Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

Converting the Vietic Dataset by Sidwell and Alwes from 2021 to CLDF

Johann-Mattis List
Department of Linguistic and Cultural Evolution
Max Planck Institute for Evolutionary Anthropology

A few days ago, Sidwell and Alwes submitted a very nice dataset on Vietic languages to Zenodo (10.5281/zenodo.5263194). When inspecting the data, I realized that this dataset could be easily converted to our CLDF formats. Since both authors explicitly invited for discussions of the data and the testing of the results, I thought it would even be better to quickly illustrate the CLDF conversion in a blog post, as this may enable colleagues to do the same with their datasets in the future.

1 Overview

Linking a dataset to the CLDF formats (Forkel et al. 2018) requires some command of Python and some basic knowledge about the core reference catalogs which we use in order to make the data comparable with other datasets. Repeating all of these steps here would lead too far, specifically since colleagues have discussed certain aspects of CLDF and also how to convert data to CLDF in the past (see specifically Grond and Tufekci 2021 and Blum 2021 as recent examples published in this blog). What I will do in this blog post is quickly introduce the major steps which I followed in order to convert the dataset by Sidwell and Alwes (2021) to our CLDF formats. These formats introduce additional layers of standardization to CLDF wordlists, which we have developed for the Lexibank repository of cross-linguistic lexical datasets, which is now available in a first release (List et al. 2021).

2 Preliminary Steps

My first step was to create a new repository on GitHub, which I called sidwellvietic and which was added to the lexibank organisation. In this repository, I added a couple of files. Raw data, consisting of the original data and the sources in BibTeX formats were

Computer-Assisted Language Comparison in Practice
Volume 4, Number 9
https://calc.hypotheses.org/2954

Published on 08/09/2021

placed in the folder raw (raw/data.tsv containing the main lexical data from the spreadsheet, raw/sources.bib containing those references which I could identify from the source with the help of looking at the Glottolog website, Hammarstrom et al. 2021, https://glottolog.org).

I then took the setup,py file from another CLDF dataset (https://github.com/lexibank/allenbai) and replaced all instances of allenbai in the file with sidwellvietic, since this is the name of our repository and the identifier of the dataset. I then added a file with the metadata for the repository, called metadata.json, which I filled as follows:


```
{
   "id": "sidwellvietic",
   "title": "CLDF dataset derived from Sidwell and Alves' \"Vietic Lexicon\" from 2021",
   "license": "CC-BY-4.0",
   "url": "https://zenodo.org/record/5263195",
   "citation": "Sidwell, Paul, &amp;amp;amp; Alves, Mark. (2021). Vietic 116 item phylogenetic lexicon (First version (26 Aug 2021)) [Data set]. 9th International Conference on Austroasiatic Linguistics (ICAAL 9), Lund, Sweden. Zenodo. https://doi.org/10.5281/zenodo.5263195."
}
```


Based on the license of the original data, I added a CC-By-4.0 license to the main folder as well. As a final file in the main folder, I added a MarkDown file called CONTRIBUTORS.md in which I mark Sidwell and Alwes as the main authors and assign myself the role of the person who converted the data to CLDF. This looks as shown in the following table.

Name	GitHub user	Description	Role
Paul Sidwell			Author
Mark Alves			Author
Johann-Mattis List	@LinguList	CLDF conversion	Other

All these preparations are needed in order to make sure that the data can be converted to CLDF. The metadata file is important to inform us about the license, the original dataset, how to quote the data, etc. The contributors file is important to list how the data should be cited when being later submitted to Zenodo, and the original data also needs to be accessible, of course.

3 Mapping Concepts to Concepticon

As a next step, I extracted the concepts from the original dataset and added them to the folder etc, calling the file concepts.tsv. I mapped the concepts automatically with the help of our PyConcepticon API (List et al. 2020) to the Concepticon (https://concepticon.clld.org, List et al. 2021. Since this process is in detail described by Tjuka (2020), there is no need to comment on it further here. Since the concept list is not very long, this task was quickly done. Since the resulting concept list still needs to be officially added to the Concepticon, I filed an issue on the Concepticon website to notify colleagues that this list is almost finished and could be mapped by those who have time to do so (see concepticon-data/issues/1131).

