Documentation for JNET 1.5
– JULIE Named Entity Tagger –

Katrin Tomanek, Erik Fässler

February 13, 2008

1 Objective

The JULIE Named Entity Tagger (JNET) is a generic and configurable multi-class named entity recognizer. Given a plain text of written natural language, it automatically detects and classifies named entity mentions. JNET's comprehensive feature sets allows to employ JNET for most domain and entity types. JNET was intensively tested on the general-language news paper domain (recognition of the classical MUC entities: person, location, organization) and several entity classes in the bio-medical domain.

As JNET employs a machine learning (ML) approach (see Section 6), a model (for the specific domain and entity classes to be predicted) needs to be trained first. Thus, JNET offers a training mode. Furthermore, JNET also provides several evaluation modes to assess the current model performance in terms of recall (R), precision (P), and f-score (F).

JNET offers the following functionalities:

- generation of training data containing multiple annotations
- training a model
- prediction using a previously trained model
- evaluation
- flexible feature parametrization

2 Changelog

since version 1.3:

- JNET can now output confidence values for each predicted entity (see below)

since version 1.2:

- ML features can now be configured by means of a configuration file.
- Piped format (PPD) changed: double pipes between the PoS tag and the entity label reduced to single pipe. Multiple annotations per token allowed now.

3 Installation

After unpacking JNETv1.5.tgz, nothing more has to be done. The program is written in Java[1]. Note that JNET was only tested with Java 1.5. In order to run JNET you need a Java 1.5 runtime environment installed on your system. In addition to the common Java libraries, JNET employs MALLET [McC02], a machine learning toolkit, and UEAStemmer, a conservative word stemmer[2]. However, no further installation steps are required here. JNET can be started by running the command "java -jar JNET-1.5.jar" (see below for usage).

4 File Formats

In this section, the file formats relevant to JNET are introduced. The first subsection explains how to generate training material processable by JNET using the *FormatConverter*. In this context, the *PPD format* and the *tagset* files will also be illustrated. The second subsection shows in detail how JNET may be configured.

4.1 Generating Training Data Containing Multiple Annotations

The FormatConverter takes multiple annotations in different files and merges them into a single file that contains all annotations (this file then has the PPD format). To use the FormatConverter call JNET with the argument *"f"*. Omitting further parameters causes JNET to print which paramters it expects:

```
usage: java -jar JNET-1.5.jar f <iobFile> <1st meta datafile>
[further meta datafiles] <outFile> <taglist (or 0 if not used)>
```

[1]Java is a registered trademark of Sun Microsystems, Inc.
[2]`http://www.cmp.uea.ac.uk/Research/stemmer/`

In other words, the following input is expected:

- the base entity annotation (*<iobFile>*),
- one or more further annotations (*<1st meta data file>, <further meta data files>*),
- the desired name of the output file (*<outFile>*),
- optionally the used entity tagset (*<taglist (or 0 if not used)>*). If you specify a tagset, then only the labels contained in this file will be used in the final output file (PPD). Other labels contained in the 1st meta data file (the entity annotation file) will be replaced by the default outside-label ("O"). If you do not use a tagset it is important to pass a "0" instead.

All *annotation files* need to have the following format: one token per line and a respective label per line, seperated by one or multiple whitespaces. As with the entity annotations, the label would be the entity label that has to be learned by JNET. Note: the default outside label is "O", i.e. when a token does not have a specific label, add an "O". At the end of a sentence there needs to be an empty line. The examples used below are taken from the tutorial (see Section 7).

An example of such an entity annotation might look like this:

```
We      O
report  O
a       O
case    O
of      O
colon   malignancy
cancer  malignancy
presenting      O
point   variation-type
mutations       variation-type
at      O
both    O
codons  variation-location
12      variation-location
and     O
22      variation-location
of      O
the     O
K-ras   gene-rna
gene    O
.       O
```

All other additional annotations (e.g. PoS annotations) look the same, i.e. have the same lengths, the same tokens, only the labels would then be different (PoS tags instead of entity labels).

