Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

Converting an Artificial Proto-Language into Data for Testing Computational Approaches in Historical Linguistics

Johann-Mattis List
Chair for Multilingual Computational Linguistics
University of Passau

This small study shows how data for an artificially created language that was supposed to reflect features of "proto-languages", predating modern languages by several thousand years, can be used in testing computational approaches in historical linguistics. In order to do so, computational workflow is described that retrieves the data automatically, creating a comparative wordlist compatible in format with software tools for historical linguistics, and then uses a baseline method for automatic cognate detection to compare an artificial language against a sample of Indo-European languages. The results show that artificial languages might help to fill a gap in testing that has so far been ignored in the literature.

1 Introduction

I was a bit surprised when I stumbled over the article "Reconstructing a Protolanguage" by Luuk and Stavroulakis (2024), which was published as part of the EvoLang conference 2024. At first, I thought the authors had tried to embark on "classical" speculative work by discussing how the language of the first representatives of humans would have sounded. But after reading their study more closely, it became clear to me that they do not try to reconstruct the proto-language of proto homo sapiens, but rather a possible proto-language that would reflect a stage of complexity that preceded the structure of modern languages.

What they do in the end is presenting something close to a grammatical sketch of some artificial language that was designed in such a way that it incorporates many of those features that have been claimed to be old in the literature. This grammatical sketch is in some sense similar to the Fabel in indogermanischer Sprache by Schleicher (1868). The difference is that the authors do not only provide the text of the fable, but include a

Computer-Assisted Language Comparison in Practice
Volume 7, Number 2
URL: https://calc.hypotheses.org/7363 DOI: 10.15475/calcip.2024.2.1

Published on 17/07/2024

glossary of words with English translations, and that they provide their text in interlinear-glossed form (see Lehmann 2004 on interlinear-glossed text).

While I wanted to file the study off at first, I realized, when looking at the data that the authors had compiled, that the wordlist — as speculative and strange as it might appear to "normal" historical linguistics — might turn out to be useful to test how well certain automated and computer-assisted approaches in historical language comparison work in the end. Since the authors designed their vocabulary on the model of many different existing languages, taking information on sound symbolism from the ASJP database (https://asjp.clld.org, Wichmann et al. 2016), and information on polysemy from CLICS (https://clics.clld.org, Rzymski et al. 2020), their data, once converted to a classical comparative wordlist, could be useful to test to which degree lookalikes confuse computational approaches in historical linguistics.

As a result of these considerations, I decided to conduct a small test, converting the data by Luuk and Stavroulakis into the wordlist format (see List et al. 2018) used in software packages like LingPy (https://pypi.org/project/lingpy, List and Forkel 2024) and EDICTOR (https://edictor.org, List 2023), and combining the data with a larger dataset on Indo-European languages to test where the "proto-language" would end up in the phylogenetic tree or network when running a baseline method for automatic cognate detection (such as the SCA method for cognate detection discussed in List et al. 2017 and implemented in LingPy).

In the following, I will describe how I conducted this analysis with the help of a short Python script with minimal dependencies. This script first downloads the data in PDF form, extracts the text data from the PDF, converts the data into a wordlist in tabular form, segments and standardizes transcriptions, links glosses to Concepticon (https://concepticon.clld.org, List et al. 2024) in order to extract a classical Swadesh list of 100 items from the data (Swadesh 1955), then searches automatically for cognates in a combined dataset of 19 Indo-European languages, and displays the results in the form of a phylogenetic tree reconstructed from the inferred pairwise language distances.

