

Accessing and Analyzing Linguistic Field Data

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 *Shoebox* format.

2 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.

2.1 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 conventional 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 `` 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.intr', '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"
```

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, 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', 'li
```

3 Processing Shoebox 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*. Together with its more recent incarnation (*Toolbox*), Shoebox 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 Shoebox data in ways that are not supported by the Shoebox software.

A Shoebox 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 Shoebox dictionary of Rotokas. (Rotokas is an East Papuan language spoken on the island of Bougainville; this data was provided by Stuart Robinson):

```
\lx kaa
\ps N.M
\cl isi
\ge cooking banana
\gp banana bilong kukim
\sfl FLORA
\dt 12/Feb/2005
\ex Taeavi iria kaa isi kovopauvea kaparapasiasia.
\xp Taeavi i bin planim gaden banana bilong kukim tasol long paia.
```

```
\xe Taeavi planted banana in order to cook it.
```

Each field consists of a field name (e.g. **lx**, for lexeme), and a value (e.g. **kaa**). Other fields are: **ps** part-of-speech; **cl** classifier; **ge** English gloss; **gp** Pidgin English 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).

3.1 Accessing Shoebox Data

We can use the `shoebox.raw()` method to access a Shoebox file to perform various operations that involve single-pass scanning of a lexicon. For example, here we compute the average number of fields for each entry:

```
>>> from nltk_lite.corpora import shoebox
>>> sum_size = num_entries = 0
>>> for entry in shoebox.raw('rotokas'):
...     num_entries += 1
...     sum_size += len(entry)
>>> print sum_size/num_entries
10
```

As we will see below, we can also use the `raw()` method in functions that add or remove fields, or to examine sequences of fields.

This `raw` method reads each field into a list, preserving the order of fields. Next we look at the `shoebox.dictionary()` method, which reads an entry into a Python dictionary. The following line generates a dictionary for each entry, and stores the result in the list `entries`:

```
>>> entries = list(shoebox.dictionary('rotokas'))
```

We can index into this list, thus `entries[3]` returns entry number 3 (which is actually the fourth entry counting from zero).

```
>>> from pprint import pprint
>>> pprint(entries[3])
{'cl': 'isi',
 'dt': '12/Feb/2005',
 'ex': 'Taeavi iria kaa isi kovopaueva kaparapasiasia.',
 'ge': 'cooking banana',
 'gp': 'banana bilong kukim',
 'lx': 'kaa',
 'ps': 'N.M',
 'sf': 'FLORA',
 'xe': 'Taeavi planted banana in order to cook it.',
 'xp': 'Taeavi i bin planim gaden banana bilong kukim tasol long paia.'}
```

3.2 Simple Entry Processing

The simplest approach to processing a Shoebox file is to scan through each entries, and for each entry, to scan through each field. As we have seen, the `shoebox.raw()` method returns entries, where each

entry is just a sequence of fields. Suppose we wanted to create a list of all the lexemes. We can do this as follows, starting by initialising `lexemes` to be the empty list.

```
>>> lexemes = []
>>> for entry in shoebox.raw('rotokas'):
...     for field in entry:
...         if field[0] == 'lx':
...             normalised_lexeme = field[1].lower()
...             lexemes.append(normalised_lexeme)
```

Note that we can construct the `lexemes` list much more economically using Python's *list comprehension* syntax, as follows:

```
>>> lexemes = [field[1].lower()
...             for entry in shoebox.raw('rotokas')
...             for field in entry if field[0] == 'lx']
```

Each field is stored as a tuple, e.g. `('lx', 'kakate')`. For each field in each entry, we check to see if the field name is `lx`. If it is, we convert the field's contents to lowercase, and append it to the `lexemes` list. Observe that this process does not store the entire lexicon in memory. Instead, the information we want is extracted during a single scan of the lexicon.

