Institute for Research in Cognitive Science

Porting SIMR to New Language Pairs

I. Dan Melamed

University of Pennsylvania
3401 Walnut Street, Suite 400A
Philadelphia, PA 19104-6228

October 1996

Site of the NSF Science and Technology Center for Research in Cognitive Science

IRCS Report 96--26

Porting SIMR to New Language Pairs

I. Dan Melamed
University of Pennsylvania

Abstract

There are three steps in porting SIMR to new language pairs. The first step is to choose an appropriate matching predicate, and collect any linguistic resources required by that matching predicate. The second step is to implement bitext space axis generation routines that are consistent with the matching predicate. The last step is to re-optimize SIMR's numerical parameters. This document explains each step in detail. It assumes that you have read and understood the paper [1]. It also assumes that you have SIMR and the porting tools properly installed.

1 Matching Predicate

SIMR's matching predicates can be based on any combination of predicate filters and oracle filters [3]. Your choice of matching predicate will depend on the languages that you are dealing with and the linguistic resources that you have at your disposal. If you are dealing with languages that share many cognates, or even phonetic cognates, then your matching predicate should test for cognateness. Cognate predicates using seem to work best with the LCSR metric [3]. If you have access to part-of-speech (POS) taggers for both languages, then your predicate should test for matching POS. If you have access to a pre-existing translation lexicon between your two languages, even a small and noisy one, then by all means use it. The `nlplib.pl` library that comes with your SIMR distribution includes matching predicates that represent all possible combinations of POS, cognate and translation lexicon filters. However, other matching predicates are certainly possible, and you are encouraged to experiment.

Your matching predicate should be a Perl5 subroutine, which takes two axis co-ordinates as arguments. The routine must look up the tokens in these co-ordinates in the appropriate global array (refer to one of the existing matching predicates for a template). It must return 1 if the two tokens are deemed likely mutual translations, and 0 otherwise.

2 Axis Generation

SIMR takes two bitext space axes as parameters. Each axis is a file with one ordinate per line. Each ordinate is a text position (in characters) and a text token, separated by a space. The position of a token is defined to be the mean position of its characters. You can use the `axis` program to generate axes from a tokenized text. Tokenizing the text is the tricky part.

Ideally, tokens should correspond to the smallest semantic units of the language in question. Usually, such units are words. The exact form of the tokens that you should put on each axis depend on your matching predicate. If you are using a POS filter, then your tokens should be POS tagged. If you are using a pre-existing translation lexicon, then make sure that your tokens use

the same character set as the lexicon. If you are using phonetic cognates, then you must represent your tokens phonetically.

On a happier note, SIMR can work without flawless tokenization. When you are not sure about proper tokenization, you can make as many guesses as you like. Nothing prohibits you from putting several tokens in the same axis position. The tokens need not even correspond to words. For example, if you have a German compound word AB, and you're not sure whether A and B should be separate tokens, just put three tokens on the axis: A, B, and AB. Another interesting example is Chinese, where the vast majority of words consist of two, three or four characters. You don't need a Chinese tokenizer to use SIMR — just create one token for each sequence of two, three and four characters. If your matching predicate is based exclusively on a pre-existing translation lexicon, then generating the axes could not be simpler. All you have to do is isolate character strings that appear in the lexicon. All other words could never be "matched" anyway.

3 Re-optimizing SIMR

SIMR has 5 numerical parameters. For optimal performance, these parameters must be re-optimized for every new language pair. They may also need to be re-optimized if the tokenizer for some language is changed, or if the input text genre is very unusual. The four steps in optimizing SIMR are described below.

3.1 Create training bitexts.

The training bitext should not be less than 500 sentences in each of the component languages. You will need two such bitexts, to enable cross-validation. Ideally, the two bitexts should be from different text genres. After selecting bitexts of the appropriate size in the language pair to which you are porting SIMR, follow the instructions for manual alignment or the instructions for online alignment in Appendix A to create two training bitexts.

3.2 Create training bitext maps.

First, create axes out of both training bitexts. The axis-generation routines are called `axis.?`, where the `?` is replaced by a language code letter. At this time, `axis.E`, `axis.F`, and `axis.S` have been implemented, to generate axes for English, French and Spanish texts, respectively. For example, if `L1Train` and `L2Train` are a French file and an English file, you might type:

```
axis.F L1Train > L1Train.axis
axis.E L2Train > L2Train.axis
```

Next, use the axes to construct the training bitext map, like this:

```
aligned2map L1Train L2Train > Train.map
```

The `aligned2map` program converts a line alignment to a character-based bitext map, by noting the character positions at the end of each pair of aligned text segments. Once you have a bitext map, you can double-check it for errors using `ADOMIT` [2].

