Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

Implementing Fuzzy Spelling Search in Dictionaries of Under-Described Languages Lacking Standard Orthographies

Kellen Parker van Dam
Chair of Multilingual Computational Linguistics
University of Passau

Non-standard orthographies are common in the world of under-described language documentation. Whether they are semi-conventionalised community spellings, orthographies partially adopted from missionary works, or hastily transcribed texts representing as-yet uncertain phonologies, there is a need to be able to work through lexical data in a way which can accommodate and respond to such non-standard transcriptions. Here, a few options are considered, with a solution for fuzzy string matching based on attested variations is presented.

1 Introduction

Developing word lists or dictionaries is often an important part of the documentation of otherwise under-described language varieties. In many cases, it may be the first time an effort to work out the phonemic inventory of the language has been undertaken as a serious scientific endeavour, although this is not always the case. In other instances, community-based efforts may have taken place regarding the spelling of the language, or earlier missionary scripts may be in widespread use. Often, however, the Roman orthographic conventions are largely not standardised, and a considerable amount of inter- and intra-speaker variation occurs. Ongoing efforts at mother-tongue education will likely result in this changing further, as a language community develops resources and increases the coverage of first-language education. This is a slow process, and often lags behind the early stages of dictionary development.

Computer-Assisted Language Comparison in Practice
Volume 7, Number 1
URL: https://calc.hypotheses.org/7160 DOI: 10.15475/calcip.2024.1.5

Published on 27/05/2024

2 The Problem

As a result of non-standard spelling conventions, a number of word lists and other forms of lexical data exist for which spelling variation may make it difficult to filter through what may be considerable amounts of data. Even on languages where work has been taking place for a decade or more, sound changes may be ongoing, or community-based decisions on how sounds should be transcribed may be in flux. This can result in a need to accommodate variant spellings, even in otherwise standardised systems. Or, as a larger amount of data becomes available, there may be a need to amend previously understood phonemic inventory descriptions and their corresponding spelling conventions. All of these scenarios are currently underway in different parts of mainland Southeast Asia, where communities are currently striving to codify their languages, sometimes with as little as a few hundred active speakers. In such cases, it is incredibly unlikely that even a trained phonologist would be able to work out the phonemic inventory of a new language in only a short time working with a language, while the urgency of data collection and dictionary development may not afford that kind of time.

As part of an ongoing dictionary project, the need has arisen for the ability to search dictionary entries by a range of non-standard spellings for Wolam Ngio (wola1254), a Tibeto-Burman language with around 6000 speakers living across the India-Myanmar border. It is an under-documented language, on which only a few resources exist, but which is widely used among community members on social media platforms. As a result, alternate spellings abound.

The primary goal is the creation of a search function for a dictionary of Wolam Ngio which can accommodate variant spellings. To this end, a number of different approaches are possible in this case.

3 Potential Solutions

3.1 Solution 1: Encode All Potential Variant Spellings in the Database

One option for accommodating variant spellings is to simply hard-code all attested variations for a given lexeme or morpheme into the data set. This would be the programmatically least intensive solution, although at the present stage that is not a major concern. However, having the database include all possible spelling variations would allow a plaintext search with little need to script the process. This solution carries with it some obvious drawbacks.

First, it would result in a larger database file. This again is not a big issue in itself, but eventually as the number of entries grows, one would need to consider practical limitations in a part of the world with unreliable and often poor network connectivity. The bigger issue, however, is maintainability. In order to encourage community

involvement — a major focus of such collaborative dictionary work — the various workflows involved in maintaining and adding to the dictionary must be as straightforward and intuitive as possible. By including all variant spellings for each lexeme, this puts the onus of managing a ballooning number of variants on the maintainers. Realistically, anything that complicates use for community members is a fatal violation, to borrow the language of Optimality Theory.

A possible workaround would be to automate spelling variations while still hard-coding them in the dictionary. Such a function is not difficult to implement, and in fact is given further below. The output would look something like the one shown in Table 1.

