MLO Modern Languages Open

Dombrowski, Q 2020 Preparing Non-English Texts for Computational Analysis. *Modern Languages Open*, 2020(1): 45 pp. 1–9. DOI: https://doi.org/10.3828/mlo.v0i0.294

ARTICLE – DIGITAL MODERN LANGUAGES

Preparing Non-English Texts for Computational Analysis

Quinn Dombrowski
Stanford University, US
qad@stanford.edu

Most methods for computational text analysis involve doing things with "words": counting them, looking at their distribution within a text, or seeing how they are juxtaposed with other words. While there's nothing about these methods that limits their use to English, they tend to be developed with certain assumptions about how "words" work – among them, that words are separated by a space, and that words are minimally inflected (i.e. that there aren't a lot of different forms of a word). English fits both of these assumptions, but many languages do not. This tutorial covers major challenges for doing computational text analysis caused by the grammar or writing systems of various languages, and ways to overcome these issues.

Introduction

Most methods for computational text analysis involve doing things with 'words': counting them, looking at their distribution within a text or seeing how they are juxtaposed with other words. While there's nothing about these methods that limits their use to English, they tend to be developed with certain assumptions about how 'words' work – among them, that words are separated by a space, and that words are minimally inflected (i.e. that there aren't a lot of different forms of a word). English fits both of these assumptions, but many languages do not. Depending on the text analysis method, a sufficiently large corpus (on the scale of multiple millions of words) may sufficiently minimize issues caused by inflection, for instance at the level commonly found in Romance languages. But for many highly inflected Slavic and Finno-Ugric languages, Arabic, Quechua, as well as historical languages such as Latin and Sanskrit, repetitions of what you think of as a 'word' will be obscured to algorithms with no understanding of grammar, when that word appears in different forms, due to variation in the number, gender or case in which that word occurs. To make it possible for an algorithm to count those various word forms as the same 'word', you need to modify the text before running the analysis. Likewise, if you're working with Japanese or Chinese, which don't typically separate words with spaces, you need to artificially insert spaces between 'words' before you can get any meaningful result. For example, 'I went to Kansai International Airport' is written in Japanese as 関西国際空港に行きました. The lack of spaces between words means that tools dependent on spaces to differentiate (and then count) words will treat this entire sentence as a single 'word'. Segmentation – the process of adding spaces – is not always an obvious or straightforward process; on one hand, it's easy to separate 'to' and 'went' from the

name of the airport (関西国際空港 'Kansai International Airport' に 'to' 行きました 'went'), but depending on what sorts of questions you are attempting to answer with the analysis, you may want to further split the proper name to separate the words 'international' and 'airport', so that they can be identified as part of a search, or contribute to instances of those words in the corpus: 関西 'Kansai' 国際 'international' 空港 'airport' に 'to' 行きました 'went'.

Goals

This tutorial covers major challenges to doing computational text analysis caused by the grammar or writing systems of various languages, and offers ways to overcome these issues. This often involves using a programming language or tool to modify the text – for instance by artificially inserting spaces between every word for languages such as Chinese that aren't regularly written that way, or replacing all nouns and verbs with their dictionary form in highly inflected languages such as Finnish. In both of these situations, the result is a text that is less easy to parse for a human reader. Removing inflection may have the effect of making it impossible to decipher the meaning of the text: if a language has relatively flexible word order, removing cases renders it impossible to differentiate subjects and objects (e.g. who loved whom). But for some forms of computational text analysis, the 'meaning' of any given sentence (as readers understand it) is less important; instead, the goal is to arrive at a different kind of understanding of a text using some form of word frequency analysis. By modifying a text so that its 'words' are more clearly distinguishable using the same conventions as found in English (spaces, minimal word inflection etc.), you can create a text derivative that is specifically intended for computation and will lead to much more interpretable computational results than if you give the algorithm a form of the text intended for human readers.

While this lesson provides pointers to code and tools for implementing changes to the text in order to adapt it for computation, the landscape of options is evolving quickly and you should not feel limited to those presented here.

Audience

Text analysis methods are most commonly used in research contexts, and frequently appear as part of 'an introduction to digital humanities' and similar courses and workshops. While these courses are taught worldwide, the example texts are, most often, in English, and the application of these text analysis methods may not be as straightforward for students working in other languages. This tutorial is intended for instructors of such workshops, to help them be better informed about the challenges and needs of students working in other languages and to provide them with pointers for how to troubleshoot issues that may arise.

