CHAPTER 4 EXERCISES

This file contains some exercises associated with the topics discussed in Chapter 4. Some of these exercises go beyond the material in Chapter 4: in particular, we didn't say all that much about tagging and parsing there, because we concluded that it was too difficult to parse tweets in a way that was actually useful. Nonetheless, it is worth including some examples of taggers and parsers here so that people can try them out for themselves. It is one thing for us to say that we don't think they're all that useful for the task addressed in the book, but readers may want to try them out for themselves.

You will need to have installed the BNC and UDT datasets, as described in *datasets.docx*, and you will also need readers for these two datasets, as described in *readers.docx*. The programs described here and in the descriptions of the various classifiers are stored in a directory with the following structure.

```
--|.--|basics--|corpora.py
               |
               |datasets.py
               |
               |directories.py
               |
               |readers.py
               |
               |utilities.py
      |
      |chapter4--|compounds.py
                 |
                 |stem1.py
                 |
                 |stem2.py
                 |
                 |stem3.py
                 |
                 |stem3a.py
                 |
                 |taggers.py
                 |
                 |tokenisers.py
      |
      |demo.py
```

You should start by running a Python interpreter at the top level in this directory and importing demo.py (which in turn will import everything in utilities.py, which is useful since this contains a number of generally useful

Words, word parts and compound words

The first thing we have to do is to find the words in our texts. We will tackle this in stages – finding the basic units, breaking them down into components in successively complex steps, and finding compounds. We will start by tokenising the text; we will then look at a basic stemming algorithm, which we will enrich by incorporating spelling rules to deal with the changes that occur at morpheme boundaries. We will then consider the ways that word parts combine to make whole words, and we will end by thinking about cases where things that appear to be free-standing words combine to make complex multi-word expressions.

The discussion below will focus on English. Similar phenomena occur in nearly all other languages, but I don't really know enough about other languages to discuss them in detail: if you want to adapt these rules for other languages, you're on your own!

Tokenising

Most languages make use of white space as the basic device for finding distinct units in written text. So a simple starting point for breaking text into units would be to split it spaces. We are going to find that this simple approach is a bit too simple, and to a slightly better job we will use regular expressions (regexes), so we might as well start by using a regex for this very simple version of the task. We define SIMPLESPACES as something which will match any non-empty sequence of non-white space characters. Tokenise takes a string and a regex and splits this string into chunks that match the regex (the programs described here should be run in the top-level directory for the repository):

```
>>> import demo
>>> from chapter4 import tokenisers
>>> tokenisers.SIMPLESPACES.pattern
```

```
'(?P<word>\\S+)'
```

```
>>> tokenisers.tokenise("Most languages make use of white space as the
basic device for finding distinct units in written text.", tokenisePat-
tern=tokenisers.SIMPLESPACES)
```

```
['Most', 'languages', 'make', 'use', 'of', 'white', 'space', 'as', 'the',
'basic', 'device', 'for', 'finding', 'distinct', 'units', 'in', 'writ-
ten', 'text.']
```

That wasn't too bad, except that we failed to split the full stop at the end off the word text. We could try changing the definition of SIMPLESPACES to be (?P<word>\.|\S+), i.e. to match either a full stop (we need the backslash here, because the dot by itself will match anything) or a sequence of non-white space characters. If we do that we find that we still get

```
['Most', 'languages', 'make', 'use', 'of', 'white', 'space', 'as', 'the',
'basic', 'device', 'for', 'finding', 'distinct', 'units', 'in', 'writ-
ten', 'text.']
```

because the full stop is itself a non-white space character, so it gets picked up by the second option anyway. So we need to make use of a negative lookahead pattern to say that when we are looking for our non-white space characters we do not want to be at a point where the next character is actually a full stop:

```
SIMPLESPACES1 = re.compile(r"""(?P<word>\.|((?!\.)\S)+)""")
```

```
>>> tokenisers.tokenise("Most languages make use of white space as the basic device for finding distinct units in written text.", tokenisePattern=tokenisers.SIMPLESPACES1)
```

```
['Most', 'languages', 'make', 'use', 'of', 'white', 'space', 'as', 'the', 'basic', 'device', 'for', 'finding', 'distinct', 'units', 'in', 'written', 'text', '.']
```

But the full stop at the end is just the beginning of the complications: other punctuation marks, times, numbers, currency markers, abbreviations, … The regex we actually use for English is (we use the extended range of characters to allow us to deal with other European languages that make use of accents):

```
chars = "A-Za-âzéêèëÇçÀÂÔôàûùîÊï"

ENGLISHPATTERN = re.compile(r"""(?P<word>(\d+,?)+((\.|:)\d+)?K?|(Mr|Mrs|Dr|Prof|St|Rd)\.|n?'t|([%s](?!n't))*[%s]|\.|\?|,|\$|£|&|:|!|"|-|\S)"""%(chars, chars))

>>> tokenisers.tokenise("It didn't cost Mr. Jones more than £123,446.99!", tokenisePattern=tokenisers.ENGLISHPATTERN)
```

```
['It', 'did', "n't", 'cost', 'Mr.', 'Jones', 'more', 'than', '£', '123,446.99', '!']
```

This kind of regex-based approach is extremely fast (slightly over 500K words/second on a standard MacBook), and can be adapted to other languages (we provide example patterns for other languages; it can be a bit tricky specifying the range of basic characters for languages that don't use the Roman alphabet – we have provided ranges for Arabic ([ء-ٛ]) and Chinese [一-龥], but for other character sets you'll have to find the start and end characters according to the Unicode order, which isn't always what you'd expect).

Stemming

But finding units in this way isn't really going to do what we want. It would be really annoying if our emotion analysis program recognised that the word *love* was associated with the emotion **love** but failed to use this when trying to classify a tweet that contained the word loves (or *depress* and *depressed*, or **joy** and **joys**, or …).

The first step in trying to deal with this involves finding a set of standard affixes (prefixes (which come before the root of the word) and suffixes (which come after it)) and just removing these. We start with the following sets:

```
PREFIXES = {"", "un", "dis", "re"}

SUFFIXES = {"", "ing", "s", "ed", "en", "er", "est", "ly", "ion"}
```

Note that we include the empty string as a prefix and as a suffix – this makes things more uniform, both in terms of the algorithms for detecting affixes and in terms of what we do with them once we have found them.

We can just strip off any affixes and prefixes that we find and assume that what's left is the root of the word. Note that we have written this as a generator, because it gives us more flexibility when collecting multiple solutions, so we have to explicitly make a list if we want to see all the solutions together.

```
def stem0(form, prefixes=PREFIXES, suffixes=SUFFIXES):
    for prefix in prefixes:
        if form.startswith(prefix):
```

```
            form1 = form[len(prefix):]
            for suffix in suffixes:
                if form1.endswith(suffix):
                    if suffix == "":
                        yield (prefix, form1, "")
                    else:
                        yield (prefix, form1[:-len(suffix)], suffix)
```

This will generate several forms for a given word, depending on whether we have empty prefixes and suffixes, but it is reasonable to assume that the form with the shortest root is the one that is closest to getting to the heart of the word:

```
>>> from chapter4 import stem1
>>> list(stem1.stem0("walked"))
[('', 'walked', ''), ('', 'walk', 'ed')]
>>> list(stem1.stem0("walking"))
[('', 'walking', ''), ('', 'walk', 'ing')]
```

Both of these have walked as the shortest root, which is what we want. This will also work with cases with prefixes and suffixes:

```
>>> list(stem1.stem0("unexpected"))
[('', 'unexpected', ''), ('', 'unexpect', 'ed'), ('un', 'expected', ''),
('un', 'expect', 'ed')]
```

Clearly un-expect-ed is the version which is most likely to contain the actual root.

But this will also do some quite stupid things.

```
>>> list(stem1.stem0("spring"))
[('', 'spring', ''), ('', 'spr', 'ing')]
>>> list(stem1.stem0("under"))
[('', 'under', ''), ('', 'und', 'er'), ('un', 'der', ''), ('un', 'd',
'er')]
```

The problem here is that we haven't checked that the thing we found as the root is actually a root. So we need a list of words that could be roots, and then we can use that to check that what we're generating does have a sensible root.

