Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

# How to handle semantic data with tables (How to do X in linguistics 3)

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

Semantic data are notoriously difficult to handle. In contrast to the form-part of the linguistic sign, meanings are not organized sequentially, but rather network-like (List 2014: 34f). As a result, we often encounter problems when trying to model complex relations between different meanings, specifically in those cases, where we have only tables as our base material. This blog post tries to summarize how major types of semantic data are handled in the Concepticon project and how they can be accessed in code.

The Concepticon project started out as a collection of concept lists that were linked with each other in order to assess to which degree these concept lists were employing the same concepts (List et al. 2016). Concepts are in this context roughly thought of as certain senses that can be elicited with the help of an elicitation gloss in field work (by asking a speaker "how do you say X in your language"). The Concepticon links concept lists, that is, lists of elicitation glosses in different elicitation languages, by defining Concept Sets, that is, concepts that recur across concept lists. All in all, this is a very straightforward and very pragmatic enterprise, which originally had the only goal of allowing us to systematically trace which datasets would provide access to which concepts.

As of now, the data in Concepticon has been constantly growing, and there is no end in sight. We have a much larger team of annotators now who are trained in checking our links rigorously and we have additional software that helps us to test if our annotations formally correspond to our expectations. By now, we have also added many concept lists that offer additional information which we were not planning to collect initially. Among these concept lists, the most straightforward ones we can distinguish are ranked lists, annotated lists, and lists with relations. You can find examples for these lists by turning to the Concepticon web application and looking for the tags in the Concept Lists tab.

Ranked lists are defined as those lists which do not only provide the elicitation glosses in alphabetical or any other order, but in a specific order that is supposed to reflect their

*Computer-Assisted Language Comparison in Practice*
Volume 4, Number 1
https://calc.hypotheses.org/2617/

Published on 13/01/2021

rank. Rank can refer to different aspects of conceptual properties or the properties of words that express these concepts, such as their stability or their frequency.

Annotated lists contain additional columns with additional information which can vary to quite some degrees, ranging from the phonetic representation of the words expressing a given concept elicitation gloss up to an indication of the authors from which other concept lists their concepts were derived.

Lists with relations are the most interesting type of concept lists we have encountered by now. This type refers to those concept lists where authors provide additional information on the relations between the concepts in a given list. These relations can appear in the form of categories, properties, or direct links between the concepts in a given list.

Given that we use an exclusively tabular model in Concepticon, where the primary data point is one line in a table representing one concept elicitation gloss, it may seem at times difficult to model complex concept relations with this schema. For this reason, it seems time to provide some examples on how to proceed when trying to code up concept data for the Concepticon.

## 1 Handling Ranked Lists

Ranks are rather straightforward to handle. They can be given directly (by providing a Rank field, which displays numbers) or indirectly, by providing a score field which can be used to rank the list. In order to preserve the unity of the ranked lists and to make them more readily accessible also programmatically, we have adopted the practice of adding always an explicit field with numerical ranks, even if this is not given in the original concept list. Since adding ranks is very trivial (in the VIM editor, I would type `%s/^/\=line('.')-1/` in order to add incrementing numbers starting from 0 to each line of my list), I won't discuss them further here. An example for a ranked list is the list of 100 items by Pozniakov (2014), where items are ranked by stability.

As a concrete coding example showing how a ranked list can be handled, consider the following lines of code, which allow you to use the Concepticon API (having set the code up as illustrated in Tjuka 2019) and display the list of ranks.

```
from pyconcepticon import Concepticon
from tabulate import tabulate

def display_rank(conceptlist, concepticon):
    table = []
    for concept in concepticon.conceptlists[conceptlist].concepts.values():
        table += [[concept.attributes['rank'], concept.english or
```

```
            concept.gloss]]
    print(tabulate(
        sorted(table, key=lambda x: x[0]),
        headers=['Concept', 'Rank']))

concepticon = Concepticon()
conceptlist = 'Pozdniakov-2014-100a'
display_rank(conceptlist, concepticon)
```

If you run this code as a script, it will display the concept list in ranked form:

```
 Concept  Rank
---------  ---------
       1  I
       2  you sg.
       3  two
       4  eye
       5  we
       6  tongue
       7  name
       8  one++
       9  dog+
      10  nail+
```

## 2 Annotated Lists

Annotated lists are mannifold, and it is not easy to make a good decision of what parts of the abundant information that scholars may provide when preparing concept lists should be kept and what parts should be discarded. A frequently recurring case of annotation are Comment or Note fields. Here, we have adopted the practice to make sure that our CSV format, which is based on a tabstop as separator, would not be bothered by newline characters inside a given table cell. As a result, we strip newlines if we encounter them in the comment fields that we adopt when adding annotated concept lists to the Concepticon. An example for an annotated concept list is the one by Kassian et al. (2010), referenced as Kassian-2010-116 in Concepticon.

