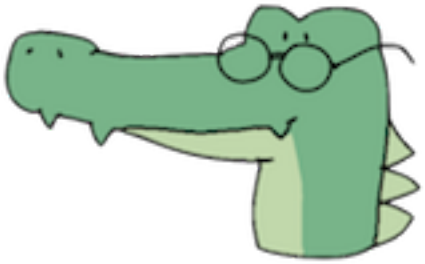


corpkit documentation

Release 2.0.13

Daniel McDonald

May 09, 2016

corpkit is a Python-based tool for doing more sophisticated corpus linguistics.

It does a lot of the usual things, like parsing, interrogating, concordancing and keywording, but also extends their potential significantly: you can create structured corpora with speaker ID labels, and easily restrict searches to individual speakers, subcorpora or groups of files.

You can interrogate parse trees, CoreNLP dependencies, lists of tokens or plain text for combinations of lexical and grammatical features. Results can be quickly edited, sorted and visualised in complex ways, saved and loaded within projects, or exported to formats that can be handled by other tools.

Concordancing is extended to allow the user to query and display grammatical features alongside tokens. Keywording can be restricted to certain word classes or positions within the clause. If your corpus contains multiple documents or subcorpora, you can identify keywords in each, compared to the corpus as a whole.

corpkit leverages Stanford CoreNLP, NLTK and pattern for the linguistic heavy lifting, and pandas and matplotlib for storing, editing and visualising interrogation results. Multiprocessing is available via joblib, and Python 2 and 3 are both supported.

Example

Here's a basic workflow, using a corpus of news articles published between 1987 and 2014, structured like this:

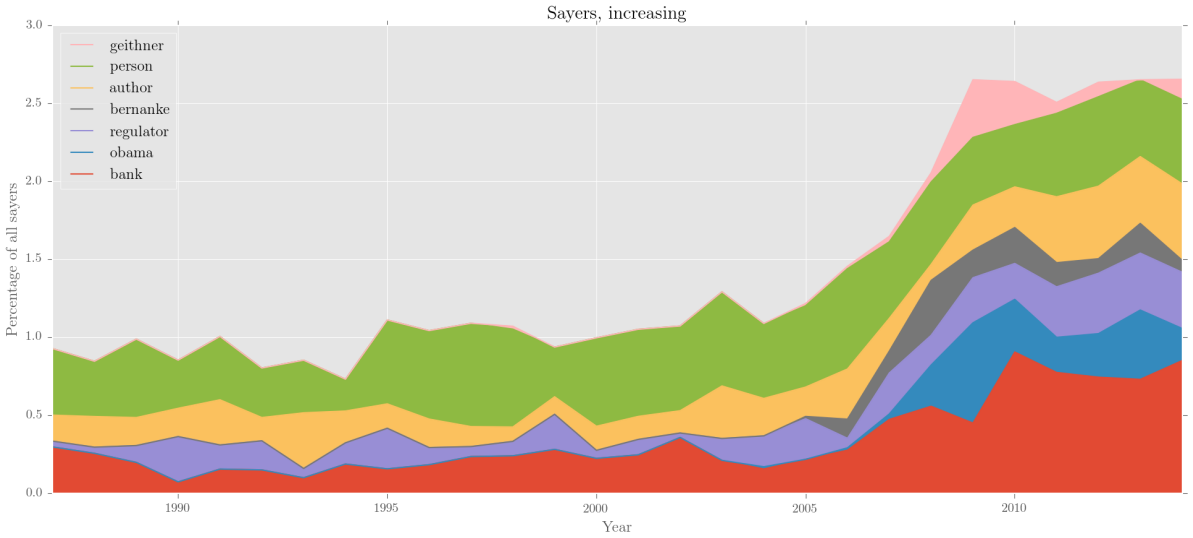
```
./data/NYT:

├---1987
|   ├---NYT-1987-01-01-01.txt
|   ├---NYT-1987-01-02-01.txt
|   ...
|
├---1988
|   ├---NYT-1988-01-01-01.txt
|   ├---NYT-1988-01-02-01.txt
|   ...
...
```

Below, this corpus is made into a *Corpus* object, parsed with *Stanford CoreNLP*, and interrogated for a lexicogrammatical feature. Absolute frequencies are turned into relative frequencies, and results sorted by trajectory. The edited data is then plotted.

```
>>> from corpkit import *
>>> from dictionaries import processes

### parse corpus of NYT articles containing annual subcorpora
>>> unparsed = Corpus('data/NYT')
>>> parsed = unparsed.parse()

### query: nominal nsubjs that have verbal process as governor lemma
>>> crit = {F: r'^nsubj$',
...         GL: processes.verbal.lemmata,
...         P: r'^N'}

### interrogate corpus, outputting lemma forms
>>> sayers = parsed.interrogate(crit, show=L)
>>> sayers.quickview(10)

   0: official    (n=4348)
   1: expert      (n=2057)
   2: analyst     (n=1369)
   3: report      (n=1103)
   4: company     (n=1070)
   5: which       (n=1043)
   6: researcher  (n=987)
   7: study       (n=901)
   8: critic      (n=826)
   9: person      (n=802)

### get relative frequency and sort by increasing
>>> rel_say = sayers.edit('%', SELF, sort_by='increase')
```

```
### plot via matplotlib, using tex if possible
>>> rel_say.visualise('Sayers, increasing', kind='area',
...                   y_label = 'Percentage of all sayers')
```

Output:



Installation

Via pip:

```
pip install corpkit
```

via Git:

```
git clone https://www.github.com/interrogator/corpkit
cd corpkit
python setup.py install
```

Parsing and interrogation of parse trees will also require *Stanford CoreNLP*. *corpkit* can download and install it for you automatically.

Graphical interface

Much of corpkit's command line functionality is also available in the *corpkit* GUI. After installation, it can be started with:

```
python -m corpkit.gui
```

Alternatively, it's available (alongside documentation) as a standalone OSX app here.

CHAPTER 1

Creating projects and building corpora

Doing corpus linguistics involves building and interrogating corpora, and exploring interrogation results. `corpkit` helps with all of these things. This page will explain how to create a new project and build a corpus.

1.1 Creating a new project

The simplest way to begin using corpkit is to import it and to create a new project. Projects are simply folders containing subfolders where corpora, saved results, images and dictionaries will be stored. The simplest way is to do it from *bash*, passing in the name you'd like for the project:

```
$ new_project psyc
# move there:
$ cd psyc
# now, enter python and begin ...
```

Or, from Python:

```
>>> import corpkit
>>> corpkit.new_project('psyc')
### move there:
>>> import os
>>> os.chdir('psyc')
>>> os.listdir('.')

['data',
 'dictionaries',
 'exported',
 'images',
 'logs',
 'saved_concordances',
 'saved_interrogations']
```

1.2 Adding a corpus

Now that we have a project, we need to add some plain-text data to the *data* folder. At the very least, this is simply a text file. Better than this is a folder containing a number of text files. Best, however, is a folder containing

subfolders, with each subfolder containing one or more text files. These subfolders represent subcorpora.

You can add your corpus to the *data* folder from the command line, or using Finder/Explorer if you prefer.

```
$ cp -R /Users/me/Documents/transcripts ./data
```

Or, in *Python*, using *shutil*:

```
>>> import shutil
>>> shutil.copytree('/Users/me/Documents/transcripts', './data')
```

If you've been using *bash* so far, this is the moment when you'd enter *Python* and *import corpkit*.

1.3 Creating a Corpus object

Once we have a corpus of text files, we need to turn it into a *Corpus* object.

```
>>> from corpkit import Corpus
### you can leave out the 'data' if it's in there
>>> unparsed = Corpus('data/transcripts')
>>> unparsed
<corpkit.corpus.Corpus instance: transcripts; 13 subcorpora>
```

This object can now be interrogated using the *interrogate()* method:

```
>>> th_words = unparsed.interrogate({W: r'th[a-z-]+'})
### show 5x5 (Pandas syntax)
>>> th_words.results.iloc[:5,:5]

S   that  the  then  think  thing
01   144  139    63     53     43
02   122  114    74     35     45
03   132   74    56     57     25
04   138   67    71     35     44
05   173   76    67     35     49
```

1.4 Parsing a corpus

Instead of interrogating the plaintext corpus, what you'll probably want to do, is parse it, and interrogate the parser output. For this, *corpkit.corpus.Corpus* objects have a *parse()* method. This relies on Stanford CoreNLP's parser, and therefore, you must have the parser and Java installed. `corpkit` will look around in your PATH for the parser, but you can also pass in its location manually with (e.g.) `corenlppath='users/you/corenlp'`. If it can't be found, you'll be asked if you want to download and install it automatically.

```
>>> corpus = unparsed.parse()
```

Note: Remember that parsing is a computationally intensive task, and can take a long time!

