A Corpus Processing and Analysis Pipeline for Quickref

Antoine Hacquard
EPITA
Research and Development Laboratory
Le Kremlin-Bicêtre, France
antoine.hacquard@lrde.epita.fr

Didier Verna
EPITA
Research and Development Laboratory
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

Quicklisp is a library manager working with your existing Common Lisp implementation to download and install around 2000 libraries, from a central archive. Quickref, an application itself written in Common Lisp, generates, automatically and by introspection, a technical documentation for every library in Quicklisp, and produces a website for this documentation.

In this paper, we present a corpus processing and analysis pipeline for Quickref. This pipeline consists of a set of natural language processing blocks allowing us to analyze Quicklisp libraries, based on natural language contents sources such as README files, docstrings, or symbol names. The ultimate purpose of this pipeline is the generation of a keyword index for Quickref, although other applications such as word clouds or topic analysis are also envisioned.

CCS CONCEPTS

• **Information systems** → **Information extraction**; **Retrieval effectiveness**; **Presentation of retrieval results**; • **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

Natural Language Processing, Indexing, Documentation

ACM Reference Format:
Antoine Hacquard and Didier Verna. 2021. A Corpus Processing and Analysis Pipeline for Quickref. In *Proceedings of the 14th European Lisp Symposium (ELS'21).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo.4714443

1 INTRODUCTION

Common Lisp [21] is a dialect of the Lisp family of programming languages. It was standardized in 1994 by the American National Standards Institute. It is an industrial-strength, multi-paradigm language. Languages in the Lisp family are among the very few to be homoiconic [8, 13], a property through which both introspection and intercession are achieved in a relatively homogeneous and simple way. The dynamic and highly introspective nature of Lisp makes it straightforward to extract information about the program structure and components, for example, for documentation purposes. Additionally, Common Lisp lets the programmer attach so-called "docstrings" (documentation strings) to almost all software components: variables, functions, classes, *etc.* When available, docstrings are a valuable source of information that can also be extracted very easily from an existing program.

1.1 The Paradox of Choice

In a somewhat paradoxical way, the technical strengths of the language bring drawbacks to its community of programmers [20, 24]. Lisp usually makes it so easy to "hack" things away that every Lisper ends up developing his or her own solution, inevitably leading to a *paradox of choice.* The result is a plethora of solutions for every single problem that every single programmer faces. Most of the time, these solutions work, but they are either half-baked or targeted to the author's specific needs and thus not general enough. Furthermore, it is difficult to assert their quality, and they are usually not (well) documented.

In this context, an important tool, community-wise, is Quicklisp. Quicklisp is both a central repository for Common Lisp libraries (there are currently around 2000 of them) and a programmatic interface for it. With Quicklisp, downloading, installing, compiling and loading a specific package on your machine (dependencies included) essentially becomes a one-liner. What Quicklisp doesn't solve, however, is the documentation problem.

1.2 Quickref

Quickref [22, 23] is a global documentation project for the Common Lisp ecosystem. It generates reference manuals for libraries available in Quicklisp automatically. Quickref is non-intrusive, in the sense that software developers do not have anything to do to get their libraries documented by the system: mere availability in Quicklisp is the only requirement.

Quickref works by introspecting libraries, and generating corresponding documentation in Texinfo format. The Texinfo files may in turn be converted into human-readable documentation, for example in PDF or HTML. Quickref may be used to create a local website documenting your current, partial, working environment, but it is also used in production, to maintain a global public website of technical reference manuals for all Quicklisp libraries. The site is kept in sync with Quicklisp.

1.3 Library Access

In order to provide access to the two thousand or so reference manuals available on the website, Quickref provides two indexes: a library index and an author index. The former is most likely to be used when looking for a library in particular, while the latter is probably only useful for people wanting the check out the generated documentation for their own work.

Suppose however that someone is looking for some functionality, without any prior idea or knowledge about which library may be appropriate. Quickref, as it is right now, is impractical for such a mining task, hence the idea of enriching it with a keyword index, a word cloud, *etc.* In order to generate such things automatically, it is necessary to process and analyze each library's *corpus*, that is, the bits of textual information providing some description of functionality (README files, docstrings, sometimes even symbol names, *etc.*). Fortunately for us, Declt, the reference manual generator on which Quickref is based, makes it very easy to access the corpuses in question. The purpose of this paper is to describe the natural language processing pipeline that we are currently building into Quickref to analyze the extracted corpuses, and ultimately provide library access by functionality.

Given the universal availability of very efficient internet search engines these days, one may wonder whether an indexing project specific to Quickref is really needed or pertinent. The following remarks answer that question.

First of all, a general search engine doesn't know about such or such library's availability in Quicklisp. On the other hand, a local index will necessarily point to readily-available libraries only. Next, and as opposed to search engines considering plenty of, and indiscriminate information sources, our indexing process is based on each library's documentation only. Therefore, it will have a natural tendency to favor well documented ones, which can be an important factor, when choosing which tool to use in your own project.

Finally, and beyond providing new kinds of indexes, other applications of this project could be envisioned later on, such as topic analysis, distribution, and visualization (a topography of the centers of interest in the Lisp community, of sorts).

