Chapter 13

Managing Linguistic Data

13.1 Introduction

Linguistic fieldwork deals with a variety of data types, the most important being lexicons, paradigms and texts. A lexicon is a database of words, minimally containing part of speech information and glosses. A paradigm, broadly construed, is any kind of rational tabulation of words or phrases to illustrate contrasts and systematic variation. A text is essentially any larger unit such as a narrative or a conversation. In addition to these data types, linguistic fieldwork involves various kinds of description, such as field notes, grammars and analytical papers.

These various kinds of data and description enter into a complex web of relations. For example, the discovery of a new word in a text may require an update to the lexicon and the construction of a new paradigm (e.g. to correctly classify the word). Such updates may occasion the creation of some field notes, the extension of a grammar and possibly even the revision of the manuscript for an analytical paper. Progress on description and analysis gives fresh insights about how to organise existing data and it informs the quest for new data. Whether one is sorting data, or generating tabulations, or gathering statistics, or searching for a (counter-)example, or verifying the transcriptions used in a manuscript, the principal challenge is computational.

In the following we will consider various methods for manipulating linguistic field data using the Natural Language Toolkit. We begin by considering methods for processing data created with proprietary tools (e.g. Microsoft Office products). The bulk of the discussion focusses on field data stored in the popular *Toolbox* format.

> **Note**
>
> Other sections, still to be written, will cover the collection and curation of corpora; the lifecycle of linguistic data, including coding/annotation and automatic learning of annotations; and language resources more generally which are crucial in linguistic research and commercial NLP.

13.2 XML and ElementTree

- inspecting and processing XML
- example: find nodes matching some criterion and add an attribute
- Shakespeare XML corpus example

13.3 Tools and technologies for language documentation and description

Language documentation projects are increasing in their reliance on new digital technologies and software tools. Bird and Simons (2003) identified and categorized a wide variety of these tools. We briefly review these here, and mention various ways that our own programs can interface with them.

13.3.1 General purpose tools

Conventional office software is widely used in computer-based language documentation work, given its familiarity and ready availability. This includes word processors and spreadsheets.

Word Processors. These are often used in creating dictionaries and interlinear texts. However, it is rather time-consuming to maintain the consistency of the content and format. Consider a dictionary in which each entry has a part-of-speech field, drawn from a set of 20 possibilities, displayed after the pronunciation field, and rendered in 11-point bold. No convential word processor has search or macro functions capable of verifying that all part-of-speech fields have been correctly entered and displayed. This task requires exhaustive manual checking. If the word processor permits the document to be saved in a non-proprietary format, such as RTF, HTML, or XML, it may be possible to write programs to do this checking automatically.

Consider the following fragment of a lexical entry: "sleep [sli:p] **vi** *condition of body and mind...*". We can enter this in MSWord, then "Save as Web Page", then inspect the resulting HTML:

```
<p class=MsoNormal>sleep
  <span style='mso-spacerun:yes'> </span>
  [<span class=SpellE>sli:p</span>]
  <span style='mso-spacerun:yes'> </span>
  <b><span style='font-size:11.0pt'>vi</span></b>
  <span style='mso-spacerun:yes'> </span>
  <i>a condition of body and mind ...<o:p></o:p></i>
</p>
```

Observe that the entry is represented as an HTML paragraph, using the `<p>` element, and that the part of speech appears inside a `<span style='font-size:11.0pt'>` element. The following program defines the set of legal parts-of-speech, `legal_pos`. Then it extracts all 11-point content from the `dict.htm` file and stores it in the set `used_pos`. Observe that the search pattern contains a parenthesized sub-expression; only the material that matches this sub-expression is returned by `re.findall`. Finally, the program constructs the set of illegal parts-of-speech as `used_pos - legal_pos`:

```
>>> import re
>>> legal_pos = set(['n', 'v.t.', 'v.i.', 'adj', 'det'])
>>> pattern = re.compile(r"'font-size:11.0pt'>([a-z.]+)<")
>>> document = open("dict.htm").read()
>>> used_pos = set(re.findall(pattern, document))
>>> illegal_pos = used_pos.difference(legal_pos)
>>> print list(illegal_pos)
['v.i', 'intrans']
```

This simple program represents the tip of the iceberg. We can develop sophisticated tools to check the consistency of word processor files, and report errors so that the maintainer of the dictionary can correct the original file *using the original word processor*. We can write other programs to *convert*

the data into a different format. For example, the following program extracts the words and their pronunciations and generates output in "comma-separated value" (CSV) format:

```
>>> import re
>>> document = open("dict.htm").read()
>>> document = re.sub("[\r\n]", "", document)
>>> word_pattern = re.compile(r">([\w]+)")
>>> pron_pattern = re.compile(r"\[.*>([a-z:]+)<.*\]")
>>> for entry in document.split("<p"):
...     word_match = word_pattern.search(entry)
...     pron_match = pron_pattern.search(entry)
...     if word_match and pron_match:
...         lex = word_match.group(1)
...         pos = pron_match.group(1)
...         print '"%s","%s"' % (lex, pos)
"sleep","sli:p"
"walk","wo:k"
"wake","weik"
```

We could also produce output in the Toolbox format (to be discussed in detail later):

```
\lx sleep
\ph sli:p
\ps v.i
\gl a condition of body and mind ...

\lx walk
\ph wo:k
\ps v.intr
\gl progress by lifting and setting down each foot ...

\lx wake
\ph weik
\ps intrans
\gl cease to sleep
```

Spreadsheets. These are often used for wordlists or paradigms. A comparative wordlist may be stored in a spreadsheet, with a row for each cognate set, and a column for each language. Examples are available from `www.rosettaproject.org`. Programs such as Excel can export spreadsheets in the CSV format, and we can write programs to manipulate them, with the help of Python's `csv` module. For example, we may want to print out cognates having an edit-distance of at least three from each other (i.e. 3 insertions, deletions, or substitutions).