`corpkit` can also work with speaker IDs. If lines in your file contain capitalised alphanumeric names, followed by a colon (as per the example below), these IDs can be stripped out and turned into metadata features in the XML.

```
JOHN: Why did they change the signs above all the bins?
SPEAKER23: I know why. But I'm not telling.
```

To use this option, use the `speaker_segmentation` keyword argument:

```
>>> corpus = unparsed.parse(speaker_segmentation=True)
```

Parsing creates a corpus that is structurally identical to the original, but with annotations as XML files in place of the original `.txt` files. There are also methods for multiprocessing, memory allocation and so on:

parse() argument	Type	Purpose
corenlppath	*str*	Path to CoreNLP
nltk_data_path	*str*	Path to *punkt* tokeniser
operations	*str*	List of annotations
copula_head	*bool*	Make copula head of dependency parse
speaker_segmentation	*bool*	Do speaker segmentation
memory_mb	*int*	Amount of memory to allocate
multiprocess	*int/bool*	Process in *n* parallel jobs

1.5 Manipulating a parsed corpus

Once you have a parsed corpus, you're ready to analyse it. `corpkit.corpus.Corpus` objects can be navigated in a number of ways. *CoreNLP XML* is used to navigte the internal structure of XML files within the corpus.

```
>>> corpus[:3]                           # access first three subcorpora
>>> corpus.subcorpora.chapter1           # access subcorpus called chapter1
>>> f = corpus[5][20]                    # access 21st file in 6th subcorpus
>>> f.document.sentences[0].parse_string # get parse tree for first sentence
>>> f.document.sentences.tokens[0].word  # get first word
```

1.6 Counting key features

Before constructing your own queries, you may want to use some predefined attributes for counting key features in the corpus.

```
>>> corpus.features
```

Output:

```
S   Characters   Tokens    Words  Closed class  Open class  Clauses  Sentences  Unmod. declarative  Passives
01     4380658  1258606  1092113        643779      614827   277103      68267               35981     16842
02     3185042   922243   800046        471883      450360   209448      51575               26149     10324
03     3157277   917822   795517        471578      446244   209990      51860               26383      9711
04     3261922   948272   820193        486065      462207   216739      53995               27073      9697
05     3164919   921098   796430        473446      447652   210165      52227               26137      9543
06     3187420   928350   797652        480843      447507   209895      52171               25096      8917
07     3080956   900110   771319        466254      433856   202868      50071               24077      8618
08     3356241   972652   833135        502913      469739   218382      52637               25285      9921
09     2908221   840803   725108        434839      405964   191851      47050               21807      8354
10     2868652   815101   708918        421403      393698   185677      43474               20763      8640
```

This can long time, as it counts a number of complex features. Once it's done, however, it saves automatically, so you don't need to do it again. There are also *postags* and *wordclasses* attributes:

```
>>> corpus.postags
>>> corpus.wordclasses
```

These results can be useful when generating relative frequencies later on. Right now, however, you're probably interested in searching the corpus yourself, however. Hit *Next* to learn about that.

CHAPTER 2

Interrogating corpora

Once you've built a corpus, you can search it for linguistic phenomena. This is done with the *interrogate()* method.

2.1 Introduction

Interrogations can be performed on any *corpkit.corpus.Corpus* object, but also, on *corpkit.corpus.Subcorpus* objects, *corpkit.corpus.File* objects and corpkit.corpus.Datalist's (slices of ``Corpus` objects). You can search plaintext corpora, tokenised corpora or fully parsed corpora using the same method. We'll focus on parsed corpora in this guide.

```
>>> from corpkit import *
### words matching 'woman', 'women', 'man', 'men'
>>> query = {W: r'/(^wo)m.n/'}
### interrogate corpus
>>> corpus.interrogate(query)
### interrogate parts of corpus
>>> corpus[2:4].interrogate(query)
>>> corpus.files[:10].interrogate(query)
### if you have a subcorpus called 'abstract':
>>> corpus.subcorpora.abstract.interrogate(query)
```

Note: Single capital letter variables in code examples represent lowercase strings (W = 'w'). These variables are made available by doing from corpkit import *. They are used here for readability.

2.2 Search types

Parsed corpora contain many different kinds of annotation we might like to search. The annotation types, and how to specify them, are given in the table below:

Search	Gloss
W	Word
L	Lemma
F	Function
P	POS tag
G/GW	Governor word
GL	Governor lemma
GF	Governor function
GP	Governor POS
D/DW	Dependent word
DL	Dependent lemma
DF	Dependent function
DP	Dependent POS
PL	Word class
N	Ngram
R	Distance from root
I	Index in sentence
T	Tregex tree

The `search` argument is generally a `dict` object, whose keys specify the annotation to search (i.e. a string from the above table), and whose values are the regular-expression or wordlist based queries. Because it comes first, and because it's always needed, you can pass it in like an argument, rather than a keyword argument.

```
### get variants of the verb 'be'
>>> corpus.interrogate({L: 'be'})
### get words in 'nsubj' position
>>> corpus.interrogate({F: 'nsubj'})
```

Multiple key/value pairs can be supplied. By default, all must match for the result to be counted, though this can be changed with `searchmode=ANY` or `searchmode=ALL`:

```
>>> goverb = {P: r'^v', L: r'^go'}
### get all variants of 'go' as verb
>>> corpus.interrogate(goverb, searchmode=ALL)
### get all verbs and any word starting with 'go':
>>> corpus.interrogate(goverb, searchmode=ANY)
```

2.3 Excluding results

You may also wish to exclude particular phenomena from the results. The `exclude` argument takes a `dict` in the same form a `search`. By default, if any key/value pair in the `exclude` argument matches, it will be excluded. This is controlled by `excludemode=ANY` or `excludemode=ALL`.

```
>>> from dictionaries import wordlists
### get any noun, but exclude closed class words
>>> corpus.interrogate({P: r'^n'}, exclude={W: wordlists.closedclass})
### when there's only one search criterion, you can also write:
>>> corpus.interrogate(P, r'^n', exclude={W: wordlists.closedclass})
```

In many cases, rather than using `exclude`, you could also remove results later, during editing.

2.4 What to show

Up till now, all searches have simply returned words. The final major argument of the `interrogate` method is `show`, which dictates what is returned from a search. Words are the default value. You can use any of the search values as a show value, plus a few extra values for n-gramming:

`show` can be either a single string or a list of strings. If a list is provided, each value is returned with forward slashes as delimiters.

```
>>> example = corpus.interrogate({W: r'fr?iends?'}, show=[W, L, P])
>>> list(example.results)

['friend/friend/nn', 'friends/friend/nns', 'fiend/fiend/nn', 'fiends/fiend/nns', ... ]
```

N-gramming is therefore as simple as:

```
>>> example = corpus.interrogate({W: r'wom[ae]n]'}, show=N, gramsize=2)
>>> list(example.results)

['a woman', 'the woman', 'the women', 'women are', ... ]
```

One further extra show value is `'c'` (count), which simply counts occurrences of a phenomenon. Rather than returning a DataFrame of results, it will result in a single Series. It cannot be combined with other values.

2.5 Working with trees

If you have elected to search trees, you'll need to write a *Tregex query*. Tregex is a language for searching syntax trees like this one:

To write a Tregex query, you specify *words and/or tags* you want to match, in combination with *operators* that link them together. First, let's understand the Tregex syntax.

To match any adjective, you can simply write:

```
JJ
```

with *JJ* representing adjective as per the Penn Treebank tagset. If you want to get NPs containing adjectives, you might use:

```
NP < JJ
```

where < means *with a child/immediately below*. These operators can be reversed: If we wanted to show the adjectives within NPs only, we could use:

```
JJ > NP
```

It's good to remember that **the output will always be the left-most part of your query**.

If you only want to match Subject NPs, you can use bracketting, and the *$* operator, which means *sister/directly to the left/right of*:

```
JJ > (NP $ VP)
```

In this way, you build more complex queries, which can extent all the way from a sentence's *root* to particular tokens. The query below, for example, finds adjectives modifying *book*:

```
JJ > (NP <<# /book/)
```

Notice that here, we have a different kind of operator. The << operator means that the node on the right does not need to be a child, but can be a descendent. the *#* means *head'—that is, in SFL, it matches the 'Thing* in a Nominal Group.

If we wanted to also match *magazine* or *newspaper*, there are a few different approaches. One way would be to use | as an operator meaning *or*:

```
JJ > (NP ( <<# /book/ | <<# /magazine/ | <<# /newspaper/))
```

This can be cumbersome, however. Instead, we could use a regular expression:

```
JJ > (NP <<# /^(book|newspaper|magazine)s*$/)
```

Though it is unfortunately beyond the scope of this guide to teach Regular Expressions, it is important to note that Regular Expressions are extremely powerful ways of searching text, and are invaluable for any linguist interested in digital datasets.

Detailed documentation for Tregex usage (with more complex queries and operators) can be found here.

