NERT 4.0 - Training Material | WP3

Author: Mathieu Fannee, Frank Landsbergen. INL Internal Review: Katrien Depuydt – November 2013.

CONTENT

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

Background

NERT is a tool that can mark and extract named entities (persons, locations, organizations or even titles) from a text file. It uses a supervised learning technique, which means it has to be trained with a manually tagged training file before it is applied to other text. In addition NERT comes with a named entity matcher module, with which it is possible to group variants or to assign modern word forms of named entities to old spelling variants.

Note that NERT basically is an extension to the Stanford package. If the named entity matcher module is not required, one could consider using the standard distribution of the Stanford package, which is also in the list of tools endorsed by Succeed.

NERT has been extended for use in IMPACT (http://www.impact-project.eu/). Among the extensions is the aforementioned matcher module, and a module that reduces spelling variation within the used data, thus leading to improved performance.

For more information on the working of the Stanford tool, see Finkel, Grenager and Manning (2005) or visit the tool's website: http://nlp.stanford.edu/software/CRF-NER.shtml. The Stanford tool is licensed under the GNU GPL v2 or later.

Technical notice

Note that some JAVA classes of the Stanford software appear to have some visibility problems in its last version 3.2.0. That is, the visibility of some constructors in the *edu.stanford.nlp.ie.crf* package wasn't set in some cases, with default limited visibility as a result. This problem have been solved in the Stanford code versions delivered with NERT.

NERT requirements

NERT is a Java application and requires Java 1.7.

The NERT package

NERT consists of a directory with the tool itself, example data and scripts:

- NERT
- data
  - matcher
  - models
  - phontrans
  - props

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

- sample_extract
- sample_train
- doc
- out
- scripts
- tool

Figure 1: contents of the NERT package

The NERT software is to be found in the 'tool' directory, which contains a file *nert4-0.jar*. We will explain how to use it in the following sections.

Extracting named entities with NERT

At the very least, three files are needed for NE-extraction with NERT. If you have those three, you are ready to go:

1) a tagged 'training file'
2) a properties file

These first two files constitute the actual *knowledge* en *settings* of the tool. And at last:

3) a tagged or untagged 'target file', from which NEs will be extracted

'Tagged' means that all NEs in the file have been tagged. The target file can be either tagged or untagged. If it is tagged, it is possible to calculate the tool's performance with the 'conlleval' script from the CONLL conferences (provided that the output is set to BIO-format, see below). This script can be downloaded at http://www.cnts.ua.ac.be/conll2000/chunking/output.html. However, note that for the actual extraction of NEs, tags in the target file are not necessary.
The properties file consists of a list of features, parameter settings and locations of the necessary files. This file will be discussed below. In the directory `data/props`, an example of such a properties file is included.

Example of use

The script `run_nert.sh` in the `scripts` directory can be used as an example for all the tools capabilities:
- It trains a model with Dutch example data using the properties file from the directory

`data/props`.

- It then uses its model to identify NEs in a target file.

Stanford is a statistical NE-tool. This means it needs to be trained on tagged material, which is what the training file is for. For good performance, it is key to train on material that is as close to the actual target data as possible in terms of time period and genre (so, train on newspapers to be able to parse newspapers, but don't train on newspapers to parse business letters). More information on how to create training and target files is given below.
Training and extracting are two separate commands. After training, the tool produces a classifier

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

('model'), which is stored as a file. This model can then be used for extracting at any later stage. Training the model is done by running the jar file *nert.jar* in the directory `tool` with the following command:

Training:
```
$ java -jar nert4-0.jar -t -props [properties file]
```

If necessary, memory can be increased as follows:

```
$ java -mx4000m -jar nert4-0.jar -t -props [properties file]
```

