Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

Contributions from 2018

Edited by

Johann-Mattis List and Tiago Tresoldi

Jena, Max-Planck Institute for the Science of Human History

Contents

Introduction (Johann-Mattis List) 3
Extracting translation data from the Wiktionary project (Tiago Tresoldi) 4
Extracting sublists from a wordlist with LingPy and Concepticon (Johann-Mattis List) 8
Cooking with CLICS (Johann-Mattis List) 14
Representing structural data in CLDF (Johann-Mattis List) 18
A fast implementation of the Consonant Class Matching method for automatic cognate detection in LingPy (Johann-Mattis List) . . 22
Enhancing morphological annotation for internal langauge comparison (Nathanael E. Schweikhard) 30
Inferring consonant clusters from CLICS data with LingPy (Johann-Mattis List) 42
From Fieldwork to Trees 1: Data preparation (Gereon A. Kaiping) . . 50
Semantic promiscuity as a factor of productivity in word formation (Nathanael E. Schweikhard) 56
From Fieldwork to Trees 2: Cognate coding (Gereon A. Kaiping) . . . 65
Merging datasets with LingPy and the CLDF curation framework (Johann-Mattis List) 71

Introduction

By comparing the languages of the world, we gain invaluable insights into human prehistory, predating the appearance of written records by thousands of years. The traditional methods for language comparison are based on manual data inspection. With more and more data available, they reach their practical limits. Computer applications, however, are not capable of replacing experts' experience and intuition. In a situation where computers cannot replace experts and experts do not have enough time to analyse the massive amounts of data, a new framework, neither completely computer-driven, nor ignorant of the help computers provide, becomes urgent.

The weblog "Computer-Assisted Language Comparison in Practice", published on the Hypotheses platform for scientific blogging, offers tutorials and discussion notes on computer-assisted approaches to the history and diversity of languages. A substantial part of its content is contributed as part of the ERC Starting Grant "Computer-Assisted Language Comparison" (CALC, 715618), funded by the European Research Council. But on the long run, we want to make this blog a platform for everybody willing to share ideas on small or big problems involving data preparation and analysis in computer-assisted or computer-based approaches to language comparison.

This document summarizes all contributions from 2018. If you want to cite them, please follow the instructions at the end of each contribution. I express my gratitude to all contributors, who helped to make this an interesting collection of tutorials, algorithms, and initial theories related to the fields of computer-assisted language comparison.

Johann-Mattis List (Jena, November 2019)

Extracting translation data from the Wiktionary project

Tiago Tresoldi (11/06/2018)

Categories: Dataset

Tags: cross-linguistic data formats, lexical data, Wiktionary

---

Wiktionary is a project for creating a multilingual, web-based free dictionary of all words in all languages. Like its sister project Wikipedia, since its inception it has been subject to criticism both in terms of its lexicographic approaches and in terms of reliability, content, procedures, and community operation (see Lepore 2006, Fuertes-Olivera 2009, Meyer 2012). Faults have also been pointed in terms of its structure which is confusing for newcomers, with parallel and unaligned information shared among the various language dictionaries, and differences in accuracy and depth among languages. Notwithstanding, data from Wiktionary is routinely employed with successful results in natural language processing and, occasionally, in linguistic research (see Otte 2011, Schlippe 2012, Medero 2009, Li 2012), as it constitutes, by far, the largest free multilingual lexical source.

The Wikimedia Foundation, the organization managing the project, releases automatically generated "dumps" of the data for free and anonymous download. However, such files cannot be used in linguistic research without a pre-processing ("parsing") stage, as they constitute more a backup than a data release: in essence, they are XML files which enclose the textual information of the dictionary articles (pages potentially holding information for more than one word and more than one language), which are encoded in the

MediaWiki markup syntax (a context-sensitive language that is notoriously difficult to parse). Data extraction is further complicated by the fact that the rendered HTML pages include information computed by functions of general and linguistic scope only available inside an environment running the Wiktionary server, as well as by Wiktionary collaborators not always following the project's guidelines and specifications. Many projects have started to tackle such problems and the difficulties in reusing the data, including a brand new initiative by Wikidata.

As such, no standard method for extracting Wiktionary information exists, with mostly project-specific solutions. An investigation of parsing tools on GitHub revealed that two main approaches are used: parsing the XML files and manipulating the entire textual fields, or parsing the individually rendered HTML pages (fetched either from a local server or over the Internet). We decided to test a simpler approach of parsing the dumps as regular text files, reading them line by line while building an internal structured version of the information, processing lines with regular expression or simple string searching methods. The first experiment, whose results are here presented, involved extracting the parallel translations for English words found in the English Wiktionary.

The data, based on the dump of 2018-06-01, includes 2,169,063 different entries from the translation of 149,530 English words and expressions in 2,358 languages (with much variation in vocabulary size among languages: 931 languages have only one entry and German, the largest language after English, has 97,091 entries). Data is offered in a tabular textual format, and all entries include (a) a unique ID, (b) a concept ID referring to the source English word, (c) a description string with the English source and a short definition (such as "dictionary/publication that explains the meanings of an ordered list of words"), (d) a language ID from the Glottolog catalog, (e) the text of the translation as given in the Wiktionary, and (f) an extra field holding com-

plementary information, when available (such as phonetic transcription of the text, noun gender, etc.). Data is also offer in a set of files (tabular textual files, bibtex sources, and JSON metadata) following the Cross-Linguistic Data Formats (CLDF), a specification designed to allow the exchange of cross-linguistic data. The code for data extraction is available on GitHub and the data is available on Zenodo as ‘‘Parallel Translations from the English Wiktionary” (DOI: 10.5281/zenodo.1286991). Many thanks to Johann-Mattis List and to Christoph Rzymski for their help with this work.

References

Fuertes-Olivera, Pedro A. (2009). “The function theory of lexicography and electronic dictionaries: Wiktionary as a prototype of collective free multiple-language internet dictionary”. In: H. Bergenholtz, S. Nielsen, and S. Tarp (eds.), *Lexicography at a Crossroads: Dictionaries and Encyclopedias Today, Lexicographical Tools Tomorrow*. Linguistic Insights: Studies in Language and Communication 90, 99–134. Bern: Peter Lang.

Lepore, Jill (2006). “Noah’s Mark”. In: *New Yorker*, November 6 2006 Issue.

Li, Shen; Graça, João V.; Taskar, Ben (2012). “Wiki-ly supervised part-of-speech tagging” (PDF). *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Jeju Island, Korea: Association for Computational Linguistics. pp. 1389–1398.

Medero, Julie; Ostendorf, Mari (2009). “Analysis of vocabulary difficulty using wiktionary” (PDF). In: *Proc. SLaTE Workshop*.

Meyer, Christian M.; Gurevych, Iryna (2012). “Wiktionary: A new rival for expert-built lexicons? Exploring the possibilities of collaborative lexicography”. In: Granger, Sylviane; Paquot, Magali, *Electronic Lexicography*. Oxford: Oxford University Press.

Otte, Pim; Tyers, Francis M. (2011). “Rapid rule-based machine translation between Dutch and Afrikaans” (PDF). In Forcada, Mikel L.; Depraetere, Heidi; Vandeghinste, Vincent. *16th*

Annual Conference of the European Association of Machine Translation, EAMT11. Leuven, Belgium. pp. 153–160.

Schlippe, Tim; Ochs, Sebastian; Schultz, Tanja (2012). "Grapheme-to-phoneme model generation for Indo-European languages" (PDF). *Acoustics, Speech and Signal Processing (ICASSP)*. Kyoto, Japan. pp. 4801–4804.

Cite this article as: Tiago Tresoldi, "Extracting translation data from the Wiktionary project," in *Computer-Assisted Language Comparison in Practice*, 11/06/2018, https://calc.hypotheses.org/32.

Exporting sublists from a wordlist with LingPy and Concepticon

Johann-Mattis List (16/07/2018)

Categories: Code

Tags: code example, concept list, Concepticon, LingPy, sublist, Swadesh list

---

When dealing with linguistic datasets, we may often want to export only a small part of our data, for example, only vocabulary in a certain range, such as the Swadesh list of 200 items or the list of 35 items by Yakhontov (originally published in Starostin 1991) . Thanks to the `pyconcepticon` API and LingPy's built-in export functions for wordlists, this task can be done just in a few lines of code, as we will see below. If you prefer to see the raw code instead of the step-by-step explanation below, you can find a GitHub Gist here.

In order to get started, I work with a dataset that was originally published along with a paper by Kolipakam et al. (2018). This dataset can be downloaded from the supplemental material accompanying the paper, and the important file is a zip-folder called `SI_robustness_cognate_coding.zip`, from which you have to extract the file `DravLex.tsv`. I take this dataset, since it is published, it is easy to get a copy of the data, and the data has been already linked to Concepticon, although not officially in Version 1.1, but you can already receive the data from our GitHub repository, where the concept list is labelled `Kolipakam-2018-100`, following our Concepticon naming convention, which takes the first author, the year of the publication, and the number of items in order to create a stable identifier for a concept list.

To get started, you need to make sure that you have the `pyconcepticon` API installed in its most recent version. You should find all important instructions for this on the GitHub repository of the Concepticon project (see clld/concepticon-data). You also need to have LingPy installed in its current 2.6 version (ideally also make sure to take the most recent version from our GitHub repository: lingpy/lingpy). Equipped with this information and the dataset, open a terminal in the folder in which you have placed the file `DravLex.tsv`, and start by loading LingPy and the `pyconcepticon` API.

```
[IN ]: from pyconcepticon.api import Concepticon
[IN ]: from lingpy import *
```

This is of course easy, but now we will process the concept list of Kolipakam et al. and try to extract those items which we can also find in the very short list of 35 items by Yakhontov. In order to do so, you need to understand how Concepticon stores data in the Python API, and this may be somewhat confusing at the first sight, since the class hierarchy is created in such a way that it directly reflects the online version of the Concepticon and makes extensive use of dictionaries (more precisely `OrderedDict`). But let's start step by step, and first, we simply load the API:

```
[IN ]: CNC = Concepticon()
```

And now we load the concept lists by Yakhontov and Kolipakam:

```
[IN ]: yakhontov = CNC.conceptlists['Yakhontov-1991-35']
[IN ]: kolipakam = CNC.conceptlists['Kolipakam-2018-100']
```

What is important here is the structure of the `Conceptlist` objects that we just loaded. They all have an attribute `concepts` , which is itself a dictionary with the identifier of a given concept as key, and a `Concept` object as value. The `Concept` object itself has again different attributes, and the most important attributes for us are the `concepticon_id` and the `english` entry. We need the `english` attribute of the data by Kolipakam et al. to determine the Concepticon identifiers in the dataset, since the dataset itself does not officially link the data to Concepticon. We use the Concepticon identifiers to compare which of the concept sets in the Dravidian data also occur in Yakhontov's list. We start by slightly modifying our two lists (`yakhontov` and `kolipakam`) first, and then extract a sublist with English glosses from them:

```
[IN ]: yakhontov = [c.concepticon_id for c in yakhontov.
   concepts.values()]
[IN ]: kolipakam = [(c.concepticon_id, c.english) for c in
    kolipakam.concepts.values()]
[IN ]: sublist = [english for english, concepticon_id in
   kolipakam if concepticon_id in yakhontov]
[IN ]: print('Overlap shows {0} items in common.'.format(
   len(sublist)))
[OUT]: Overlap shows 31 items in common.
```

We can now load the `DravLex.tsv` file and output it in such a way that only a subset of concepts is selected which occur in our sublist. For this, we use the `output`-method of the `Wordlist` class in LingPy, but in contrast to the normal output procedure, we specify a subset and a condition. Since the condition is evaluated internally in form of Python code passed to LingPy, this looks a bit ugly, as we define a dictionary that specifies columns that occur in the wordlist, and these columns' content is then checked against the Python code that we pass as string, but it is the fastest way to accomplish this task in LingPy:

```
[IN ]: wl = Wordlist('DravLex.tsv')
[IN ]: wl.output('tsv', filename='DravLex-sublist', subset
   =True, rows={"concept": "in "+str(sublist)})
```

