Documentation for

UIMA Wrapper for JULIE Lab Named Entity Tagger

Version 2.0.0

Katrin Tomanek Erik Fässler

Jena University Language & Information Engineering (JULIE) Lab

Fürstengraben 30

D-07743 Jena, Germany

katrin.tomanek@uni-jena.de

1 Objective

The UIMA Wrapper for JULIE Lab Named Entity Tagger (UIMA-JNET) is an UIMA wrapper for the JULIE Lab named entity tagger (JNET). It is part of the JULIE Lab JCORE NLP tool suite[1] which contains several NLP components (all UIMA compliant) from sentence splitting to named entity recognition and normalization as well as a comprehensive UIMA type system.

For detecting and classifing named entity mentions in a document, this analysis engine employs the JULIE Lab Named Entity Tagger (JNET). JNET is a named entity tagger that uses a machine learning (ML) approach. It generates (ML-)features in order to recognise named entities in a given text of written natural language. JNET offers the possibility to configure the feature generation and allows to use arbitrary available annotations, so-called meta data, for its features. As JNET does the named entity recognition, the UIMA-JNET provides all needed data. It takes sentence and token annotations from its CAS as well as meta data potentially required by JNET. It then writes the found named entity mentions and optionally a given corresponding resource ID back to the CAS.

[1] http://www.julielab.de/

2 About this documentation

This is a documentation on using the UIMA-compliant version of JNET. UIMA-JNET is a wrapper to JNET, which actually does all the named entity recognition. To get more information on JNET itself, please refer to its documentation.

3 Requirements and Dependencies

UIMA-JNET is completely written in Java using Apache UIMA[2]. It requires Java 1.7 (or above).

The input and output of an AE takes place by annotation objects. The classes corresponding to these objects are part of a *JULIE Lab UIMA Type System*.[3] When refering to UIMA annotation types we mean types from the JULIE Lab UIMA type system.

3.1 JNET

UIMA-JNET is based on JNET which employs the machine learning toolkit MALLET [**?**]. JNET does the named entity recognition. It receives the text to be tagged sentence-by-sentence. On a token-level, each token is assigned a class by the tagger which is either one of the entity classes or, in case that this token is not part of an entity mention, the common `OUTSIDE` label "O".

For recognizing named entities, JNET generates a set of machine learning features. It is possible to provide already available annotations of the text to be tagged. These so-called *meta datas*, e.g. PoS information, are employed for feature generation. It may be configured which meta data is supposed to be used. This is done by a *configuration file* that may be passed to JNET in any training process. The file consists of key-value pairs whereas the key is a setting-variable and the value its concrete setting. The keys refering to meta data settings are of the form *"xxx_feat_ data"*. "xxx" stands for the meta data name and the rest of the string for the setting-variable which receives a value. For details on this topic please see the JNET documentation[4]. In the following tabular the configuration settings only interesting in relation to the UIMA UIMA-JNET are explained. Please refer to the JNET documentation for more information.

[2]`https://uima.apache.org`

[3]The *JULIE UIMA type system* can be obtained separately from `http://www.julielab.de/`.

[4]JNET and its documentation can be obtained separately from `http://www.julielab.de/`

KEY	ALLOWED VALUES	DESCRIPTION
xxx_feat_data	string	a string representing the path to the corresponding UIMA-typesystem class for this annotation
xxx_feat_valMethod	string	the name of the method of the class referenced to in "xxx_feat_data" for getting the value of the annotation (without parameter brackets)
xxx_begin_flag	true / false	indicates if an IOB-like begin flag should be used; useful for annotations spanning multiple tokens

Table 1: Meta Data Feature Settings of JNET which are related to the UIMA UIMA-JNET

3.2 Model

UIMA-JNET needs a model previously trained. For training you need to use JNET directly; just type the command:

```
runJNET.sh jar t
```

,whereas the *´t'* at the end of the string indicates a training process, will result in the following output:

```
usage: runJNET.sh jar t <trainData.ppd> <tags.def> <model-out-file>
[featureConfigFile]
```

Training requires training data in *piped format (PPD)* (*<trainData.ppd>*), the tagset (*<tags.def>)* and the feature 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. For details about the piped format, tagsets and feature configuration files please refer to the JNET documentation.

3.3 Connections Between Model Training and Running UIMA-JNET

The machine learning features[5] created during a training process are also generated during the prediction process. This is done on the training text or the text to be predicted, respectively. A model contains the storage of ML-features builded while training. Using a model for prediction requires to generate the features that were used for the creation

[5]on "features" is refered in the meaning of a generic feature predicate rather than a concrete feature instance

of the model. Therefore it is necessary to provide the same information. The meta datas used in the training process are required to be available in the prediction process. That is, if PoS information were used for the training process, a PoS annotation of the text to be predicted must also be provided. In the UIMA environment the CAS on which the UIMA-JNET refers needs to have the meta information available that were used during the training process in order to work properly.

4 Using the AE – Descriptor Configuration

In UIMA, each component is configured by a descriptor in XML. In the following we describe how the descriptor required by this AE can be created (or modified) with the *Component Descriptor Editor*, an Eclipse plugin which is part of the UIMA SDK.