4000MB should be enough for the training of the model, but, if necessary and available, more memory can be used as well. When the tool does not successfully create a model during training, insufficient memory might be a reason.
The properties file gives the tool the location of the file or files it has to train with, parameter settings and the location where to write its model to (see below for more detail).
In the examples below, nert4-0.jar is called from the main directory. Note that the paths to all files in the training, extraction and matching examples are relative, so beware that the paths are correct.
Basic extraction with BIO-input and BIO-output is done as follows:

```
$ java -jar tools/nert4-0.jar -e -loadClassifier [model] -testFile [testfile]
```

We experienced cases in which the tool crashed during extraction, and this had to do with an out-of-memory error that was solved by increasing memory (similar as that for the training process). The '-loadClassifier' and '-testFile' (or 'testDir', see below) arguments are compulsory. There are several optional extraction settings that can be added, and that will be discussed below:

```
$ java -jar tools/nert4-0.jar -e -loadClassifier [model] -testfile [testfile]
-in [txt/bio/xml] -out [txt/bio/xml] -nelist [file] -xmltags [tags] -starttag
[tag] -endtag [tag] -sv -svphontrans [file] -svlist [file]
```

NERT sends its output to STDOUT. Again, a higher amount of memory can be used as well. For extraction, a properties file is not needed. In principle, the settings from the training file will be passed on through the model. A set of relevant parameter settings can be passed to the tool via the command line. They will be discussed in the next section.

Settings

Input and output files

For training, one or more files or a reference to a directory with relevant files can be used, and the path has to be given in the properties file. There are three options:

```
trainFile=FILE
trainFiles=FILE1;FILE2;FILE3
trainDirs=DIR
```

For extraction, a single file or a reference to a directory can be used in the command line:

```
$ [ … ] -testfile [target file]
$ [ … ] -testDirs [directory]
```

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

Note that NERT prints the results to standard output. This means that when using a directory, all files within this directory are printed subsequently, as a whole. In order to be able to distinguish the original target files, NERT starts the output of each target file with a print of the filename when the flag '-testDira is used'.
NERT can create a file that contains a list of all found NEs with the following command:

```
$ [ … ] –NElist FILE
```

Input and output formats

NERT 2.0 (and higher) can handle three text formats: BIO, text and xml. As default, it expects BIO-format as input, and it will use this for output as well. When you are using files in text or xml format or you want a particular output in the extraction process, you need to tell NERT:

- training, in the properties file: `format=txt/xml/bio`
- extracting, on the command line: `–in bio/txt/xml –out bio/txt/xml`

BIO-format

'BIO' is an acronym and stands for the kind of tags used: B(egin), I(nside) and O(ut). Basically, each word is on a separate line, followed by the tag:

```
Arjen POS B-PER
Robben POS I-PER
should POS O
have POS O
scored POS O
against POS O
Spain POS B-LOC
. POS O
```

The middle POS tag is optional; it is not used by the tool. However, if you leave it out, it is necessary to tell the tool in the properties file the structure of your bio-input:

Default:

```
format=bio
map= word=0,tag=1,answer=2
```

without the POS-tag:

```
format=bio
map= word=0,answer=1
```

It is recommended to add a whitespace after each sentence, and tokenize your data so that periods, commas, etc. are on separate lines instead of being glued to the end of a word, since this improves performance.
If the BIO-format is needed, the script `tag2biotag.pl` in the `scripts` directory can be used. For input, it needs a text file with each word on a new line, and NEs tagged as <NE_PER|ORG|LOC>Named Entity</NE>, e.g.:

```
<NE_PERArjen Robben</NE>
should
have
scored
against
<NE_LOC>Spain</NE>
```

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

Txt-format

NERT can also handle text format, in which the tags are wrapped around the NEs:

```
<NE_PER>Arjen Robben</NE> should have scored against <NE_LOC>Spain</NE>.
```

Again, NERT needs to know which format you are using, both in training and extraction:

- training, in the properties file: `format=txt`
- extraction, on the command line: `-in txt`

