SWI-Prolog Natural Language Processing Primitives

Jan Wielemaker
VU University Amsterdam
The Netherlands
E-mail: J.Wielemaker@cs.vu.nl

June 18, 2015

Abstract

This package contains some well known basic routines for natural language processing and information retrieval. The current version of this package is very limited, which makes the name somewhat misleading. Suggestions and contributions are welcome.

Contents

1 Double Metaphone – Phonetic string matching

The library `double_metaphone` implements the *Double Metaphone* algorithm developed by Lawrence Philips and described in "The Double-Metaphone Search Algorithm" by L Philips, C/C++ Users Journal, 2000. Double Metaphone creates a key from a word that represents its phonetic properties. Two words with the same Double Metaphone are supposed to sound similar. The Double Metaphone algorithm is an improved version of the *Soundex* algorithm.

double_metaphone(+*In, -MetaPhone*)
: Same as `double_metaphone/3`, but only returning the primary metaphone.

double_metaphone(+*In, -MetaPhone, -AltMetaphone*)
: Create metaphone and alternative metaphone from *In*. The primary metaphone is based on english, while the secondary deals with common alternative pronounciation in other languages. *In* is either and atom, string object, code- or character list. The metaphones are always returned as atoms.

1.1 Origin and Copyright

The Double Metaphone algorithm is copied from the Perl library that holds the following copyright notice. To the best of our knowledge the Perl license is compatible to the SWI-Prolog license schema and therefore including this module poses no additional license conditions.

2 Porter Stem – Determine stem and related routines

The `porter_stem` library implements the stemming algorithm described by Porter in Porter, 1980, "An algorithm for suffix stripping", Program, Vol. 14, no. 3, pp 130-137. The library comes with some additional predicates that are commonly used in the context of stemming.

porter_stem(+*In, -Stem*)
: Determine the stem of *In*. *In* must represent ISO Latin-1 text. The `porter_stem/2` predicate first maps *In* to lower case, then removes all accents as in `unaccent_atom/2` and finally applies the Porter stem algorithm.

unaccent_atom(+*In, -ASCII*)
: If *In* is general ISO Latin-1 text with accents, *ASCII* is unified with a plain ASCII version of the string. Note that the current version only deals with ISO Latin-1 atoms.

tokenize_atom(+*In, -TokenList*)
: Break the text *In* into words, numbers and punctuation characters. Tokens are created to the following rules:

`[-+][0-9]+(\.[0-9]+)?([eE][-+][0-9]+)?`	number
`[:alpha:][:alnum:]+`	word
`[:space:]+`	skipped
anything else	single-character

Character classification is based on the C-library iswalnum() etc. functions.

It is likely that future versions of this library will provide `tokenize_atom/3` with additional options to modify space handling as well as the definition of words.

atom_to_stem_list(+*In, -ListOfStems*)
Combines the three above routines, returning a list holding an atom with the stem of each word encountered and numbers for encountered numbers.

2.1 Origin and Copyright

The code is based on the original Public Domain implementation by Martin Porter as can be found at `http://www.tartarus.org/martin/PorterStemmer/`. The code has been modified by Jan Wielemaker. He removed all global variables to make the code thread-safe, added the unaccent and tokenize code and created the SWI-Prolog binding.

3 library(snowball): The Snowball multi-lingual stemmer library

See also `http://snowball.tartarus.org/`

This module encapsulates "The C version of the libstemmer library" from the Snowball project. This library provides stemmers in a variety of languages. The interface to this library is very simple:

- `snowball/3` stems a word with a given algorithm
- `snowball_current_algorithm/1` enumerates the provided algorithms.

Here is an example:

```
?- snowball(english, walking, S).
S = walk.
```

snowball(+*Algorithm, +Input, -Stem*) *[det]*
Apply the Snowball *Algorithm* on *Input* and unify the result (an atom) with *Stem*.

The implementation maintains a cache of stemmers for each thread that accesses `snowball/3`, providing high-perfomance and thread-safety without locking.

Arguments

Algorithm	is the (english) name for desired algorithm or an 2 or 3 letter ISO 639 language code.
Input	is the word to be stemmed. It is either an atom, string or list of chars/codes. The library accepts Unicode characters. *Input* must be *lowercase*. See `downcase_atom/2`.

Errors

- `domain_error(snowball_algorithm, Algorithm)`
- `type_error(atom, Algorithm)`
- `type_error(text, Input)`

snowball_current_algorithm(*?Algorithm*) *[nondet]*

True if *Algorithm* is the official name of an algorithm suported by `snowball/3`. The predicate is `semidet` if *Algorithm* is given.

4 library(isub): isub: a string similarity measure

author Giorgos Stoilos

See also *A string metric for ontology alignment* by Giorgos Stoilos, 2005.

The `library(isub)` implements a similarity measure between strings, i.e., something similar to the *Levenshtein distance*. This method is based on the length of common substrings.

isub(+*Text1:atomic, +Text2:atomic, +Normalize:bool, -Similarity:float*) *[det]*

Similarity is a measure for the distance between *Text1* and *Text2*. E.g.

```
?- isub('E56.Language', 'languange', true, D).
D = 0.711348.
```

If *Normalize* is `true`, `isub/4` applies string normalization as implemented by the original authors: *Text1* and *Text2* are mapped to lowercase and the characters "._" are removed. Lowercase mapping is done with the C-library function towlower(). In general, the required normalization is domain dependent and is better left to the caller. See e.g., `unaccent_atom/2`.

Arguments

Similarity	is a float in the range [0.0..1.0], where 1.0 means *most similar*

Index