1.4 Pipeline Overview

Figure 1 depicts the pipeline used to process and analyze the corpuses extracted from each library by Declt.

(1) Each corpus is first *tokenized*, that is, split into chunks which usually (but not necessarily) correspond to words. The tokens are then *tagged*, meaning that they are associated with their syntactical class (noun, verb, *etc.*). After this stage, we are able to filter specific token classes (*e.g.* retain only nouns, verbs, *etc.*).
(2) Next, the retained tokens are *stemmed*, meaning that their lexical root is extracted, and used to attempt matching with a canonical form found in a dictionary. This process is called *lemmatization.* After this stage, only the canonicalized known lemmas (*i.e.*, found in said dictionary), are retained.
(3) A TF-IDF (Term Frequency / Inverse Document Frequency) value is computed for every such lemma. This value is a statistical indication of how relevant each lemma is to the corresponding library. Only the most pertinent ones are kept around (the exact number of such retained lemmas may vary).
(4) Finally, the (possibly intersecting) sets of most pertinent keywords describing each library are aggregated in order to produce the desired output (keyword index, word cloud, *etc.*).

It is worth mentioning right away that in this pipeline, two out of four blocks (the first two) are *pre-processing* steps, devoted to sanitizing the corpuses, while only stages three and four actually perform the job of information processing. The importance of pre-processing in this pipeline is due to TF-IDF working on syntactic tokens only, without any semantic information. For example, without pre-processing, tokens such as "test", "tests", and "testing" would be treated independently, as if they meant different things.

At the time of this writing, the first three blocks in this pipeline are fully operational. Keyword aggregation, on the other hand, is a difficult problem, and the aggregator block is still subject to experimentation. Also, note that we intend, at a later time, to release the code of each block as independent, open-source libraries.

The remainder of this paper is organized as follows. Sections 2 to 5 provide a more in-depth description and discussion of the tokenizer / PoS-Tagger, stemmer / lemmatizer, and TF-IDF blocks respectively. Section 6 describes the challenges posed by the keyword index generation problem, the experiments already conducted, and some possible ideas for further experimentation.

2 POS-TAGGING

PoS-Tagging (for "Part-of-Speech" tagging) is a technique allowing to determine the syntactic class of words, that is, whether they are common nouns, verbs, articles, *etc.* The syntactic classes of words may be important information to perform semantic analysis of a corpus for different reasons. For example, some categories of words, like determinants, convey very little or no useful meaning at all, so we want to filter them out early, rather than carrying them around until the TF-IDF block makes the same decision (although for a different reason: they appear frequently, but everywhere). Also, in the aim of generating a keyword index, it may be interesting to experiment with different sets of retained information, such as only nouns, nouns and verbs, *etc.*

2.1 Implementation

There are many ways to implement a PoS-Tagger, notably with HMMs (Hidden Markov Models), unsupervised learning, or machine learning [9]. In the Common Lisp ecosystem, we are aware of one PoS-Tagger library, namely "Tagger" [5], written by Xerox in 1990, which uses HMMs.

HMMs are statistical Markov Models used to learn an *unknown* Markov Process with hidden states, by observing another process, known this time, and depending on it. HMMs are widely used in PoS-Tagging to disambiguate syntactic classification. The biggest problem of PoS-Tagging is that a word can have several syntactic classes associated with it, depending on the context. For example, the word "can" may be either a verb, or a noun (as in "soda can"). Using HMMs, a PoS-Tagger first learns the probability of a certain sequence of syntactic classes occurring. Then, it disambiguates unknown words by using the syntactic class sequence with the highest probability.

Suppose for example that after an article such as "the", the class probabilities for the next word are 40% `noun`, 30% `adjective`, and 20% `number`. When seeing "The can", a PoS-Tagger will thus correctly classify "can" as a noun.

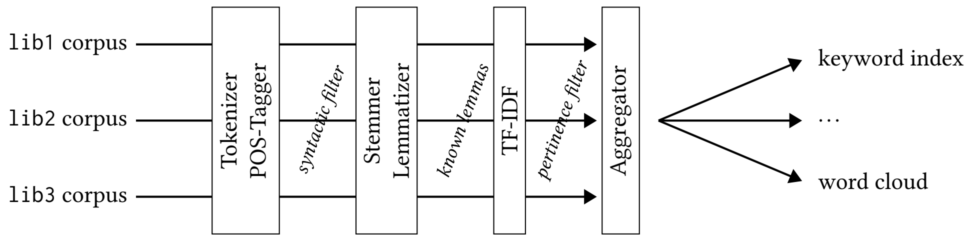

Figure 1: Keyword index generation pipeline

Because HMM-based PoS-Tagging needs a word's surrounding context in order to decide on its syntactical class, it must appear very early in the pipeline, namely, before such contextual information is removed. On the other hand, the input of the tagger needs to have been tokenized already. Therefore, the tokenization and tagging steps are tightly coupled, which is why they appear as a single block in our pipeline.

