3. Words: The Building Blocks of Language

Language can be divided up into pieces at a variety of different grain sizes, ranging from sounds up to utterances and even to documents. In this chapter, we will focus on a very important level for much work in computational linguistics: words. Just what are words, and how should we represent them in a machine? At first, these may seem like trivial questions, but it turns out that there are some important issues involved in defining and encoding words.

To help us explore these issues, consider the following simple task: *count the number of words that appear in a document*. We'll use the following article (**wsj_0063**) from the Wall Street Journal as our example text:

```
Sea Containers Ltd. said it might increase the price of its
$70-a-share buy-back plan if pressed by Temple Holdings Ltd., which
made an earlier tender offer for Sea Containers.  Sea Containers, a
Hamilton, Bermuda-based shipping concern, said Tuesday that it would
sell $1.1 billion of assets and use some of the proceeds to buy about
50% of its common shares for $7 apiece.  The move is designed to
ward off a hostile takeover attempt by two European shipping concerns,
Stena Holding AG and Tiphook PLC.  In May, the two companies,
through their jointly owned holding company, Temple, offered $5 a
share, or $777 million, for Sea Containers...
```

To count the number of words in this article, we can simply divide it up into a list of individual word strings, and check the length of that list. The most obvious way to divide the article up is to split its text string on any sequence of spaces, newlines, or other "whitespace" characters:

```
>>> words = text.split()
>>> print words
['Sea', 'Containers', 'Ltd.', 'said', 'it', 'might', 'increase', 'the', 'price', 'o
 'its', '$70-a-share', 'buy-back', 'plan', ..., 'up', '62.5', 'cents.']
>>> len(words)
430
```

However, there are a few problems with this solution. First, it is not clear that every sequence of non-whitespace characters should be considered a word. For example, our example article contains the following sentence:

```
A spokesman for Temple estimated that Sea Containers' plan -- if all
the asset sales materialize -- would result in shareholders
receiving only $36 to $45 a share in cash.
```

Our simple algorithm treats each occurrence of the sequence -- as a word; but most people would consider them to be punctuation marks, not words.

Second, there may be some character sequences that are not separated by whitespace, but which we still want to consider separate words. One example of this is contractions, such as *didn't*. If we're analyzing the meaning of a sentence, it might be more useful to treat this character string as two separate words, *did* and *not*. Similarly, it sometimes makes more sense to treat hyphenated strings as two words. For example, in the expression *pro-New York*, it seems quite unnatural to say that *pro-New* is a single word.

Finally, there may be times when we want to treat a character sequence that contains whitespace as a single word. For example, it is sometimes useful to treat proper names such as "John Smith" or "New York" as individual words, even though they contain spaces.

As an especially subtle example, consider the numeric expressions in the following sentence (drawn from the MedLine corpus):

```
The corresponding free cortisol fractions in these sera were 4.53
+/- 0.15% and 8.16 +/- 0.23%, respectively.
```

Should we say that the numeric expression "4.53 +/- 0.15%" is three words? Or should we say that it's a single compound word? Or should we say that it is actually *nine* words, since it's read "four point five three, plus or minus fifteen percent"? Or should we say that it's not a "real" word at all, since it wouldn't appear in any dictionary? The answer will most likely depend on what task we're trying to solve.

Note

If we turn to languages other than English, segmenting words can be even more of a challenge. For example, in Chinese orthography, characters correspond to monosyllabic morphemes. Many morphemes are words in their own right, but many words contain more than one morpheme; most of them consist of two morphemes. However, there is no visual representation of word boundaries in Chinese text. For example, consider the following three-character string: 爱 国 人 (in pinyin plus tones: ai4 'love' (verb), guo3 'country', ren2 'person'). This could either be segmented as [爱 国] 人 — 'country-loving person' or as 爱 [国 人] — 'love country-person'.