That's all we have to do. In order to verify that we really exported only a part of the wordlist, we can reload it and count its basic parameters (number of languages, concepts, and words):

```
[IN ]: wl2 = Wordlist('DravLex-sublist.tsv')
[IN ]: print('{0}:{1} languages, {2}:{3} concepts, and
   {4}:{5} words'.format(wl.width, wl2.width, wl.height,
   wl2.height, len(wl), len(wl2)))
[OUT]: 20:20 languages, 100:31 concepts, and 2114:660
   words
```

Another way to achieve this goal which has the advantage of allowing you to circumvent to write the data to file and reload it is to create a new wordlist object from the original wordlist object. In order to do so, we create a dictionary with integers as key where the key 0 reflects the header of the wordlist object and the keys link to values that are a list, just as we know it from the normal wordlist objects.

```
[IN ]: D = {0: [c for c in wl.columns]}
```

We can fill this now still empty dictionary by iterating over all entries in our original wordlist and checking whether the concepts occur in our sublist:

```
[IN ]: for idx, concept in wl.iter_rows('concept'):
......     if concept in sublist:
......         D[idx] = [entry for entry in wl[idx]]
```

This dictionary can then directly be passed to the `Wordlist` class and loaded in the same way in which we would load a normal wordlist.

```
[IN ]: wl2 = Wordlist(D)
```

We can again quickly verify that this yields the same expected output.

```
[IN ]: print('{0}:{1} languages, {2}:{3} concepts, and
   {4}:{5} words'.format(wl.width, wl2.width, wl.height,
   wl2.height, len(wl), len(wl2)))
[OUT]: 20:20 languages, 100:31 concepts, and 2114:660
   words
```

Which of the methods to use depends on personal preferences and also the task at hand. It may be preferable to load a sublist on the fly and manipulate it further in LingPy, and it may be useful to save it to file, which is faster with our subset option in LingPy's output function for wordlists. When playing a bit with the conditions, many more things can be done in order to manipulate wordlist objects within Python, without having to manipulate them manually, or by writing wordlist content to other datatypes in order to handle them with additional libraries, like, for example, Pandas. The only problem is, at least in my experience, that it seems to be difficult for users to grasp the major concepts behind this practice in LingPy, as they have been developed long before tools like Pandas were common ways of manipulating arrays and tabular data.

References

Kolipakam, V., F. Jordan, M. Dunn, S. J. Greenhill, R. Bouckaert, R. Gray, and A. Verkerk (2018). "A Bayesian phylogenetic study of the Dravidian language family." *Royal Society Open Science* 5.171504. 1-17.

Starostin, S. (1991). *Altajskaja problema i proischoždenije japonskogo jazyka [The Altaic problem and the origin of the Japanese language].* Nauka: Moscow.

Cite this article as: Johann-Mattis List, "Exporting Sublists from a Wordlist with LingPy and Concepticon," in *Computer-Assisted Language Comparison in Practice*, 16/07/2018, https://calc.hypotheses.org/58.

Cooking with CLICS

Johann-Mattis List (08/08/2018)

Categories: Code, Dataset

Tags: CLICS, colexification network, Concepticon, example

Robert Forkel just published a very nice cookbook example for our CLICS database (List et al. 2018f, http://clics.clld.org), where you can find out how to manipulate the data further, apart from just installing it and running it to replicate our analyses.

This cookbook tells you how the underlying SQLITE database is structured and how you can, after installing CLICS and the respective packages, access the data to conduct studies of your own.

As a little example of what you can do with the new CLICS API, let me illustrate in this post, how we can use the old CLICS data (underlying the version 1.0 by List et al. 2014, http://clics.lingpy.org), available from here, in the new application, specifically the standalone that we provide.

In order to get started, we begin by installing the pyclics API. For this, I assume that Python3 is installed in a recent version on our system, along with `pip` the command for downloading and installing new packages, and the version-control system `git` .

In order to install the `pyclics` API, simply type the following in your terminal:

```
$ git clone https://github.com/clics/clics2
$ cd clics2
$ pip install -e .
```

Now, you can install the old CLICS data underlying version 1.0.

```
$ pip install -e git+https://github.com/clics/clics1.
   git@v1.1#egg=lexibank_clics1
```

In addition, you need to install `pyglottolog` and `pyconcepticon` , but not with `pip` , but rather as local clones (as you need to know where they are installed. So we recommend to install both packages by opening your terminal in your preferred folder (ideally the one where you installed `pyclics`).

```
$ cd ..
$ git clone https://github.com/clld/concepticon-data
$ cd concepticon-data
$ pip install -e .
$ cd ..
$ git clone https://github.com/clld/glottolog
$ pip install -e .
$ cd ..
```

Now you can load the data into your personal CLICS database by simply typing:

```
$ clics load ./concepticon-data ./glottolog
```

In order to calculate the colexification network, just type:

```
$ clics -t 2 colexification
```

In order to create a standalone application for the data which you can put on a server or browse (when using Firefox as webbrowser) even locally, type:

```
$ clics -t 2 communities
$ clics -t 3 subgraph
```

You will find the application in the path `clics2/app/` . Just click on the file `index.html` and open it, and you can see an interface that reminds of the old "look-and-feel" of CLICS.

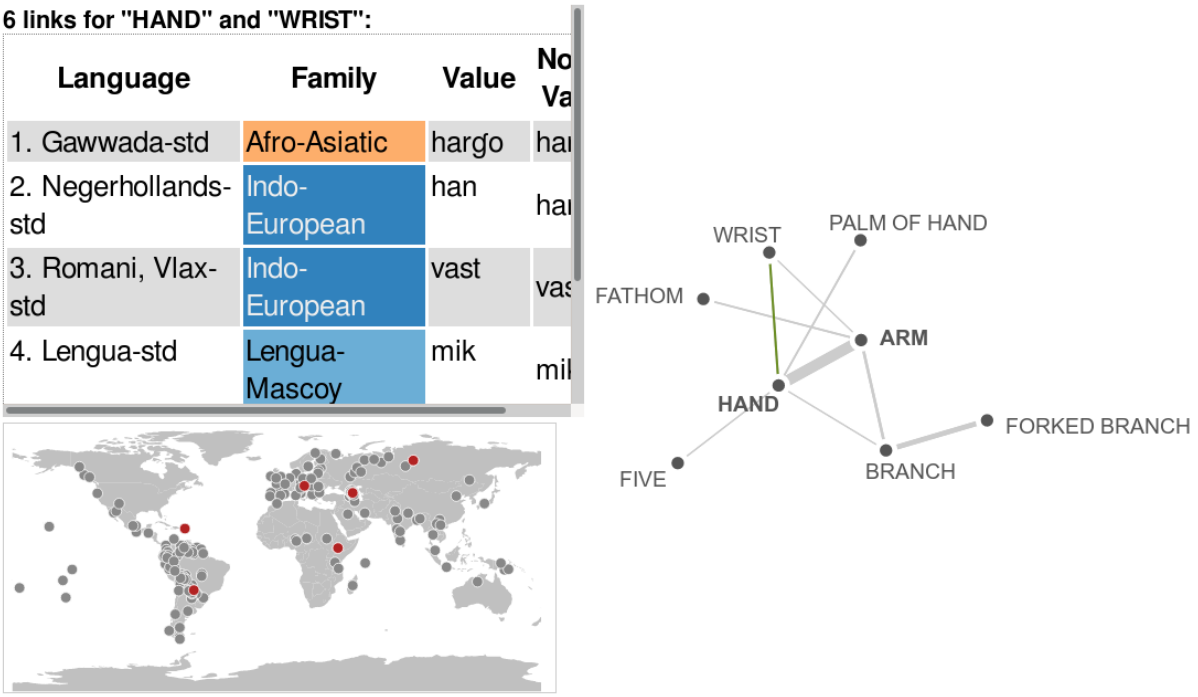

Figure 1: Local CLICS application.

If you want to have a closer look at the network without following all the code examples above, you can also directly access it at http://calc.digling.org/clics1, where we have uploaded the version that we created ourselves in order to test this example.

References

List, J.-M., T. Mayer, A. Terhalle, and M. Urban (eds.) (2014). *CLICS: Database of Cross-Linguistic Colexifications*. Version 1.0. Forschungszentrum Deutscher Sprachatlas: Marburg. http://www.webcitation.org/6ccEMrZYM.

List, J.-M., S. J. Greenhill, C. Anderson, T. Mayer, T. Tresoldi, and R. Forkel (eds.) (2018). *CLICS: Database of Cross-Linguistic Colexifications*. Max Planck Institute for the Science of Human History: Jena.

Cite this article as: Johann-Mattis List, "Cooking with CLICS," in *Computer-Assisted Language Comparison in Practice*, 08/08/2018, https://calc.hypotheses.org/384.

Representing structural data in CLDF

Johann-Mattis List (03/09/2018)

Categories: Code, Dataset

Tags: CLDF, cross-linguistic data formats, Python, structural dataset

---

The Cross-Linguistic Data Formats initiative (CLDF, https://cldf.clld.org, Forkel et al. 2018) has helped a lot in preparing the CLICS^2 database of cross-linguistic colexifications (https://clics.lingpy.org, List et al. 2018), since linking our data to Concepticon (https://concepticon.clld.or, List et al. 2016) and Glottolog (https://glottolog.org, Hammarström et al. 2018) has provided incredible help in merging the different datasets into a big comparative dataset.

CLDF, however, is not restricted to lexical data, but can also be successfully used to store structural data, although — due to the nature of structural data — it is much more difficult to compare different datasets.

In a recent publication by Szeto et al. 2018 the authors use structural data to compare different Chinese dialect varieties typologically. The data itself is provided in the paper, but unfortunately, the authors do not share it in form of text files, but list it in tables in the original publication. They also do not share the NEXUS file they created in order to analyze the data with the Neighbor-net algorithm (Bryant and Moulton 2004) implemented by the SplitsTree software package (Huson 1998).

Thanks to the help of David Morrison, who extracted the presence-absence matrix from the table in the original paper, I was now able to convert their data to the CLDF format for structural datasets. The data is available via

GitHub at cldf-datasets/szetosinitic. The folder contains both the "raw" data that I used (a KML-file with the dialect locations submitted as supplement to the paper, as well as the feature matrix extracted by David Morrison, and the parameter description as typed off by myself), as well as the CLDF data, and a script that converts the raw data into CLDF. In addition, I have added a script that converts the data to a NEXUS file that can be directly read into SplitsTree. Robert Forkel furthermore added automatic tests that will be carried out if users propose changes to the GitHub repository by making a pull request.

In order to test the code for these different conversions, you can simply clone (=download) the data with git:

```
1 $ git clone https://github.com/cldf-datasets/szetosinitic
```

Afterwards, you should install the dependencies, as listed in the file `pip-requirements.txt` (make sure you use Python3 for this task, as Python2 is not supported):

```
1 $ pip install -r pip-requirements.txt
```

Once this is done, you can test the conversion of the raw data into the CLDF data by typing:

```
1 $ python chinese.py
```

This won't give you any visual feedback, but it will in fact recreate all the CLDF data in the folder `cldf` , and it was the way I created the CLDF data in a first instance.

To receive the NEXUS file, just type:

```
$ python nexus.py
```

This will read the CLDF data that you just created and write it in NEXUS format ot the file `chinese.nex` .

To illustrate that one can use this basic procedure for more than just one dataset, I added an older and smaller dataset published by Norman (2003) which I happened to have typed off quite some time before. This dataset can be found on GitHub at cldf-datasets/normansinitic and you can follow exactly the same steps in order to convert this data as well into NEXUS format.

In the future, I hope we can provide more functionality in the CLDF package itself, so that people could use the `cldf` command that is installed as well if you install the `pycldf` package, to convert a dataset from CLDF to different formats needed for computations. However, given the diversity of formats, with very different flavors of NEXUS being required by different software packages, we should better wait a bit until we have extracted the most important use cases that can be encountered.

References

Bryant, D. and V. Moulton (2004). "Neighbor-Net. An agglomerative method for the construction of phylogenetic networks". *Molecular Biology and Evolution* 21.2. 255-265.

Forkel, R., J.-M. List, S. J. Greenhill, C. Rzymski, S. Bank, M. Cysouw, H. Hammarström, M. Haspelmath, G. Kaiping, and R. Gray (forthcoming). "Cross-Linguistic Data Formats, advancing data sharing and re-use in comparative linguistics". *Scientific Data*.