The aforementioned Tagger library happens to offer a powerful and highly customizable tokenizer, and a PoS-Tagger linked with it. The tokenizer uses an automaton to parse sentences, and can be customized with rational expressions. Our two biggest customizations on the tokenizer were to accept dashes in tokens rather than considering them as separators (otherwise, words like "command-line" would have been split), and to add a rational expression to recognize URLs as unique tokens (many URLs exist in our corpuses, even in plain text documentation).

The Tagger library poses a problem however: it accepts ASCII characters only. In the majority of the cases, this is not so much of a problem because the natural language in use is almost exclusively English, and a purely ASCII text encoded in Unicode remains readable as-is. On the other hand, some libraries do have special characters in their README files, breaking the tagger. A good example of this, is the `April` [18] library, which compiles a subset of the APL language. Many special characters in there are APL tokens, which are not ASCII.

In order to solve this problem, we pre-process all of our documents with "Free Recode", an open source tool to transliterate files between many encodings. Non-ASCII characters are replaced with interrogation marks. This side-effect actually has little or no impact on our pipeline at all, because our current corpus contains only English documentation. Most of it is already plain ASCII, and the few non-ASCII characters we found were either fancy "prettification" of README files (*e.g.* smileys), or in code snippets.

2.2 Tests and Results

In order to get an early feedback on the behavior of this pre-processing stage, we ran our complete pipeline in different PoS-Tagger modes. Sample results are presented in Table 1. That table displays the 10 keywords appearing the most frequently after the TF-IDF block (all libraries included). The numbers in parentheses are the number of libraries associated with each keyword. In all cases, the Tagger library's tokenizer is used. When tagging is turned off, there is no filtering on the syntactic classes of the tokens. Otherwise, the table presents results when only common nouns, or a combination of common nouns and verbs are retained.

All tokens	Only nouns	Nouns and Verbs
lisp (110)	library (175)	test (110)
test (63)	file (150)	file (108)
message (51)	function (138)	license (107)
common-lisp (51)	license (133)	library (107)
name (49)	value (117)	function (92)
file (48)	document (117)	name (81)
value (47)	package (114)	value (81)
stream (46)	name (114)	package (78)
function (46)	test (102)	stream (69)
server (45)	project (101)	load (69)

Table 1: Top 10 keywords w/ different syntactic filters

We observe that even without tagging, we don't see "noisy" words such as articles appearing in the top 10 list. That is because at the end of the pipeline, the TF-IDF pass will detect that such words, being frequent basically everywhere, are in fact not specific to any library in particular. On the other hand, the PoS-Tagger would help filtering those words earlier in the pipeline. It is also apparent that PoS-Tagger helps filtering out uninteresting tokens such as "lisp" or "common-lisp". Indeed, these end up being filtered out as either proper nouns (as in "Lisp is a …"), or adjectives (as in "a Lisp library").

Whether to keep verbs around, or only common nouns, remains an open question. Verbs may contain useful information for describing what a library does. For example, it is likely that a library for unit testing will make frequent use of the word "test" both as a noun, and as a verb. If we keep both around, the final weight of "test" as a unique lemma will increase (which *is* a good thing in that particular case, and is in fact visible in Table 1). This will also happen every time a verb and a noun are slightly different, but are lemmatized identically. On the other hand, many verbs are also uninteresting ("be", "get", "come", *etc.*), and it is impossible to know in advance whether their distribution across all libraries would be such that the TF-IDF block would filter them out. Finally, there are also problematic cases in which a verb and a noun convey different meanings, which would hinder the accuracy of our results. One possible solution to this problem would be to tag nouns and verbs in order to keep them as separate entities, but as mentioned before, there are also cases where keeping them separate is undesirable.

3 STEMMING

Stemming is the process of reducing a word to its root, or canonical form in the linguistic sense, notably by removing prefixes or suffixes.

No stemmer	Porter	Snowball	Lancaster
node	node	node	nam
server	elem	elem	parsable
test	src	src	node
stream	parse	server	src
template	stream	parse	byte
event	server	stream	stream
trivial	trivial	see	trivia
x	byte	trivial	el
connection	test	test	x
image	x	byte	test

Table 2: Stemmer-dependent results for the final index

Although stemming does not constitute a block in our pipeline *per se*, it still is an important part of the process, for reasons that will become apparent in the next section.

Because a stemmer removes everything but the linguistic root of a word, the resulting "stem" may not be a complete word at all. This is a potential problem for us, because in the end, we want an index composed of actually existing words, so the stems themselves can't always be used directly.

3.1 Implementation

Many stemming algorithms exist, and they are usually quick and straightforward to implement. The two most popular approaches are based, either on rule systems, or on training of stochastic algorithms [7]. The rule-based approach offers a better trade-off between simplicity of implementation and quality of the produced stems, so this is the approach we favor.

Figure 2 illustrates a typical use-case of a rule-based stemmer. There are usually two categories of rules: transformation rules and deletions rules. A transformation rule transforms a prefix (respectively, a suffix) into a simpler version. A deletion rule deletes the prefix (respectively, the suffix).

