Etxt2DB component generator

1. BASIC INFORMATION

Tool name

Etxt2DB component generator: a tool for generating U-Compare components for Entity Recognition

Overview and purpose of the tool

Etxt2DB [1] is a framework for specifying and executing Entity Recognition (ER) programs. These programs accept as input a text containing potentially interesting entities to be extracted and produce the input text annotated with the recognized entities.

The Etxt2DB functioning mode involves two distinct phases. First, the training phase consists in creating a model based on a given ER technique and one or more resources that guide the creation of the classification model. Examples of these resources are dictionaries for rule-based ER techniques [2] or training data for statistical learning techniques (e.g., Conditional Random Fields [3]). Second, in the execution phase, a classification model previously created receives as input plain text and produces annotations corresponding to the recognized entities.

The Etxt2DB framework consists of a software layer, built on top of Minorthird [4] and Lingpipe [5], offering a command-like specification language. Existing Machine Learning Java APIs (such as Minorthird and Lingpipe) provide implementations of Entity Recognition techniques. Some developers of ER applications do not want to get involved in the implementation details of the techniques used. Instead, they are willing to focus on: (i) the choice of the technique to be used; (ii) the resources used in the process (e.g., dictionaries); and (iii) a good set of features that help the ER program to take adequate decisions. The objective of the Etxt2DB specification language is to turn the development and tuning of ER programs easier for developers that are mainly concerned with these topics.

In the context of the METANET project, the goal was to build a component-generator tool that encapsulates Etxt2DB. In the training phase, this tool accepts a training data set as input and produces a classification model and a U-Compare component that is able to interpret that model. In the execution phase, the component produced is loaded into the U-Compare platform and then is ready to be used for recognizing entities from text.

2. TECHNICAL INFORMATION

Software dependencies and system requirements

The E-txt2db component creator is implemented in Java. Thus, it requires the installation of Java SDK (version 1.6 or above recommended).

While it is possible to run this software with less than 2GB RAM with a dataset of 500 documents, it is recommended that the user prepares to adapt to higher values according to the size of the dataset that is used (a larger dataset will require much more memory during training).

Installation

No installation is required.

Execution instructions

Assuming that Java SDK is installed, this software can be run from any terminal in any Operative System (Windows, Mac OS, Linux).

Starting from the folder where the jar file is stored, the command to run the software is:

```
java -jar etxt2dbComponentGenerator.jar [PATH_TO_SPECIFICATION] [PATH_TO_STORE_RESULTS]
```

where *[PATH_TO_SPECIFICATION]* is the path to the specification file to be used to generate the component (the format of this file is described in the next section) and *[PATH_TO_STORE_RESULTS]* is the path to the directory where the E-txt2db trained model and the U-Compare component must be stored.

Additionally, if the user gives as input a large dataset for training (e.g., more than 500 documents), we recommend the use of the virtual machine variables (-Xmx*[SIZE]* and –Xms*[SIZE]*) to increase the amount of memory that can be used by the program.

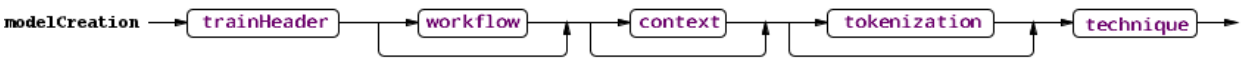
Input/Output data formats

Input data formats

There are three types of input documents that are relevant for the user: *(i)* the specification file; *(ii)* the training data; and *(iii)* the type configuration file.

Specification file: this input document contains the specification of the ER program that you want to produce. This specification is provided using a variation of the language of E-txt2db to produce new models. To present the syntax of this language, we use a graph-based approach. In the graph, the rectangular nodes represent keywords and the nodes with rounded tips represent blocks that must be parameterized by the user.

All the commands in this language have the same basic structure given by the following graph:

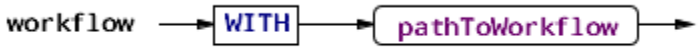


The basic clauses of the command to create a model are:

1. *trainHeader*, provides information about the name of the model that will be created. The graph representation of this clause is the following:

Where *modelName* is the name of the model. This name becomes the identifier of the classification model. It must contain one or more characters from all the letters, digits, and symbols ".", ":", "-" and "_". The name must start by a letter or the symbol "_".