A descriptor contains information on different aspects. The following subsection refers to each sub aspect of the descriptor which is, in the Component Descriptor Editor, a separate *tabbed page*. For an indepth description of the respective configuration aspects or tabs, please refer to the *UIMA SKD User's Guide*[6], especially the chapter on "Component Descriptor Editor User's Guide".

To define your descriptor go through each tabbed pages mentioned here, make your respective entries (especially in page *Parameter Settings* you will be able to configure UIMA-JNET to your needs) and save the descriptor as `SomeName.xml`.

Overview This tab provides general informtion about the component. For the UIMA-JNET you need to provide the information as specified in Table 2.

Aggregate Not needed here, as this AE is a primitive.

Parameters See Table 3 for a specification of the configuration parameters of this AE. Do not check "Use Parameter Groups" in this tab.

Parameter Settings The specific parameter settings are filled in here. For each of the parameters defined in 4, add the respective values here (has to be done at least for each parameter that is defined as mandatory). See Table 4 for the respective parameter settings of this AE.

[6]`http://incubator.apache.org/uima/`

Subsection	Key	Value
Implementation Details	Implementation Language	Java
	Engine Type	primitive
Runtime Information	updates the CAS	check
	multiple deployment allowed	check
	outputs new CASes	don't check
	Name of the Java class file	`de.julielab.jules.ae.netagger.EntityAnnotator`
Overall Identification Information	Name	UIMA-JNET
	Version	2.3
	Vendor	JULIE Lab
	Description	you may keep this empty

Table 2: Overview/General Settings for AE.

Type System On this page, go to *Imported Type* and add the following layers of the *JULIE UIMA Type System* (Use "Import by Location"): `julie-basic-types.xml`, `julie-morpho-syntax-types.xml`, and `julie-semantics-types.xml`. In case you use special subtypes of `EntityMention`, of course, this part of the type system needs to be added as well.

Capabilities UIMA-JNET needs as input annotations from type `de.julielab.jules.types.Sentence` and `de.julielab.jules.types.Token`. It returns annotations from type `de.julielab.jules.types.EntityMention`. As output type you might specify any subtype of `EntityMention`. If you do so, do not forget to add the respective type system defining this type. See Table 5.

Index Nothing needs to be done here.

Resources Nothing needs to be done here.

5 Further Explanation of some Functionalities of UIMA-JNET

This section explains some of UIMA-JNET's features in more detail.

Parameter Name	Parameter Type	Mandatory	Multivalued	Description
ModelFilename	String	yes	no	specifies which model JNET should use
EntityTypes	String	yes	yes	specifies which JNET named entity label should be represented by which typesystem class within UIMA
ExpandAbbreviations	Boolean	no	no	if set to true then abbreviations (if annotated in the CAS) are expanded by their full form. Entity recognition is then performed on the full form (instead of the short form).
ShowSegmentConfidence	Boolean	no	no	if set to true, a the classifier's confidence on each entity is calculated. See Section 5.1 for more details.
NegativeList	String	no	no	a list with entity mentions (covered text) and label which, when predicted as entity, will be ignored, i.e., is not written to the CAS.
ConsistencyPreservation	String	no	no	coma-separated list of modes. Keep blanc if consistency preservation should not be used. See Section 5.2 for more details.

Table 3: Parameters of this AE.

5.1 Segment Confidence

The calculation of the segment confidence follows the approach proposed in [?]). The segment confidence is written to the feature `confidence` of the annotation type (or any subtypes of) `EntityMention`.

Parameter Name	Parameter Syntax	Example
ModelFilename	just give the complete path to the model file	`/my/path/to/genemodel.mod.gz`
EntityTypes	given by pairs in the form of "<JNET label>=<path typesystem class>"	"gene-dna=de.julielab.jules.types.EntityMention"; note that although you might specifiy different types here, all types must be subtypes of `EntityMention`.
ExpandAbbreviations	boolean	see above
ShowSegmentConfidence	Boolean	see above
NegativeList	one entry per line: "entity mention@label". You might omit the label (if so then the this is a negative entry for an arbitry entity label), however, in this case you should have an @ at the end of the line!	IL-2@gene-protein IL-2 receptor HDA1@gene-generic
ConsistencyPreservation	String	"string, full2acro"

Table 4: Parameter settings of this AE.

Type	Input	Output
de.julielab.jules.types.Sentence	√	
de.julielab.jules.types.Token	√	
de.julielab.jules.types.EntityMention		√

Table 5: Capabilities of this AE.

Note: entity-level confidence calculation might seriously slow down UIMA-JNET. Thus, for processing large amounts of documents we advice not to use this feature.

5.2 Consistency Preservation

The idea of this feature is that within one document the same string should recieve the same entity label (if at all). As UIMA-JNET does the tagging on sentence-level, it might happen that within one sentence as certain token is tagged as an entity whereas in another sentence it is not. This is often the case when an entity is used in an "untypical" context (i.e., it is surrounded by words very different from the training material). To avoid inconsistent annotation, the consistency preservation mode makes sure that when a token (or a multi-token expression) is once recognized as an entity mention, all other occurences of this token within an document are also labeled as the same entity mention.

Further, if an acronym was detected as an entity, but the respective full-form was not (or vice versa)[7], the respective `EntityMention` annotation is copied.

This feature is currently under construction and will have to be improved. Use it carefully!

Available modes are:

- string
- full2acro
- acro2full

[7]For this feature to work you need to run the acronym tagger beforehand!