Make a bitext map from both training bitexts. Pick one map to be the training map; the other map will be your "held-out" map.

3.3 Create optimization environment.

SIMR has been successfully optimized for at least two language pairs using simulated annealing. The generic simulated annealing program sa is available for this purpose. Directions for using sa are contained in the sa code, and reproduced in Figure 1.

Figure 1: *instructions for* sa

```
##############################################################################
# Author:  I. Dan Melamed
# Computes:     optimum parameters using simulated annealing
#               with an exponential inhomogeneous cooling schedule
# Temperature is modulated to produce mean jump probabilities
# ranging from 0.5 down to 0.001.
#
# The first line of the parameter file should name the shell script
#      that generates a model, given a set of parameter values.
# The second line should name the file where this script will store
#      the model.  The file name will have the pid and parameter
#      values appended to it.
# The third line should name the script that will score the model.
#
# Subsequent lines of the parameter file should have one line for each
# parameter, with the following fields delimited by white space:
#        par. name, min, max, step size, print format, seed value
# The 7th and subsequent fields are ignored and can be used for
#        comments, etc.
# Parameters whose seed value is the string "random" will be randomly
#       generated from the specified range.
# Parameters whose min value == their max value are constant and never
#       varied.
# The step size should be an element of {..., 100, 10, 1, 0.1, 0.01, ...}.
#
# The parameter file is rescanned after every scoring, so that the
# search space can be modified in the middle of a run.
# The program remembers the parameter sets it has seen and their
# scores.  Thus, it never has to evaluate the same model twice.  If a
# parameter set is revisited, its score is simply redisplayed,
# followed by "<R>" for "repeat".
##############################################################################
```

Here is how sa might be called, given a parameter file called SIMR.par:

```
sa 20 48 1 .2 SIMR.par > sa.out.SIMR.$$
```

Figure 2 contains an actual parameter file that has been used for optimizing SIMR on Spanish/English. The 5 parameters are exactly those required by SIMR. If you use sa to reoptimize SIMR, you should

have to change only the 1st and 3rd lines of this parameter file. The `SIMR.training.wrapper` is

Figure 2: *sample* `sa` *parameter file*

```
tcsh SIMR.training.wrapper mail.S.axis mail.E.axis lcsrcogp 0
map
monormse SE.mail.map
Csize   7       12      1       "%d"    random  # chainsize
MaxPA   3       12      1       "%d"    random  # max. point ambiguity
MAngD   .05     .3      .01     "%4.2f" random  # max. angle deviation
MaxLRE  10      30      1       "%4.1f" random  # max. linear regression error
MinCLR  .66     .9      .01     "%4.2f" random  # min. cognate length ratio
```

just a script that calls `SIMR`, but simplifies parameter passing for the optimization routine. To use the `SIMR.training.wrapper` for a new pair of languages you only need to update the pointers to the stop-lists and to the pre-existing translation lexicon, if you are using one. `monormse` computes the RMS error between a "control" bitext map and a monotonized version of the input test map. Substitute the name of your training map as the control map parameter to `monormse`.

3.4 Run the optimization.

Run `sa` on your new parameter file to optimize `SIMR`'s parameters. Figure 3 contains the top few lines of a typical sa output file. When the optimization run is finished, find the set of parameters

Figure 3: *top few lines of* `sa` *output, when used with the parameter file in Figure 2*

```
#############################################################
Run #20   T     Csize   MaxPA   MAngD   MaxLRE  MinCLR          Score
#############################################################
48       1.44   "7"     "11"    "0.16"  "17.0"  "0.88"            9.55       Accepted
47       1.38   "7"     "11"    "0.20"  "17.0"  "0.88"            9.58       Accepted
46       1.31   "8"     "11"    "0.20"  "17.0"  "0.88"            9.54       Accepted
45       1.25   "8"     "11"    "0.20"  "17.0"  "0.86"            9.54       Accepted
44       1.19   "8"     "11"    "0.20"  "17.0"  "0.82"            9.45       Accepted
43       1.14   "8"     "11"    "0.24"  "17.0"  "0.82"            9.45       Accepted
42       1.08   "9"     "11"    "0.24"  "17.0"  "0.82"            9.83
40       0.98   "8"     "11"    "0.21"  "17.0"  "0.82"            9.45       Accepted
39       0.94   "8"     "11"    "0.24"  "17.0"  "0.82"            9.45  -R-  Accepted
38       0.89   "8"     "11"    "0.24"  "17.0"  "0.85"            9.46       Accepted
```

that results in the lowest error (in the "Score" column). A quick way to do this is to feed the output of `sa` into `sort +7n | more` . Keep a record of what the actual RMS error was with the optimal parameter set.