Databases. Sometimes lexicons are stored in a full-fledged relational database. When properly normalized, these databases can implement many well-formedness constraints. For example, we can require that all parts-of-speech come from a specified vocabulary by declaring that the part-of-speech field is an *enumerated type*. However, the relational model is often too restrictive for linguistic data, which typically has many optional and repeatable fields (e.g. dictionary sense definitions and example sentences). Query languages such as SQL cannot express many linguistically-motivated queries, e.g. *find all words that appear in example sentences for which no dictionary entry is provided*. Now supposing that the database supports exporting data to CSV format, and that we can save the data to a file `dict.csv`:

```
    "sleep","sli:p","v.i","a condition of body and mind ..."
    "walk","wo:k","v.intr","progress by lifting and setting down each foot ..."
    "wake","weik","intrans","cease to sleep"
```

Now we can express this query in the following program:

```
>>> import csv
>>> lexemes = []
>>> defn_words = []
>>> for row in csv.reader(open("dict.csv")):
...     lexeme, pron, pos, defn = row
...     lexemes.append(lexeme)
...     defn_words += defn.split()
>>> undefined = list(set(defn_words).difference(set(lexemes)))
>>> undefined.sort()
>>> print undefined
['...', 'a', 'and', 'body', 'by', 'cease', 'condition', 'down', 'each',
'foot', 'lifting', 'mind', 'of', 'progress', 'setting', 'to']
```

13.4 Processing Toolbox Data

Over the last two decades, several dozen tools have been developed that provide specialized support for linguistic data management. (Please see Bird and Simons 2003 for a detailed list of such tools.) Perhaps the single most popular tool for managing linguistic field data is *Shoebox*, recently renamed *Toolbox*. Toolbox uses a simple file format which we can easily read and write, permitting us to apply computational methods to linguistic field data. In this section we discuss a variety of techniques for manipulating Toolbox data in ways that are not supported by the Toolbox software.

A Toolbox file consists of a collection of *entries* (or *records*), where each record is made up of one or more *fields*. Here is an example of an entry taken from a Toolbox dictionary of Rotokas. (Rotokas is an East Papuan language spoken on the island of Bougainville; this data was provided by Stuart Robinson, and is a sample from a larger lexicon):

```
\lx kaa
\ps N
\pt MASC
\cl isi
\ge cooking banana
\tkp banana bilong kukim
\pt itoo
\sf FLORA
\dt 12/Aug/2005
\ex Taeavi iria kaa isi kovopaueva kaparapasia.
\xp Taeavi i bin planim gaden banana bilong kukim tasol long paia.
\xe Taeavi planted banana in order to cook it.
```

This lexical entry contains the following fields: `lx` lexeme; `ps` part-of-speech; `pt` part-of-speech; `cl` classifier; `ge` English gloss; `tkp` Tok Pisin gloss; `sf` Semantic field; `dt` Date last edited; `ex` Example sentence; `xp` Pidgin translation of example; `xe` English translation of example. These field names are preceded by a backslash, and must always appear at the start of a line. The characters of the field names must be alphabetic. The field name is separated from the field's contents by whitespace. The contents can be arbitrary text, and can continue over several lines (but cannot contain a line-initial backslash).

13.4.1 Accessing Toolbox Data

We can use the `toolbox.parse_corpus()` method to access a Toolbox file and load it into an `elementtree` object.

```
>>> from nltk_lite.corpora import toolbox
>>> lexicon = toolbox.parse_corpus('rotokas.dic')
```

There are two ways to access the contents of the lexicon object, by indexes and by paths. Indexes use the familiar syntax, thus `lexicon[3]` returns entry number 3 (which is actually the fourth entry counting from zero). And `lexicon[3][0]` returns its first field:

```
>>> lexicon[3][0]
<Element lx at 77bd28>
>>> lexicon[3][0].tag
'lx'
>>> lexicon[3][0].text
'kaa'
```

We can iterate over all the fields of a given entry:

```
>>> print toolbox.to_sfm_string(lexicon[3])
\lx kaa
\ps N
\pt MASC
\cl isi
\ge cooking banana
\tkp banana bilong kukim
\pt itoo
\sf FLORA
\dt 12/Aug/2005
\ex Taeavi iria kaa isi kovopaueva kaparapasia.
\xp Taeavi i bin planim gaden banana bilong kukim tasol long paia.
\xe Taeavi planted banana in order to cook it.
```

The second way to access the contents of the lexicon object uses paths. The lexicon is a series of `record` objects, each containing a series of field objects, such as `lx` and `ps`. We can conveniently address all of the lexemes using the path `record/lx`. Here we use the `findall()` function to search for any matches to the path `record/lx`, and we access the text content of the element, normalising it to lowercase.

```
>>> [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
['kaa', 'kaa', 'kaa', 'kaakaaro', 'kaakaaviko', 'kaakaavo', 'kaakaoko',
'kaakasi', 'kaakau', 'kaakauko', 'kaakito', 'kaakuupato', ..., 'kuvuto']
```