Three notable suffix stemmers exist in the literature: Porter [15], Snowball [16] (*a.k.a.* Porter 2), and Lancaster [14]. These stemmers are well suited to process English, as most of the word variations occur at their end in this language. We implemented the three of them in Common Lisp, and we used NLTK [3] as a reference point for debugging our implementations. NLTK is the most well known, and *de facto* standard Python library for NLP (Natural Language Processing), and incorporates a large number of stemmers. Note that we are aware of only one pre-existing Common Lisp implementation of a stemmer [6], a Porter stemmer, more specifically. We still decided to write our own because it is rather straightforward, and also because the NLTK implementation, which we want to follow, sometimes departs from the original specification in ways that would have been difficult to implement in the existing Common Lisp implementation, which is not very flexible.

3.2 Tests and Results

In order to get an early feedback on the behavior of stemming, we ran our complete pipeline which each of them, and also without stemming at all, that is, using the output of the PoS-Tagger directly. Sample results are presented in Table 2. We notice a global improvement of the final index when stemming is used. Indeed, interesting words (such as "parse") are brought up, while less interesting ones (such as "x") are brought down. We also notice that the results with the Lancaster stemmer are not so good: many final words are in fact not actual words. This is due to the fact that Lancaster is a "strong" stemmer: it has a tendency to over-stem words, which leads to the same root for words and typos. The Snowball stemmer is considered to give the best results, as it is the only one which manages to bring down "x" to *not* be in the first ten words.

4 LEMMATIZATION

Besides stemming, the other classical approach to word normalization in the literature is lemmatization, which consists in using the dictionary form of a word as its canonical representation (instead of a stem). The main advantage of this approach is that in the aim of building a word index, the output of a lemmatizer can be used directly, as opposed to that of a stemmer which requires reconstructing a word afterwards.

4.1 Implementation

Lemmatization can be implemented in many different ways. Approaches range from rule-based systems (similar to stemmers, but with more complicated rules), dictionary look-up, machine-learning, *etc.* As our bibliographical research didn't reveal anything satisfactory in terms of Lisp implementation of a lemmatizer (either not in Quicklisp or part of a larger library), we decided to implement our own. The approach we chose is that of dictionary look-up, as described in [10]; a solution both elegant and easy to implement. In short, a word is compared with all words in a dictionary of "lemmas", and the closest one (according to a so-called "edit distance") is chosen as its canonical form. A pre-processing step consisting of stemming the word before measuring its edit distance is discussed in the paper, and shown to give better results (hence the importance of stemming anyway). The Common Lisp library `mk-string-metrics` offers a set of built-in edit distances. We use this library to implement our lemmatizer.

We conducted a set of experiments in order to decide on the best combination of stemming algorithms (among the 3 described in the previous section), edit distances (we choose to only test the 5 available in `mk-string-metrics` but there are plenty of others in the literature [12], [4], [2], [19]), and dictionaries. The following sections report on those experiments.

4.2 Stemmer / Edit Distance Selection

In order to decide on which stemmer algorithm and which edit distance to use, we tested the possible combinations and counted the number of correct lemmas generated by the lemmatizer. The ground truth (*i.e.* the correct lemmas for each word) was simply found on the internet, where a lot of resources related to lemmatization exist for verifying the correctness of an implementation[1].

Table 3 shows the obtained results. These results confirm one of the paper's claims, which is that the use of a stemmer has a huge impact on the quality of the results. The other noticeable thing is that the Lancaster stemmer performs quite poorly. This, again, can be explained by the fact that Lancaster tends to produce very

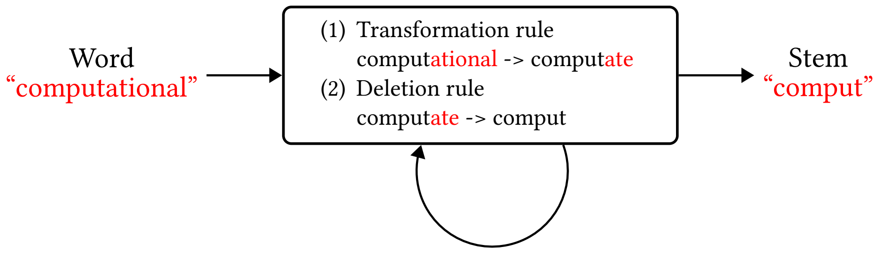

Figure 2: Ruled-based stemming

	Distance				
Stemmer	Jaccard	Jaro-Winkler	Damerau-Levenshtein	Levenshtein	Overlap
None	428	659	655	656	25
Porter	840	934	970	970	132
Snowball	842	934	970	970	132
Lancaster	483	612	590	591	32

Table 3: Number of correct lemmas on a list of 1226 words

16.000 lemmas	30.000 lemmas	33.249 lemmas
clause (54)	clause (52)	library (167)
client (40)	server (38)	file (145)
server (39)	client (35)	function (134)
project (37)	hotel (34)	license (124)
value (37)	project (34)	document (113)
message (35)	value (33)	value (110)
node (34)	node (31)	name (105)
user (33)	message (29)	package (103)
test (30)	begin (29)	test (100)
begin (30)	aside (27)	project (96)

Table 4: Dictionary-dependent results for the final index

short stems, which skews the edit distance computation. Finally, we can see that the best results are obtained with Porter or Snowball stemmers, and with Levenshtein or Damereau-Levenshtein edit distances. As Snowball is an improvement over Porter, and Damereau-Levenshtein over Levenshtein, it is only logical that these four have approximately the same results. At that point, we decided to retain the Snowball stemmer, as it also performed more efficiently, time-wise, and the Levenshtein distance, because it is slightly faster than the Damereau-Levenshtein one.

