CHAPTER 4 EXERCISES

[Datasets 1](#__RefHeading___Toc20115_3197853673)

[Readers 4](#__RefHeading___Toc20117_3197853673)

[Word parts 5](#__RefHeading___Toc19778_3197853673)

[Spelling rules 5](#__RefHeading___Toc19780_3197853673)

[Morphology 8](#__RefHeading___Toc19782_3197853673)

[Pointwise Mutual Infomation 12](#__RefHeading___Toc19784_3197853673)

[Tagging 15](#__RefHeading___Toc20214_3197853673)

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.

To run the exercises and examples given here you need to start a copy of Python3 (we have tested them using Python 3.8.8: some of the programs in the book rely on libraries that may not work with this version of the language) running in the directory where you have downloaded the code.

Datasets

We use a number of datasets in this book. Some of them are freely available, some are freely downloadable but have licences or terms and conditions that restrict what you can do with them, some you have to pay for. We provide tools for unpacking most of the freely available datasets used in the book and putting them where they need to be, and for a few we provide tools for downloading them as well. Some of them have restrictions on what you can do with them, and most of them say that while we can write programs that do things using them we cannot directly redistribute them (but we can tell you where to download them from). You should always look at the licences before downloading them. We have tried to stick to corpora without restrictive licences where we can, and will always point to the licence conditions when we tell you where to get them from, **but it is up to you to obey the terms and conditions.**

The programs we will be using expect the corpora to be inside a directory called CORPORA. You can choose where that should be kept, but you should specify it inside the program file basics/corpora.py. I keep everything inside a folder called /Library/WebServer/CGI-Executables/SENTIMENTS, which is perhaps a slightly odd place to keep it but it works for me, so my basics/corpora.py contains the line

CORPORA = "/Library/WebServer/CGI-Executables/SENTIMENTS/CORPORA"

You need to set this to wherever you're going to want to keep the corpora.

To download and install the BNC, do the following.

Read the licence at <http://www.natcorp.ox.ac.uk/docs/licence.html>: you must obey the terms and conditions of this licence if you are going to use this resource. Then download it from

https://ota.bodleian.ox.ac.uk/repository/xmlui/bitstream/handle/20.500.12024/2554/2554.zip?sequence=3&isAllowed=y

and put the file 2554.zip that you get from there in CORPORA/DATASETS (i.e. inside a directory called DATASETS inside the one where you're putting the corpora). Then, in the main directory where you are keeping the program code do:

>>> from basics import datasets

>>> bnc = datasets.BNC()

This will unzip 2554.zip and put it in a directory with the following structure inside CORPORA/TEXTS, where the actual data is in files inside A0, A1, .., Once you've done this you can delete 2554.zip to save space – it's quite big so you may not want to keep it lying around once you've unpacked it.

--|BNC--|download--|Doc--|HTML

|

|Src

|

|Texts--|A--|A0

|

|A1

|

|A2

|

|...

|

|B--|B0

|

|B1

|

|B2

|

| ...

|

| ...

Similarly we can downloadf the UDT:

>>> udt = datasets.UDT()

The UDT consists of a collection of treebanks supplied by different people, with a variety of licences. Different treebanks from within the overall collection have different licences. … DO NOT USE THESE TREEBANKS UNLESS YOU AGREE TO THE LICENCES.

Are you sure that your planned use of this material falls within these terms and conditions? (y/n)

y

Trying

https://lindat.mff.cuni.cz/repository/xmlui/bitstream/handle/11234/1-5150/ud-treebanks-v2.12.tgz?sequence=1&isAllowed=y

This will put the UDT in a directory with the following structure inside CORPORA/TREEBANKS (it's inside TREEBANKS rather than TEXTS because the data is annotated so that it can be interpreted as a set of trees), where the actual data is in files in the wholething files inside the per language subdirectories. There is a lot more material in the UDT itself, but since we do not use it for anything we collect all the trees from files within a subdirectory into a single file called wholething and then remove everything else (except for the licence details for the subdirectory) to save space:

--|UDT--|ud-treebanks-v2.12--|UD\_Abaza-ATB--|LICENSE.txt

|

|wholething

|

|UD\_Afrikaans-AfriBooms--|LICENSE.txt

|

|wholething

|

|UD\_Akkadian-PISANDUB--|LICENSE.txt

|

|wholething

|

|...