13.4.2 Adding and Removing Fields

It is often convenient to add new fields that are derived from existing ones. Such fields often facilitate analysis. For example, let us define a function which maps a string of consonants and vowels to the corresponding CV sequence, e.g. `kakapua` would map to `CVCVCVV`.

```
>>> import re
>>> def cv(s):
...     s = s.lower()
```

```
...     s = re.sub(r'[^a-z]',     r'_', s)
...     s = re.sub(r'[aeiou]',    r'V', s)
...     s = re.sub(r'[^V_]',      r'C', s)
...     return (s)
```

This mapping has four steps. First, the string is converted to lowercase, then we replace any non-alphabetic characters `[^a-z]` with an underscore. Next, we replace all vowels with `V`. Finally, anything that is not a `V` or an underscore must be a consonant, so we replace it with a `C`. Now, we can scan the lexicon and add a new `cv` field after every `lx` field.

```
>>> from nltk_lite.etree.ElementTree import SubElement
>>> for entry in lexicon:
...     for field in entry:
...         if field.tag == 'lx':
...             cv_field = SubElement(entry, 'cv')
...             cv_field.text = cv(field.text)
```

Let's see what this does for a particular entry. Note the last line of output, which shows the new CV field:

```
>>> print toolbox.to_sfm_string(lexicon[53])
\lx kaeviro
\ps V
\pt A
\ge lift off
\ge take off
\tkp go antap
\sc MOTION
\vx 1
\nt used to describe action of plane
\dt 03/Jun/2005
\ex Pita kaeviroroe kepa kekesia oa vuripierevo kiuvu.
\xp Pita i go antap na lukim haus win i bagarapim.
\xe Peter went to look at the house that the wind destroyed.
\cv CVVCVCV
```

(NB. To insert this field in a different position, we need to create the new `cv` field using `Element('cv')`, assign a text value to it then use the `insert()` method of the parent element.)

This technique can be used to make copies of Toolbox data that *lack* particular fields. For example, we may want to sanitise our lexical data before giving it to others, by removing unnecessary fields (e.g. fields containing personal comments.)

```
>>> retain = ('lx', 'ps', 'ge')
>>> for entry in lexicon:
...     entry[:] = [f for f in entry if f.tag in retain]
>>> print toolbox.to_sfm_string(lexicon[53])
\lx kaeviro
\ps V.A
\ge lift off
\ge take off
```

13.4.3 Formatting Entries

We can also print a formatted version of a lexicon. It allows us to request specific fields without needing to be concerned with their relative ordering in the original file.

```
>>> lexicon = toolbox.parse_corpus('rotokas.dic')
>>> for entry in lexicon[70:80]:
...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     ge = entry.findtext('ge')
...     print "%s (%s) '%s'" % (lx, ps, ge)
kakae (???) 'small'
kakae (CLASS) 'child'
kakaevira (ADV) 'small-like'
kakapikoa (???) 'small'
kakapikoto (N) 'newborn baby'
kakapu (V) 'place in sling for purpose of carrying'
kakapua (N) 'sling for lifting'
kakara (N) 'arm band'
Kakarapaia (N) 'village name'
kakarau (N) 'frog'
```

Note

Producing CSV output
We could have produced comma-separated value (CSV) format with a slightly different print statement: `print '"%s";"%s";"%s"\n' % (lx, ps, ge)`

We can use the same idea to generate HTML tables instead of plain text. This would be useful for publishing a Toolbox lexicon on the web. It produces HTML elements `<table>`, `<tr>` (table row), and `<td>` (table data).

```
>>> html = "<table>\n"
>>> for entry in lexicon[70:80]:
...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     ge = entry.findtext('ge')
...     html += "  <tr><td>%s</td><td>%s</td><td>%s</td></tr>\n" % (lx, ps, ge)
>>> html += "</table>"
>>> print html
<table>
  <tr><td>kakae</td><td>???</td><td>small</td></tr>
  <tr><td>kakae</td><td>CLASS</td><td>child</td></tr>
  <tr><td>kakaevira</td><td>ADV</td><td>small-like</td></tr>
  <tr><td>kakapikoa</td><td>???</td><td>small</td></tr>
  <tr><td>kakapikoto</td><td>N</td><td>newborn baby</td></tr>
  <tr><td>kakapu</td><td>V</td><td>place in sling for purpose of carrying</td></tr>
  <tr><td>kakapua</td><td>N</td><td>sling for lifting</td></tr>
  <tr><td>kakara</td><td>N</td><td>arm band</td></tr>
  <tr><td>Kakarapaia</td><td>N</td><td>village name</td></tr>
  <tr><td>kakarau</td><td>N</td><td>frog</td></tr>
</table>
```

XML output

```
>>> import sys
>>> from nltk_lite.etree.ElementTree import ElementTree
>>> tree = ElementTree(lexicon[3])
>>> tree.write(sys.stdout)
<record>
  <lx>kaa</lx>
  <ps>N</ps>
  <pt>MASC</pt>
  <cl>isi</cl>
  <ge>cooking banana</ge>
  <tkp>banana bilong kukim</tkp>
  <pt>itoo</pt>
  <sf>FLORA</sf>
  <dt>12/Aug/2005</dt>
  <ex>Taeavi iria kaa isi kovopaueva kaparapasia.</ex>
  <xp>Taeavi i bin planim gaden banana bilong kukim tasol long paia.</xp>
  <xe>Taeavi planted banana in order to cook it.</xe>
</record>
```

13.4.4 Exploration

In this section we consider a variety of analysis tasks.