Hammarström, H., R. Forkel, and M. Haspelmath (2018). *Glottolog*. Version 3.3. Max Planck Institute for Evolutionary Anthropology: Leipzig.

Huson, D. (1998). "SplitsTree: analyzing and visualizing evolutionary data". *Bioinformatics* 14.1. 68-73.

List, J.-M., M. Cysouw, and R. Forkel (2016). "Concepticon. A resource for the linking of concept lists". In: Proceedings of the Tenth International Conference on Language Resources and Evaluation . 2393-2400.

List, J.-M., S. J. Greenhill, C. Anderson, T. Mayer, T. Tresoldi, and R. Forkel (2018): "CLICS^2. An improved database of cross-linguistic colexifications assembling lexical data with help of cross-linguistic data formats". *Linguistic Typology* 22.2. 277-306.

Norman, J. (2003). "The Chinese dialects. Phonology". In: Thurgood, G. and R. LaPolla (eds.): *The Sino-Tibetan languages*. Routledge: London and New York. 72-83.

Szeto, P., U. Ansaldo, and S. Matthews (2018). "Typological variation across Mandarin dialects: An areal perspective with a quantitative approach". *Linguistic Typology* 22.2. 233-275.

Cite this article as: Johann-Mattis List, "Representing Structural Data in CLDF," in *Computer-Assisted Language Comparison in Practice*, 03/09/2018, https://calc.hypotheses.org/445.

A fast implementation of the Consonant Class Matching method for automatic cognate detection in LingPy

Johann-Mattis List (01/10/2018)

Categories: Code

Tags: cognate detection, consonant class matching method, implementation, LexStat, LingPy, sound classes

---

LingPy's `LexStat` class for cognate detection confuses those who want to apply it, since the name of the Python class is the same as the name of one of the methods the class provides, but the class can be used for other types of cognate detection as well. I recommend all users of LingPy that they give a read to our most recent tutorial on LingPy's cognate detection method (List et al. 2018), since the three most important methods are discussed there in detail, namely the edit distance method for cognate detection, which makes use of the simple, normalized edit distance, the SCA method, based on the Sound-Class-Based Alignment algorithm (List 2014), and the LexStat method (ibid.). Applying these methods in LingPy is fairly simple and described in detail in our aforementioned tutorial. But LingPy offers an additional method for cognate detection that has the advantage of being extremely fast and thus especially suitable for exploratory data analysis of very large datasets. This method is called `turchin` in LingPy, named after the first author of a paper presenting the method (Turchin et al. 2010), but the method itself, which Turchin et al. name "Consonant Class Matching" method, goes originally back

to Dolgopolsky 1964), and has long since been implemented as a part of the STARLING software package (http://starling.rinet.ru/program.php).

The method is fairly simple. For a given wordlist with a certain number of concepts translated into a certain number of languages, all words are converted to consonant classes, following either Dolgopolsky's original proposal (Dolgopolsky 1964) or later modifications (Kassian et al. 2015). After conversion, only the first two consonants of each word are compared, and the basic rule is that if two words match in their first two consonant classes, they should be considered as cognate. Attentive readers may now immediately ask: What do you do if words don't have a consonant? To handle cases like this, the method has two specific additional rules to convert a word to its first two consonant classes. The first rule says that words starting with a vowel will be treated as if they started with a glottal stop and thus assigned the consonant class "H". The second rule, which is not often mentioned in the literature, is that, if a word consists of only one vowel, the second consonant will also be rendered as a "H".

Whether this is a good idea or not, is in fact not important. Previous tests have shown that the method in general does not yield too many false positives, but instead may miss many good cognates (List et al. 2017). That means that the method is rather *conservative*, at least as long as it is applied to words within the same concept slot. Even more important than its conservative behaviour (as linguists we always prefer false negatives over false positives), however, is that the method is extremely fast, and its complexity is linear: the amount of time we need to cluster words into cognate sets with the Consonant Class Matching method is directly proportional to the number of words in our sample. The reason for this is that the cognate-match criterion of the CCM method is transitive: if word A has the same consonant classes as word B, and word C has the same consonant classes as word B, word C must also have the same consonant classes as word A.

We can also say: the criterion by which we partition a couple of words into cognate sets is just their consonant classes themselves. The consonant classes can be directly used as the labels for the resulting clusters, since the criterion of cognate set assignment is identity in those classes. As a result, we can compute the clusters by simply computing the consonant classes per word in our data, which explains why the complexity is linear.

The problem with the `turchin` method in the `LexStat` class of LingPy is that the implementation is *not* linear in terms of complexity, since it internally uses a distance matrix for all word pairs in a slot in which words that are cognate according to the CCM criterion are given the distance 0 and the other words are given the distance 1. This distance matrix is then analysed with the UPGMA algorithm (Sokal and Michener 1958) or one of the other luster approaches offered by LingPy. While this will still be fast, since the clusters are always well-balanced, the method may easily break if you try to identify cognates across, say, 1000 languages with 200 concepts for each of them. This is not only due to the matrix-implementation in LingPy (which is not needed and was created in a time when I did not see that the CCM method has a linear solution), but also due to the complex operations and conversions that are done whenever you load a dataset into LingPy's `LexStat` class.

Writing a workaround is in fact very easy, and presenting this workaround is exactly what I want to do in this blogpost. My hope is that we manage to add this to the next official release of LingPy so that users can profit from the very fast computation of cognate sets for large dataset, without having to wait for hours until the normal LexStat approaches yield (at times disappointing) results. In the following, I will illustrate in a step-by-step guide how cognate sets can be inferred with a linear implementation of the CCM method.

To get started, we import the LingPy library.

```
from lingpy import *
```

We then load our wordlist, which can be any wordlist in LingPy or CLDF format (Forkel et al. 2018). I will be lazy in this test and simply import the data from Kessler (2001), which is available from LingPy's test suite.

```
from lingpy.tests.util import test_data
wl = Wordlist(test_data('KSL.qlc'))
```

We then make a function that converts a given tokenized sound sequence to its first two consonant classes, following the criteria mentioned above.

```
def to_ccm(tokens):
    cv = tokens2class(tokens, 'cv')
    cl = tokens2class(tokens, 'dolgo')
    dolgo = ''
    if cv[0] == 'V':
        dolgo += 'H'
    else:
        dolgo += cl[0]
    for c, v in zip(cv[1:], cl[1:]):
        if c == 'C':
            dolgo += v
    if len(dolgo) == 1:
        dolgo += 'H'
    return dolgo[:2]
```

Now we just add a new column to our wordlist, but instead of adding only consonant classes, we add the concepts as well, since we want to make sure that cognates are only assigned inside a given concept slot. For this, we use the `Wordlist.add_entries` method, which can combine the information given

in two and more columns and add it to a new column (but this application is a bit tricky and needs some practice if one wants to use it correctly).

```
wl.add_entries(
    'cog',
    'tokens,concept',
    lambda x, y : x[y[1]]+'-'+to_ccm(x[y[0]]))
```

Having calculated a new column that contains the name of the concept for each word plus its sound classes, we can conveniently use the `Wordlist.renumber` function in LingPy to convert the data in this column to integers. Note that we use the `override` keyword, since the original data already contains cognate sets in a column called `cogid`.

```
wl.renumber('cog', override=True)
```

To see whether we succeded, let us at the same time compute a little tree from the data. Don't be scared: since Kessler's data contains unrelated languages, and we use a very crude algorithm for cognate detection, the result will show a tree for the languages, even if there is no evidence for their relatedness.

```
wl.calculate('tree', ref='cogid')
```

Last not least, we print the tree in Ascii-Art on the terminal:

```
print(wl.tree.asciiArt())
```

And the result will look like this:

```
                    /-Navajo
          /edge.0--|
         |          \-Turkish
-root----|
         |          /-Hawaiian
         |         |
          \edge.4--|                    /-English
                   |          /edge.1--|
                   |         |          \-German
                    \edge.3--|
                             |          /-Albanian
                              \edge.2--|
                                        \-French
```

Obviously, this method should not be used to avoid that experts code or check the cognates, but it may turn out to be very useful for studies that want to quickly explore the signal in a certain dataset, and check whether it is worthwhile to compare a given set of languages further. In the future, I hope that I will find time to add this implementation of the CCM method to the lingpy package, to make it even easier for interested scholars to use it on their data.

References

Dolgopolsky, A. (1964). “Gipoteza drevnejšego rodstva jazykovych semej Severnoj Evrazii s verojatnostej točky zrenija [A probabilistic hypothesis concering the oldest relationships among the language families of Northern Eurasia]”. *Voprosy Jazykoznanija* 2. 53-63.

Forkel, R., J.-M. List, S. J. Greenhill, C. Rzymski, S. Bank, M. Cysouw, H. Hammarström, M. Haspelmath, G. Kaiping, and R. Gray (forthcoming). “Cross-Linguistic Data Formats, advancing data sharing and re-use in comparative linguistics”. *Scientific Data*. .

Kassian, A., M. Zhivlov, and G. Starostin (2015). “Proto-Indo-European-Uralic comparison from the probabilistic point of view”. *The Journal of Indo-European Studies* 43.3-4. 301-347.

Kessler, B. (2001). “The significance of word lists”. *Statistical tests for investigating historical connections between languages*. CSLI Publications: Stanford.

List, J.-M. (2014). *Sequence comparison in historical linguistics*. Düsseldorf University Press: Düsseldorf.

List, J.-M., S. J. Greenhill, and R. Gray (2017). “The potential of automatic word comparison for historical linguistics”. *PLOS ONE* 12.1. 1-18.

List, J.-M. , M. Walworth, S. J. Greenhill, T. Tresoldi, and R. Forkel (2018). “Sequence comparison in computational historical linguistics”. *Journal of Language Evolution*. 3 (2). 130-144.

Sokal, R. and C. Michener (1958). “A statistical method for evaluating systematic relationships”. *University of Kansas Scientific Bulletin* 28. 1409-1438. 2.3. 130-144.

Turchin, P., I. Peiros, and M. Gell-Mann (2010). “Analyzing genetic connections between languages by matching consonant classes”. *Journal of Language Relationship* 3. 117-126.

Cite this article as: Johann-Mattis List, “A fast implementation of the Consonant Class Matching method for automatic cognate detection in LingPy,” in *Computer-Assisted Language Comparison in Practice*, 01/10/2018, https://calc.hypotheses.org/477.

Enhancing morphological annotation for internal language comparison

Nathanael E. Schweikhard (10/10/2018)

Categories: Annotation

Tags: EDICTOR, morphological annotation, standardization

In language comparison, there is a long history of using concept-based wordlists to get insights into the degree of similarity between languages, going back at least to Morris Swadesh (Swadesh 1950). For these purposes, words from different languages that share the same meaning are compared, either manually or with computational methods. The latter have the advantages of being both faster and more consistent. However, there are also limits to what computer-based methods can detect for the time being.

One of the biggest problems in this context is that none of the currently available methods for automatic cognate detection can infer partial cognates directly if no information on morpheme boundaries is provided by the user. As a result, if morpheme boundaries are missing and morphological differences are frequent in the data one wants to investigate, automatic cognate detection can be seriously hampered (List, Greenhill, and Gray 2017).

For example, Latin *sōl* /soːl/ and French *soleil* /sɔlɛj/ (both "sun") would be considered more similar by an algorithm if it "knew" that the French word can be analyzed as consisting of two morphemes, and that it should only consider the first one, i. e. /sɔl/. It might also be the case that the same morphemes appear as allomorphs within a given language, e. g. due to vowel harmony or

Figure 1: Screenshots from EDICTOR (List 2017) contrasting automatic sound alignment without and with morphological annotation.

umlaut, but currently there are no automatic methods to detect allomorphic variation inside a language.

In order to tackle these problems one could either try to use existing methods for automatic morpheme boundary detection (Creutz and Lagus 2005) to preprocess our linguistic datasets, or one could increase the information that is made available to the algorithms in order to enable them to compare morphemes instead of full words (Hill and List 2017; List, Lopez, and Bapteste 2016).

Given that automatic morpheme detection does not yield reliable results at the moment, especially not for small wordlists as they are typically used in historical linguistics, we are currently trying to formalize the second approach by developing standards and examples for best practice to augment wordlists with morphological information. In doing so, we do not only hope to optimize our annotation frameworks in order to establish a workflow for future annotations, but also to create example annotations for future tests of novel methods for automatic morpheme detection.

