CL-NLP — a Natural Language Processing library for Common Lisp

Vsevolod Domkin

vseloved@gmail.com

Abstract

CL-NLP is the new Common Lisp Natural Language Processing library the development of which has started in 2013. The purpose of the library is to assemble a comprehensive suite of NLP algorithms, models and adapters to popular NLP resources for Common Lisp. Similar projects in other languages include the Python Natural Language Toolkit NLTK [NLTK], which is the most popular starting point for educational work and academic research, Stanford CoreNLP [CoreNLP] and Apache OpenNLP [OpenNLP] libraries.

The motivations for its creation include the following:

- Lisp is very well suited for NLP projects, providing broad support for statistical calculations, symbolic computation, as well as string and tree mainpulation;
- unfortunately, a lot of work was done in the NLP area in Common Lisp before the advent of open-source, and its artifacts are scattered across various libraries, university Internet web-sites and books, not gathered under one roof, curated and supported. The idea behind CL-NLP is to provide a central repository for such artifacts in the future;
- the existing open source Common Lisp NLP tools are insufficient.

This article presents an overview of the current state of CL-NLP, a discussion of its implementation and a plan for its further development.

Keywords: Natural Language Processing, Programming Environments, Software Architectures, Reusable Software,Language Constructs and Features

1. Overview

1.1. Previous work

There is an existing NLP Lisp library — cl-langutils [langutils], which was considered as a candidate to serve as a base for this effort, but its original authors had abandoned it, and it has not seen active development for a long period of time with only occasional bugfixes. The main concern with cl-langutils is that it doesn't provide a modular foundation suitable for supporting many alternative ways to solving the same tasks which is required to assemble the suite of algorithms that should become CL-NLP.

Apart from langutils, other repositories of NLP-related Common Lisp code include:

- code from “Natural Language Processing in Lisp” book [NLPinLisp]
- code from “Natural Language Understanding” book [NLU]

- code from “Paradigms of Artificial Intelligence Programming” book [PAIP]
- code from “Artificial Intelligence Programming” book [AIProg]
- CMU AI repository [CMUAI]
- Lexiparse project [Lexiparse]
- Sparser project [sparser]
- CL-EARLY-PARSER project [CLEarly]
- Basic-English-Grammar project [BasicEngGrammar]
- Various Wordnet interfaces, including cl-wordnet [CLWordnet] and cffi-wordnet [cffiWordnet]
- Soundex project [Soundex]

1.2. Library design

The most important design goal for CL-NLP is to provide a modular extensible foundation to accumulate over time various NLP algorithms and models. To achieve it the library uses several layers of modularization facilities: systems, packages and CLOS generic functions.

At the top level the library is provided as two ASDF systems:

- `cl-nlp` implements the core functionality;
- `cl-nlp-contrib` provides various adapters to external systems, that have additional dependencies not essential to the core library.

At the namespacing level the library is split into packages of three types:

- basic packages that provide the foundational data structures and utilities to serve as the common “language” of CL-NLP – these include: `nlp.core`, `nlp.util`, `nlp.test-util`, and `nlp.corpora`;
- functional packages in `cl-nlp` system that implement a broad range of functionality in one of the areas of the libraries scope: `nlp.syntax`, `nlp.generation`, `nlp.phonetics` etc;
- functional packages in `cl-nlp-contrib` system that implement specific adapters to external NLP resources: `nlp.contrib.wordnet`, `nlp.contrib.ms-ngrams`;
- `nlp-user` package which collects all the public symbols from the other packages and provides additional facilities to enhance the usability of interactive development with CL-NLP.

Common Lisp's lack of hierarchical packages is often regarded as a shortcoming of the language. Yet, package hierarchy may be emulated by using an appropriate naming scheme, like the scheme of CL-NLP packages and the use of some utility functions. At the same time the package system is built with separation of concerns in mind that is lacking in hierarchical namespace systems of such languages as Python and Java that tie package names to the file system and make it difficult to evolve such hierarchy without the help of special tools. Besides, this coupling doesn't allow to easily aggreagate names from more than 1 file in a module that is often desirable.

At the next level each package exports a number of generic functions for major operations that serve as the API entry point: `tokenize`, `lemmatize`, `tag`, `parse` etc. The methods of such generic functions implement specific algortihms of tokenization, lemmatization etc. By convention the first argument of each generic function should be an instance of the class which specifies, what algorithm should be implemented. For instance, such classes as `regex-tokenizer`, `markov-chain-generator`, `hmm-tagger` are defined. These instances provide parametrization possibilities for the algorithms and allow for code reuse through inheritance.

