies in a better way using the following code. See Figure 3-14.

```
from spacy import displacy

displacy.render(sentence_nlp, jupyter=True,
                options={'distance': 110,
                         'arrow_stroke': 2,
                         'arrow_width': 8})
```

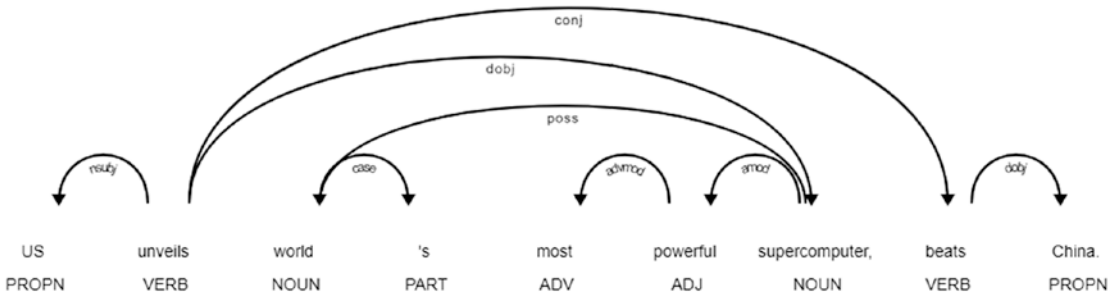


***Figure 3-14.*** *Visualizing our news headline dependency tree using spaCy*

You can also leverage NLTK and the Stanford Dependency Parser to visualize and build the dependency tree. We showcase the dependency tree in its raw and annotated forms. We start by building the annotated dependency tree and showing it using Graphviz. See Figure 3-15.

```
from nltk.parse.stanford import StanfordDependencyParser
sdp = StanfordDependencyParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar', path_to_models_jar='E:/stanford/
stanford-parser-full-2015-04-20/stanford-parser-3.5.2-models.jar')
```

```
# perform dependency parsing
result = list(sdp.raw_parse(sentence))[0]

# generate annotated dependency parse tree
result
```

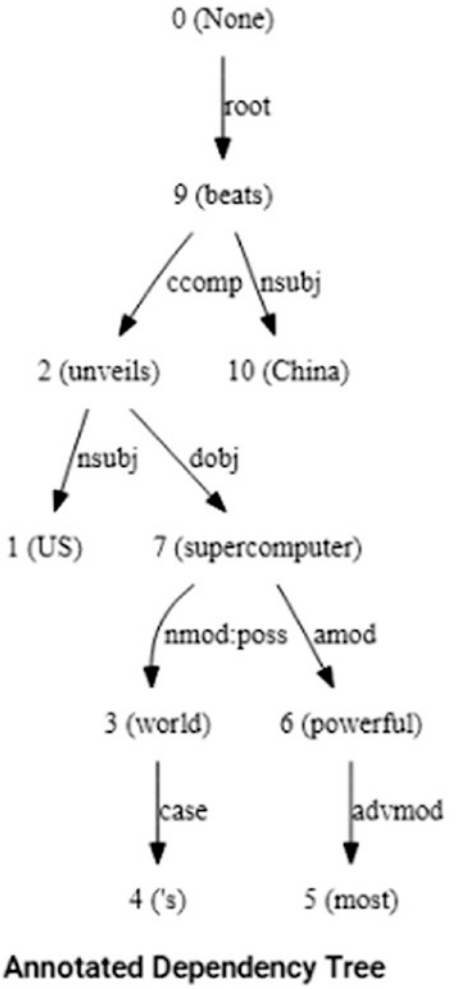


***Figure 3-15.*** *Visualizing our news headline annotated dependency tree using NLTK and the Stanford Dependency Parser*