3.5 Validate the training bitexts.

Even if the alignment instructions in Section 3.1 are followed to the letter, it is extremely easy to make a mistake while hand-aligning the training bitexts. If the lowest RMS error achieved by the optimization run is higher than 7 or so, then there is a good chance that the training bitexts contain alignment errors[1]. Even one such error completely invalidates the optimization run, because the objective function is extremely sensitive to outliers.

For this reason, the `SIMR` distribution contains a suite of map and text analysis tools. The following is an annotated example of how you can use these tools to track down alignment errors in the training bitexts. It is a session transcript of an actual error-hunting mission that took place during `SIMR`'s optimization on the Spanish/English language pair.

```
Script started on Fri Jul 19 22:11:53 1996
/home/data/ES/aligned >sort +7n sa.out.SIMR.10100 | more
############################################################
############################################################
Run #20   T      Csize   MaxPA    MAngD      MaxLRE     MinCLR      Score
41      1.03     "8"     "11"     "0.22"     "25.0"     "0.79"      9.52          Accepted
23      0.44     "7"     "10"     "0.18"     "28.0"     "0.77"      9.54     -R-  Accepted
24      0.46     "7"     "9"      "0.18"     "28.0"     "0.77"      9.54          Accepted
26      0.50     "7"     "10"     "0.18"     "28.0"     "0.77"      9.54          Accepted
22      0.42     "7"     "10"     "0.18"     "28.0"     "0.78"      9.55          Accepted
27      0.53     "7"     "10"     "0.18"     "28.0"     "0.79"      9.55          Accepted
28      0.55     "7"     "12"     "0.18"     "28.0"     "0.79"      9.55          Accepted
42      1.08     "8"     "12"     "0.22"     "25.0"     "0.79"      9.55          Accepted
11      0.25     "8"     "8"      "0.19"     "29.0"     "0.79"      9.65          Accepted
43      1.14     "7"     "12"     "0.22"     "25.0"     "0.79"      9.71          Accepted
12      0.26     "8"     "9"      "0.19"     "29.0"     "0.79"      9.74          Accepted
14      0.28     "8"     "9"      "0.19"     "29.0"     "0.78"      9.74
28      0.55     "7"     "12"     "0.20"     "29.0"     "0.79"      9.90
--More--
```

The above `sort` of `sa` output shows that the lowest RMS error attained during the optimization was more than 9 — that's a red flag. `sa` does not delete the models that it generates; it just renames them to a file with a `.Zap.0.` prefix. We can take a peek at the details of the optimal model by using the `lineeval` command. This command is like `monormse`, expect that it outputs each error value to `STDOUT`, and the aggregate RMS error only to `STDERR`. Like `monormse`, `lineeval` takes the "control" bitext map as the first parameter. Recall that the naming of `sa`'s models is controlled by the parameter file and the model's parameters.

```
/home/data/ES/aligned >lineeval SE.mail.li \
                        .Zap.0.map.8.11.0.22.25.0.0.79.10101 > temp.dev
No control end-point; test truncated.

Root Mean Squared Error in characters:  9.15
Mean Absolute Error in characters:  3.08
```

[1] Experience has shown 7 to be a reasonable upper limit for French/English and for Spanish/English. A much higher RMS error may be typical for less similar language pairs.

The file `temp.dev` now contains one deviation (error) value per point in the control map. We can find outliers by generating a histogram:

```
/home/data/ES/aligned >bucket 20 0 temp.dev
2       From -36.8 to -30.0     = 0.0036
9       From -30.0 to -10.0     = 0.0160
534     From -10.0 to 10.0      = 0.9502
11      From 10.0 to 30.0       = 0.0196
2       From 50.0 to 70.0       = 0.0036
3       From 70.0 to 90.0       = 0.0053
1       From 130.0 to 137.3     = 0.0018
```

The histogram shows that, at six points, the optimal map deviated from the training map by more than 50 characters. One of the points deviated by more than 130 characters. That's very suspicious for an optimal map. The offending lines can be found like this:

```
/home/data/ES/aligned >numberlines 1 < temp.dev | awk '$2 > 50'
52 84.2413793103442
54 57.6451612903224
150 137.378355256565
151 89.3525725445543
152 77.4234576735658
153 55.9234545545452
```