2.6 Tree return values

Though you can use the same Tregex query for tree searches, the output changes depending on what you select as the *return* value. For the following sentence:

```
These are prosperous times.
```

you could write a query:

```
r'JJ < __'
```

Which would return:

Show	Gloss	Output
W	Word	*prosperous*
T	Tree	*(JJ prosperous)*
p	POS tag	*JJ*
C	Count	*1* (added to total)

2.7 Working with dependencies

When working with dependencies, you can use any of the long list of search and return values. It's possible to construct very elaborate queries:

```
>>> from dictionaries import process_types, roles
### nominal nsubj with verbal process as governor
>>> crit = {F: r'^nsubj$',
...         GL: processes.verbal.lemmata,
             GF: roles.event,
...          P: r'^N'}
### interrogate corpus, outputting the nsubj lemma
>>> sayers = parsed.interrogate(crit, show=L)
```

You can also select from the three dependency grammars used by CoreNLP: one of `'basic-dependencies'`, `'collapsed-dependencies'`, or `'collapsed-ccprocessed-dependencies'` can be passed in as `dep_type`:

```
>>> corpus.interrogate(query, dep_type='collapsed-ccprocessed-dependencies')
```

2.8 Multiprocessing

Interrogating the corpus can be slow. To speed it up, you can pass an integer as the `multiprocess` keyword argument, which tells the `interrogate()` method how many processes to create.

```
>>> corpus.interrogate({T: r'__ > MD'}, multiprocess=4)
```

Note that too many parallel processes may slow your computer down. If you pass in `multiprocessing=True`, the number of processes will equal the number of cores on your machine. This is usually a fairly sensible number.

2.9 N-grams

N-gramming can be done simply by using an n-gram string (`N`, `NL`, `NP` or `NPL`) as the *show* value. Two options for n-gramming are `gramsize=n`, where `n` determines the number of tokens in the n-gram, and `split_contractions=True`, which controls whether or not words like *doesn't* are treated as one token or two.

```
>>> corpus.interrogate({W: 'father'}, show='NL', gramsize=3, split_contractions=False)
```

2.10 Saving interrogations

```
>>> interro.save('savename')
```

Interrogation savenames will be prefaced with the name of the corpus interrogated.

You can also quicksave interrogations:

```
>>> corpus.interrogate(T, r'/NN.?/', save='savename')
```

2.11 Exporting interrogations

If you want to quickly export a result to CSV, LaTeX, etc., you can use Pandas' DataFrame methods:

```
>>> print(nouns.results.to_csv())
>>> print(nouns.results.to_latex())
```

2.12 Other options

`interrogate()` takes a number of other arguments, each of which is documented in the API documentation.

If you're done interrogating, you can head to the page on *Editing results* to learn how to transform raw frequency counts into something more meaningful. Or, Hit *Next* to learn about concordancing.

CHAPTER 3

Concordancing

Any interrogation is also optionally a concordance. If you use the `do_concordancing` keyword argument, your interrogation will have a `concordance` attribute containing concordance lines. Like interrogation results, concordances are stored as Pandas DataFrames. `maxconc` controls the number of lines produced.

```
>>> withconc = corpus.interrogate(T, r'/JJ.?/ > (NP <<# /man/)',
                                  do_concordancing=True, maxconc=500)
```

If you don't want or need the interrgation data, you can use the *`concordance()`* method:

```
>>> conc = corpus.concordance(T, r'/JJ.?/ > (NP <<# /man/)')
```

3.1 Displaying concordance lines

How concordance lines will be displayed really depends on your interpreter and environment. For the most part, though, you'll want to use the *`format()`* method.

```
>>> lines.format(kind='s'
                 n=100
                 window=50,
                 columns=[L, M, R])
```

`kind` allows you to print as CSV (`'c'`), as LaTeX (`'l'`), or simple string (`'s'`). `n` controls the number of results shown. `window` controls how much context to show in the left and right columns. `columns` accepts a list of column names to show.

Pandas' set_option can be used to customise some visualisation defaults.

3.2 Working with concordance lines

You can edit concordance lines using the *`edit()`* method. You can use this method to keep or remove entries or subcorpora matching regular expressions or lists. Keep in mind that because concordance lines are DataFrames, you can use Pandas' dedicated methods for working with text data.

```
### get just uk variants of words with variant spellings
>>> from dictionaries import usa_convert
>>> concs = result.concordance.edit(just_entries=usa_convert.keys())
```

Concordance objects can be saved just like any other `corpkit` object:

```
>>> concs.save('adj_modifying_man')
```

You can also easily turn them into CSV data, or into LaTeX:

```
### pandas methods
>>> concs.to_csv()
>>> concs.to_latex()

### corpkit method: csv and latex
>>> concs.format('c', window=20, n=10)
>>> concs.format('l', window=20, n=10)
```

You can use the *calculate()* method to generate a frequency count of the middle column of the concordance. Therefore, one method for ensuring accuracy is to:

1. Run an interrogation, using `do_concordance=True`
2. Remove false positives from the concordance result
3. Use the calculate method to regenerate the overall frequency

If you'd like to randomise the order of your results, you can use `lines.shuffle()`

CHAPTER 4

Editing results

Corpus interrogation is the task of getting frequency counts for a lexicogrammatical phenomenon in a corpus. Simple absolute frequencies, however, are of limited use. The `edit()` method allows us to do complex things with our results, including:

Each of these will be covered in the sections below. Keep in mind that because results are stored as DataFrames, you can also use Pandas/Numpy/Scipy to manipulate your data in ways not covered here.

4.1 Keeping or deleting results and subcorpora

One of the simplest kinds of editing is removing or keeping results or subcorpora. This is done using keyword arguments: *skip_subcorpora*, *just_subcorpora*, *skip_entries*, *just_entries*. The value for each can be:

1. A string (treated as a regular expression to match)
2. A list (a list of words to match)
3. An integer (treated as an index to match)

```
>>> criteria = r'ing$'
>>> result.edit(just_entries = criteria)
```

```
>>> criteria = ['everything', 'nothing', 'anything']
>>> result.edit(skip_entries = criteria)
```

```
>>> result.edit(just_subcorpora = ['Chapter_10', 'Chapter_11'])
```

You can also span subcorpora, using a tuple of `(first_subcorpus, second_subcorpus)`. This works for numerical and non-numerical subcorpus names:

```
>>> just_span = result.edit(span_subcorpora=(3, 10))
```

4.2 Editing result names

You can use the `replace_names` keyword argument to edit the text of each result. If you pass in a string, it is treated as a regular expression to delete from every result:

```
>>> ingdel = result.edit(replace_names=r'ing$')
```

You can also pass in a `dict` with the structure of `{newname: criteria}`:

```
>>> rep = {'-ing words': r'ing$', '-ed words': r'ed$'}
>>> replaced = result.edit(replace_names=rep)
```

If you wanted to see how commonly words start with a particular letter, you could do something creative:

```
>>> from string import lowercase
>>> crit = {k.upper() + ' words': r'(?i)^%s.*' % k for k in lowercase}
>>> firstletter = result.edit(replace_names=crit, sort_by='total')
```

4.3 Spelling normalisation

When results are single words, you can normalise to UK/US spelling:

```
>>> spelled = result.edit(spelling='UK')
```

You can also perform this step when interrogating a corpus.

4.4 Generating relative frequencies

Because subcorpora often vary in size, it is very common to want to create relative frequency versions of results. The best way to do this is to pass in an `operation` and a `denominator`. The `operation` is simply a string denoting a mathematical operation: '+', '-', '*', '/', '%'. The last two of these can be used to get relative frequencies and percentage.

Denominator is what the result will be divided by. Quite often, you can use the string `'self'`. This means, after all other editing (deleting entries, subcorpora, etc.), use the totals of the result being edited as the denominator. When doing no other editing operations, the two lines below are equivalent:

```
>>> rel = result.edit('%', 'self')
>>> rel = result.edit('%', result.totals)
```

The best denominator, however, may not simply be the totals for the results being edited. You may instead want to relativise by the total number of words:

```
>>> rel = result.edit('%', corpus.features.Words)
```

Or by some other result you have generated:

```
>>> words_with_oo = corpus.interrogate(W, 'oo')
>>> rel = result.edit('%', words_with_oo.totals)
```

There is a more complex kind of relative frequency making, where a `.results` attribute is used as the denominator. In the example below, we calculate the percentage of the time each verb occurs as the *root* of the parse.

```
>>> verbs = corpus.interrogate(P, r'^vb', show=L)
>>> roots = corpus.interrogate(F, 'root', show=L)
>>> relv = verbs.edit('%', roots.results)
```

4.5 Keywording

`corpkit` treats keywording as an editing task, rather than an interrogation task. This makes it easy to get key nouns, or key Agents, or key grammatical features. To do keywording, use the `K` operation:

```
### use predefined global variables
>>> from corpkit import *
>>> keywords = result.edit(K, SELF)
```

This finds out which words are key in each subcorpus, compared to the corpus as a whole. You can compare subcorpora directly as well. Below, we compare the `plays` subcorpus to the `novels` subcorpus.