Our goal here is to explore the possibilities of integrating any kind of morphological structures, be they synchronic or diachronic, within a consistent annotation framework. In the following, I want to give a short overview on some new ideas we came up with during the last months.

Preliminaries of annotation

We established a standardized way of how the data needs to be prepared and formatted, following the standard input formats of LingPy (List, Greenhill, and Forkel 2017) and EDICTOR (List 2017), which are largely compatible with the format specifications laid out by the CLDF initiative (Forkel et al. 2018). This is necessary to guarantee the comparability of the results, the re-usability of helper scripts, the integrability with existing annotation solutions, and machine-readability in general. The format specification requires that the wordlists are turned into a spreadsheet saved as TSV-file with a header row and one row for each wordform. The column containing this data is titled FORM. For each other kind of information, a distinct column is used. It needs to be made sure that each field contains only a single string of characters (including spaces if phrases occur among the data), and not multiple forms. If the data with which one is working does not follow these requirements or if other corrections to it are needed, they need to be adjusted accordingly. The original data is always preserved in the column VALUE, so nothing is lost (unless one excludes a given word form completely from the comparison). Comments and notes explaining some specific aspects of the entries under question are placed in the column NOTE.

The first column is called ID and contains a consecutive integer ID that needs to be larger than zero. This is necessary in order to further refine the data with the help of EDICTOR (List 2017) and has the additional advantage of making it easy to put the rows back into the original order if they were reordered. Since several languages might be included in one file and we want to allow for language-external comparison as well, the language of the data is specified in the column DOCULECT (following the terminology of Good and Cysouw 2013). Wordlists are typically created with the use of elicitation glosses and those are included in a column given the header CONCEPT.

ID	DOCULECT	CONCEPT	VALUE	FORM	NOTE
147	Old High German	the goat	geiz	geiz	
148	Old High German	the he-goat	boc, buc	boc	
149	Old High German	the kid	zickîn	zickîn	
150	Old High German	the horse	(h)ros	hros	
151	Old High German	the stallion	hengist	hengist	wrong meaning
152	Old High German	the stallion		reino	correction
153	Old High German	the mare	meriha	meriha	
154	Old High German	the foal or colt	folo	folo	
155	Old High German	the donkey	esil	esil	
156	Old High German	the mule	mûl	mûl	

Table 1: Entries from the Old High German wordlist of the World Loanword Database (Haspelmath 2009) prepared for further annotation.

Especially when using one's own data it is highly recommended to also add a column with the Concepticon-ID (List et al. 2018, see also https://concepticon.clld.org) corresponding to the elicitation gloss in order to make sure that later comparison across different datasets will be facilitated.

Segmented IPA-transcriptions

After these preliminary preparations, the words are turned into sequences of IPA-characters. This facilitates comparing languages since the original spelling is often quite idiosyncratic and several computational analyses which one might want to use the data for are only possible if it is provided in IPA. In order to create IPA from orthographic sources, there are different possibilities: One could convert all orthographic entries to their IPA version manually, one could try to extract the data from an available database

that lists phonetic transcriptions for orthographic entries (as many modern dictionaries do), or one could write an orthography profile (see Moran and Cysouw 2018, and for an introduction to how to create this kind of file see this tutorial) and convert the data with help of the Python implementation provided by the segments package.

Unfortunately, orthography profiles only work for very regular orthographies like those used by field workers who for convenience transcribe languages in alphabetic letters. For orthographies with a longer history, orthography profiles often cannot be used, since the orthography does not contain full information on the pronunciation. For example, in German, vowel length is not always marked by the spelling. Therefore for these kinds of spelling systems, one of the other options or a combination thereof needs to be chosen.

The IPA-column is named TOKENS. This is the default name for it in EDICTOR. Also, the individual sounds (all sounds that are judged as a sound unit by the researcher, i.e., including affricates and diphthongs, depending on the language under investigation) are separated by spaces.

Standardizing annotation practice

After all these previous (and at times tedious) steps of data preparation, we can finally start to annotate the morphemes in our data by marking morpheme borders with a plus sign. Here again this can either be done manually or by using some computer-assisted approach. If common morphemes are known to the researcher, a very simple (but in our experience also efficient) approach for marking at least a larger part of the morphemes semi-automatically is to use search and replace functionalities (in combination with regular expressions if needed). In this way, nearly all instances of, for example, the German prefix { *ver* } can be easily annotated by searching for ^f

ID	DOCULECT	CONCEPT	FORM	TOKENS
632	German	rotten	verfault	f ɛ r + f a ʊ l + t
633	German	rotten	vermodert	f ɛ r + m oː d ə r + t
634	German	drink	trinken	t r ɪ ŋ k + ə n
635	German	hunger	Hunger	h ʊ ŋ ə r
636	German	famine	Hungersnot	h ʊ ŋ ə r + s + n oː t
637	German	thirst	Durst	d ʊ r s t
638	German	suck	saugen	z au g + ə n
639	German	chew	kauen	k au + ə n
640	German	swallow	schlucken	ʃ l ʊ k + ə n
641	German	swallow	verschlucken	f ɛ r + ʃ l ʊ k + ə n

Table 2: Entries from the German wordlist of the Intercontinental Dictionary Series (Key and Comrie 2015), in IPA with morpheme borders.

ɛ r and replacing it with f ɛ r + (the ^ marks that it must be at the beginning of a word). Afterwards it is necessary to go through the whole list manually to check for erroneously segmented instances and add those morphemes that have not been caught with the automatic procedure. The resulting annotated data is left in the column TOKENS, but for the un-annotated IPA a backup-column could be created and it is useful to add a column for notes.

Once the data has been annotated in this way, we follow an idea proposed in Hill and List (2017) by adding glosses to the morphemes. This helps us specifically to disambiguate homophone morphemes which do not go back to the same ancestor, and, of course, this step can also be done at the same time when correcting the initial morpheme boundaries. The glosses are added

into a column called MORPHEMES. The form is free: which words to use as a gloss for a morpheme depends only on practicality, as long as the entry does not have any spaces (as EDICTOR segments the MORPHEMES content by spaces). What is important, however, is that identical morphemes across different words are given the same gloss, and that those that are not identical are given different glosses. In our tests, we use English as our glossing language and derive the morpheme gloss from the main meaning of each given word. Thus, German { *nah* } ("near"), for example, is glossed as NEAR. In cases when there are more homonyms in the language investigated than in English, we recommend to add _B, _C etc. to the glosses to distinguish them consistently. In contrast to the very free form proposed in Hill and List (2017), we have made good experience in writing content morphemes in upper case in order to mark them visually. Glosses for grammatical morphemes on the other hand are written in lower case and start with an underscore. Thus, we write _infinitive for the infinitive suffix in our German test data. This is based on a new feature of the EDICTOR, by which all morpheme glosses preceded by an underscore are displayed in transparent and small font to further mark the different in status for grammatical and content morphemes. Additionally, the underscore can be quickly added or deleted by right-clicking on a morpheme in the interface.

Distinguishing content from grammatical forms is not always an easy task, and at it is likely that scholars will disagree about individual decisions. To base our annotations on a clear-cut criterion, we considered as content morphemes only those that appear also as free morphemes or that are confixes (e. g. *Schwieger* -, "-in-law"). But the definition of a free morpheme as well as the border between a confix and an affix is far from clear and free morphemes can also be grammatical. Similarly, it is also not always clear where to put a morpheme border as definitions of what is a morpheme vary and may depend on the research question or the information available. Ideally, one would use the

same criterion for glossing not only throughout the same, but also across different languages, but this may prove to be difficult in practice since problems with the definition might only be noticed during the course of annotation. In order to reduce the amount of typing, a script can again be used to find and annotate the most common morphemes. All data, however, will usually need to be checked manually, since it is not possible to distinguish automatically between homophonous and recurring morphemes.

Deeper levels of annotation

As mentioned above, not only morpheme borders but also allomorphs should be included in the information provided to the computer. These non-homophone cognates can be marked in the MORPHEMES column during the previous step by giving them the same gloss names but differentiating them with a number at their end. So { *näh* } in German *näherkommen* (“approach”) is glossed in our examples in Table 3 below as NEAR2. In a further step, we annotate the actual roots by adding a column that we call ROOTS and one that we call ROOTIDS, where we no longer distinguish between NEAR and NEAR2. Although this annotation may seem rather complex, it has the clear advantage that it allows us to distinguish different kinds of word families inside the same language, namely those where morphemes recur in the same, and those where they recur in different forms (be it due to allophony or internal sound change).

What we cannot do yet

Morphological annotation works with strings of characters and predefined fields. This means that some types of information which one might want

ID	CONCEPT	FORM	TOKENS	MORPHEMES	COGIDS	ROOTS	ROOTIDS
1239	approach	nahen	n a: + ə n	NEAR _infinitive	953 42	NEAR _infinitive	890 42
1240	approach	hingehen	h ɪ n + g e: + ə n	TO GO _infinitive	926 441 42	TO GO _infinitive	865 414 42
1241	approach	näherkommen	n ɛ: + ə r + k ɔ m + ə n	NEAR2 _comp COME _infinitive	954 130 955 42	NEAR _comp COME _infinitive	890 128 891 42
1242	approach	sich nähern	z ɪ x _ n ɛ: + ə r + n	ONESELF NEAR2 _comp _infinitive2	414 954 130 329	_reflexive NEAR _comp _infinitive	394 890 128 42
1243	enter	hineingehen	h i: n + ai n + g e: + ə n	TO IN GO _infinitive	926 389 441 42	TO IN GO _infinitive	865 371 414 42
1244	enter	eintreten	ai n + t r e: t + ə n	IN TREAD _infinitive	389 933 42	IN TREAD _infinitive	371 872 42
1245	enter	hereinkommen	h ɛ r + ai n + k ɔ m + ə n	FROM IN COME _infinitive	939 389 940 42	FROM IN COME _infinitive	878 371 152 42
1246	carry (bear)	tragen	t r a: g + ə n	CARRY _infinitive	956 42	CARRY _infinitive	436 42
1247	carry (bear)	schleppen	ʃ l ɛ p + ə n	CARRY_B _infinitive	957 42	CARRY_B _infinitive	892 42

Table 3: Entries from the German wordlist of the Intercontinental Dictionary Series (Key and Comrie 2015) with morphological glosses and root annotation.

to include are difficult to implement. For an etymological annotation in which not only transparent morpheme borders are included, one will likely encounter cases in which morpheme borders have become fuzzy due to phonological mergers. An example would be the German word *Messer* (/mɛsɐ/, "knife") which in modern German is monomorphemic but goes back to a compound (Old High German *mezzi-sahs*, later *mezzi-rahs*, literally "food-knife", see Watkins 1990: 295). For some languages, cases like these can be found even when taking a synchronic perspective.

Additionally, it would be interesting to include information on the kinds of sound changes that a morpheme or word underwent during its history. But we have not decided where to add this information, and how to standardize it, specifically also because it will be difficult to find a principled and standardized way to do so across different languages. If there was only one sound shift per morpheme, it would be quite possible to develop a straightforward proposal for annotation, but considering that morphemes are not limited in the amount of phonological changes they may accumulate, we find it difficult to come up with a proposal at this stage of the research. However, since we annotate allomorphs consistently, we are already able to identify all allomorphs of a given root in our data. Therefore, it will be easy to add information on

sound changes later, once we managed to find a useful representation format.

Finally, we could not (yet) find any systematic way to model analogical relations between words. Given the importance and frequency of analogy in arguments in historical linguistics, we will try to come up with proposals for this in future versions of our annotation framework.

References

Creutz, M., and K. Lagus (2005). "Unsupervised Morpheme Segmentation and Morphology Induction from Text Corpora Using Morfessor 1.0". *Publications in Computer and Information Sciences* 81. Helsinki: Helsinki University of Technology.

Forkel, Robert, Johann-Mattis List, Simon J. Greenhill, Christoph Rzymski, Sebastian Bank, Michael Cysouw, Harald Hammarström, Martin Haspelmath, Gereon A. Kaiping, and Russell D. Gray (2018). "Cross-Linguistic Data Formats, Advancing Data Sharing and Re-Use in Comparative Linguistics". *Nature Scientific Data*.