## 3 Lists with Relations

The most interesting concept lists are those which provide additional relations between the concepts. If we think of relations as of a graph in which different nodes are linked, we can distinguish different types of relations by considering different types of graphs. A very simple graph is an unweighted, undirected network in which nodes are linked by

undirected links and the links are not further annotated. This graph can be easily represented in a table by providing one column for the concept and one for the links, which should be separated by some separation character (a simple space in our example here, assuming that concept identifiers do not have spaces).

| ENGLISH | LINKS |
|---|---|
| HAND | ARM |
| ARM | HAND |
| FOOT | LEG BALL |
| BALL | FOOT |
| LEG | FOOT |

If we have a weighted graph, where links are annotated by a specific weight, we can handle the weights (and any other kind of annotation) in a table by providing an extra column, which we could call WEIGHTS and adding the weights in numerical form, again using spaces as separators.

| ENGLISH | LINKS | WEIGHTS |
|---|---|---|
| HAND | ARM | 1.0 |
| ARM | HAND | 1.0 |
| FOOT | LEG BALL | 1.0 0.1 |
| BALL | FOOT | 1.0 |
| LEG | FOOT | 1.0 |

We can easily turn this graph into a directed network by discarding asymmetric links.

| ENGLISH | LINKS | WEIGHTS |
|---|---|---|
| HAND | ARM | 1.0 |
| ARM | | |
| FOOT | BALL | 0.1 |
| BALL | | |
| LEG | FOOT | 1.0 |

What is crucial for the display of a true graph with the help of a table in this example is that our Links field needs to provide the same identifiers as we used to define our concepts (which are given in our case in the field Concept). Furthermore, the list must be exhaustive: an identifier that appears only in LINKS but not in Concept will be wrong, as we have not “defined” this concept. With respect to the separators, one can choose whatever separator one wants to choose, and provide this information with the help of the metadata:

```
{
 "name": "LINKS",
 "separator": " ",
 "datatype": {"base": "string"}
}
```

Another very frequent type of relations comes along in the form of specific “tags” provided for individual concepts (see another blog post from last year by Ilia Chechuro on tagged markup, Chechuro 2019). When using tags, we assign each concept in a given concept list a certain number of characteristics. Tags differ from our direct network annotation, since tags can be arbitrarily chosen, while the LINKS field in our network concept lists needs to contain entries from our ENGLISH field (or alternatively an ID field).

| ENGLISH | TAGS |
|---|---|
| HAND | five body-part up |
| ARM | body-part up |
| FOOT | body-part low |
| BALL | tool |
| LEG | body-part low |

When modeling tags and concepts in a network, the resulting network is a bipartite network in which two distinct types of nodes occur, one drawn from our ENGLISH field, and one drawn from our TAGS field, and links can only be assigned between the different types and are therefore undirected. A bipartite network also corresponds to a hypergraph, that is, a graph in which edges can connect more than one node (Jacques and List 2019: FN 14), but knowing this is not necessarily important when adding tags to a concept list. Weighting tags is of course also possible. One could add a specific WEIGHTS field to the concept list, or one could add the weights directly to the tag, using a secondary separator.

| ENGLISH | TAGS |
|---|---|
| HAND | five:0.1 body-part:2.0 up:1.0 |
| ARM | body-part:0.1 up:1.0 |
| FOOT | body-part:0.2 low:0.1 |
| BALL | tool:3.0 |
| LEG | body-part:1.0 low:0.2 |

In the example above, I have used a colon as a secondary operator. In the same way, we could also have defined our weighted graph above, and there are arguments in favor and against such a notation. The advantage is that all data are in one place, so it is clear

that when looking at the data alone, that the weights belong to the tags, while our solution by adding a WEIGHTS column does not provide the semantics that would help us to associate WEIGHTS with LINKS. The disadvantage is of course the readability, but also the danger of yielding very complex cell-internal structures that are difficult to test and resolve.

When adding cell-internal separators, it is therefore always important to make this clear with the metadata. While we can use regular expressions to specify the structure of the metadata, we have still not completely solved the problem of not only handling these complex cases consistently, but also making sure that they are rigorously tested.

As an example how you can actively load and analyze a concept list that was provided in the form of a network (with links and weights), let us look at the following example which loads the concept list reflecting the Database of Semantic Shifts (Zalizniak et al. 2020), which was recently added to Concepticon and entails a rather complex network structure.