Reduplication: First, we will develop a program to find reduplicated words. In order to do this we need to store all verbs, along with their English glosses. We need to keep the glosses so that they can be displayed alongside the wordforms. The following code defines a Python dictionary `lexgloss` which maps verbs to their English glosses:

```
>>> lexgloss = {}
>>> for entry in lexicon:
...     lx = entry.findtext('lx')
...     if lx and entry.findtext('ps')[0] == 'V':
...         lexgloss[lx] = entry.findtext('ge')
kasi (burn); kasikasi (angry)
kee (shatter); keekee (chipped)
kauo (jump); kauokauo (jump up and down)
kea (confused); keakea (lie)
kape (unable to meet); kapekape (embrace)
kapo (fasten.cover.strip); kapokapo (fasten.cover.strips)
kavo (collect); kavokavo (perform sorcery)
karu (open); karukaru (open)
kare (return); karekare (return)
kari (rip); karikari (tear)
kae (blow); kaekae (tempt)
```

Next, for each verb `lex`, we will check if the lexicon contains the reduplicated form `lex+lex`. If it does, we report both forms along with their glosses.

```
>>> for lex in lexgloss:
...     if lex+lex in lexgloss:
...         print "%s (%s); %s (%s)" %\
...             (lex, lexgloss[lex], lex+lex, lexgloss[lex+lex])
```

Complex Search Criteria: Phonological description typically identifies the segments, alternations, syllable canon and so forth. It is relatively straightforward to count up the occurrences of all the different types of CV syllables that occur in lexemes.

In the following example, we first import the regular expression and probability modules. Then we iterate over the lexemes to find all sequences of a non-vowel `[^aeiou]` followed by a vowel `[aeiou]`.

```
>>> from nltk_lite.tokenize import regexp
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> lexemes = [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
>>> for lex in lexemes:
...     for syl in regexp(lex, pattern=r'[^aeiou][aeiou]'):
...         fd.inc(syl)
```

Now, rather than just printing the syllables and their frequency counts, we can tabulate them to generate a useful display.

```
>>> for vowel in 'aeiou':
...     for cons in 'ptkvsr':
...          print '%s%s:%4d ' % (cons, vowel, fd.count(cons+vowel)),
...     print
pa:  83  ta:  47  ka: 428  va:  93  sa:   0  ra: 187
pe:  31  te:   8  ke: 151  ve:  27  se:   0  re:  63
pi: 105  ti:   0  ki:  94  vi: 105  si: 100  ri:  84
po:  34  to: 148  ko: 430  vo:  48  so:   2  ro:  89
pu:  51  tu:  37  ku: 175  vu:  49  su:   1  ru:  79
```

Consider the `t` and `s` columns, and observe that `ti` is not attested, while `si` is frequent. This suggests that a phonological process of palatalisation is operating in the language. We would then want to consider the other syllables involving `s` (e.g. the single entry having *su*, namely *kasuari* 'cassowary' is a loanword).

Prosodically-motivated search: A phonological description may include an examination of the segmental and prosodic constraints on well-formed morphemes and lexemes. For example, we may want to find trisyllabic verbs ending in a long vowel. Our program can make use of the fact that syllable onsets are obligatory and simple (only consist of a single consonant). First, we will encapsulate the syllabic counting part in a separate function. It gets the CV template of the word `cv(word)` and counts the number of consonants it contains:

```
>>> def num_cons(word):
...     template = cv(word)
...     return template.count('C')
```

We also encapsulate the vowel test in a function, as this improves the readability of the final program. This function returns the value `True` just in case `char` is a vowel.

```
>>> def is_vowel(char):
...     return (char in 'aeiou')
```

Over time we may create a useful collection of such functions. We can save them in a file `utilities.py`, and then at the start of each program we can simply import all the functions in one go using `from utilities import *`. We take the entry to be a verb if the first letter of its part of speech is a `V`. Here, then, is the program to display trisyllabic verbs ending in a long vowel:

```
>>> for entry in lexicon:
...     lx = entry.findtext('lx')
...     if lx:
...         ps = entry.findtext('ps')
...         if num_cons(lx) == 3 and ps[0] == 'V'\
...           and is_vowel(lx[-1]) and is_vowel(lx[-2]):
...             ge = entry.findtext('ge')
...             print "%s (%s) '%s'" % (lx, ps, ge)
kaetupie (V.B) 'tighten'
kakupie (V.B) 'shout'
kapatau (V.B) 'add to'
kapuapie (V.B) 'wound'
kapupie (V.B) 'close tight'
kapuupie (V.B) 'close'
karepie (V.B) 'return'
karivai (V.A) 'have an appetite'
kasipie (V.B) 'care for'
kaukaupie (V.B) 'shine intensely'
kavorou (V.A) 'covet'
kavupie (V.B) 'leave.behind'
...
kuverea (V.A) 'incomplete'
```

Finding Minimal Sets: In order to establish a contrast segments (or lexical properties, for that matter), we would like to find pairs of words which are identical except for a single property. For example, the words pairs *mace* vs *maze* and *face* vs *faze* — and many others like them — demonstrate the existence of a phonemic distinction between *s* and *z* in English. NLTK-Lite provides flexible support for constructing minimal sets, using the `MinimalSet()` class. This class needs three pieces of information for each item to be added: `context`: the material that must be fixed across all members of a minimal set; `target`: the material that changes across members of a minimal set; `display`: the material that should be displayed for each item.

