Stanford NER - Training Material | WP3

Author: Mathieu Fannee, Frank Landsbergen, Bob Boelhouwer. INL internal review: Jesse de Does

Content

Background

Stanford NER 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.

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.

Stanford NER requirements

Stanford NER is a Java application and requires Java 1.6+.

Extracting named entities with Stanford NER

The distribution of Stanford NER provides two scripts which can be used to run the software with all defaults to tag NE on a plain text input file.
On windows the command is '`run.bat <file name>`', on linux/unix you can use '`./run.sh <file name>`'.

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

For a more general use, three files are needed for NE-extraction with Stanford-NER:

1) a classifier or trained model
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 `classifiers`, some example of such a properties files are included.

Training

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.
After training, the tool produces a classifier ('model'), which is stored as a file. This model can then be used for NE-tagging at a later stage.
Stanford NER allows all properties to be specified on the command line, but it is easier to use a *properties file*. Here is a simple properties file (pretty much like the one above!), but explanations for each line are in comments, specified by "#":

```
# location of the training file
trainFile = jane-austen-emma-ch1.tsv
# location where you would like to save (serialize) your
# classifier; adding .gz at the end automatically gzips the file,
# making it smaller, and faster to load
serializeTo = ner-model.ser.gz

# structure of your training file; this tells the classifier that
# the word is in column 0 and the correct answer is in column 1
map = word=0,answer=1

# This specifies the order of the CRF: order 1 means that features
# apply at most to a class pair of previous class and current class
# or current class and next class.
maxLeft=1

# these are the features we'd like to train with
# some are discussed below, the rest can be
# understood by looking at NERFeatureFactory
useClassFeature=true
useWord=true
# word character ngrams will be included up to length 6 as prefixes
# and suffixes only
useNGrams=true
noMidNGrams=true
maxNGramLeng=6
usePrev=true
```

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

```
useNext=true
useDisjunctive=true
useSequences=true
usePrevSequences=true
# the last 4 properties deal with word shape features
useTypeSeqs=true
useTypeSeqs2=true
useTypeySequences=true
wordShape=chris2useLC
```

Here is that properties file as a downloadable link: austen.prop.
Once you have such a properties file, you can train a classifier with the command:

```
java -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -prop austen.prop
```

An NER model will then be serialized to the location specified in the properties file (ner-model.ser.gz) once the program has completed. To check how well it works, you can run the test command:

```
java -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier
ner-model.ser.gz -testFile jane-austen-emma-ch2.tsv
```

You can download the text of chapter 2 of *Emma* and the gold standard annotated version of chapter 2.
In the output, the first column is the input tokens, the second column is the correct (gold) answers, and the third column is the answer guessed by the classifier. By looking at the output, you can see that the classifier finds most of the person named entities but not all, mainly due to the very small size of the training data (but also this is a fairly basic feature set). The code then evaluates the performance of the classifier for entity level precision, recall, and F1. It gets 80.95% F1. (A commonly used script for NER evaluation is the Perl script conlleval, but either it needs adapted or else the raw IO input format used here needs to be mapped to IOB encoding for it to work correctly and give the same answer.)
So how do you apply this to make your own non-example NER model? You need 1) a training data source, 2) a properties file specifying the features you want to use, and (optional, but nice) 3) a test file to see how you're doing. For the training data source, you need each word to be on a separate line and annotated with the correct answer; all columns must be tab-separated. If you want to explicitly specify more features for the word, you can add these in the file in a new column and then put the appropriate structure of your file in the map line in the properties file. For example, if you added a third column to your data with a new feature, you might write "`map= word=0, answer=1, mySpecialFeature=2`".
Right now, most arbitrarily named features (like mySpecialFeature) will *not* work without making modifications to the source code. To see which features can already be attached to a CoreLabel, look at `edu.stanford.nlp.ling.AnnotationLookup`. There is a table which creates a mapping between key and annotation type. For example, if you search in this file for LEMMA_KEY, you will see that lemma produces a LemmaAnnotation. If you have added a new annotation, you can add its type to this table, or you can use one of the known names that already work, like tag, lemma, chunk, web.
If you modify AnnotationLookup, you need to read the data from the column, translate it to the desired object type, and attach it to the CoreLabel using a CoreAnnotation. Quite a few CoreAnnotations are provided in the class appropriately called CoreAnnotations. If the particular one you are looking for is not present, you can add a new subclass by using one of the existing CoreAnnotations as an example.

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

If the feature you attached to the CoreLabel is not already used as a feature in NERFeatureFactory, you will need to add code that extracts the feature from the CoreLabel and adds it to the feature set. Bear in mind that features must have unique names, or they will conflict with existing features, which is why we add markers such as "-GENIA", "-PGENIA", and "-NGENIA" to our features. As long as you choose a unique marker, the feature itself can be any string followed by its marker and will not conflict with any existing features. Processing is done using a bag of features model, with all of the features mixed together, which is why it is important to not have any name conflicts.
Once you've annotated your data, you make a properties file with the features you want. You can use the example properties file, and refer to the NERFeatureFactory for more possible features.
Finally, you can test on your annotated test data as shown above or annotate more text using the -textFile command rather than -testFile.
Stanford NER 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.

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]
```

Note that Stanford NER prints the results to standard output. This means that when using a directory, all files within this directory are printed subsequently, as a whole.
Stanford NER can create a file that contains a list of all found NEs with the following command:

```
$ [ … ] -NElist FILE
```

Input and output formats

Plain text or XML input is expected and the `PlainTextDocumentReaderAndWriter` is used. The classifier will tokenize the text and treat each sentence as a separate document. The output can be specified to be in a choice of three formats: slashTags (e.g., `Bill/PERSON Smith/PERSON died/O ./O`), inlineXML (e.g., `<PERSON>Bill Smith</PERSON> went to <LOCATION>Paris</LOCATION>` .), or xml, for stand-off XML (e.g., `<wi num="0" entity="PERSON">Sue</wi> <wi num="1" entity="O">shouted</wi>`). There is also a binary choice as to whether the spacing between tokens of the original is preserved or whether the (tagged) tokens are printed with a single space (for inlineXML or slashTags) or a single newline (for xml) between each one.

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. Stanford NER'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.
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 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.

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]
```

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

```
serializeTo=[path and name of the model that will be created]
map= word=0,tag=1,answer=2                       # describes columns of input

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

#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
```

License

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