4.3 Dictionary Selection

For dictionary selection, we started by experimenting with two dictionaries of 16.000 and 30.000 lemmas respectively, found on the internet (unfortunately, we lost track of the source of these dictionaries in the process, but we *will* publish them later, along with the code).

Sample results are presented in Table 4 (the third dictionary / column will be described in a few paragraphs). The indexes generated with those dictionaries have a big flaw: they contain words that are not in the base corpus. The most obvious example of this is the word "hotel" occurring at the fifth position in the second index. These words appear somewhat "magically" for a conjunction of two events: they exist in the dictionary but not in the corpus, and we're trying to lemmatize a word that is (obviously) in the corpus, but not in the dictionary. Because every word in the corpus *needs* to be matched to a word in the dictionary, such words will be lemmatized weirdly.

More specifically, lemmas are chosen by optimizing the edit distance (minimizing or maximizing it, depending on the actual distance in use), which is a *continuous* measure. This means that while a lemma will *always* be found, the edit distance may still be bad. In other words, there are times when even the best solution is a bad one.

4.3.1 White-Listing. A natural solution to this problem is to use the dictionary as some sort of "white-list", by imposing a threshold on the computed edit distance. Whenever a normalized edit distance is found to be lower than the selected threshold, the confidence in the lemmatization process is considered too low, and the word discarded from the subsequent TF-IDF statistic. Using such a threshold is a convenient way to make a distinction between words which do have a lemma in the dictionary, and words which don't (hence, words which we don't want to keep around). After some experimentation (mostly, looking at the results), we decided that a threshold of 0.8 is appropriate for making a decision.

4.3.2 Custom Dictionary. An even better solution to this problem would be to make sure that the dictionary we use does not contain words absent from our corpuses in the first place. This leads to the idea of generating a *custom* dictionary, from the lemmatization of the words present in our corpuses directly. Of course, creating such a dictionary leads to a bootstrapping problem, as lemmatizing our corpus would require using the dictionary we are trying to create. Thus, we need an external lemmatizer.

Here, again, we used the one from NLTK (in fact, we also tried the lemmatizer from the Stanford NLP library [11], written in Scala

No lemmatization	Lemmatization
library (96)	library (175)
file (82)	file (150)
function (79)	function (138)
data (74)	license (133)
value (71)	value (117)
license (70)	document (117)
documentation (67)	package (114)
test (65)	name (114)
name (64)	test (102)
body (57)	project (101)

Table 5: Indexes obtained with and without lemmatization

out of curiosity). The generated dictionary contains approximately 33.500 lemmas. Note that in theory, we should rebuild it every time Quicklisp is updated. Whether this is a critical issue remains to be seen however. Indeed, Quicklisp is already quite large, so the probability that an update induces a very important change in the corpus is likely to be low. On the other hand, an outdated custom dictionary may start to miss words, or contain irrelevant ones again, so it is still important to continue using the aforementioned threshold-based white-listing step.

Finally, note that with this custom dictionary, we are certain to only get lemmas existing in our corpuses, but we are not completely sure that the "technical jargon", frequent in our community's specialized version of English is fully recognized by NLTK. It is difficult to evaluate the risk of an unknown technical word being weirdly lemmatized by NLTK, but we're hopeful that if it happens at all, it remains marginal. NLTK uses the Wordnet database, which is very large, and also encodes relations between words (such as singular/plural, synonyms, *etc.*). A possible path to get more insight into this problem could be to evaluate the behavior of NLTK on the Common Lisp Hyperspec's glossary (which is likely to be a quite complete reference for technical jargon), and maybe adjust the reference dictionary accordingly. Another one would be to properly recognize code pieces from markup information (see Section 8). Finally, it would be highly beneficial to keep even non-Lisp jargon around. Pseudo-words such as "cmdline", acronyms such as "GUI", *etc.*, behave just like regular words in our communities, and should probably be treated as such. How to collect them into our custom dictionary is yet another problem.

The third column in Table 4 shows the top 10 keywords obtained with this custom dictionary, and confirms that the results are better. For example, irrelevant words such as "begin" or "aside" are gone, even though our dictionary contains more lemmas in total than the two other ones.

4.4 Final tests and results

In order to get an early feedback on the behavior of this pre-processing stage, we ran our complete pipeline with and without lemmatization. Recall that without lemmatization, it is the output of the PoS-Tagger which is processed by the TF-IDF block directly. Sample results are presented in Figure 5. The question of whether lemmatization is useful, and under which precise conditions remains open. In general, lemmatization is expected to be useful because it allows to treat variations on a single keyword together. On the other hand, a lemmatized keyword may not be the most informative, and we believe that this is exactly what happens with "documentation" and "document" in Figure 5. Assuming that documentation libraries (such as Quickref and Declt) are those which bring the keyword "documentation" up, it is unfortunate that in the lemmatized case, this keyword is transformed into "document" which, in fact, is less informative. This problem suggests that using an ad-hoc, carefully tuned dictionary may turn out to be important.