Where can we get a list of words. There are all sorts of places you could look: we use the WordNet lexicon, as embodied in the NLTK:

```
from nltk.corpus import wordnet
def readAllWords():
    return set(wordnet.all_lemma_names())
```

That gets us a set of 147K forms, which is enough for our purposes. We then change the definition of stem0 to check that the root is in fact in this set:

```
def stem(form, prefixes=PREFIXES, allwords=ALLWORDS, suffixes=SUFFIXES):
```

```
    for i in range(len(form)):
        if form[:i] in prefixes:
            for j in range(i+1, len(form)+1):
                if form[i:j] in allwords:
                    if form[j:] in suffixes:
                        yield (form[:i], form[i:j],form[j:])
```

This works fairly well, though there are quite a lot of cases where there is an inflected form listed in the lexicon as well as the root, in which case we get multiple analyses. Still, looking for the one with the shortest root does generally give the right answer:

```
>>> list(stem1.stem("unexpected"))
```

```
[('', 'unexpected', ''), ('un', 'expect', 'ed'), ('un', 'expected', '')]
```

WordNet contains *unexpected*, expected and *expect* as lemmas, so we get all three as answers, but as before the one with **expect** as the root is clearly the most basic form. And since *spr* is not in the WordNet set, we don't get stupid analyses of *spring*:

```
>>> list(stem1.stem("spring"))
```

```
[('', 'spring', '')]
```

Spelling rules

The version of `stem` in `stem1`, then, provides a reasonable starting point, but it won't cope with forms where something changes at the junction betweem a word and an affix. To cope with this, we need to consider the notion of spelling rules.

For all languages, the spoken form precedes the written form. It often happens that it takes a while for the way that sounds of a language are captured in the way it is written to settle down (look at the various spellings that Shakespeare used for his own name!), but for most of the languages currently in wide use there is now a systematic relationship between the spoken and written forms.

This relationship may be systematic, but it isn't always straightforward. In particular, there may be conventions about how certain sounds are represented which change when an affix is added, even if the actual sound is unchanged. The most obvious case of this in English is for words like *save* and *hope*, where the final *e* says that the central vowel should be pronounced like /oh/ rather than /o/ or /oo/. This final *e* gets deleted when a suffix that starts with a vowel is added, but the pronunciation of the middle vowel is unchanged: *save+ing ==> saving*, *hope+ed ==> hoped*. Similarly, if a suffix that begins with vowel is added to a word that has a final stressed short vowel followed by a single consonant then the final consonant is doubled, with very little change in the pronunciation: *put+ing ==> putting*, *infer+ed ==> inferred*. Note that this rule does not apply if the final vowel is not stressed: *enter+ed ==> entered.*

There are also cases where adding an affix actually changes the pronounciation. If you add the suffix *s* to a word that ends with a hard consonant then the pronunciation is just the pronunciation of the original word with /s/ added: *put+s ==> puts*, *end+s ==> ends*. If you add it to a word that ends with *s, x, ch* the pronunciation changes, with a short vowel inserted in between the root and suffix because it is difficult to reconfigure your articulators (tongue, teeth, …) to get from one to the other: fox+s ==> foxes, catch+s ==> catches.

It is not always easy to tell whether a spelling rule is there to tell you how to pronounce something (inferring vs entering) or to reflect a change in the pronunciation (*puts, foxes*). Since we are working almost entirely with texts, the reasons for a spelling change don't matter all that much. What matters is that we can capture them.

The following set cover a lot of English cases. It is easiest to apply them if you also have a lexicon, because otherwise it can be difficult to tell whether something is a complete word – if you don't have a lexicon it isn't possible to tell that *foreseen* is *foresee+en* but *windscreen* isn't *windscree+en*. Having a lexicon is, however, a mixed blessing, because it is likely to include some derived forms as well as just the roots.

The rules below employ a number of conventions:

- the lefthand side of a rule matches part of a surface form, the righthand side says what the underlying form might look like, + marks the point where there is a break between two components of a word. Thus *sion ==> de + ion* is a rule which would match the last part of *decision* and suggest that the underlying form might be *decide+ion*.
- *C, C0, C1, …* are variables that represent consonants, *V, V0, V1, …* represent vowels and *X, X0, X1, …* represent arbitrary letters; multiple occurrences of the same name in a rule must all match the same letter. So *C ation ==> C e + ion* says that any form that ends with a consonant *C* followed by ation might have an underlying form like *Ce + ion*, e.g. that *excitation* might have the underlying form *excite+ion* (i.e. with the final *at* deleted).
- If we write a variable followed by a : and then some set of letters then the surface form should match the letters in the surface form and the variable should be bound to whatever was matched, so *C y X:ing ==> C ie + X* will match a consonant followed by y followed by ing and rewrite it as *C* was followed by ie followed by *X*, e.g. *dying ==> die+ing, tying ==> tie+ing*.
- We can use regular expressions to specify matches, e.g. *X1:(e(d|r|st)|ly|ness) ==> y + X1* says that if the surface form contains *ied* or *ier* or iest or *ily* or *iness* then the underlying form might be *y + ed (tried ==> try+ed)* or *y + er (happier ==> happy+er)* or *y + est (happiest ==> happy+est)* or *y + ly (happily)* or *y + ness (happiness).*

These rules are permissive: they say that a given surface form may correspond to some underlying form – that *heed* could be *hee+ed*, not that it must be that. That's what we need a lexicon for, to realise that *hee* isn't actually a root so the underlying form can't be *hee+ed*. Indeed, in some cases there is an underlying form that involves the rule and one that doesn't – *weed* could be the past form of the verb wee or the singular form of the noun *weed*.

```
C ation ==> C e + ion
ion ==> te + ion
sion ==> de + ion
C y X:ing ==> C ie + X
C X:ly ==> C le + X
C aid ==> C ay + ed
i X1:(e(d|r|st)|ly|ness) ==> y + X1
i e s ==> y + s
X:((d|g|t)o)|x|s(h|s)|ch es ==> X + s
C0 rous ==> C0 e r + ous
C0 i C1 (?!(.*(a|e|i|o|u))) ==> C0 y + C1
X0 (?!(?P=X0)) C X1:e(d|n|r|st)|ing (?!(.*(e|i))) ==> X0 C e + X1
^ C V0 X1:e(d|n|r|st)|ing (?!(.*(e|i))) ==> C V0 e + X1
u X1:e(d|n|r|st)|ing ==> ue + X1
```

```
C0 V C1 C1 X:(e(d|n|r|st)|ing) ==> C0 V C1 + X
```

Given these rules, we can find the stems for words where there are boundary effects where word parts are joined and where there are multiple affixes:

```
>>> from chapter4 import stem2
>>> stem2.allstems("impossibly")[0]
'in-possible+ly'
>>> stem2.allstems("demonstrations")[0]
'demonstrate+ion+s'
>>> stem2.allstems("happier")[0]
'happy+er'
```

In the examples above we have picked the first analysis returned by stem2.allstems, i.e. the one with the shortest root. We don't always want to do that, because it is quite possible for there to be more than one legal analysis: *adder==> add+er* (“digital circuit that performs addition of numbers”), *adder ==> adder* (“species of venomous snake”), *weed ==> weed* (“a plant considered undesirable in a particular situation”), *weed ==> wee+ed* (“urinated”). We therefore have to make a decision: do we always take the first analysis, or do we have to consider all the analyses we get?

In some cases we get multiple entries because the lexicon itself contains (some of) the derived forms from a given root. In general the first analysis will be the one that gets us right down to the root, in which case it probably will be the one we want.

```
>>> stem2.allstems("aesthetically")
['aesthete+ic+al+ly', 'aesthetic+al+ly', 'aesthetical+ly', 'aestheti-
cally']
>>> stem2.allstems("unreconstructed")
['un-re-construct+ed', 'un-reconstruct+ed', 'un-reconstructed', 'unrecon-
structed']
```

But sometimes we get completely different roots:

```
>>> stem2.allstems("adders")
['add+er+s', 'adder+s']
>>> stem2.allstems("weed")
['wee+ed', 'weed']
```

In any situation where we want to deploy a stemmer we will have to make a decision about this. It is hard to see how you could write a program that knew that *wee-ed* and *weed* are different words but *state+ed* and *stated* are different forms of the same word, where *stated* just happens to be in the lexicon.

What to do? Cases like *weed* and *adders* are much (much!) less common than ones like *aesthetically*, so the obvious thing to do is to always take the first (most reduced) analysis. Once in a very long while you'll miss a case where the different analyses are actually different words, but most of the time you'll get the most reduced version of all the variations of a single word that are listed in the dictionary.

Morphology

`stem2` return analyses for combinations of morphemes that don't actually go together.

```
>>> stem2.allstems("screen")
['scree+en', 'screen']
>>> stem2.allstems("underevaluation")
['un-de-re-value+ion', 'un-de-re-valuation', 'un-de-revaluation', 'un-derevaluation']
```

scree is a word ("a mass of small loose stones that form or cover a slope on a mountain"), *en* is a suffix, so `stem2` thinks that *screen* might be *scree+en*; *un, de* and *re* are prefixes, *valuate* is a word, so `stem2` thinks that *underevaluation* might be *un-de-re-valuate+ion*.