The *tagset*, is expected to contain one entity label per line. These tags are just the entity labels you want to use. See below for an example tagset (for variation event entity types). Note: the tagset always has to contain the (default) outside label ("O"):

```
variation-event
variation-location
variation-state-altered
variation-state-generic
variation-state-original
variation-type
O
```

Performing the conversion using the FormatConverter will result in a file that contains all tokens and their annotations in the *piped format* (PPD). This is illustrated by the following example:

```
Almost|RB|O all|DT|O of|IN|O these|DT|O mutations|NNS|variation-event have|VBP|O
been|VBN|O localized|VBN|O in|IN|O codons|NNS|variation-location
12|CD|variation-location ,|,|O 13|CD|variation-location and|CC|O
61|CD|variation-location .|.|O
```

The token is followed by a pipe and a meta data tag alternating. For demonstration consider the string up to the first whitespace in the example above. The token "Almost" preceds a pipe ("|"). After this first pipe a meta data, here the PoS information, is shown. A second pipe follows. If available, a second meta data would appear after the second pipe. However, as only one meta data is used here, the string is finished by the entity label.

4.1.1 Feature Configuration File

A configuration file may be passed to JNET where the features to be used can be parameterized. Both the training mode and the evaluation modes (because they include model training as well) can consume such a file.

The information within a configuration file serves to customise the behaviour of JNET in creating its ML features. As the actual feature instances are generated depending on the respective training material, a configuration file together with the training material determines the features (and thus the model).

Next, details to the configurations are given. Generally, a configuration file consists of key-value pairs, one in a line. See Table 1 for an enumeration of these key-value pairs.

There are simple features, which can just be turned on or off (e.g. whether word stemming should be used or not), and more configurable features for which, when turned on, some parameter can be set (such as the context feature for which the size of the context can be set).

Further, there are so-called *meta-features*, i.e. binary features based on (external) information (which thus has to be provided in the training material, see above: FormatConverter and further meta data file, Section 4.1). An example of such a meta-feature are PoS tags. On which meta data a key-value pair (in the configuration file) refers is determined by the very first prefix of the key. For the PoS information, the corresponding configuration file may contain the pair *"pos_feat_enabled = true"*. It is important to note that the substring *"pos"* of the key only serves as identification of the pairs which belong to the same meta data. You could also call it *"nightisdark_feat_enabled = true"* and it would not make any difference. This stays in contrast to the rest of the key string - the form *"xxx_feat_enabled"* for indicating that the meta data refered to as *"xxx"* is used or not used must not vary!

Such a feature configuration file might look like this:

```
# enabling or disabling some features
stemming_enabled = true
feat_wc_enabled = true
feat_bwc_enabled = true
feat_bioregexp_enabled = true

# details for part-of-speech meta information
pos_feat_enabled = true
pos_feat_unit = pos
pos_feat_position = 1

# the horizon in which features should be generated
offset_conjunctions = (-1) (0) (1)

# the character for indicating that the annotation for a
# single token is not known/not available

gap_character = @
```

Now, let us assume we would like to consider chunking information for our named entity recognition. This first requires, that we have chunk annotations. We would then modify the above example feature configuration file by adding the following lines:

KEY	ALLOWED VALUES	DESCRIPTION
feat_wc_enabled	true / false	enables or disables the word class feature
feat_bwc_enabled	true / false	enables or disables the brief word class feature
feat_bioregexp_enabled	true / false	enables or disables some features primary used for bio or bio-medical texts
stemming_enabled	true / false	enables or disables stemming for feature generation
offset_conjunctions	integer list	determines the feature generation environment of a token and combinations of token features; numbers correspond to token positions relatively to the actually viewed token. (0) stands for the actual token, (-1) for the preceding token etc. (-2) (-1) (0) (1) indicates that features for the tokens (-2), (-1), (0) and (1) are generated. something like (-1 -2) or (-1, -2) would combine the features of (-1) and (-2)
gap_character	character	character that serves for indicating that the annotation for a token is not available/not known in the training material
xxx_feat_enabled	true / false	the meta data named "xxx" is used if and only if the value equals true
xxx_feat_unit	string	how this meta data should be called internally; appears in some outputs
xxx_feat_position	positive integer	the rank of this meta data in all meta datas appearing in the training material (token|meta1|meta2|...|entity label)

Table 1: Defined key-value pairs in feature configuration files.

```
chunk_feat_enabled = true
chunk_feat_unit = chunks
chunk_feat_position= 2
```

(That means, in our PPD file, the chunk information is at the second position (whereas

PoS information was on the first position).)