With text format, NERT expects the tags in the example above as default: <NE_PER>JOHN</NE>. If different tags are used, these need to be specified. In this specification, the actual tag (e.g. PER, LOC, or ORG), is represented by the word 'TAG" (in capitals):

- training, in the properties file:

```
starttag=<NE TAG>   #for <NE PER>, <NE LOC>,
                                        <NE ORG>
endtag=</TAG>#for </LOC>, </PER> etc.

 starttag=<NE type="TAG">  #for<NE type="PER">,
                      possibly followed #by
                      attributes, e.g. <NE
                      type="PER" #id="2>
```

- extraction, on the command line:

```
-starttag '<NE TAG>' or -starttag <NE type="TAG">'
-endtag '</TAG>'
```

If a wrong starttag and/or endtag is given, NERT will most likely crash.

In extraction, when a text file is given that has tags, NERT will use the structure of these tags for its own output, while marking the original reference tags with the tags <REF_ORG>Timbouctou</REF>. For example:

```
<PER>John</PER> and <PER>Yoko</PER>
```

with starttag '<TAG>' and endtag '</TAG>' will be outputted as:

```
<PER><REF_PER>John</REF></PER> and <PER><REF_PER>Yoko</REF></PER>
```

in which the inner tags represent the original tags and the outer tags the ones supplied by the NERT.

As a final note, although NERT is trying to preserve the original outline of a text document, there will most probably be differences in the output of whitespaces.

Xml-format

When using xml-format, the same principles apply as for txt regarding the tags. NERT deals with xml input, provided that it is told to consider only text between specific tags. Say we have the xml-file below:

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

```
<?xml version="1.0" encoding="UTF-8"?>
  <…>
    <…>
      <Text>
        Sally sells sea shells at the sea shore.
       <Text>
      <Text>
        Peter Piper picked a pack of pickled peppers.
       <Text>
    </…>
  </…>
</xml>
```

We have to tell NERT to only consider the text between the <Text> tags. This is done as follows:

- training, in the properties file: `xmltags=Text`
  or with multiple tags: `xmltags=Text;Unicode;`
- extracting, on the command line: `-xmltags Text`
  or with multiple tags: `-xmltags 'Text;Unicode'`

NERT deals with XML by simply skipping all text that is not between the specified tag(s). The relevant chunks are considered subsequently. Note that this means that in the above example, it will first train/extract the first sentence and then the following. Any NEs that would be stretched over these two chunks, would therefore be missed. Thus, the xml-format is recommended only when large chunks of text are covered by a specific tags. In other cases, it is necessary to convert the text to either text- or BIO-format.

The spelling variation reduction module

In training, NERT learns to recognize NEs by trying to identify relevant clues about both the NEs and their context. Examples of clues are use of capitals, position in the sentence or preceding or following words or groups of words (such as *in* + LOCATION).This means that the tool is sensitive to variations in the spelling of words. For example, the sentences *I come from London*, *I come fro London* and *I come frcm London* all have different words preceding the location *London* for the tool, although they are all (made up) variants of the word *from*. Thus, the tool would benefit if these variations would be diminished, and this is what the spelling variation reduction module intends to do.

The module tries to reduce spelling variation on the input data by matching potential variants, creating internal rewrite rules and by executing these rewrite rules before the tool actually uses the input. The actual output remains unchanged. In the above example, it would identify the words *from*, *fro* and *frc* as variants and create the rewrite rules *fro=>from* and *frc=>from*. These rewrite rules are applied to the input data, the tool is ran, and, in the case of extraction, the original text is used for output.

In extraction, the module looks in both the target file, the words from the original training file and, if present, gazetteer lists (which are all stored in the used model). For example, if a model has been trained with the word *fro*, it pays to create a rewrite rule in which variants of this word in the target file are rewritten to *fro*. Similarly, if the gazetteer lists contain the location *London* while the target file has the location *Londen*, a rewrite rule *Londen=>London* is created, thus enabling the tool to recognize *Londen* as a gazetteer.