To handle this, we need to know what affixes go with what kinds of root and what effect they have. There are two cases to consider:

- **inflectional** affixes just add information to an existing word – adding *-ing* to a verb says that the event denoted by the root is incomplete/ongoing, adding *-s* to a noun says that the noun depicts a set of several entities, adding *-est* to an adjective says that the most extreme form of the property denoted by the adjective applies. We can think of inflectional affixes as helping to complete the word they are attached to, e.g. we can say that an English verb needs a tense marker to complete itself, or that a French noun needs a gender marker and a number marker to complete itself. To make this work it is convenient to allow empty affixes, e.g. if we think of a verb as being incomplete without a tense marker then we may want to add an empty marker (denoting the infinitive and non-third-person present tense forms) to the visible affixes *-s, -ing, -ed, -en*. We can then assign the label v->tns (verb missing a tns marker to its right) to the root forms of English verbs, or (n->num)->gender (noun that wants to find a gender marker to its right and then a number marker also to its right: note the bracketing) to the root forms of French verbs.
- **derivational** affixes make a new word from an old one, sometimes changing its meaning (adding *un-* to an adjective negates it, adding *re-* to a verb means doing it again) and sometimes changing its part-of-speech – adding *-ment* changes a verb to a noun, adding *-ise* changes a noun to a verb. In this case we talk of the affix needing to find a suitable item to combine with – that *re-* is of type **(v->tns)->(v->tns)** (if you provided a root form of a verb to its right then it would make something of the same type) and *-ise* is of type **(v->tns)<-(n->num)** (if you gave it a root form of a noun to its left it would turn into a root form of a verb).

We therefore use a table to convert roots so that they specify the affixes they need:

```
ROOTS = {"v": "v->tns",
         "n": "n->num",
         "a": "a->degree",
         "r": "r"}
```

If we use this to help construct our lexicon, we find that the root forms have labels that say what they need to complete themselves – *detest* would be a verb if you gave it a tense marker, *desk* would be a noun if you gave it a number marker:

```
>>> from chapter4 import stem3
>>> stem3.ALLWORDS["detest"]
[('v', '->', 'tns')]
```

```
>>> stem3.ALLWORDS["desk"]
[('n', '->', 'num')]
```

We also need a collection of affixes:

```
PREFIXES = fixaffixes(
    {"un": "((a->degree)->tns)->(v->tns)",
     "re": "(v->tns)->(v->tns)",
     "dis": "(v->tns)->(v->tns)"})

SUFFIXES = fixaffixes(
    {# inflectional suffixes
     "": "tns; num; degree",
     "ing": "tns; (a<-(v->tns))",
     "ed": "tns; (a<-(v->tns))",
     "s": "tns; num",
     "en": "tns",
     "er": "degree",
     "est": "degree",

     # derivational suffixes
     "ly": "r<-a",
     "ic": "a<-(n->num)",
     "al": "a<-a",
     "er": "(n->num)<-(v->tns)",
     "ion": "(n->num)<-(v->tns)",
     "ment": "(n->num)<-(v->tns)",
     "ous": "a<-(n->num)",
     "less": "a<-(n->num)",
     "ness": "(n->num)<-(v->tns)",
     "able": "a<-(v->tns)",
     })
```

Looking at the inflectional suffixes we see that *-en* has been specified as a tense marker and *-est* as a degree marker. *-s* can be either a number marker (to go with a noun) or a tense marker (to go with a verb), so it gets a disjoint entry – `"s": "tns; num"`.

The derivational suffixes are described as things that need something to their **left** – if the suffix is in charge then we have to talk about what it needs to its left, e.g. *-ment* is something which will combine with an untensed form of a verb to produce a noun that needs a number marker (note that the version of `allstems` in `stem3` has an extra argument, `module`, which we use to specify where the roots and affixes are defined

and also how they are to be combined: this makes it easier to switch between versions, since we can continue to use the `stem3` implementation of allstems while getting the affixes, the dictionary and the machinery for combining words and affixes from different place):

```
>>> stem3.allstems("engagements", module=stem3)[0]
('engage+ment+s', ['n'])
>>> stem3.allstems("disengagements", module=stem3)[0]
('dis-engage+ment+s', ['n'])
```

Once we know what kinds of things can combine, we can eliminate analyses like the one above for *under-valuation*. In the examples below, stem2 and stem3 both (correctly) decompose *revaluation* using *re-* and *-ion* as affixes, but only stem2 decomposes underevaluation into three derivational prefixes plus the root and a derivational suffix:

```
>>> stem2.allstems("revaluation")[0]
're-valuate+ion'
>>> stem3.allstems("revaluation", module=stem3)[0]
('re-valuate+ion+', ['n'])
>>> stem2.allstems("underevaluation")[0]
'un-de-re-valuate+ion'
>>> stem3.allstems("underevaluation", module=stem3)[0]
[('underevaluation+', ['n'])
```

So using complex labels that say what can combine with what improves things. But remember that we said that inflectional affixes add information to the root, and the labels we have used above don't do anything about this.

We can extend our descriptions in `chapter4/stem3a` as below (just including the inflectional suffixes here for simplicity):

```
ROOTS = {"v": "v[tense=T, finite=F, number=N, person=P]
               ->tense[tense=T, finite=F, number=N, person=P]",
          "n": "n[number=N]->num[number=N]",
          "a": "a[comp=C]->cmp[comp=C]",
          "r": "r"}

SUFFIXES = fixaffixes(
    {"": "tense[finite=infinitive];
          tense[finite=tensed, tense=present];
          num[number=singular]; cmp[comp=base]",
     "ing": "tense[finite=participle, tense=present]",
```

```
    "ed": "tense[finite=participle, tense=present, voice=passive];
           tense[tense=past, voice=active]",
    "s": "tense[finite=tensed, tense=present, number=singular,
                                             person=third];
          num[number=plural]",
    "en": "tense[finite=participle]",
    "est": "cmp[comp=superlative]",
...})
```

The roots now say what information they will borrow from the inflectional affixes, e.g. nouns will now borrow the value for person from the person affix, and verbs will borrow the values for tense, finiteness, number and person; and the suffixes now supply values for these features, e.g. *-ing* marks a verbs as being an active present participle and *-ed* marks verbs as being either past tense forms or passive present participles.

Given these more complex labels, we can find out much more about what an instance of a word is like, and we can also use them to rule out cases involving items with inappropriate combinations of features. For this to work we have to employ the notion of **unification** – partial matching where variables are used to say "match this with that, and remember what you have done": `stem3a` contains the revised roots and affixes, and also provides a function for combining roots and affixes which uses unification to manage the process of borrowing information about labels: using stem3, we can see that watches could be either a noun or a verb; using stem3a we see it could be a plural noun or singular present tense verb:

```
>>> from chapter4 import stem3a
>>> printall(stem3.allstems("watches", module=stem3))
('watch+s', ['n', 'v'])

>>> printall(stem3.allstems("watches", module=stem3a))
('watch+s', [{'hd': 'n', 'number': 'plural'},
             {'hd': 'v', 'tense': 'present', 'finite': 'tensed',
               'number': 'singular', 'person': 'third'}])
```

The main lesson is as in Chapter 4 of the book:

> The lesson is clear: if you want words stripped right down to their roots, you will have to provide a substantial amount of clear information about word classes and about the effects that the various affixes have. If you take a simple-minded approach and are not too worried about getting right to the heart of each form, and about finding out its exact properties, then you can do the task substantially faster, but even at 14.7K words/second morphological analysis not going to be a major bottleneck.

Pointwise Mutual Infomation

So words can be decomposed into smaller parts, and doing this can be a useful step in language processing tasks. It is generally worth knowing that *love, loves, loving, loved, lovely, lover* and *lovers* are all based on the root *love*; spotting that *loved* and *unloved* have a common root may be less useful, since they have almost diametrically opposite meanings, so it is sensible to restrict attention to affixes that are actually useful for the current task, whatever that may be. Nonethless, decomposing surface forms into a root and a set of affixes is generally a useful thing to do.

But at the same time, it sometimes happens that pairs (or larger groups) of words have meanings that cannot be derived from the meanings of the words themselves (recall the examples heavyweight and understudy from Chapter 4). As noted in Chapter 4, this can be a bigger issue in some other languages (e.g. Mandarin) than it is in English, but for ease of presentation we will concentrate on English here.