5 TF-IDF

Even though most of the delicate work actually happens during the pre-processing phase, the heart of our pipeline consists in extracting meaningful words from our corpus. By "meaningful", we mean words which convey the most relevant and decisive information. For this task, we use the TF-IDF statistic [17].

TF-IDF is a measure that aims at reflecting the importance of a word in a document. It uses two parameters to operate: the frequency of the word in the document and the number of documents containing this word. The main idea behind this approach is that a word both frequent in a document and frequent in all documents is not very specific to the document in question, and thus, is not a good descriptor for this document. On the other hand, a word that is very frequent in one document, but which appears nowhere else, brings a great amount of information on the document in question, and can thus be used as a keyword representing it.

In the Quickref context, a "document" corresponds to the corpus of text extracted by Declt from one specific library. TF-IDF is run on each library, for which the best x words are retained, x being an adjustable parameter.

5.1 Tests and Results

An important question, before running a TF-IDF on each library's corpus, is to decide on what we actually use as a corpus for each library. As mentioned before, README files and docstrings are a natural choice, but we can also think of using symbol names (of functions, variables, *etc.*) as the code is also usually explicit about what it does. We could also use ASDF's system descriptions, when provided, but we haven't tried that yet. More specifically, we ran our pipeline on the following corpus variations.

(1) README files only.
(2) README files, plus docstrings for all exported functionality (public API).
(3) The above, plus the symbol names for all exported functionality. The rationale is that carefully chosen API names may be indicative of the library's purpose.
(4) The above, plus docstrings for the library's internals (so, essentially all docstrings available).
(5) The above, plus the library's internals symbol names (so, essentially all symbols).

Sample results are presented in Table 6. As more or less expected, it is probably not a good idea to add the documentation of a library's internals in the corpuses, as the text found there most probably deals more with the implementation of the library's functionality, than the functionality itself. This is visible, for example with the appearance of keywords such as "string", "vector", or "class" in the

	README	**+ Docstrings**	**+ Symbols**
Public API	library (175)	file (148)	stream (123)
	file (150)	string (142)	file (117)
	function (138)	value (139)	value (105)
	license (133)	stream (132)	string (105)
	value (117)	object (125)	name (105)
	document (117)	name (118)	user (105)
	package (114)	license (109)	object (95)
	name (114)	function (108)	test (90)
	test (102)	type (106)	type (90)
	project (101)	test (104)	error (88)
+ Internals	library (175)	string (145)	string (140)
	file (150)	stream (138)	file (136)
	function (138)	file (137)	stream (133)
	license (133)	value (118)	object (110)
	value (117)	object (114)	value (109)
	document (117)	name (102)	vector (101)
	package (114)	class (100)	class (96)
	name (114)	test (97)	test (94)
	test (102)	license (96)	function (92)
	project (101)	function (95)	message (91)

Table 6: Results for different corpus variations

top 10, which are likely to be related to typing information known statically, and advertised as such.

Even when restricting ourselves to the public API's corpus, including docstrings and / or symbol names doesn't seem to add much to the pertinence of the results. Even public docstrings are in fact likely to contain static typing information (such as "string"), function parameters descriptions (such as "object"), *etc.* In fact, we have ultimately no control whatsoever on the type, quality, or quantity of documentation (if any) provided by the developers, which makes keyword extraction a very hard problem.

6 AGGREGATION

An even harder problem is how to aggregate an appropriate selection of keywords coming from different libraries (probably with some overlap), into a sufficiently descriptive and pertinent index. The difficulty here comes from the fact that we would like 100% library coverage (we want every Quicklisp library to be pointed to by at least one keyword) but we also want a reasonably sized final index. How to achieve this goal is still mostly unanswered, but we have already conducted some experiments, reported below, and we also have some ideas that yet remain to be tested.

6.1 Histogram-Based Selection

The first approach we have experimented with is based on the cross-library keyword appearance histogram. For each of the retained x keywords from every library, we count the number of libraries it appears in, and we sort them by decreasing frequency. We then select the minimum number of keywords required to reach a 100% coverage.

This process is very simple to implement, and as a side-effect, can also be the base for generating a word cloud. Indeed, if a keyword is representative of many libraries, it probably means that the corresponding topic is subject to a lot of activity. On the other hand, this approach also poses accuracy problems, and makes it hard to adjust the relevant parameters properly (this is where a choice on the value of x becomes crucial). More specifically, because we want every library to be indexed, a trade off is to be made between the number of keywords retained per library (hence, accuracy), and the size of the final index.

If, for example, we keep only one keyword per library, this keyword will indeed be very descriptive of that particular library, and so is less likely to apply to many of them. Consequently, it is very probable that the final index will be very large (at worst, one different keyword for every single library, that is, approximately 2.000).

If, on the other hand, we keep a large number of keywords for every library, there is more likelihood that the retained keywords will overlap from one library to another, letting us reach a 100% coverage faster. However, we also risk retaining keywords that are not so relevant.

