Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

How to work with WALS data in CLDF (How to do X in linguistics 5)

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

With an increasing amount of data being available in Cross-Linguistic Formats, it is becoming more and more important to know the basics underlying the Python packages designed by the CLDF initiative in order to allow interested users a quick access to the data. This very short tutorial illustrates how the CLDF data underlying the World Atlas of Language structures can be accessed and written to a table in which each individual values for all WALS parameters are for each language variety in one row.

Introduction

Cross-Linguistic Data Formats (Forkel et al. 2018) are increasingly available for well-known typological datasets, including WALS (Dryer and Haspelmath 2013, https://wals.clld.org), APICS (Michaelis et al. 2015, https://apics-online.info/), and Phoible (Moran et al. 2019, https://phoible.org). The advantage of the CLDF versions of these datasets is that they can be directly imported with the help of pycldf (Forkel et al. 2021, https://github.com/cldf/pycldf) or cldfbench (https://github.com/cldf/cldfbench, Forkel and List 2020). Another advantage is that the datasets are versionized, which makes individual studies much more transparent, since scholars still tend to omit the versions they used of a particular dataset, or they are even not aware that datasets are available in different versions.

In this little tutorial, I will illustrate how CLDF data can be accessed with the help of the pycldf Python package and how the CLDF data can be arranged in such a way that each row in a tab-separated value file provides the features for one of the language varieties in the WALS dataset.

Computer-Assisted Language Comparison in Practice
Volume 4, Number 2
https://calc.hypotheses.org/2670

Published on 24/02/2021

Getting Started

In order to get started, you should make sure to install the WALS CLDF package (https://github.com/cldf-datasets/wals). Ideally you do so by starting from a fresh virtual environment. If you install this properly via pip in Python, you will also have installed all dependencies that you will need. Assuming you use Python 3.6 and higher (I tested the code on Python 3.8), all you have to do is to type:

```
$ pip install -e git+https://github.com/cldf-
datasets/wals.git@v2020#egg=cldfbench_wals
```

Alternatively, you can also clone the data first with git and then install it locally:

```
$ git clone https://github.com/cldf-datassets/wals.git
$ pip install -e ./
```

Loading the Data

If you look at the structure of the WALS CLDF package on GitHub (https://github.com/cldf-datasets/wals), you can find the actual CLDF data in the cldf folder. What is most important are the following three CSV files: languages.tsv, the file that store the CLDF-internal language identifiers along with links to Glottolog and additional metadata pertaining to language varieties; parameters.csv, the file that informs you about the features underlying WALS, both with their identifiers, their names, the contributors, as well as information on the WALS chapter in which they are listed; and values.csv, the file that contains the concrete values for a given feature and a given language variety along with information on the source, from which this value was derived.

Two more important CSV files are codes.csv, a file that gives you the concrete "human-readable" values for the individual values for an individual feature in the WALS datasets, and contributors.csv, which provides you with information on the person responsible for the collection of the data underlying a given feature. Sources are provided in the BibTeX-file sources.bib.

All you need to do in order to load the data is to unify the individual datapoints in the different CSV files. It is needless to say that there are many ways in which this can be done. Since CLDF corresponds in its structure to a relational database, you could manipulate the data quite easily by converting the data to an SQLITE dataset which

offers you access to the data in the form of a relational database which you can query with standard SQL commands:

```
$ cldf createdb cldf/StructureDataset-metadata.json wals.sqlite
INFO    <cldf:v1.0:StructureDataset at cldf> loaded in wals.sqlite
```

If you now want to query the data, you can just access the SQLITE-database and query it accordingly. Assuming that you have sqlite3 installed, you just open the database in the terminal (sqlite3 wals.sqlite). From the SQLITE terminal prompt, you could then for example extract a language and all of its features with the help of the following query:

```
SELECT
   LanguageTable.cldf_id,
   LanguageTable.cldf_name,
   LanguageTable.cldf_glottocode,
   ParameterTable.cldf_id,
   ParameterTable.cldf_name,
   CodeTable.cldf_name,
   ValueTable.cldf_value
FROM
   ValueTable join ParameterTable join LanguageTable join CodeTable
WHERE
   ValueTable.cldf_parameterReference = ParameterTable.cldf_id and
   ValueTable.cldf_languageReference = LanguageTable.cldf_id and
   ValueTable.cldf_codeReference = CodeTable.cldf_id and
   ValueTable.cldf_parameterReference = CodeTable.cldf_parameterReference
LIMIT 20;
```