3.1 Tokens and Types

Let us return to our task: *count the number of words that appear in a document*. In the previous section, we interpreted "the number of words" to mean "the number of times any word was used." However, another interpretation is possible: we might want to know how many distinct "dictionary entries" were used, regardless of the number of times that each dictionary item was repeated. For example, the following sentence contains seven "word uses" but only uses four "dictionary entries":

```
John likes Mary and Mary likes John.
```

Two distinguish between these two interpretations, we will define two new terms, "token" and "type." A word *token* is an individual occurrence of a word in a concrete context. A word *type* is a "dictionary entry" for a word. A word type is somewhat abstract; it's what we're talking about when we say that we know the meaning of the word *deprecate*, or when we say that the words *barf* and *vomit* are synonyms. On the other hand, a word token is something which exists in time and space. For example, we could talk about my uttering a token of the word *grunge* in Edinburgh on July 14, 2003; equally, we

can say that the last word token in the WSJ text is a token of the word type *cents*, or that there are four tokens of the type *Sea* in the text.

The terms *token* and *type* can also be applied to other linguistic entities. For example, a *sentence token* is an individual occurrence of a sentence; but a *sentence type* is an abstract sentence, without context. If someone repeats a sentence twice, they have uttered two sentence tokens, but only one sentence type. When the kind of token or type is obvious from context, we will simply use the terms token and type.

We can use this new terminology to express the two interpretations for our task unambiguously:

1. Count the number of word tokens that appear in a document.

2. Count the number of word types that appear in a document.

In the previous section, we considered how we might accomplish the first of these tasks; now, let's turn our attention to the second task, that of counting word types. Again, we will start with a simple approach: split the article's text on any sequence of whitespace, and count the number of times each substring occurs. But this time, we'll use a *set* instead of a *list* to collect the words:

```
>>> word_types = list(set(text.split()))
>>> word_types.sort()
>>> word_types
['$1.1', '$130', '$36', '$45', ..., 'with', 'would', 'yesterday,']
```

['all', '45', 'surplus', 'being', 'move', 'month', '130', 'through', 'leeway', ... 'chief', 'reaction', 'In', 'the'] >>> print len(word_types) 255

A quick glance at the contents of *word_types* reveals some of the shortcomings of this approach. These shortcomings can be divided into two categories:

- *Tokenization*: Which substrings of the original text contain word tokens?

- *Type definition*: How do we decide whether two tokens have the same type?

We encountered many of the issues with tokenization when we looked at the task of counting tokens. For example, should -- be counted as a word token? In addition to the issues we've already seen, our algorithm's ignorance about punctuation can cause it to count the same word type multiple times. For example, the substrings *price.* and *price* will be treated as two different word types. When we were counting tokens, this didn't affect our overall answer, since both *price.* and *price* count as a single token. But since the two strings are not identical, our algorithm counts them as two separate types. These tokenization issues could be addressed by defining a more advanced algorithm for splitting the text into word tokens (i.e., a *tokenizer*).

The type definition issues are exemplified by the fact that *the* and *The* are listed as two separate word types. We would like to say that these two tokens have the same type, but just happen to be written differently. But since our simple algorithm uses strict equality to divide word tokens into types, it treats them as two distinct types. A more subtle question is whether the two tokens *asset* and *assets* should be considered to share a type or not. On the one hand, they would both be listed under the same entry in a dictionary; but on the other hand, there is a definite semantic difference between them. As with some of the questions about tokenization, our decision about whether to treat these two tokens as having the same type or not will depend on the problem we're trying to solve.

3.1.1 Example: Stylistics

So far, we've seen how to count the number of tokens or types in a document. But it's much more interesting to look at *which* tokens or types appear in a document. We can use a Python dictionary to count the number of occurrences of each word type in a document:

```
>>> counts = {}
>>> for word in text.split():
...     if word in counts:
...         counts[word] += 1
...     else:
...         counts[word] = 1
```

The first statement, `counts = {}`, initializes the dictionary, while the next four lines successively add entries to it and increment the count each time we encounter a new token of a given type. To view the contents of the dictionary, we can iterate over its keys and print each entry:

```
>>> for word in sorted(counts)[:10]:
...     print counts[word], word
1 $1.1
2 $130
1 $36
1 $45
1 $490
1 $5
1 $62.625,
1 $620
1 $63
2 $7
```

We can also print the number of times that a specific word we're interested in appeared:

```
>>> print counts['might']
3
```

Applying this same approach to document collections that are categorized by genre, we can learn something about the patterns of word usage in those genres. For example, the following table was constructed by counting the number of times various modal words appear in different genres in the Brown Corpus:

Use of Modals in Brown Corpus, by Genre						
Genre	**can**	**could**	**may**	**might**	**must**	**will**
skill and hobbies	273	59	130	22	83	259
humor	17	33	8	8	9	13
fiction: science	16	49	4	12	8	16
press: reportage	94	86	66	36	50	387
fiction: romance	79	195	11	51	46	43
religion	84	59	79	12	54	64

Observe that the most frequent modal in the reportage genre is *will*, suggesting a focus on the future, while the most frequent modal in the romance genre is *could*, suggesting a focus on possibilities.