Adding New 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. Here we will do it for a single entry only:

```
>>> raw_entries = list(shoebox.raw('rotokas'))
>>> for field in raw_entries[50]:
...     print "\\%s %s" % field
...     if field[0] == "lx":
...         print "\\cv %s" % cv(field[1])
\lx kaeviro
\cv CVCVCVCV
\ps V.A
\ge lift off
\ge take off
\gp go antap
\nt used to describe action of plane
\dt 12/Feb/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.
```

Removing Fields: We can also use this technique to make copies of Shoebox 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')
>>> raw_entries = list(shoebox.raw('rotokas'))
>>> for entry in raw_entries[50:55]:
...     for field in entry:
...         if field[0] in retain:
...             print "\\%s %s" % field
...     print
\\lx kaeviro
\\ps V.A
<BLANKLINE>
\\lx kagave
\\ps N.F
<BLANKLINE>
\\lx kaie
\\ps V.A
<BLANKLINE>
\\lx kaiea
\\ps N.N
<BLANKLINE>
\\lx kaikaio
\\ps N.N
<BLANKLINE>
```

Formatting Entries: We can use the `shoebox.dictionary()` method to 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.

```
>>> entries = list(shoebox.dictionary('rotokas'))
>>> for entry in entries[70:80]:
...     lex = entry['lx']
...     pos = entry['ps']
...     dfn = entry['ge']
...     if 'eng' in entry:
...         dfn = entry['eng']
...     print "%s (%s) '%s'" % (lex, pos, dfn)
kakupikoto (N.N2) 'newborn baby'
kakapu (V.B) 'place in sling for purpose of carrying'
kakapua (N.N) 'sling for lifting'
kakara (N.N) 'bracelet'
Kakarapaia (N.PN) 'village name'
kakarau (N.F) 'stingray'
Kakarera (N.PN) 'name'
Kakareraia (N.???) 'name'
kakata (N.F) 'cockatoo'
kakate (N.F) 'bamboo tube for water'
```

Producing CSV output

We could have produced comma-separated value (CSV) format with a slightly different print statement: `print "%s"; "%s"; "%s"\n" % (lex, pos, dfn)`

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

```
>>> html = "<table>\n"
>>> for entry in entries[70:80]:
...     lex = entry['lx']
...     pos = entry['ps']
...     dfn = entry['ge']
...     if 'eng' in entry:
...         dfn = entry['eng']
...     html += "    <tr><td>%s</td><td>%s</td><td>%s</td></tr>\n" % (lex, pos, dfn)
>>> html += "</table>"
>>> print html
<table>
  <tr><td>kakapikoto</td><td>N.N2</td><td>newborn baby</td></tr>
  <tr><td>kakapu</td><td>V.B</td><td>place in sling for purpose of carrying</td></tr>
  <tr><td>kakapua</td><td>N.N</td><td>sling for lifting</td></tr>
  <tr><td>kakara</td><td>N.N</td><td>bracelet</td></tr>
  <tr><td>Kakarapaia</td><td>N.PN</td><td>village name</td></tr>
  <tr><td>kakarau</td><td>N.F</td><td>stingray</td></tr>
  <tr><td>Kakarera</td><td>N.PN</td><td>name</td></tr>
  <tr><td>Kakareraia</td><td>N.???</td><td>name</td></tr>
  <tr><td>kakata</td><td>N.F</td><td>cockatoo</td></tr>
  <tr><td>kakate</td><td>N.F</td><td>bamboo tube for water</td></tr>
</table>
```

3.3 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 lexemes, along with the 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 lexemes to their English glosses:

```
>>> lexgloss = {}
>>> for entry in shoebox.dictionary('rotokas'):
...     if 'lx' in entry and entry['ps'][0] == 'V':
...         lexgloss[entry['lx']] = entry['ge']
```

Next, for each lexeme `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])
kuvu (fill.up); kuvukuvu (stamp the ground)
kitu (save); kitukitu (scrub clothes)
kopa (ingest); kopakopa (gulp.down)
kasi (burn); kasikasi (angry)
koi (high pitched sound); koikoi (groan with pain)
kee (chip); keekkee (shattered)
```

```

kauo (jump); kauokauo (jump up and down)
kea (deceived); keakea (lie)
kove (drop); kovekove (drip repeatedly)
kape (unable to meet); kapekape (grip with arms not meeting)
kapo (fasten.cover.strip); kapokapo (fasten.cover.strips)
koa (skin); koakoa (remove the skin)
kipu (paint); kipukipu (rub.on)
koe (spoon out a solid); koekoe (spoon out)
kovo (work); kovokovo (surround)
kiru (have sore near mouth); kirukiru (crisp)
kotu (bite); kotukotu (grind teeth together)
kavo (collect); kavokavo (work black magic)
kuri (scrape); kurikuri (scratch repeatedly)
karu (unhook); karukaru (open)
kare (return); karekare (return)
kari (break); karikari (shred)
kiro (write); kirokiro (write)
kae (carry); kaekae (tempt)
koru (make return); korukoru (obstruct)
ku (finished with); kuku (spoonfeed)
kosi (exit); kosikosi (exit)

```

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()
>>> 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: 84  ta: 43  ka: 414  va: 87  sa: 0  ra: 185
pe: 32  te: 8  ke: 139  ve: 25  se: 1  re: 62
pi: 97  ti: 0  ki: 88  vi: 96  si: 95  ri: 83
po: 31  to: 140  ko: 403  vo: 42  so: 3  ro: 86
pu: 49  tu: 35  ku: 169  vu: 44  su: 1  ru: 72

```

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

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 shoebox.dictionary('rotokas'):
...     if 'lx' in entry:
...         lex = entry['lx']
...         pos = entry['ps']
...         if num_syls(lex) == 3 and is_vowel(lex[-1]) and is_vowel(lex[-2]) and pos[0] == 'v':
...             dfn = entry['ge']
...             print "%s (%s) '%s'" % (lex, pos, dfn)
kaetupie (V.B) 'tighten'
kakupie (V.B) 'yodel'
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) 'intense sunlight'
kavorou (V.A) 'intercept'
kavupie (V.B) 'leave behind'
kekepie (V.B) 'show'
keruria (V.A) 'determined'
ketoopie (V.B) 'make sprout from seed'
koatapie (V.B) 'accept'
koetapie (V.B) 'satisfy curiosity'
kokovae (V.A) 'sing'
kokovua (V.B) 'shave the hair line'
kopiipie (V.B) 'kill'
korupie (V.B) 'take outside'
kosipie (V.B) 'make exit'
kovopie (V.B) 'use to make work'
```

kukuvai (V.B) 'cover the head from rain or sun'
 kuvaupie (V.B) 'leave alone'
 kuverea (V.A) 'not all right'

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
<i>bib</i> , <i>bid</i> , <i>big</i>	first two letters	third letter	word
<i>deal</i> (N), <i>deal</i> (V)	whole word	pos	word (pos)

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.

```
>>> lexemes = [entry['lx'] for entry in shoebox.dictionary('rotokas')]
...           if 'lx' in entry]
>>> position = 1
>>> parameters = [(lex[:position] + '_' + lex[position+1:],
...               lex[position],
...               lex)
...               for lex in lexemes if len(lex) == 4]
```

Now, we define a function that builds creates and populates the `MinimalSet` object. For each **context**, **target**, **display** triple, it adds an entry to the minimal set.

```
>>> from nltk_lite.utilities import MinimalSet
>>> def build_min_set(parameters):
...     min_set = MinimalSet()
...     for context, target, display in parameters:
...         min_set.add(context, target, display)
...     return min_set
```

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

```
>>> ms = build_min_set(parameters)
>>> for context in ms.contexts(3):
...     print context + ':',
...     for target in ms.targets():
...         print "%-4s" % ms.display(context, target, "-"),
...     print
k_si: kasi -    kesi -    kosi
```

```

k_ru: karu kiru keru kuru koru
k_pu: kapu kipu - - kopu
k_ro: karo kiro - - koro
k_ri: kari kiri keru 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.