Good, Jeff, and Michael Cysouw (2013). "Languoid, Doculect, Glossonym: Formalizing the Notion of 'Language'." *Journal of Language Documentation and Conservation* 7: 331–59. https://scholarspace.manoa.hawaii.edu/handle/10125/4606.

Haspelmath, Martin, and Uri Tadmor, (eds.) (2009). *WOLD*. Leipzig: Max Planck Institute for Evolutionary Anthropology. https://wold.clld.org/.

Hill, Nathan W., and Johann-Mattis List (2017). "Challenges of Annotation and Analysis in Computer-Assisted Language Comparison: A Case Study on Burmish Languages." *Yearbook of the Poznań Linguistic Meeting*, no. 3: 47–76.

Key, Mary Ritchie, and Bernard Comrie, (eds.) (2015). *IDS*. Leipzig: Max Planck Institute for Evolutionary Anthropology. https://ids.clld.org/.

List, Johann-Mattis, Michael Cysouw, Simon J. Greenhill, and Robert Forkel, (eds.) (2018). *Concepticon*. Jena: Max Planck Institute for the Science of Human History. http://concepticon.clld.org/.

List, Johann-Mattis (2017). "A Web-Based Interactive Tool for Creating, Inspecting, Editing, and Publishing Etymological Datasets." In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics. System Demonstrations*,

9–12. Valencia: Association for Computational Linguistics. http://aclweb.org/anthology/E/E17/E17-3003.pdf.

List, Johann-Mattis, Simon J. Greenhill, and Russell D. Gray (2017). "The Potential of Automatic Word Comparison for Historical Linguistics." *PLoS ONE*, no. 1, 12 (January). Public Library of Science: 1–18. doi: http://dx.doi.org/10.1371/journal.pone.0170046.

List, Johann-Mattis, Simon J. Greenhill, and Robert Forkel (2017). "LingPy. A Python Library for Quantitative Tasks in Historical Linguistics." Jena: Max Planck Institute for the Science of Human History. doi: https://doi.org/10.5281/zenodo.1065403.

List, Johann-Mattis, Philippe Lopez, and Eric Bapteste (2016). "Using Sequence Similarity Networks to Identify Partial Cognates in Multilingual Wordlists." In: *Proceedings of the Association of Computational Linguistics 2016*, 2: Short Papers:599–605. Berlin: Association of Computational Linguistics. http://anthology.aclweb.org/p16-2097.

Moran, Steven, and Michael Cysouw (2018). *The Unicode Cookbook for Linguists: Managing Writing Systems Using Orthography Profiles*. Translation and Multilingual Natural Language Processing 10. Berlin: Language Science Press.

Swadesh, Morris. 1950. "Salish Internal Relationships." *International Journal of American Linguistics* 16 (4): 157–67.

Watkins, Calvert. 1990. "Etymologies, Equations, and Comparanda: Types and Values, and Criteria for Judgment." In: *Linguistic Change and Reconstruction Methodology*, edited by Philip Baldi, Part 1.45:289–303. Trends in Linguistics. Studies and Monographs. Berlin; New York: Mouton de Gruyter.

Cite this article as: Nathanael E. Schweikhard, "Enhancing morphological annotation for internal language comparison," in *Computer-Assisted Language Comparison in Practice*, 10/10/2018, https://calc.hypotheses.org/570.

Inferring consonant clusters from CLICS data with LingPy

Johann-Mattis List (07/11/2018)

Categories: Code, Dataset

Tags: CLICS, consonant clusters, LingPy, prosody

LingPy (List et al. 2017) offers a great deal of functions for string manipulation. Although most of those functions are readily documented (see lingpy.org for details), and the basic ideas have also been described in my dissertation (List 2014), it seems that not many users are aware of these additional possibilities, which the library offers.

In the following, I want to illustrate how we can use LingPy to learn something about consonant clusters occurring in the data underlying the CLICS database (List et al. 2018, clics.clld.org). I have illustrated in an earlier post how one can use the CLICS software API to cook one's own CLICS application. I will thus assume that you know how to install CLICS (following the instructions on our GitHub page) and the data underlying it.

As a simple shortcut, you can install all required datasets by downloading the data underlying this blog post from GitHub Gist and typing:

```
1 $ pip install -r pip-requirements.txt
```

Once you have done this, please follow the instructions at the CLICS website mentioned above to prepare the CLICS datasets by converting them to CLDF.

Starting from this, and assuming that you have an actual version of LingPy installed, I will now illustrate how you can extract all data that is readily *segmented* (in the sense of Moran and Cysouw 2018), i.e., by using the space as a segmentation marker, and placing it between all sounds that do not constitute a valid sound unit (see also List et al. 2018b). But additionally, we will compute *prosodic strings*, an idea that I already discussed in my dissertation (List 2014), and which allows us to distinguish different *kinds* of consonant in a string, based on whether they appear in an environment in which sonority *increases* or *decreases*. In this way, we can make our own cross-linguistic collection of consonant clusters.

But let's start by loading the releavant libraries.

```
from lingpy import *
from pyclics.api import Clics
from pyclics.models import Form
from tqdm import tqdm
from collections import defaultdict
```

To obtain quick access to all the data available in our `clics.sqlite3` database, we need to modify the function in the CLICS API slightly, in such a way that the code yields the segmented entries, not the ascified CLICS value that we use for the computation of cross-linguistic colexifications. This is achieved with help of the following function.

```
def iter_wordlists(db, varieties):
    languages = {(v.source, v.id): v for v in varieties}
    for (dsid, vid), v in sorted(languages.items()):
        forms = [Form(*row) for row in db.fetchall("""
select
    f.id, f.dataset_id, f.form, f.segments,
    p.name, p.concepticon_id, p.concepticon_gloss,
    p.ontological_category, p.semantic_field
from
    formtable as f, parametertable as p
where
    f.parameter_id = p.id
    and f.dataset_id = p.dataset_id
    and p.concepticon_id is not null
    and f.language_id = ?
    and f.dataset_id = ?
order by
    f.dataset_id, f.language_id, p.concepticon_id
""", params=(vid, dsid))]
        assert forms
        yield v, forms
```

In the version of prosodic strings that we want to use for this application, LingPy distinguishes four basic types of prosodic environment: vowels (V), consonants in ascending sonority environment (C), consonants in descending sonority environment (c), and tones (T). To obtain only the consonant clusters from a given sound sequence, we thus need a function that checks if we are currently in consonant environment, returning all clusters from a string. This is done with help of the following function.

```
def get_clusters(tokens, prostring):
    clusters = ['']
    for t, c in zip(tokens, prostring):
        if c == 'C':
            if clusters[-1].startswith('<'):
                clusters[-1] += ' '+t
            else:
                clusters += ['</ '+t]
        elif c == 'c':
            if clusters[-1].startswith('>'):
                clusters[-1] += ' '+t
            else:
                clusters += ['>/ '+t]
        else:
            clusters += ['']
    return [x for x in clusters if x]
```

This function will mark clusters in ascending environment by adding `</` in the beginning of the cluster, and descending environment by adding a `>/` . One could think of more elegant ways of marking or handling this, but for our purpose, it is sufficient.

We can now start with the actual code. We start by defining different variables, namely: the CLICS object that allows us to get access to CLICS data, a dictionary that will later be converted to a LingPy Wordlist, to allow for an easy writing to file, and the clusters that we obtained from analyzing the data.

```
clics = Clics('.')
D, idx = {0: ['doculect', 'concept', 'segments', 'cv']}, 0
clusters = defaultdict(lambda : defaultdict(int))
```

We can now start to load all varieties in CLICS. Usually, I write a `print` statement after this, since loading the data takes some time, and I prefer some feedback.

```
varieties = clics.db.varieties
print('[i] loaded clics varieties')
```

Now, we can loop over the data. We do this with help of the `tqdm` function that gives visual feedback, since this may take some time. The basic idea of this loop is to retrieve the segments from the CLICS database, check if its valid (using **`try`** and `except ValueError`), and convert it to its prosodic string, using the `prosodic_string` function provided by LingPy. In addition, we use the `get_clusters` function to extract the different types of consonant clusters, count them, and store them in our `clusters` variable.

```
for v, forms in tqdm(iter_wordlists(clics.db, varieties),
   total=len(varieties)):
    for form in forms:
        idx += 1
        clics_form = form.clics_form.strip()
        if clics_form:
            try:
                tokens = clics_form.split()
                prostring = prosodic_string(tokens,
                   _output='CcV')
                D[idx] = [
                        v.gid,
                        form.concepticon_id,
                        clics_form,
                        prostring
                        ]
            except ValueError:
                pass

        clrs = get_clusters(tokens, prostring)
        for clr in clrs:
            clusters[clr, len(
                clr.split())][v.gid.split('-')[-1]] += 1
```

Now, that we have obtained all the data, we can write the data to file. First, we write a typical LingPy wordlist, which contains an additional column that we call "CV". The resulting file is a plain TSV file that can also be inspected with interfaces like EDICTOR (List 2017).

```
wl = Wordlist(D)
wl.output(
        'tsv',
        filename='cv-patterns',
        ignore='all',
        prettify=False
        )
```

Now, as a final step, we write the data in our `clusters` dictionary to file. Here, we write a gain to a TSV file, but we do this "manually" by iterating over the dictionary.

```
with open('cv-clusters.tsv', 'w') as f:
    f.write('Cluster\tLength\tFrequency\n')
    for (cluster, length), rest in sorted(
            clusters.items(),
            key=lambda x: len(x[1]),
            reverse=True
            ):
        f.write('{0}\t{1}\t{2}\n'.format(
            cluster, length, len(rest))
            )
```

Once this is done, we can do many things with the data. We can inspect the occurrence of certain clusters, see whether we find areal patterns, check, which languages are richest in terms of consonant clusters, etc. I won't discuss any of these possibilities in detail here, as this post was simply written in order to

illustrate how easily we can extract the data with help of tools like LingPy and the CLICS API. So if you are interested in the actual results of this little study, I suggest you test the code yourself and see what you get. For convenience, I have uploaded the whole script to a GitHub Gist, which you can find here.

References

List, J.-M. (2014). *Sequence comparison in historical linguistics*. Düsseldorf University Press: Düsseldorf.

List, J.-M. (2017). "A web-based interactive tool for creating, inspecting, editing, and publishing etymological datasets". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics. System Demonstrations*. 9-12.

List, J.-M., S. J. Greenhill, and R. Forkel (2017). *LingPy. A Python library for quantitative tasks in historical linguistics*. Software Package. Version 2.6. Max Planck Institute for the Science of Human History: Jena. http://lingpy.org.

List, J.-M., M. Walworth, S. J. Greenhill, T. Tresoldi, and R. Forkel (2018). "Sequence comparison in computational historical linguistics". In: *Journal of Language Evolution* 3.2. 130–144.

List, J.-M., S. J. Greenhill, C. Anderson, T. Mayer, T. Tresoldi, and R. Forkel (2018). "CLICS^2. An improved database of cross-linguistic colexifications assembling lexical data with help of cross-linguistic data formats". In: *Linguistic Typology* 22.2. 277-306.

Moran, S. and M. Cysouw (2018): *The Unicode Cookbook for Linguists: Managing writing systems using orthography profiles*. Language Science Press: Berlin.

Cite this article as: Johann-Mattis List, “Inferring consonant clusters from CLICS data with LingPy,” in *Computer-Assisted Language Comparison in Practice*, 07/11/2018, https://calc.hypotheses.org/998.

From Fieldwork to Trees 1: Data preparation

Gereon A. Kaiping (14/11/2018)

Categories: Code

Tags: Austronesian Languages, code example, EDICTOR, lexical data, MS Excel, Python

---

A colleague of mine has recently returned from his fieldwork, where he collected data on on the dialectal variation of the Alorese language of Alor and Pantar in the East Nusa Tenggara province of Indonesia. He collected data on 13 Alorese varieties, including word list data. One obvious step for comparing the dialects is to mark which forms are obviously cognate and then use a standard tree (or network) construction algorithm to display the shared signal in the data. With standard tools and a bit of Python glue, this is an easy task. A script for 3 steps can be found in my repository on github. In this first part, I will describe how to get an Excel file into a format LingPy can deal with.

I was given a MS Excel file by my colleague. The file contains a single sheet with 13 Alorese word lists in matrix format, the top left of it looking like the following.