We can also measure the lexical diversity of a genre, by calculating the ratio of word types and word tokens, as shown in the following table. (Genres with lower diversity have a higher number of tokens per type.)

Word Types and Tokens in Brown Corpus, by Genre			
Genre	**Token Count**	**Type Count**	**Ratio**
skill and hobbies	82345	11935	6.9
humor	21695	5017	4.3
fiction: science	14470	3233	4.5
press: reportage	100554	14394	7.0
fiction: romance	70022	8452	8.3
religion	39399	6373	6.2

We can carry out a variety of interesting explorations simply by counting words. In fact, the field of *Corpus Linguistics* focuses almost exclusively on creating and interpreting such tables of word counts. So far, our method for identifying word tokens has been a little primitive, and we have not been able to separate punctuation from the words. We will take up this issue in the next section.

3.1.2 Example: Lexical Dispersion

Word tokens vary in their distribution throughout a text. We can visualize word distributions, to get an overall sense of topics and topic shifts. For example, consider the pattern of mention of the main characters in Jane Austen's *Sense and Sensibility*: Elinor, Marianne, Edward and Willoughby. The following plot contains four rows, one for each name. Each row contains a series of lines, drawn to indicate the position of each token.

Figure 1: Lexical Dispersion

Observe that *Elinor* and *Marianne* appear rather uniformly throughout the text, while *Edward* and *Willoughby* tend to appear separately. Here is the program that generated the above plot. It uses Python's **`Tkinter`** graphics library, declaring a **`canvas`** and adding lines to it using **`create_line()`**.

```
>>> from Tkinter import Canvas
>>> from nltk_lite.corpora import gutenberg
>>> def dispersion_plot(text, words, rowheight, rowwidth):
...     canvas = Canvas(width=rowwidth, height=rowheight*len(words))
...     scale = float(rowwidth)/len(text)
...     for i in range(len(words)):
...         for (position, word) in text:
...             x = position * scale
...             if word == words[i]:
```

```
...                 y = i * rowheight
...                 canvas.create_line(x, y, x, y+rowheight-1)
...     canvas.pack()
...     canvas.mainloop()
>>> text = list(enumerate(gutenberg.raw('austen-sense')))
>>> words = ['Elinor', 'Marianne', 'Edward', 'Willoughby']
>>> dispersion_plot(text, words, 15, 800)
```

3.1.3 Exercises

1. Write a program to create a table of word frequencies by genre, like the one given above for modals. Choose your own words and try to find words whose presence (or absence) is typical of a genre. Discuss your findings.

2. Write a program to generate a table of token/type ratios, as we saw above. Include the full set of Brown Corpus genres. Which genre has the lowest diversity. Is this what you would have expected?

3. Pick a text, and explore the dispersion of particular words. What does this tell you about the words, or the text?

3.2 Tokenization and Normalization

Tokenization is the task of extracting a list of elementary tokens that constitute a piece of language data. As we've seen, tokenization based solely on whitespace is too simplistic for most applications. In this section we will take a more sophisticated approach, using regular expression to specify which character sequences should be treated as words. We will also consider important ways to normalize tokens.

3.2.1 Tokenization with Regular Expressions

The function **`tokenize.regexp()`** takes a text string and a regular expression, and returns the list of substrings that match the regular expression. To define a tokenizer that includes punctuation as separate tokens, we could do the following:

```
>>> from nltk_lite import tokenize
>>> text = '''Hello.  Isn't this fun?'''
>>> pattern = r'\w+|[^\w\s]+'
>>> list(tokenize.regexp(text, pattern))
['Hello', '.', 'Isn', "'", 't', 'this', 'fun', '?']
```

The regular expression in this example will match a sequence consisting of one or more word characters **`\w+`**. It will also match a sequence consisting of one or more punctuation characters (or non-word, non-space characters **`[^\w\s]+`**). This is a negated range expression; it matches one or more characters which are not word characters (i.e., not a match for **`\w`**) and not a whitespace character (i.e., not a match for **`\s`**). We use the disjunction operator **`|`** to combine these into a single complex expression **`\w+|[^\w\s]+`**.