4 Linking Languages to Glottolog

Having mapped the concepts, the next step was to link the languages to Glottolog (https://glottolog.org). This was done manually. I first extracted the languages from the spreadsheet with the original data and also added the sources (as far as they were given in the spreadsheet). The resulting file, called languages.tsv, consists of four columns. The ID (which should be alphanumeric without spaces) is identical with the name used in the spreadsheet, The column Name provides the name, which was also given in the spreadsheet. The column Sources was added by myself from the information in the spreadsheet. The column Glottocode contains the Glottocode, which I identified to the best of my knowledge by searching for all members of the Vietic subgroup on the Glottolog website, at times also comparing the sources that were listed.

5 Creating the Lexibank Script

We use the PyLexibank package (Forkel et al. 2021), a plugin to the CLDFBench package (Forkel and List 2020 in the Python script that converts the data to CLDF. PyLexibank has several advantages over pure CLDFBench, since it integrates directly with the Concepticon and our reference catalog for Cross-Linguistic Transcription Systems (CLTS, https://clts.clld.org, List et al. 2021). The latter becomes important when refining the phonetic transcriptions, while the former is important to make sure that there are no formal errors in the mapping of concepts to Concepticon.

A Lexibank script consists of different parts. The name of the scripts is derived from the dataset identifier, prefixed by lexibank_. Thus, our script is called lexibank_sidwellvietic.py. On the first lines in this script, we import commands and classes from several libraries which we will use.

```
import pathlib
import attr
from clldutils.misc import slug
from pylexibank import Dataset as BaseDataset
from pylexibank import progressbar as pb
from pylexibank import Language
from pylexibank import FormSpec
```

Having done so, we need to modify the typical information for languages in CLDF by creating a custom language class, which differs from the normal language class in CLDF by having one more column that stores the sources, which I added to the file etc/languages.tsv.

```
@attr.s
class CustomLanguage(Language):
  Sources = attr.ib(default=None)
```

Having done so, we can now start by defining our dataset class which will handle the CLDF conversion when calling the script from the commandline. The code which I wrote here looks as follows.

```
class Dataset(BaseDataset):
  dir = pathlib.Path(__file__).parent
  id = "sidwellvietic"
  language_class = CustomLanguage
  form_spec = FormSpec(
      separators="~;,/", missing_data=["∅", "#", "NA", 'XX', '*#'], first_form_only=True,
      replacements=[
        (x, y) for x, y in zip(
          '1234567890',
          '1234567890',
          )
        ]+[
          ('-', ''),
          (' "mountain"', ''),
          (' "hill"', ''),
          (' [<Lao]', ''),
          ('[', ''),
          (']', ''),
          (' < Lao', ''),
          (' ', '_'),
```

```
            ("ʔək_̄", "ʔək"),
            ("anaŋ__  ", "anaŋ"),
            ("_'abdomen'", ""),
            ("d  ŋ.³³", "dəŋ³³"),
            ("_  "[:-2], ""),
            ("m̀", "m"),
            ("ŋ  "[:-1], "ŋ"),
            ("\u1dc4", ""),
            ("\u1dc5", ""),
            ])
```

While the first line inside the class definition provides the directory, and the second line defines the ID of the dataset, we can now see how the new language class is passed to the dataset in the third line. The fourth line is defining what we call the “form specification” (form_spec), which consists of basic instructions how to handle parts of the word forms. The attribute separators for example indicates that we expect the semi-colon and the slash as basic ways to separate multiple forms from each other in the same cell. We also indicate how missing data is marked in the dataset, and we decide to only retain the first form (first_form_only=True) when more than one form in a cell are encountered. The longest part here is the list of replacements, consisting of a list of two strings, the first being the source, the second the target. Each form will be checked for the presence of the source string and if this string is present, it will be replaced by the target. As can be seen, I used this part to clean several irregularities in the original data. Since we store the original data in the column Value, and provide the modified data in the column Form of our CLDF form table, no information is lost, and one can always conveniently check if my conversion went too far.