2 Requirements

In order to use the script described here, certain requirements need to be fullfilled. One must make sure to have installed LingPy (List and Forkel 2024, https://pypi.org/project/lingpy, Version 2.6.13) — for general handling of sound sequences and cognate detection, PyPDF (Fenniak 2024, https://pypi.org/project/pypdf, Version 4.2.0) — for the reading and writing of PDF files, and PySem (List 2024, https://pypi.org/project/pysem, Version 0.8.0) — for the automated mapping of glosses to Concepticon. This can be done by pasting the following line into the command line.

```
pip install lingpy==2.6.13 pypdf==4.2.0 pysem==0.8.0
```

Additionally, one must make sure to have downloaded the dataset of Indo-European languages by Starostin (2005), which I originally converted to formats compatible with LingPy in List (2014), and which has now been published as part of the Lexibank repository for multilingual wordlists that adhere to CLDF standards (https://lexibank.clld.org, see List et al. 2022 regarding Lexibank, and Forkel et al. 2018 regarding CLDF). The easiest way to obtain this dataset is to download it with the help of GIT. Pasting the following lines into the command line should do the trick (provided GIT is installed on your system).

```
git clone https://github.com/sequencecomparison/starostinpie
cd starostinpie
git checkout v1.0
cd ..
```

Alternatively, you can also download the data from GitHub (https://github.com/SequenceComparison/starostinpie/tree/v1.0) and unpack it in the folder from which you want to run the script shared in this little study.

3 Workflow

3.1 Importing Data

We start by importing the data we need for the comparison. We use packages from the Python standard library to download the data (in the form of a PDF file) from the repository where it has been shared by the authors (urllib), to store the downloaded data in a temporary directory (tempfile), and to access paths across different platforms (pathlib). We also import the three above-mentioned external libraries (pypdf, lingpy, pysem).

```
from urllib.request
import urlopen
import tempfile
import pathlib

from pypdf import PdfReader
from lingpy import ipa2tokens, Wordlist, LexStat
from pysem import to_concepticon
```

3.2 Download and Covert PDF to Text

In order to conver the PDF file to text, we first download the data and write the PDF data to a file in a temporary directory. Once this has been done, we read the file with the PyPDF PDF reader and iterate over all PDF pages, each time extracting all text that can be found there in plain text form. The information is stored in the dictionary pages.

```
url = "https://gitlab.com/protolanguage1/protolanguage-supplement-lexicon/-
/raw/main/SUPPLEMENTARY_MATERIALS.pdf?inline=false"
pages = {}
with tempfile.TemporaryDirectory() as t:
    with urlopen(url) as req:
        data = req.read()
    path = pathlib.Path(t) / "data.pdf"
    with open(path, "wb") as f:
        f.write(data)
    pdf = PdfReader(path)
    for i, page in enumerate(pdf.pages):
        pages[i] = page.extract_text()
```

3.3 Convert Text to Wordlist

We must now convert the text data to a wordlist (compatible with LingPy and EDICTOR). This requires us to loop over the first three pages where the wordlist can be found in the PDF supplementing Luuk and Stavroulakis. According to the internal logic of the original data, lines with actual glosses and word forms in the data must contain the equal sign (=), which is used to separate a word in the artificial proto-language from a number of glosses separated by a comma.

```
data = []
for i in range(3):
    rows = [row for row in pages[i].split("\n") if
            "=" in row]
    for row in rows:
        word, concepts = row.strip().split("=")
```

Since the glosses may well differ in their meaning and cannot be simply interpreted as extended dictionary definitions for the same sense, our strategy consists in iterating over all glosses separately and trying to map them automatically to Concepticon with the help of the to_concepticon function offered by PySem.

```
    for concept in concepts.strip().split(", "):
        mappings = to_concepticon(
                [{"gloss": concept}]
                )[concept]
        if mappings:
```

We only retain those glosses for which a mapping could be identified. In order to standardize the phonetic transcription, we correct an error in the phonetic transcription used by the authors (they use the character ɣ to refer to the character ɣ), and we use LingPy's ipa2tokens method to automatically segment the word forms into a tokenized representation in which individual sounds are identified and separated by a space (see List et al. 2018 for details). We retain the original word form as the original value in the data, add the form, where the transcription is modified by the simple replacement, and then add a segmentized representation of the token on top of all that. While we could of course discard the value and form representations of the phonetic transcriptions, we keep them for reasons of transparency and since it is easier to debug problems at later times.

```
            value = word.strip()
            form = value.replace("ɣ", "ɣ")
            tokens = ipa2tokens(form)

            data += [[
                "Proto",
                concept,
                mappings[0][0],
                mappings[0][1],
                value,
                form,
                tokens
                ]]
```