. code-block:: python

```
>>> from corpkit import *
>>> keywords = result.edit(K, result.ix['novels'], just_subcorpora='plays')
```

You could also pass in word frequency counts from some other source. A wordlist of the British National Corpus is included:

```
>>> keywords = result.edit(K, 'bnc')
```

4.6 Sorting

You can sort results using the `sort_by` keyword. Possible values are:

- *'name'* (alphabetical)
- *'total'* (most common first)
- *'infreq'* (inverse total)
- *'increase'* (most increasing)
- *'decrease'* (most decreasing)
- *'turbulent'* (by most change)
- *'static'* (by least change)
- *'p'* (by p value)
- *'slope'* (by slope)
- *'intercept'* (by intercept)
- *'r'* (by correlation coefficient)
- *'stderr'* (by standard error of the estimate)
- *'<subcorpus>'* by total in <subcorpus>

```
>>> inc = result.edit(sort_by='increase', keep_stats=False)
```

Many of these rely on Scipy's `linregress` function. If you want to keep the generated statistics, use `keep_stats=True`.

4.7 Calculating trends, P values

`keep_stats=True` will cause slopes, p values and stderr to be calculated for each result.

4.8 Saving results

You can save edited results to disk.

```
>>> edited.save('savename')
```

4.9 Exporting results

You can generate CSV data very easily using Pandas:

```
>>> result.results.to_csv()
```

CHAPTER 5

Visualising results

One thing missing in a lot of corpus linguistic tools is the ability to produce high-quality visualisations of corpus data. `corpkit` uses the *`corpkit.interrogation.Interrogation.visualise`* method to do this.

Note: Most of the keyword arguments from Pandas' plot method are available. See their documentation for more information.

5.1 Basics

`visualise()` is a method of all *`corpkit.interrogation.Interrogation`* objects. If you use *from corpkit import* *, it is also monkey-patched to Pandas objects.

Note: If you're using a *Jupyter Notebook*, make sure you use `%matplotlib inline` or `%matplotlib notebook` to set the appropriate backend.

A common workflow is to interrogate a corpus, relative results, and visualise:

```
>>> from corpkit import *
>>> corpus = Corpus('data/P-parsed', load_saved=True)
>>> counts = corpus.interrogate({T: r'MD < __'})
>>> reldat = counts.edit('%', SELF)
>>> reldat.visualise('Modals', kind='line', num_to_plot=ALL).show()
### the visualise method can also attach to the df:
>>> reldat.results.visualise(...).show()
```

The current behaviour of `visualise()` is to return the `pyplot` module. This allows you to edit figures further before showing them. Therefore, there are two ways to show the figure:

```
>>> data.visualise().show()
```

```
>>> plt = data.visualise()
>>> plt.show()
```

5.2 Plot type

The visualise method allows `line`, `bar`, horizontal bar (`barh`), `area`, and `pie` charts. Those with *seaborn* can also use `'heatmap'` (docs). Just pass in the type as a string with the `kind` keyword argument. Arguments such as `robust=True` can then be used.

```
>>> data.visualise(kind='heatmap', robust=True, figsize=(4,12),
...                x_label='Subcorpus', y_label='Event').show()
```

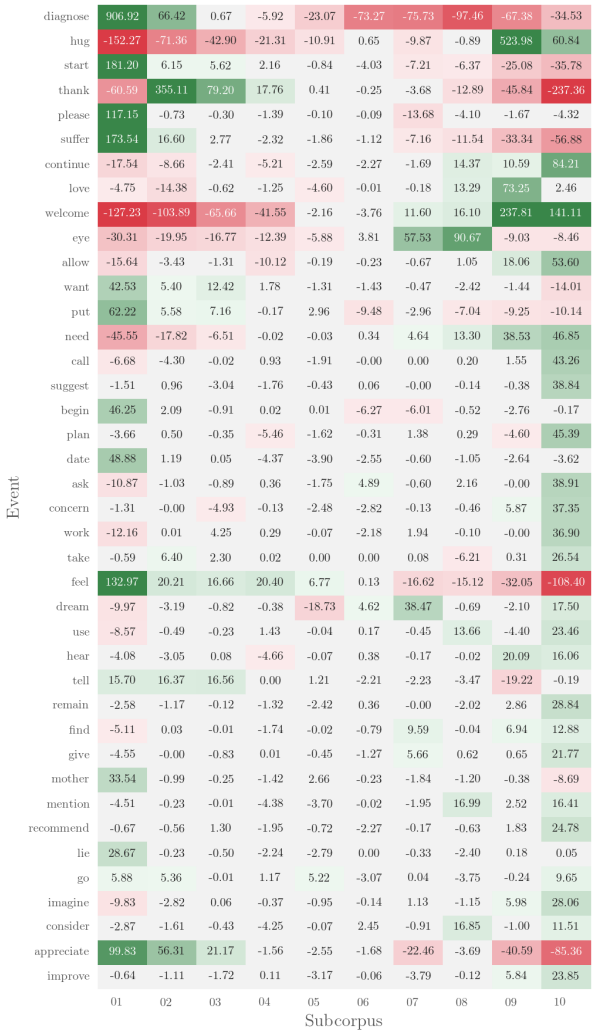
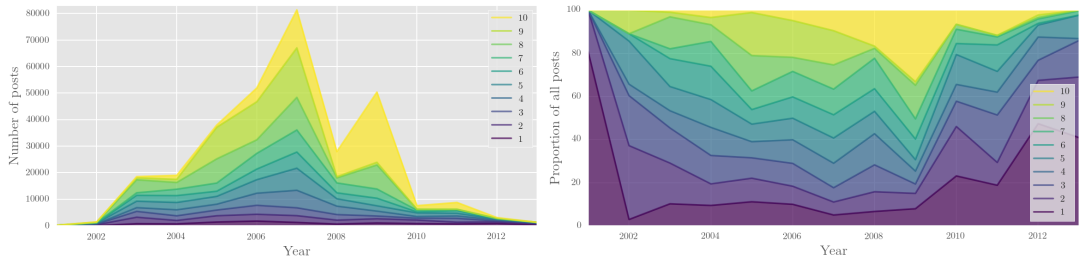


Fig. 5.1: Heatmap example

Stacked area/line plots can be made with `stacked=True`. You can also use `filled=True` to attempt to make all values sum to 100. Cumulative plotting can be done with `cumulative=True`. Below is an area plot beside an area plot where `filled=True`. Both use the `vidiris` colour scheme.



5.3 Plot style

You can select from a number of styles, such as `ggplot`, `fivethirtyeight`, `bmh`, and `classic`. If you have *seaborn* installed (and you should), then you can also select from *seaborn* styles (`seaborn-paper`, `seaborn-dark`, etc.).

5.4 Figure and font size

You can pass in a tuple of `(width, height)` to control the size of the figure. You can also pass an integer as `fontsize`.

5.5 Title and labels

You can label your plot with *title*, *x_label* and *y_label*:

```
>>> data.visualise('Modals', x_label='Subcorpus', y_label='Relative frequency')
```

5.6 Subplots

`subplots=True` makes a separate plot for every entry in the data. If using it, you'll probably also want to use `layout=(rows,columns)` to specify how you'd like the plots arranged.

```
>>> data.visualise(subplots=True, layout=(2,3)).show()
```

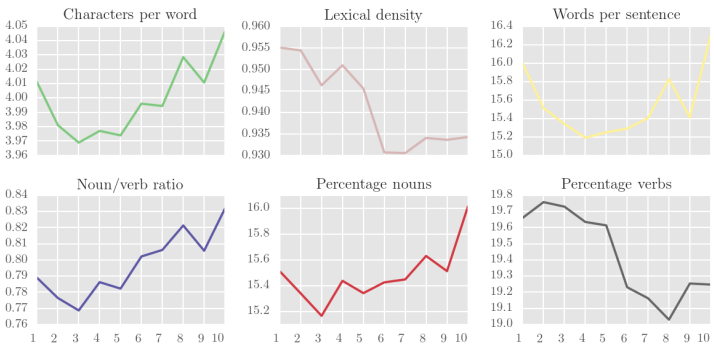


Fig. 5.2: Line charts using subplots and layout specification

5.7 TeX

If you have LaTeX installed, you can use `tex=True` to render text with LaTeX. By default, `visualise()` tries to use LaTeX if it can.

5.8 Legend

You can turn the legend off with `legend=False`. Legend placement can be controlled with `legend_pos`, which can be:

Margin	Figure		Margin
outside upper left	upper left	upper right	outside upper right
outside center left	center left	center right	outside center right
outside lower left	lower left	lower right	outside lower right

The default value, `'best'`, tries to find the best place automatically (without leaving the figure boundaries).

If you pass in `draggable=True`, you should be able to drag the legend around the figure.

5.9 Colours

You can use the `colours` keyword argument to pass in:

1. A colour name recognised by *matplotlib*
2. A hex colour string
3. A colourmap object

There is an extra argument, `black_and_white`, which can be set to `True` to make greyscale plots. Unlike `colours`, it also updates line styles.

5.10 Saving figures

To save a figure to a project's *images* directory, you can use the `save` argument. `output_format='png'/'pdf'` can be used to change the file format.

```
>>> data.visualise(save='name', output_format='png')
```

5.11 Other options

There are a number of further keyword arguments for customising figures:

A number of these and other options for customising figures are also described in the *corpkit.interrogation.Interrogation.visualise* method documentation.