The approach we suggested in Chapter 4 was to use **pointwise mutual information** (PMI), looking for pairs of words that occur together more often than they should. The PMI programs mentioned in the book can be run as below. They only work well if you have a large amount of data – finding out how whether a pair of words occur together more often than they ought to will only work if they do occur together fairly frequently, and to find lots of pairs of words that co-occur frequently you need quite a bit of data. We therefore start by loading a corpus: we will use the BNC (remember to install the BNC as described in *datasets.docx*) , but if you haven't got that, or you are working with another language, you'll need your own corpus. Note that the reader returns a generator, which we consolidate into a list for convenience (`listN` makes a list out of the first `N` items returned by the generator: in the examples below we use the first 10 million words from the BNC – the results will be slightly different if you use larger subsets, but everything will take longer); and that we supply a pattern which the names of leaf (data) files should match, because the BNC contains other material that we do not want to use (READ.ME files and suchlike)

```
>>> from chapter4 import compounds
>>> from basics import readers
>>> # get 100000000 words from the BNC (this will take while)
>>> bnc = listN(readers.reader(readers.BNC.PATH,
                               readers.BNCWordReader,
                               pattern=".*/[A-Z0-9]*\.xml"),
                100000000)
```

Now calculate the PMI. We ignore the commonest `t2` words, because they won't contribute useful pairs, and we require a pair to occur at least `t1` times before we include it. Then `pmi` is the list of pairs with PMI scores, sorted in decreasing order; `pmiTable` is the table of all pairs and their scores; `words` and `pairs` are counts of individual words and pairs.

```
>>> pmi, pmiTable, words, pairs= compounds.doItAllPMI(bnc)
```

```
10000000 words
get rid of the top 100 words because they will make too many pairs (e.g.
'of-the')
TOP SCORERS: {'now', 'is', 'not', 'were', 'up', 'like', 'He', '’',
'said', '(', 'out', 'them', 'two', 'an', 'we', 'you', 'on', 'can', 'did',
'all', "n't", '—', 'of', 'than', ',', 'but', 'do', 'But', 'had', 'over',
'at', 'him', 'that', 'been', 'it', 'would', 'about', 'then', 'there',
'will', 'she', 'other', 'into', 'this', 'by', 'new', '.', 'who', 'In',
'so', 'their', ')', 'most', 'people', 'also', '?', 'which', 'It', 'no',
```

```
'', 'with', 'he', 'his', 'if', "'s", 'when', 'after', 'first', 'what',
'its', 'some', 'be', 'I', 'have', 'more', 'they', 'for', 'the', 'was',
'me', ';', 'her', 'from', 'only', 'as', 'my', 'one', 'are', 'time', ':',
'a', 'and', 'Mr', 'or', 'The', 'could', 'in', 'has', 'A', 'to'}

185499 distinct words found (10000000 tokens)

Getting pairs that occur at least 50 times

1539 pairs found

Calculating PMI
```

```
>>> printall("%s %.1f %s"%(y, x, z) for x, y, z in pmi[:20])
```

```
gon-na 15.7 55

vice-versa 15.6 53

et-al 15.3 53

carbon-dioxide 14.7 140

proportional-representation 14.4 97

du-n 14.4 118

tens-thousands 13.7 66

status-quo 13.7 59

salt-pepper 13.5 56

swimming-pool 13.4 59

olive-oil 13.2 94

reverse-punch 13.2 63

en-route 13.1 50

judicial-review 13.1 50

civil-servants 13.0 156

civil-aviation 13.0 51

pas-de 12.8 57

prime-minister 12.8 331

managing-director 12.6 167

human-beings 12.6 83
```

We get a mixture of things – frozen pairs (either foreign language pairs – *et al, status quo* – or English pairs – *gonna* is in fact the two words *going to* pronounced as just one word meaning *will*), pairs where the meaning of the whole is derivable from the meanings of the parts (*olive oil, carbon dioxide*) and pairs where the meaning of the whole is not 100% obvious from the meanings of the parts (*swimming* makes a different contribution in *swimming pool, swimming lesson, swimming bird*) or is indeed almost unrelated to the meanings of the parts (*civil servants* are not servants who are very polite, and *civil engineering* is not a very polite form of engineering).

Greenhouse-gases, crime-prevention and grass-roots all appear in the top 3% of PMI scores, and are all likely to carry emotional baggage that the components don't (we had to calculate the PMI tables for the en-

tire BNC to get these scores, because these words don't show up frequently enough to be included unless we have all the data from the BNC: calculating PMI can require quite a lot of data):

```
>>> pmiTable["greenhouse-gases"]
(12.322885857554724, 120)
>>> pmiTable["crime-prevention"]
(10.540598239864938, 202)
>>> pmiTable["grass-roots"]
(9.962986958665278, 141)
>>> for i, x in enumerate(pmi):
     if x[1] in ["grass-roots", "greenhouse-gases", "crime-prevention"]:
print("%.3f %s"%(i/len(pmi), x))

0.005 (12.322885857554724, 'greenhouse-gases', 120)
0.019 (10.540598239864938, 'crime-prevention', 202)
0.028 (9.962986958665278, 'grass-roots', 141)
```

We see that *greenhouse-gases* lies in the top 0.5% of PMI pairs, *crime-prevention* in the top 2% and *grass-roots* in the top 3%. It looks as though compounds whose meaning (and emotional weight) are different from the meanings and weights of the individual words do have very high PMI scores, but for English there will also be quite a lot of cases where the compound carries very much the same emotional weight of one of the words, so it is not clear just how useful this will be for English. The situation may well be different for other languages, notably Asian languages where compounding is either common (e.g. Malay) or almost universal (e.g. Mandarin), so it is worth looking to see what happens with your data if you are working with such a language.

Tagging

The meaning of a text isn't determined just by the words it contains. The way they are organised is also extremely important:

1. John loves Mary
2. Mary loves John
3. I did know John did not do it
4. I did not know John did it

(1) and (2) contain the same words, but they mean completely different things. Likewise (3) and (4) (indeed (3) and (4) have more or less opposite meanings – it is clear from (3) that John didn't do it, whereas it is a reasonable supposition from (4) that he did). This is important for many tasks, but it may be less useful for our goal of finding out what emotion(s) some text expresses, if only because informal language tends to break the usual conventions and hence it is all but impossible to actually find the organisation of such texts. Still, rather than just dismiss this as an issue we should look at what happens when we try to assign a structure to the kind of text that we are interested in.

This is usually done in two steps: assign a **part-of-speech tag** (POS tag) to each word in the text and then use this totry to find the organisation. There are tricks that can be used to merge these two tasks, delaying

making a decision about the POS tags until we have more information about the way the text as a whole is organised, but for the purposes of this chapter we will treat them as two pipe-lined tasks – find the POS tags, then find the structure.

We will look at three taggers, and we will train and test them on a range of datasets.

Word frequency tagger (BASETAGGER): the simplest thing to do is to count how often each word is assigned a given tag and use the most frequent one in every case. We also collect the frequencies of two and three letter prefixes and suffixes to be used when we get a word that is not in the training data, e.g. we can guess that *discombobulating* is a verb on the grounds that *-ing* is probably a verb ending.

HMM-based tagger (HMMTAGGER): this will inevitably make mistakes. The first occurrence of *love* in *I love my love with all my heart* is a verb, the second is a noun. The word frequency tagger cannot assign different tags to these two occurrences, so it must get one wrong (the actual frequencies in the UDT are {'PROPN': 0.15, 'VERB': 0.58, 'NOUN': 0.26, so it will assign them both the label VERB. Interestingly the proportions in the BNC are {'NN': 0.66, 'VV': 0.34}, so a tagger based on the BNC would assign them both as NNs. You need to be sure that the texts you are working with are similar to the ones you train on). To deal with this, you have to look at the surrounding context to realise that *I* is most likely followed by a verb and *my* is most likely followed by a noun. There are various ways of doing this: we use a hidden Markov model (HMM), but unlike uses of HMMs in some other contexts (e.g. for speech recognition) we can estimate the transition probabilities directly rather than having to go through a complex series of iterative approximations.

NLTK standard tagger (NLTKTAGGER): the NLTK supplies a standard tagger for English. Exactly what this tagger is depends on which release of the NLTK you have, but whatever is currently used will be pretty good. It does not, however, provide taggers for every language, so it is worth knowing how to train your own even if the NLTK English one is indeed the best.