Because the UDT contains treebanks with a variety of different licences, we provide code for finding ones with Creative-Commons-Attribution-Share-Alike and Creative-Commons-Attribution-NonCommercial-ShareAlike licences. The former can be used for almost any activity, the latter cannot be used for any commercial purposes (see <https://creativecommons.org/licenses/>? for the full details).

>>> ccasa, ccancsa = datasets.UDT.readlicences()

>>> printall(ccasa)

UD\_Abaza-ATB

UD\_Afrikaans-AfriBooms

UD\_Akkadian-PISANDUB

UD\_Akkadian-RIAO

...

>>> printall(ccancsa)

UD\_Basque-BDT

UD\_Bulgarian-BTB

UD\_Czech-FicTree

...

We can then get all the names of all the files with Creative-Commons-Attribution-Share-Alike licences for a given language as follows:

>>> ccasaEnglish = [l for l in ccasa if l.startswith("UD\_English")]

>>> ccasaEnglish

['UD\_English-Atis', 'UD\_English-ESLSpok', 'UD\_English-EWT', 'UD\_English-PUD', 'UD\_English-Pronouns']

Readers

Once we have our datasets, we have to be able to do something with them. For many tasks, we want to extract the words or the words with their tags. For this, we use the notion of readers from Chapter 4. We specify where we want to read from (readers has a value stored for the path to the UDT, so we just join the specific file we want to read from to that), how we want to read leaf files (e.g. using UDTTaggedWordReader, defined in readers, and the pattern that files we want to actually read must match (because we don't want to read the license files). Note that what we get when we make a reader is in fact a generator: we can enumerate items from it one at a time, or we can solidify It into a list.

>>> from basics import readers

>>> u = readers.reader(os.path.join(readers.UDT, 'UD\_English-Atis'), readers.UDTTaggedWordReader, pattern=".\*wholething")

>>> u

<generator object reader at 0x7f947a4486d0>

>>> l = list(u)

>>> len(l)

61879

>>> printall(l[:10])

('what', 'PRON')

('is', 'AUX')

('the', 'DET')

('cost', 'NOUN')

('of', 'ADP')

('a', 'DET')

('round', 'NOUN')

('trip', 'NOUN')

('flight', 'NOUN')

('from', 'ADP')

We can also supply a list of files and get a reader that returns everything in any of them:

>>> allEnglish = [os.path.join(readers.UDT, f) for f in ccsaEnglish]

>>> allEnglish

['CORPORA/TREEBANKS/UDT/ud-treebanks-v2.12/UD\_English-Atis', 'CORPORA/TREEBANKS/UDT/ud-treebanks-v2.12/UD\_English-ESLSpok', 'CORPORA/TREEBANKS/UDT/ud-treebanks-v2.12/UD\_English-EWT', 'CORPORA/TREEBANKS/UDT/ud-treebanks-v2.12/UD\_English-PUD', 'CORPORA/TREEBANKS/UDT/ud-treebanks-v2.12/UD\_English-Pronouns']

>>> u = readers.reader(allEnglish, readers.UDTTaggedWordReader, pattern=".\*wholething")

>>> l = list(u)

>>> len(l)

360891

Word parts

We looked at an increasingly complicated series of programs for finding the underlying structure of words, starting with stem1, which basically did stemming (i.e. it removed affixes from inflected forms, but without paying much attention to the changes that happens at the borders when you add affixes to a root), and ending with stem3, which did morphological analysis (i.e. it analysed the underlying structure of surface forms in ways which paid attention to the boundary effects and to the constraints on which affixes can attach to which words). We didn't follow this up in great detail, but it may well be important to get this right, so we will look in more detail at stem2 to see what the limitations and issues associated with it are, and we will extend it to stem3, which is more complete than anything that is easily available on-line and which even so has a number of problems.

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!

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("redistributions")[0]

're-distribute+ion+s'

>>> stem2.allstems("unsurprisingly")[0]

'un-expect+ed+ly'

>>> stem2.allstems("unsurprisingly")[0]

'un-surprise+ing+ly'

>>> stem2.allstems("unreconstructed")[0]

'un-re-construct+ed'

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 really 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', 'aesthetically']

>>> stem2.allstems("unreconstructed")

['un-re-construct+ed', 'un-reconstruct+ed', 'un-reconstructed', 'unreconstructed']

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-valuate+ion', 'un-de-re-valuation', 'un-de-revaluation', 'underevaluation']

*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 afstem3
* s 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:

>>> 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 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):

>>> 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 *undervaluation*. 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 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:

>>> 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

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, 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; 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