Table 1: Input and output of the function for spelling variant creation.

input		**outputs**	
‘nyoum’	→	[‘ngiom’,	‘ngiom’
		‘ngium’,	‘niom’
		‘nioum’,	‘nium’
		‘nyiom’,	‘nyioum’
		‘nyium’,	‘nyom’
		‘nyoum',	‘nyum’]

This conversion would be trivial to run at the time the database is first created from the tabular data. This still complicates the matter of maintainability, requiring an additional step and thus still adding considerable data to the database. As one example, based on attested spellings within Wolam Ngio, the lexeme *tah.louh* BEGIN would require 48 alternate forms. A single morpheme nyoum would need 12. The more morphemes involved, the more variations are needed, and exponentially. Additionally, certain grapheme clusters such as <ngi> already require a large number of additional forms due to the effects of palatalisaiton, nasal assimilation, or a number of other potential complicating factors.

In lieu of any other alternative this might be a reasonable solution, but better options are there.

3.2 Solution 2: Implement a Fuzzy Search Directly in the Database Query

Instead, a fuzzy search could be implemented directly in the query itself. The main benefit of running the fuzzy search directly in the query is speed. With SQL doing most of the heavy lifting we skip the slow process of generating complex queries in JavaScript to send to the database. Additionally, this offers some degree of portability. Specifically, by having the database handle the fuzziness, the dictionary or word list can be more easily ported to other platforms or languages, whether that be Python, Flutter, PHP or something else. The more that can be pushed onto the database itself rather than the user interface or server-side application, the better. We could implement a search query that

uses regular expressions, saving us the trouble of choosing the alternate forms in the code itself.

However, while certain SQL database systems allow regex searches — PostgreSQL being one of the more popular ones with this feature — SQLite does not, and we use SQLite in our case since the entire database is contained within a single file without need for running any SQL server software, again offering a certain kind of portability.

Unfortunately, due to this choice of using SQLite, our options are more limited. The best bet would likely be to search within a list. For example, searching for <nyiu> MOTHER, phonemically /ɲiu[41]/, would need to include *ngio, ngio', ngiu, ngiu', nio, nio', niu, niu', nyio, nyio', nyiu, nyiu', nyo, nyo', nyu,* and *nyu'*, never mind the potential inclusion of tonal contrasts. The digraph needs to be included to account for hypercorrection, where /ŋj/ clusters are often shifted to [ɲ]. The apostrophe, representing a glottal stop, is included due to final /h/ often being added to CV syllables, but also occasionally representing a glottal stop. These two features must be handled together since it is impossible to know without already knowing the meaning of a word whether final <h> represents /h/ or /ʔ/. For this reason, while there is no glottal stop on MOTHER, all final strings are assumed to be glottal stops based on conventions of the dictionary's spelling system which normalises all instances of /ʔ/ to <'>, but common spelling does not, and so we need to anticipate their presence.

Again, implementing this directly in the query would be all too easy if regular expressions were available. In SQLite, however, this means either pre-generating the replacements in the client and sending a query like this:

```
SELECT * FROM wolam WHERE headword IN (
"ngio", "ngio'", "ngiu", "ngiu'", "nio", "nio'", "niu", "niu'",
"nyio", "nyio'", "nyiu", "nyiu'", "nyo", "nyo'", "nyu",
and "nyu'" );
```

Or alternatively, pre-generating a more complex query in the client through logic operators and sending that, or pre-writing all the rules into a similarly complex but larger query one time and running that. This could work, but interpretations of phonologies change as people learn more about the language as mentioned above — even if it their own — and by hard-coding such a lengthy set of replacement rules again removes one of the major requirements of the system: ease of maintainability. We will come back to this, but lacking regular expressions, our second possible solution will not work.

For SQLite then, we move past the fuzzy search being handled by the query itself, and instead move on to Option 3.