3.4 Combine with the Indo-European Data

In order to combine the monolingual wordlist for the artificial proto-language with the Indo-European data by Starostin (2005), we must match Concepticon glosses across both datasets. When reading the Indo-European data with the help of LingPy, the information on the Concepticon mapping is stored in the column CONCEPTICON (using the Concepticon Concept Set ID, an integer, as base value to represent the Concept Set). In our proto-language wordlist, we have stored both the Concepticon ID and the Concepticon Gloss. Since dealing with Concepticon Glosses is often easier, specifically when trying to inspect a dataset, we want to retrieve the Concepticon Gloss from the

Concepticon ID in the whole dataset we want to create. This can be easily done by creating lookup tables in the form of Python dictionaries, as shown below.

```
pie = Wordlist.from_cldf("starostinpie/cldf/cldf-metadata.json")
id2gl = {row[2]: row[3] for row in data}
overlap = set([pie[idx, "concepticon"] for idx in pie])
```

We can now create a wordlist dictionary that consists of keys with integers larger than 0, with the key for 0 providing the column header of the resulting wordlist.

```
wln = {0: [
    "doculect",
    "concept",
    "value",
    "form",
    "tokens",
    ]}
```

We now add the proto-language data row by row, increasing the ID of the row each time by one. We only include those concepts whose Concepticon ID also occurs in the Indo-European data, and we make sure this is the case with the help of the set overlap that contains all Concepticon IDs in the Indo-European data.

```
count = 1
for row in data:
    if row[2] in overlap:
        wln[count] = [
                "Proto",
                row[3],
                row[4],
                row[5],
                row[6]]
        count += 1
```

Having added the proto-language data to the dictionary, we can now add the Indo-European data. Here, we convert the Concepticon ID to the Concepticon Gloss using the dictionary lookup. We also make sure to include only entries with a concept counterpart in the proto-language data.

```
for idx in pie:
    if pie[idx, "concepticon"] in id2gl:
        concept = id2gl[pie[idx, "concepticon"]]
        wln[count] = [
                pie[idx, "doculect"],
                concept,
                pie[idx, "value"],
                pie[idx, "form"],
                pie[idx, "tokens"]
                ]
        count += 1
```

3.5 Searching for Cognates

We can now enter the final stage of the workflow by searching for cognates, calculating a phylogenetic tree, based on the pairwise distances derived from shared cognate sets, which we analyze with the help of the UPGMA algorithm (Sokal and Michener).

We start by initializing a LexStat object and computing cognates automaticlaly, using the SCA method (List et al. 2017). This method is not very deep and does not take regular sound correspondences into account, but it yields reasonable results most times and provides a good starting point for in-depth analysis with human correction. When running the method, we use base parameters.

```
lex = LexStat(wln)
lex.cluster(
    method="sca",
    ref="cogid",
    threshold=0.45,
    cluster_method="upgma"
    )
```

Tree calculation in LingPy is rather straightforward, as the following line shows. Trees can also easily be printed to the terminal, which is very handy when inspecting a dataset initially.

```
lex.calculate(
    "tree",
    ref="cogid",
    tree_calc="upgma"
    )
print(lex.tree.asciiArt())
```

Last but not least, we write the wordlist to file. This will help us to load the wordlist with the automatically inferred cognates into the EDICTOR tool, which allows us to inspect the data conveniently and in due detail. Note that in this command in LingPy, we use the keyword subset set to True in order to indicate that we do not want all data to be exported to the wordlist file. Instead, we only select the columns specified in the keyword cols.

```
lex.output(
    "tsv",
    filename="wordlist",
    ignore="all",
    prettify=False,
    subset=True,
    cols=[
        "doculect",
        "concept",
        "value",
        "form",
        "tokens",
        "cogid"]
)
```