CHAPTER 6

Managing projects

corpkit has a few other bits and pieces designed to make life easier when doing corpus linguistic work. This includes methods for loading saved data, for working with multiple corpora at the same time, and for switching between command line and graphical interfaces. Those things are covered here.

6.1 Loading saved data

When you're starting a new session, you probably don't want to start totally from scratch. It's handy to be able to load your previous work. You can load data in a few ways.

First, you can use `corpkit.load()`, using the name of the filename you'd like to load. By default, `corpkit` looks in the `saved_interrogations` directory, but you can pass in an absolute path instead if you like.

```
>>> import corpkit
>>> nouns = corpkit.load('nouns')
```

Second, you can use `corpkit.loader()`, which provides a list of items to load, and asks the user for input:

```
>>> nouns = corpkit.loader()
```

Third, when instantiating a `Corpus` object, you can add `load_saved=True` keyword argument to load any saved data belonging to this corpus as an attribute.

```
>>> corpus = Corpus('data/psyc-parsed', load_saved=True)
```

A final alternative approach stores all interrogations within an *corpkit.interrogation.Interrodict* object object:

```
>>> r = corpkit.load_all_results()
```

6.2 Managing multiple corpora

`corpkit` can handle one further level of abstraction for both Corpus and Interrogations. *corpkit.corpus.Corpora* models a collection of *corpkit.corpus.Corpus* objects. To create one, pass in a directory containing corpora, or a list of paths/`Corpus` objects:

```
>>> from corpkit import Corpora
>>> corpora = Corpora('data')
```

Individual corpora can be accessed as attributes, by index, or as keys:

```
>>> corpora.first
>>> corpora[0]
>>> corpora['first']
```

You can use the `interrogate()` method to search them, using the same arguments as you would for *interrogate()*.

Interrogating these objects often returns an *corpkit.interrogation.Interrodict* object, which models a collection of DataFrames.

Editing can be performed with *edit()*. The editor will iterate over each DataFrame in turn, generally returning another `Interrodict`.

Note: There is no `visualise()` method for Interrodict objects.

---

multiindex() can turn an `Interrodict` into a *Pandas MultiIndex*:

```
>>> multiple_res.multiinedx()
```

collapse() will collapse one dimension of the `Interrodict`. You can collapse the x axis (`'x'`), the y axis (`'y'`), or the Interrodict keys (`'k'`). In the example below, an `Interrodict` is collapsed along each axis in turn.

```
>>> d = corpora.interrogate({F: 'compound', GL: r'^risk'}, show=L)
>>> d.keys()
    ['CHT', 'WAP', 'WSJ']
>>> d['CHT'].results
    ....  health  cancer  security  credit  flight  safety  heart
    1987      87      25        28      13       7       6      4
    1988      72      24        20      15       7       4      9
    1989     137      61        23      10       5       5      6
>>> d.collapse(axis=Y).results
    ...  health  cancer  credit  security
    CHT    3174    1156     566       697
    WAP    2799     933     582      1127
    WSJ    1812     680    2009       537
>>> d.collapse(axis=X).results
    ...  1987  1988  1989
    CHT   384   328   464
    WAP   389   355   435
    WSJ   428   410   473
>>> d.collapse(axis=K).results
    ...   health  cancer  credit  security
    1987     282     127      65        93
    1988     277     100      70       107
    1989     379     253      83        91
```

topwords() quickly shows the top results from every interrogation in the `Interrodict`.

```
>>> data.topwords(n=5)
```

Output:

```
TBT             %   UST             %   WAP             %   WSJ             %
health      25.70   health      15.25   health      19.64   credit       9.22
security     6.48   cancer      10.85   security     7.91   health       8.31
cancer       6.19   heart        6.31   cancer       6.55   downside     5.46
flight       4.45   breast       4.29   credit       4.08   inflation    3.37
safety       3.49   security     3.94   safety       3.26   cancer       3.12
```

6.3 Using the GUI

`corpkit` is also designed to work as a GUI. It can be started in `bash` with:

```
$ python -m corpkit.gui
```

The GUI can understand any projects you have defined. If you open it, you can simply select your project via `Open Project` and resume work in a graphical environment.

CHAPTER 7

About

I'm Daniel McDonald (@interro_gator), a linguistics PhD student and Research Fellow at the University of Melbourne, though currently visiting Saarland Uni, Germany. *corpkit* grew organically out of the code I had developed to make sense of the data I encountered in my research projects. I made a basic command line interface for interrogating, editing and plotting corpora, which eventually turned into a GUI and this fancy class-based Python module. Pull requests are more than welcome!

CHAPTER 8

Corpus classes

Much of *corpkit*'s functionality comes from the ability to work with `Corpus` and `Corpus`-like objects, which have methods for parsing, tokenising, interrogating and concordancing.

8.1 *Corpus*

class `corpkit.corpus.`**`Corpus`**(*path*, ***kwargs*)

Bases: `object`

A class representing a linguistic text corpus, which contains files, optionally within subcorpus folders.

Methods for concordancing, interrogating, getting general stats, getting behaviour of particular word, etc.

`document`

Return the parsed XML of a parsed file

`read`(***kwargs*)

Read file data. If data is pickled, unpickle first

Returns str/unpickled data

`subcorpora`

A list-like object containing a corpus' subcorpora.

Example

```
>>> corpus.subcorpora
<corpkit.corpus.Datalist instance: 12 items>
```

`speakerlist`

Lazy-loaded data.

`files`

A list-like object containing the files in a folder.

Example

```
>>> corpus.subcorpora[0].files
<corpkit.corpus.Datalist instance: 240 items>
```

`features`

Generate and show basic stats from the corpus, including number of sentences, clauses, process types, etc.

Example

```
>>> corpus.features
    ..  Characters  Tokens  Words  Closed class words  Open class words  Clauses
    01       26873    8513   7308                4809              3704     2212
    02       25844    7933   6920                4313              3620     2270
```

```
03       18376    5683   4877                3067             2616     1640
04       20066    6354   5366                3587             2767     1775
```

wordclasses
Lazy-loaded data.

postags
Lazy-loaded data.

configurations(*search*, ***kwargs*)
Get the overall behaviour of tokens or lemmas matching a regular expression. The search below makes DataFrames containing the most common subjects, objects, modifiers (etc.) of 'see':

Parameters **search** – Similar to *search* in the *interrogate()* /

concordance() methods. 'W/L keys match word or lemma; 'F: key specifies semantic role (*'participant'*, *'process'* or *'modifier'*. If *F* not specified, each role will be searched for. :type search: dict

Example

```
>>> see = corpus.configurations({L: 'see', F: 'process'}, show = L)
>>> see.has_subject.results.sum()
    i           452
    it          227
    you         162
    we          111
    he           94
```

Returns *corpkit.interrogation.Interrodict*

interrogate(*search*, **args*, ***kwargs*)
Interrogate a corpus of texts for a lexicogrammatical phenomenon.

This method iterates over the files/folders in a corpus, searching the texts, and returning a *corpkit.interrogation.Interrogation* object containing the results. The main options are *search*, where you specify search criteria, and *show*, where you specify what you want to appear in the output.

Example

```
>>> corpus = Corpus('data/conversations-parsed')
### show lemma form of nouns ending in 'ing'
>>> q = {W: r'ing$', P: r'^N'}
>>> data = corpus.interrogate(q, show = L)
>>> data.results
    ..  something  anything  thing  feeling  everything  nothing  morning
    01         14        11     12        1           6        0        1
    02         10        20      4        4           8        3        0
    03         14         5      5        3           1        0        0
    ...                                                                ...
```

Parameters **search** (*str or dict. dict is used when you have multiple criteria.*) – What the query should be matching. - t: tree - w: word - l: lemma - p: pos - f: function - g/gw: governor - gl: governor's lemma form - gp: governor's pos - gf: governor's function - d/dependent - dl: dependent's lemma form - dp: dependent's pos - df: dependent's function - i/index - n/ngrams (deprecated, use `show`) - s/general stats

Keys are what to search as *str*, and values are the criteria, which is a Tregex query, a regex, or a list of words to match. Therefore, the two syntaxes below do the same thing:

Example

```
>>> corpus.interrogate(T, r'/NN.?/')
>>> corpus.interrogate({T: r'/NN.?/'})
```

Parameters

- **searchmode** (str – *'any'/'all'*) – Return results matching any/all criteria
- **exclude** (dict – *{L: 'be'}*) – The inverse of *search*, removing results from search
- **excludemode** (str – *'any'/'all'*) – Exclude results matching any/all criteria
- **query** – A search query for the interrogation. This is only used

when *search* is a string, or when multiprocessing. If *search* is a *dict*, the query/queries are stored there as the values instead. When multiprocessing, the following is possible:

Example

```
>>> {'Nouns': r'/NN.?/', 'Verbs': r'/VB.?/'}
### return an :class:`corpkit.interrogation.Interrodict` object:
>>> corpus.interrogate(T, q)
### return an :class:`corpkit.interrogation.Interrogation` object:
>>> corpus.interrogate(T, q, show = C)
```

Parameters **show** – What to output. If multiple strings are passed in as a `list`, results

will be colon-separated, in the suppled order. If you want to show ngrams, you can't have multiple values. Possible values are the same as those for `search`, plus:

•a/distance from root

•n/ngram

•nl/ngram lemma

•np/ngram POS

•npl/ngram wordclass

Parameters **lemmatise** – Force lemmatisation on results. Deprecated:

instead, output a lemma form with the *show* argument :type lemmatise: bool

Parameters **lemmatag** – Explicitly pass a pos to lemmatiser (generally when data is unparsed),

or when tag cannot be recovered from Tregex query :type lemmatag: False/'n'/'v'/'a'/'r'

Parameters

- **spelling** (*False/'US'/'UK'*) – Convert all to U.S. or U.K. English
- **dep_type** (*str -- 'basic-dependencies'/'a',*) – The kind of Stanford CoreNLP dependency parses you want to use

'collapsed-dependencies'/'b', 'collapsed-ccprocessed-dependencies'/'c'

Parameters

- **save** (*str*) – Save result as pickle to *saved_interrogations/<save>* on completion
- **gramsize** (*int*) – size of n-grams (default 2)
- **split_contractions** (*bool*) – make *"don't"* et al into two tokens
- **multiprocess** (*int / bool (to determine automatically)*) – how many parallel processes to run
- **files_as_subcorpora** (*bool*) – treat each file as a subcorpus
- **do_concordancing** (*bool/'only'*) – Concordance while interrogating, store as *.concordance* attribute
- **maxconc** (*int*) – Maximum number of concordance lines

Returns A *corpkit.interrogation.Interrogation* object, with

.query, *.results*, *.totals* attributes. If multiprocessing is invoked, result may be a *corpkit.interrogation.Interrodict* containing corpus names, queries or speakers as keys.

parse(*corenlppath=False*, *operations=False*, *copula_head=True*, *speaker_segmentation=False*, *memory_mb=False*, *multiprocess=False*, *split_texts=400*, **args*, ***kwargs*)

Parse an unparsed corpus, saving to disk

Parameters **corenlppath** – folder containing corenlp jar files (use if *corpkit* can't find

it automatically) :type corenlppath: str

Parameters

- **operations** (*str*) – which kinds of annotations to do
- **speaker_segmentation** – add speaker name to parser output if your corpus is

script-like :type speaker_segmentation: bool

Parameters

- **memory_mb** (*int*) – Amount of memory in MB for parser
- **copula_head** (*bool*) – Make copula head in dependency parse
- **multiprocess** (*int*) – Split parsing across n cores (for high-performance computers)

Example

```
>>> parsed = corpus.parse(speaker_segmentation = True)
>>> parsed
<corpkit.corpus.Corpus instance: speeches-parsed; 9 subcorpora>
```

Returns The newly created *corpkit.corpus.Corpus*

tokenise(**args*, ***kwargs*)

Tokenise a plaintext corpus, saving to disk

Parameters **nltk_data_path** (*str*) – path to tokeniser if not found automatically

Example

```
>>> tok = corpus.tokenise()
>>> tok
<corpkit.corpus.Corpus instance: speeches-tokenised; 9 subcorpora>
```

Returns The newly created *corpkit.corpus.Corpus*

concordance(**args*, ***kwargs*)

A concordance method for Tregex queries, CoreNLP dependencies, tokenised data or plaintext.

Example

```
>>> wv = ['want', 'need', 'feel', 'desire']
>>> corpus.concordance({L: wv, F: 'root'})
   0   01  1-01.txt.xml                But , so I  feel     like i do that for w
   1   01  1-01.txt.xml                       I  felt     a little like oh , i
   2   01  1-01.txt.xml   he 's a difficult man I  feel     like his work ethic
   3   01  1-01.txt.xml                    So I  felt     like i recognized li
   ...                                                                    ...
```

Arguments are the same as `interrogate()`, plus:

Parameters **only_format_match** – if True, left and right window will just be words, regardless of

what is in `show` :type only_format_match: bool

Parameters **only_unique** (*bool*) – only unique lines

Returns A *corpkit.interrogation.Concordance* instance

interroplot(*search*, ***kwargs*)

Interrogate, relativise, then plot, with very little customisability. A demo function.

Example

```
>>> corpus.interroplot(r'/NN.?/ >># NP')
<matplotlib figure>
```

Parameters

- **search** (*dict*) – search as per *interrogate()*
- **kwargs** (*keyword arguments*) – extra arguments to pass to `visualise()`

Returns None (but show a plot)

save(*savename=False*, ***kwargs*)

Save corpus class to file

```
>>> corpus.save(filename)
```

Parameters **savename** (*str*) – name for the file

Returns None

8.2 *Corpora*

class `corpkit.corpus.`**Corpora**(*data=False*, ***kwargs*)

Bases: *corpkit.corpus.Datalist*

Models a collection of Corpus objects. Methods are available for interrogating and plotting the entire collection. This is the highest level of abstraction available.

Parameters **data** (*str* (path containing corpora), *list* (of corpus paths/Corpus) – Corpora to model

objects) *corpkit.corpus.Corpus* objects)

features

Generate and show basic stats from the corpus, including number of sentences, clauses, process types, etc.

Example

```
>>> corpus.features
    ..  Characters  Tokens  Words  Closed class words  Open class words  Clauses
    01       26873    8513   7308                4809              3704     2212
    02       25844    7933   6920                4313              3620     2270
    03       18376    5683   4877                3067              2616     1640
    04       20066    6354   5366                3587              2767     1775
```

postags

Lazy-loaded data.

wordclasses

Lazy-loaded data.

8.3 *Subcorpus*

class corpkit.corpus.**Subcorpus**(*path*, *datatype*)
Bases: *corpkit.corpus.Corpus*

Model a subcorpus, containing files but no subdirectories.

Methods for interrogating, concordancing and configurations are the same as *corpkit.corpus.Corpus*.

8.4 *File*

class corpkit.corpus.**File**(*path*, *dirname*, *datatype*)
Bases: *corpkit.corpus.Corpus*

Models a corpus file for reading, interrogating, concordancing

document
Return the parsed XML of a parsed file

read(***kwargs*)
Read file data. If data is pickled, unpickle first

Returns str/unpickled data

8.5 *Datalist*

class corpkit.corpus.**Datalist**(*data*)
Bases: object

A list-like object containing subcorpora or corpus files.

Objects can be accessed as attributes, dict keys or by indexing/slicing.

Methods for interrogating, concordancing and getting configurations are the same as for *corpkit.corpus.Corpus*

interrogate(**args*, ***kwargs*)
Interrogate the corpus using *interrogate()*

concordance(**args*, ***kwargs*)
Concordance the corpus using *concordance()*

configurations(*search*, ***kwargs*)
Get a configuration using *configurations()*

CHAPTER 9

Interrogation classes

Once you have searched a `Corpus` object, you'll want to be able to edit, visualise and store results. Remember that upon importing *corpkit*, any `pandas.DataFrame` or `pandas.Series` object is monkey-patched with `save`, `edit` and `visualise` methods.

9.1 *Interrogation*

class `corpkit.interrogation.`**`Interrogation`**(*results=None*, *totals=None*, *query=None*, *concordance=None*)

Bases: `object`

Stores results of a corpus interrogation, before or after editing. The main attribute, *`results`*, is a Pandas object, which can be edited or plotted.

`results` = **None**
pandas *DataFrame* containing counts for each subcorpus

`totals` = **None**
pandas *Series* containing summed results

`query` = **None**
dict containing values that generated the result

`concordance` = **None**
pandas *DataFrame* containing concordance lines, if concordance lines were requested.

`edit`(**args*, ***kwargs*)
Manipulate results of interrogations.

There are a few overall kinds of edit, most of which can be combined into a single function call. It's useful to keep in mind that many are basic wrappers around *pandas* operations—if you're comfortable with *pandas* syntax, it may be faster at times to use its syntax instead.

Basic mathematical operations

First, you can do basic maths on results, optionally passing in some data to serve as the denominator. Very commonly, you'll want to get relative frequencies:

Example

```
>>> data = corpus.interrogate({W: r'^t'})
>>> rel = data.edit('%', SELF)
>>> rel.results
    ..    to  that   the  then ...   toilet  tolerant  tolerate  ton
    01 18.50 14.65 14.44  6.20 ...     0.00      0.00      0.11 0.00
    02 24.10 14.34 13.73  8.80 ...     0.00      0.00      0.00 0.00
    03 17.31 18.01  9.97  7.62 ...     0.00      0.00      0.00 0.00
```

For the operation, there are a number of possible values, each of which is to be passed in as a *str*:

+, -, /, *, %: self explanatory

k: calculate keywords

a: get distance metric

SELF is a very useful shorthand denominator. When used, all editing is performed on the data. The totals are then extracted from the edited data, and used as denominator. If this is not the desired behaviour, however, a more specific *interrogation.results* or *interrogation.totals* attribute can be used.