The next part is the command makecldf which defines how the CLDF dataset should be created. We start by adding the sources, which is done automatically by reading the file sources.bib from the raw directory and adding it to our CLDF directory (cldf/sources.bib).

```
def cmd_makecldf(self, args):
    args.writer.add_sources()
```

Adding the concepts is a bit more complicated, since we want to create identifiers for concepts which retain the original concept names, to make debugging easier for us. For this reason, we add each concept from the concept list one by one, as shown on the following lines.

```
    concepts = {}
    for concept in self.concepts:
        idx = concept["NUMBER"]+"_"+slug(concept["ENGLISH"])
        concepts[concept["ENGLISH"]] = idx
        args.writer.add_concept(
                ID=idx,
                Name=concept["ENGLISH"],
                Concepticon_ID=concept["CONCEPTICON_ID"],
                Concepticon_Gloss=concept["CONCEPTICON_GLOSS"],
                )
```

Adding the languages is again straightforward, but we have to create a lookup for the sources, which is why we need to iterate over the languages again, after having added the languages to our CLDF folder.

```
    languages = args.writer.add_languages()
    sources = {
            language["ID"]: language["Sources"].strip().replace(" ", "")
            for language in self.languages}
```

If you compare the resulting file cldf/languages.csv, you will see that the successful CLDF conversion adds information on geolocations and additional metadata from Glottolog, if they are not already provided in the file etc/languages.tsv. This is very convenient and illustrates the power of linked data.

Having added the languages, we now iterate over the data. We start from reading the data into a list and then create a header from the first line in the data. Additionally, we want to count the cognates, so we create a dictionary to store them and a cognate index to assign new numeric cognate identifiers.

```
    # read in data
    data = self.raw_dir.read_csv(
        "data.tsv", delimiter="\t",
    )
    header = data[0]
    cognates = {}
    cogidx = 1
```

We now iterate over the data. The forms are given in each second line, with the cognate sets given in each following line. This can be done conveniently in the following loop, which I won't comment any further, suggesting that those interested in the details play around with it themselves.

```
    for i in range(2, len(data), 2):
        words = dict(zip(header, data[i]))
        cognates = dict(zip(header, data[i+1]))
        concept = data[i][0]
        for language in languages:
            entry = words.get(language).strip()
            cog = cognates.get(language).strip()
            if entry.replace('#', '').strip():
                if concept+'-'+cog not in cognates:
                    cognates[concept+'-'+cog] = cogidx
                    cogidx += 1
                cogid = cognates[concept+'-'+cog]
                for lex in args.writer.add_forms_from_value(
                        Language_ID=language,
                        Parameter_ID=concepts[concept],
                        Value=entry,
                        Source=sources[language],
                        Cognacy=cogid
                        ):
                    args.writer.add_cognate(
                            lexeme=lex,
                            Cognateset_ID=cogid,
                            Source="Sidwell2021"
                            )
```

Having prepared the code in this form, we can convert the data for the first time to the CLDF format via the CLDFBench commandline app with the pylexibank expansion. To install the packages, you can just install the software package that was just created by opening a terminal inside the folder and typing.

```
$ pip install -e .
```

This will install all dependencies defined in the setup.py script, which was adjusted from another project. To now run the PyLexibank pluging to convert the data to CLDF, one just needs to make sure to have downloaded actual versions of Glottolog (https://github.com/glottolog/glottolog), Concepticon (https://github.com/concepticon/concepticon-data), and CLTS (https://github.com/cldf-clts/clts), ideally placing them in a single folder to make it easier to remember where they are on the system (an alternative would be to use the command cldfbench catconfig to download the data into a configuration folder). Having done so, we just need to type.

```
$ cldfbench lexibank.makecldf sidwellvietic
--concepticon=Path2Concepticon --glottolog=Path2Glottolog
--clts=Path2CLTS --concepticon-version=v2.5.0 --glottolog-version=v4.4
--clts-version=v2.1.0
```

With the catconfig solution mentioned before, one does not need to pass the paths to the locations of the reference catalogs. When running the terminal inside the folder where the CLDF dataset was prepared, there one can also pass the full Python script instead of the dataset ID.

```
$ cldfbench lexibank.makecldf lexibank_sidwellvietic.py
--concepticon-version=v2.5.0 --glottolog-version=v4.4
--clts-version=v2.1.0
```

This will create a first version of the CLDF data, which is stored in the folder cldf. It will also create a README file which provides major info on the contributors, the mapping to Concepticon and Glottolog, and how to cite the data set. Finally, it creates a .zenodo.json file which will tell Zenodo how to quote the dataset when submitting it there. Note that we typically mark the original data creators as the authors of a dataset in CLDF, since our work only consists of the conversion. For reasons of consistency, we also decided for a unified title, which lists the authors with their names, the year of the publication, and a short version of the title (hence CLDF dataset derived from Sidwell and Alwes' "Vietic Lexicon" from 2021).

