A Spanish Stemming Algorithm Implementation in PROLOG and C#

Dennis D. Perez Barrenechea
Artificial Intelligence Center - The University of Georgia
Athens, Georgia 30602-7415 U.S.A.
http://www.ai.uga.edu/

October, 2006

Abstract

This paper presents two implementations of a Spanish stemming algorithm in Prolog and C#. The basis for the implementations is a Porter-like algorithm published by the Snowball Project. Some additions to the original algorithm are proposed and included in the programs, allowing them to identify a larger number of words and suffixes.

1 Introduction

A stemming algorithm is a technique used in Information Retrieval (IR) and some other applications of Natural Language Processing (NLP), which removes suffixes from a word in order to obtain a stem or base form which could be easily matched in databases or documents (Jurafsky 2000). Its use is based on the premise that two words with the same stem have very close semantic content. The several possible variations of the derivatives, inflected forms, gender and number changes, and other phenomena, make the grouping

of all the variants under a common stem advisable. Applications that do not take these effects into account may end up with difficulties when comparing queries and documents, or dispersal effects in word frequency calculations.

2 The Improved Snowball Spanish Stemming Algorithm

This work is based on the Spanish stemming algorithm published by the Snowball project (Snowball 1999). The algorithm starts extracting sections from a word and labeling them as *RV* and *R2*. *RV* is defined as the region of the word that starts after the third letter, or null if not exists. To define *R2*, *R1* needs to be defined. *R1* is the region after the first non-vowel following a vowel, or null if not exists. For example, in the word *precios*, the first non-vowel following a vowel would be the *c*. Therefore, *R1* would be *ios*. Similarly, in the word *bellísimo*, the first non-vowel following a vowel is the first *l*, therefore *R1* would be given by *lísimo*.

R2, on the other hand, is a region that starts after the first non-vowel following a vowel in *R1*, or null if none exists. In the first example, *R2* would be null, since there's no other letter following the *s*, the first non-vowel following a vowel in *R1*. In the second example, *R2* would be given by *imo*.

The first step in the algorithm per se, called *step 0*, is the removal of the attached pronouns. The rule is to remove the longest of the following in *RV*:

me, se, sela, selo, selas, selos, la, le, lo, las, les, los, nos

They will be removed only if the come after

iéndo, ándo, ár, ér, ír, iendo, ando, ar, er, ir, [u]yendo

Note that the *u* in *uyendo* could be outside *RV*. Finally, acute accents must be removed.

The next step (*Step 1*) removes any standard suffixes from *R2*. The list of suffixes considered as standard by the algorithm is given below:

anza, anzas, ico, ica, icos, icas, ismo, ismos, able, ables, ible, ibles, ista, istas, oso, osa, osos, osas, amiento, amientos, imiento, imientos

icadora, icador, icación, icadoras, icadores, icaciones, icante, icantes, icancia, icancias

adora, ador, ación, adoras, adores, aciones, ante, antes, ancia, ancias

logía, logías

ución, uciones

encia, encias

ativamente, ivamente, osamente, icamente, adamente, amente

antemente, ablemente, iblemente, mente

abilidad, abilidades, icidad, icidades, ividad, ividades, idad, idades

ativa, ativo, ativas, ativos, iva, ivo, ivas, ivos

If *logía* or *logías* is removed, it must be replaced by *log*. Similarly, if *ución* or *uciones* is removed, it must be replaced by *u*. Finally, if *encia* or *encias* is removed, it must be replaced by *ente*.

Step 2 is only performed if *Step 1* performed no modifications, over *RV*. The step is divided in two: *Step 2a* would remove verb suffixes beginning with *y*, and *Step 2b* other verbs suffixes. *Step 2b* is only done if *Step2a* failed to remove a suffix.

On *Step2a*, the algorithm looks in *RV* for the longest of the following:

ya, ye, yan, yen, yeron, yendo, yo, yó, yas, yes, yais, yamos

Any of these is only removed if is preceded by a *u* that is not needed to be in *RV*.

