ISMLA Project: Malayalam Glosser

Thora Daneyko

March 16, 2018

Abstract

Miau

1 Introduction

Bla bla bla test മലയാളി bla bla.

2 Malayalam Language Processing

2.1 About Malayalam

Malayalam is a Dravidian language spoken by over 30 million people in the southern Indian state Kerala. Like most Dravidian languages, Malayalam has SOV word order and a rich agglutinative exclusively suffixing morphology. The verbal morphology is especially complex, as verbs can be marked for various tenses, aspects and moods and may be chained together into long compounds to express subtle differences in meaning (Asher and Kumari 1997).

2.2 NLP challenges

2.2.1 Parsing the Malayalam script

Malayalam is written in Malayalam script, an abugida descended from the Brahmi script. The basic characters represent a syllable composed of a consonant and the inherent vowel /a/. The inherent vowel can be changed by attaching a vowel diacritic to the base character. Hence, the symbol ക represents the syllable /ka/, but with the diacritic for /i/ or /ē/ it becomes കി /ki/ or കേ /kē/. Similarly, the inherent vowel may be deleted using the diacritic that is known as *candrakkala* 'half moon' in Malayalam and *virama* or *halant* in many other Indic languages to represent a consonant without vowel, as in ക് /k/. In Malayalam, however, the *candrakkala* has a phonetic value of its own, often transcribed as a short close or mid unrounded vowel ([ɨ] or [ə]). The only consonants that can appear at the end of a word without being followed by this vowel are /m/, /n/, /ɳ/, /l/, /ɭ/ and /r/. For this reason, Malayalam has its own characters for these sounds without the inherent vowel (except /m/, which is represented by the *anusvāraṁ* diacritic ◌ം), called *cillu*: ൻ, ൺ, ൽ, ൾ and ർ.

Each base character and diacritic has its own Unicode code point (The Unicode Consortium 2007, p. 334ff). Hence, the syllable ക /ka/ consists of one, കി /ki/ of two and ക് /k/ also of two code points. A simple one-to-one mapping on Latin characters is therefore not possible. Vowel diacritics which are visually composed of two others, but denote a single vowel, also have their own code points. For example the diacritic for /o/ ◌ൊ (as in കൊ /ko/) is not a sequence of /e/ ◌െ (as in കെ /ke/) and /ā/ ◌ാ (as in കാ /kā/), but a single, independent code point (The Unicode Consortium 2007, p. 334f). However, the sequence base glyph + /o/ is visually indistinguishable of base glyph + /e/ + /ā/ in most fonts, so both variants can be observed in Malayalam texts. The *cillu*s now have their own code points as well (The Unicode Consortium 2008), however, before Unicode 5.1, these were typed as base glyph + *candrakkala* + zero-width joiner (U+200D) (The Unicode Consortium 2007, p. 336f), remnants of which are also still commonly present in Malayalam texts on the web.

Conversion from Malayalam script into some other format therefore holds a few difficulties that one must be aware of. However, converting Malayalam script into some alphabetic representation is an important preprocessing step for morpheme splitting, since Malayalam morphemes are not necessarily syllabic and can therefore only hardly be represented and analyzed in the Malayalam script.

2.2.2 Tokenization

The Malayalam script generally separates words by whitespaces, just like the Latin script. However, there is a tendency to merge adjacent words in writing. Thus, the two-word sentence ടീച്ചർ ആണ് *ṭīccar āṇ˘* ‘is a teacher’ may also be written as a single word: ടീച്ചറാണ് *ṭīccar̲āṇ˘*. This may include any number of words from any part of speech and does not only occur in literature, as in (1), but also in everyday speech and writing, as in (2).

(1) മേഘം പോലെ കറുപ്പുനിറഞ്ഞോട്ടുകൂടിയവർ ആണ്.
Mēghaṁ pōle kar̲uppunir̲aññōṭukūṭiyavar āṇ˘.

മേഘം	പോലെ	കറുപ്പ്	നിറഞ്ഞോട്	കൂടി	അവർ	ആണ്
mēghaṁ	pōle	kar̲upp˘	nir̲añ-ñ-ōṭ˘	kūṭi	avar	āṇ˘
cloud	like	black	be.full-PSTPART-SOC	with	they	be

‘They are black like clouds.’ (Vēṇugōpālan 2009, p. 179)

(2) അതിന് നിനക്കെന്താ?
Atin˘ ninakkentā?