The following code demonstrates this principle int the implementation of the simple regular-expression based word tokenizer.

```
(defgeneric tokenize (tokenizer string)
  (:documentation
   "Tokenize STRING with TOKENIZER. Outputs 2 values:
    - list of words
    - list of spans as beg-end cons pairs"))

(defclass regex-word-tokenizer (tokenizer)
  ((regex :accessor tokenizer-regex :initarg :regex
          :initform
          (re:create-scanner
           "\\w+|[!\"#$%&'*+,./:;<=>?@^`~…\\(\\)⟨⟩{}\\[\\|\\]‒–—
―«»“”‘’¶-]")
          :documentation
          "A simpler variant would be [^\\s]+ —
           it doesn't split punctuation, yet sometimes it's desirable."))
  (:documentation
   "Regex-based word tokenizer."))

(defmethod tokenize ((tokenizer regex-word-tokenizer) string)
  (loop :for (beg end) :on (re:all-matches (tokenizer-regex tokenizer)
                                           string)
                       :by #'cddr
     :collect (subseq string beg end) :into words
     :collect (cons beg end) :into spans
     :finally (return (values words
                              spans))))
```

Additionally, the CLOS method-combination facilities are extensively used to factor auxiliary actions out of the main methods implementations.

```
(defmethod tokenize :around ((tokenizer tokenizer) string)
  "Pre-split text into lines and tokenize each line separately."
  (let ((offset 0)
        words spans)
    (loop :for line :in (split #\Newline string) :do
       (multiple-value-bind (ts ss) (call-next-method tokenizer line)
         (setf words (nconc words ts)
               spans (nconc spans (mapcar #`(cons (+ (car %) offset)
                                                  (+ (cdr %) offset))
                                          ss)))
         (incf offset (1+ (length line)))))
    (values words
            spans)))
```

To further simplify development with CL-NLP singleton instances of certain classes with default parameter values are defined where it is possible. For instance, there's a default `<word-tokenizer>` singleton, which is an instance of `postprocessing-regex-word-tokenizer` class. Also by convention these instances have their names in angular brackets and there is a special macro for efficiently defining them. This is how the macro is invoked:

```
(define-lazy-singleton word-chunker
    (make-instance 'regex-word-tokenizer
                   :regex (re:create-scanner "[^\\s]+"))
  "Dumb word tokenizer, that will not split punctuation from words.")
```

It uses `define-symbol-macro` in its definition:

```
(defmacro define-lazy-singleton (name init &optional docstring)
  "Define a function NAME, that will return a singleton object,
   initialized lazily with INIT on first call.
   Also define a symbol macro <NAME> that will expand to (NAME)."
  (with-gensyms (singleton)
    `(let (,singleton)
       (defun ,name ()
         ,docstring
         (or ,singleton
             (setf ,singleton ,init)))
       (define-symbol-macro ,(mksym name :format "<~A>") (,name)))))
```

One of the biggest shortcomings of the design of NLP libraries based on conventional approach to object-orientation, such as NLTK or OpenNLP, is that the algorithms are implemented inside the concrete classes. Taking into account the extensive usage of inheritance this brings to the situation in practice, when parts of the implementation are scattered around several classes and files which greatly impedes its readability and in effect clarity. Another problem is the need to properly instantiate the classes before performing the tasks which may not be straighforward, especially in multi-threaded systems, as not all of the classes are implented in a thread-safe manner. The approach taken by CL-NLP which extensively utilizes CLOS capabilities tries to solve these problems while maintaining the extensible nature of object-oriented programs.

2. Main modules

2.1. Util

The `nlp.util` package defines the basic set of utilities to handle individual characters, words, text strings, files, trees of symbols, perform common mathematical operations and some supplementary utilities. The package `nlp.util` provides specific procedures to run unit tests and test algorithms with various corpora.