We start by training a `BASETAGGER` and an `HMMTAGGER` on the entire collection of English datasets in the UDT (remember to install the BNC and UDT as described in *datasets.docx*) and on a 1M word subset of the BNC (tagging doesn't tend to need as much data as things to do with meanings, e.g. PMI) (note that we are using `BNCTaggedWordReader` rather than the `BNCWordReader` that we used for the PMI exercises):

```
>>> from basics import readers, datasets

>>> from chapter4 import taggers

>>> bnctagged1m = listN(readers.reader(readers.BNC.PATH, readers.BNC-
TaggedWordReader, pattern=".*\.xml"), 1000000)

>>> udttagged = list(readers.reader(readers.UDT.PATH, readers.UDTTagged-
WordReader, pattern=".*English.*wholething"))

>>> bncBASETAGGER = taggers.BASETAGGER(bnctagged1m)

>>> bncHMMTAGGER = taggers.HMMTAGGER(bnctagged1m)

>>> udtBASETAGGER = taggers.BASETAGGER(udttagged, taglength=10)

>>> udtHMMTAGGER = taggers.HMMTAGGER(udttagged, taglength=10)

>>> nltkTAGGER = taggers.NLTKTAGGER()

>>> alltaggers = [bncBASETAGGER, bncHMMTAGGER, udtBASETAGGER, udtHMMTAG-
GER, nltkTAGGER]

>>> printall(taggers.applyAllTaggers("I love my love with all my heart",
alltaggers))
```

```
             BASE (BNC)  HMM (BNC)   BASE (UDT)  HMM (UDT)   NLTK

I            PN          PN          PRON        PRON        PRP
```

love	NN	VV	VERB	VERB	VBP
my	DP	DP	PRON	PRON	PRP$
love	NN	NN	VERB	VERB	NN
with	PR	PR	ADP	ADP	IN
all	DT	DT	DET	DET	DT
my	DP	DP	PRON	PRON	PRP$
heart	NN	NN	NOUN	NOUN	NN

There are two things to note about this table:

- the taggers all use different tag sets – the taggers trained on the BNC sys that *I* is a PN, the ones trained on the UDT say it's a PRON, the NLTK tagger says it's a PRP. In this case these are just different names for the same things, but we will see later cases where one tagger provides a more fine-grained set of tags than another, and that might be something that concerns us. Note that these differences arise because of the tags used when the datasets were created – they're nothing to do with the algorithms, they come from the datasets.
- The base tagger trained on the BNC and both the taggers trained on the UDT assign the same tags to both instances of love – the base BNC tagger says they're both nouns, both the UDT taggers say they're both verbs. Again, this is a consequence of the distribution of tags in the two datasets, but it is significant that even the HMM-based version of the UDT dataset fails to distinguish them.

There are a number of words that are particularly difficult to assign tags to. The word *to* in *I went to London to see the queen* has two uses – firstly as a perfectly ordinary preposition saying where I went and second as some kind of conjunction/complementiser/particle linking *see the queen* to the rest of the sentence. Here's what the taggers do with it.

```
>>> printall(taggers.applyAllTaggers("I went to London to see the queen", alltaggers))
```

	BASE (BNC)	HMM (BNC)	BASE (UDT)	HMM (UDT)	NLTK
I	PN	PN	PRON	PRON	PRP
went	VV	VV	VERB	VERB	VBD
to	TO	PR	ADP	ADP	TO
London	NP	NP	PROPN	PROPN	NNP
to	TO	TO	ADP	PART	TO
see	VV	VV	VERB	VERB	VB
the	AT	AT	DET	DET	DT
queen	NN	NN	NOUN	NOUN	NN

The HMM models trained on the BNC and on the UDT both give *to* a different tag in the two positions, which is what we want. The NLTK just gives up and says its tag should be TO. Which of these is the better approach? If a tagger can reliably distinguish between the two then it is obviously better to be given the fine-grained tag, but given that this is a difficult distinction to make, maybe it's better to just tag it as TO (after all, that's never going to be wrong!) and leave it to the next stage of processing to do what it can with this tag. This kind of **underspecification** allows us to pass the responsibility for making a decision to a higher level where there may be more information available.

Similarly, the word *that* has four different roles in *I know that that man is the one that did that*, so a tagger that can assign these roles correctly would be more use than one that can't (it is worth noting that the French translation of this actually has four distinct words playing the roles that *that* is playing in this sentence: *je sais que cet homme est celui qui a fait ça*). If we just look at the HMM-based taggers and the NLTK tagger (because the base ones can only assign a single tag, so they must fail to deal properly with these instances of that) we get

```
>>> taggers3 = [bncHMMTAGGER, udtHMMTAGGER, nltkTAGGER]

>>> printall(taggers.applyAllTaggers("I know that that man is the one
that did that", taggers3))
```

```
          HMM (BNC)       HMM (UDT)       NLTKTAGGER
I         PN              PRON            PRP
know      VV              VERB            VBP
that      CJ              SCONJ           IN
that      DT              PRON            DT
man       NN              NOUN            NN
is        VB              AUX             VBZ
the       AT              DET             DT
one       CR              NOUN            NN
that      CJ              PRON            WDT
did       VD              AUX             VBD
that      DT              PRON            IN
```

The two HMM-based taggers do something reasonable with the first occurrence, saying that it's an item that links *know* and *that man is the one that did that*, and NLTKTAGGER sort of does that by saying that it's a preposition (**IN** is the tag that this tagger uses for preposition), though it's obviously not an ordinary preposition like *in* or *on*. The BNC-HMM tagger and the NLTK correctly say that the second occurrence is as a determiner (a word like *a* or *the* that introduces a noun phrase), but the UDT-HMM tagger gets this one wrong The NLTK then correctly marks the third occurrence as some kind of relative pronoun, and the UDT-HMM tagger correctly marks the fourth occurrence as a simple pronoun. This is clearly a hard task – the BNC-HMM tagger gets it right twice out of four times, the UDT-HMM tagger gets it right twice, and the NLTK tagger gets it right twice and sort of right once. It would almost certainly be better just to give it the tag **THAT** (which will never be wrong) and leave it to the next stage of processing to sort out what to do with it.

If we train a French tagger then we do get the following:

```
>>> udtfrenchtagged = list(readers.reader(readers.UDT.PATH, readers.UDT-
TaggedWordReader, pattern=".*French.*wholething"))

>>> len(udtfrenchtagged)
```

```
854308
```

```
>>> udtfrtagger = taggers.HMMTAGGER(udtfrenchtagged, taglength=10)

>>> s = """je sais que cet homme est celui qui a fait ça"""

>>> printall("\t".join(r) for r in udtfrtagger(s))
```

```
je    PRON
```

```
sais  VERB
que   SCONJ
cet   DET
homme NOUN
est   AUX
celui PRON
qui   PRON
a     AUX
fait  VERB
ça    PRON
```

This time we get three cases, even though the translation actually has four different words, because both *qui* and *ça* are tagged as simple pronouns, but qui should be tagged as a relative pronoun.

The lesson, as ever, is that you should never believe what anyone tells you about a tagger: **try it out for yourself, check its correctness for yourself, check that what it produces is suitable for your goals** (I never get sick of this lesson!).

Parsing

Tagging is almost never an end in itself. Who really cares whether that is a determiner or a pronoun or complementiser or a relative pronoun? But it is a crucial precursor to parsing, which is a crucial step on the way to understanding. If you can't tell that *I went to see the queen* and *I want to see the queen* have different structures then you won't be able to tell that the first of these is the answer to *why did you go?* and the second is the answer to *what do you want?*

In order to work out whether parsing is going to be useful for our task, we will look at what two well-known parsers , namely the Stanford Dependency Parser (Manning et al., 2014) and MALT Parser (Nivre et al., 2006), do with various sentences. These are not the only parsers that there are out there, but they are fairly well-reputed and, with a bit of fiddling around, they can be imported into the NLTK.

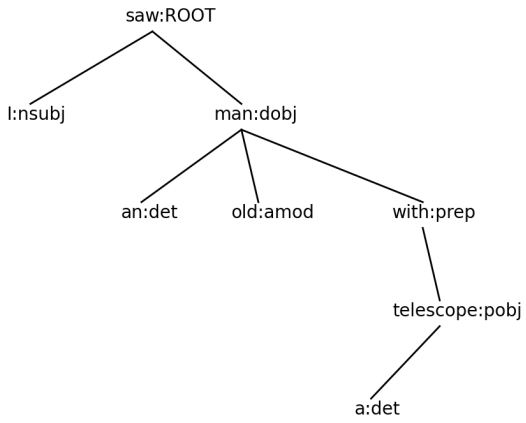
As with nearly all modern parsers, the SDP and MALT have to be trained on a treebank, and the quality of the output depends to a very large extent on the quality of the treebank. In particular, if the treebank does not have very similar properties to the data that you are going to apply the parser to then it is unlikely that it will cope very well (to consider an extreme version of this, if you trained a parser on an English treebank you would hardly expect it to perform very well on French data. The situation we will find ourselves in is not as bad as that, but it will turn out not be great). We will use the NLTK wrappers for these parsers, and we will use pre-trained models that can be obtained from the sites where the parsers themselves are kept. You will need to download the required NLTK packages and copies of the parsers themselves and make sure everything is where it should be: the examples below require the following files to be installed inside the directory `chapter4/parsers` (in addition to the files that are already in there):

```
--|parsers--|
            |engmalt.linear-1.7.mco
            |
            |lib--|liblinear-1.8.jar
                  |
```

```
        |libsvm.jar
        |
        |log4j.jar
    |maltparser-1.9.2.jar
    |
    |stanford-corenlp-4.2.0-models-arabic.jar
    |
    |stanford-parser-4.2.0-javadoc.jar
    |
    |stanford-parser-4.2.0-models.jar
    |
    |stanford-parser-4.2.0-sources.jar
    |
    |stanford-parser.jar
```

These files can be obtained from https://www.maltparser.org/ and https://nlp.stanford.edu/software/lex-parser.shtml#Download. They both have fairly non-restrictive licences, but you should check the licence details before downloading them.