2. *workflow*, is an optional clause that indicates the U-Compare workflow to execute in order to extract annotations that are used during the creation of the model. If this clause is not specified then we assume that no U-Compare annotations will be used during the training phase. The graph representation of this clause is the following:

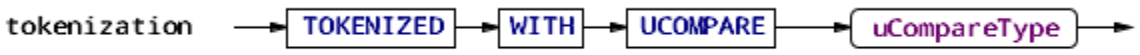


Where *pathToWorkflow* is a string containing the path to the workflow file.

3. *context*, is an optional clause that indicates the context to be used by the ER techniques to extract information. When this clause is used, documents are split according to the context and the model is trained to work specifically on that type of context. There are four types of contexts: (i) *Document*; (ii) *Paragraph*; (iii) *Sentence*; and (iv) *Line*. When this clause is not specified, the context that is chosen is *Document*. The graph representation of this clause is the following:

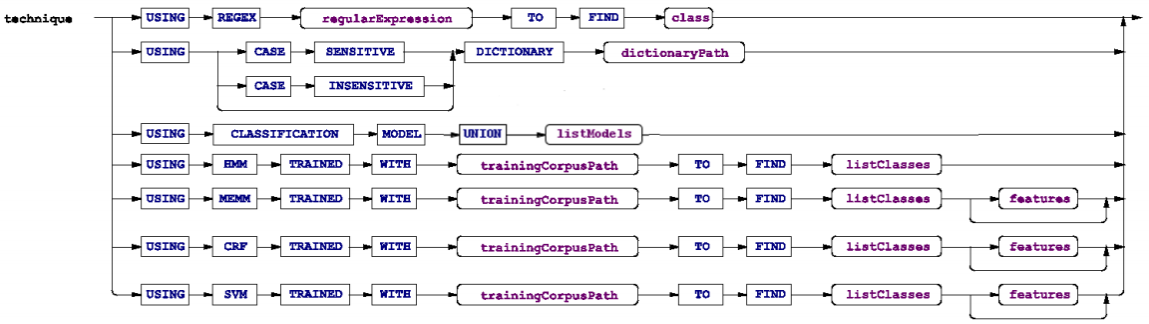
Where *contextName* is one of the four types of contexts described before.

4. *tokenization*, is an optional clause that indicates the type of U-Compare annotation to perform the tokenization of the documents. This clause is flexible in what concerns the type of annotation that is used (you can choose any type of annotation). However, be advised that there are some types of annotation that perform this job better (e.g., Token, POSToken) than others. The graph representation of this clause is the following:



Where uCompareType is the type of annotation to be used to perform the Tokenization.

5. *technique*, indicates the ER technique that is used by the created model. The syntax for this model is different for each technique, as represented by the following figure:

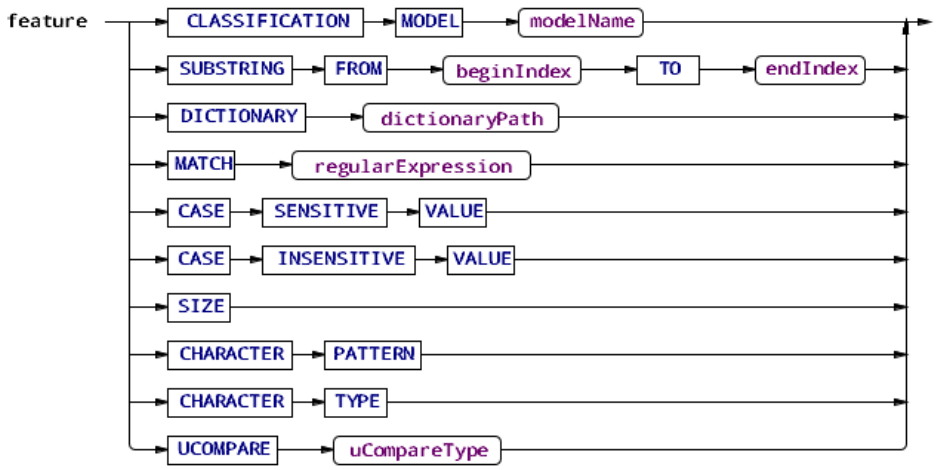


The first path of the graph represents the syntax to create a classification model based on regular expressions. In this path, *regularExpression* is a string delimited by "" or '' that indicates the regular expression the classification model uses. The end of this path, the *class* variable, indicates the class that must be assigned to the segments that match the regular expression. The name of the class follows the same lexical rules as the name of the classification models.