The module works by transforming all words to a phonetic transcription and by comparing these versions of the words with each other. Words with the same phonetic transcription are considered variants.

This means that the rules for phonetic transcription are crucial for a proper working of this

module. The module has a set of default rules, but the user can load its own set if needed:

- training, in the properties file: `useSpelvar=true`
  `svPhonTrans=FILE`
- extraction, on the command line: `-sv –svphontrans FILE [ … ]`

The arguments 'useSpelvar=true' and '-sv' are the ones that initiate the spelling variation reduction module.
The rules are read by the tool and used in a simple Java *replaceAll* function. Thus, regular expressions can be used in them, but this is not necessary:

```
sz=>s
sz\b=>s
\w=>
\bcometh\b=>come
```

Before the module applies the rules, each word is put in lowercase, so only lowercase characters should be used on the left hand side of the rules. The first example rule tranforms all occurrences of 'sz' to 's'. The second uses '\b' which means it will only consider 'sz' at word boundaries. The third example rule replaces all non-word characters with nothing, thus removing them from the string. One can also use the rewrite rules to replace (or remove) complete words.
For each word, the rules are applied one by one, in the order of the file they are in. It is important to consider this order: sz=>s after the rule z=>s is useless, because all 'z' will already have been removed from the string.
Tests on Dutch historical data have shown that the module is capable of improving the scores up to a few procent. However, having the proper rewrite rules is key here. We found that more rules did not necessarily lead to better performance, due to the fact that more rules lead to more wrong variant matches. In general, the following advice can be given:

- Remove non-word characters such as dashes, whitespaces, commas and periods (\w=>)
- Check the data for commonly occurring variations. For example, Dutch 'mensch' vs. 'mens', and 'gaen' vs. 'gaan'.
- Check the effect of the rewrite rules. 'sch=>s' would work for 'mensch' but would also wrongfully change 'schip' (ship) into 'sip'. 'sch\b=>s' works better but skips the plural 'menschen'.
- Focus on words that identify named entities, such as prepositions and titles. For example, Dutch 'naer' and 'naar' (to). For those words, it pays to write a specific rule, e.g. '\bnaer\b=>naar'.

Regarding the latter remark, a script `find_NE_identifiers.sh` is added to the scripts directory, which can be used to help identifying useful words. When run on a text (in BIO-format) in which the NEs are tagged, like the training file, it lists all words preceding the NEs. These preceding words are often important predictors for NEs, and performance generally improves when reducing the amount of variation in them. The list will generally contain many prepositions and titles. The script is run as follows:

```
$ sh find_NE_identifiers.sh [file] > [outputfile]
```

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

NERT can print a list of created rewrite rules (variant=>word) to a file when using the following command:

- training, in the properties file: `printspelvarpairs=FILE`
- extraction, on the command line `-svlist FILE`

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

Creating training, target and properties files

Training and target files

A first step is to select and produce an appropriate training file. NERT's performance depends strongly on the similarity between the training file and the test file: when they are exactly alike, the tool can reach an f1-score of 100%. Generally speaking, the more different both files are, the lower the performance will become (although other factors also affect the tool's performance). We therefore recommend using part of a particular batch of texts for training. That is, if you have a 1 million words dataset of 19th century newspapers and 1.5 million words dataset of 18th century books, we recommend to keep them separate and to create two training files.
The size of the training file affects performance as well: the larger, the better. Below the f1-scores for a training file of ~100,000 words on different Dutch text types are shown to give an indication (table 1). The parliamentary proceedings score best, because OCR-quality is good, but mainly because it is a very homogeneous text type.