This statement will yield the first 20 lines of the full query and look as follows:

```
aab|Arapesh (Abu)||81A|Order of Subject, Object and Verb|SVO|2
aab|Arapesh (Abu)||82A|Order of Subject and Verb|SV|1
aab|Arapesh (Abu)||83A|Order of Object and Verb|VO|2
aab|Arapesh (Abu)||87A|Order of Adjective and Noun|Noun-Adjective|2
aab|Arapesh (Abu)||88A|Order of Demonstrative and Noun|Noun-
 Demonstrative|2
aab|Arapesh (Abu)||89A|Order of Numeral and Noun|Noun-Numeral|2
aab|Arapesh (Abu)||92A|Position of Polar Question Particles|Final|2
aab|Arapesh (Abu)||93A|Position of Interrogative Phrases in Content
 Questions|Not initial interrogative phrase|2
aab|Arapesh (Abu)||97A|Relationship between the Order of Object and
Verb and the Order of Adjective and Noun|VO and NAdj|4
aab|Arapesh (Abu)||112A|Negative Morphemes|Negative particle|2
aab|Arapesh (Abu)||116A|Polar Questions|Question particle|1
aab|Arapesh (Abu)||143A|Order of Negative Morpheme and Verb|VNeg|2
```

```
aab|Arapesh (Abu)||143E|Preverbal Negative Morphemes|None|4
aab|Arapesh (Abu)||143F|Postverbal Negative Morphemes|VNeg|1
aab|Arapesh (Abu)||143G|Minor morphological means of signaling
negation|None|4
aab|Arapesh (Abu)||144A|Position of Negative Word With Respect to
Subject, Object, and Verb|SVONeg|4
aab|Arapesh (Abu)||144B|Position of negative words relative to
 beginning and end of clause and with respect to adjacency to
 verb|End, not immed postverbal|6
aab|Arapesh (Abu)||144D|The Position of Negative Morphemes in SVO
 Languages|SVONeg|4
aab|Arapesh (Abu)||144H|NegSVO Order|No NegSVO|4
aab|Arapesh (Abu)||144I|SNegVO Order|No SNegVO|8
```

In this way, it is considerably easy to extract the dataset in any tabular form in which you might need it. The disadvantage is of course that SQL is not that easy for many people to grasp, and I include myself here: I had to fiddle quite a lot with the SQLITE version of the WALS dataset in order to receive the code that I presented here, and I am even not sure if this way of *handling* the query is the preferred one.

If you want to access the data directly in Python, you need to have some idea of how you want to represent the linked data internally. My choice would be to have all data in a single dictionary structure where the WALS language variety code is the key, and the values are the concrete values for all features for the given language.

In order to achieve this, we first need to import cldfbench (which gives us access to the WALS data) and we need to import the collections built-in library of Python, which provides access to ordered dictionary structures.

```
from cldfbench import get_dataset
from collections import OrderedDict
```

Having done this, we can now access the major tables of the CLDF dataset, namely the languages and the parameters (features), and we can create a first dictionary that will store the language varieties with their values.

```
wals = get_dataset("wals").cldf_reader()
languages = OrderedDict(
   {row["ID"]: row for row in wals.iter_rows("LanguageTable")})
parameters = OrderedDict(
   {row["ID"]: row for row in wals.iter_rows("ParameterTable")})
parameter_list = list(parameters)
```

```
codes = OrderedDict(
   {row["ID"]: row for row in wals.iter_rows("CodeTable")})
varieties = OrderedDict(
   {lang["ID"]: ["" for x in parameters] for language in languages.values()})
```

You can see that the Dataset.iter_rows-command provides access to the languages, the parameters, the codes (which give you human-readable names for the numeric values in the value table), and also to the values in the form of a list of ordered dictionaries. In order to assign the values to each language variety, the code has so far created an ordered dictionary (this would also work with a normal dictionary, of course) and added a list of empty values for all parameters. Since we cannot access an ordered dictionary with numerical indices, I also created a list of the parameters, since we will need to infer the index of each parameter when inserting them into our so far empty list.