Now we extract the suspicious line ranges from the training bitexts:

```
/home/data/ES/aligned >sellines 149 154 < mail.S
Por ejemplo, si su ltimo comando de eliminacin fue d 2-5, al
escribir u recuperar los mensajes 2, 3, 4, y 5.
Observe que todas las eliminaciones se hacen permanentes al salir de
mailx mediante el comando q;
es decir, las cartas eliminadas ya no pueden recuperarse.
De cualquier modo, puede salir de mailx con el comando x, dejando
intacto su buzn -como se mencion anteriormente, si sale con x las
cartas ledas se marcarn con una U, las cartas eliminadas se
recuperarn, etc.
Cmo imprimir cartas
/home/data/ES/aligned >sellines 149 154 < mail.E
If you want to undo your last deletion, just type u at the mailx
prompt immediately after the deletion.  For example, if your last
deletion command was d 2-5, typing u will undelete messages 2, 3, 4,
and 5.
Note that all deletions are made permanent when you quit mailx with
the q command;
that is, deleted letters can no longer be undeleted.
You can, however, quit mailx with the x command, leaving your mailbox
intact
```

```
- as mentioned previously, quitting with x will leave read letters
marked with a U, deleted letters undeleted, and so forth.
Printing Letters
```

Sure enough, these line ranges were not aligned correctly. Where the Spanish "Por ejemplo" follows a linebreak, the English "For example" does not. Conversely for the English "- as mentioned previously" and the Spanish "-como se mencion anteriormente".

If you discover errors like this, you must correct your training bitexts and restart the optimization process from the beginning.

3.6 Cross-validate.

The last step is to validate the optimized parameters, to make sure that `SIMR` has not been overtrained. Use `SIMR` to find a bitext map for the held-out bitexts, and compare them to the held-out bitext map using `monormse`. If the RMS Error on the held-out data is within a couple of points of the "Score" of the optimal parameter set on the training data, then the parameters generalize reasonably well and can be relied upon. Otherwise, you need to retrain on a different training set. You can either switch the training and held-out data sets or start from scratch, with a bigger bitext. There is no guaranteed method for optimizing this kind of parameter set.

References

[1] I. D. Melamed, "A Geometric Approach to Mapping Bitext Correspondence," *Conference on Empirical Methods in Natural Language Processing*, Philadelphia, U.S.A, 1996.

[2] I. D. Melamed "Automatic Detection of Omissions in Translations," *Proceedings of the 16th International Conference on Computational Linguistics,* Copenhagen, Denmark, 1996.

[3] I. D. Melamed, "Automatic Evaluation and Uniform Filter Cascades for Inducing N-best Translation Lexicons," *Proceedings of the Third Workshop on Very Large Corpora*, Boston, MA, 1995.

Appendix A

The following two sets of instructions are written so that they can be handed to a bilingual annotator.

Instructions for Manual Alignment of Parallel Texts

We are experimenting with automatic ways to analyze translations. One of the first steps in the analysis is to find corresponding segments between an original text and its translation. We have developed a method to find such corresponding segments automatically. However, the method must be fine-tuned before it can be applied. In order to fine-tune the automatic method, we need a bilingual person like you to hand-align the corresponding segments between a text and its translation. The automatic method will then fine-tune itself, by checking itself against the "gold standard" that you provide.

We will supply you with a text in two languages. We recommend that you read these instructions at least twice before starting work, and ask us any questions that come up. It will probably take you several hours to complete the work. Be sure to use a pencil, because you will probably need to make corrections.

Segment the texts.

Start with one of the two texts. Read through it, looking for text units that are likely to be preserved in translation. Good candidates include section headings, list items, sets of computer commands and whole sentences. Mark the beginning and the end of each text unit with a slash (/). When two text units follow each other, you should end up with two slashes between them.

Now do the same thing in the other text.

Find corresponding segments.

This is the hard part.

Place the two texts side by side.

For each text unit that you marked in one text, try to find a corresponding text unit in the other text. Label corresponding units by writing their number above the text unit in both texts. Start with the number "1" and use consecutive numbers up to however many corresponding text units you find.

If, for some unit that you marked in one text, you do not find a corresponding unit in the other text, you must unmark the unit in the first text by erasing the slashes around it. The goal is to end up with the same number of slashes in both texts, and the same number of numbered text units.

Accuracy is more important than the number of text units that you find. If you are not sure whether a particular text unit has a good match in the other text, then don't mark it.