We can also look at the actual dependency components in the form of triplets using the following code snippet.

```
# generate dependency triples
[item for item in result.triples()]
```

```
[(('beats', 'VBZ'), 'ccomp', ('unveils', 'VBZ')),
 (('unveils', 'VBZ'), 'nsubj', ('US', 'NNP')),
 (('unveils', 'VBZ'), 'dobj', ('supercomputer', 'NN')),
 (('supercomputer', 'NN'), 'nmod:poss', ('world', 'NN')),
 (('world', 'NN'), 'case', ("'s", 'POS')),
 (('supercomputer', 'NN'), 'amod', ('powerful', 'JJ')),
 (('powerful', 'JJ'), 'advmod', ('most', 'RBS')),
 (('beats', 'VBZ'), 'nsubj', ('China', 'NNP'))]
```

This gives us a detailed view into each token and the dependency relationships between tokens. Let's build and visualize the raw dependency tree now. See Figure 3-16.

```
# print simple dependency parse tree
dep_tree = result.tree()
print(dep_tree)
```

```
(beats (unveils US (supercomputer (world 's) (powerful most))) China)
```

```
# visualize simple dependency parse tree
dep_tree
```

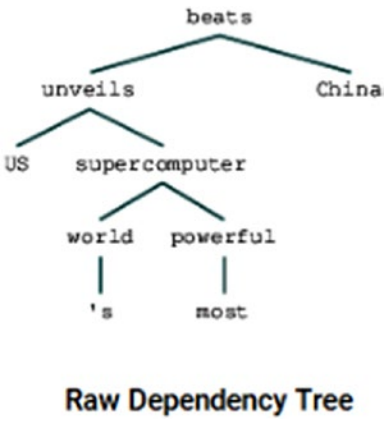


***Figure 3-16.*** *Visualizing our news headline raw dependency tree using NLTK and Stanford Dependency Parser*

Notice the similarities with the tree we obtained earlier in Figure 3-15. The annotations help with understanding the type of dependency among the different tokens. You can also see how easily we can generate dependency parse trees for sentences and analyze and understand relationships and dependencies among the tokens. The Stanford Parser is quite stable and robust. It integrates well with NLTK. We recommend using the NLTK or spaCy parsers, as both of them are quite good.

## Constituency Parsing

Constituent based grammars are used to analyze and determine the constituents that a sentence is composed of. Besides determining the constituents, another important objective is to determine the internal structure of these constituents and how they link to each other. There are usually several rules for different types of phrases based on the type of components they can contain and we can use them to build parse trees. Refer to the "Constituency Grammars" subsection under "Grammar" in the "Language Syntax and Structure" section of Chapter 1 to refresh your memory and look at some examples of sample parse trees.

In general, constituency based grammar helps specify how we can break a sentence into various constituents. Once that is done, it helps in breaking down those constituents into further subdivisions; this process repeats until we reach the level of individual tokens or words. Typically, these grammar types can be used to model or represent the internal structure of sentences in terms of a hierarchically ordered structure of their constituents. Each word usually belongs to a specific lexical category in the case and forms the head word of different phrases. These phrases are formed based on rules called phrase structure rules.

*Phrase structure rules* form the core of constituency grammars, because they talk about syntax and rules that govern the hierarchy and ordering of the various constituents in the sentences. These rules cater to two things primarily:

- They determine what words are used to construct the phrases or constituents.
- They determine how we need to order these constituents.

The generic representation of a phrase structure rule is **S → AB** , which depicts that the structure **S** consists of constituents **A** and **B** , and the ordering is **A** followed by **B** . While there are several rules (refer to Chapter 1 if you want to dive deeper), the most

important rule describes how to divide a sentence or a clause. The phrase structure rule denotes a binary division for a sentence or a clause as **S → NP VP**, where **S** is the sentence or clause, and it is divided into the subject, denoted by the noun phrase (**NP**) and the predicate, denoted by the verb phrase (**VP**).