```

>>> parameters = [(entry['lx'], entry['ps'][0], "%s (%s)" % (entry['ps'][0], entry['gloss']))
...                 for entry in shoebox.dictionary('rotokas') if 'lx' in entry]
>>> ms = build_min_set(parameters)
>>> 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 (work); V (work)
kavori: N (lobster); V (collect crayfish or lobster)
korita: N (cutlet?); V (cut up meat)
keru: N (bone); V (harden like bone)
kirokiro: N (bush used for sorcery); V (write)
kaapie: N (fishhook); V (capture)
kou: C (heap); V (defecate)

```

3.4 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.

3.4.1 Fuzzy Spelling (notes)

Confusable sets of segments: if two segments are confusable, 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 confusable. 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')
['kokopecto', 'kokopuoto', 'kokepato', 'koovoto', 'koepato', 'kooupato', 'kopato',
>>> fuzzy_spell('kogou')
['kogo', 'koou', 'kokeu', 'koko', 'kokoa', 'kokoi', 'kokoo', 'koku', 'koe', 'kooku']
```

3.4.2 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 KOROROVIVIRA 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 KUYA
KUSI KOVO KOAI
```

3.5 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.

3.5.1 Discovering Entry Patterns

Print most frequent fields

```
>>> fd = FreqDist()
>>> for entry in shoebox.raw('rotokas'):
...     for field in entry:
...         fd.inc(field[0])
>>> fd.sorted_samples()[:10]
['ge', 'ex', 'xe', 'xp', 'gp', 'lx', 'ps', 'dt', 'rt', 'eng']
```

Discovering patterns of fields:

```
>>> fd = FreqDist()
>>> for entry in shoebox.raw('rotokas'):
...     marker_list = [field[0] for field in entry]
...     markers = ':'.join(marker_list)
...     fd.inc(markers)
>>> 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
```

Finding frequent pairs of fields:

```
>>> fd = FreqDist()
>>> for entry in shoebox.raw('rotokas'):
...     previous = "0"
...     for field in entry:
...         current = field[0]
...         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', '0->0']
```

Some shoebox entries have nested structure. Thus they correspond to a tree over the fields. We can check for well-formedness by parsing the field names, e.g.:

```
>>> from nltk_lite import parse
>>> grammar = parse.cfg.parse_grammar('''
... 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)

>>> fd = FreqDist()
>>> for entry in shoebox.raw('rotokas'):
...     marker_list = [field[0] for field in entry]
...     if rd_parser.get_parse_list(marker_list):
...         print "+", marker_list
...     else:
...         print "-", marker_list

```

3.5.2 Looking at Timestamps

```

>>> fd = FreqDist()
>>> from string import split
>>> for entry in shoebox.dictionary('rotokas'):
...     if 'dt' in entry:
...         (day, month, year) = split(entry['dt'], '/')
...         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 Shoebox 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 : #####
```

4 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/>).

4.1 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.language
```



```

http://purl.org/dc/terms/ http://www.language-archives.org/OLAC/1.0/dcterms.xsd
http://www.language-archives.org/OLAC/1.0/ http://www.language-archives.org/OLAC/
<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</dc:contributor>
<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-Claire'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.

5 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.

6 Exercises

1. 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.
2. 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.
3. Write a program to filter out just the date field (**dt**) without having to list the fields we wanted to retain.
4. Print an index of a lexicon. For each lexical entry, construct a tuple of the form (**gloss**, **lexeme**), then sort and print them all.

5. Write a program to find any parts of speech (**ps** field) that occurred less than ten times. Perhaps these are typing mistakes?
6. 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`
7. What is the frequency of each consonant and vowel contained in lexeme fields?
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. 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 shoebox-format file.
10. Write a program to add a new field **sy1** which gives a count of the number of syllables in the word.
11. 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.
12. How many entries were last modified in 2004?

About this document...

This chapter is a draft from *Introduction to Natural Language Processing*, by Steven Bird, James Curran, Ewan Klein and Edward Loper, Copyright © 2006 the authors. It is distributed with the *Natural Language Toolkit* [<http://nltk.sourceforge.net>], under the terms of the *Creative Commons Attribution-ShareAlike License* [<http://creativecommons.org/licenses/by-sa/2.5/>].