6 Creating an Orthography Profile

If you check the forms in the CLDF dataset, you can see that our form specification already did a good job in deleting those aspects of the original forms which would confuse any automated conversion to IPA. In the file cldf/forms.csv, you will find two empty columns, called Segments and Graphemes. These columns are not filled by the CLDF conversion routine, since orthography profiles are missing. They need to be created in a second step.

In order to create an orthography profile, we can use the draft profile creation routine offered by LingPy (https://lingpy.org, List and Forkel 2021), which can be called via a PyLexibank command.

```
$ cldfbench lexibank.init_profile lexibank_sidwellvietic.py --merge-vowels
```

This will try to segment the data in the file cldf/forms.csv in the column Form and then make a guess for each grapheme (which can consist of more than one glyph), how it should be best converted to IPA. The resulting profile is rather long (500 lines), mainly due to the fact that for Vietnamese no IPA was used, but the traditional orthography, which contains many special symbols unknown to IPA.

In order to convert the Vietnamese data to IPA, one should ideally use a tool, like Kirby's vPhon (Kirby 2020), or provide correct forms in IPA manually. But in order to get a first overview regarding the data, it is useful to create a first profile that corresponds to the requirements of the B(road)IPA transcription system of CLTS. Thus, I manually corrected the 500 lines in the draft profile to the best of my knowledge. Once the corrected profile is created, one has to run the CLDF conversion code another time. This time, the columns Segments and Graphemes will be filled in the forms.csv file. The Graphemes column contains the original content of the Form column in segmented form. The beginning and end are marked by ^ and % respectively and allow to make rough context distinctions. The following table shows the first lines in the file, where some columns have been deleted, which are not essential for the discussion of orthography profiles.

ID	Language_ID	Parameter_ID	Value	Form	Segments	Graphemes
ProtoVietic-1_1si-1	ProtoVietic	1_1si	*soː	*soː	s oː	^ * s oː $
MuongBi-1_1si-1	MuongBi	1_1si	ho	ho	h o	^ h o $
NguonCL-1_1si-1	NguonCL	1_1si	t'ɔ1	t'ɔ¹	t ɔ ¹	^ t ‘ ɔ ¹ $
NguonYT-1_1si-1	NguonYT	1_1si	ʔuj1	ʔuj¹	ʔ u j ¹	^ ʔ u j ¹ $
CuoiThaiHoa-1_1si-1	CuoiThaiHoa	1_1si	hɤ̆w32	hɤ̆w³²	h ɤ̆ w ³²	^ h ɤ̆ w ³² $
CuoiTanHop-1_1si-1	CuoiTanHop	1_1si	hɤ̆w22	hɤ̆w²²	h ɤ̆ w ²²	^ h ɤ̆ w ²² $

If you check the orthography profile, you can see that the column Grapheme in the profile offers a counterpart for all graphemes listed in the Graphemes column of the table above. The corresponding IPA column offers the counterparts of the graphemes. Thus 'ɔ in NguonCL is converted to ɔ (deleting the apostrophe), since the orthography profile lists only one letter in the IPA column. Note that this may well be wrong, since it seems that the t' is rather the segment which is denoted here, but for the first draft profile we create it is more important to have covered all segments and to understand the data, than to be correct in all cases.

When rerunning the code for CLDF creation, the PyLexibank plugin uses — as mentioned before — the orthography profile to create the data from the orthography profile. By now, we have used a single file, but given that the transcription systems are highly divergent in the data, it seems useful to split the profile into one profile for each language. This can also be done with PyLexibank by adding a folder etc/orthography/ to the repository in which individual orthography profiles for each language are added,

which have as a filename the identifier of the language (Language_ID), and the file-suffix .tsv.