Figure 3 contains plots of the library coverage (in percentage) as a function of the final index size, for different values of x, that is, when retaining different numbers of keywords per library. These plots confirm what intuition tells. When $x = 50$ for example, a 100% library coverage is reached with a final index of 200 keywords, but those keywords are likely to *not* be so specific. When $x = 5$, on the other hand, the final index will require more than 3.000 words, all probably quite relevant.

As mentioned before, because of the inherent structure of the histogram we use, the top 1 keyword will have many libraries associated with it, the next one slightly fewer, and so on. This is important, and problematic, for two reasons.

(1) When a user searches a library for a specific use, a keyword leading to a hundred different choices is likely to be of little help.
(2) The number of libraries associated with a keyword is not the same for all keywords, which makes the final index somewhat heterogeneous.

This is why we also plan to investigate other approaches.

6.2 Other Potential Solutions

A first alternative approach could be to sort the output of TF-IDF not by decreasing frequency, but by a pertinence factor of some sort (doing in some sense a meta-TF-IDF on top of the original one), and keep enough of the top ones to reach a 100% coverage. The pertinence factor in question could be the inverse of average ranking of a keyword in each library's top list, a normalized sum of all TF-IDF values, or any other measure yet to be thought of.

Yet another possibility would be to take a completely opposite approach, and start from the fact that in order to be usable, a keyword shouldn't point to more than, say, $n = 10$ libraries. We could then arrange to select all such keywords until we reach a 100% coverage (probably adjusting n to get a reasonably sized index in the process).

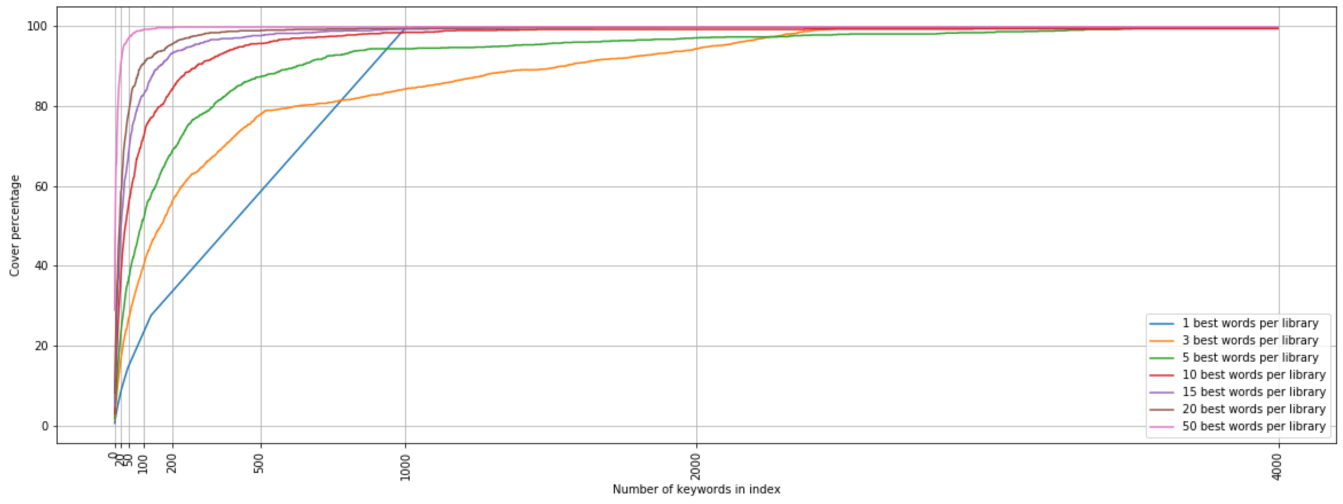

Figure 3: Library coverage *vs.* final index size, for different values of x

7 CONCLUSION

In this paper, we presented a natural language processing pipeline for Quickref, allowing us to analyze corpus extracted from Quicklisp libraries docstrings, README files, or symbol names. This pipeline is relatively lightweight, as it amounts to no more than 2000 lines of code.

As part of this process, we have used an existing PoS-Tagging library, and we have developed our own native Common Lisp implementations of stemming and lemmatization algorithms. As of this writing, the code is not in production yet, but we plan to cleanup, package, and release our stemmers and lemmatizer as standalone libraries in short term.

The complete pipeline, including the histogram-based aggregation approach, is currently integrated in an experimental version of Quickref, but other solutions remain to be tested before putting the whole thing in production.

8 PERSPECTIVES

Apart from the aggregation problem, some other plans for future work are worth mentioning.

