Stanford NER CRF (http://nlp.stanford.edu/software/CRF-NER.html) FAQ

Questions

Questions with answers

1. **How can I train my own NER model?**

   The documentation for training your own classifier is somewhere between bad and non-existent. But nevertheless, everything you need is in the box, and you should look through the Javadoc for at least the classes `CRFClassifier` and `NERFeatureFactory`.

   The training data should be in tab-separated columns, with minimally the word tokens in one column and the class labels in another column. (You need "supervised training data" of continuous text with the class labeled for each word. You give the data file, the meaning of the columns, and what features to generate via a properties file. Or at least that's the normal and best way - you can also put all the information as command-line arguments. The "map" property is used to define the meaning of the columns. Columns are numbered from 0. One column should be called "answer" and has the NER class, while another should be called "word" and has the tokens. Existing feature extractors also know about some other column names such as "tag". There is considerable documentation of what features different properties generate in the Javadoc of NERFeatureFactory, though ultimately you have to go to the source code to answer some questions....

   Here's a sample NER properties file:

```
trainFile = training-data.col
serializeTo = ner-model.ser.gz
map = word=0,answer=1

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

The rest of this answer gives a simple, but complete example of training an NER classifier. Suppose we want to build an NER system for Jane Austen novels. We might train it on chapter 1 of Emma (ner-example/jane-austen-emma-ch1.txt). Download that file. You can convert it to one token per line with our tokenizer (included in the box) with the following command:

```
java -cp stanford-ner.jar edu.stanford.nlp.process.PTBTokenizer jane-austen-emma-ch1.txt > jane-austen-emma-ch1.tok
```

We then need to make training data where we label the entities. There are various annotation tools available, or you could do this by hand in a text editor. One way is to default to making everything an other (for which the default label is "O" in our software, though you can specify it via the `backgroundSymbol` property) and then to hand-label the real entities in a text editor. The first step can be done with Perl using this command:

```
perl -ne 'chomp; print "$_\tO\n"' jane-austen-emma-ch1.tok > jane-austen-emma-ch1.tsv
```

and if you don't want to do the second, you can skip to downloading our input file (ner-example/jane-austen-emma-ch1.tsv). We have marked only one entity type, PERS for person name, but you could easily add a second entity type such as LOC for location, to this data. The training file parser isn't very forgiving: You should make sure each line consists of solely content fields and tab characters. Spaces don't work. Extra tabs will cause problems. A blank line separates two "documents". A "document" can be just a sentence or a larger unit like a paragraph. This is the unit of CRF inference. Documents shouldn't be too large, or you waste a lot of memory and risk numerical problems.

You will then also want some test data to see how well the system is doing. You can download the text of chapter 2 of Emma (ner-example/jane-austen-emma-ch2.txt) and the gold standard annotated version of

chapter 2 (ner-example/jane-austen-emma-ch2.tsv).

Stanford NER CRF 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
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` (ner-example/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
```

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 (http://www.cnts.ua.ac.be/conll2002/ner/bin/conlleval.txt), but either it needs to be 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.

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`.

2. **How can I train an NER model using less memory?**

Here are some tips on memory usage for CRFClassifier:

1. Ultimately, if you have tons of features and lots of classes, you need to have lots of memory to train a CRFClassifier. We frequently train models that require several gigabytes of RAM and are used to typing `java -mx4g` .
2. You can decrease the memory of the limited-memory quasi-Newton optimizer (L-BFGS). The optimizer maintains a number of past guesses which are used to approximate the Hessian. Having more guesses makes the estimate more accurate, and optimization is faster, but the memory used by the system during optimization is linear in the number of guesses. This is specified by the parameter `qnSize` . The default is 25. Using 10 is perfectly adequate. If you're short of memory, things will still work with much smaller values, even just a value of 2.
3. Use the flag `saveFeatureIndexToDisk = true` . The feature names aren't actually needed while the core model estimation (optimization) code is run. This option saves them to a file before the optimizer runs, enabling the memory they use to be freed, and then loads the feature index from disk after optimization is finished.
4. Decrease the order of the CRF. We usually use just first order CRFs (`maxLeft=1` and no features that refer to the `answer` class more than one away - it's okay to refer to word features any distance away). While the code supports arbitrary order CRFs, building second, third, or fourth order CRFs will greatly increase memory usage and normally isn't necessary. Remember: `maxLeft` refers to the size of the class contexts that your features use (that is, it is one smaller than the clique size). A first order CRF can still look arbitrarily far to the left or right to get information about the observed data context.
5. Decrease the number of features generated. To see all the features generated, you can set the property `printFeatures` to `true` . CRFClassifier will then write (potentially huge) files in the current directory listing the features generated for each token position. Options that generate huge numbers of features include `useWordPairs` and `useNGrams` when `maxNGramLeng` is a large number.
6. Decrease the number of classes in your model. This may or may not be possible, depending on what your modeling requirements are. But time complexity is proportional to the number of classes raised to the clique size.
7. Use the `flag useObservedSequencesOnly=true` . This makes it so that you can only label adjacent words with label sequences that were seen next to each other in the training data. For some kinds of data this actually gives better accuracy, for other kinds it is worse. But unless the label sequence patterns are dense, it will reduce your memory usage.
8. Of course, shrinking the amount of training data will also reduce the memory needed, but isn't very desirable if you're trying to train the best classifier. You might consider throwing out sentences with no entities in them, though.
9. If you're concerned about runtime memory usage, some of the above items still apply (number of features and classes, useObservedSequencesOnly, and order of the CRF), but in addition, you can use the flag `featureDiffThresh` , for example `featureDiffThresh=0.05` . In training, CRFClassifier will train one model, drop all the features with weight (absolute value) beneath the given threshold, and then train a second model. Training thus takes longer, but the resulting model is smaller and faster at runtime, and usually has very similar performance for a reasonable threshold such as 0.05.

3. **How do I train one model from multiple files?**

Instead of setting the `trainFile` property or flag, set the trainFileList property or flag. Use a comma

separated list of files.

4. **What is the API for using CRFClassifier in a program?**

   Typically you would load a classifier from disk with the `CRFClassifier.getClassifier()` method and then use it to classify some text. See the example `NERDemo.java (ner-example/NERDemo.java)` file included in the Stanford NER download. The two most flexible classification methods to call are called `classify()`. These return a `List<CoreLabel>`, or a list of those, and take the same type or a String, respectively. A `CoreLabel` has everything you could need: the original token, its (Americanized, Penn Treebank) normalized form used in the system, its begin and end character offsets, a record of the whitespace around it, and the class assigned to the token. Print some out and have a look (`NERDEMO.java` will print some for a sample sentence, if run with no arguments). There are also a number of other classification methods that take a String of text as input, and provide various forms of user-friendly output. The method `classifyToCharacterOffsets` returns a list of triples of an entity name and its begin and end character offsets. The method `classifyToString(String, String, boolean)` will return you a String with NER-classified text in one of several formats (plain text or XML) with or without token normalization and the preservation of spacing versus tokenized. One of the versions of it may well do what you would like to see. Again, see `NERDemo.java (ner-example/NERDemo.java)` for examples of the use of several (but not all) of these methods.

5. **Can I set up the Stanford NER system up to allow single-jar deployment rather than it having to load NER models from separate files?**

   Yes! But you'll need to make your own custom jar file. If you insert into the jar file an NER model with name myModel and you put it inside the jar file, say under the `/classifiers/` path (anything works!) as `/classifiers/myModel`, then you can load it when running from a jar file with a command like:

```
java -mx500m -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier
classifiers/myModel -textFile sample.txt
```

6. **For our Web 5.0 system, can I run Stanford NER as a server/service/servlet?**