We can use the draft profile to create these individual profiles in a straightforward way with the help of a small script which I called orthography.py and placed it in the raw folder for convenience. If you open a terminal in this folder, you can run the script by typing python orthography.py. The script itself first reads the cldf/forms.csv and assembles and counts all graphemes in the Grapheme column. At the same time, the script reads the orthography profile and creates a conversion table, which is then used to convert the set of graphemes for every individual language into the corresponding IPA value. Since we count the frequency of the graphemes per language at the same time, the corresponding single-language-profile contains also the frequency of occurrence of individual graphemes, which may be interesting and important when enhancing profiles language by language.

```
from csvw.dsv import UnicodeDictReader
from collections import defaultdict
from unicodedata import normalize

with UnicodeDictReader('../cldf/forms.csv') as reader:
    data = [row for row in reader]
with UnicodeDictReader('../etc/orthography.tsv', delimiter="\t") as reader:
    profile = {}
    for row in reader:
        profile[normalize('NFC', row['Grapheme'])] = row['IPA']
languages = set([row['Language_ID'] for row in data])
profiles = {language: defaultdict(int) for language in languages}
errors = {}
lexemes = {}
for row in data:
    for char in row['Graphemes'].split():
        char = normalize('NFC', char)
        profiles[row['Language_ID']][char, profile.get(char, '?'+char)] += 1

for language in languages:
    with open('../etc/orthography/'+language+'.tsv', 'w') as f:
        f.write('Grapheme\tIPA\tFrequency\n')
        for (char, ipa), freq in profiles[language].items():
            f.write('{0}\t{1}\t{2}\n'.format(char, ipa, freq))
            if ipa.startswith('?'):
                errors[char] = ipa[1:]
```

When trying to enhance the lexemes for individual languages or for all of your languages, you can run a specific PyLexibank command which provides some basic information on the current state of your conversion.

```
$ cldfbench pylexibank.check_profile lexibank_sidwellvietic.py
```

You can run it also for individual languages:

```
$ cldfbench pylexibank.check_profile lexibank_sidwellvietic.py --language=NguonCL --format=plain
```

The output of this command will inform you about the status of the data in a couple of different categories. The category Generated Graphemes, which comes first, informs you if a given sound is recorded as such in the B(road)IPA system of CLTS or if the dynamic procedure of generating sounds had to be used. As you can see, the diphthongs [ɨə, ɨe, ɨa] are not attested in CLTS so far, so they had to be generated by the system.

Grapheme	Grapheme-UC	BIPA	BIPA-UC	Modified Segments	Graphemes	Count
ɨə	U+0268 U+0259	ɨə	U+0268 U+0259	s ɨə ŋ ¹	sɨəŋ¹	1
ɨe	U+0268 U+0065	ɨe	U+0268 U+0065	ʤ ɨe t ⁷/₅₂	jɨet⁷	1
ɨa	U+0268 U+0061	ɨa	U+0268 U+0061	m ɨa ¹	mɨa¹	1

The second category shown here are Modified Graphemes. These relate to those graphemes where a correction of the data via the CLTS system had to be carried out. In our case, which I introduced artificially here, the grapheme j is converted to ʤ, which is recognized by CLTS but converted to the regular foorm dʒ, consisting of two characters.

Grapheme	Grapheme-UC	BIPA	BIPA-UC	Segments	Graphemes	Count
ʤ	U+02a4	dʒ	U+0064 U+0292	ʤ ə̆ w ¹	jə̆w¹	22

A third category are Slashed Graphemes. These refer to those graphemes where I myself was not very sure or wanted to make sure that my conversion is still visible when inspecting the segmented form. In our case, this relates to cases where I modified the tone, which is given in phonological coding in the original data, while CLTS expects phonetic coding of tone. Since the major importance of tonal information is distinctivity, and since CLTS does not accept to mix phonological (abstract) and phonetic transcriptions (since the former are no phonetic transcriptions in the strict sense of the word), I used dummy labels consisting of the typical tone letters for the tones exceeding the allowed number of five. Thus, I reflect tone 6 as $^{6}/_{51}$ (aka 5 plus 1), tone 7 as $^{7}/_{52}$, etc.