Dataset	*Time period*	*OCR -quality*	*time period*	*f1-score*
prose, poetry, plays, non-fiction	18th c.	n/a	18th c.	70.80
	19th c.	n/a	19th c.	78.68
Parliamentary proceedings	19th c.	okay	19th c.	83.31
	20th c,	okay	20th c.	88.50
various Dutch newspapers	18th c.	poor	18th c.	73.49
	19th c.	poor	19th c.	83.92

Table 1. F1-scores of various datasets with a training file of ~100,000 words, without the use of the spelling variation module.

Another way of giving the training file a better coverage of the target file is to randomly select sentences from the data. We found that this method leads to a better performance then when, for example, the first 100,000 words from the data is used for training and the rest for testing. The script `splitFiles.pl` in the `scripts` directory can be used to create such a random set of sentences. For input it needs a text file with each sentence beginning on a new line and the desired number of words. It then creates two output files, one with the desired number of words and one with the remaining text. These files can then be used as training and target files.

```
$ perl splitFiles.pl [textfile] [number of words of output file 1] [num]
```

The third argument [num] is the total number of files that are created. Use 1 to create 1 training file and 1 target file. The script `splitFiles_BIO.pl` works the same as `splitFiles.pl`, but uses a file in BIO-format as input.
For the tagging of the training file we used the Attestation Tool from deliverable EE2.4, but other tools can of course be used as well. In the documentation of the current deliverable EE2.3, a

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

document with NE-keying guidelines is included that can be useful. Although it is written for use with the Attestation Tool, its guidelines are generally applicable.
If the BIO-format is needed, the script `tag2biotag.pl` in the `scripts` directory can be used. For input, it needs a text file with each word on a new line, and NEs tagged as <NE_PER|ORG|LOC>Named Entity</NE>.

Improving data

When using OCR'd data, tool performance on person names generally increases when the training and target files are cleaned up a bit. Generally, the main things to look out for are 'errors' due to faulty OCR and tokenization as shown below.

Where *Is* *Dr* *.* *Who* *?*	should be	*Where* *Is* *dr.* *Who* *?*
O. *J* *.* *Simpson*	should be	*O.* *J.* *Simpson*
The *New* *TOM* *W* *A* *I* *T* *S* *Al* *Bum*	should be	*The* *New* *Tom* *Waits* *album*

The NER-package comes with a few Perl scripts that can fix most of the above, but it is always a good idea to double check the results. Note also that using these scripts affects your source text. The scripts work with BIO-text input and print in standard output. The scripts can be used as follows:

```
$ perl convertToLowercase.pl < [BIO-file]
```

changes all CAPITALIZED WORDS to words with Initial Capitals

```
$ perl fixInitials.pl < [BIO-file]
```

detects periods that are preceded by a single capitalized letter and a whitespace, or words listed in the script ('mr', 'Mr', 'dr', 'Dr', 'st', 'St', 'ir', 'Ir', 'jr', 'Jr', 'wed', 'Wed').

```
$ fixAbbrev.pl < [BIO-file]
```

a script specific for Dutch: changes 'v.' to 'van' and 'd.' to 'de'

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

Creating a properties file

A properties file consists of a list of features, parameter settings and locations of the necessary files and a link to its location should be added as an argument when training the model. An example properties file can be found at `data/props/`. Below, the contents of a properties file are shown, with a short description of the most important features:

```
trainFile=[path and name of single training file]
trainFiles=[training file1;training file2;training file3]
trainDirs=[directory with training file]
serializeTo=[path and name of the model that will be created]
map= word=0,tag=1,answer=2                    # structure of the BIO-format

useSpelVar=true                               # use any of the spelvarmodules below
svphontrans=[path and name of file]           # file with phonetic transcription
                                              # rules
printSpelVarPairs=[path and name of file]     # print all created and listed
                                              rewrite
                                              # rules to file


useGazettes=true                              # use gazetteers listed below
sloppyGazette=true
gazette=[path to list1;list2;list3; …]        # location of gazetteer lists

format=[txt/bio/xml]                          #input format. Default=bio
starttag=<NE_TAG>                             #shape of NE-tags in txt, xml format
endtag=</NE>
xmltags=[tag1;tag2;tag3]                      #relevant xml-tags. Leave out <>
```