This is a very frequent shape of comparative word lists, so I hope the procedure described in the following – and the script I will provide – will help with other language or dialect comparison tasks.

A first thing to notice is that some cells contain a single form in what looks like reasonably decent IPA, but other cells (cf. *"what"* in **alb**, which quite clearly contains two forms, one shared with **alk** and one with **dul**) contain synonyms,

English	Indonesian	dul	alk	alb
1sg	saya	gɔ	go	go
2sg (informal)	kamu	mi	mo	mo
2sg (polite)	Anda			
3sg	dia	no	no	no
1pl excl	kami	tite	kame	kame
1pl incl	kita	tite	kame	ite
2pl	kalian	punauŋ	mi	mi
3pl	mereka	fe:	fe	fe
this	ini	ɦ̃adʒa	ha, kia	hã
that	itu	tedʒa	kal:i	kəte
here	di sini	hadʒa	hã	ha ɔnɔŋ
there	di sana	fei kalei	felio	fali kali
who?	siapa	hafa	feiru	fiaru
what?	apa	paru	pai	pai, paru
where?	di mana	naŋ ga ɔrɔ	oro pai noŋ	naŋga, ɔrɔ naŋga, naŋga dʒafa
when?	kapan	ɛrɛ pira	erepira	ɛrə pira
how?	bagaimana	namo naŋga	namonaŋga	nəmga, nəmən;ga

Table 1: Modified sample of data from the Excel file.

which appear to be separated by , . In the past, I have seen people not being very consistent about what separators they use, so I run a quick check by searching in the Excel sheet: While the two gloss columns, **English** and **Indonesian**, contain several different separators (, , ; , / , and also brackets), the transcribed data looks good in this respect: There are no / in the actual word list. The single ; in the matrix, which is the one you can see in **alb** "how?" above, looks like it was used as an ad-hoc separator to stop <ng> to become /ŋ/ in a search-and-replace step I expect happened at some point. There are only a handful of instances of brackets, and commas seem to consistently sep-

arate different forms.In order to work with this data in LingPy (List et al. 2018) or in any program that supports the CLDF standard (Forkel et al. 2018), the data needs to be converted into a *long table* format, where each row lists one form, indexed by language and concept.

Unfortunately, the CLDF format has different default column headers from LingPy/ edictor. All three support custom column headers, but CLDF makes it very easy to use them (without any additional effort in all but the most restrictive use case, even) while I tend to find it quite a hassle to convince edictor of custom column names, so we will use edictor's defaults. That means that we want column headers `ID` for the row IDs (which may not start at 0), `DOCULECT` for the names of the varieties, `CONCEPT` for the gloss column, `IPA` for the column containing the forms and `TOKENS` for the column of segmented forms. For metadata-free CLDF, the corresponding header names would be `ID`, `Language_ID`, `Concept_ID`, `Form`, and `Segments`. We will create a file to bridge between these two name sets later.

While reading all the forms, it makes sense to also segment them already, because we need that to happen anyway for automatic cognate coding. For segmenting IPA transcribed data, I will use Robert Forkel's `segments` python package in its default mode. Pavel Sofroniev's `ipatok` might be an alternative. This is also the step where we could use `pyclts` to check the quality of the transcription, but that would go beyond the scope of this example.

Python has an Excel reader module called `xlrd`. Using it, the core functionality of the conversion script looks like this.

```
import csv
import xlrd
import segments

book = xlrd.open_workbook("Wordlists.xlsx")
sheet = book.sheet_by_index(0)

def cell(row, col):
  return sheet.cell_value(row, col)

tokenizer = segments.Tokenizer()
def segment(word):
  return tokenizer(word, ipa=True, separator=" _ ")

column_names = [cell(0, c) for c in range(sheet.ncols)]

with open("wordlist.tsv", "w") as out:
  write = csv.writer(out, dialect="excel-tab").writerow
  write(["ID", "CONCEPT", "DOCULECT", "IPA", "TOKENS"])
  i = 1
  for row in range(1, sheet.nrows):
    concept = cell(row, 0)
    for col in range(2, sheet.ncols):
      lect = cell(0, col)
      for form in cell(row, col).split(","):
        form = form.strip()
        segments = segment(form)
        write([i, concept, lect, form, segments])
        i += 1
```

The output is the following TSV file which can be used eg. in edictor.

In order to make this script re-useable, it is obviously useful to replace the hard-coded assumptions (file paths, separators, number of gloss languages) with command line arguments and such like. If we want to use LingPy for cognate coding, we also have to skip empty forms like 7–9 above with an `if`

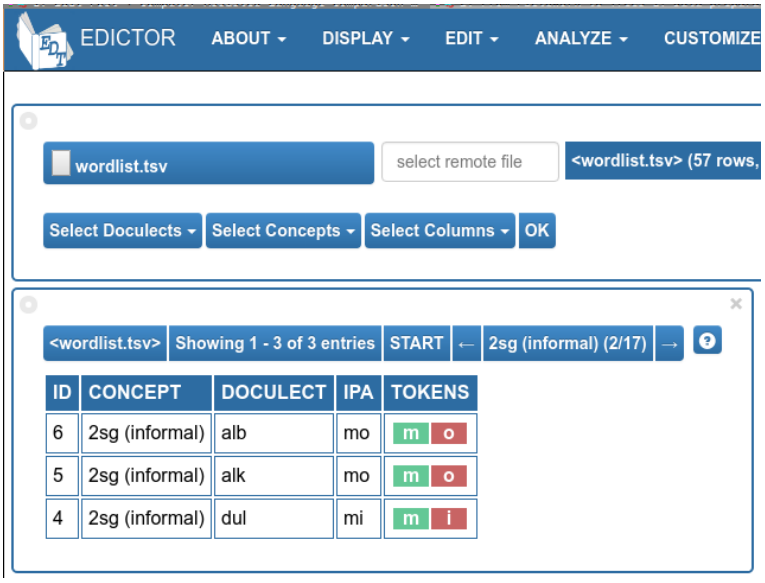

Figure 1: Forms 4–6 from wordlist.tsv in Edictor.

`not form`: **`continue`**. But the core of the conversion are just the last 10-or-so lines of this script, and then I have my colleague's language in a format ready for further use, and how I use it will be content of a later post.

ID	CONCEPT	DOCULECT	IPA	TOKENS
1	1sg	dul	gɔ	g ɔ
2	1sg	alk	go	g o
3	1sg	alb	go	g o
4	2sg (informal)	dul	mi	m ɨ
5	2sg (informal)	alk	mo	m o
6	2sg (informal)	alb	mo	m o

Table 2: Sample of data from wordlist.tsv in tabular form.

References

List, Johann-Mattis & Greenhill, Simon J. & Forkel, Robert (2018). *LingPy. A Python Library for Quantitative Tasks in Historical Linguistics. Version 2.6.3*. Zenodo. doi: 10.5281/zenodo.1203193. https://zenodo.org/record/1203193#.W-xUdhBRfc8 (14 November, 2018).

Forkel, Robert & List, Johann-Mattis & Greenhill, Simon J. & Rzymski, Christoph & Bank, Sebastian & Cysouw, Michael & Hammarström, Harald & Haspelmath, Martin & Kaiping, Gereon A. & Gray, Russell D. (2018). "Cross-Linguistic Data Formats, advancing data sharing and re-use in comparative linguistics". *Scientific Data* 5. 180205. doi: 10.1038/sdata.2018.205.

Cite this article as: Gereon A. Kaiping, "From Fieldwork to Trees 1: Data preparation," in *Computer-Assisted Language Comparison in Practice*, 14/11/2018, https://calc.hypotheses.org/803.

Semantic promiscuity as a factor of productivity in word formation

Nathanael E. Schweikhard (19/11/2018)

Categories: Terminology

Tags: productivity, promiscuity, word formation

The blog post introduces ideas discussed in our project about taking a closer look at word formation from a semantic (or semasiological) point of view. Since this so far underinvestigated approach to word formation processes lacks proper terminology, a new term to denote the central research question of concept-based type-frequency is introduced and contrasted with related established terminology.

Productivity in the context of word formation is a term of many definitions (Bauer 2003:1). A not necessarily exhaustive list is provided by Gaeta and Ricca (2015), based on Rainer (1987):

a. the number of words formed with a certain W[ord]F[ormation]R[ule];
b. the number of new words coined with a certain WFR in a given time span;
c. the possibility of coining new words with a certain WFR;
d. the probability of coining new words with a certain WFR;
e. the number of possible (or generatable by rule) words formed with a certain WFR;

f. the relation between occurring and possible words formed with a certain WFR.

Bauer (2003) mainly concerns himself with definitions c. to e., namely to which degree a morphological pattern (i. e. what was above referred to as a word formation rule) is available to be used in creating new words, not how much it is actually used. According to him, this can be determined by looking at the number of cases available for applying the pattern, the number of (absolute or relative) constraints when applying the pattern, and the usefulness of the word formations that it can create.

Yet in either case, and in all the definitions given, the focus lies on either the output side of word formation or on the restrictions of a specific morphological pattern. These restrictions may lie in the input (e. g. certain patterns only being applied to the input of specific characteristics like for example a certain phonetic shape) but the *meaning* of the input itself has not been studied much so far, although it is clear that word formation would not be possible without it.

When investigating the semantics underlying word formation processes, the general question from the perspective of the meaning of the input could be stated as follows: *Independent of any specific kind of word formation pattern, are there morphemes that appear more commonly as bases of word formation than others, and if so, what do they have in common?* In contrast to classical research on word formation, which often investigates *potential* word formations, this question can probably best be studied by investigating *existing* word formation patterns, that is, by asking how often one morpheme occurs in how many words (considering the whole lexicon of a given language, or a certain section of it).

Shifting the focus from the forms to the concepts would also facilitate cross-linguistic comparison, especially if one starts from basic con-

cepts, like the ones traditionally used in fieldwork and lexical studies on historical language comparison (compare the Concepticon resource, https://concepticon.clld.org, for an overview, List et al. 2016). Such research would, of course, not assume that concepts are the same across languages, but rather employ the idea of using *comparative concepts* in the sense of Haspelmath (2010:668).

Given that – to my knowledge – a systematic investigation of the "semantic productivity" of concepts across languages has not been carried out so far, we can only speculate why – if at all – some words expressing specific concepts would recur more frequently as the base of new words than others. Provided that we can identify cross-linguistic trends, the Embodiment theory, according to which language is shaped by our physical characteristics, might provide an explanation. As one of the few studies devoted to the topic, Geisler (2018) demonstrates for German that some of the morphemes denoting concepts most deeply rooted in our early childhood experience (such as "to stand" or "to fall"), seem to be used extremely frequently in derivation for a large variety of meanings derived from the conceptual base.

Unfortunately, there does not seem to be a proper term for what was referred to as "semantic productivity" or "concept-based type-frequency" above. Although the phenomenon by which a morpheme reappears as part of word formations due to its semantics has been sporadically mentioned and discussed before, no proper term has made it into handbooks or glossaries of linguistics. The one who comes closest to it is Blank (1997:21), who uses and slightly redefines the terms *attraction* and *expansion* (coined by Sperber 1923) in order to develop a theory of semantic change[1]:

[1]My translation, original text: "Expansion und Attraktion sind komplementäre Prozesse: Bei der Expansion handelt es sich um ein semasiologisches Verfahren: Hier erhält ein Wort neue Bedeutungen. Die Expansion kann somit auch bei der Erstbenennung von Neuerungen eine wichtige Rolle spielen. Bei der Attraktion hingegen handelt es sich primär um ein onomasiologisches Verfahren: Für ein und denselben Sachverhalt

> Expansion and attraction are complementary processes: Expansion is a semasiologic procedure: Here a word gains new meanings. Expansion therefore can also play a role in the first-time denoting of an innovation. Attraction on the other hand is primarily an onomasiologic procedure: New denotions are created for one and the same concept. Secundarily attraction of course also leads to semantic shift, namely in those words that are used for denoting the «attractive» concept. (Blank 1997: 21)

We can clearly see that Blank's term *expansion* describes the process underlying the phenomenon for which we do not yet have a name. However, as Blank states himself, expansion is based on a semasiological perspective: A specific word is gaining a new meaning additionally to those it already had. Yet the phenomenon we discussed so far focuses on an onomasiological approach that investigates how concepts contribute to semantic expansions of the words denoting them. Furthermore, expansion refers to a concrete process, situated in a specific language and time frame, whereas our starting point was – among others – the question of whether and to which degree universal tendencies could be encountered when comparing the "expansivity" of concepts across the world's languages.