3.7 Running the Workflow Script

As can be seen from the description above, the workflow described here consists of a single Python script. This script has also been shared in the form of a GitHub GIST (t https://gist.github.com/LinguList/f12bfd9acff2bec91525e1e6511e5adb.). To run the workflow, you can download the script (make sure to have installed dependencies and downloaded the Indo-European data by Starostin, as mentioned in the previous section), and run the script via the terminal.

```
python proto.py
```

4 Results

The UPGMA tree created by the combined analysis of the artificial “proto-language” and the Indo-European languages in the sample is show in Figure 1. It confirms our minimal expectation that the cognate detection method used here should separate the artificial language from the rest of the languages.

When inspecting the results a bit more closely, however, for example by loading the wordlist file into EDICTOR (via https://edictor.org), one can easily see that the method identifies a considerable amount of matches between the artificial language that was

created in such a way that it would account for some general features that can be observed for many languages in the world and our Indo-European sample. Specifically the pronouns ta “this” and tu “that” have many hits among Indo-European languages. With EDICTOR’s panel that allows to analyze and compare cognate sets in a dataset (ANALYZE → Cognate Sets) we can easily see that there are 38 words in the artificial language for which the method identifies at least one word from the Indo-European languages to be cognate.

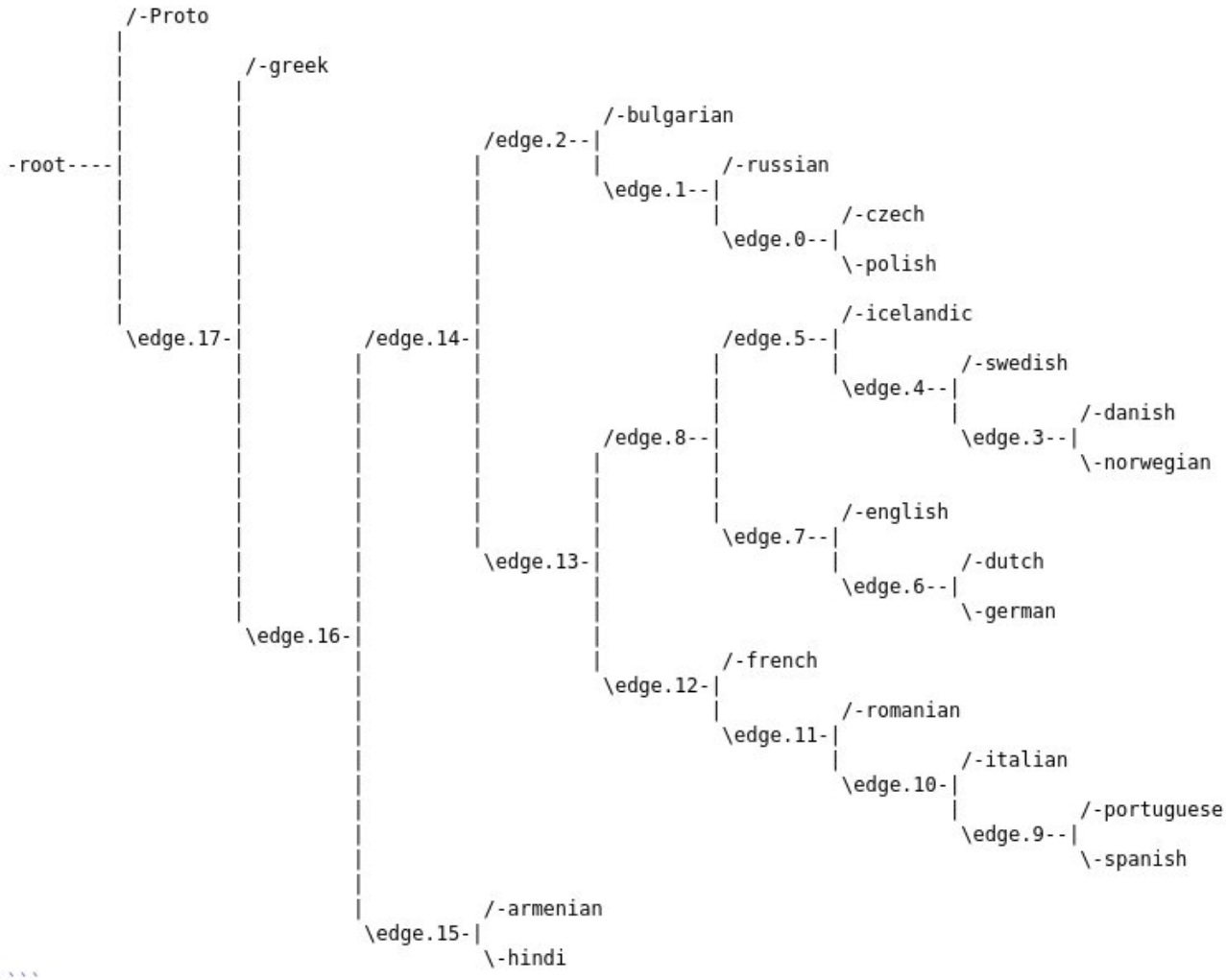

Figure 1: UPGMA tree (printed to the terminal) of the data.

EDICTOR allows to download the data in NEXUS format (Maddison et al. 1997) that can be directly used to inspect the data with the SplitsTree software package (Huson 1998), which allows us to visualize the data in the form of a splits network with the help of the NeighorNet algorithm (Bryant and Moulton 2004). The NEXUS file created from EDICTOR is also shared as part of the supplementary material. The resulting NeighborNet is shown in Figure 2. As can be seen from this figure, the artificial "proto-language" is no longer as much of an outsider in our approach but rather appears close to

those languages that are more distantly related to the rest of the Indo-European languages in the sample (notably Greek).