#the following features can be left like this:

```
noMidNGrams=false
useDistSim=false
useReverse=true
useTitle=true
useClassFeature=true
useWord=true
useNGrams=true
maxNGramLeng=6
usePrev=true
useNext=true
useSequences=true
usePrevSequences=true
maxLeft=1
useTypeSeqs=true
useTypeSeqs2=true
useTypeySequences=true
wordShape=chris2useLC
useDisjunctive=true
```

Note: in order for the spelling variation reduction module to work properly, ‘useWord=true’ is necessary, and if gazetteers are used, ‘sloppyGazettes=true’ is necessary as well.

Using the NERT named entity matcher module

The matcher module gathers named entities (NEs) and arranges them in distinct groups of spelling variants. Second, the matcher can assign modern word forms of NEs to old spelling

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

variants.
So, the matcher module matches variants of NEs, such as *Leijden* and *Leyden*, and it can also be used to match historical spelling variants of NEs to their modern form, such as *Leyden* to *Leiden*. It does so by comparing phonetic transcriptions of NEs, and calculating the distance between them by breaking them up in chunks and by calculating the number of chunks two NEs have in common. This value is then corrected for string length and normalized on a scale from 0 – 100, with 100 being a perfect match.
Phonetic transcription takes place on the basis of a set of rules, which have to be given to the matcher. Examples of phonetic transcription are /mastrigt/ for the NE *Maastricht* and */*franserepublik*/* for *Fransche Republiek*. NERT comes with a set of default rules that have proven to work well for Dutch. However, for other languages, (some of) the rules might have to be altered.

Using the matcher

You can tell NERT to start the matcher by using the *–m* flag as a first flag, and use the *–props* flag to tell the matcher the location of a properties file. This properties file holds the values of a set of parameters and the location of all relevant files.

```
$ java -jar tools/nert.jar -m -props propsfile.props
```

The matcher needs the following data:

- One or more files with NEs (format: one NE on each line)
- A properties file
- For lemmatizing: one or more files with NEs (format: one NE on each line)
- A file with phonetic transcription rules (optional)
- A file with surname identifiers for person names (optional)
- A file with transcription rules for roman numbers (optional)

The exact use of this data and all possible settings in the properties file are discussed below.

Examples

Say we have a single file with NEs and we would like the matcher to group all NEs within that file that are variants. The file is /myfiles/NE/NE-file.txt. In the properties file we then put the following:

```
file=/myfiles/NE/NE-file.txt
```

If you have your NEs in more than one file, they can be referred to by their directory:

```
dir=/myfiles/NE
```

If you want your NEs in NE-file.txt not to be matched to each other, but to NEs in a different file, e.g. NE-file2.txt, you can use the ‘lemmaFile’ or ‘lemmaDir’ option:

```
file=/myfiles/NE/NE-file1.txt
lemmaFile=/myfiles/lemmata/NE-file2.txt
```

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

The matcher's output will be the NEs from NE-file1.txt, with their possible variants from NE-file2.txt.
The matcher can be told in which column in the input to look for the relevant data:

```
line=type:0,ne:1
lemmaLine=type:2,ne:3
```

The first line indicates that in the general file(s), the type of NE can be found in the first column and the actual NE in the second. The second line indicate that in the lemma file(s), the type is in the third column and the NE in the fourth. The matcher prints all output preceding the first indicated column.
The option 'ignoreWordsWithTag' can be used when you would like the matcher to ignore parts of an NE's string:

```
ignoreWordsWithTag=%
```

For example, in the NE *Jean Philippe d'Yvoy %Baron% van Ittersum tot Leersum*, the matcher will ignore the part *Baron*. It is important that both opening and closing tags are used, otherwise the ignore-option will be skipped.