As was said before, *Step 2b* is only performed if *Step 2a* failed to remove a suffix. The longest of these suffixes must be removed from *RV*:

en, es, éis, emos, guen, gues, guéis, guemos

arían, arias, arán, arás, aríais, aria, aréis, aríamos, aremos, ará, aré, erían, erías, erán, eras, eríais, ería, eréis, eríamos, eremos, erá, eré, irían, irías, irán, irás, iríais, iría, iréis, iríamos, iremos, irá, iré, aba, ada, ida, ía, ara, iera, ad, ed, id, ase, iese, aste, iste, an, aban, ían, aran, ieran, asen, iesen, aron, ieron, ado, ido, ando, iendo, ió, ar, er, ir, as, abas, adas, idas, ías, aras, ieras, ases, ieses, ís, áis, abais, íais, arias, ierais, aseis, ieseis, asteis, isteis, ados, idos, amos, ábamos, íamos, imos, áramos, iéramos, iésemos, ásemos

Any of *guen*, *gues*, *guéis* or *guemos* must be replaced by *g*, and both the *g* and *u* do not need to be in *RV*.

Finally, *Step 3* must be performed always. It removes any remaining suffix (residual). This step search for the longest of the following:

os, a, o, á, í, ó, e, é, ue, ué

In the case of *ue* and *ué*, they can only be removed if preceded by a *g*, which could or not be in *RV*.

Once all steps have been completed, all acute accents must be removed from the remaining characters. This is the stem that the algorithm returns as output.

After careful revision of the features of this algorithm, two improvements were quickly identified and added to the implementation. The first included new instances of attached pronouns, in *Step 0*. The added suffixes are presented below:

te, telo, melo, telos, melos, tela, mela, telas, melas

Variations like, for example, the verb *tomar* (to drink): *tomarmelo*, *tomarmelos*, *tomarmelas*, *tomartelas*, were not being included in the original algorithm.

The second improvement had to deal with diminutives and superlatives, suffixes highly used in colloquial speaking (and writing) in Spanish. Even

though they are not standard for the different Spanish-speaking countries (some countries may use *-ito* as diminutive, while others *-ico* or *-illo*), they are extensively used. The list of the diminutives and superlatives included in the implementation are given below:

> ito, ita, azo, aza, lin, lina, in, ina, on, ona, itillo, itilla, cillo, cilla, illo, illa, itico, itica, ico, ica, ote, ota

The step was included as *Step 4* over the original algorithm. Note these modifiers only affect to adjective words in Spanish. They should be removed only if included in *RV*.

3 Implementation in Prolog

Two predicates are available on the Prolog implementation. The first one of them,

```
stemWord(+Word, -Stem)
```

will allow the user to stem a word in Spanish. The other predicate,

```
stemFile(+InputFilename, +OutputFilename)
```

will load a text file whose name is specified as the first parameter and will stem every word it finds inside it, storing the resulting stems into the file specified in the output filename (second parameter).

For example, `stemWord("casamiento",X).` will return `X` as `"cas"`. In the case of stemming a whole file, the new file (in either a directory specified in the parameter or the default directory) will be generated. For example, for stemming the content of `sample1.txt` into a file named `sample1.out`:

```
stemFile("C:\\TMP\\sample1.txt","C:\\TMP\\sample1.out").
```

The content of `sample1.txt` is:

> Perú, país mágico y milenario, posee una diversidad y riqueza poco comunes en el mundo y ofrece al visitante infinitas alternativas y la posibilidad de vivir una experiencia única: Historia, cultura, naturaleza, aventura y mucho más en un solo destino.

The content of the stemmed text file, `sample1.out` is:

> Peru pais mag y milenari pose un divers y riquez poc comun en el mund y ofrec al visit infinit altern y la posibil de viv un experienci un Histori cultur naturalez aventur y much mas en un sol destin

Each word is given in a different line.

Performance has always been considered through the development of this solution in Prolog. By design, Prolog is optimized to work with lists and unification [1] . The use of this powerful tool plus some other well-known optimization techniques like tail recursion [2] assures that the program will make a good use of the memory resources as well as improve response time.