In the example above, *SELF* (or *'self'*) is equivalent to:

Example

```
>>> rel = data.edit('%', data.totals)
```

Keeping and skipping data

There are four keyword arguments that can be used to keep or skip rows or columns in the data:

•*just_entries*

•*just_subcorpora*

•*skip_entries*

•*skip_subcorpora*

Each can accept different input types:

•*str*: treated as regular expression to match

•*list*:

–of integers: indices to match

–of strings: entries/subcorpora to match

Example

```
>>> data.edit(just_entries=r'^fr',
...           skip_entries=['free','freedom'],
...           skip_subcorpora=r'[0-9]')
```

Merging data

There are also keyword arguments for merging entries and subcorpora:

•*merge_entries*

•*merge_subcorpora*

These take a *dict*, with the new name as key and the criteria as value. The criteria can be a str (regex) or wordlist.

Example

```
>>> from dictionaries.wordlists import wordlists
>>> mer = {'Articles': ['the', 'an', 'a'], 'Modals': wordlists.modals}
>>> data.edit(merge_entries=mer)
```

Sorting

The *sort_by* keyword argument takes a *str*, which represents the way the result columns should be ordered.

•*increase*: highest to lowest slope value

•*decrease*: lowest to highest slope value

•*turbulent*: most change in y axis values

•*static*: least change in y axis values

•*total/most*: largest number first

•*infreq/least*: smallest number first

•*name*: alphabetically

Example

```
>>> data.edit(sort_by='increase')
```

Editing text

Column labels, corresponding to individual interrogation results, can also be edited with *replace_names*.

Parameters **replace_names** (*str/list of tuples/dict*) – Edit result names, then merge duplicate entries

If *replace_names* is a string, it is treated as a regex to delete from each name. If *replace_names* is a dict, the value is the regex, and the key is the replacement text. Using a list of tuples in the form *(find, replacement)* allows duplicate substitution values.

Example

```
>>> data.edit(replace_names={r'object': r'[di]obj'})
```

Parameters **replace_subcorpus_names** – Edit subcorpus names, then merge duplicates.

The same as *replace_names*, but on the other axis. :type replace_subcorpus_names: *str/list of tuples/dict*

Other options

There are many other miscellaneous options.

Parameters

- **keep_stats** (*bool*) – Keep/drop stats values from dataframe after sorting
- **keep_top** (*int*) – After sorting, remove all but the top *keep_top* results
- **just_totals** (*bool*) – Sum each column and work with sums
- **threshold** – When using results list as dataframe 2, drop values

occurring fewer than n times. If not keywording, you can use:

'high': *denominator total / 2500*

'medium': *denominator total / 5000*

'low': *denominator total / 10000*

If keywording, there are smaller default thresholds

Parameters **span_subcorpora** – If subcorpora are numerically named, span all

from *int* to *int2*, inclusive :type span_subcorpora: *tuple – (int, int2)*

Parameters

- **projection** (tuple – *(subcorpus_name, n)*) – a to multiply results in subcorpus by n

- **remove_above_p** (*bool*) – Delete any result over *p*
- **p** (*float*) – set the p value
- **revert_year** – When doing linear regression on years, turn annual

subcorpora into 1, 2 ... :type revert_year: *bool*

Parameters

- **print_info** (*bool*) – Print stuff to console showing what's being edited
- **spelling** (*str – 'US'/'UK'*) – Convert/normalise spelling:

Keywording options

If the operation is *k*, you're calculating keywords. In this case, some other keyword arguments have an effect:

Parameters **keyword_measure** – what measure to use to calculate keywords:

ll: log-likelihood 'pd': percentage difference

type keyword_measure: *str*

Parameters **selfdrop** – When keywording, try to remove target corpus from

reference corpus :type selfdrop: *bool*

Parameters **calc_all** – When keywording, calculate words that appear in either

corpus :type calc_all: bool

Returns *corpkit.interrogation.Interrogation*

visualise(*title=''*, *x_label=None*, *y_label=None*, *style='ggplot'*, *figsize=(8, 4)*, *save=False*, *legend_pos='best'*, *reverse_legend='guess'*, *num_to_plot=7*, *tex='try'*, *colours='Accent'*, *cumulative=False*, *pie_legend=True*, *rot=False*, *partial_pie=False*, *show_totals=False*, *transparent=False*, *output_format='png'*, *interactive=False*, *black_and_white=False*, *show_p_val=False*, *indices=False*, *transpose=False*, ***kwargs*)

Visualise corpus interrogations using *matplotlib*.

Example

```
>>> data.visualise('An example plot', kind='bar', save=True)
<matplotlib figure>
```

Parameters

- **title** (*str*) – A title for the plot
- **x_label** (*str*) – A label for the x axis
- **y_label** (*str*) – A label for the y axis
- **kind** (*str ('line'/'bar'/'barh'/'pie'/'area')*) – The kind of chart to make
- **style** (*str ('ggplot'/'bmh'/'fivethirtyeight'/'seaborn-talk'/etc)*) – Visual theme of plot
- **figsize** (*tuple (int, int)*) – Size of plot
- **save** (*bool/str*) – If bool, save with *title* as name; if str, use str as name
- **legend_pos** (*str ('upper right'/'outside right'/etc)*) – Where to place legend
- **reverse_legend** (*bool*) – Reverse the order of the legend
- **num_to_plot** (*int/'all'*) – How many columns to plot

- **tex** (*bool*) – Use TeX to draw plot text
- **colours** (*str*) – Colourmap for lines/bars/slices
- **cumulative** (*bool*) – Plot values cumulatively
- **pie_legend** (*bool*) – Show a legend for pie chart
- **partial_pie** (*bool*) – Allow plotting of pie slices only
- **show_totals** (*str -- 'legend'/'plot'/'both'*) – Print sums in plot where possible
- **transparent** (*bool*) – Transparent .png background
- **output_format** (*str -- 'png'/'pdf'*) – File format for saved image
- **black_and_white** (*bool*) – Create black and white line styles
- **show_p_val** (*bool*) – Attempt to print p values in legend if contained in df
- **indices** (*bool*) – To use when plotting "distance from root"
- **stacked** (*str*) – When making bar chart, stack bars on top of one another
- **filled** (*str*) – For area and bar charts, make every column sum to 100
- **legend** (*bool*) – Show a legend
- **rot** (*int*) – Rotate x axis ticks by *rot* degrees
- **subplots** (*bool*) – Plot each column separately
- **layout** (*tuple -- (int, int)*) – Grid shape to use when *subplots* is True
- **interactive** (*list -- [1, 2, 3]*) – Experimental interactive options

Returns matplotlib figure

save(*savename*, *savedir='saved_interrogations'*, ***kwargs*)

Save an interrogation as pickle to `savedir`.

Example

```
>>> o = corpus.interrogate(W, 'any')
### create ./saved_interrogations/savename.p
>>> o.save('savename')
```

Parameters

- **savename** (*str*) – A name for the saved file
- **savedir** (*str*) – Relative path to directory in which to save file
- **print_info** (*bool*) – Show/hide stdout

Returns None

quickview(*n=25*)

view top n results as painlessly as possible.

Example

```
>>> data.quickview(n=5)
    0: to    (n=2227)
    1: that  (n=2026)
    2: the   (n=1302)
    3: then  (n=857)
    4: think (n=676)
```

Parameters **n** (*int*) – Show top *n* results

Returns None

multiindex(*indexnames=None*)

Create a *pandas.MultiIndex* object from slash-separated results.

Example

```
>>> data = corpus.interrogate({W: 'st$'}, show=[L, F])
>>> data.results
    ..  just/advmod  almost/advmod  last/amod
    01           79             12          6
    02          105              6          7
    03           86             10          1
>>> data.multiindex().results
    Lemma        just almost last first   most
    Function  advmod advmod amod  amod advmod
    0             79     12    6     2      3
    1            105      6    7     1      3
    2             86     10    1     3      0
```

Parameters **indexnames** (*list of strings*) – provide custom names for the new index, or leave blank to guess.

Returns *corpkit.interrogation.Interrogation*, with

pandas.MultiIndex as *results* attribute

topwords(*datatype='n'*, *n=10*, *df=False*, *sort=True*, *precision=2*)

Show top n results in each corpus alongside absolute or relative frequencies.

Parameters

- **datatype** (*str (n/k/%)*) – show abs/rel frequencies, or keyness
- **n** (*int*) – number of result to show
- **df** (*bool*) – return a DataFrame
- **sort** (*bool*) – Sort results, or show as is
- **precision** (*int*) – float precision to show

Example

```
>>> data.topwords(n=5)
    1987           %   1988           %   1989           %   1990           %
    health     25.70   health     15.25   health     19.64   credit      9.22
    security    6.48   cancer     10.85   security    7.91   health      8.31
    cancer      6.19   heart       6.31   cancer      6.55   downside    5.46
    flight      4.45   breast      4.29   credit      4.08   inflation   3.37
    safety      3.49   security    3.94   safety      3.26   cancer      3.12
```

Returns None

9.2 *Interrodict*

class corpkit.interrogation.**Interrodict**(*data*)

Bases: collections.OrderedDict

A class for interrogations that do not fit in a single-indexed DataFrame.