2.2. Core

The `nlp.core` package defines the basic data sctructures and algorithms used in other CL-NLP modules. They are also intended for stand-alone usage. These include:

- the basic `token` data-structure and `tokenization` generic function with several basic tokenization algorithms for splitting text chunks, words and sentences;
- language modelling with ngrams, implemented with the following classes:
    - `ngrams` is a low level class which wraps access to an underlying storage of ngrams (an in-

memory hash-table, a database etc.) It supports such functions as getting ngram frequency individually or in batchs (`freq` and freqs), approximated probability (`prob`, `probs`) and log of probability (`logprob`, `logprobs`), and conditional probabilites (`cond-prob`, `cond-probs`, `cond-logprob`, `cond-logprobs`). The default ngrams implementation is the `table-ngrams` class that stores them in a hash-table.

  - `language-model` is a more high-level interface that incapsulates access to a group of ngrams of different orders to provide smoothing capabilities for the methods `freqs`, `probs`, `logprobs` and `cond-logprobs`. Two implementations are provided: `plain-lm` without any smoothing and `stupid-backoff-lm` which implements the Stupid Backoff smoothing algorithm [LargeLMinMT]. Additionally, in `cl-nlp-contrib` the implementation of access to Microsoft Web N-gram Services [MSWebNgrams]. More language models should be added in the future to support other smoothing language models [SmoothingLMs], as well as adapter for external language modelling software, such as BerkeleyLM [BerkeleyLM].

2.3. Corpora

The `nlp.corpora` package implements functions to load and access commonly used lingusitic corpora, such the Brown corpus [BrownCorpus] or Penn Treebank [PennTreebank]. The basic data-structures for corpus management are:

- `text` which holds an individual unit of the corpus' text data in the raw, clean-up and tokenized forms;
- `corpus` which holds a collection of texts with possible various groupings of them (for example by-category, by-author etc).

2.3. Syntax

The `nlp.syntax` package provides the generic functions `tag`, `parse`, and `parse-n` (which returns the N most likely parses of the sentence). The implementations of taggers include an `hmm-tagger` and a `glm-tagger`. And for parsing a common PCFG-based algorithm is implemented with the lazy algorithm for finding N best parses [kBestParsing].

2.5. Wordnet

The `nlp.contrib.wordnet` package implements the interface to Wordnet lexical database of English [Wordnet]. There are at least two other libraries which provide access to Wordnet from Common Lisp that were mentioned previously. This libraries were not used as a basis for the implementation of Wordnet interface in CL-NLP. The reason for that was that is that they both interface (directly or indirectly) with the custom Wordnet file format. There are many alternative Wordnet storage formats, including a relational database ones [WordNetSQL]. The advantages of SQL-based representation are:

- it is standard so there may be multiple ways to interact with it, including the stock SQL clients;
- SQL interaction is well-supported in client libraries;
- it is transparent;
- it is distributed as a single file;
- it is rather fast and may be optimized if necessary.

That is why Wordnet interaction in CL-NLP was implemented using SQL Wordnet representation, specifically, an sqlite database. CLSQL [CLSQL] was chosen as the client library, because it supports many SQL databases and so allows to change SQL backend in the future if necessary.

The other benefit of implementing Wordnet support this way is that it provides another alternative solution for Lisp-Wordnet interface in addition to the existing ones.

2.6. Other modules

Other modules are for such functionality as phonetics, morphology, text generation, text classification and clustering, semantic analysis etc. are under development.

3. Implementation notes

3.1. Trade-offs

There are several qualities of code for which CL-NLP can be optimized. They include:

- simplicity – how simple is the implementation of the functionality;
- performance – how effective and optimized are the library's algorithms;
- extensibility – how easy it is to add new functionality;
- uniformity – how much does the code follow a single set of standards and conventions.

Extensibility and uniformity receive top priority in the implementation of CL-NLP. As for performance, the algorithms are made as efficient as possible without compromising the aforementioned qualities. The reasons for that are:

- in many use cases optimal performance is not required;
- when optimal performance is required, it is possible to create a custom optimized implementation of the algorithm. At the same time, supporting and extending a lot of custom implementations not following a set of unifying principles is an unnecessary burden.

As for simplicity, sometimes the implementation is made more general to allow for different use cases and provide several extension points. This, in effect, adds some complexity comapred to te most straightforward solution, but the complexity is justified by the overall preference to extensibility.