Dependency parsers produce (obviously enough) dependency trees. These can be drawn as below (some of the layout of the trees we draw is a bit flaky – we use `matplotlib` as the basic drawing tool, but it's difficult to calculate exactly how much space a string will take when we place it using `matplotlib`, which can make laying out a tree a bit tricky):



I saw an old man with a telescope

This picturesis fairly self-explanatory: *I* is the **nsubj** of *saw*, *man* is the **direct object** of *saw*, *old* is an **amod** of *man*, … Programs are not, however, very good at handling nice drawings of trees, and it is common practice to use CONLL format for representing them. The following is the 5-column CONLL representation of the same tree:

```
1       I       PRP     2       nsubj
2       saw     VBD     0       null
3       an      DT      5       det
4       old     JJ      5       amod
```

```
5     man   NN    2     dobj
6     with  IN    5     prep
7     a     DT    8     det
8     telescope   NN    6     pobj
```

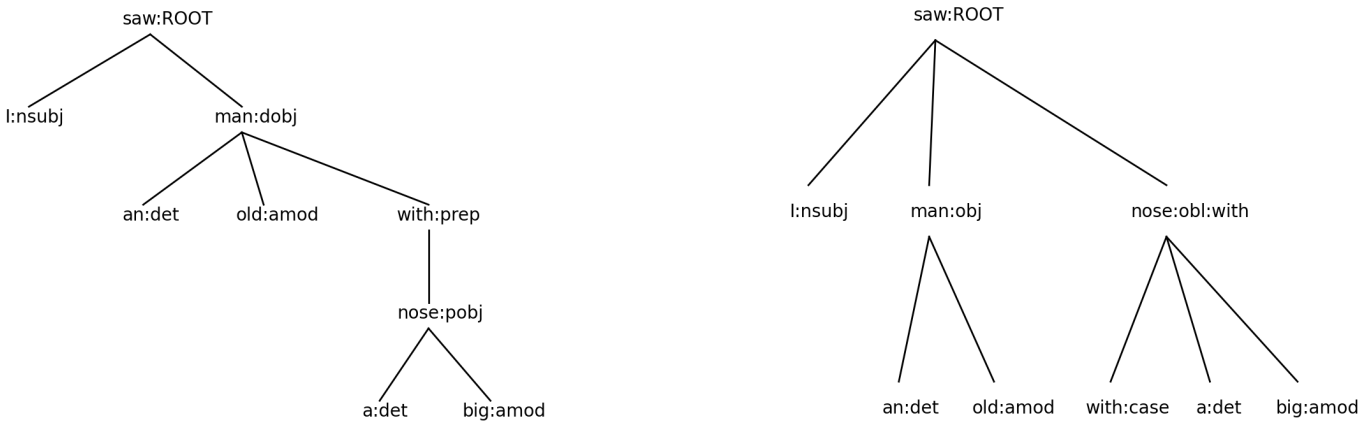
This says that word 1, *I,* is a pronoun which is the **nsubj** of word 2, and that word 2, *saw*, is a verb which has word 0 as its parent (i.e. it has no parent, it is the root of the tree), and so on. Both the parsers that we will be looking at produce something like CONLL format output: the code in chapter4/parsers/conll.py will use matplotlib to produce reasonable looking trees from CONLL format, so if you have a different parser that you want to try then this code will also draw its output as trees.

We will start by trying out our parsers on some fairly simple well-formed texts. In every case, we will draw the output of MALT followed by the output of SDP. The first two were obtained by going into `chapter4/parsers` and doing

```
>>> import malt, stanford, conll
>>> conll.plotconll(malt.maltparse("I saw an old man with a big nose"))
>>> conll.plotconll(stanford.sdpparse("I saw an old man with a big nose"))
```

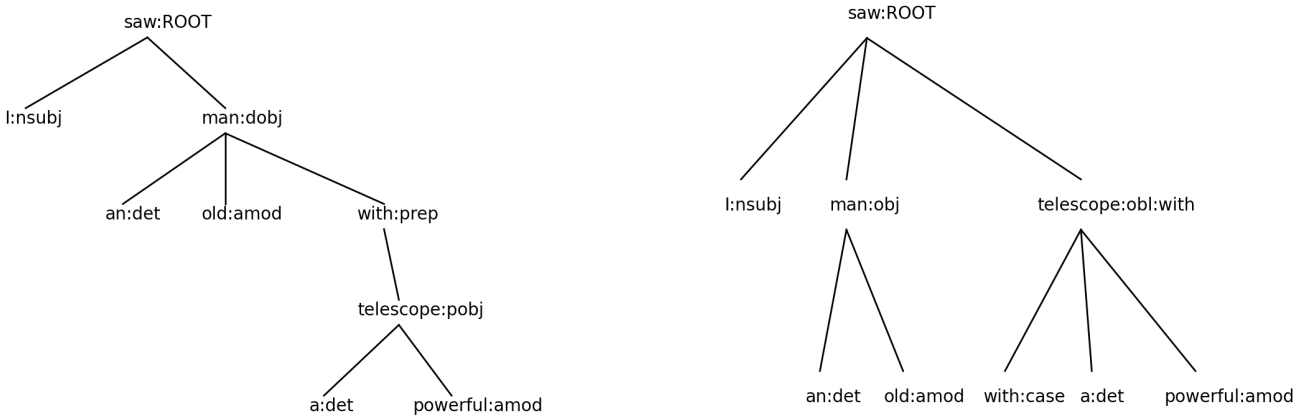
You have to do this from inside `chapter4/parsers` because otherwise it's very difficult to tell the parsers to look for `.jar` files and the models.

Getting the right analysis of this sentence involves working out the role of *with a big nose*. The trees that the two parsers gave are below:



I saw an old man with a big nose

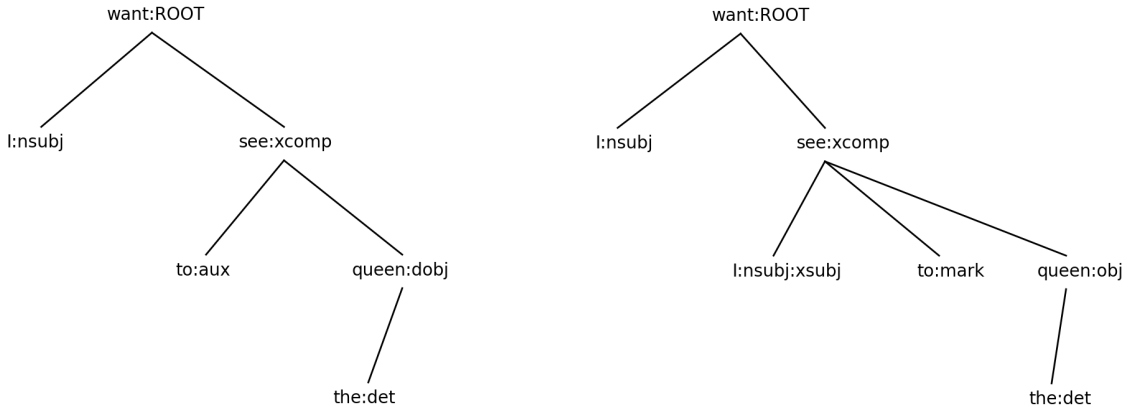
MALT (on the left) says that what I saw was an old man with a big nose, SDP says that I used a big nose to see an old man. The MALT parse looks sensible, the SDP one doesn't. If we replace *big nose* by *powerful telescope* we get the trees below.



I saw an old man with a powerful telescope

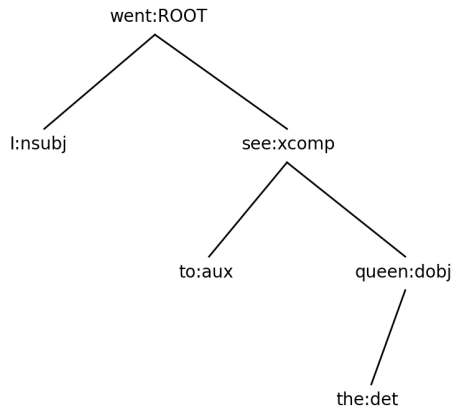
This time MALT says that what I saw was an old man who had a powerful telescope, SDP says that I used a powerful telescope to see an old man. Deciding what to do with modifiers like *with a big nose* or *with a powerful telescope* is difficult, and most parsers will frequently get it wrong.Still, both parsers do say that I saw an old man for both sentences, so maybe getting the modifiers a bit wrong wouldn't undermine our attempts to assign emotions to whole sentences.