There are a number of ways we might want to improve this regular expression. For example, it currently breaks *$22.50* into four tokens; but we might want it to treat this as a single token. Similarly,

we would want to treat *U.S.A.* as a single token. We can deal with these by adding further clauses to the tokenizer's regular expression. For readability we break it up and insert comments, and use the **re.VERBOSE** flag, so that Python knows to strip out the embedded whitespace and comments.

```
>>> import re
>>> text = 'That poster costs $22.40.'
>>> pattern = re.compile(r'''
...     \w+                # sequences of 'word' characters
...   | \$?\d+(\.\d+)?     # currency amounts, e.g. $12.50
...   | ([\A\.])+          # abbreviations, e.g. U.S.A.
...   | [^\w\s]+           # sequences of punctuation
... ''', re.VERBOSE)
>>> list(tokenize.regexp(text, pattern))
['That', 'poster', 'costs', '$22.40', '.']
```

It is sometimes more convenient to write a regular expression matching the material that appears *between* tokens, such as whitespace and punctuation. The **tokenize.regexp()** function permits an optional boolean parameter **gaps**; when set to **True** the pattern is matched against the gaps. For example, here is how **tokenize.whitespace()** is defined:

```
>>> list(tokenize.regexp(text, pattern=r'\s+', gaps=True))
['That', 'poster', 'costs', '$22.40.']
```

3.2.2 Normalization

Earlier we talked about counting word tokens, and completely ignored the 'context' in which these tokens appeared. For a sentence like *I saw the saw*, we would have treated both *saw* tokens as instances of the same type. However, one is a form of the verb *SEE*, and the other is the name of a cutting instrument (we use uppercase to indicate a **lexeme**, or **lemma**). These two forms of *see* are unrelated, appearing in very different contexts, and should never be counted together. We can distinguish such **homographs** with the help of context; often the previous word suffices. We will explore this idea of context briefly, before addressing the main topic of this section.

A **bigram** is simply a pair of words. For example, in the sentence *She sells sea shells by the sea shore*, the bigrams are *She sells*, *sells sea*, *sea shells*, *shells by*, *by the*, *the sea*, *sea shore*.

As a first approximation to discovering the distribution of a word, we can look all the bigrams it occurs in. Let's consider all bigrams from the Brown Corpus which have the word *often* as first element. Here is a small selection, ordered by their counts:

```
often ,             16
often a             10
often in            8
often than          7
often the           7
often been          6
often do            5
often called        4
often appear        3
often were          3
often appeared      2
often are           2
often did           2
```

```
often is            2
often appears       1
often call          1
```

Observe that *often* is frequently followed by a comma. This suggests that *often* is common at the end of phrases. We also see that *often* precedes verbs, presumably as an adverbial modifier. Thus, *often saw* is likely to involve the verb *SEE*. Note that this list includes many forms of the same word: *been*, *were*, *are* and *is* are all forms of *BE*. It would be useful if we could group these together, or **lemmatize** them, replacing the inflected forms by their lemma. Then we could study which *verbs* are typically modified by a particular adverb. Applied to the above list, lemmatization would yield the following results, which gives us a more compact picture of the distribution of *often*.

```
,           16
a           10
BE          13
in          8
than        7
the         7
DO          7
APPEAR      6
CALL        5
```

Lemmatization is a rather sophisticated process which requires a mixture of rules for regular inflections and table look-up for irregular morphological patterns. Within NLTK, a simpler approach is offered by the *Porter Stemmer*, which strips inflectional suffixes from words, collapsing the different forms of *APPEAR* and *CALL*. (Note that this stemmer does not attempt to identify *was* as a form of the lexeme *BE*.) Run the Porter Stemmer demonstration as follows:

```
>>> from nltk_lite.stem import porter
>>> porter.demo()
```

Lemmatization and stemming can be regarded as special cases of **normalization**. They identify a canonical representative for a group of related wordforms. By its nature, normalization collapses distinctions. An example is case normalization, where all variants are mapped into a single format. What counts as the normalized form will vary according to context. Often, we convert everything into lower case, so that words which were capitalized by virtue of being sentence-initial are treated the same as those which occur elsewhere in the sentence. Case normalization will also collapse the *New* of *New York* with the *new* of *my new car*.