Output options

The Matcher outputs only those files that are listed in the option 'onlyShowFile', and this can deviate from the actual input:

```
dir=/dir_A
onlyShowFile=/dir_A/file-A
```

This is particularly useful if we would like to have the matcher group variants from a set of lists, but we are only interested in the output of one of them. If you want the output of more than one file, use 'onlyShowFiles', with semi-colon separated filenames.
The NE matcher has different ways to print its output. The default output is as follows:

```
Amsteldam     Amsterdam
Amsteldam     Amstelredam
Amstelredam   Amsteldam
Amstelredam   Amsterdam
Amsterdam     Amsteldam
Amsterdam     Amstelredam
```

This is called the 'pairview' output, since each line shows 1 pair of NEs. If you rather want the matcher to list all variants of a single NE per line, use the groupview flag in your properties file:

```
groupview=true
```

This will print:

```
Amsteldam     Amsterdam     Amstelredam
Amstelredam   Amsteldam     Amsterdam
Amsterdam     Amsteldam     Amstelredam
```

The flag 'showScores' can be used to let the NE matcher also print the matching scores for each variant. 'showScores=true' in the properties file gives:

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

```
Leeuwarden   Leewarden (100)     Gemeente Leeuwarden (100) Lieuwarden (76)
```

The flag 'showPhoneticTranscription' can be used to have the NE matcher print the actual phonetic transcription used in the matching process. For example:

```
Braddock [bbrraddokk]     Bradock [bbrraddokk]      Braddocke [bbrraddokk]
```

By default, the NE matcher shows all matches with a score higher than or equal to 50. Generally, scores lower than 70-75 will contain many false positives, so you can alter the minimal score by using *minScore* in the properties file:

```
minScore=75
```

Note that it might be a good idea to use a minimal score that is not *too* high, since it is harder to filter out false positives than to figure out the false negatives, that is, the matches it has overlooked. The matcher's score can be used to quickly track the false positives.
You can also tell the matcher to only print the *N* best scores. For this, use the following flag:

```
nBest=5
```

The matcher looks at both the settings of minScore and nBest. Say we have a word with 8 matches with scores 100, 100, 80, 80, 80, 75, 75 and 50. With minScore = 50 and nBest = 2, we only see the first 2 results. With minScore = 80 and nBest = 8, we only see the first 4 results, because scores lower than 80 are not considered.

- use minScore = 0 and any nBest > 0 to always show the *N* best results, regardless of their score
- use nBest = -1 to limit the matches to any minimal score

The option 'showDoubles=false' can be used to have the Matcher only print out unique NE's and their matches.

Types

The matcher can also handle NE-types (e.g. LOC, ORG, PER). For this, it needs its input in the following way,

```
LOC Amsterdam
LOC Leeuwarden
PER Piet Jansen
```

with NE-type and NE separated by a whitespace. You need to tell the matcher that you're having types in your input file(s) by stating the following line in your properties file:

```
hasType=true
```

Note that this only tells the matcher *how* to read the input files. The matcher will still match all NEs, regardless of their type. If you want the matcher to match only PERs with PERs and LOCs with LOCs, use the following:

```
useType=true
```

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

By default, the types will disappear in the matcher's output, but you can tell the matcher to print them anyway by adding the following line to the properties file:

```
printTypes=true
```

This will print:

```
LOC AmsterdamLOC Amsteldam
```

Finally, the *verbose* flag can be used for some more general output to STDERR. The flag *punishDiffInitial* is used to punish the matching scores of NEs that do not start with the same character. Its value is subtracted from the final score. The default value is 10. The flag *perFilter* (default value: *true*) sets the use of the PERfilter, which tries to handle person names more intelligently (see explanation above).