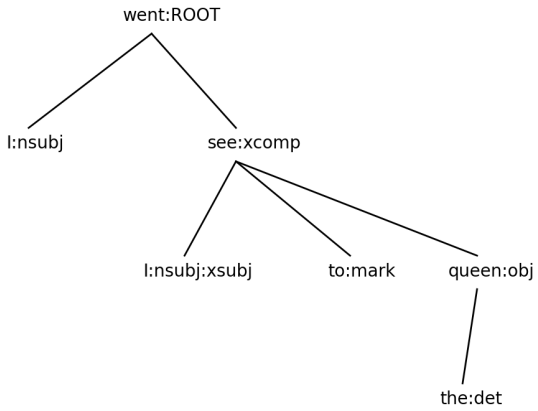
Our next pair involve thinking about how a verb might relate to the other elements of a sentence. We start with *I want to see the queen*.



I want to see the queen

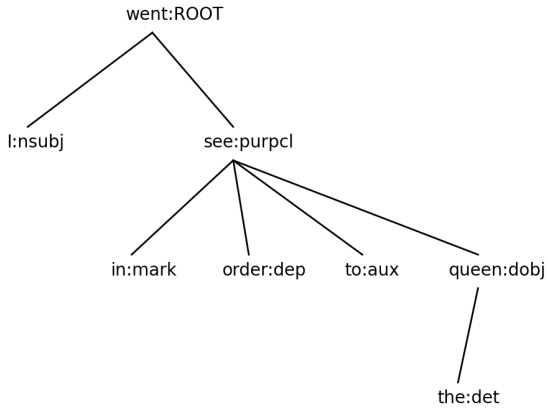
Both parsers say that I want something, namely to see the queen. SDP does something clever, because it realises that actually want I want is for me to see the queent, so it inserts a second copy of I as the subject of want. This is clever, and useful, but it does mean that SDP trees aren't actually dependency trees, since a single word can have more than one parent. Still, if it's useful then that's probably a good thing. Again, they both give exactly the sme analysis to *I went to see the queen*, which is obviously quite different:

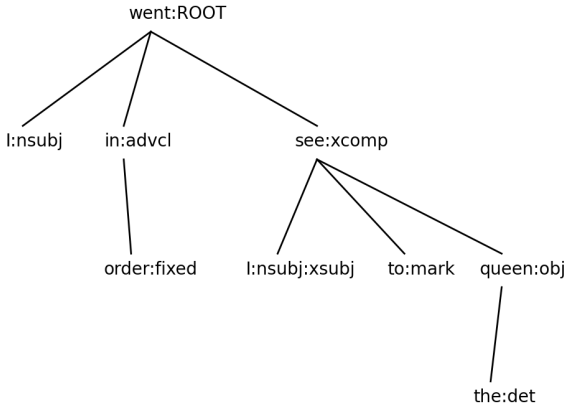




I went to see the queen

Both parsers assign the same structure to *I want to see the queen* and *I went to see the queen*. This is surely wrong --- I didn't went something, I just went: *to see the queen* is **why** I went, not **what** I went. We want something more like the MALT analysis of *I went in order to see the queen,* which says that *in order to see the queen* was the **purpose** of my going *(*but not the SDP one, which still says that *to see the queen* is what I did, but that I did it in order, whatever that might mean).

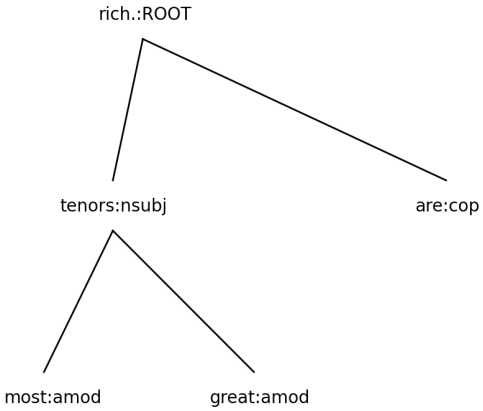


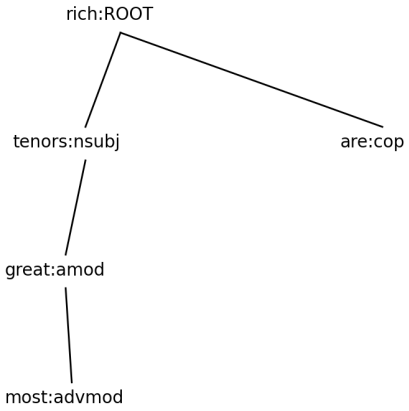


I went in order to see the queen

Will this matter for our task? It may well do, since knowing whether something is the aim of an action or is a part of it could well affect the emotional content of the whole text.

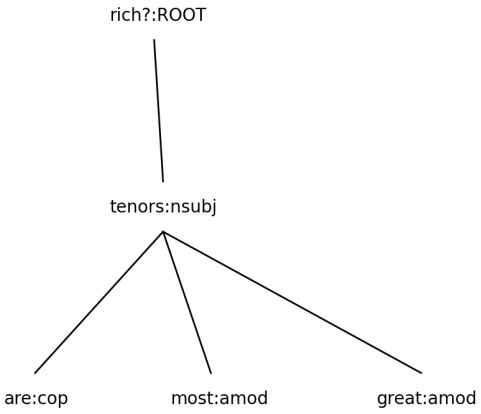
As a final example of the kinds of thing that can go wrong when you ask a parser to analyse a simple well-formed sentence we will consider what happens when you change a statement into a question: *most great tenors are rich.* vs *are most great tenors rich?*

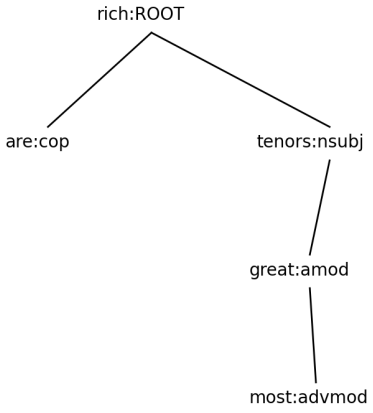




Most great tenors are rich

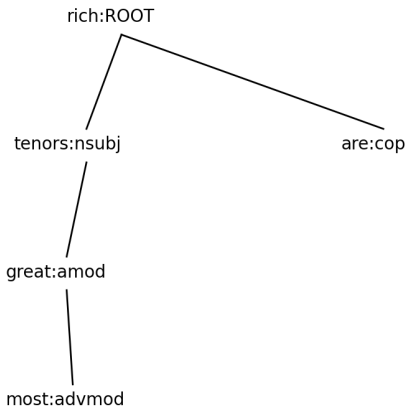
These analyses both treat *rich* as the head of the sentence and *are* as a daughter of *rich*, which is a bit odd, but is fairly standard with dependency parsers. MALT makes *most* and *great* daughters of *tenors*, which seems reasonable, whereas SDP treats *most* as a modifier of *great*, making *most great* equivalent to *great-est*, which doesn't seem right. Anyway, that's what they do with this sentence. We now look at *Are most great tenors rich?*



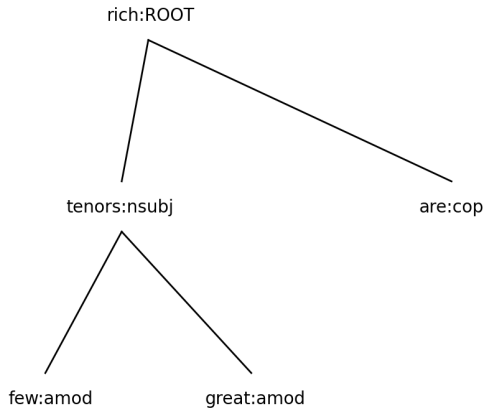


Are most great tenors rich?

The MALT tree for *are most great tenors rich?* is completely different from its tree for *most great tenors are rich.* That means that it be impossible to see that *most great tenors are rich* is the answer to *are most great tenors rich.* SDP produce basically the same tree for these two, but the fact that it produces different trees for *most great tenors are rich.* and *few great tenors are rich.* poses similar problems (both the trees below are SDP trees):



most great tenors are rich



few great tenors are rich

So parsers can make mistakes. Even so, most of the time the top level structures are good enough that we get a general picture – in the first set of examples we always get that I saw a man, even if we're not too sure about where I saw him; in the second set I was indeed involved in *going* and *wanting* events, even if we're not sure about the role of *to see the queen*; and the third group are all about great tenors being rich, though some of the other components get treated erratically and it is hard to see how the statement and question forms match up.