5 Using JNET

JNET is a command-line tool. To call JNET go to the directory where you unpacked the downloaded file and type the following:

```
java -jar JNET-1.5.jar <your arguments>
```

This will then directly call the class *JNETApplication* which serves as interface to JNET. All functionality, as listed in Section 1, can be called from this application. Running JNET without further parameters, you will be informed about the available modes:

```
usage: java -jar JNET-1.5.jar <mode> <mode-specific-parameters>

Available modes:
f: converting multiple annotations to one file
s: 90-10 split evaluation
x: cross validation
c: compare goldstandard and prediction
t: train
p: predict
oc: output model configuration
oa: output the model's output alphabet
```

Thus, the first parameter JNET expects, determines the operation mode. For example, if you want to train a model, JNET expects you to give a *'t'* as first parameter. For performing a 90-10 split evaluation a *'s'* as first parameter is needed. If you run JNET only with the first parameter the program will display the required parameters for the corresponding operation mode. E.g., if you run JNET only with the *'t'* parameter you will be noticed that it needs in addition an annotated file, a file containing the used tagset, the model file name and optionally a feature configuration file. Working with JNET always follows this scheme (for details see below).

5.1 Training

In order to train a model you need training material like that generated by the Format-Converter, that is in PPD format. Furthermore, you need to specify a tagset (see Section 4.1) and a feature configuration file (optionally). Starting JNET only with the parameter *'t'* will result in the following output:

```
usage: java -jar JNET-1.5.jar t <trainData.ppd> <tags.def> <model-out-file>
[featureConfigFile]
```

Training requires training data in piped format (<*trainData.ppd*>), the tagset (<*tags.def*>) and the future model file name (<*model-out-file*>). Optionally, you may pass a feature configuration file. The output of a training process is a model which may be used for prediction.

5.2 Prediction

For tagging a given plain text you need the used tagset and a model. In addition you have to determine the name of the output file:

```
usage: java -jar JNET-1.5.jar p <unlabeled data.ppd> <tag.def> <modelFile>
<outFile> <estimate segment conf>
```

The format of the text on which the prediction is to take place is required to equal the format of the training data. It must match the PPD format and also has to contain the same number of meta information. This is because you are to provide the meta data used for training also in the prediction process in order to generate adequate features. Obviously, the entity labels are not known for the prediction PPD file (as this is what we want to predict); thus, employ an arbitrary place holder here (e.g. *"X"*) just to meet the format specifications. But remember that you have to give exactly as much information in your predicting material as is known in the model you use for the prediction. The FormatConverter should serve well here.

When *estimate segment conf* is set to 'true', confidence estimates are printed for all entity mentions. The estimation of the classifier's confidence on each entity is based on the approach proposed by [CM04]. Note: entity-level confidence calculation might seriously slow down JNET. Thus, for processing large amounts of documents we advice to use this feature carefully.

The output of a prediction process resembles the entity annotations (see Section 4.1; by the way: this format is often also called "iob"), i.e. a file that consists of token-annotation pairs, one pair per line. When activated, confidence estimates are printed in a third column. See Figure 1 for an example.

5.3 Evaluation

JNET provides several standard evaluation modes. Each of them returns the performance in terms of recall (R), precision (P), and f-score (F).

```
Small   malignancy
cell    malignancy
carcinoma       malignancy
of      malignancy
the     malignancy
gallbladder     malignancy
:       O
a       O
clinicopathologic       O
,       O
```

Figure 1: JNET's prediction output.

5.3.1 Comparing Prediction and Gold Standard

For comparing the output of a prediction process with a given gold standard you need the prediction (*<predData.iob>*) and the gold standard (*<goldData.iob>*). Then you can run JNET in mode *'c'*:

```
usage: java -jar JNET-1.5.jar c <predData.iob> <goldData.iob> <tag.def>
```

Both are required to be plain text files and to be in the same format as the entity annotation (which is the same as the output of the prediction mode). They need to be of the same length. That is, the number of tokens and respectively the number of lines must match.