Examples of Minimal Set Parameters			
Minimal Set	**Context**	**Target**	**Display**
bib, *bid*, *big*	first two letters	third letter	word
deal (N), *deal (V)*	whole word	pos	word (pos)

Table 13.1:

We begin by creating a list of parameter values, generated from the full lexical entries. In our first example, we will print minimal sets involving lexemes of length 4, with a target position of 1 (second segment). The `context` is taken to be the entire word, except for the target segment. Thus, if lex is `kasi`, then context is `lex[:1]+'_'+lex[2:]`, or `k_si`. Note that no parameters are generated if the lexeme does not consist of exactly four segments.

```
>>> from nltk_lite.utilities import MinimalSet
>>> pos = 1
>>> ms = MinimalSet((lex[:pos] + '_' + lex[pos+1:], lex[pos], lex)
...                 for lex in lexemes if len(lex) == 4)
```

Now we print the table of minimal sets. We specify that each context was seen at least 3 times.

```
>>> for context in ms.contexts(3):
...     print context + ':',
...     for target in ms.targets():
...         print "%-4s" % ms.display(context, target, "-"),
...     print
k_si: kasi -    kesi kusi kosi
k_va: kava -    -    kuva kova
k_ru: karu kiru keru kuru koru
k_pu: kapu kipu -    -    kopu
k_ro: karo kiro -    -    koro
k_ri: kari kiri keri kuri kori
k_pa: kapa -    kepa -    kopa
k_ra: kara kira kera -    kora
k_ku: kaku -    -    kuku koku
k_ki: kaki kiki -    -    koki
```

Observe in the above example that the context, target, and displayed material were all based on the lexeme field. However, the idea of minimal sets is much more general. For instance, suppose we wanted to get a list of wordforms having more than one possible part-of-speech. Then the target will be part-of-speech field, and the context will be the lexeme field. We will also display the English gloss field.

```
>>> entries = [(e.findtext('lx'), e.findtext('ps'), e.findtext('ge'))
...                for e in lexicon
...                if e.findtext('lx') and e.findtext('ps') and e.findtext('ge')]
>>> ms = MinimalSet((lx, ps[0], "%s (%s)" % (ps[0], ge))
...           for (lx, ps, ge) in entries)
>>> for context in ms.contexts()[:10]:
...     print "%10s:" % context, "; ".join(ms.display_all(context))
  kokovara: N (unripe coconut); V (unripe)
     kapua: N (sore); V (have sores)
      koie: N (pig); V (get pig to eat)
      kovo: C (garden); N (garden); V (work)
    kavori: N (crayfish); V (collect crayfish or lobster)
    korita: N (cutlet?); V (dissect meat)
      keru: N (bone); V (harden like bone)
  kirokiro: N (bush used for sorcery); V (write)
    kaapie: N (hook); V (snag)
       kou: C (heap); V (lay egg)
```

The following program uses `MinimalSet` to find pairs of entries in the corpus which have different attachments based on the *verb* only.

```
>>> from nltk_lite.utilities import MinimalSet
>>> ms = MinimalSet()
>>> for entry in ppattach.dictionary('training'):
...     target  = entry['attachment']
...     context = (entry['noun1'], entry['prep'], entry['noun2'])
...     display = (target, entry['verb'])
...     ms.add(context, target, display)
>>> for context in ms.contexts():
...     print context, ms.display_all(context)
```

Here is one of the pairs found by the program.

(1)
```
received (NP offer) (PP from group)
rejected (NP offer (PP from group))
```

This finding gives us clues to a structural difference: the verb *receive* usually comes with two following arguments; we receive something *from* someone. In contrast, the verb *reject* only needs a single following argument; we can reject something without needing to say where it originated from.

13.4.5 Example Applications: Improving Access to Lexical Resources

A lexicon constructed as part of field-based research is a potential *language resource* for speakers of a language. Even when the language in question has a standard writing system, many speakers will not be literate in the language. They may be able to attempt an approximate spelling for a word, or they may prefer to access the dictionary via an index which uses the language of wider communication. In this section we deal with the first of these. The second is left to the reader as an exercise. We will also generate a wordfinder puzzle which can be used to test knowledge of lexical items.