These grammars have various production rules and usually a context free grammar (CFG) or Phrase Structured Grammar is sufficient for this. A constituency parser can be built based on such grammars/rules, which are usually collectively available as context-free grammar (CFG) or phrase-structured grammar. The parser will process input sentences according to these rules and help in building a parse tree. A sample tree is depicted in Figure 3-17.

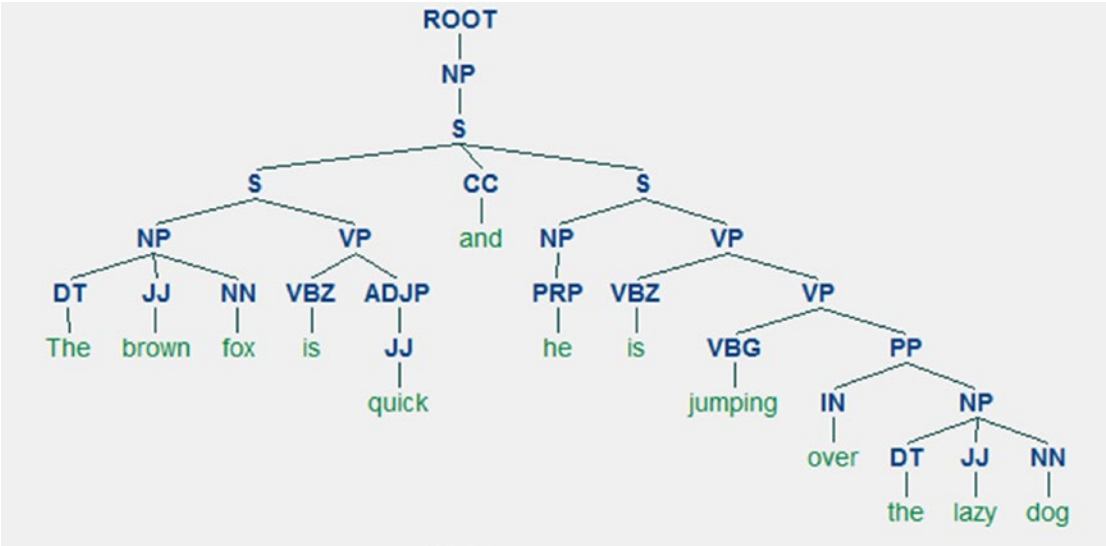


***Figure 3-17.*** *An example of constituency parsing showing a nested hierarchical structure*

The parser brings the grammar to life and can be said to be a procedural interpretation of the grammar. There are various types of parsing algorithms some of which are mentioned as follows:

- Recursive Descent parsing
- Shift Reduce parsing
- Chart parsing
- Bottom-up parsing

- Top-down parsing
- PCFG parsing

Going through these in detail would be impossible in the current scope. However, NLTK provides some excellent information on them at http://www.nltk.org/book/ch08.html in their official book. We describe some of these parsers briefly and look at PCFG parsing in detail when we implement our own parser later.

- *Recursive Descent parsing* usually follows a top-down parsing approach; it reads in tokens from the input sentence and tries to match them with the terminals from the grammar production rules. It keeps looking ahead by one token and advances the input read pointer each time it gets a match.
- *Shift Reduce parsing* follows a bottom-up parsing approach where it finds sequences of tokens (words/phrases) that correspond to the right side of grammar productions and then replaces it with the left side for that rule. This process continues until the whole sentence is reduced to give us a parse tree.
- *Chart parsing* uses dynamic programming to store intermediate results and reuses them when needed to get significant efficiency gains. In this case, chart parsers store partial solutions and look them up when needed to get to the complete solution.

## Building Constituency Parsers

We will be using NLTK and the Stanford Parser to generate parse trees since they are state-of-the-art and work very well.

---

**Prerequisites** Download the official Stanford Parser from http://nlp.stanford.edu/software/stanford-parser-full-2015-04-20.zip, which seems to work quite well. You can try a later version by going to http://nlp.stanford.edu/software/lex-parser.shtml#Download and checking the Release History section. After downloading, unzip it to a known location in your filesystem. Once you're done, you are now ready to use the parser from NLTK, which we will be exploring shortly.