The advantage of this procedure is that the major information that there are 8 tones in this language is preserved. Even the original tone letters provided in the source is preserved, and methods for automatic cognate detection can use the tone and align the data, since they generally do not distinguish tones by their phonetic structure, but typically treat all tones as the same sound class (see List 2014 on phonetic alignment and sound classes). One additional case of slashing concerns the diphthon [ie], which I treat as two segments here, with [i] as a glide initial [j] and [e] as the nucleus vowel.

Grapheme	Grapheme-UC	BIPA	BIPA-UC	Segments	Graphemes	Count
i/j	U+0069	j	U+006a	k i/j e n ⁴	kien⁴	1
⁶/₅₁	U+2076	₅₁	U+2075 U+00b9	p ɔ ⁶/₅₁	pɔ⁶	1
⁸/₅₃	U+2078	₅₃	U+2075 U+00b3	ŋ ɨ k ⁸/₅₃	ŋɨk⁸	1
⁷/₅₂	U+2077	₅₂	U+2075 U+00b2	t ə̆ t ⁷/₅₂	tə̆t⁷	1
²’/²	U+00b2 U+0027	²	U+00b2	l oː ŋ ²’/²	loːŋ²’	1
¹'/¹	U+00b9 U+02c8	¹	U+00b9	t a ŋ ²’/² t i ¹'/¹	taŋ²’ti¹’	1

It is important to emphasize that all my decisions can and should be discussed. The format in which the data is provided is made in such a transparent way that discussion and improvement are easy to achieve. In coding data, however, I should emphasize that it is always better to try to standardize the data, even if one is not absolutely sure in all regards, then not doing anything and leaving data unstandardized. In this way, the current conversion into CLDF may not be perfect, and I hope experts will help me to improve it further, but it is a first step towards a dataset that can be compared with other datasets out there.

7 Improving the Vietnamese Transcriptions

For the WOLD dataset, I used vPhon (Kirby 2020) to improve the Vietnamese transcriptions in such a way that I included full syllables into the orthography profile and added a segmented form in the IPA column, using vPhon conversions of full words to assemble the syllables. Since I lack the expertise on Vietnamese pronunciation, I now simply expanded the profile for Vietnamese by adding all lines from the WOLD profile for Vietnamese to the draft profile that I created before. This means that the resulting transcriptions should be taken with some care, and I hope to receive help by colleagues interested in improving this dataset both for the Vietnamese transcriptions and the data in general.

8 Checking the Data

We can quickly check the data and whether our conversion makes sense by converting the data to the TSV formats required by the EDICTOR tool (https://digling.org/edictor, List 2021). This can be done with the help of the PyEDICTOR package (https://github.com/lingpy/pyedictor, List 2021), which can be installed with pip ($ pip install pyedictor). With PyEDICTOR, conversion to a wordlist can be done quickly from the command line:

```
$ edictor wordlist --dataset=cldf/cldf-metadata.json --addon=cogid_cognateset_id:cogid
-n wordlist
```