3.3 Solution 3: Fuzzy Search as a Prepared Query in the Code

Specifically, we will implement a simple finite state transducer (FST). This is possibly the most expensive solution, since it requires the processing to happen either in the client's browser or in the server code, but the costs are still negligible. For the limited data variation we have, this will still be incredibly performant. By handling fuzzy string matching with an FST, the costs are still exceptionally low, and we are only worrying about a word or two at a time. It is also the least portable in the sense of porting to other languages since it means we have to rewrite the FST in whatever new language we may want to move to. Still, the amount of code necessary to implement this is minimal, so porting would also not be difficult.

Importantly, this option also gives us the most control over the results. By having the replacements in a self contained object, we can easily modify on that to update our conversions based on evolving needs of the language community.

This caters to another requirement of the system which has not yet been mentioned: trustworthiness. For any community-facing resource to be accepted and used, and therefore useful, it needs to be trusted by the users. From the perspective of the community, any mistakes will be taken as sloppiness or a lack of understanding of the language on the part of, well, me as the linguist. Additionally, language resources which are purported to present a more "scientific" view of the language but fail to accurately address expectations of the end user will ultimately be rejected. Thus an overly phonemicised system abstracted from the speaker's interaction with it is also to be avoided. We can take the steps discussed in the following section.

4 Fuzzy Search in JavaScript

A number of steps are involved here, and the order in which they appear in the code does not necessarily need to follow as described below. A full working example is available on GitHub (patkaiist/calc-fst) which can be run directly as a command line Node.js script.

4.1 Step 1: Input Conversions

The first step in terms of finding actual alternative spellings is to compress orthographic phoneme strings into single glyphs. Since the iterating operation for replacements is happening at the glyph level, we need to convert all digraphs and trigraphs to single-glyph equivalents so that the replacements can go one glyph at a time and not miss anything.

Here, term is the string passed to the application as the search string, defined in our application as data sent via a POST request to our dictionary web application.

```
term = term.replace(
  /ng|ny|ni|sh|si|ch|chh|j/g,
  function (match) {
    switch (match) {
      case "chh":
        return "ç";
      case "ch":
      case "j":
        return "ʧ";
      case "ng":
        return "ŋ";
      case "ny":
      case "ni":
        return "ɲ";
      case "sh":
      case "si":
        return "ʃ";
      default:
        return match;
    }
  }
);
```

In most cases these will be the IPA equivalents, but not always; <c> us used in the language orthography but is not equivalent to /c/, so instead we use <ç> to represent the aspirated affricate /tsh/. This won't matter, and will not affect the final results other than to ensure that <c> doesn't get missed otherwise. We use the ligature <ʧ> for the /ʧ/ phoneme, and <ç> for the aspirated equivalent. We could just as easily use 👍 and ✊, as long as we have single glyphs.

4.2 Step 2: Replacements Object

Now we create the opposite: an object relating a single-glyph to array of possible spelling variants. This is the object which can also be readily updated and changed as necessary should conventions change over time among speakers.

```
const replacements = {
  ɲ: ["ni", "nyi", "ny"],
  ŋ: ["ng", "ny"],
  ʃ: ["sh", "s", "si"],
  ʧ: ["ch", "j"],
  ç: ["ch", "chh"],
  ...
};
```

This includes many more replacements, such as dealing with diaeresis on different vowels, vocalic digraphs, and variations on the use of <h> by speakers. The replacement object also necessarily creates some letter combinations which are not actually found, such as <nyi> becoming <nii> as one variation. These are handled further on in the code.

All values in this object are based on real-world substitutions which occur, such as palatalisation of <ng> /ŋ/ when followed by /i/ being interpreted as underlyingly <ny> /ɲ/. This helps us account for hypercorrection and other potential unexpected spellings. The full list of replacements gets quite large, but by doing it in this way, they can be easily seen and adjusted should any mistakes arise.

Replacements are also intentionally unidirectional. For example <ei> can be converted to <e> or <i>, but we don't want to go the other way. The <ei> spelling is more conservative and reflects the pronunciation of the standard language Thang, while Wolam has undergone a sound change from **ei* to /i/, variably written <e> or <i>. For <iu> and <io>, these are listed once each in both directions, since those are variations we can expect to happen either way.