അതിന്	നിനക്ക്	എന്ത്	ആണ്
at-in˘	nin-akk˘	ent˘	āṇ˘
that-DAT	you-DAT	what	be

‘Why do you care?’ (Moag 1994, p. 165)

The above examples already indicate that even on the phonetic level this process is not always as simple as in ടീച്ചറാണ് *ṭīccar̲āṇ˘*, where the two words are just merged together. The changes that the affected words undergo when written as one are referred to as *external sandhi* (Devadath et al. 2014). Its counterpart, *internal sandhi*, describes the changes that occur when bound morphemes, such as case endings, are added to a stem. However, these rules are often specific to the suffix in question. The most common *external sandhi* rules that regularly apply when merging arbitrary words in a sentence are the following:

- Insertion of a glide between two vowels (/y/ or /v/ depending on the roundedness of the first vowel), as in (1) കൂടിയവർ *kūṭiyavar* (കൂടി *kūṭi* + അവർ *avar*).
- Dropping of the *candrakkala* vowel when merging with a word starting with a vowel, as in (2) നിനക്കെന്താ(ണ്) *ninakkentā(ṇ˘)* (നിനക്ക് *ninakk˘* + എന്ത് *ent˘* + ആണ് *āṇ˘*).
- The *candrakkala* vowel becoming /u/ when merging with a word starting with a consonant, as in (1) കറുപ്പുനിറഞ്ഞോടുകൂടി *kar̲uppunir̲aññōṭukūṭi* (കറുപ്പ് *kar̲upp˘* + നിറഞ്ഞോട് *nir̲aññōṭ˘* + കൂടി *kūṭi*).
- Doubling of an initial consonant (especially plosives) when preceded by a vowel or *cillu* consonant. This is very frequent in compounds, such as അരിപ്പെട്ടി *arippeṭṭi* ‘rice box’ (അരി *ari* + പെട്ടി *peṭṭi*) or പാൽക്കുപ്പി *pālkkuppi* ‘milk bottle’ (പാൽ *pāl* + കുപ്പി *kuppi*) (Asher and Kumari 1997, p. 397). It also occurs in chains of verbs, e.g. when merging the verb കൊടുക്കുക *koṭukkuka* ‘to give’ with the past tense form of the verb പെടുക *peṭuka* ‘to fall into’ to create the passive expression കൊടുക്കപ്പെട്ടു *koṭukkappeṭṭu* ‘was given’ (കൊടുക്ക *koṭukka* + പെട്ടു *peṭṭu*) (Asher and Kumari 1997, p. 269).
- (Orthographic change only:) The *cillu*s and the *anusvāraṁ* becoming their full counterparts before a vowel, as in സുഖമാണോ? *sukhamāṇō?* ‘how are you/are you well?’ (സുഖം *sukhaṁ* + ആണോ *āṇō*) (Moag 1994, p. 30).
- Dropping of the *anusvāraṁ* before a consonant, as in പുസ്തകപ്രേമം *pustakaprēmaṁ* ‘love of books’ (പുസ്തകം *pustakaṁ* + പ്രേമം *prēmaṁ*) (Asher and Kumari 1997, p. 398).

For a Malayalam tokenizer, it is therefore not sufficient to extract tokens separated by whitespaces and punctuation, it must also be able to identify and split merged words and reverse the *sandhi* that has altered the participating tokens. This task is not trivial, and several strategies have been developed to perform it.

RESEARCH

3 The Malayalam Glosser

Bla bla

3.1 Transliteration

3.1.1 Supported Scripts

3.1.2 Transliterators

MalayalamTranscriptor

3.2 Morphology Generation

MorphGen

3.3 Tokenization

MalayalamGlosser

3.4 Dictionary lookup

MalayalamDictionary

3.4.1 Efficiency considerations

Considering that the dictionary may be very large and that the main function of the Glosser is to look words up in this dictionary, being able to load and query it very quickly is essential for the performance of the Glosser. Hence, I experimented with a few alternatives for storing the dictionary data and investigated their efficiency. The tests elaborated below are not very exact or well-designed and were only meant to quickly assess the usefulness of the considered methods.

`HashMap` vs. `ReverseTrie`

The straightforward way to represent a dictionary as a Java object is a `HashMap`. Apart from being readily available and easy to use, querying a `HashMap` is fast. However, this also means that all entries are stored as their complete String representation, which may consume quite a lot of space. Considering that the inflected forms of the words share most of their characters, a trie representation seemed quite suitable and might be able to save space compared to a simple `HashMap`. Since Malayalam is exclusively suffixing, I programmed a `ReverseTrie` which reads and retrieves the strings from last to first character, in order to save as much space as possible. A useful side effect of this is that the tokenizer does not need to look up all suffixes of a compound word in the dictionary, but can simply do a suffix search of the `ReverseTrie` to get the longest contained suffix.

In order to compare the performance of a `HashMap` and `ReverseTrie` based dictionary, I measured the memory used by the program before loading the dictionary data and after creating the `HashMap` and Trie (calculated as `Runtime.totalMemory() - Runtime.freeMemory()` after a `System.gc()` call). Then I let the dictionary find the longest known suffix of the test String *aviṭeyuḻḻataṟiññu* (*aviṭe uḻḻatˇ aṟiññu* "knew (he) was there") 1,000,000 times and measured the time needed by a MashMap and `ReverseTrie` based dictionary (calculated using `System.currentmillis()`). Finally, I rewrote the tokenizer to also work with a `ReverseTrie` and tested how long tokenization of a short conversation from Moag (1994) took it with the two dictionary types.