Fuzzy Spelling (notes)

Confusible sets of segments: if two segments are confusible, map them to the same integer.

```
>>> group = {
...     ' ':0,                          # blank (for short words)
...     'p':1,  'b':1,  'v':1,          # labials
...     't':2,  'd':2,  's':2,          # alveolars
...     'l':3,  'r':3,                  # sonorant consonants
...     'i':4,  'e':4,                  # high front vowels
...     'u':5,  'o':5,                  # high back vowels
...     'a':6                           # low vowels
... }
```

Soundex: idea of a signature. Words with the same signature considered confusible. Consider first letter of a word to be so cognitively salient that people will not get it wrong.

```
>>> def soundex(word):
...     if len(word) == 0: return word  # sanity check
...     word += '    '                  # ensure word long enough
...     c0 = word[0].upper()
...     c1 = group[word[1]]
...     cons = filter(lambda x: x in 'pbvtdslr ', word[2:])
...     c2 = group[cons[0]]
...     c3 = group[cons[1]]
...     return "%s%d%d%d" % (c0, c1, c2, c3)
>>> print soundex('kalosavi')
K632
>>> print soundex('ti')
T400
```

Now we can build a soundex index of the lexicon:

```
>>> soundex_idx = {}
>>> for lex in lexemes:
...     code = soundex(lex)
...     if code not in soundex_idx:
```

```
...             soundex_idx[code] = set()
...         soundex_idx[code].add(lex)
```

We should sort these candidates by proximity with the target word.

```
>>> from nltk_lite.utilities import edit_dist
>>> def fuzzy_spell(target):
...     scored_candidates = []
...     code = soundex(target)
...     for word in soundex_idx[code]:
...         dist = edit_dist(word, target)
...         scored_candidates.append((dist, word))
...     scored_candidates.sort()
...     return [w for (d,w) in scored_candidates[:10]]
```

Finally, we can look up a word to get approximate matches:

```
>>> fuzzy_spell('kokopouto')
['kokopeoto', 'kokopuoto', 'kokepato', 'koovoto', 'koepato', 'kooupato', 'kopato', 'kopiito
>>> fuzzy_spell('kogou')
['kogo', 'koou', 'kokeu', 'koko', 'kokoa', 'kokoi', 'kokoo', 'koku', 'kooe', 'kooku']
```

Wordfinder Puzzle

Here we will generate a grid of letters, containing words found in the dictionary. First we remove any duplicates and disregard the order in which the lexemes appeared in the dictionary. We do this by converting it to a set, then back to a list. Then we select the first 200 words, and then only keep those words having a reasonable length.

```
>>> words = list(set(lexemes))
>>> words = words[:200]
>>> words = [w for w in words if 3 <= len(w) <= 12]
```

Now we generate the wordfinder grid, and print it out.

```
>>> from nltk_lite.misc.wordfinder import wordfinder
>>> grid, used = wordfinder(words)
>>> for i in range(len(grid)):
...     for j in range(len(grid[i])):
...         print grid[i][j],
...     print
O G H K U U V U V K U O R O V A K U N C
K Z O T O I S E K S N A I E R E P A K C
I A R A A K I O Y O V R S K A W J K U Y
L R N H N K R G V U K G I A U D J K V N
I I Y E A U N O K O O U K T R K Z A E L
A V U K O X V K E R V T I A A E R K R K
A U I U G O K U T X U I K N V V L I E O
R R K O K N U A J Z T K A K O O S U T R
I A U A U A S P V F O R O O K I C A O U
V K R R T U I V A O A U K V V S L P E K
A I O A I A K R S V K U S A A I X I K O
P S V I K R O E O A R E R S E T R O J X
O I I S U A G K R O R E R I T A I Y O A
```

```
R R R A T O O K O I K I W A K E A A R O
O E A K I K V O P I K H V O K K G I K T
K K L A K A A R M U G E P A U A V Q A I
O O O U K N X O G K G A R E A A P O O R
K V V P U J E T Z P K B E I E T K U R A
N E O A V A E O R U K B V K S Q A V U E
C E K K U K I K I R A E K O J I Q K K K
```

Finally we generate the words which need to be found.

```
>>> for i in range(len(used)):
...     print "%-12s" % used[i],
...     if float(i+1)%5 == 0: print
KOKOROPAVIRA KORO ROVIVIRA KAEREASIVIRA KOTOKOTOARA  KOPUASIVIRA
KATAITOAREI  KAITUTUVIRA   KERIKERISI   KOKARAPATO   KOKOVURITO
KAUKAUVIRA   KOKOPUVIRA    KAEKAESOTO   KAVOVOVIRA   KOVAKOVARA
KAAREKOPIE   KAEPIEVIRA    KAPUUPIEPA   KOKORUUTO    KIKIRAEKO
KATAAVIRA    KOVOKOVOA     KARIVAITO    KARUVIRA     KAPOKARI
KUROVIRA     KITUKITU      KAKUPUTE     KAEREASI     KUKURIKO
KUPEROO      KAKAPUA       KIKISI       KAVORA       KIKIPI
KAPUA        KAARE         KOETO        KATAI        KUVA
KUSI         KOVO          KOAI
```

13.4.6 Generating Reports

Finally, we take a look at simple methods to generate summary reports, giving us an overall picture of the quality and organisation of the data.

