XMLmind Spell Checker - Dictionary Builder User's Guide

Xavier Franc, Pixware <xsc-support@xmlmind.com>

Abstract

This documentation gives general instructions to build, modify and enhance compiled dictionaries for XMLmind Spell Checker.

1. Introduction 1
2. Dictionary types 1
    2.1. Extending dictionaries 2
3. Storage Structure of Dictionaries 2
4. Using the builder 3
    4.1. System Requirements 3
    4.2. Obtaining and installing the builder 3
    4.3. Command and options 3
5. Building from `ispell` dictionaries 4
6. Format of the hints file 5
    6.1. Character declarations 6
    6.2. The `%mistake` directive 6
    6.3. The `%kbline` directive 7
    6.4. Miscellaneous 7

1. Introduction

The XMLmind Spell Checker component comes with a command-line tool called the``Dictionary Builder'' that allows creating compiled dictionaries from plain word lists.

To build a compiled dictionary, the builder takes as input the following data:

- An expanded word list, which is a simple text file with words separated by whitespace (space, tabs, or newlines). Actually, *several* such lists can be accepted by the builder: they are simply merged.
- A *hints file* that defines language-specific properties, plus hints for the spell checker engine to work in a smarter way.

  The syntax and the semantics of hints files are described below.
- It is also possible to specify *common prefixes* (a word list file), a set of prefixes that can be prepended to any word (if the `AllowPrefixes` option is enabled). For example, if you define "super-" as a common prefix, any legal word prefixed with "super-" will be accepted. This is a feature to use cautiously, therefore it can be disabled by an option.
- Additionally, the builder accepts another word list file that defines the most frequent words in the language. This is used to "boost" those frequent words toward the top of the suggestion lists, and thus provides statistically better suggestions.

2. Dictionary types

There are three types of dictionary supported by XMLmind Spell Checker:

- Compiled dictionaries (`.cdi`), which are the main topic of this documentation, are a fast and memory-efficient representation of large word lists.

- A Composite Dictionary allows to present a group of dictionaries as a single dictionary (more precisely the resulting dictionary is the union of all its components). For example a country variant like `en-US` aggregates a big English common base with a smaller US-specific dictionary. This mechanism reduces memory and disk usage.

  A Composite Dictionary appears as a simple text file beginning with the magic line "`@multilink:`", followed by lines containing the URL of sub-dictionaries. URL are generally relative to the composite dictionary, but can also be absolute. Referenced dictionaries can in turn be Composite.

  For example, this can be the '`default`' dictionary for `en-US` (it refers to the common English dictionary `en/base.cdi`).

  ```
  @multilink:
  ../en/base.cdi
  spec.cdi
  ```

- There are also plain text dictionaries, used for example for personal dictionaries. Their structure is very simple: one word per line. The default text encoding is UTF-8, but XMLmind dictionaries use ISO-8859-1.

  > **Note**
  >
  > Text dictionaries use much more memory than compiled dictionaries, and are reloaded each time they are selected (while compiled dictionaries are managed in a cache). Therefore the use of text dictionaries should be reduced to small amounts of words (a few hundreds at most).

2.1. Extending dictionaries

Through the Composite Dictionaries mechanism explained above, it is possible to extend a dictionary in a simple way. For example, one can define a second dictionary named 'extended' beside the 'default' dictionary with this contents:

```
@multilink:
default
?http://www.dictionaries.com/myextensions.cdi
```

Here we have an absolute (and imaginary) HTTP URL pointing to a compiled dictionary.

Notice the (*optional*) question mark preceding the URL: it is a protection against an access failure: if the read operation fails, no exception is raised and the dictionary loading proceeds as if the URL were absent.

3. Storage Structure of Dictionaries

Dictionaries provided by XMLmind are delivered either in *Dictionary Archives* with a `.dar` extension, which are actually jar files.

There is one `.dar` for each language available (currently English, French, German, Spanish) plus an archive that gathers all languages (`dicts.dar`) which can be used for example in Java™ Web Start applications.