For instructors of modern languages, text analysis methods can also have a place in intermediate to advanced language courses (see Cro & Kearns). For instance, while many digital humanities researchers now use more nuanced methods than word clouds, they can still be employed in a language pedagogy context to provide a big-picture visualization of word frequency – starting with the generic and obvious (prepositions, articles, pronouns etc.) and becoming more and more related to the content of the text as students apply and refine a stopword list (a list of words that should be removed prior to doing the word counts and generating the visualization). Depending on the text, even a word cloud may make visible the impact of inflection, as it may contain multiple forms of a given 'word', which can spur discussion about what constitutes a 'word'. Intuitively, we think of *saber* ('to know' in Spanish) as the 'same word' as *sé* 'I know', *sabemos* 'we know', *sabía* 'knew' and so on, but what do we gain and lose if we treat them as 'different words', the way a computer would by default?

Text encoding

Text encoding – or how the low-level information about how each letter/character is actually stored on a computer – is important when working with any text that involves characters beyond unaccented Latin letters, numerals and a small number of punctuation marks.[1] It may be tempting to think languages that use the Latin alphabet are safe from a particular set of challenges faced by other writing systems when it comes to computational text analysis. In reality though, many writing systems that use the Latin alphabet include at least a few letters with diacritics (e.g. é, ñ, or ż), and these letters cause the same issues as a non-Latin alphabet, albeit on a smaller scale. While a text in French, Spanish or Polish may be decipherable even if all of these characters are mangled (e.g. ma□ana for mañana is unlikely to cause confusion, and even a less obvious case such as a□os for años is often distinguishable by context), issues with text encoding may cause bigger problems later in your analysis – including causing code to not run at all. For languages with a non-Latin alphabet, text encoding problems will render a text completely unreadable and must be resolved before doing anything at all with the text. Unicode (UTF-8) encoding is the best option when working with text in any language, but particularly non-English languages.

What is Unicode?

Unicode is the name of a computing industry standard for encoding and displaying text in all writing systems of the world. While there are scripts that are not yet part of Unicode as of 2020 (including Demotic and some Egyptian hieroglyphs), researchers affiliated with the Unicode consortium have done a tremendous amount of work starting in the late 1980s to differentiate *characters* (graphemes, the smallest units of a writing system) versus *glyphs* (variant renderings of a character, which look a little different but have the same meaning) for the world's writing systems, and assign unique *code points* to each character. With some writing systems – including Chinese and various medieval scripts – the decision of what constitutes a character as opposed to a glyph is at times controversial. Scholars who disagree with previous decisions or who feel that they have identified a character that is not represented in Unicode, can put forward proposals for additions to the standard. While the Unicode consortium that shapes the development of the standard is primarily made up of large tech companies, scholars and researchers play a significant role in shaping decision-making at the language level (Anderson).

Why is Unicode important?

Before Unicode was widely adopted, there were many other standards that developed and were deployed in language-specific contexts. Windows-1251 is an encoding system that was widely used for Cyrillic and is still used on 11% of websites with .ru (Russian) domain names (W3Techs). A competing, but less common, Cyrillic encoding for Russian was KOI8-R, and a similar one, KOI8-U, was used for Ukrainian. For Japanese, you may still encounter websites using Shift JIS encoding. For Chinese, you can find two major families of encoding standards prior to Unicode, Guobiao and Big5. A major advantage of Unicode, compared to these other encoding standards, is that it makes it possible to seamlessly read text in multiple languages and alphabets. Previously, if you had a bilingual parallel edition of a text on a single webpage with languages that used two different writing systems, you would have to toggle between

[1] Note that 'encoding' here refers to the comparatively low-level technical process of standardizing which bits represent which letters in various alphabets. This is a different use of the term than the 'encoding' in the Text Encoding Initiative (TEI), https://tei-c.org, which captures structural and/or semantic features of text in a potentially machine-readable way.

multiple text encodings – reducing one side of the text, then the other, to gibberish as you switched between them.

If you work in a language with a non-Latin alphabet, odds are good that you'll encounter text that doesn't use Unicode encoding at some point in your work. Long-running digital text archives, in particular, are likely candidates for not having migrated to Unicode. If you try to open a text file using the wrong kind of encoding, you won't see text in the alphabet you're expecting to see, but rather a kind of gibberish that will soon become familiar. (For instance, Windows-1251 Cyrillic looks like Latin characters with diacritics: "Äîñòîåâñêèé Ôåäîð Ìèõàéëîâè÷. Ïðåñòóïëåíèå è íàêàçàíèå" for "Достоевский Федор Михайлович. Преступление и наказание" – Dostoevsky Fyodor Mikhailovich. *Crime and Punishment*.)