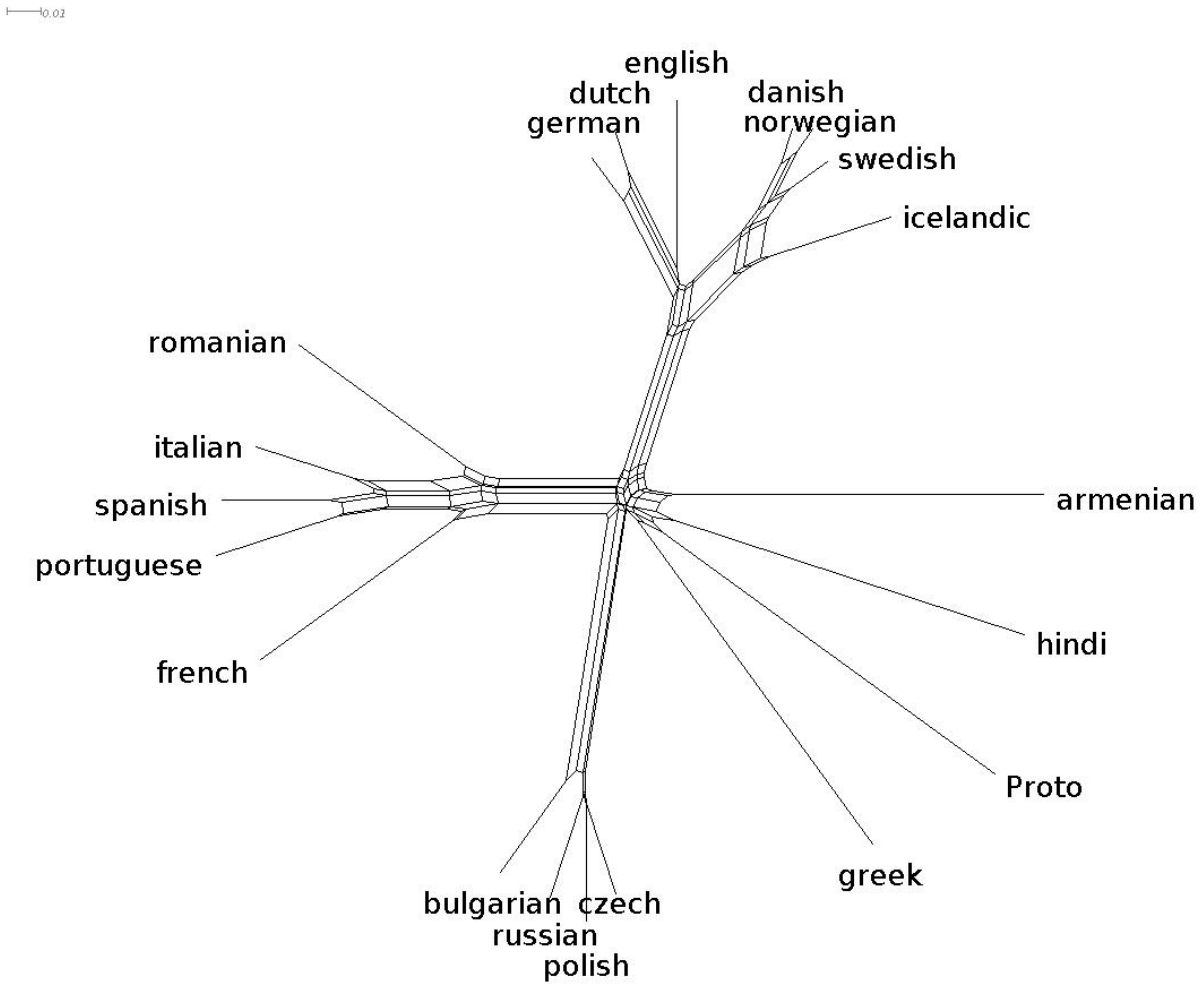

Figure 2: NeighborNet of the data.

This shows that the large number of hits that the SCA method finds for the artificial language against the Indo-European languages cannot be singled out by finding more signal among the Indo-European languages themselves. When calculating an unrooted Neighbor-joining tree (Saitou and Nei 1987) instead of the UPGMA tree we saw before (which can be done directly in SplitsTree), we can see this confirmed in the fact that the artificial language now clusters with Greek in our sample as closest neighbor, as shown in Figure 3.

5 Outlook

When concentrating on computer-assisted approaches in historical language comparison, we often work exclusively with genetically related languages, trying to confirm that the

methods find those cognates we have already identified using classical techniques applied by experts in the respective language families. Tests on unrelated languages are less often carried out, although they would be very useful in order to find out what the limits of exclusively automatic approaches are. With the study by Luuk and Stavroulakis, I realized that there may be an additional kind of test data that might deserve more attention. This data would consist in artificial languages that have been designed in such a way that they look like average languages, although they have not been actively derived from any particular spoken language.