There is a simple conventional structure for these archives: the language name appears as a directory, the actual dictionaries are elements of these directories. This corresponds with the dictionary naming convention used in XMLmind Spell Checker.

A `.dar` archive can contain several languages. For example here are the contents of `en.dar` (English with country variants):

```
en/
en/base.cdi
en/allvar.cdi
en/default
en-CA/
en-CA/default
```

```
en-CA/spec.cdi
en-GB/
en-GB/default
en-GB/spec.cdi
en-US/
en-US/default
en-US/spec.cdi
```

As of version 1.1p3, it is also possible to access an dictionary using any URL supported by the Java Runtime Environment.

4. Using the builder

4.1. System Requirements

The builder requires a Java™ Runtime Environment (JRE) version 1.4+.

A powerful machine is recommendable. The compression algorithm used by the builder is very memory intensive. You should have at least 128Mb of memory (or of swap space at a pinch) on your machine. The `dictbuilder` script sets the memory limit to a high value. In exceptional cases, you might have to set it higher. As an indication, compiling our big French word list (800,000 words, 9.8Mb) requires 30 seconds and 87Mb of memory on a 1 GHz Pentium III.

4.2. Obtaining and installing the builder

The builder is no longer included in the SDK. It must be downloaded separately from www.xmlmind.com/spellchecker/.

In all cases, the builder is a command-line utility: a shell file named `dictbuilder` on Unix or MacOS, `dictbuilder.bat` on Windows.

4.3. Command and options

General form of the command line:

```
dictbuilder ?options? word_list ... word_list ?-sub word_list ... word_list?
```

It is also possible to *use a compiled dictionary as input*. This is the way to create a new version of an existing dictionary if you do not possess the source word list.

General options:

-cs *`character_encoding`*
: Encoding used in word lists, frequent word list and hints files. This must be an encoding supported by Java™ runtime.

> ### Important
>
> This option must be placed *before* the files it applies to.

-hints *`hints_file`*
: Specifies the hints file.

> ### Important
>
> Specifying a hints file is almost always needed as this file is used to specify which characters may be used to form a word.
>
> The hints files used to build XMLmind's `en`, `fr`, `de`, and `es` dictionaries are found here: en.hints, fr.hints, de.hints, es.hints. Note that the encoding of all these hints files is ISO-8859-1.

-freq *word_list*
: List of frequent words.

-prefixes *word_list*
: List of standard prefixes.

-sub *word_list* ... *word_list*
: Every word list whose path follows this option will be *subtracted* from the resulting dictionary, instead of being merged with. It means that every word belonging to this word list will be absent from the result. This option should be placed after the input word lists.

-o *output_file*
: Specifies the compiled dictionary output file. The convention is to use a `.cdi` extension, but there is no obligation.

Other options:

-verbose
: Explain what is being done.

-dump *out_word_list*
: After merging all the compiled and textual word lists specified in the command line and after subtracting words if the `-sub` option is used, output the resulting word list in specified text file. As always, the encoding of the generated text file is specified using the `-cs` option.

Example 1: Create compiled dictionary `mylang.cdi` out of word lists `mywords.txt` and `extrawords.txt`. The encoding of all text files specified in the command line is `ISO-8859-2`. Hints file is `mylang.hints`. Frequent words are contained in `frqw.txt`. Standard prefixes are contained in `myprefixes.txt`.

```
dictbuilder -cs ISO-8859-2 -hints mylang.hints -freq frqw.txt -prefixes myprefixes.txt \
    mywords.txt extrawords.txt -o mylang.cdi
```

Example 2: Add words contained in `added_words.txt` to compiled dictionary `de.cdi`. Compile the resulting word list as `new_de.cdi`.

```
dictbuilder -cs ISO-8859-1 -hints de.hints de.cdi added_words.txt -o new_de.cdi
```

Example 3: Subtract words contained in `removed_words.txt` from compiled dictionary `de.cdi`. Compile the resulting word list as `new_de.cdi`.

```
dictbuilder -cs ISO-8859-1 -hints de.hints de.cdi \
    -sub removed_words.txt -o new_de.cdi
```

Example 4: Output in text file `de.txt` all the words contained in compiled dictionary `de.cdi`.

```
dictbuilder -verbose -cs ISO-8859-1 -hints de.hints de.cdi -dump de.txt
```

5. Building from `ispell` dictionaries

In the realm of public-domain spell-checking, `ispell` and its “affix files” are a popular way of defining word lists. See http://fmg-www.cs.ucla.edu/geoff/ispell.html for an introduction to ispell.