Making sure your text uses Unicode encoding

Most computational text analysis tools and code assume that the input text(s) use UTF-8 (Unicode) encoding. If the input text is not in UTF-8, you may get an error message, or the tool may provide an 'analysis' of the unreadable gibberish (**Figure 1**).

It is not obvious what encoding a text file uses: that information isn't included in the file properties available on Windows or Mac. There isn't even an easy way to write Python code to reliably detect a file's encoding. However, most *plain text editors* have some way to open a text file using various encodings until you find one that renders the text readable, as well as some way to save a text file with UTF-8 encoding. A plain text editor is software that natively reads and writes .txt files, without adding in its own additional formatting (which Notepad does in Windows). Atom is a cross-platform (Windows/Mac/Linux) plain text editor that you can install if you don't already have a preferred editor.[3]

There are numerous *packages* (add-ons) for Atom that provide additional functionality. One of these is called convert-file-encoding.[4] Download and install this add-on following the instructions in the Atom documentation.[5]

Once you've installed the convert-file-encoding package, open your text file in Atom. By default, Atom tries to open everything as UTF-8. If everything displays correctly, your file already uses Unicode encoding. If the text is gibberish, go to *Edit* > *Select encoding*, and

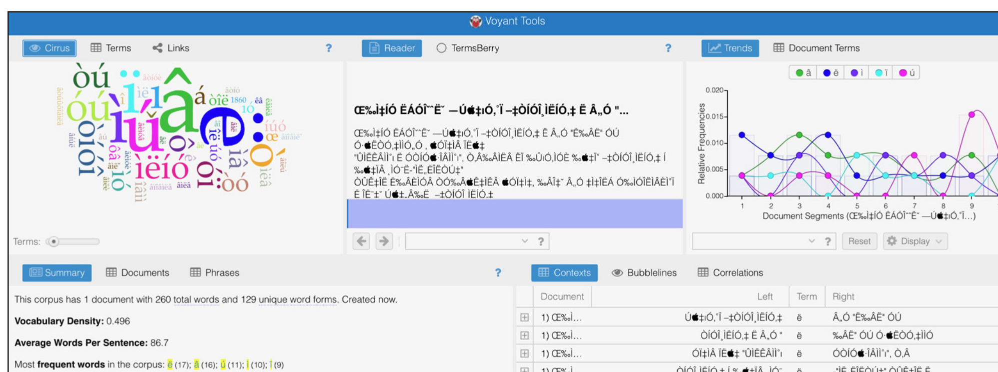

Figure 1: Voyant 'analysis' of Windows-1251 encoded Russian text.[2]

[2] Voyant Tools, https://voyant-tools.org/.
[3] Atom is available for download at https://atom.io/.
[4] The convert-file-encoding package is available at https://atom.io/packages/convert-file-encoding.
[5] Atom documentation is available at https://flight-manual.atom.io/using-atom/sections/atom-packages/.

choose a possible candidate encoding. The encodings are listed in Atom by what languages they cover, so you can try different options for your language if you're not sure.

Once your text appears normally, go to *Packages* > *Convert to encoding* and select *UTF-8*. Then save your file.

Segmentation

For Chinese and Japanese text, you need to *segment* your text, or artificially insert spaces between 'words', before you can use it for computational text analysis. For Chinese, some scholars treat every character as a 'word'. This destroys compounds but is more predictable than using a segmenter. For both Chinese and Japanese, segmenters work best when the text does not contain a lot of jargon or highly specialized vocabulary, or non-standard orthography (e.g. Japanese children's writing, which often uses the *hiragana* syllabary where a fully literate adult would use *kanji*).

Stanford NLP (natural language processing) provides a Chinese segmenter[6] with algorithms based on two different segmentation standards.[7]

For Japanese, segmentation is available through the mecab software.[8]

Rakuten MA is a Javascript-based segmenter that supports Chinese and Japanese.[9] There is also a Python implementation, Rakuten MA Python.[10] If you have trouble with mecab but aren't comfortable writing Python code yourself, there's a Jupyter Notebook available for segmenting Japanese.[11] See the *Programming Historian* tutorial 'Introduction to Jupyter Notebooks' (Dombrowski et al.) for a description of Jupyter Notebooks and how to use them.