Term variation is a particularly challenging factor in biomedical texts. For example, the following are just some of the possible variants for *nuclear factor kappa B*, the name for a family of proteins:

```
nuclear factor kappa B, nuclear factor Kappa-B, nuclear factor kappaB,
nuclear factor kB, NF-KB, NF-kb, NF-kB, NFKB, NFkB, NF kappa-B
```

Although work is ongoing to standardize biomedical nomenclature, the rate at which new entities are discovered and described outstrips these efforts at present.

3.2.3 Exercises

1. **Regular expression tokenizers:** Save the Wall Street Journal example text from earlier in this chapter into a file texttt{corpus.txt}. Write a function **`load(f)`** to read the file into a

string. Use **`tokenize.regexp()`** to create a tokenizer which tokenizes the various kinds of punctuation in this text. Use a single regular expression, with inline comments using the **`re.VERBOSE`** flag.

2. **Sentence tokenizers:** (Advanced) Develop a sentence tokenizer. Test it on the Brown Corpus, which has been grouped into sentences.

3. Use the Porter Stemmer to normalize some tokenized text, calling the stemmer on each word.

3.3 Lexical Resources (INCOMPLETE)

[This section will contain a discussion of lexical resources, focusing on WordNet, but also including the **`cmudict`** and **`timit`** corpus readers.]

3.3.1 Pronunciation Dictionary

Here we access the pronunciation of words...

```
>>> from nltk_lite.corpora import cmudict
>>> from string import join
>>> for word, num, pron in cmudict.raw():
...     if pron[-4:] == ('N', 'IH0', 'K', 'S'):
...         print word, "/", ' '.join(pron)
ATLANTIC'S / AH0 T L AE1 N IH0 K S
AUDIOTRONICS / AO2 D IY0 OW0 T R AA1 N IH0 K S
AVIONICS / EY2 V IY0 AA1 N IH0 K S
BEATNIKS / B IY1 T N IH0 K S
CALISTHENICS / K AE2 L AH0 S TH EH1 N IH0 K S
CENTRONICS / S EH2 N T R AA1 N IH0 K S
CHETNIKS / CH EH1 T N IH0 K S
CLINIC'S / K L IH1 N IH0 K S
CLINICS / K L IH1 N IH0 K S
CONICS / K AA1 N IH0 K S
CYNICS / S IH1 N IH0 K S
DIASONICS / D AY2 AH0 S AA1 N IH0 K S
DOMINIC'S / D AA1 M AH0 N IH0 K S
EBONICS / IY0 B AO1 N IH0 K S
ELECTRONICS / AH0 L EH2 K T R AA1 N IH0 K S
...
ONYX / AA1 N IH0 K S
...
PHILHARMONIC'S / F IH2 L HH AA0 R M AA1 N IH0 K S
PHOENIX / F IY1 N IH0 K S
PHONICS / F AA1 N IH0 K S
...
TELEPHONICS / T EH2 L AH0 F AA1 N IH0 K S
TONICS / T AA1 N IH0 K S
UNIX / Y UW1 N IH0 K S
...
```

3.3.2 Wordnet Semantic Network

Note

Before using WordNet it must be installed on your machine. Please see the instructions on the NLTK website.

Access wordnet as follows:

```
>>> import wordnet
```

Help on the interface is available using `help(wordnet)`.

Wordnet contains four dictionaries: `N` (nouns), `V` (verbs), `ADJ` (adjectives), and `ADV` (adverbs). Here we will focus on just the nouns.