Phonetic transcription rules

As mentioned earlier, the matcher uses default rules to convert each NE to a 'phonetic transcription'. These rules can be overridden by supplying the matcher with a file with other rules, and be putting the path to this file in the properties file:

```
phonTrans=/myfiles/phonTransRules.txt
```

The rules are simple rewrite rules which Java applies to each NE one by one with a single 'replaceAll' method. For example, look at the following two rules:

```
ch=>g          # replace any /ch/ with /g/
d\b=>t         # replace any /d/ at a word boundary with /t/
```

Before the matcher applies these rules, the string is converted to lowercase. For example, if the above rules are applied, the NE *Cattenburch* becomes */cattenburg/* and *Feijenoord* becomes */feijenoort/* .
Since the matcher goes over the applied rules one by one, it is important to take the order of the rules into account. Consider for example:

```
z=>s
tz=>s
```

The latter of the two rules will never be used, since all z's are already turned into /s/ because of the first rule. The rules can also be used to simply remove characters or whole substrings from the NE, e.g.:

```
gemeente=>     #       replaces 'gemeente' with '' (void)
/W=>           #       replaces all non-word characters with '' (void)
```

NERT comes with an example file with the phonetic transcription rules for Dutch in the `matcher` directory. Note that these rules do not have to be passed to the matcher because they are the default rules.

Dealing with person names

With the exception of those strings that the matcher is told to ignore (with the phonetic transcription rules), it uses the entire NE for matching. For person names, this might easily lead to false negatives for names such as *Kurt Vonnegut*, *Vonnegut* and *Kurt Vonnegut, jr.*, because of

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

the differences in string length.
The matcher has a built-in option to try and do a simple estimation of the structure of person names, and thus, to recognize that *P. de Vries*, *Piet de Vries* and *Pieter Cornelis Sebastianus de Vries* are (possible) variants. This option is set by the following flag:

```
perFilter=true
```

This is done by letting the matcher look for possible clues for surnames. In the given example, the word *de* is such a clue, and the matcher will consider all names preceding *de* as given names and all names following *de* as surnames. The given names are abbreviated and only the initial(s) is/are used in matching. Thus, the three examples above are reduced to *P de Vries*, *P de Vries* and *PCS de Vries*. The matcher will try to match the names by their surname first. If it finds a match, it will then look at the initials. If these match as well, it will assume that we are dealing with a variant. In this strategy, *P de Vries* and *PCJ de Vries* match, but *P de Vries* and *J de Vries* do not, while *de Vries* matches with any of the above mentioned NEs by lack of an initial.
A list of these signalling words can be added in a file and given to the matcher:

```
surnameIdentifiers=FILE
```

With the file containing a simple list of words, one on each line. An example file for Dutch in the `matcher` directory. If the matcher cannot find any clue as to which is the surname, it will only consider the last word of the NE and use this for matching. This is also the case when the perFilter is used but no file is specified (e.g. ‘perFilter=true’ and ‘surnameIdentifiers=’ or without the entire latter line).
The perFilter gets into trouble with person names such as *Johannes X* or *Frederik de 2e*, since the matcher will only use *X* and *2e* as its matching strings because of the word *de*. For this reason, the matcher checks the NE for use of roman numbers first. If it finds any, it will consider the first name instead of the last.
Note that *Frederik de 2e* and *Frederik de Tweede* should also be considered this way. For this reason, the user can provide the matcher with a file containing rewrite rules for words and their roman counterparts, such as tweede=>II:

```
string2roman=FILE
```

As for the surname identifiers, an example file for Dutch in the `matcher` directory. If string2roman is not specified or left empty, the matcher will still find roman numbers but not the ones that are spelled out.

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.

License and IPR protection

The tool is produced at the Instituut voor Nederlandse Lexicologie (INL) in Leiden, Netherlands. It is licensed under the same license as the Stanford tool, that is the GNU GPL v2 or later.

Succeed is supported by the European Union under FP7-ICT and coordinated by Universidad de Alicante.