Stopwords

Stopwords are words that are filtered out as the first step of text analysis. Many tools have a configuration option where you can define which words should be treated as stopwords.[12] Stopword removal is essential for some methods (including word clouds and topic modelling), to avoid having your results flooded with articles, copulas, prepositions and the like. Other methods, such as word vectors (which analyse words in their context as a way to explore semantic relationships within large corpora), rely on stopwords for important information about the semantic value of words, and stopwords should be retained in the text.

Stopwords are language specific, and more nuanced use of stopwords can involve *text-specific* lists that also exclude things like character names (which are likely to occur with high frequency, but that frequency may or may not be meaningful depending on your research question). If you're using a tool that supports the use of stopword lists, you should check to make sure that a default, almost certainly English, stopword list isn't being applied to your non-English text.

Some tools provide reasonable built-in stopword lists for multiple languages. Voyant offers generally reasonable lists for thirty-four languages, along with a combined 'multilingual' setting, and an option for defining your own list. These lists are not identical: the Russian list

[6] The Stanford NLP segmenter can be downloaded at https://nlp.stanford.edu/software/segmenter.shtml.

[7] This Chinese part-of-speech tagger tutorial begins with a step-by-step guide to segmenting with the Stanford NLP segmenter: https://github.com/quinnanya/dlcl204/blob/master/chinese/pos_chinese.md.

[8] Mecab can be downloaded at https://taku910.github.io/mecab/.

[9] Rakuten MA is available at https://github.com/rakuten-nlp/rakutenma.

[10] Raktuen MA Python is available at https://github.com/ikegami-yukino/rakutenma-python.

[11] The Jupyter notebook for running Rakuten MA Python is available at https://github.com/quinnanya/japanese-segmenter.

[12] See the settings for the Topic Modeling Tool (https://senderle.github.io/topic-modeling-tool/documentation/2018/09/27/optional-settings.html) or general purpose text exploration environment Voyant (https://voyant-tools.org/docs/#!/guide/stopwords).

includes the words for many numbers (including пятьдесят 'fifty'), the Spanish list has no numbers but does include various forms of *emplear* 'use', and the Czech list includes no numbers whatsoever but does have a number of words related to news (e.g. články 'articles'), hinting at the domain and context of its origins. Is it the right thing to do to eliminate written-out numbers from a Russian text, or any references to 'articles' in a Czech text? It all depends on what you're trying to learn from the text analysis. Students should examine – and, if necessary, modify liberally – any stopword list before applying it to their text. If you're a digital humanities instructor, be careful about uncritically recommending stopword lists for languages you can't read yourself. As an initial vetting step, at least run any list you find through Google Translate first, and read through it. There are many resources online that aggregate stopword lists for any number of languages, without considering that many of those lists were developed for very particular use cases, and might, for instance, remove all words about computers, along with the more-expected prepositions.

Your stopword list should be influenced by other changes you make to your text. In general, stopword lists are all lower case, due to the lower-casing that is typically part of the text analysis process. If you lemmatize your text (as described below), you won't need to include every possible form of pronouns: just the lemma. If you don't plan to lemmatize your text before the stopword list is applied, you'll need to work through every number, gender and/or case of undesired pronouns, adjectives, verbs and so forth, to ensure they are all excluded. Remember, these methods are matching, character-for-character, what you put on the list, and including the dictionary form of a word does *not* by extension include all conjugations, declensions or other variant forms.

Lower-casing

Capital letters and lower-case letters, in bicameral writing systems (those that have the concept of capitalization, unlike Japanese, Hebrew, Georgian or Korean), are *different characters* from the point of view of text analysis algorithms. *Dad*, *dad* and *Sad* are all treated as separate words, where the latter two are both parsed as having a different first letter from the first. To address this issue, texts are commonly 'lower-cased', or converted to all lower-case characters, before they are further processed with stopword removal or used for analysis. Most text analysis tools (e.g. with graphical user interfaces, like Voyant and the Topic Modeling Tool) handle this automatically, even for non-Latin alphabets. If you're writing analysis code yourself, don't forget this step.

Punctuation removal

What we easily recognize as punctuation is just another character from the point of view of most algorithms. This leads to problems when the following are all treated as different 'words':

- cats
- "cats
- "cats,
- (cats)
- cats!
- cats!!
- cats?!
- cats.

Some tools automatically remove punctuation as part of pre-processing, some tools include punctuation on the stopwords list and others require you to remove it from the text yourself.