   Yes, you can. You might look at `edu.stanford.nlp.ie.NERServer` as an example of having the CRFClassifier run on a socket and wait for text to annotate and then returning the results. Here's a complete Unix/Linux/Mac OS X example, run from inside the folder of the distribution:

```
$ cp stanford-ner.jar stanford-ner-with-classifier.jar
$ jar -uf stanford-ner-with-classifier.jar classifiers/english.all.3class.distsim.crf.ser.gz
$ java -mx500m -cp stanford-ner-with-classifier.jar edu.stanford.nlp.ie.NERServer -port 9191
-loadClassifier classifiers/english.all.3class.distsim.crf.ser.gz &
# The server is now started, now separately open a client to it $ java -cp stanford-ner-with-
classifier.jar edu.stanford.nlp.ie.NERServer -port 9191 -client
Input some text and press RETURN to NER tag it, or just RETURN to finish.
President Barack Obama met Fidel Castro at the United Nations in New York.
President/O Barack/PERSON Obama/PERSON met/O Fidel/PERSON Castro/PERSON at/O the/O
United/ORGANIZATION Nations/ORGANIZATION in/O New/LOCATION York/LOCATION ./O
```

With a bit of work, we're sure you can adapt that example to work in a REST, SOAP, AJAX, or whatever system. If not, pay us a lot of money, and we'll work it out for you. Or if you don't want to do that, maybe look at what Hat Doang did (https://github.com/dat/stanford-ner). An adaptation of that code now runs our online demo (http://nlp.stanford.edu:8080/ner/).

7. **What options are available for formatting the output of the classifier?**

   At the command line, there are five choices available for determining the output format of Stanford NER. You can choose an `outputFormat` of: `xml`, `inlineXML`, `tsv`, `taggedEntities` or `slashTags` (the default). See the examples of each below (these are `bash` shell command lines, the last bit of which suppresses message printing, so you can see just the output).

   Even more options are available if you are using the API. The `classifier.classifyToString(String text, String outputFormat, boolean preserveSpaces)` method support an additional boolean flag as to whether to preserve whitespace in the output. This gives in total 10 output styles, some of which make more sense than others. The 5 command-line options are chosen from these 10, with the two XML output options preserving spaces, while the other three do not). Other output formats can be chosen or created by using other `classify.*` methods in the API. One choice is just to get a list of Java objects that are classified versions of the input. You can then print them out however your heart desires! Another popular choice is the method `classifyToCharacterOffsets(String)` which returns a List of just the entities found, together with their character offset spans. You can also get out k-best entity labeling output and the probabilities assigned to different labels. See the examples in `NERDemo.java (ner-example/NERDemo.java)`.

```
$ cat PatrickYe.txt
I complained to Microsoft about Bill Gates.
  They     told me to see the mayor of New York.
$
$ java -mx500m -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier classifie
I/O complained/O to/O Microsoft/ORGANIZATION about/O Bill/PERSON Gates/PERSON ./O
They/O told/O me/O to/O see/O the/O mayor/O of/O New/LOCATION York/LOCATION ./O
$
$ java -mx500m -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier classifie
I complained to <ORGANIZATION>Microsoft</ORGANIZATION> about <PERSON>Bill Gates</PERSON>.
  They     told me to see the mayor of <LOCATION>New York</LOCATION>.
$
$ java -mx500m -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier classifie
<wi num="0" entity="O">I</wi> <wi num="1" entity="O">complained</wi> <wi num="2" entity="O">to</wi
  <wi num="0" entity="O">They</wi>     <wi num="1" entity="O">told</wi> <wi num="2" entity="O">me</
$
$ java -mx500m -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier classifie
I       O
complained      O
to      O
Microsoft       ORGANIZATION
about   O
Bill    PERSON
Gates   PERSON
.       O

They    O
told    O
me      O
to      O
see     O
the     O
mayor   O
of      O
New     LOCATION
York    LOCATION
.       O

$
$ java -mx500m -cp stanford-ner.jar edu.stanford.nlp.ie.crf.CRFClassifier -loadClassifier classifie
                I complained to
Microsoft       ORGANIZATION    about
Bill Gates      PERSON  .

                They told me to see the mayor of
New York        LOCATION        .

$
```

The last format, `tabbedEntities` , was custom-designed to be helpful for people who would like to dump the output into a spreadsheet and then to work from there. It is an interesting mixed textual/column data representation, which we hope people find useful. The whole tokenized text is presented in textual order, in 3

tab-separated columns. Things in the first column are recognized entities, the second column gives their category, and the third column contains all the text between recognized entities. You should be able to easily load this file into a spreadsheet, R, or a database, and then to do aggregation or queries over recognized entities.