If you want to build a dictionary for a language that is not in the distribution, you will probably find (if it exists) a word list in ispell format. Check the following URL: http://fmg-www.cs.ucla.edu/geoff/ispell-dictionaries.html

Notice that most of these dictionaries are covered by the GNU Public License. Therefore you have to check whether it is suitable for your needs: it is always suitable for personal use, but redistributing compiled dictionaries could raise a legal issue with some authors.

You need of course an operational `ispell` installed on your machine (most probably a Unix box). It is strongly recommended to have a 8-bit clean version, with a compile-time option "`MASK BITS`" set to 64. This sounds esoteric, but it should be the case with most recent Linux distributions.

The basic job is to expand the affixed word list into a plain word list. For this you have to first compile into a ispell ``hashfile'':

```
buildhash mydict.txt mydict.aff mydict.hash
```

Here the `mydict.txt` file is the affixed word list, `mydict.aff` the affix file.

Then, the word list is expanded this way:

```
ispell -e -d ./mydict.hash < mydict.txt > mydict.wl
```

Then the expanded word list (here `mydict.wl`) can be used with the builder.

6. Format of the hints file

When encountering an unrecognized word, the spell checker engine tries to alter it in different ways and find the altered forms in its dictionaries. Standard alterations are: inserting a character (to correct an omission), suppressing a character (to correct a superfluous keystroke), swapping two adjacent characters, replacing one character by another (especially pairs of characters that are neighbors on a keyboard).

The spell checker engine can also use specific knowledge of the considered language. Some spell checker engines convert the word into phonetics and lookup the phonetic form. This is powerful for seriously bad misspellings. The drawback is that writing phonetic conversion rules is tedious and delicate.

The method used by XMLmind Spell Checker is both simple and powerful: it consists of specifying groups of character sequences that are easily mistaken one for the other. For example, if the sequences "ph"and "f" are said to be easily mistaken one for the other, the engine will quickly find the correct spelling for "elefant", because it will try to substitute "f" by "ph".

Most often, such hints reflect phonetic similarity, but they can also deal with more specific cases (for example in French, people often write "ceuil" instead of "cueil" in words like "recueil", "accueil" etc.). Some spell checkers treat such frequent mistakes by using special catalogs, but the method implemented in XMLmind Spell Checker is more general and powerful.

In short, the hints files define two types of information:

- *Characters allowed in words and their properties* (see below)
- *Proximity between particular sequences of characters*: (directive `%mistake`)

  This directive specifies that some character sequences are likely to be confused. This is a hint for helping the spell-checking engine find smart suggestions.

  Example: the following rule means that f ff and ph sound the same in can be tried instead of each other. This rule can help to sort out 'giraphes' and 'elefants'...

  ```
  %mistake f ff ph
  ```

  Another example: in French, "ell" "èl", resp. "au" "eau" "ô" sound similarly; this can be expressed by two rules like:

  ```
  %mistake ell èl
  %mistake au eau ô
  ```

  With such a rule, if one mistakenly writes "burau", the proper suggestion "bureau" will come atop the suggestion list more easily.