Individual interrogations can be looked up via dict keys, indexes or attributes:

Example

```
>>> out_data['WSJ'].results
>>> out_data.WSJ.results
>>> out_data[3].results
```

Methods for saving, editing, etc. are similar to `corpkit.corpus.Interrogation`. Additional methods are available for collapsing into single (multiindexed) DataFrames.

edit (**args*, ***kwargs*)
Edit each value with *edit()*.

See *edit()* for possible arguments.

Returns A *corpkit.interrogation.Interrodict*

multiindex (*indexnames=None*)
Create a *pandas.MultiIndex* version of results.

Example

```
>>> d = corpora.interrogate({F: 'compound', GL: '^risk'}, show=L)
>>> d.keys()
    ['CHT', 'WAP', 'WSJ']
>>> d['CHT'].results
    ....  health  cancer  security  credit  flight  safety  heart
    1987      87      25        28      13       7       6      4
    1988      72      24        20      15       7       4      9
    1989     137      61        23      10       5       5      6
>>> d.multiindex().results
    ...                health  cancer  credit  security  downside
    Corpus Subcorpus
    CHT    1987            87      25      13        28        20
           1988            72      24      15        20        12
           1989           137      61      10        23        10
    WAP    1987            83      44       8        44        10
           1988            83      27      13        40         6
           1989            95      77      18        25        12
    WSJ    1987            52      27      33         4        21
           1988            39      11      37         9        22
           1989            55      47      43         9        24
```

Returns A *corpkit.interrogation.Interrogation*

save (*savename*, *savedir='saved_interrogations'*, ***kwargs*)
Save an interrogation as pickle to *savedir*.

Parameters

- **savename** (*str*) – A name for the saved file
- **savedir** (*str*) – Relative path to directory in which to save file
- **print_info** (*bool*) – Show/hide stdout

Example

```
>>> o = corpus.interrogate(W, 'any')
### create ``saved_interrogations/savename.p``
>>> o.save('savename')
```

Returns None

collapse (*axis='y'*)
Collapse Interrodict on an axis or along interrogation name.

Parameters **axis** (*str: x/y/n*) – collapse along x, y or name axis

Example

```
>>> d = corpora.interrogate({F: 'compound', GL: r'^risk'}, show=L)
>>> d.keys()
    ['CHT', 'WAP', 'WSJ']
>>> d['CHT'].results
    ....  health  cancer  security  credit  flight  safety  heart
    1987      87      25        28      13       7       6      4
```

```
            1988      72      24        20      15       7       4       9
            1989     137      61        23      10       5       5       6
        >>> d.collapse().results
            ...  health  cancer  credit  security
            CHT    3174    1156     566       697
            WAP    2799     933     582      1127
            WSJ    1812     680    2009       537
        >>> d.collapse(axis='x').results
            ...  1987  1988  1989
            CHT   384   328   464
            WAP   389   355   435
            WSJ   428   410   473
        >>> d.collapse(axis='key').results
            ...   health  cancer  credit  security
            1987     282     127      65        93
            1988     277     100      70       107
            1989     379     253      83        91
```

Returns A *corpkit.interrogation.Interrogation*

topwords (*datatype='n'*, *n=10*, *df=False*, *sort=True*, *precision=2*)
Show top n results in each corpus alongside absolute or relative frequencies.

Parameters

- **datatype** (*str (n/k/%)*) – show abs/rel frequencies, or keyness
- **n** (*int*) – number of result to show
- **df** (*bool*) – return a DataFrame
- **sort** (*bool*) – Sort results, or show as is
- **precision** (*int*) – float precision to show

Example

```
        >>> data.topwords(n=5)
            TBT             %   UST             %   WAP             %   WSJ             %
            health      25.70   health      15.25   health      19.64   credit       9.22
            security     6.48   cancer      10.85   security     7.91   health       8.31
            cancer       6.19   heart        6.31   cancer       6.55   downside     5.46
            flight       4.45   breast       4.29   credit       4.08   inflation    3.37
            safety       3.49   security     3.94   safety       3.26   cancer       3.12
```

Returns None

get_totals()
Helper function to concatenate all totals

9.3 Concordance

class corpkit.interrogation.**Concordance**(*data*)
Bases: pandas.core.frame.DataFrame

A class for concordance lines, with methods for saving, formatting and editing.

format (*kind='string'*, *n=100*, *window=35*, *columns='all'*, ***kwargs*)
Print concordance lines nicely, to string, LaTeX or CSV

Parameters

- **kind** (*str*) – output format: *string/latex/csv*
- **n** (int/*'all'*) – Print first *n* lines only
- **window** (*int*) – how many characters to show to left and right

- **columns** (*list*) – which columns to show

Example

```
>>> lines = corpus.concordance({T: r'/NN.?/ >># NP'}, show=L)
### show 25 characters either side, 4 lines, just text columns
>>> lines.format(window=25, n=4, columns=[L,M,R])
    0                  we 're in  tucson     , then up north to flagst
    1  e 're in tucson , then up  north      to flagstaff , then we we
    2  tucson , then up north to  flagstaff  , then we went through th
    3   through the grand canyon  area       and then phoenix and i sp
```

Returns None

calculate()
Make new Interrogation object from (modified) concordance lines

shuffle(*inplace=False*)
Shuffle concordance lines

Parameters **inplace** (*bool*) – Modify current object, or create a new one

Example

```
>>> lines[:4].shuffle()
    3  01  1-01.txt.xml   through the grand canyon  area       and then phoenix and i sp
    1  01  1-01.txt.xml  e 're in tucson , then up  north      to flagstaff , then we we
    0  01  1-01.txt.xml                  we 're in  tucson     , then up north to flagst
    2  01  1-01.txt.xml  tucson , then up north to  flagstaff  , then we went through th
```

edit(**args*, ***kwargs*)
Delete or keep rows by subcorpus or by middle column text.

```
>>> skipped = conc.edit(skip_entries=r'to_?match')
```

CHAPTER 10

Functions

corpkit contains a small set of standalone functions.

10.1 *as_regex*

corpkit.other.**as_regex**(*lst*, *boundaries='w'*, *case_sensitive=False*, *inverse=False*)

Turns a wordlist into an uncompiled regular expression

Parameters

- **lst** (*list*) – A wordlist to convert
- **boundaries** (*str -- 'word'/'line'/'space'; tuple -- (leftboundary, rightboundary)*) –
- **case_sensitive** (*bool*) – Make regular expression case sensitive
- **inverse** (*bool*) – Make regular expression inverse matching

Returns regular expression as string

10.2 *load*

corpkit.other.**load**(*savename*, *loaddir='saved_interrogations'*)

Load saved data into memory:

```
>>> loaded = load('interro')
```

will load `./saved_interrogations/interro.p` as loaded

Parameters

- **savename** (*str*) – Filename with or without extension
- **loaddir** (*str*) – Relative path to the directory containg *savename*
- **only_concs** (*bool*) – Set to True if loading concordance lines

Returns loaded data

10.3 *load_all_results*

corpkit.other.**load_all_results**(*data_dir='saved_interrogations'*, ***kwargs*)

Load every saved interrogation in data_dir into a dict:

```
>>> r = load_all_results()
```

Parameters **data_dir** (*str*) – path to saved data

Returns dict with filenames as keys

10.4 new_project

corpkit.other.**new_project**(*name*, *loc='.'*, ***kwargs*)

Make a new project in loc.

Parameters

- **name** (*str*) – A name for the project
- **loc** (*str*) – Relative path to directory in which project will be made

Returns None

CHAPTER 11

Wordlists

11.1 Closed class word types

Various wordlists, mostly for subtypes of closed class words

corpkit.dictionaries.wordlists.**wordlists = wordlists(pronouns=[u'all', u'another', u'any', u'anybody', u'an**
wordlists(pronouns, conjunctions, articles, determiners, prepositions, connectors, modals, closedclass, stopwords, titles, whpro)

11.2 Systemic functional process types

Inflected verbforms for systemic process types.

corpkit.dictionaries.process_types.**processes**

11.3 Stopwords

A list of arbitrary stopwords.

corpkit.dictionaries.stopwords.**stopwords**

11.4 Systemic/dependency label conversion

Systemic-functional to dependency role translation.

corpkit.dictionaries.roles.**roles = roles(actor=['agent', 'agent', 'csubj', 'nsubj'], adjunct=['(prep|nmod)(_|:).***
roles(actor, adjunct, auxiliary, circumstance, classifier, complement, deictic, epithet, event, existential, finite, goal, modal, modifier, numerative, participant, participant1, participant2, polarity, postmodifier, predicator, premodifier, process, qualifier, subject, textual, thing)

11.5 BNC reference corpus

BNC word frequency list.

corpkit.dictionaries.bnc.**bnc**

11.6 Spelling conversion

A dict with U.S. English spellings as keys, U.K. spellings as values.

corpkit.dictionaries.word_transforms.**usa_convert**

Cite

If you'd like to cite *corpkit*, you can use:

```
McDonald, D. (2015). corpkit: a toolkit for corpus linguistics. Retrieved from
https://www.github.com/interrogator/corpkit. DOI: http://doi.org/10.5281/zenodo.28361
```

T

V

W