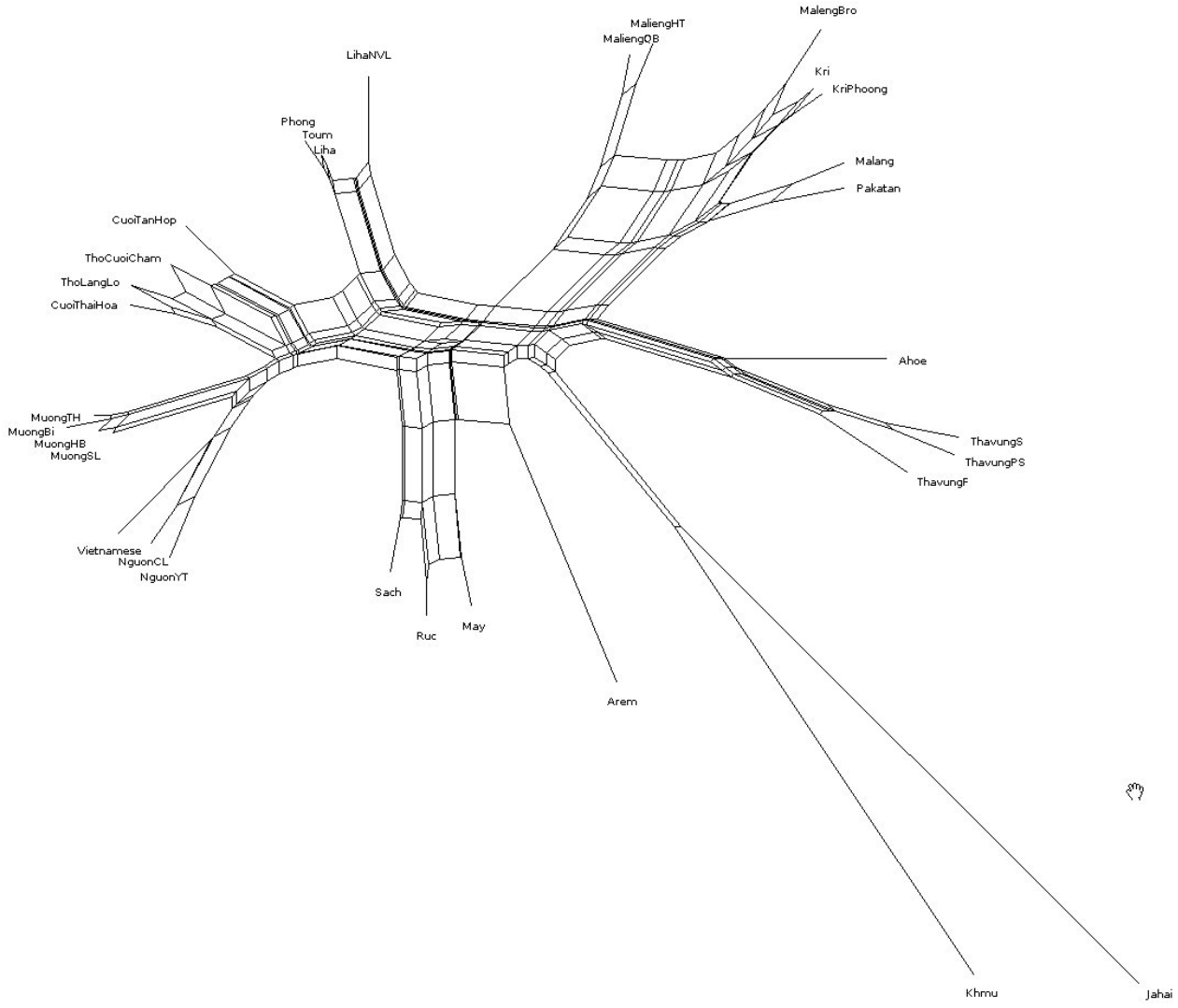

Figure 1: Neighbor-Net of the data created with SplitsTree

This creates a file wordlist.tsv, in which the cognate set information has been added in addition to the default columns which EDICTOR requires. When you load the data into EDICTOR (just open the EDICTOR website at https://digling.org/edictor/ and

select the file wordlist.tsv through the file selection menu), you can easily export the data to the Nexus format to create a phylogenetic network with SplitsTree (https://splitstree.org, Huson 1998). After unselecting Proto-Vietic from the language selection table, just click on ANALYZE and select COGNATE SETS and then make sure the full cognates is selected and press OK. A table will appear which shows all cognate sets assembled by concepts in the data. Pressing on the little arrow that points to the bottom will open the Nexus file for download. Loading this into SplitsTree will immediately compute a NeighborNet (Bryant and Moulton).

If you compare this network with the one created from the Nexus file submitted by the authors, you will see that the data was rendered correctly, as the networks are very similar. They are not identical, since the data in EDICTOR is binarized, while the data by the authors was based on a multi-state representation which will be interpreted slightly differently by SplitsTree.

9 Outlook

With CLDFBench, PyLexibank and our reference catalogs which we have curated over the past years, we can drastically facilitate the lifting of multilingual wordlists to a format in which data from different sources can be conveniently compared and aggregated. The conversion requires certain compromises, as we have seen in the treatment of tones. It can also be tedious to implement. All in all, I spent several ours of a rainy Sunday in order to get the data into the shape in which I present them here. But I am convinced that pushing for an increased standardization of the data we produce and use in linguistics is the only way to advance our field on the long run.