---

The Stanford Parser generally uses a *PCFG (probabilistic context-free grammar) parser*. A PCFG is a context-free grammar that associates a probability with each of its production rules. The probability of a parse tree generated from a PCFG is simply the production of the individual probabilities of the productions used to generate it. Let's put this parser to action now! See Figure 3-18.

```
# set java path
import os
java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
os.environ['JAVAHOME'] = java_path

# create parser object
from nltk.parse.stanford import StanfordParser
scp = StanfordParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar',
                     path_to_models_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser-3.5.2-models.jar')

# get parse tree
result = list(scp.raw_parse(sentence))[0]
# print the constituency parse tree
print(result)

(ROOT
  (SINV
    (S
      (NP (NNP US))
      (VP
        (VBZ unveils)
        (NP
          (NP (NN world) (POS 's))
          (ADJP (RBS most) (JJ powerful))
          (NN supercomputer))))
    (, ,)
    (VP (VBZ beats))
    (NP (NNP China))
    (. .)))
```

```
# visualize the parse tree
from IPython.display import display
display(result)
```

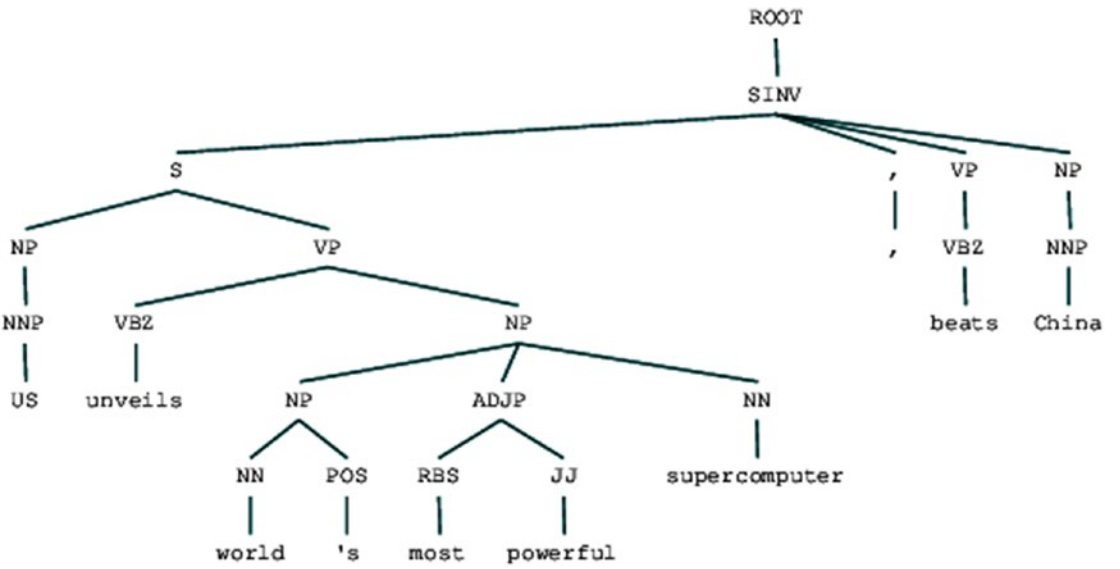


***Figure 3-18.*** *Constituency parsing on our sample news headline*

We can see the nested hierarchical structure of the constituents in the preceding output as compared to the flat structure in shallow parsing. In case you are wondering what SINV means, it represents *an inverted declarative sentence*, i.e. one in which the subject follows the tensed or modal verb. Refer to the "Penn Treebank Reference" at `https://web.archive.org/web/20130517134339/http://bulba.sdsu.edu/jeanette/thesis/PennTags.html` as needed to look up other tags.