First, suppose that we wanted to compute the average number of fields for each entry. This is just the total length of the entries (the number of fields they contain), divided by the number of entries in the lexicon:

```
>>> sum(len(entry) for entry in lexicon) / len(lexicon)
10
```

Next, let's consider some methods for discovering patterns in the lexical entries. Which fields are the most frequent?

```
>>> fd = FreqDist()
>>> for entry in lexicon:
...     for field in entry:
...         fd.inc(field.tag)
>>> fd.sorted_samples()[:10]
['ge', 'ex', 'xe', 'xp', 'gp', 'lx', 'ps', 'dt', 'rt', 'eng']
```

Which sequences of fields are the most frequent?

```
>>> fd = FreqDist()
>>> for entry in lexicon:
...     fd.inc(':'.join(field.tag for field in entry))
>>> top_ten = fd.sorted_samples()[:10]
>>> print '\n'.join(top_ten)
lx:rt:ps:ge:gp:dt:ex:xp:xe
lx:ps:ge:gp:dt:ex:xp:xe
lx:ps:ge:gp:dt:ex:xp:xe:ex:xp:xe
```

```
lx:rt:ps:ge:gp:dt:ex:xp:xe:ex:xp:xe
lx:ps:ge:gp:nt:dt:ex:xp:xe
lx:ps:ge:gp:dt
lx:ps:ge:ge:gp:dt:ex:xp:xe:ex:xp:xe
lx:rt:ps:ge:ge:gp:dt:ex:xp:xe:ex:xp:xe
lx:ps:ge:ge:gp:dt:ex:xp:xe
lx:rt:ps:ge:ge:gp:dt:ex:xp:xe
```

Which pairs of consecutive fields are the most frequent?

```
>>> fd = FreqDist()
>>> for entry in lexicon:
...     previous = "0"
...     for field in entry:
...         current = field.tag
...         fd.inc("%s->%s" % (previous, current))
...         previous = current
>>> fd.sorted_samples()[:10]
['ex->xp', 'xp->xe', '0->lx', 'ge->gp', 'ps->ge', 'dt->ex', 'lx->ps',
'gp->dt', 'xe->ex', 'lx->rt']
```

Once we have analyzed the sequences of fields, we might want to write down a grammar for lexical entries, and look for entries which do not conform to the grammar. In general, toolbox entries have nested structure. Thus they correspond to a tree over the fields. We can check for well-formedness by parsing the field names. We first set up a putative grammar for the entries:

```
>>> from nltk_lite import parse
>>> grammar = parse.cfg.parse_cfg('''
...   S -> Head "ps" Glosses Comment "dt" Examples
...   Head -> "lx" | "lx" "rt"
...   Glosses -> Gloss Glosses
...   Glosses ->
...   Gloss -> "ge" | "gp"
...   Examples -> Example Examples
...   Examples ->
...   Example -> "ex" "xp" "xe"
...   Comment -> "cmt"
...   Comment ->
...   ''')
>>> rd_parser = parse.RecursiveDescent(grammar)
```

Now we try to parse each entry. Those that are accepted by the grammar prefixed with a `'+'`, and those that are rejected are prefixed with a `'-'`.

```
>>> for entry in lexicon[10:20]:
...     marker_list = [field.tag for field in entry]
...     if rd_parser.get_parse_list(marker_list):
...         print "+", ':'.join(marker_list)
...     else:
...         print "-", ':'.join(marker_list)
- lx:ps:ge:gp:sf:nt:dt:ex:xp:xe:ex:xp:xe
- lx:rt:ps:ge:gp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:gp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:gp:nt:sf:dt
```

```
- lx:ps:ge:gp:dt:cmt:ex:xp:xe:ex:xp:xe
+ lx:ps:ge:ge:ge:gp:cmt:dt:ex:xp:xe
+ lx:rt:ps:ge:gp:cmt:dt:ex:xp:xe:ex:xp:xe
+ lx:rt:ps:ge:ge:gp:dt
- lx:rt:ps:ge:ge:ge:gp:dt:cmt:ex:xp:xe:ex:xp:xe:ex:xp:xe
+ lx:rt:ps:ge:gp:dt:ex:xp:xe
```

Looking at Timestamps

```
>>> fd = FreqDist()
>>> from string import split
>>> for entry in lexicon:
...     date = entry.findtext('dt')
...     if date:
...         (day, month, year) = split(date, '/')
...         fd.inc((month, year))
>>> for time in fd.sorted_samples():
...     print time[0], time[1], ':', fd.count(time)
Feb 2005 : 307
Dec 2004 : 151
Jan 2005 : 123
Feb 2004 : 64
Sep 2004 : 49
May 2005 : 46
Mar 2005 : 37
Apr 2005 : 29
Jul 2004 : 14
Nov 2004 : 5
Oct 2004 : 5
Aug 2004 : 4
May 2003 : 2
Jan 2004 : 1
May 2004 : 1
```

To put these in time order, we need to set up a special comparison function. Otherwise, if we just sort the months, we'll get them in alphabetical order.

```
>>> month_index = {
...     "Jan" : 1, "Feb" : 2,  "Mar" : 3,  "Apr" : 4,
...     "May" : 5, "Jun" : 6,  "Jul" : 7,  "Aug" : 8,
...     "Sep" : 9, "Oct" : 10, "Nov" : 11, "Dec" : 12
... }
>>> def time_cmp(a, b):
...     a2 = a[1], month_index[a[0]]
...     b2 = b[1], month_index[b[0]]
...     return cmp(a2, b2)
```

The comparison function says that we compare two times of the form `('Mar', '2004')` by reversing the order of the month and year, and converting the month into a number to get `('2004', '3')`, then using Python's built-in `cmp` function to compare them.