4.3 Step 3: The Finite State Transducer

Once we have our two conversions ready above, we can now apply them. We use the following functions to actually handle the conversions.

```
function generateVariations(input) {
  if (!input || input.length === 0) {
    return [""];
  }
  const segments = input.replace(/\./g, '.·').split('·');
  let variations = [''];
  for (const segment of segments) {
    const segmentVariations = [];
    for (const variation of variations) {
      const segmentVariants = generateSegmentVariations(segment);
      for (const segmentVariant of segmentVariants) {
        let variant = variation + segmentVariant;
        segmentVariations.push(replaceWithFirst(variant));
      }
    }
  variations = segmentVariations;
  }
  return variations
}
```

```
function generateSegmentVariations(segment) {
  let variations = [segment];
  for (const pattern in replacements) {
    let index = 0;
    while ((index = segment.indexOf(pattern, index)) !== -1) {
      for (const replacement of replacements[pattern]) {
        let variant = segment.slice(0, index)
          + replacement
          + segment.slice(index + pattern.length);
        variations.push(replaceWithFirst(variant));
      }
    index += 1;
    }
  }
  return variations;
}
```

Some additional features are there, such as replacing a full stop with a middle dot. We use the middle dot anyway in the display, and the conversion happening this early prevents any issues with the role of a full stop in regular expressions.

As mentioned above, the replacements also introduce a few issues which we want to resolve to keep things a little cleaner in the results. One issue with the replacements object above is it produces <ii> segments, which are not meaningful and may prevent some results from being shown. We therefore clean the output once more with the following, in which we include any problematic sequence in the replace() regex matching:

```
function cleanOutput(input) {
  return input.replace(
    /ii|iu/g, function(match) {
      switch (match) {
        case 'ii':
          return 'i';...
        default:
          return match;
      }
    }
  );
}
```

This can be modified as needed for any further changes to the original replacements object, but generally, for this particular dictionary, omitting non <a> double vowels is sufficient.

4.4 Step 4: Assembling the Query

Finally, we put it all together into an SQL query which we can then run on our database and return in whatever front end system we want to run. The query can be as simple or as complex as needed. Here, we have a simple version which returns at most 25 matches.

```
let query = 'SELECT * FROM `table` WHERE ('
for (let i = 0; i < output.length; i++) {
  query += '`headword` LIKE "' + output[i] + '"';
  if (i < output.length - 1) {
    query += " OR ";
  }
}
query += ') ORDER BY `headword` ASC LIMIT 25;';
```

Here we are only looking at the headword, but in practice it would be trivial to also search within definitions or any other field as needed.

5 Conclusion

The demand for dictionaries is high among communities speaking otherwise under-represented languages. In many cases, these may begin as word lists collected by linguists or community scholars who may not yet have determined the best orthographic representation. In other cases, spelling variants may be prevalent even in situations where a standard orthography has been in use, either due to contractions, assimilation, folk etymologies, tone changes, or any other of a number of factors which may be reflected in spelling.

To maximise the usefulness of a dictionary for such a language — especially those which are primarily community-facing but also for those in use by scholars — the ability to correctly find the appropriate word in spite of potential spelling variation is of major importance. It is all the more important in cases where the language is not taught in schools, and therefore may lack any type of standardised form being disseminated to the speakers.

The solution presented here is just one of many. It represents an easily controlled approach which allows variants to be specified based on the conventions in use by speakers or data collectors, with the option to run conversions recursively for maximal

matching. Additionally, it provides a more controlled set of results than simple wildcard matching such as that used on many popular sites for under-described language dictionaries, which ultimately prevent the search function from being useful by returning too many results and not giving options for more selective searches.

Supplementary Material
A minimal working example and all code needed to run the examples described here are curated on GitHub (https://github.com/patkaiist/calc-fst). To check the code discussed in action, the code has also been shared in the form of a JSFiddle (https://jsfiddle.net/patkaiist/4f16rqtb).
Acknowledgements
Example code is based on work for the Wolam Ngiopit online dictionary, developed in collaboration with Ms. Keen Thaam.