There are various ways that you can build your own constituency parsers, including creating your own CFG production rules and then using a parser to use that grammar. To build your own CFG, you can use the `nltk.CFG.fromstring` function to feed in your own production rules and then use parsers like `ChartParser` or `RecursiveDescentParser`, which both belong to the NLTK package. Feel free to build some toy grammars and play around with these parsers.

We now look at a way to build a constituency parser that scales well and is efficient. The problem with regular CFG parsers like chart and recursive descent parsers is that they can get easily overwhelmed by the sheer number of total possible parses and can

become extremely slow. This is where weighted grammars like PCFG (Probabilistic Context Free Grammar) and probabilistic parsers like the Viterbi parser prove to be more effective.

A PCFG is a context free grammar that associates a probability with each of its production rules. The probability of a parse tree generated from a PCFG is simply the production of the individual probabilities of the productions that were used to generate it. We will use NLTK's `ViterbiParser` to train a parser on the `treebank` corpus, which provides annotated parse trees for each sentence in the corpus. This parser is a bottom-up PCFG parser that uses dynamic programming to find the most likely parse at each step. We start our process of building our own parser by loading the necessary training data and dependencies.

```
import nltk
from nltk.grammar import Nonterminal
from nltk.corpus import treebank
# load and view training data
training_set = treebank.parsed_sents()
print(training_set[1])
```

```
(S
  (NP-SBJ (NNP Mr.) (NNP Vinken))
  (VP
    (VBZ is)
    (NP-PRD
      (NP (NN chairman))
      (PP
        (IN of)
        (NP
          (NP (NNP Elsevier) (NNP N.V.))
          (, ,)
          (NP (DT the) (NNP Dutch) (VBG publishing) (NN group))))))
  (. .))
```

Now we build the production rules for our grammar by extracting the productions from the tagged and annotated training sentences and adding them.

```
# extract the productions for all annotated training sentences
treebank_productions = list(
                        set(production
                            for sent in training_set
                            for production in sent.productions()
                        )
                    )
# view some production rules
treebank_productions[0:10]
```

```
[VP -> VB NP-2 PP-CLR ADVP-MNR,
 NNS -> 'foods',
 NNP -> 'Joanne',
 JJ -> 'state-owned',
 VP -> VBN PP-LOC,
 NN -> 'turmoil',
 SBAR -> WHNP-240 S,
 QP -> DT VBN CD TO CD,
 NN -> 'cultivation',
 NNP -> 'Graham']
```

```
# add productions for each word, POS tag
for word, tag in treebank.tagged_words():
    t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
    for production in t.productions():
        treebank_productions.append(production)

# build the PCFG based grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'), treebank_
productions)
```

Now that we have our necessary grammar with production rules, we create our parser using the following snippet by training it on the grammar and then try to evaluate it on our sample news headline.

```
# build the parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)

# get sample sentence tokens
tokens = nltk.word_tokenize(sentence)

# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-87-2b0fd95b2fbd> in <module>()
     16
     17 # get parse tree for sample sentence
---> 18 result = list(viterbi_parser.parse(tokens))

ValueError: Grammar does not cover some of the input words: "'unveils',
'beats'".
```