  A more common example: in most languages, some letters appear sometimes doubled, (for example: "spell"), sometimes not doubled (for example: "repel"). To deal with such mistakes, the following rule could be given:

  ```
  %mistake ll l
  ```

The `%mistake` directive can of course be applied to simple characters:

```
%mistake a à â
%mistake e é è ê ë
```

- *Special case: keyboard proximity between characters*:

  An erroneous occurrence of a character can come from the keyboard layout: a finger slip can replace a key by its neighbor. It could be expressed by a `%mistake` rule for each pair of adjacent keys, but this would be tedious to write: the `%kbline` directive provides an easy way to define the keyboard proximity rules. (See below)

6.1. Character declarations

XMLmind Spell Checker requires a declaration for characters used in word lists. This helps to detect malformed words.

By default, the ASCII uppercase and lowercase letters, digits, hyphen and dot are declared as acceptable 'word characters'.

To declare supplementary characters, use the `%chars` directive. It takes one argument (i.e. no space inside) which is a string of characters to declare. For example:

```
%chars àâéèëêîôùû
```

The `%chars` directive declares the characters may appear anywhere in the word.

Two other directives `%noninitial` and `%nonfinal` allow to refine this. They define whether a character may appear at the first or the last position in a word. For example:

```
%noninitial '
%nonfinal '
```

means that the apostrophe may appear only inside a word, not at the beginning (`%noninitial`) or at the end (`%nonfinal`).

By default the hyphen and the dot are non-initial and non-final.

These directives are rarely used beyond the example above (Namely in French and Italian).

6.2. The `%mistake` directive

The syntax is very simple:

```
%mistake[modifier] seq1 seq2 ... seqN
```

This means that each time one of these sequences is found in an unknown word, the spell-checking engine will attempt to replace it by one of the other sequences of the same rule and lookup the newly formed hypothesis in the dictionary.

To put it more clearly, let's consider the rule `%mistake f ff ph` and assume that the word 'elefant' is encountered. The engine here will try to replace "f" by "ff" and "ph", generating and looking up in the dictionary "eleffant" and "elephant", and in principle will find the latter as a suggestion.

The modifier is an indication of how likely the substitutions are. The possible forms are '-' (less likely), or '+' (more likely). Several modifiers can be combined. For example, in French we could have the following directives:

```
%mistake+  a â à
%mistake++ i î
%mistake-  i y
```

It means that stumbling over grave or circumflex accents is quite likely, while confusing a 'i' with 'y' is less likely.

Note: the `%mistake--` likelihood is the default for *any* pair of letters. So it is generally useless to specify more than one '-'.

It is suggested to use this directive with moderation, as it can slow down the engine. Especially, directives with many sequences lead to a higher combinatorial complexity.

Special cases: characters ^ and $

These characters have a special meaning. When used in a sequence, they make the sequence match only when appearing respectively at the beginning or the end of a word. For example:

```
%mistake  ^kn ^n
%mistake  $ gh$ w$
```

The first rule tells that at the beginning of a word "kn" can be mistaken for (sounds like) a "n". The second rule means that at the end of a word, "gh" or "w" can be forgotten or erroneously added ("$" alone means "nothing" or "silent").

Note

It makes no sense to mix sequences with and without a "$" (resp. a "^"). However it is possible for a sequence to have both (whole word). This should be used with moderation.

6.3. The `%kbline` directive

This is in fact a kind of shortcut to replace many `%mistake` directives: the argument is a string of horizontally adjacent characters of a keyboard. The directive specifies that each character is ``close to" its one or two neighbors.

For example here, "q" is close to "w", "w" to "e", "e" to "r" etc.

```
# English keyboard:

%kbline qwertyuiop
%kbline asdfghjkl
%kbline  zxcvbnm
```

The likelihood defined is roughly equivalent to the one of `%mistake-`. Modifiers can also be applied to `%kbline`. Thus `%kbline+` is roughly equivalent to `%mistake`.

6.4. Miscellaneous

Another directive controls the compound words: `%compoundmin` *`length`*

This directive means that compound words (without hyphens) are automatically allowed, provided that the length of each component is at least the length specified in the directive. This is meant for German and Nordic languages.

For example in German, the directive `%compoundmin 3` means that words like "aus" and "gehen" can be automatically composed into "ausgehen", and that "in" and "gehen" will not allow "ingehen" (because the length of "in" is less than 3).