Now we can get the times found in the Toolbox entries, sort them according to our `time_cmp` comparison function, and then print them in order. This time we print bars to indicate frequency:

```
>>> times = fd.samples()
>>> times.sort(cmp=time_cmp)
>>> for time in times:
...     print time[0], time[1], ':', '#' * (1 + fd.count(time)/10)
May 2003 : #
Jan 2004 : #
Feb 2004 : #######
May 2004 : #
Jul 2004 : ##
Aug 2004 : #
Sep 2004 : #####
Oct 2004 : #
Nov 2004 : #
Dec 2004 : ################
Jan 2005 : #############
Feb 2005 : ##############################
Mar 2005 : ####
Apr 2005 : ###
May 2005 : #####
```

13.4.7 Exercises

1. ☼ Write a program to filter out just the date field (`dt`) without having to list the fields we wanted to retain.

2. ☼ Print an index of a lexicon. For each lexical entry, construct a tuple of the form `(gloss , lexeme)`, then sort and print them all.

3. ☼ What is the frequency of each consonant and vowel contained in lexeme fields?

4. ☼ How many entries were last modified in 2004?

5. ◑ Write a program that scans an HTML dictionary file to find entries having an illegal part-of-speech field, and reports the *headword* for each entry.

6. ◑ Write a program to find any parts of speech (`ps` field) that occurred less than ten times. Perhaps these are typing mistakes?

7. ◑ We saw a method for discovering cases of whole-word reduplication. Write a function to find words that may contain partial reduplication. Use the `re.search()` method, and the following regular expression: `(..+)\1`

8. ◑ We saw a method for adding a `cv` field. There is an interesting issue with keeping this up-to-date when someone modifies the content of the `lx` field on which it is based. Write a version of this program to add a `cv` field, replacing any existing `cv` field.

9. ◑ Write a program to add a new field `syl` which gives a count of the number of syllables in the word.

10. ◑ Write a function which displays the complete entry for a lexeme. When the lexeme is incorrectly spelled it should display the entry for the most similarly spelled lexeme.

11. ★ Obtain a comparative wordlist in CSV format, and write a program that prints those cognates having an edit-distance of at least three from each other.

12. ★ Build an index of those lexemes which appear in example sentences. Suppose the lexeme for a given entry is *w*. Then add a single cross-reference field `xrf` to this entry, referencing the headwords of other entries having example sentences containing w. Do this for all entries and save the result as a toolbox-format file.

13.5 Language Archives

Language technology and the linguistic sciences are confronted with a vast array of language resources, richly structured, large and diverse. Multiple communities depend on language resources, including linguists, engineers, teachers and actual speakers. Thanks to recent advances in digital technologies, we now have unprecedented opportunities to bridge these communities to the language resources they need. First, inexpensive mass storage technology permits large resources to be stored in digital form, while the Extensible Markup Language (XML) and Unicode provide flexible ways to represent structured data and ensure its long-term survival. Second, digital publication on the web is the most practical and efficient means of sharing language resources. Finally, a standard resource description model and interchange method provided by the Open Language Archives Community (OLAC) makes it possible to construct a *union catalog* over multiple repositories and archives (see `http://www.language-archives.org/`).

13.5.1 Managing Metadata for Language Resources

OLAC metadata extends the *Dublin Core* metadata set with descriptors that are important for language resources.

The container for an OLAC metadata record is the element `<olac>`. Here is a valid OLAC metadata record from the Pacific And Regional Archive for Digital Sources in Endangered Cultures (PARADISEC):

```
<olac:olac xsi:schemaLocation="http://purl.org/dc/elements/1.1/ http://www.lang
  http://purl.org/dc/terms/ http://www.language-archives.org/OLAC/1.0/dcterms.x
  http://www.language-archives.org/OLAC/1.0/ http://www.language-archives.org/O
  <dc:title>Tiraq Field Tape 019</dc:title>
  <dc:identifier>AB1-019</dc:identifier>
  <dcterms:hasPart>AB1-019-A.mp3</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-A.wav</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-B.mp3</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-B.wav</dcterms:hasPart>
  <dc:contributor xsi:type="olac:role" olac:code="recorder">Brotchie, Amanda</d
  <dc:subject xsi:type="olac:language" olac:code="x-sil-MME"/>
  <dc:language xsi:type="olac:language" olac:code="x-sil-BCY"/>
  <dc:language xsi:type="olac:language" olac:code="x-sil-MME"/>
  <dc:format>Digitised: yes;</dc:format>
  <dc:type>primary_text</dc:type>
  <dcterms:accessRights>standard, as per PDSC Access form</dcterms:accessRights
  <dc:description>SIDE A<p>1. Elicitation Session - Discussion and
    translation of Lise's and Marie-Claire's Songs and Stories from
    Tape 18 (Tamedal)<p><p>SIDE B<p>1. Elicitation Session: Discussion
```

```
    of and translation of Lise's and Marie-Clare's songs and stories
    from Tape 018 (Tamedal)<p>2. Kastom Story 1 - Bislama
    (Alec). Language as given: Tiraq</dc:description>
</olac:olac>
```

Note

The remainder of this section will discuss how to manipulate OLAC metadata.

13.6 Linguistic Annotation

Note

to be written

13.7 Further Reading

Bird, Steven (1999). Multidimensional exploration of online linguistic field data *Proceedings of the 29th Meeting of the North-East Linguistic Society*, pp 33-50.

Bird, Steven and Gary Simons (2003). Seven Dimensions of Portability for Language Documentation and Description, *Language* 79: 557-582.

About this document...