Access the senses of a word (synsets) using `getSenses()`

```
>>> dog = wordnet.N['dog']
>>> for sense in dog.getSenses():
...     print sense
'dog' in {noun: dog, domestic dog, Canis familiaris}
'dog' in {noun: frump, dog}
'dog' in {noun: dog}
'dog' in {noun: cad, bounder, blackguard, dog, hound, heel}
'dog' in {noun: frank, frankfurter, hotdog, hot dog, dog, wiener, wienerwurst, weer
'dog' in {noun: pawl, detent, click, dog}
'dog' in {noun: andiron, firedog, dog, dog-iron}
```

Each synset has a variety of pointers to other synsets. See `dir(wordnet)` for a list. Access one of these using `getPointerTargets()`, and specify the pointer type as the argument.

```
>>> dog_canine = dog.getSenses()[0]
>>> for sense in dog_canine.getPointerTargets(wordnet.HYPONYM):
...     print sense
{noun: pooch, doggie, doggy, barker, bow-wow}
{noun: cur, mongrel, mutt}
{noun: lapdog}
{noun: toy dog, toy}
{noun: hunting dog}
{noun: working dog}
{noun: dalmatian, coach dog, carriage dog}
{noun: basenji}
{noun: pug, pug-dog}
{noun: Leonberg}
{noun: Newfoundland}
{noun: Great Pyrenees}
{noun: spitz}
{noun: griffon, Brussels griffon, Belgian griffon}
{noun: corgi, Welsh corgi}
{noun: poodle, poodle dog}
{noun: Mexican hairless}
```

Each synset has a unique hypernym. Thus, from any synset we can trace paths back to the most general synset *entity*. First we define a function to return the hypernym of a synset:

```
>>> def hypernym(sense):
...     try:
...         return sense.getPointerTargets(wordnet.HYPERNYM)[0]
...     except IndexError:
...         return None
```

Now we can write a simple program to display these hypernym paths:

```
>>> def hypernym_path(sense, depth=0):
...     if sense != None:
...         print "  " * depth, sense
...         hypernym_path(hypernym(sense), depth+1)
>>> for sense in dog.getSenses():
...     hypernym_path(sense)
'dog' in {noun: dog, domestic dog, Canis familiaris}
  {noun: canine, canid}
    {noun: carnivore}
      {noun: placental, placental mammal, eutherian, eutherian mammal}
        {noun: mammal}
          {noun: vertebrate, craniate}
            {noun: chordate}
              {noun: animal, animate being, beast, brute, creature, fauna}
                {noun: organism, being}
                  {noun: living thing, animate thing}
                    {noun: object, physical object}
                      {noun: entity}
```

3.3.3 Exercises

0. Familiarize yourself with the Pywordnet interface.

1. Investigate the holonym / meronym pointers for some nouns. Note that there are three kinds (member, part, substance), so access is more specific, e.g. **`MEMBER_MERONYM`**, **`SUBSTANCE_HOLONYM`**.

2. Write a program to score the similarity of 2 nouns as the depth of their first common hypernym. Evaluate your findings against the Rubenstein-Goodenough set of word pairs.

3.4 Simple Statistics with Tokens

We can do more sophisticated counting using *frequency distributions*. Abstractly, a frequency distribution is a record of the number of times each *outcome* of an *experiment* has occurred. For instance, a frequency distribution could be used to record the frequency of each word in a document (where the "experiment" is examining a word, and the "outcome" is the word's type). Frequency distributions are generally created by repeatedly running an experiment, and incrementing the count for a sample every time it is an outcome of the experiment. The following program produces a frequency distribution that records how often each word type occurs in a text, and prints the most frequently occurring word:

```
>>> from nltk_lite.probability import FreqDist
>>> from nltk_lite.corpora import genesis
>>> fd = FreqDist()
>>> for token in genesis.raw():
...     fd.inc(token)
>>> fd.max()
'the'
```

Once we construct a frequency distribution that records the outcomes of an experiment, we can use it to examine a number of interesting properties of the experiment. Some of these properties are summarized below:

Frequency Distribution Module		
Name	**Sample**	**Description**
Count	fd.count('the')	number of times a given sample occurred
Frequency	fd.freq('the')	frequency of a given sample
N	fd.N()	number of samples
Samples	fd.samples()	list of distinct samples recorded
Max	fd.max()	sample with the greatest number of outcomes

We can also use a `FreqDist` to examine the distribution of word lengths in a corpus. For each word, we find its length, and increment the count for words of this length.

```
>>> def length_dist(text):
...     fd = FreqDist()                          # initialize an empty frequency dist
...     for token in genesis.raw(text):          # for each token
...         fd.inc(len(token))                   # found another word with this lengt
...     for i in range(15):                      # for each length from 0 to 14
...         print "%2d" % int(100*fd.freq(i)), # print the percentage of words with
...     print

>>> length_dist('english-kjv')
 0  2 14 28 21 13  7  5  2  2  0  0  0  0  0
>>> length_dist('finnish')
 0  0  9  6 10 16 16 12  9  6  3  2  2  1  0
```

3.4.1 Conditional Frequency Distributions

A *condition* specifies the context in which an experiment is performed. Often, we are interested in the effect that conditions have on the outcome for an experiment. A *conditional frequency distribution* is a collection of frequency distributions for the same experiment, run under different conditions. For example, we might want to examine how the distribution of a word's length (the outcome) is affected by the word's initial letter (the condition).