But when we come to deal with informal language the situation becomes significantly worse. There are three major source of problems:

- Informal language contains numerous misspellings, abbreviations, emojis, non-text items, …

  Im think ima lay in bed all day and sulk is tagged by MALT as **Im:NNP, think:VBP, ima:JJ, lay:NN, in:IN, bed:NN, all:DT, day:NN, and:CC, sulk:NN**. There's not much else it can do with *Im* and *ima*, though a human reader will guess that *Im* is actually *I'm* and *ima* is something like *I will*: but if these items don't occur in the training data (which they won't if the training data was well-formed text) then the tagger that MALT relies on won't be able to get them right.

  At home sick... 🎼The blues 🎼won't cure it so I need ideas 🎸😭 is tagged as **At:IN, home:NN, sick:NN, ...::, 🎼:VBZ, The:DT, blues:NNS, 🎼:VBP, won't:JJ, cure:NN, it:PRP, so:IN, I:PRP, need:VBP, ideas:NNS, 🎸😭:VB**: again the emojis 🎼, 🎸 and 😭 won't appear anywhere in trained set made up of normal well-formed text (and surprisingly the tokeniser and tagger used by MALT don't do anything sensible with *won't)*.

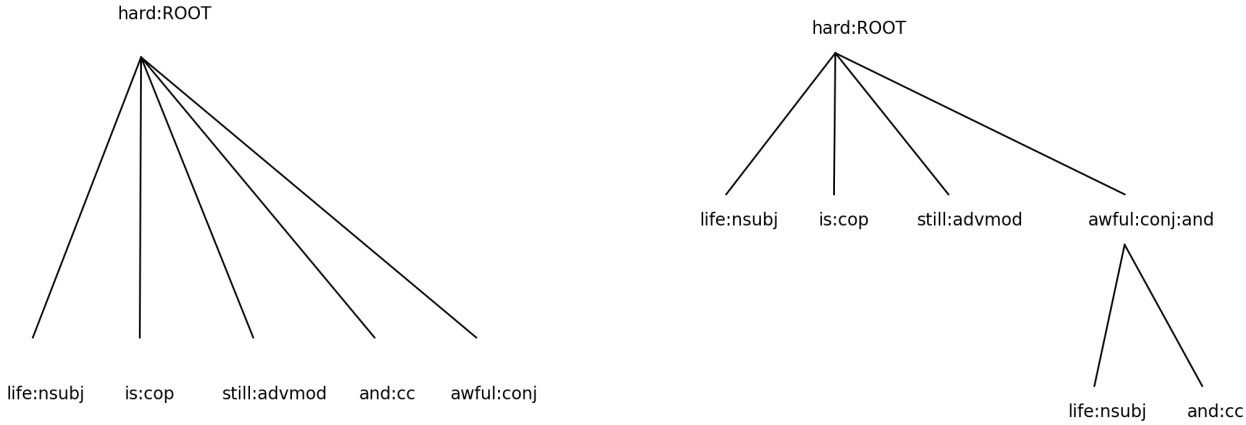
  No parser that has been trained on a standard vocabulary can do anything sensible with these. But if the text isn't tagged correctly then no parser can deal with it, even if it is actually well-formed.

- Because the parser will have been trained on well-formed text, it won't know what to do with text that does not obey the standard conventions. What on earth could a parser do with a tweet like *@ys really?? after so long that's shocking tbh* or *#start ur day wit a smile #buviobby*?

  This tweet just doesn't have a structure that fits the normal rules of English. But if a tweet doesn't have a sensible structure, it doesn't make sense to try to use a parser to find one for it. What would you want a parser to do with this tweet anyway?

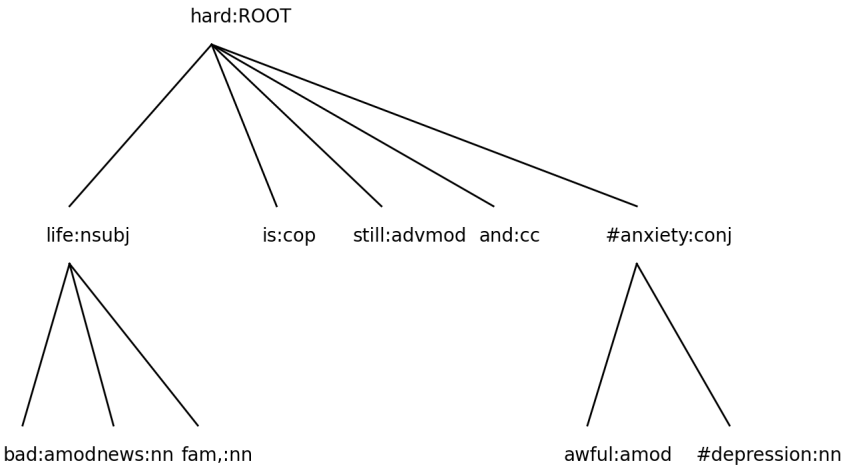
- Most dependency parsers are designed to be robust – to return a tree no matter what. But informal texts often contain well-formed fragments: the tweet *bad news fam, life is still hard and awful #depression #anxiety* is, again, not well-formed overall but it does contain *life is still hard and awful* as a well-formed fragment. It would be nice if your parser could spot this well-formed fragment and

simply mark the rest as being a bag of possibly emotion-bearing words (note again the abbreviation fam, which will cause almost any tagger problems). Our two parsers do handle this fragment sensibly if they are given it in isolation (well the SDP parse (on the right) is more useful than the MALT one, but on its own terms the MALT is entirely unhelpful):

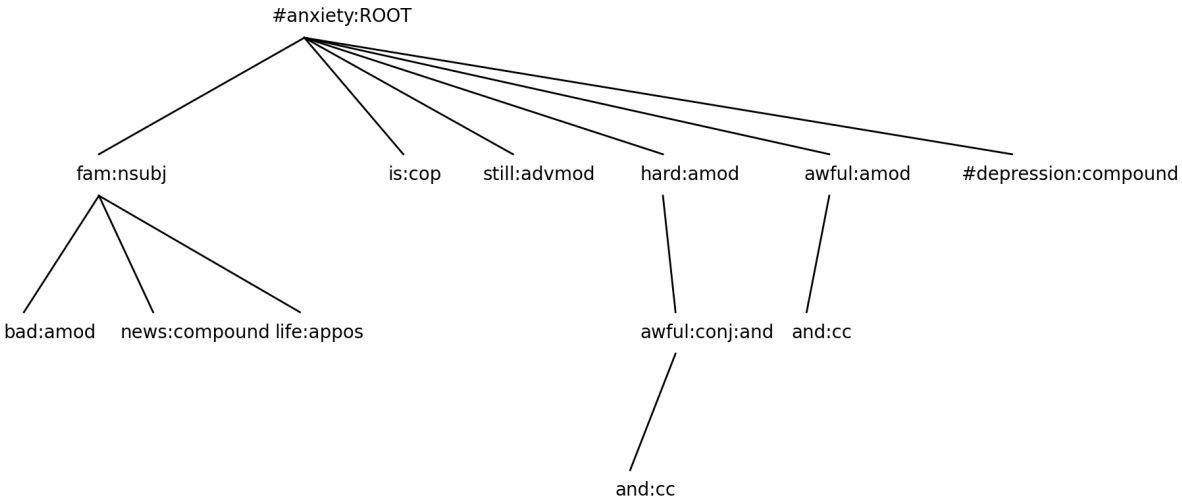


Life is still hard and awful (SDP parse)

But when we try them on the whole thing we get the following (SDP parse below MALT because they won't fit side-by-side):



bad news fam, life is still hard and awful #depression #anxiety (SDP Parse)



bad news fam, life is still hard and awful #depression #anxiety (SDP Parse)

> The overall trees don't make much sense, but then you wouldn't expect them to, given that the tweet doesn't fit any of the normal rules of English. But more significantly, **the relations between the words in the well-formed fragment are different when the fragment is treated in isolation**; and **there is no indication that this text is in any way anomalous**. So while the parsers are robust, in the sense that they will produce a tree for any input text, there is no way of telling whether the trees actually reflect the intended meaning. And without that they aren't much use.

So after all this work, the conclusion on tagging and parsing is that no orthodox approach to these tasks will produce anything useful for the texts that we are interested in. This is not just about the two specific parsers we have use for the examples. These two are pretty good, at least as far as freely available tagger:parser combinations are concerned, but they cannot deal with texts that do not fit the pattern of the texts they have been trained on, and it would be difficult to provide them with training data that does match the texts we are interested in, because they don't generally fit any pattern. This will be true for any data-driven tagger:parser combination. **Informal texts are not like well-formed texts, so you cannot use programs that have been trained on well-formed texts for them; and providing a training set made of informal texts will be impossible unless you have a clear theory of the structure that they follow.**