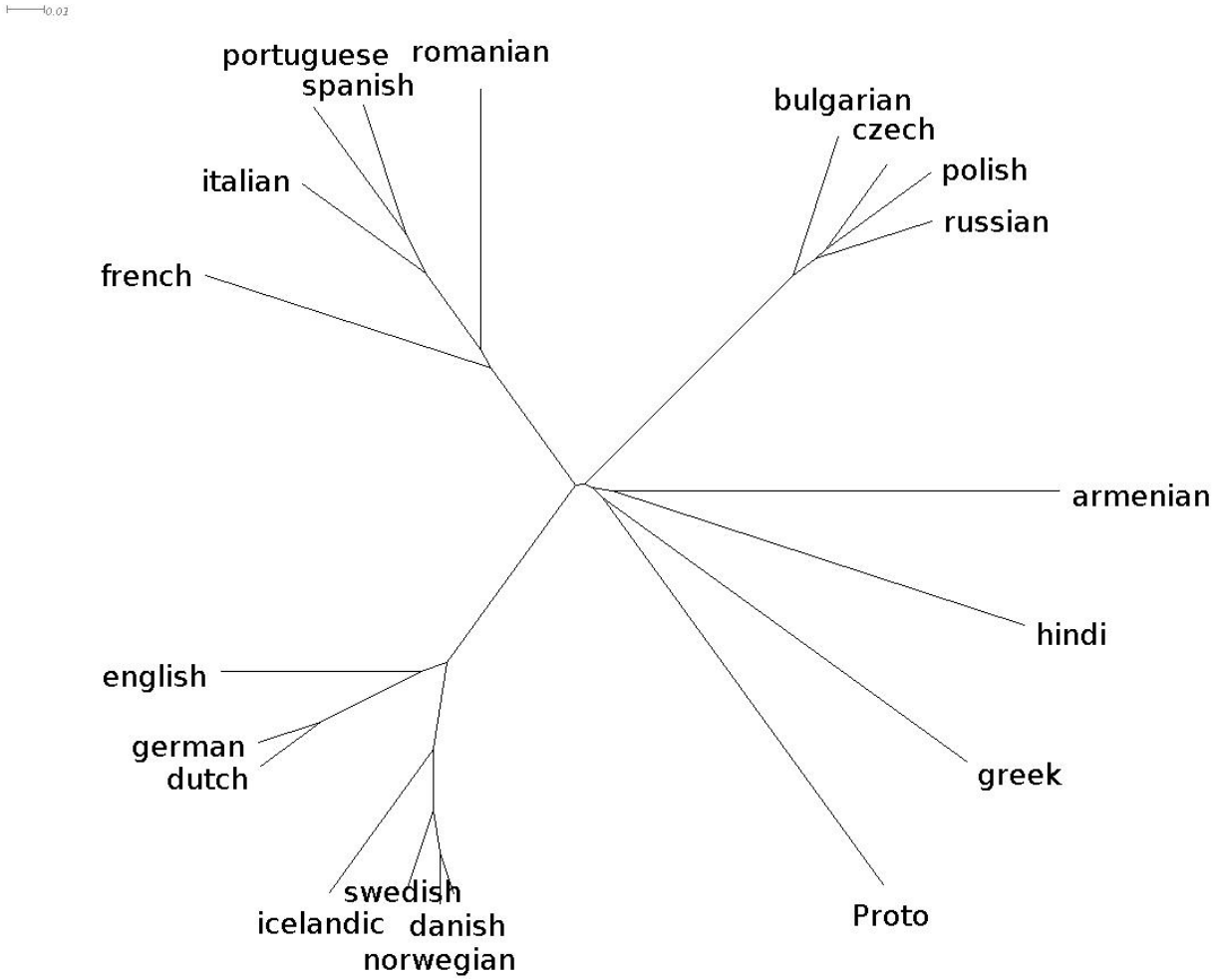

Figure 3: Neighbor-joining tree of the data.

As strange as the idea to construct a "proto-language" may thus seem at first glance, the implementation of the authors pointed me to a gap in our current evaluation of computational methods in historical linguistics. We tend to compare what we think should be related and what we think should not be related. However, we rarely conduct stress tests for our methods, by having them deal with extreme cases, such as a language

that ticks all boxes of sound symbolism that one can think of, or a language that goes into extremes when it comes to polysemy.

For future work in computational historical linguistics, I think, it would be very useful to develop new approaches that help us to create several artificial languages that we could use to improve our insights into our automatic approaches. One could investigate whether language models could be constructed to generate languages that are unrelated but similar to the languages in the world, or one could manually create a set of prototypical constructed languages for historical language comparison. In any case, I would hope that such an approach would not only help us to improve our current methods, but also allow us to get deeper insights into the potential pitfalls resulting from convergent evolution and chance similarities in historical language comparison.

References

Supplementary Material
The code presented in this study has been shared in the form of a GitHub GIST that can be accessed at https://gist.github.com/LinguList/f12bfd9acff2bec91525e1e6511e5adb. The code example was tested on a Linux machine with Python 3.12.
Funding Information
This project has received funding from the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme (Grant agreement No. 101044282). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.