Both expansion and what we are referring to here form counterparts to Blank's term *attraction*. Attraction means that a concept attracts new words or word formations denoting it. If, on the other hand, a word's meaning expands, it can be used to coin new words, which – in turn – would be expected to denote "attractive" concepts.

To form a wide range of new words, words first need to expand their meaning, which itself is also based mainly on the meanings the words already have. While meaning expansion can happen to any word independent of its previ-

werden neue Bezeichnungen geschaffen. Sekundär führt natürlich auch die Attraktion zu Bedeutungswandel, nämlich bei den Wörtern, die zur Bezeichnung des «attraktiven» Sachverhaltes herangezogen wurden."

ous meanings, we here propose that some meanings might have a stronger tendency to undergo this expansion.

After longer discussions in our project, during which we tested many different terms as candidates to denote that some morphemes may be more important for word formation due to their semantics, we decided to take inspiration from microbiological terminology and use the term *promiscuity*. Investigating the *promiscuity of concepts*, be it cross-linguistically or within one language, would thus entail that we investigate to what degree word formation is driven by the semantics of the words or morphemes being recycled to form new words.

But why "promiscuity"? In the natural sciences, the term *promiscuity* has been used since at least the early 20th century – and more commonly since the 1970s – as

> [the] ability of a protein, organism, etc., to interact with a variety of targets or in a non-specific manner; *spec.* the propensity of a plasmid, pathogenic organism, etc., to infect many different hosts. (OED)

Given that biology and linguistics often share terminology and metaphors (List et al. 2016a), we can also find examples in linguistics, where promiscuity is used as a term, albeit it is less widespread and mainly used in a quite restricted sense. Zwicky (1987:136), for example, speaks of "promiscuous attachment" by which he means the "attachment to i[nflected]-forms of virtually any syntactic category" (he also uses the term "promiscuity" in Zwicky 1986 and talks about the same concept without having a term for it in Zwicky and Pullum 1983). Similarly, Haspelmath (2018:315) uses "promiscuous" for bound forms that attach to more than one word class. In his definition, promiscuous morphemes form thus a category between affixes and roots.

In a slightly different approach, Zólyomi et al. (2017:21) use "promiscuous" to refer to the mixing of cases, i. e. cases extending their usage to functions pre-

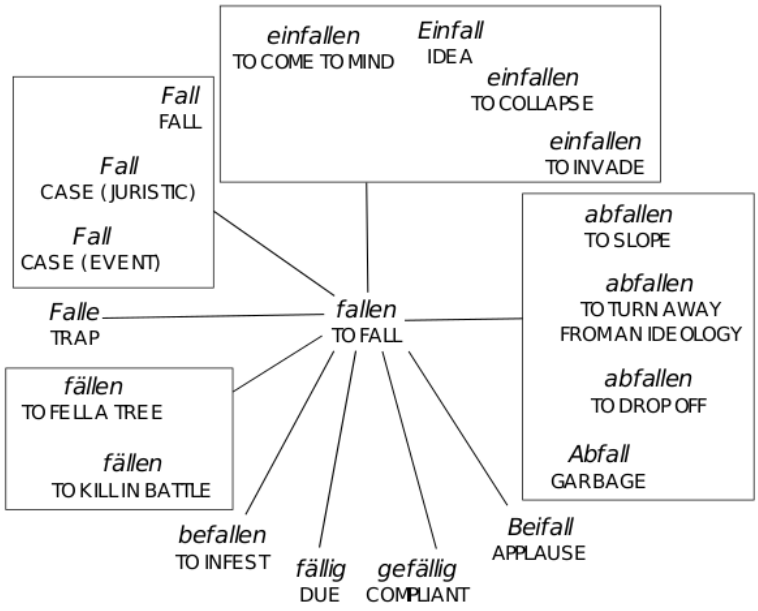

Figure 1: The concept "to fall", denoted by German *fallen*, showing its high promiscuity in a selection of the wide range of meanings derived from it. This includes the expansion of meanings of existing word formations, here marked by grouping together derivations sharing the same stem.

viously inhabited by other cases. However, they write this in the context of second language learners and not of "normal" language change, and apparently do not intend to use it as a technical term.

What all these uses of the term have in common is that they talk about the *openness* of something to connect to a variety of different kinds of other things, a meaning that is clearly based on the term's more colloquial usage.

The *semantic promiscuity* proposed here (called such to differentiate it from the more syntactic or grammatical concept by Zwicky and Haspelmath) could first be said to differ from that original meaning by being concerned with the type-frequency of a morpheme in general, independent of its choice of word

formation pattern or, e. g., of the grammatical variety of other morphemes it connects to.

However, while semantic promiscuity, unlike its grammatical counterpart, is not referring to concepts being less restricted in the grammatical aspects of the bases they can take, it nevertheless refers to something similar, namely that morphemes denoting promiscuous concepts are less restricted regarding the semantics of the morphemes they attach to in word formation and the semantics of the new words formed thereby, either by being already polysemous themselves, or by being more open to develop polysemy, i. e. to undergo semantic expansion.

An example of how promiscuity and expansion interact can be seen in Figure 1 above. Having provided this preliminary description of what it is we are talking (or want to talk) about, the next step will be to actually investigate it cross-linguistically.

References

Bauer, Laurie (2003). *Morphological Productivity*. Cambridge: Cambridge University Press.

Blank, Andreas (1997). "Prinzipien des lexikalischen Bedeutungswandels am Beispiel der romanischen Sprachen". *Beihefte zur Zeitschrift für romanische Philologie* 285. Tübingen: Niemeyer.

Gaeta, Livio, and Davide Ricca (2015). "Productivity." In *Word-Formation: An International Handbook of the Languages of Europe*, edited by Peter O. Müller, Ingeborg Ohnheiser, Susan Olsen, and Franz Rainer, 22:842–858. Handbücher zur Sprach- und Kommunikationswissenschaft 2. Berlin;New York: De Gruyter Mouton.

Geisler, Hans (2018). "Sind Unsere Wörter von Sinnen? Überlegungen zu den sensomotorischen Grundlagen der Begriffsbildung." In *Worte über Wörter. Festschrift zu Ehren von Elke Ronneberger-Sibold*, edited by Kerstin Kazzazi, Karin Luttermann, Sabine Wahl, and Thomas A. Fritz, 131–142. Tübingen: Stauffenburg.

Haspelmath, Martin (2010). "Comparative Concepts and Descriptive Categories in Crosslinguistic Studies." *Language: A Jou rnal of the Linguistic Society of America* , no. 86, 3: 663–687.

List, Johann-Mattis, Michael Cysouw, Simon J. Greenhill, and Robert Forkel, (eds.) (2018). *Concepticon*. Jena: Max Planck Institute for the Science of Human History. http://concepticon.clld.org/.

List, Johann-Mattis, Michael Cysouw, and Robert Forkel (2016). "Concepticon. A Resource for the Linking of Concept Lists." In *Proceedings of the Tenth International Conference on Language Resources and Evaluation*, edited by Nicoletta Calzolari, Khalid Choukri, Thierry De-

clerck, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, 2393–2400. Portorož: European Language Resources Association.

List, Johann-Mattis, Jananan Sylvestre Pathmanathan, Philippe Lopez, and Eric Bapteste (2016). "Unity and Disunity in Evolutionary Sciences: Process-Based Analogies Open Common Research Avenues for Biology and Linguistics." *Biology Direct*.

OED Online. July 2018. Oxford University Press. http://www.oed.com/view/Entry/152428?redirectedFrom=promiscuity (accessed November 16, 2018).

Rainer, Franz (1987). "Grammatik und Wortbildung romanischer Sprachen." In *Produktivitätsbegriffe in der Wortbildungslehre*, edited by Wolf Dietrich, Hans-Martin Gauger, and Horst Geckeler, 187–202. Tübingen: Narr.

Sperber, Hans (1923). *Einführung in die Bedeutungslehre*. Bonn; Leipzig: Schroeder.

Zólyomi, Gábor, Szilvia Jáka-Sövegjártó, and Melinda Hagymássy (2017). *An Introduction to the Grammar of Sumerian*. Budapest: Eötvös University Press.

Zwicky, Arnold M. (1986). "Incorporating the Insight of Autolexical Syntax." *OSU Working Papers in Linguistics* 32: 139–143.

Zwicky, Arnold M. (1987). "Suppressing the Zs." *Journal of Linguistics* 23 (1): 133–148. http://www.jstor.org/stable/4175870 .

Zwicky, Arnold M., and Geoffrey K. Pullum (1983). "Clitization vs. Inflection: English N'T." *Language* 59 (3): 502–513.

Cite this article as: Nathanael E. Schweikhard, "Semantic promiscuity as a factor of productivity in word formation," in *Computer-Assisted Language Comparison in Practice*, 19/11/2018, https://calc.hypotheses.org/1169.

From Fieldwork to Trees 2: Cognate coding

Gereon A. Kaiping (27/11/2018)

Categories: Code, Dataset

Tags: Austronesian Languages, CLDF, code example, cognate detection, cross-linguistic data formats, lexical data

---

In a previous post (Kaiping 2018), I described how to convert matrix-shape word lists given in Excel into the long format LingPy and other software can work with. My motivation for this was to provide my colleague Yunus Sulistyono with a good way to compare the lexicon of his Alorese [alor1247] dialects, and to understand the relationship between them. In this post, the data is automatically cognate coded and converted into CLDF.

If we have a file like the following, we can easily use LingPy's functionality to get cognate classes.

ID	CONCEPT	DOCULECT	IPA	TOKENS
1	1sg	dul	gɔ	g ɔ
2	1sg	alk	go	g o
3	1sg	alb	go	g o
4	2sg (informal)	dul	mi	m i
5	2sg (informal)	alk	mo	m o
6	2sg (informal)	alb	mo	m o

Table 1: Sample of data from wordlist.tsv in tabular form.

This whole file (`wordlist.tsv`) contains data on dialects of one language (at least in simplified terms of the self-identification of the speakers, who refer to all these varieties as "bahasa Alor" – how valid that categorization actually is might be one outcome of his project) all transcribed by the same researcher, so we should not expect a lot of problems from inconsistent transcription or from cognates that cannot be recognized as such in the data.

Some word boundaries are marked, so we can make use of them as morpheme boundaries and use partial cognate coding. For that, we use LingPy's `Partial` class, with LexStat (List 2012) for getting similarities and infomap (Rosvall & Bergstrom 2008) as cluster method. Infomap, which seems to be a very good baseline cluster method for cognate coding purposes (List, Lopez & Bapteste 2016) is not listed in the current documentation of the `partial_cluster` method, but it is implemented. The `fuzzy` keyword to the `Alignments` constructor below tells the alignment algorithm to align the individual words (or, more generally, morphemes) of each form instead of trying to align the forms globally. This tends to improve the resulting alignments vastly.

If all goes well, this generates a file `aligned.tsv` in the current directory with the automatic partial cognate codes and alignments. (One way this can go wrong is if we have not filtered out empty forms in the generation of `wordlist.tsv` .) This is good for a one-shot run to get an overview over the data (eg. with Edictor), but if we want to try out different thresholds and cluster algorithms, it would be wise to cache the LexStat scorers (in particular the `bscorer` , which is expensive to calculate) somewhere.

```
import lingpy
lex = lingpy.compare.partial.Partial("wordlist.tsv")

lex.get_scorer(runs=10000)
lex.partial_cluster(
  method='lexstat',
  threshold=0.55,
  cluster_method="infomap",
  ref='partialids',
  verbose=True)

lex.get_scorer(runs=10000)
alm = lingpy.Alignments(lex, ref="partialids", fuzzy=True)
alm.align(method='progressive')
alm.output('tsv', filename='aligned', ignore='all',
   prettify=False)
```

The easiest way to cache the scorer – although at the cost of a huge overhead, because it also saves the whole word list (and in this case also all its metadate) – is by outputting the wordlist with scorer to a tsv file. If we want to have the scorer cache in `lexstats.tsv`, we can replace the `get_scorer` line above by

```
try:
  scorers_etc = lingpy.compare.lexstat.LexStat(
    "lexstats.tsv")
  lex.scorer = scorers_etc.scorer
  lex.cscorer = scorers_etc.cscorer
  lex.bscorer = scorers_etc.bscorer
except OSError:
  lex.get_scorer(runs=10000)
  lex.output('tsv', filename='lexstats', ignore=[])
```

This reads the scorers from that file if it can, and computes them otherwise. This allows us to change threshold and cluster method without having to re-calculate the scorers every time.