Unfortunately, we get an error when we try to parse our sample sentence tokens with our newly built parser. The reason is quite clear from the error that some of the words in our sample sentence are not covered by the Treebank-based grammar because they are not present in our treebank corpus. Since this constituency based grammar uses POS tags and phrase tags to build the tree based on the training data, we will add the token and POS tags for our sample sentence in our grammar and rebuild the parser.

```
# get tokens and their POS tags and check it
tagged_sent = nltk.pos_tag(nltk.word_tokenize(sentence))
print(tagged_sent)
```

```
[('US', 'NNP'), ('unveils', 'JJ'), ('world', 'NN'), ("'s", 'POS'), ('most',
'RBS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','), ('beats',
'VBZ'), ('China', 'NNP'), ('.', '.')]
```

```
# extend productions for sample sentence tokens
for word, tag in tagged_sent:
    t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
    for production in t.productions():
        treebank_productions.append(production)
```

```
# rebuild grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'),
                                            treebank_productions)
# rebuild parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)
# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))[0]

# print parse tree
print(result)

(S
  (NP-SBJ-2
    (NP (NNP US))
    (NP
      (NP (JJ unveils) (NN world) (POS 's))
      (JJS most)
      (JJ powerful)
      (NN supercomputer)))
  (, ,)
  (VP (VBZ beats) (NP-TTL (NNP China)))
  (. .)) (p=5.08954e-43)

# visualize parse tree
result
```

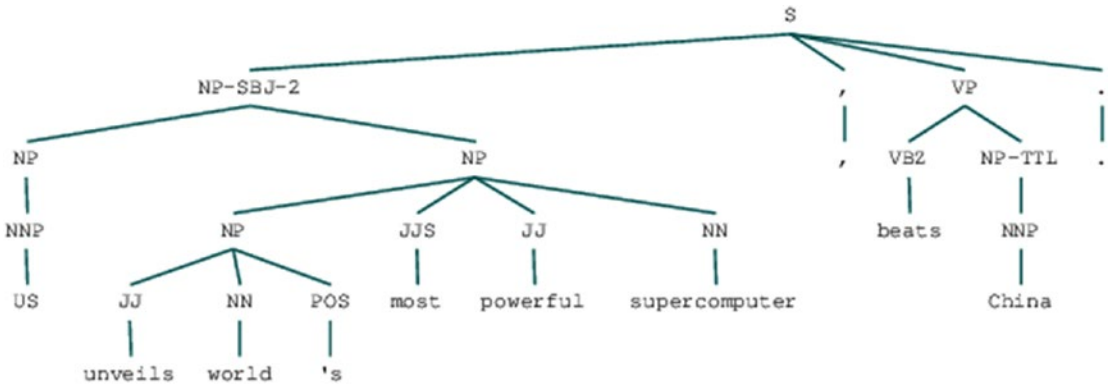


***Figure 3-19.*** *Constituency parse tree for our sample news headline based on Treebank annotations*

We are now able to successfully generate the parse tree for our sample news headline. You can see the visual representation of the tree in Figure 3-19. Remember that this is a probabilistic PCFG parser and you can see the overall probability of this tree mentioned in the output when we printed our parse tree. The notations of the tags followed here are all based on the Treebank annotations discussed earlier. Thus, this shows us how to build our own constituency-based parser.

# Summary

We have covered a lot concepts, techniques, and implementations with regard to text processing and wrangling, syntactic analysis, and understanding text data. A lot of the concepts from Chapter 1 should seem more relevant and clear now that we have implemented them on real examples. The content covered in this chapter is two-fold.

We looked at concepts related to text processing and wrangling. You now know the importance of processing and normalizing text and, as we move on to future chapters, you will see why it becomes more and more important to have well processed and standardized textual data. We have covered a wide variety of techniques for wrangling including text cleaning and tokenization, removing special characters, case conversions, and expanding contractions. We also looked at techniques for correcting text, like spelling corrections. We also built our own spelling corrector and contraction expander in the same context. We found a way to leverage WordNet and correct words with repeated characters. Finally, we looked at various stemming and lemmatization techniques and learned about ways to remove irrelevant words, known as *stopwords*.

The next part of our chapter was dedicated to analyzing and understanding text syntax and structure. We revisited concepts from Chapter 1, including POS tagging, shallow parsing, dependency parsing, and constituency parsing. You now know how to use taggers and parsers on real-world textual data and how to implement your own taggers and parsers. We dive more into analyzing and deriving insights from text in future chapters using various machine learning techniques including classification, clustering, and summarization. Stay tuned!