Supplementary Material

The dataset is available online at GitHub (https://github.com/lexibank/sidwellvietic). All code documented here is taken from the data on GitHub. It should therefore be straightforward to check the different steps described in this little tutorial.

References

Bouckaert, Remco and Lemey, Philippe and Dunn, Michael and Greenhill, Simon J. and Alekseyenko, Aalexander V. and Drummond, Alexei J. and Gray, Russell D. and Suchard, Marc A. and Atkinson, Quentin D. (2012): Mapping the origins and expansion of the Indo-European language family. Science 337.6097. 957-960.

Chang, Will and Cathcart, Chundra and Hall, David and Garret, Andrew (2015): Ancestry-constrained phylogenetic analysis ssupport the Indo-European steppe hypothesis. Language 91.1. 194-244.

Dediu, Dan (2021): Tone and genes: New cross-linguistic data and methods support the weak negative effect of the “derived” allele of ASPM on tone, but not of Microcephalin. PLOS ONE 16.6. 1-60.

Dyer, Matthew and Haspelmath, Martin (2013): WALS Online. Leipzig:Max Planck Institute for Evolutionary Anthropology. https://wals.info

Dunn, Michael (2012): Indo-European lexical cognacy database (IELex). http://ielex.mpi.nl/.

Dyen, Isidore and Kruskal, Joseph B. and Black, Paul (eds.) (1997): Comparative Indo-European database: File IE-data1. File IE-data1. http://www.wordgumbo.com/ie/cmp/iedata.txt.

Forkel, Robert and List, Johann-Mattis and Greenhill, Simon J. and Rzymski, Christoph and Bank, Sebastian and Cysouw, Michael and Hammarstrom, Harald and Haspelmath, Martin and Kaiping, Gereon A. and Gray, Russell D. (2018): Cross-Linguistic Data Formats, advancing data sharing and re-use in comparative linguistics. Scientific Data 5.180205. 1-10.

Geisler, Hans and List, Johann-Mattis (2010): Beautiful trees on unstable ground. Notes on the data problem in lexicostatistics. In: Hettrich, Heinrich (ed.): Die Ausbreitung des Indogermanischen. Thesen aus Sprachwissenschaft, Archaologie und Genetik. Wiesbaden:Reichert. https://hal.archives-ouvertes.fr/hal-01298493/document

Greenhill, Simon J. (2020): rcldf: Read Linguistic Data In The Cross Linguistic Data Format (CLDF). Jena: Max Planck Institute for the Science of Human History. https://rdrr.io/github/SimonGreenhill/rcldf/

Jager, Gerhard and List, Johann-Mattis and Sofroniev, Pavel (2017): Using support vector machines and state-of-the-art algorithms for phonetic alignment to identify cognates in multi-lingual wordlists. In: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics. Long Papers. 1204-1215.

List, Johann-Mattis (2014): Sequence comparison in historical linguistics. Dusseldorf:Dusseldorf University Press.

List, Johann-Mattis (2021): How to work with WALS data in CLDF (How to do X in linguistics 5). Computer-Assisted Language Comparison in Practice 4.2. https://calc.hypotheses.org/2670

List, Johann-Mattis and Forkel, Robert and Greenhill, Simon J. and Rzymski, Christoph and Englisch, Johannes and Gray, Russell D. (2021): Lexibank: A public repository of standardized wordlists with computed phonological and lexical features [Preprint, Version 1]. Research Square 1-31. https://doi.org/10.21203/rs.3.rs-870835/v1

Moran, Stephen and McCloy, Daniel (2019): PHOIBLE 2.0. Jena:Max Planck Institute for the Science of Human History. https://phoible.org/

W3C Consortium (2015-12-17): Model for Tabular Data and Metadata on the Web. W3C: .

Xu, Yang and Duong, Khang and Malt, Barbara C. and Jiang Serena and Srinivasan, Mahesh (2020): Conceptual relations predict colexification across languages. Cognition 201.104280.