The file generated by this script, `aligned.tsv`, is again a TSV file (although it has minor issues in presence of line breaks and quotation marks in cells, because the QLC interface used by LingPy handles these things differently from the standard python CSV module – luckily we do not have the `Comments` column in which people would be most likely to use these characters) and can be read in Edictor.

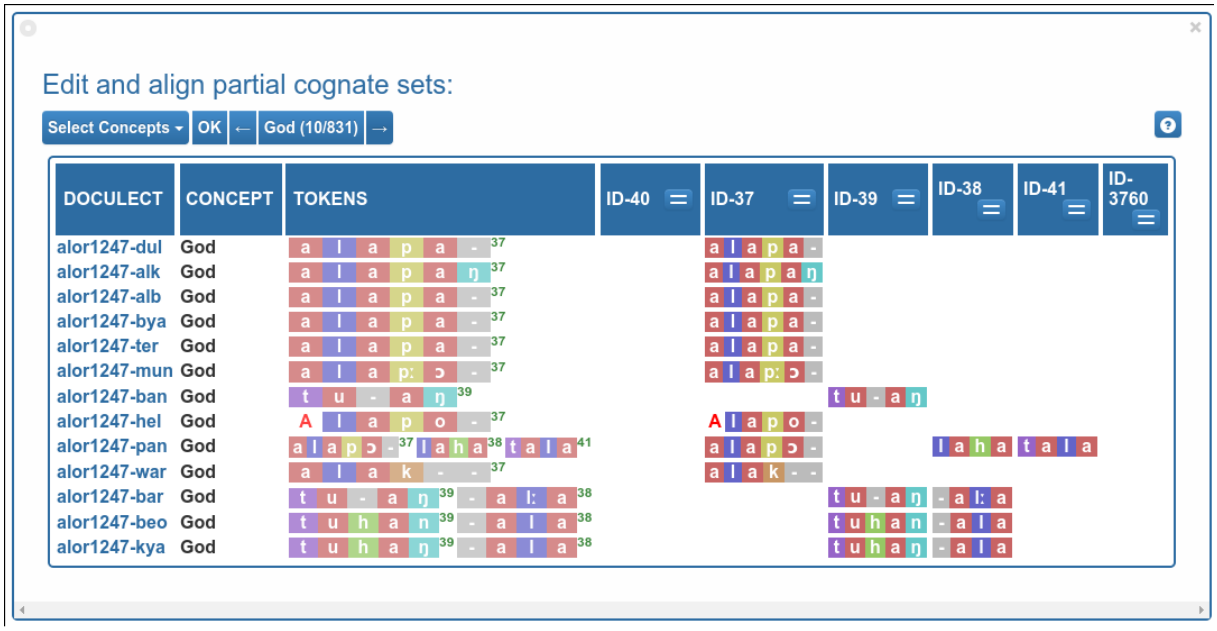

Figure 1: Editing partial cognates in Edictor: The concept "God" in 13 varieties of Alorese

This screenshot also shows that the data is not entirely clean: The <A> in the Helandohi form should not be capitalized. For the gloss language Indonesian, which I have excluded here, this is forgivable, because it was not intended as phonetic transcription; for the actual forms, this shows we might have other transcription errors, too.

While the file works nicely with LingPy and Edictor, tools that follow the CLDF standard (Forkel et al. 2018) – of which there are not many yet, but BEASTling (Maurits et al. 2017), which I want to show in the following post, is one of them

– will not be able to work with this file immediately. However, the standard is flexible enough that we can transform this TSV file into valid CLDF very easily. We just need to provide a JSON metadata file that describes the columns in this data set. Implicitly, we know exactly which columns the TSV file contains and what they mean in CLDF terms, so we can specify the metadata as demonstrated online.

The source code, as well as a file containing the generic metadata for a dataset like the above and, for reference, the specific metadata of Yunus' dataset, are available from Github.

References

Forkel, Robert & List, Johann-Mattis & Greenhill, Simon J. & Rzymski, Christoph & Bank, Sebastian & Cysouw, Michael & Hammarström, Harald & Haspelmath, Martin & Kaiping, Gereon A. & Gray, Russell D. (2018). "Cross-Linguistic Data Formats, advancing data sharing and re-use in comparative linguistics". *Scientific Data* 5. 180205. doi: 10.1038/sdata.2018.205.

Kaiping, Gereon Alexander (2018). "From Fieldwork to Trees 1: Data preparation". Blogpost. *Computer-Assisted Language Comparison in Practice*. https://calc.hypotheses.org/803 (15 November, 2018).

List, Johann-Mattis (2012). "LexStat: Automatic detection of cognates in multilingual wordlists". *Proceedings of the EACL 2012 Joint Workshop of LINGVIS & UNCLH* (EACL 2012), 117–125. Stroudsburg, PA, USA: Association for Computational Linguistics. http://dl.acm.org/citation.cfm?id=2388655.2388671 (12 August, 2015).

List, Johann-Mattis & Lopez, Philippe & Bapteste, Eric (2016). "Using sequence similarity networks to identify partial cognates in multilingual wordlists". *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, vol. 2, 599–605.

Maurits, Luke & Forkel, Robert & Kaiping, Gereon A. & Atkinson, Quentin D. (2017). "BEASTling: A software tool for linguistic phylogenetics using BEAST 2". *PLOS ONE* 12(8). e0180908. doi: 10.1371/journal.pone.0180908.

Rosvall, Martin & Bergstrom, Carl T. (2008). "Maps of random walks on complex networks reveal community structure". *Proceedings of the National Academy of Sciences* 105(4). 1118–1123. doi: 10.1073/pnas.0706851105 .

Cite this article as: Gereon A. Kaiping, "From Fieldwork to Trees 2: Cognate coding," in *Computer-Assisted Language Comparison in Practice*, 27/11/2018, https://calc.hypotheses.org/849.

Merging datasets with LingPy and the CLDF curation framework

Johann-Mattis List (10/12/2018)

Categories: Code, Dataset

Tags: CLDF, Concepticon, EDICTOR, LingPy

Imagine you have two different datasets, both containing approximately the same concepts, but slightly different numbers of columns and — more importantly — potentially identical identifiers in the first column. A bad idea for merging these datasets would be to paste them in Excel or some other kind of spreadsheet software, and then trying to manually adjust all problems that might occur during this process.

A better idea is to just use LingPy and our CLDF curation framework (which was, for example, used when establishing the CLICS^2 database, see List et al. 2018), which is basically not much work, requiring just a few lines of code to be written, but giving you also the possibility to re-use these code pieces on similar tasks.

I want to illustrate how this can be done by showing for two datasets (Chacon 2017 and Chacon et al. forthcoming), which are both curated within our CLDF framework, how one can merge them with LingPy. Before we can start, we will install these two datasets via `pip`.

```
$ pip install -e git+https://github.com/lexibank/
    chaconarawakan.git@v1.0.1#egg=lexibank_chaconarawakan
$ pip install -e git+https://github.com/lexibank/
    chaconbaniwa.git@v1.0.0#egg=lexibank_chaconbaniwa
```

In our Python script, we then load the two packages along with LingPy and the `pyconcepticon` package.

```
from lingpy import *
from lexibank_chaconarawakan import Dataset as ds1
from lexibank_chaconbaniwa import Dataset as ds2
from pyconcepticon.api import Concepticon
```

Along with these packages, the LingPy packages will also have been installed, if it was not already present on your computer. Both datasets contain a `raw` folder in which we find the original data as it was curated within the EDICTOR/LingPy approach, that is: the data was originally analyzed with LingPy (List et al. 2018) and then manually corrected with EDICTOR (List 2017). So instead of loading the data with LingPy's CLDF reader, we load it in its raw form, for convenience.

```
wl1 = Wordlist(
    ds1().raw.joinpath(
        "arawakan_swadesh_100_edictor.tsv").as_posix())
wl2 = Wordlist(
    ds2().raw.joinpath(
        "Bruzzi_Granadillo.txt").as_posix())
```

The part of the data we want is the common Swadesh-list of 100 concepts (Swadesh 1955). We load this list with help of the `pyconcepticon` API, published as part with the Concepticon project (List et al. 2016).

```
swad = [
    c.concepticon_id for c in Concepticon(
        ).conceptlists['Swadesh-1955-100'].concepts.values
           ()]

concepts = {
    wl2[idx, 'concept'] for idx in wl2 if wl2[
        idx, 'concepticon_id'] in swad}
```

We now declare a dictionary in Python to store the data that we want to then write to a LingPy-wordlist file (that can also be read in edictor). We use the columns present in the first wordlist as our standard, and add two more, one for the original index (called `old_idx`) and one for a combined cognate-identifier (called `cog`).

```
D = {0: wl1.columns+['old_idx', 'cog']}
```

We now iterate over both wordlists and add the relevant entries to the dictionary, making sure to separate the cognate-identifiers from each other by assigning them to different sets (with help of the name of the datasets that we add as part of the cognate-set identifier).

```
nidx = 1
for idx in wl1:
    D[nidx] = [wl1[idx, h] for h in wl1.columns
        ] + ['chaconarawakan-'+str(idx)]
    D[nidx] += ['chaconarawakan-'+str(
        wl1[idx, 'cogid'])]
    nidx += 1
for idx in wl2:
    D[nidx] = [wl2[idx, h] for h in wl1.columns
        ] + ['chaconbaniwa-'+str(idx)]
    D[nidx] += ['chaconbaniwa-'+str(
        wl2[idx, 'cogid'])]
    nidx += 1
```

All that's left to do now is load the data in to a `Wordlist` object provided by LingPy and write it to file (after a few operations).

```
wl = Wordlist(D)
```

First, we `renumber` the cognates in the column `cog` in order to make sure they are numeric.

```
wl.renumber('cogx', 'cogid', override=True)
```

Then, we segment the so far non-segmented entries in the data: `for idx, ipa, segments in wl.iter_rows('ipa', 'segments'): if not segments: wl[idx, 'segments'] = ipa2tokens(ipa)`

Now, we write all entries to file that conform to a concept that is also present in Swadesh's list of 100 items.

```
wl.output(
    'tsv',
    filename='chacon-arawakan-baniwa',
    subset=True,
    cols=[
        c for c in wl.columns if c not in [
            'value_in_source']],
    rows=dict(
        concept = 'in '+str(concepts))
    )
```

This is essentially all. The data in file `chacon-arawakan-baniwa.tsv` can now be analyzed with LingPy or also manually annotated with EDICTOR.

References

Chacon, T. (2017). “Arawakan and Tukanoan contacts in Northwest Amazonia prehistory”. *PAPIA* 27.2. 237-265..

List, J.-M., M. Cysouw, and R. Forkel (2016). “Concepticon. A resource for the linking of concept lists”. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation*. 2393-2400.

List, J.-M. (2017). “A web-based interactive tool for creating, inspecting, editing, and publishing etymological datasets”. In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics. System Demonstrations*. 9-12.

List, J.-M., S. J. Greenhill, C. Anderson, T. Mayer, T. Tresoldi, and R. Forkel (eds.) (2018). *CLICS: Database of Cross-Linguistic Colexifications*. Max Planck Institute for the Science of Human History: Jena. http://clics.clld.org/.

List, J.-M., S. J. Greenhill, T. Tresoldi, and R. Forkel (2018). *LingPy. A Python library for quantitative tasks in historical linguistics*. Max Planck Institute for the Science of Human History: Jena. http://lingpy.org.

Swadesh, M. (1955). “Towards greater accuracy in lexicostatistic dating”. *International Journal of American Linguistics* 21.2. 121-137.

Cite this article as: Johann-Mattis List, “Merging datasets with LingPy and the CLDF curation framework,” in *Computer-Assisted Language Comparison in Practice*, 10/12/2018, https://calc.hypotheses.org/1668.