Despite the many shared suffixes, the `HashMap` was smaller than the `ReverseTrie`, taking up 8,318,164.8 bytes on average during five test runs, while the Trie required 12,590,051.2 bytes. However, the memory used by the `HashMap` varied greatly, ranging from only 5,160,456 to 9,801,392 bytes, while the Trie always consumed almost exactly the same amount of memory. This indicates that the measurements might have been distorted by background processes such as the garbage collection. However, the `HashMap` still seems to be considerably smaller.

As expected, the `ReverseTrie` outperformed the `HashMap` on the looped suffix search of *aviṭeyuḻḻataṟiññu.* The Map took an average of 999 milliseconds during five test runs, while the Trie only needed 312.4 ms. However, the performance of the Trie was very unstable, ranging from 140 to 518 ms between runs, while the `HashMap` always needed between 908 and 1049 ms, which is still much slower than the slowest suffix search of the Trie.

On a real Malayalam text, where only few words are long compounds such as *aviṭeyuḻḻataṟiññu*, both methods were equally fast. During 10 glossings of the Moag conversation, the Map based tokenization took 156.3 ms on average and the Trie based tokenization 161.7 ms. Both ran very stable.

All in all, the `HashMap` seems to be the better choice, since it is smaller than the Trie and equally fast on normal Malayalam texts. The Trie is faster when tokenizing long compound words, which however are not frequent enough to justify preferring it over the `HashMap`.

File storage vs. Serialization

Loading the dictionary data into the underlying `HashMap` (or `ReverseTrie`) takes a considerable amount of time at launch. Hence, I considered serializing the Map or Trie object to be able to load it quicker. Since the Java serialization is known to rather slow, I used the FST Fast Serialization library for my tests. I first read the dictionary data from the text file and created the `HashMap` and `ReverseTrie` from it, measuring the time needed. Then I serialized the two objects and took the time required to deserialize them.

During five test runs, parsing the text file into an object took 278.2 ms on average for the `HashMap` and 310.6 ms for the Trie. Deserializing the same objects required 563.4 ms on average for the `HashMap` and 339.8 ms for the Trie.

Loading the data from a text file is thus faster than deserializing a previously created object.

The file storing the serialized `ReverseTrie` was twice as large as the file with the `HashMap`. This confirms my assertions from the previous section that the Trie takes more space than the `HashMap`.

3.5 UI Design

4 Conclusion

References

Asher, Ronald E. and T. C. Kumari (1997). *Malayalam.* Psychology Press.

Bindu, MS and Sumam Mary Idicula (2011). "High Order Conditional Random Field Based Part of Speech Tagger and Ambiguity Resolver for Malayalam-a Highly Agglutinative Language." In: *International Journal of Advanced Research in Computer Science* 2.5.

Devadath, V. V. (2016). "A Shallow Parser for Malayalam." MA thesis. Hyderabad: International Institute of Information Technology.

Devadath, V. V. et al. (2014). "A Sandhi Splitter for Malayalam." In: *Proceedings of the 11th International Conference on Natural Language Processing*, pp. 156–161.

Gamliel, Ophira, ed. (forthcoming). *God's Own Language. Malayalam Grammar Text Book.* Draft from July 2016.

Jayan, Jisha P, RR Rajeev, S Rajendran, et al. (2011). "Morphological analyser and morphological generator for malayalam-tamil machine translation." In: *International Journal of Computer Applications* 13.8, pp. 0975–8887.

Kuncham, Prathyusha et al. (2015). "Statistical sandhi splitter for agglutinative languages." In: *International Conference on Intelligent Text Processing and Computational Linguistics.* Springer, pp. 164–172.

Manju, K, S Soumya, and Sumam Mary Idicula (2009). "Development of a POS tagger for Malayalam-an experience." In: *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom'09. International Conference on.* IEEE, pp. 709–713.

Moag, Rodney F. (1994). *Malayalam: A University Course and Reference Grammar.* Austin: University of Texas, Center for Asian Studies.

Nisha, M and PC Reghu Raj (2016). "Sandhi Splitter for Malayalam Using MBLP Approach." In: *Procedia Technology* 24, pp. 1522–1527.

Rajeev, RR and Elizabeth Sherly (2007). "Morph analyser for malayalam language: A suffix stripping approach." In: Proceedings of 20th Kerala Science Congress.

Sebastian, Mary Priya and G Santhosh Kumar (n.d.). "Machine Learning Approach to Suffix Separation on a Sandhi Rule Annotated Malayalam Data Set." In:

Suneera, CM and MT Kala (n.d.). "A Rule Based Approach For Malayalam-English Translation." In:

The Unicode Consortium (2007). *The Unicode Standard, Version 5.0.* Boston: Addison-Wesley.
– (2008). *Unicode 5.1.0.* URL: `http://www.unicode.org/versions/Unicode5.1.0/`.
Vēṇugōpālan, P., ed. (2009). *Āścaryacūḍāmaṇi. Sampūrṇṇamāya āṭṭaprakāravum kramadīpikayum.* Tiruvanantapuram: Mārgi.