5.3.2 90-10 Split Evaluation

For performing a 90-10 split evaluation you have to pass the (training data) PPD file (*data.ppd*) on which the evaluation is to be made to JNET. This data is then randomly split into 10% for evaluation, and another 90% for training. Moreover the tagset and the name of the evaluation output is required:

```
usage: java -jar JNET-1.5.jar s <data.ppd> <tags.def> <pred-out>
[featureConfigFile]
```

An evaluation contains a training process. Thus you pass a feature configuration file.

5.3.3 Evaluation Output

The output of a 90-10 split evaluation or of a cross evaluation contains one token per line. Every token is followed by the entity label given by the JNET prediction and then by the label that should be there (according to the training data provided), that is, by the label corresponding to the gold standard. In addition the used meta infos are shown behind the gold label.

In the example below, only PoS information has been used as meta data (last column).

```
PCR     O       O       NN
-       O       O       HYPH
SSCP    O       O       NN
and     O       O       CC
subsequent      O       O       JJ
sequencing      O       O       NN
revealed        O       O       VBD
that    O       O       IN
GGT     O       variation-state-original        NN
(       O       O       -LRB-
glycine O       variation-state-original        NN
,       O       O       ,
wild    O       O       JJ
-       O       O       HYPH
type    O       O       NN
)       O       O       -RRB-
```

5.3.4 Cross Validation

During cross validation, the prodived training material is randomly split into n subsets (<*x-rounds*> specifies the number of subsets). Then $n - 1$ subsets are used for training, the remaining one for evaluation. This is repeated n times, the performance values (R/P/F) are the mean average over the performance values of each round.

The arguments for cross valiation are the same as for 90-10 split evaluation, except that you have to specify the number of evaluation rounds additionally:

```
usage: java -jar JNET-1.5.jar x <trainData.ppd> <tags.def> <pred-out> <x-rounds>
[featureConfigFile]
```

The output of cross validation is the same as of 90-10 split evaluation.

5.3.5 Model information

Running JNET with the arguments *'oc'* outputs the feature configuration specified during training for this model; argument *'oa'* shows the tagset for which the model was trained. Of course, for both modes you will have to specify the respective model.

6 Background/Algorithms

JNET is based on Conditional Random Fields (CRFs) [LMP01], a sequential learning algorithm. It was inspired by ABNER, a named entity recognition application based on CRFs as well [Set04].

7 Tutorial

By means of the demo-files contained in the `tutorial` directory[3], the use of JNET will be shown. It will be described in detail how to train a model, how to predict and how an evaluation is performed.

7.1 Training a model

This is done by calling JNET with the *"f"* parameter. The arguments are expected as follows:

```
usage: java -jar JNET-1.5.jar t <trainData.ppd> <tags.def> <model-out-file>
[featureConfigFile]
```

The training data *(<trainData.ppd>)* must match the piped format. A small section of the training file `variation.ppd` located in the `tutorial` directory is given as an example:

```
A|DT|O stabilizing|VBG|O beta-catenin|NN|O mutation|NN|O (|-LRB-|O
S|NN|variation-state-original 45|NN|variation-location
F|NN|variation-state-altered )|-RRB-|O appears|VBZ|O in|IN|O the|DT|O same|JJ|O
cell|NN|O line|NN|O that|WDT|O carried|VBD|O the|DT|O mutated|VBN|O
E-cadherin|NN|O gene|NN|O .|.|O
```

[3] These files contain annotations taken from the PennBioIE corpus. We converted them to the IOB format and added the PoS tags with our PoS tagger.

The file `vartags.def` is provided as an appropriate tagset *(<tags.def>)*. It contains one tag per line:

```
variation-event
variation-location
variation-state-altered
variation-state-generic
variation-state-original
variation-type
O
```

Additionally, the model file name *(<model-out-file>)* is needed. The use of a feature configuration file *([featureConfigFile])* is optional. An example for a feature configuration file is provided with `featconf.conf`:

```
pos_feat_enabled = true
pos_feat_unit = pos
pos_feat_position = 1
pos_begin_flag = false

offset_conjunctions = (-1)(1)

gap_character = @

stemming_enabled = true
feat_wc_enabled = true
feat_bwc_enabled = true
feat_bioregexp_enabled = true
```

The command

```
java -jar JNET-1.5.jar t tutorial/variation.ppd tutorial/variations.tags
mymodel.mod tutorial/featconf.conf
```

will result in the creation of a model named *mymodel.mod*.