8. **Why can't I reproduce the results of your CoNLL 2003 system? Or: How do I get better CoNLL 2003 English performance?**

   The classifier used for the CoNLL 2003 shared task isn't the same as the one we distribute. The former was a 3rd order CMM (MEMM) classifier, while the latter is a 1st order CRF. So you shouldn't have any detailed assumptions about being able to replicate the results of that paper. (But the features used in that paper are fairly carefully documented, and an earlier version of NERFeatureFactory was used in that system, so between the two you should be able to work out pretty exactly the features that paper used.)

   For our distributed CRF (2009 version), the CoNLL classifier that we distribute wasn't very well tuned. Here are two better models:

   > http://nlp.stanford.edu/software/conll.closed.iob2.crf.ser.gz (http://nlp.stanford.edu/software/conll.closed.iob2.crf.ser.gz)
   > http://nlp.stanford.edu/software/conll.distsim.iob2.crf.ser.gz (http://nlp.stanford.edu/software/conll.distsim.iob2.crf.ser.gz)

   These haven't been tested on the CoNLL English testb data, but based on their performance on the testa data (91.3 and 93.0 F1), we'd predict that they'd get about 85.3 and 87.5 F1 respectively on the testb data. So, their performance is within range of previous CoNLL 2003 results. The first is trained only on the training data, while the second makes use of distributional similarity features trained on a larger data set. Also, these models are IOB2 classifiers unlike the IO classifiers which we distribute with the software.

9. **How do I get German NER working right? My characters are messed up.**

   This is almost always a character encoding issue. You need to know how your German text files are encoded (normally either one of the traditional 8-bit encodings ISO-8859-1 or ISO-8859-15, or Unicode utf-8). Then you either need to know the default platform character encoding of your computer or to override it (these days the default encoding is normally utf-8, unless you're on Mac OS X, where Mac OS X Java still uses the 8-bit MacRoman encoding as its default, despite most of the rest of the OS using utf-8). You can override the encoding of read and written files with the `-inputEncoding` and `-outputEncoding` flags.

10. **What is the asymptotic memory growth of the classifier?**

    The asymptotic memory growth in number of states `S` depends on the order of the CRF. If the CRF is order `n`, then the memory growth will be `S^(n+1)`.

11. **Can an existing model be extended?**

    Unfortunately, no.

12. **Can you release the data used to build the publicly released models**

    Unfortunately, no. The licenses that come with that data do not allow for redistribution. However, you can see the models (http://nlp.stanford.edu/software/CRF-NER.html#Models) section of the NER page for a brief description of what the data sets were.

13. **Is the NER deterministic? Why do the results change for the same data?**

    Yes, the underlying CRF is deterministic. If you apply the NER to the same sentence more than once, though, it is possible to get different answers the second time. The reason for this is the NER remembers whether it has seen a word in lowercase form before.

    The exact way this is used as a feature is in the word shape feature, which treats words such as "Brown" differently if it has or has not seen "brown" as a lowercase word before. If it has, the word shape will be "Initial upper, have seen all lowercase", and if it has not, the word shape will be "Initial upper, have not seen all lowercase".