Since performance is an important goal in the Prolog solution, the algorithm is optimized to work fast when dealing with large texts. For this objective to be achieved, the stemFile predicate "tokenizes while it stems". What happens is that the program reads every character on the input file and stores them on a current word stack, until it finds an end of word character (i.e. a blank space). This stack is a list with the word in inverse order. Since what the stemmer really does is to remove suffixes from words, this setting is perfect for trying unification and to remove an existing suffix. Therefore, at this point the algorithm calls the `stemWordInv(+WordInv, -StemInv)` predicate, which receives as first parameter a list of characters, an inverted word; and returns as second parameter a list of character, which is the inverted stem.

[1] The mechanism of binding the contents of variables; Can be viewed as a kind of one-time assignment.

[2] In computer science, tail recursion (or tail-end recursion) is a special case of recursion where the recursive call is the last thing that happens in a function.

Some of the predicates do not use tail recursion to optimize the code. An example is `getR2(+Word, -R2)`, which call itself recursively until no other letter is present and when returning from the recursive call, counts the number of non-vowels after a vowel to find *R2*, as the definition of this part of the word is made.

For the inverse word analysis to work, the other predicates which detail the suffixes to be removed from the original word are encoded in inverse form, to make efficient use of unification and the lists. Being *Step 0* the only one from all steps were any pronoun suffix may be removed if any previous combination of characters exists previous to the presence of this suffix in the word, both the predicates `pronoun(+WordInv, -WordTmp)` and `prepronoun(+WordTmp, -StemInv)` combine to make any removal from the original word.

Step 1 recieves *R2* to check for a standard suffix to be removed. To achieve this goal, the `standard_suffix(?Suffix, ?ReplacingChars)` predicate is used, which not only details the sequence of characters to remove (the first parameter), but also the sequence of characters that may replace this removed ones. Note that no character may be given as a parameter here.

Step 2 is similar to *Step 1*, but in this case the algorithm uses *RV*. In this case, the `verb_suffix_b(?VerbSuffix, ?ReplacingChars)` predicate details both the sequence to search and remove, and the replacing sequence; And the `verb_suffix(?VerbSuffix)` predicate allows to search for the verb suffixes detailed in *Step 2a*.

Finally, with the `residual_suffix(?Suffix , ?Replace)` predicate, both *Step 3* and *Step 4* are implemented. This predicate contains the suffixes detailed in both the residual suffixes step and the diminutives/superlatives step, our additions to the original Snowball algorithm.

The stemWord predicate only inverts the word and calls the stemWordInv predicate, to finally invert the resulting inverted stem.

4 Implementation in C#

The C# implementation contains two files: The `SpanishStemmer` class, which provides static methods for stemming a word in Spanish, and can be added to any application in .Net. A console application is also provided, which allows the user to stem any number of text files by generating the corresponding stemmed texts all at once.

Even though filenames with paths can be used, it is recommended to place all text files in the same directory than `SpanishStemmerApp.exe`. All output files will be generated in the same path than the original file and with the same name, but with extension .out.

To run the application, execute the windows console (`cmd.exe`), and run `SpanishStemmerApp.exe`. A list of the files to be stemmed must be given as a parameter. For example:

```
C:\SpanishStemmer\SpanishStemmerApp.exe Sample1.txt Sample2.txt
```

will execute the stemming algorithm on `Sample1.txt` and `Sample2.txt` files, generating the `Sample1.out` and `Sample2.out` output files in the original directory.

This implementation, as opposed to the one in Prolog, resembles as much as possible the original algorithm. Minimal performance-related changes have been applied to it, to maintain the readability of the code. The `SpanishStemmer` class provides a static function, `stemWord()`, which receives the original word as input and returns the stem of the word.

5 References

Jurafsky, Daniel and Martin, James (2000) *Speech and Language Processing*, Prentice-Hall. pp. 82-83.

Snowball Project (1999) *A Spanish Stemming Algorithm*. Available online at http://snowball.tartarus.org/algorithms/spanish/stemmer.html.