For tools that remove punctuation automatically, you should check to make sure that all the punctuation present in your language is being removed successfully. Punctuation removal may be based on English, so punctuation not found in English (such as « » or 「」, the Russian and Japanese quotation marks, respectively) may not be included. Running the text through a tokenizer algorithm (such as the one provided by the Stanford NLP library for Python, which currently supports fifty-three languages) can also separate punctuation from text, but may make other changes you haven't anticipated. For instance, in English, a contraction like 'she's' gets split into two 'words', *she* and *'s*, which is a reasonable choice reflecting the word's origins, but can lead to initial confusion when you discover the 'word' *'s* in the results of your analysis.

Lemmatizing

If you're working with a highly inflected language (i.e. if your language has multiple grammatical cases, or a complex verbal system where different persons and numbers have different forms), you may need to *lemmatize* your text to get meaningful results from any text analysis method. Lemmatization attempts to convert the word forms actually found in a text into their dictionary form. For languages with less inflection (including Romance languages), many scholars don't feel the need to lemmatize because some methods, such as topic modelling, end up successfully clustering together different forms of a word, even given a small amount of variation. It could be a worthwhile activity with students to compare text analysis results with and without lemmatization for these languages.

A lot of work goes into developing NLP code for lemmatizing text, and not all lemmatizers perform equally well on all kinds of text: the informal language of tweets and the formal language of newspapers are different, to say nothing of literary and historical language. English is by far the best-resourced language, given the longstanding academic and commercial interest in improving NLP tools for at least modern English. Many languages lack effective lemmatizers, or any lemmatizers at all. If there's no lemmatizer for the language that you want to work with, another possibility is to look for a *stemmer*. Stemmers are a shortcut to the same fundamental goal as lemmatizers: reducing variation within a text, in order to more effectively group similar words. Rather than replacing the word forms in a text with the proper dictionary form, a stemmer looks for patterns of letters to chop off at the beginning and/or end of words, to get to something similar to (but often distinct from) the root of the word. Stemmers don't effectively handle suppletive word forms (e.g. 'children' as a plural of 'child'), or other word forms that diverge from the usual grammatical 'rules', but they may work well enough to reduce overall variation in the word forms present in a text, if no lemmatizer is available. The truncated forms produced by a stemmer may, however, be harder to recognize and connect back to the original form when you're looking at the results of your analysis.

The current state-of-the-art (whatever state that may be) for lemmatizing most languages is usually not available through an easy-to-use tool: you should expect to use the command line and/or write code. As a few illustrative examples:

- For Russian, Yandex (the major Russian search engine) has released software called MyStem for lemmatizing Russian.[13] A wrapper is available that makes this code usable in Python, PyMyStem.[14]

[13] MyStem is available at https://yandex.ru/dev/mystem/.
[14] PyMyStem is available at https://github.com/nlpub/pymystem3.

- For Basque, eustagger-lite (Ezeiza, N. et al.) processes text using the following steps: tokenization, segmentation, identifying grammatical part-of-speech, treatment of multiword expressions and morphosyntactic disambiguation.[15]
- While the concept of lemmatization doesn't quite carry over to Korean grammar, the KoNLPy package can be used for some kinds of potentially helpful text pre-processing (Kim).[16]
- The Classical Languages Toolkit (cltk.org) provides lemmatization for Latin, Greek and Old French, with other languages under development.[17]
- Lemmatization isn't enough for agglutinative languages such as Turkish, where very long words can be constructed by stringing together morphemes. The resulting complex words (e.g. Çekoslovakyalılaştıramadıklarımızdanmışsınız, 'you are reportedly one of those that we could not make Czechoslovakian') are rare, and therefore not ideal to use for word counts, but may consist of morphemes that *are* repeated with a frequency in the text that more closely resembles other languages' concept of a 'word'. Byte-pair encoding (Mao) is one algorithm that has been used as a reasonably effective shortcut to 'subword encoding' (similar to lemmatization, but for linguistic components smaller than a word, such as Turkish morphemes) without requiring tokenization or morphological analysis. Scholars have also worked on more nuanced, linguistically motivated segmentation using supervised morphological analysis as a way of addressing the challenges posed by agglutinative languages (Ataman et al.).
- Lemmatization isn't applicable to Chinese.[18]

Conclusion

Text preparation is essential for computational text analysis but how, specifically, you need to modify the text – and how best to go about doing that – will vary based on the research question, the method and the language. To even begin making sense of the output of computational text analysis, it is important to understand how the input text was processed, and to take precautions to ensure that default settings derived from English were not applied to languages with very different grammar or orthography.