    This feature can be turned off in recent versions with the flag `-useKnownLCWords false`

14. **How can I NER tag already tokenized text?**

    Use the following options:

    ```
    -tokenizerFactory edu.stanford.nlp.process.WhitespaceTokenizer -tokenizerOptions "tokenizeNLs=true"
    ```

15. **What options are available for text input processing?**

    Flags that may be useful for handling different types of text input are:

    - `-plainTextDocumentReaderAndWriter CLASSNAME` Specify a class to read text documents (which extends `DocumentReaderAndWriter`)
    - `-tokenizerFactory CLASSNAME` Specify a class to do tokenization (which extends `TokenizerFactory`)
    - `-tokenizerOptions "tokenizeNLs=true,asciiQuotes=true"` Give options to the tokenizer, such as the two example options here.

16. **Does the NER use part-of-speech tags?**

    None of our current models use pos tags by default. This is largely because the features used by the Stanford POS tagger (http://nlp.stanford.edu/software/tagger.html) are very similar to those used in the NER system, so there is very little benefit to using POS tags.

    However, it certainly is possible to train new models which do use POS tags. The training data would need to have an extra column with the tag information, and you would then add `tag=X` to the `map` parameter.

17. **How do you use gazettes with Stanford NER?**

    None of the models we release were trained with gazette features turned on.

    However, it is not difficult to train a new model which does use the gazette features. Set the `useGazettes` parameter to true and set `gazette` to the file you want to use. You need to supply the gazette at training time, and then the NER will learn features based on words in that gazette. The gazette will be included in the model, so does not need to be redistributed or given at test time. Any gazettes supplied at test time will be treated as additional information to be included in the gazette.

    There are two different ways for gazette features to match. The first is exact match, which means that if `John Bauer` is a gazette entry, the feature will only fire when `John Bauer` is encountered in the text. This is turned on when the `cleanGazette` option is set to true. The other is partial match, which is turned on with the `sloppyGazette` option. In that case, features fire whenever any of the words match, so `John` by itself would also trigger a feature in this case.

The gazette files should be of the format

```
CLASS1 this is an example
CLASS2 this is another example
...
```

Although the most obvious way to use the gazettes is to have `CLASS1`, `CLASS2`, etc represent the classes the model is trained to recognize, that is not strictly necessary. For example, you could have a model trained to recognize PERSON, and have two different gazette classes which represent lists of people; one could be first names and one could be last names, even if you intend them all to be features the NER generally uses to recognize names.

If a gazette is used, this does not guarantee that words in the gazette are always used as a member of the intended class, and it does not guarantee that words outside the gazette will not be chosen. It simply provides another feature for the CRF to train against. If the CRF has higher weights for other features, the gazette features may be overwhelmed.

If you want something that will recognize text as a member of a class if and only if it is in a list of words, you might prefer either the regexner or the tokensregex tools included in Stanford CoreNLP. The CRF NER is not guaranteed to accept all words in the gazette as part of the expected class, and it may also accept words outside the gazette as part of the class.

An example of how to use the gazette is presented in `austen.gaz.prop` (ner-example/austen.gaz.prop) and `austen.gaz.txt` (ner-example/austen.gaz.txt). Note that both entries in the first and second chapters show up in austen.gaz.txt; otherwise, the gazette would not help recognize names that had not shown up anywhere in the training data. If you train the NER model with the provided gazette, you will note that `Captain Weston` is correctly identified all three times in the new model, whereas the original model does not label Captain correctly in all cases.

18. **Can Stanford NER be integrated with Solr?**

    Yes. You could start from this searchbox blog post (http://www.searchbox.com/named-entity-recognition-ner-in-solr/).

You can discuss other topics with Stanford NER developers and users by joining the `java-nlp-user` mailing list (https://mailman.stanford.edu/mailman/listinfo/java-nlp-user) (via a webpage). Or you can send other questions and feedback to `java-nlp-support@lists.stanford.edu` (mailto:java-nlp-support@lists.stanford.edu).

(https://twitter.com/stanfordnlp)

(/local/calendar.shtml) JavaNLP (/javanlp/) (javadocs (/nlp/javadoc/javanlp/)) · machines (/local/machines.shtml) AI Speakers (http://ai.stanford.edu/portfolio-view/distinguished-speaker-series) · Q&A (/local/qa/)