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
>>> for text in genesis.items:
...     for word in genesis.raw(text):
...         cfdist[text].inc(len(word))
```

> **Note**
>
> The above program requires the NLTK corpora to be installed and an environment variable to be set. Please see the NLTK installation instructions.

To plot the results, we construct a list of points, where the x coordinate is the word length, and the y coordinate is the frequency with which that word length is used:

```
>>> for cond in cfdist.conditions():
...     wordlens = cfdist[cond].samples()
...     wordlens.sort()
...     points = [(i, cfdist[cond].freq(i)) for i in wordlens]
```

We can plot these points using the **`Plot`** function defined in **`nltk_lite.draw.plot`**, as follows: **`Plot(points).mainloop()`**

3.4.2 Predicting the Next Word

Conditional frequency distributions are often used for prediction. *Prediction* is the problem of deciding a likely outcome for a given run of an experiment. The decision of which outcome to predict is usually based on the context in which the experiment is performed. For example, we might try to predict a word's text (outcome), based on the text of the word that it follows (context).

To predict the outcomes of an experiment, we first examine a representative *training corpus*, where the context and outcome for each run of the experiment are known. When presented with a new run of the experiment, we simply choose the outcome that occurred most frequently for the experiment's context.

We can use a **`ConditionalFreqDist`** to find the most frequent occurrence for each context. First, we record each outcome in the training corpus, using the context that the experiment was run under as the condition. Then, we can access the frequency distribution for a given context with the indexing operator, and use the **`max()`** method to find the most likely outcome.

We will now use a **`ConditionalFreqDist`** to predict the most likely next word in a text. To begin, we load a corpus from a text file, and create an empty **`ConditionalFreqDist`**:

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
```

We then examine each token in the corpus, and increment the appropriate sample's count. We use the variable **`prev`** to record the previous word.

```
>>> prev = None
>>> for word in genesis.raw():
...     cfdist[prev].inc(word)
...     prev = word
```

> **Note**
>
> Sometimes the context for an experiment is unavailable, or does not exist. For example, the first token in a text does not follow any word. In these cases, we must decide what context to use. For this example, we use **`None`** as the context for the first token. Another option would be to discard the first token.

Once we have constructed a conditional frequency distribution for the training corpus, we can use it to find the most likely word for any given context. For example, taking the word *living* as our context, we can inspect all the words that occurred in that context.

```
>>> word = 'living'
>>> cfdist[word].samples()
['creature,', 'substance', 'soul.', 'thing', 'thing,', 'creature']
```

We can set up a simple loop to generate text: we set an initial context, picking the most likely token in that context as our next word, and then using that word as our new context:

```
>>> word = 'living'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
living creature that he said, I will not be a wife of the land
of the land of the land
```

This simple approach to text generation tends to get stuck in loops, as demonstrated by the text generated above. A more advanced approach would be to randomly choose each word, with more frequent words chosen more often.

3.4.3 Exercises

1. **Zipf's Law**: Let *f(w)* be the frequency of a word *w* in free text. Suppose that all the words of a text are ranked according to their frequency, with the most frequent word first. Zipf's law states that the frequency of a word type is inversely proportional to its rank (i.e. f*r=k, for some constant k). For example, the 50th most common word type should occur three times as frequently as the 150th most common word type.

   a) Write a function to process a large text and plot word frequency against word rank using the nltk_lite.draw.plot module. Do you confirm Zipf's law? (Hint: it helps to set the axes to log-log.) What is going on at the extreme ends of the plotted line?

   b) Generate random text, e.g. using `random.choice("abcdefg ")`, taking care to include the space character. Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, and generate the Zipf plot as before, and compare the two plots. What do you make of Zipf's Law in the light of this?