3.2. Stylistic issues

As in most programming language communities, there are many debates about good and bad style of Common Lisp programs. In general, CL-NLP follows the Google Common Lisp Styleguide [GoogleCLStyleguide]. Besides, the following stylistic principl apply:

1. Using the whole language.

   Some people suggest to exclude some Common Lisp functions/macros from the program lexicon on the basis that there are better alternative versions in the standard. The classic example of this is `do` and `loop` macros that both allow to express arbitrary iteration algorithms, but in substantially different manners. So the proposal is to choose one of the two and use it consistently. The approach taken in CL-NLP is the opposite: to use the construct that allows to express the computation in the most concise and clear way regardless of the constructs used in other parts of the system. This approach is also taken in anticipation of the need to integrate many algorithms coming from different people and sources, which may rely on different coding standards.

2. Extending the language to achieve more declarative and concise code.

   The “Growing a Language” [GrowingALang] approach is characteristic to Lisp systems, but at the same time it is often proposed not to use utility extensions to the language that merely change the surface syntax, but do not add anything to the semantics. CL-NLP is built on a different premise – that overlooking syntactic convenience is detrimental to the code clarity. It uses the reasonable-utilities library [rutils] which provides a lot of extensions to the Common Lisp standard library to improve the usability of handling strings, hash-tables, files, sequences etc.

3. Providing multiple choice to the user.

   There is a broad range of use cases for CL-NLP: from experimental work to production usage. To support different usage scenarios and in effect a different interaction model with the library clients, the library should provide different interfaces. For instance, for interactive use it is convenient to have all the functions immediately available, to operate with short operation names, to have sensible defaults. The requirements for production systems are more in the areas of possibility to granuarily control the used operations, to optimize their performance and resource requirements and to maintain the resulting code. CL-NLP's development goal is to support all these modes of operations by providing many alternative usage paths with the help of packaging, aliases, default implementations and pre-configured classes and other means.

4. References

[AIProg] Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, James R. Meehan, 1987, “Artificial Intelligence Programming”

[BasicEngGrammar] http://www.cliki.net/Basic-English-Grammar

[BerkeleyLM] https://code.google.com/p/berkeleylm/

[BrownCorpus] W. N. Francis, H. Kucera, 1979, “A Standard Corpus of Present-Day Edited American English, for use with Digital Computers, Revised and Amplified”

[cffiWordnet] https://github.com/kraison/cffi-wordnet[Kakkonen] Tuomo Kakkonen, 2007, "Framework and Resources for Natural Language Parser Evaluation"

[CLEarly] http://www.cliki.net/CL-EARLEY-PARSER

[CLSQL] http://clsql.b9.com/

[CLWordnet] https://github.com/TheDarkTrumpet/cl-wordnet

[CMUAI] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/0.html

[CoreNLP] Stanford Core NLP, http://www-nlp.stanford.edu/software/corenlp.shtml

[GoogleCLStyleguide] Robert Brown, François-René Rideau, “Google Common Lisp Style Guide”, http://google-styleguide.googlecode.com/svn/trunk/lispguide.xml

[GrowingALang] Guy L. Steele Jr., 1999, “Growing a Language”

[kBestParsing] Liang Huang, David Chiang, 2005, “Better k-best Parsing”

[langutils] Ian Eslick, Hugo Liu, 2005, "Langutils: A Natural Language Toolkit for Common Lisp"

[LargeLMinMT] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, Jeffrey Dean, “Large

Language Models in Machine Translation"

[Lexiparse] Drew Mcdermott, 2005, "Lexiparse: A Lexicon-based Parser for Lisp Applications"

[MSWebNgrams] http://web-ngram.research.microsoft.com/

[NLPinLisp] Gerald Gazdar, Chris Mellish, "Natural Language Processing in Lisp/Prolog/Pop11", source code available from: https://bitbucket.org/msorc/nlp-in-lisp

[NLTK] NLTK, http://nltk.org/

[NLU] James Allen, 1994, "Natural Language Understanding (2nd Edition)"

[OpenNLP] Apache OpenNLP, http://opennlp.apache.org/

[PAIP] Peter Norvig, 1992, "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp"

[PennTreebank] http://www.cis.upenn.edu/~treebank/

[rutils] https://github.com/vseloved/rutils

[SmoothingLMs] Gina-Anne Levow, "Smoothing N-gram Language Models"

[Soundex] http://www.cliki.net/Soundex

[sparser] https://code.google.com/p/sparser

[Wordnet] http://wordnet.princeton.edu/

[WordNetSQL] http://sourceforge.net/projects/wnsql/