Our pipeline is currently unaware of any markup used in README files notably (Markdown, HTML, *etc.*). A number of specific tweaks are in place in order to remove markup tags from the corpus (for example, by recognizing and filtering URLs out during the tokenization phase). Also, in the case of frequently used markup formats (such as Markdown), the syntactic “noise” produced by the tags is likely to be filtered out as non-pertinent by the TF-IDF block, precisely because of its frequency in many libraries. Yet, it would be better to be aware of the markup formats in use, and use that information during the tagging process. The first advantage that comes to mind is to be able to properly differentiate natural language parts from code samples, in order to select what we want to keep around for TF-IDF (see Section 5.1). Correctly identifying markup tags could also be useful to spot inline code excerpts (or just words) embedded in natural language paragraphs, and give them special treatment (for instance, considering them as “technical jargon”; see Section 4.3.2). More generally, it could be interesting to think about the kind of information that other tags, such as bold or italics, provide. For instance, bold or italics may be an incentive to give more weight to the targeted textual part. Even more generally, potentially useful information can sometimes be extracted from pure, tagless, text. For example, it is customary to render Lisp references (function parameters, variables, *etc.*) in uppercase in plain docstrings.

Previously, we mentioned that we use a suffix stemmer because that is where most of the variations occur in English. We could not find any prefix stemmer in the literature, and we currently don't know if that would be worth looking for, even for English, and perhaps in combination with the current suffix one.

As far as dictionaries are concerned, we mentioned that the best results were obtained by creating our own custom dictionary with the help of an external lemmatizer. Another possibility would be to start from an existing dictionary, but keep track of missing words, and create only a custom *addition* to the original dictionary with those words. Even if we gain a little in terms of dictionary bootstrapping time, it is not very likely that this solution would get us anything more in terms of pertinence, notably because existing dictionaries are still likely to contain a lot of words that are uninteresting for us, or that actually never occur in our corpus.

Finally, one final question that could arise eventually is that of the actual language in use. Currently, we assume English (which is unlikely to pose any problem with Quicklisp), but if we even want to handle other languages, the problem will become more complicated. In particular, our current PoS-Tagger will not be usable anymore.

REFERENCES

[1] Lemma lists. https://lexically.net/wordsmith/support/lemma_lists.html. Accessed: 2021-03-06.

[2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48, September 2000.

doi: 10.1109/SPIRE.2000.878178.

[3] Steven Bird, Edward Loper, and Ewan Klein. Natural language processing with python. https://www.nltk.org/book/, 2009.

[4] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the 2003 International Conference on Information Integration on the Web*, IIWEB'03, pages 73–78. AAAI Press, 2003. doi: 10.5555/3104278.3104293.

[5] Doug Cutting and Jan Pedersen. The Xerox part-of-speech tagger version 1.2. https://github.com/g000001/tagger, 1993.

[6] Steven M. Haflich. The Porter stemming algorithm, a Common Lisp implementation. https://github.com/varjagg/porter-stemmer, 2002.

[7] Anjali Jivani. A comparative study of stemming algorithms. *International Journal on Computer Technology and Applications*, 2:1930–1938, November 2011.

[8] Alan C. Kay. *The Reactive Engine*. PhD thesis, University of Hamburg, 1969.

[9] Deepika Kumawat and Vinesh Jain. Pos tagging approaches: A comparison. *International Journal of Computer Applications*, 118(6):32–38, May 2015. ISSN 09758887. doi: 10.5120/20752-3148.

[10] Dimitrios P. Lyras, Kyriakos N. Sgarbas, and Nikolaos D. Fakotakis. Using the levenshtein edit distance for automatic lemmatization: A case study for modern greek and english. In *19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007)*, volume 2, page 428–435, Oct 2007. doi: 10.1109/ICTAI.2007.41.

[11] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, page 55–60. Association for Computational Linguistics, 2014. doi: 10.3115/v1/P14-5010.

[12] Andrew McCallum, Kedar Bellare, and Fernando Pereira. A conditional random field for discriminatively-trained finite-state string edit distance. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence. UAI2005*, Jul 2012.

[13] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3:214–220, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367223.

[14] Chris D. Paice. Another stemmer. *ACM SIGIR Forum*, 24(3):56–61, Nov 1990. ISSN 0163-5840. doi: 10.1145/101306.101310.

[15] M. F. Porter. An algorithm for suffix stripping. *Program: Electronic Library and Information Systems*, 14(3):130–137, 1980. doi: 10.1108/eb046814.

[16] M. F. Porter and Richard Boulton. Snowball stemmer. 2001.

[17] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, Jan 1988. ISSN 03064573. doi: 10.1016/0306-4573(88)90021-0.

[18] Andrew Sengul. April: Array programming re-imagined in lisp. https://github.com/phantomics/april, 2019.

[19] Syeda ShabnamHasan, Fareal Ahmed, and Rosina Surovi Khan. Approximate string matching algorithms: A brief survey and comparison. *International Journal of Computer Applications*, 120(8):26–31, Jun 2015. ISSN 09758887. doi: 10.5120/21247-4048.

[20] Mark Tarver. The bipolar Lisp programmer. http://marktarver.com/bipolar.html, 2007.

[21] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[22] Didier Verna. Quickref: Common Lisp reference documentation as a stress test for Texinfo. In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 40, pages 119–125. TEX Users Group, September 2019.

[23] Didier Verna. Parallelizing Quickref. In *12th European Lisp Symposium*, pages 89–96, Genova, Italy, April 2019. ISBN 9782955747438. doi: 10.5281/zenodo.2632534.

[24] Rudolf Winestock. The Lisp curse. http://winestockwebdesign.com/Essays/Lisp_Curse.html, April 2011.