2. **Predicting the next word**: The word prediction program we saw in this chapter quickly gets stuck in a cycle. Modify the program to choose the next word randomly, from a list of the *n* most likely words in the given context. (Hint: store the *n* most likely words in a list `lwords` then randomly choose a word from the list using `random.choice()`.)

   a) Select a particular genre, such as a section of the Brown Corpus, or a genesis translation, or one of the Gutenberg texts. Train your system on this corpus and get it to generate random text. You may have to experiment with different start words. How intelligible is the text? Discuss the strengths and weaknesses of this method of generating random text.

b) Try the same approach with different genres, and with different amounts of training data. What do you observe?

c) Now train your system using two distinct genres and experiment with generating text in the hybrid genre. As before, discuss your observations.

3. Write a program to implement one or more text readability scores (see http://en.wikipedia.org/wiki/Readability).

4. (Advanced) **Statistically Improbable Phrases:** Design an algorithm to find the statistically improbable phrases of a document collection. http://www.amazon.com/gp/search-inside/sipshelp.html/

3.5 What is a Word?

Linguists have devoted considerable effort to distinguishing different ways in which the term *word*:dt is used. So far we have focused on *orthographic words*:dt strings of characters which can be separated by various textual criteria such as the presence of whitespace. However, for linguists, the spoken word is often regarded as primary, in the sense that natural languages have all evolved as spoken mediums, and children learn to speak long before they can write and read (if indeed they ever do). Subjectively, we hear spoken utterances as a succession of words, but it is rarely the case that there are perceptible gaps between spoken words in conversational speech. Nevertheless, spoken words can and do occur in isolation. Using a variety of (sometime language-dependent) criteria, certain phonological units are classed as *phonological words*, and these units need not correspond to orthographic words: an example is the utterance *wanna* as a contraction of the words *want* and *to*. Returning to an example mentioned before, note that the form *n't* in *didn't* cannot form a phonological word by itself, and is sometimes called a **clitic** or **leaner**; it needs to to combine with a 'host' word before it can be uttered in normal speech.

Independent of the distinction between spoken and written language, an important notion is that of a **lexeme** or **lexical item**. This corresponds broadly to the notion of a word that you might look up in a dictionary of English or some other language. For example, in order to find the meaning of the word *said*, you need to know first that it is a particular **grammatical form** of the lexeme *SAY*. (We adopt the standard convention of representing lexemes with upper-case forms.) Similarly, *say*, *says* and *saying* are also grammatical forms of *SAY*. While *said*, *says* and *saying* are morphologically inflected, *say* lacks any morphological inflection and is therefore termed the **base** form. In English, the base form is conventionally used as the **lemma** (or **citation form**) for a word. It is the lemma that is chosen to represent the corresponding lexeme. (For example, the main entry for lexical item will be listed under the word's lemma.)

Many of the word-like forms that occur in text have received little attention from linguists but are nevertheless so prevalent that they need to be dealt with in many NLP applications. These include abbreviations such as *Dept.* or *Mr.* and acronyms such as *NATO*, *Interpol* or *SQL*. Also of interest are symbols such as $ (*dollar*) and @ (*at*). Unlike ordinary punctuation marks, these symbols (sometimes called **logograms**) stand for words, and would be spoken as such if the text were read aloud.

3.6 Summary

In this chapter we saw that we can do a variety of interesting language processing tasks that focus solely on words. Tokenization turns out to be far more difficult than expected. Other kinds of tokenization, such as sentence tokenization, are left for the exercises. No single solution works well across-the-board, and we must decide what counts as a token depending on the application domain. We also looked at normalization and saw how it collapses distinctions between tokens.

In the next chapter we will look at word classes and automatic tagging. We will continue to use lightweight methods to recognize linguistic structure.

3.7 Further Reading

John Hopkins Center for Language and Speech Processing, 1999 Summer Workshop on Normalization of Non-Standard Words: Final Report http://www.clsp.jhu.edu/ws99/projects/normal/report.pdf

SIL Glossary of Linguistic Terms: http://www.sil.org/linguistics/GlossaryOfLinguisticTerms/

Language Files: Materials for an Introduction to Language and Linguistics (Eighth Edition), The Ohio State University Department of Linguistics, http://www.ling.ohio-state.edu/publications/files/