We first import our basic libraries to work with the Concepticon and then also import the networkx package (Hagberg et al. 2020).

```
from pyconcepticon import Concepticon
from tabulate import tabulate
import networkx as nx
```

After having done this, we define a function to load the network. This function is intended to work for generic networks that are consistently encoded.

```
def get_network(
      conceptlist,
      concepticon=None,
      nodes='number',
      edges='links',
      node_attributes=None,
      edge_attributes=None,
      ):
   def ca(concept, attribute):
      """Shortcut for retrieving a concept attribute."""
      return getattr(
            concept,
            attribute,
```

```
            concept.attributes.get(attribute))

    concepticon = concepticon or Concepticon()
    G = nx.Graph()
    for concept in concepticon.conceptlists[conceptlist].concepts.values():
        nodeid = ca(concept, nodes)
        links = ca(concept, edges)
        nodeattrs = {aname: ca(concept, a) for a, aname in
                node_attributes.items()}

        if nodeid not in G:
            G.add_node(nodeid, **nodeattrs)
        else:
            for k, v in nodeattrs.items():
                G.nodes[nodeid][k] = v

        for i, link in enumerate(links):
            edgeattrs = {aname: ca(concept, a)[i] for a, aname in
                    edge_attributes.items()}
            G.add_edge(nodeid, link, **edgeattrs)
    return G
```

Important aspects of this function are the keywords for the nodes (defaulting to the "number" field), the edges (defaulting to "links") as well as the node and edge attributes, which are passed in the form of dictionaries with a key pointing to the name of the value as it can be found in the concept list and the value defaulting to the intended name it should have when creating the network.

We load the network as follows.

```
G = get_network(
        'Zalizniak-2020-2590',
        node_attributes={'english': 'gloss'},
        edge_attributes={
            'weights': 'weight',
            'urls': 'url',
            'directions': 'direction'}
        )
```

Once we have done this, we can filter the edges, by deleting those which have not enough examples (as encoded by the "weights" attributes in our concept list).

```
del_edges = []
for nA, nB, data in G.edges(data=True):
   if data['weight'] < 10:
      del_edges += [(nA, nB)]

G.remove_edges_from(del_edges)
```

Once this has been done, we can make a small analysis of the data, by printing those connected components in the network which have more than 10 nodes.

```
for i, nodes in enumerate(nx.connected_components(G)):
   if len(nodes) > 10:
      print('# Component {0}'.format(i+1))
      table = [[node, G.nodes[node]['gloss'], len(G[node])] for node in nodes]
      print(tabulate(
         sorted(table, key=lambda x: x[2], reverse=True),
         headers=['ID', 'Concept', 'Degree']
         ))
      print('')
```

Here, I deliberately chose the weights in such a way that we can inspect the results since the network is very large). The first connected component is shown below. In general, the components are a bit surprising, but since the data are at times very sparse, one should not wonder too much about this.

```
# Component 1
  ID  Concept                                  Degree
----  -------------------------------------  --------
   1  to beat, hit                                  5
 101  fire                                          3
 790  to be at war                                  2
 130  to shoot                                      2
 360  to swear, curse, abuse one another            2
 132  to kill                                       2
  53  tree                                          1
 278  to finish                                     1
 212  to bark (of a dog)                            1
 697  to win                                        1
1171  to play (a musical instrument)                1
2353  inflammation
```



# Outlook

Handling complex concept relations in our Concepticon reference catalog is a relatively new development. While we are confident that we can use our existing solutions for handling the data, there is still a lot to do. In the future, we should add explicit tests for those additional aspects of our data we consider important to share in a comparable and standardized way.

# References

Ilia Chechuro, “Why Tag Markup may be Useful for Lexical Data,” in *Computer-Assisted Language Comparison in Practice*, 03/06/2020, https://calc.hypotheses.org/2476.

Jacques, Guillaume and List, Johann-Mattis (2019): Save the trees: Why we need tree models in linguistic reconstruction (and when we should apply them). *Journal of Historical Linguistics* 9.1. 128-166.

Hagberg, Aric and Dan Schult and Pieter Swart (2020): NetworkX. Version 2.5. https://networkx.org

Kassian, Alexei and Starostin, George S. and Dybo, A. and Chernov, Vasiliy (2010): The Swadesh wordlist. An attempt at semantic specification. *Journal of Language Relationships* 4. 46-89.

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

List, Johann-Mattis and Cysouw, Michael and Forkel, Robert (2016): Concepticon. A resource for the linking of concept lists. In: Proceedings of the Tenth International Conference on Language Resources and Evaluation. 2393-2400.

Pozdniakov, Konstantin. 2014. O poroge rodstva i indekse stabil’nosti v bazisnoj leksike pri massovom sravnenii: Atlantičeskie jazyki[On the threshold of relationship and the “stability index” of basic lexicon in mass comparison: Atlantic languages]. Journal of Language Relationship 11. 187-237.

Annika Tjuka, “Adding concept lists to Concepticon: A guide for beginners,” in *Computer-Assisted Language Comparison in Practice*, 29/01/2020, https://calc.hypotheses.org/2225.

Zalizniak, Anna A. and Smirnitskaya, Anna and Russo, Maksim and Mikhailova, Tatiana and Bobrik, Marina and Gruntov, Ilya and Orlova, Maria and Bibaeva, Maria and Voronov, Mikhail (2020: Database of Semantic Shifts (Version from 07/10/2020). URL: http://datasemshift.ru, Moscow: Institute of Linguistics of the Russian Academy of Sciences.