Fortunately, there is a growing community of scholars working on computational text analysis, and other digital humanities methods, as applied to languages other than English. For scholars working with digital humanities methods, a community has begun to form around the mailing list and resources posted on the Multilingual DH website (https://www.multilingualdh.org), which is applying to become a special interest group of the Alliance of Digital Humanities Organizations. These resources, and their applications to digital humanities research as well as language pedagogy, continue to be refined, and self-identified 'newcomers' are welcome and encouraged to join the conversation.

Author Information

Quinn Dombrowski supports digitally-facilitated research in the Division of Literatures, Cultures & Languages at Stanford University in the USA. In addition to working on digital humanities projects for a wide variety of non-English languages, Quinn serves on the Global

[15] Eustagger-lite is available at http://ixa2.si.ehu.es/eustagger/.
[16] KoNLPy is available at http://konlpy.org/en/latest/, along with a tutorial for how to use it for text pre-processing at https://lovit.github.io/nlp/2019/01/22/trained_kor_lemmatizer/.
[17] The Classical Languages Toolkit is available at http://cltk.org/.
[18] At the same time, see this discussion about attempts to decompose characters into radicals as if the radicals were lemmas: https://www.quora.com/Does-the-Chinese-language-have-concepts-of-lemmatization-and-stemming-just-as-English-has.

Outlook::DH executive board and leads Stanford's Textile Makerspace. Quinn's publications include "What Ever Happened to Project Bamboo?" about the failure of a digital humanities cyberinfrastructure initiative, "Drupal for Humanists", and "Crescat Graffiti, Vita Excolatur: Confessions of the University of Chicago" about library graffiti.

References

Anderson, Deborah. *The Script Encoding Initiative, the Unicode Consortium, and the Character Encoding Process*. Signa nr. 6 April 2004. https://www.signographie.de/cms/upload/pdf/SIGNA_Anderson_SEI_1.0.pdf. Accessed 30 January 2020.

Ataman, Duygu, Matteo Negri, Marco Turchi and Marcello Federico. 'Linguistically Motivated Vocabulary Reduction for Neural Machine Translation from Turkish to English'. *Prague Bulletin of Mathematical Linguistics*, vol. 108, no. 1, 2017, pp. 331–42. DOI: https://doi.org/10.1515/pralin-2017-0031

Cro, Melinda A. and Sarah K. Kearns. 'Developing a Process-Oriented, Inclusive Pedagogy: At the Intersection of Digital Humanities, Second Language Acquisition, and New Literacies'. *Digital Humanities Quarterly*, vol. 14, no. 1, 2020. http://www.digitalhumanities.org/dhq/vol/14/1/000443/000443.html. Accessed 30 April 2020. DOI: https://doi.org/10.46430/phen0087

Dombrowski, Quinn, Tassie Gniady and David Kloster. *Introduction to Jupyter Notebooks*. The Programming Historian. 12 December 2019. https://programminghistorian.org/en/lessons/jupyter-notebooks. Accessed 30 January 2020.

Ezeiza, Nerea, Iñaki Alegria, Jose Maria Arriola, Ruben Urizar and Itziar Aduriz. 'Combining Stochastic and Rule-Based Methods for Disambiguation in Agglutinative Languages'. *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, vol. 1, 1998, pp. 380–4. DOI: https://doi.org/10.3115/980845.980909

Kim, Hyunjoong. 말뭉치를 이용한 한국어 용언 분석기 (Korean Lemmatizer), 22 January 2019. https://lovit.github.io/nlp/2019/01/22/trained_kor_lemmatizer/. Accessed 30 January 2020.

Mao, Lei. 'Byte Pair Encoding'. *Lei Mao's Log Book*, 2019. https://leimao.github.io/blog/Byte-Pair-Encoding/. Accessed 30 January 2020.

W3Techs. Distribution of character encodings among websites that use .ru. Updated 30 January 2020. https://w3techs.com/technologies/segmentation/tld-ru-/character_encoding. Accessed 30 January 2020.

How to cite this article: Dombrowski, Q 2020 Preparing Non-English Texts for Computational Analysis. *Modern Languages Open,* 2020(1): 45 pp. 1–9. DOI: https://doi.org/10.3828/mlo.v0i0.294

Published: 28 August 2020

Copyright: © 2020 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See http://creativecommons.org/licenses/by/4.0/.