Given a model, it is possible to print out to the console the used tagset and feature configuration. Printing out the tagset is done by

```
java -jar JNET-1.5.jar oa mymodel.mod
```

printing out the feature configuration is done by

```
java -jar JNET-1.5.jar oc mymodel.mod
```

7.2 Prediction

A prediction is performed using the *"p"* parameter when calling JNET. The following parameters are expected:

```
usage: java -jar JNET-1.5.jar p <unlabeled data.ppd> <tag.def> <modelFile>
<outFile>
```

As stated in section 5.2 the unlabeled input data *(<unlabeled data.ppd>)* is needed to contain the same meta data as the training data of the used model. Therefore it requires to match the piped format. The file `variations_unlabeled.ppd` serves as an example:

```
Point|NN|X mutations|NNS|X have|VBP|X the|DT|X potential|NN|X to|TO|X
activate|VB|X the|DT|X K-ras|NN|X gene|NN|X if|IN|X they|PRP|X occur|VBP|X
in|IN|X the|DT|X critical|JJ|X coding|NN|X sequences|NNS|X .|.|X
```

In this case the character "X" is used instead of the unknown labels. The tagset *(<tag.def>)* equals the tagset showed above. If you followed the instructions of this tutorial concerning the training of a model, the file `mymodel.mod` could be used as a model *(<modelFile>)*. A prediction command on the file `variations_unlabeled.ppd` might look like this:

```
java -jar JNET-1.5.jar p tutorial/variations_unlabeled.ppd tutorial/variations.def
mymodel.mod myprediction.iob
```

The output of such a prediction process is a file that contains one token and its detected label per line. Performing a prediction on the file `variations_unlabeled.ppd` outputs a file whose first lines are showed here:

```
Point variation-type
mutations variation-type
have O
the O
potential O
to O
activate O
the O
K-ras O
gene O
```

7.3 Evaluation

To run a 10-fold cross-validation with the tutorial feature set on the tutorial data, just run the following command:

```
java -jar JNET-1.5.jar x tutorial/variations_labeled.ppd
tutorial/variations.tags xvalprediction 10 tutorial/featconfig.conf
```

This will create the file `xvalprediction` whereto the predictions of each of the cross-validations will be printed. Further, the overall performance measure is shown in terms of recall, precision, and f-measure.

8 Available Models

The directory `models` contains models (and the respective tag definition) trained on the PennBioIE corpus[4]. We have trained different models for different entities to be recognized:

- a gene model: classifies entities into the classes protein, rna, and generic. (With 10-fold cross-validation, JNET achieves a performance of 83.6% F-score on these classes)
- a variation events model: for the entity subclasses: event, location, state-altered, state-generic, and state-original, and type. (With 10-fold cross-validation, JNET achieves a performance of 80.8% F-score on these classes)
- a malignancy model: for different subclasses, such as e.g. clinical stage, developemental stage, type, site, etc. (With 10-fold cross-validation, JNET achieves a performance of 71.5% F-score on these classes)

References

[CM04] Aron Culotta and Andrew McCallum. Confidence estimation for information extraction. In Daniel Marcu Susan Dumais and Salim Roukos, editors, *HLT-NAACL 2004: Short Papers*, pages 109–112, Boston, Massachusetts, USA, 2004. Association for Computational Linguistics.

[LMP01] John D. Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[4] http://bioie.ldc.upenn.edu/

[McC02] Andrew McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[Set04] Burr Settles. Biomedical named entity recognition using conditional random fields and rich feature sets. In Nigel Collier, Patrick Ruch, and Adeline Nazarenko, editors, *COLING 2004 International Joint workshop on Natural Language Processing in Biomedicine and its Applications (NLPBA/BioNLP) 2004*, pages 107–110, Geneva, Switzerland, 2004. COLING.