Translation is often not "one-to-one." You will likely run into the following situations:

conflations

Sometimes two sentences in one text will be translated into one sentence in the other text. When this happens, you must re-mark the two sentences in the first text as one text unit, by removing the slashes between the two sentences.

omissions

Sometimes a translator will forget to translate a piece of text. If you see that one of the text units that you marked in one text is missing from the other text, you should pretend that the corresponding unit in the other text is "empty." I.e. you should just put two slashes in the other text where the "empty" text unit would go if it weren't empty. Thus, you might end up with four or more slashes in a row.

inversions

Sometimes text units switch places during translation. It is important for our algorithm that the corresponding text units appear in the same order in both texts. Therefore, whenever you see two text units in one text that are in the opposite order from how their translations appear in the other text, you must mark the two text units as one unit, in both texts. It doesn't matter if things are not in the same order inside two corresponding text units, as long as the units themselves are in the same order.

replacements

Sometimes a text unit will not be translated but will be simply replaced with something else. You may see a pattern like $(A, B, C) \rightarrow (a, x, c)$, where A is translated as a and C is translated as c, but B is simply replaced with x, where x is not a translation of B. If you see this pattern, mark the segments A, a, C, and c.

Thank you for your help with this project.

Instructions for Online Alignment of Parallel Texts

We are experimenting with automatic ways to analyze translations. One of the first steps in the analysis is to find corresponding segments between an original text and its translation. We have developed a method to find such corresponding segments automatically. However, the method must be fine-tuned before it can be applied. In order to fine-tune the automatic method, we need a bilingual person like you to hand-align the corresponding segments between a text and its translation. The automatic method will then fine-tune itself, by checking itself against the "gold standard" that you provide.

We will supply you with a text in two languages. We recommend that you read these instructions at least twice before starting work, and ask us any questions that come up. It will probably take you several hours to complete the work.

We will assume that you have access to a word-processor that can display the two texts side by side.

Set up.

Start up your word-processor (WP). Make it display the two text files side by side.

Since at least one of the files is not English, make sure that your WP can properly display all the characters, or at least a large enough fraction to make the text legible.

If this is not already your default, make your WP display the cursor row-position for both texts. In emacs-based WP's (mule, jove, etc.), the command is M-x line-number-mode.

If your WP is set up to do automatic line filling (a.k.a. word wrapping), then turn if OFF. In emacs-based WP's (mule, jove, etc.), the command is M-x auto-fill-mode.

Segment the texts.

Start with one of the two texts. Read through it, looking for text units that are likely to be preserved in translation. Good candidates include section headings, list items, sets of computer commands and whole sentences. Put each text unit on one line by adding or deleting carriage returns where necessary.

Now do the same thing in the other text.

Align corresponding units.

This is the hard part.

Look for corresponding text units in the two texts, and place them on the same line number, by adding or deleting blank lines where necessary. You will need to pay attention to the cursor row-position display to do this. It's OK if a text unit is longer than the width of your WP window – don't add carriage returns to break it up. The goal is to end up with the same number of text units in both texts. Therefore, you should end up with the same number of lines in both files.

Accuracy is more important than the number of text units that you find. If you are not sure how to break up a larger text unit, then don't break it up.

Translation is often not "one-to-one." You will likely run into the following situations:

conflations

Sometimes two sentences in one text will be translated into one sentence in the other text. When this happens, you must treat the two sentences in the first text as one text unit.

omissions

Sometimes a translator will forget to translate a piece of text. If you see a text unit in one text that is missing from the other text, you should pretend that the corresponding unit in the other text is "empty." I.e. you should just leave a blank line on the same line number in the other text where the "empty" text unit would go if it weren't empty. The blank line should have nothing on it but a carriage return.

inversions

Sometimes text units switch places during translation. It is important for our algorithm that the corresponding text units appear in the same order in both texts. Therefore, whenever you see two text units in one text that are in the opposite order from how their translations appear in the other text, you must treat the two text units as one unit, in both texts, by keeping both on the same line. It doesn't matter if things are not in the same order inside two corresponding text units, as long as the units themselves are in the same order.

replacements

Sometimes a text unit will not be translated but will be simply replaced with something else. You may see a pattern like $(A, B, C) \rightarrow (a, x, c)$, where A is translated as a and C is translated as c, but B is simply replaced with x, where x is not a translation of B. If you see this pattern, align A with a, B with b, and C with c. Do NOT treat replacements like two-sided omissions.

Thank you for your help with this project.