>>> bnc = list(readers.reader(readers.BNC,

readers.BNCWordReader,

pattern=".\*/[A-Z0-9]\*\.xml"))

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, t1=50, t2=100)

111651731 words

get rid of the top 100 words because they will make too many pairs (e.g. 'of-the')

TOP SCORERS: {'people', 'their', 'can', 'know', 'the', 'very', ';', 'are', 'he', 'for', 'into', 'It', 'at', ')', 'more', 'my', 'your', 'only', 'be', 'all', 'but', 'no', 'said', 'a', 'she', 'I', 'on', 'any', '.', '!', '(', 'up', 'they', 'In', 'did', 'that', 'him', 'if', 'so', 'would', ',', 'have', 'its', ':', 'He', 'this', "'s", '—', 'out', 'time', '’', 'The', 'and', 'you', 'than', 'to', "n't", '?', 'is', 'an', 'had', 'me', 'could', 'been', 'by', 'one', 'who', 'or', 'But', 'not', 'do', 'will', 'then', 'which', 'it', 'with', 'from', 'we', 'there', 'over', 'his', 'also', 'when', 'about', 'her', 'what', 'in', 'two', 'just', 'some', 'other', '‘', 'was', 'were', 'them', 'A', 'of', 'has', 'as', 'like'}

760415 distinct words found (111651731 tokens)

Getting pairs that occur at least 50 times

67372 pairs found

Calculating PMI

The top 20 pairs are as below. They're mainly foreign language expressions which have been imported into English or technical terms (or both!), and as such it seems unlikely that they will carry much emotional baggage, and certainly not much emotional baggage that the second word would not carry by itself.

>>> printall(pmi[:20])

(16.77865907423096, 'fromage-frais', 60)

(16.668033200696925, 'lapis-lazuli', 59)

(16.291224648965255, 'fait-accompli', 61)

(16.25241115260958, 'comings-goings', 66)

(16.019803097385253, 'restorative-proctocolectomy', 54)

(15.884724109045331, 'lamina-propria', 120)

(15.85072676734132, 'retrograde-cholangiopancreatography', 57)

(15.823507989567783, 'kung-fu', 120)

(15.790696499182797, 'bom-bom', 110)

(15.69714767941302, 'sclerosing-cholangitis', 150)

(15.622820911271566, 'nouveau-roman', 54)

(15.598384061334848, 'mens-rea', 96)

(15.456839941106432, 'bona-fide', 222)

(15.455973314828082, 'autonomic-neuropathy', 73)

(15.434654007444665, 'barium-enema', 58)

(15.284455562334932, 'apical-papilla', 58)

(15.218547653080698, 'ultra-vires', 181)

(15.213913376585205, 'cystic-fibrosis', 164)

(15.207229495425194, 'myocardial-infarction', 176)

(15.193081988807787, 'plasminogen-activator', 203)

We can restrict our attention to more frequent pairs, e.g. to pairs that occur at least 500 times, but the pattern doesn't change very much:

>>> fpairs = [p for p in pmi if p[-1] > 500]

>>> printall(fpairs[:20])

(14.10789557602586, 'ulcerative-colitis', 708)

(13.730483221346029, 'vice-versa', 667)

(13.600053898897935, 'gall-bladder', 603)

(12.956064741908307, 'carbon-dioxide', 976)

(12.55829842681955, 'mentally-handicapped', 564)

(12.328294856503494, 'et-al', 2963)

(12.083121068394941, 'ha-ha', 665)

(11.92260342140775, 'du-n', 1569)

(11.549559577632149, 'swimming-pool', 724)

(11.428007498740625, 'prime-minister', 988)

(11.40111340735783, 'gon-na', 12242)

(11.353146870773811, 'wan-na', 2485)

(11.245880258326437, 'civil-servants', 927)

(11.209146352340168, 'managing-director', 1237)

(11.10587427266316, 'nineteen-ninety', 819)

(11.066883146984042, 'chief-executive', 1793)

(11.045523297225484, 'joint-venture', 592)

(11.040113326188095, 'nineteenth-century', 2649)

(10.981133044248542, 'raw-materials', 617)

(10.953957053061597, 'twentieth-century', 1318)

We do get fewer technical terms and foreign language pairs, but it again seems unlikely that the pairs carry significantly different emotional weight from that carried by their constituent parts. There are cases with reasonably high PMI scores where this is true, but they aren't easy to identify. 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:

>>> 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)

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

We will consider three taggers -- a very simple one that just uses word frequencies, an HMM-based one which pays attention to the