Having done all this, we can now finally also iterate over the value table. Here, we determine, based on the language identifier, the parameter identifier, and the code identifier, for each value to which variety it should be assigned, which parameter it reflects, and which code it represents and insert the value in our list.

```
for row in wals.iter_rows("ValueTable"):
   pid = parameter_list.index(row["Parameter_ID"])
   varieties[row["Language_ID"]][pid] = codes[row["Code_ID"]]["Name"]
```

If you now query the first 20 non-empty entries in the data for the language variety with code aab, you can do that in order to check and make sure the SQLITE query yields the same output as when accessing the data from Python.

```
count = 0
for i, param in enumerate(parameters):
   if count == 20:
      continue
   if varieties["aab"][i]:
      print(param, varieties["aab"][i])
      count += 1
```

The output of this code is:

```
81A SVO
82A SV
83A VO
```

```
87A Noun-Adjective
88A Noun-Demonstrative
89A Noun-Numeral
92A Final
93A Not initial interrogative phrase
97A VO and NAdj
112A Negative particle
116A Question particle
143A VNeg
143E None
143F VNeg
143G None
144A SVONeg
144B End, not immed postverbal
144D SVONeg
144H No NegSVO
144I No SNegVO
```

Writing the Data to File

In order to write the data to file, all you would have to do when using the SQLITE query and the SQLITE3 software is to switch the output to TSV files and to run the command from above:

```
sqlite> .output wals_by_language.tsv
...
```

But note that the data in this form will not list all values per variety but rather represent a long table format in which a value for a given feature in a given language variety is listed in one row. Assuming that this is not necessarily what you want, it is easier in this case to use Python for writing the data to file, since here we have already arranged the data in such a way that a given language variety has been assigned all its individual values.

In order to write the data in Python to file, you only need to iterate over the dictionary with all language varieties. If you want to access more language-specific information, you need to access the language data in the language dictionary. The following lines of code do all of this and specifically add geocoordinates, Glottocodes, and language family information for each variety in the WALS dataset.

```
with open("wals_by_language.tsv", "w", encoding="utf-8") as f:
    f.write("\t".join([
        "ID",
        "Name",
        "Glottocode",
```

```
    "Family",
    "Latitude",
    "Longitude",
    ])+"\t"+"\t".join(
      [row["Name"] for row in parameters.values()]
      )+"\n")
  for variety, values in varieties.items():
    f.write("\t".join([
      variety,
      languages[variety]["Name"] or "",
      languages[variety]["Glottocode"] or "",
      languages[variety]["Family"] or "",
      str(languages[variety]["Latitude"] or ""),
      str(languages[variety]["Longitude"] or "")
      ])+"\t"+"\t".join(values)+"\n")
```

Supplement for Code Example

The code that I discuss in this tutorial can also be directly accessed in the form of a GitHub Gist, which you find at the following link: https://gist.github.com/LinguList/a3e38e0ef9e2fdd781ec8c08f7710ed0

References

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.

Forkel, Robert and List, Johann-Mattis (2020): CLDFBench. Give your Cross-Linguistic data a lift. In: Proceedings of the Twelfth International Conference on Language Resources and Evaluation. 6997-7004.

Forkel, Robert and Rzymski, Christoph and Bank, Sebastian (2021): PyCLDF (Version 1.18.0). Jena:Max Planck Institute for the Science of Human History.

Matthew Dryer and Martin Haspelmath (2013): The World Atlas of Language Structures online. Leizpig:Max Planck Institute for Evolutionary Anthropology.

Susanne Maria Michaelis and Philippe Maurer and Martin Haspelmath and Magnus Huber (2013): APiCS Online. Jena:Max Planck Institute for the Science of Human History.

Steven Moran and Daniel McCloy (2019): PHOIBLE 2.0. Jena: Max-Planck Institute for the Science of Human History.