The second path of the graph corresponds to the syntax available to create a dictionary-based classification model. In this path, *dictionaryPath* is a string indicating the path to the dictionary file. Note that, to create a dictionary, a user can optionally indicate if it is a case sensitive or case insensitive dictionary. By default, the dictionary is case sensitive.

The third path of the graph allows a user to create a new Classification model as a union of previously created Classification models. In this path, *listModels* is a comma separated list of previously created Classification models. Applying the resulting model of this command to an input text is equivalent to consecutively applying all the models in *listModels* to the text.

The remaining paths correspond to the creation of Classification models based on machine learning techniques. In these paths, *trainingCorpusPath* is a string indicating the path to the training corpus. The *listClasses* is a list of one or more comma separated classes that the Classification model must learn to identify using the training corpus. Alternatively, the list may consist on the symbol "*", if the user wants the model to learn how to identify all the classes that are present in the training corpus. If the machine learning technique used is MEMM, CRF or SVM, the user can, optionally, customize the features used in the classification process. This customization is performed through the *features* block. The syntax of the *features* block is as follows:



In order to read more about these features and how they work, please report to [1]. Next, we present an example of a specification file:

```
CREATE CLASSIFICATION MODEL AS seminarsModelUCompareFeatures
WITH "examples/workflows/OpenNLP_AE_only.ucz"
CONTEXT Sentence
TOKENIZED WITH UCOMPARE Token
USING SVM TRAINED WITH "examples/data/train/"
TO FIND speaker, location, stime, etime
CAPTURING FEATURES CASE SENSITIVE VALUE,
                                CHARACTER PATTERN,
                                CHARACTER TYPE,
                                CASE INSENSITIVE VALUE,
                                UCOMPARE POSToken
```

Training data: the training data correspond to plain text files with XML tags annotating relevant text segments that correspond to named entities, as illustrated in the following example:

```
CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

<speaker>Joe Mertz</speaker>
Center for Innovation in Learning, CMU

<date>Friday, February 17</date>
12:00pm-1:00pm
<location>Student Center Room 207</location> (CMU)
ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to
help design instructions for training circuit board assemblers. The model was built in the Soar
cognitive architecture, and was initially endowed with only minimal prerequisite knowledge of the
task, and an ability to learn instructions. Lessons for teaching expert assembly skills were
developed by iteratively drafting and testing instructions on the simulated student.

Please direct questions to Pamela Yocca at 268-7675
```

Type configuration file: this file is used to define the UIMA annotations that can be used by E-txt2db models. It is not expected that this configuration file changes too much. However, it is left

out of the jar file in order to allow for some flexibility in the available types without changing the code.

Each line of the configuration file represents one annotation type. The format of each line is as follows:

```
key                 java.class.representing.the.type          distinguinshingMethods*
```

The *key* corresponds to the identifier that is used in the specification file to represent a certain type. The *java.class.representing.the.type* is the package and class that represent the annotation. Finally, the *distinguishingMethods* are the methods that can be used to distinguish between different annotations of the same type (e.g., the POSToken annotations can be of several different types according to the POS tag). Each line may contain zero or more *distinguishingMethods*.

Our distribution of the E-txt2db component generator contains a default type configuration file that can be found in the *config* folder:

```
Token               org.u_compare.shared.syntactic.Token
POSToken            org.u_compare.shared.syntactic.POSToken          getPosString
Sentence            org.u_compare.shared.syntactic.Sentence
Chunk               org.u_compare.shared.syntactic.Chunk             getLabelString
Dependency          org.u_compare.shared.syntactic.Dependency        getLabelString
NamedEntity         org.u_compare.shared.semantic.NamedEntity
Person              org.u_compare.shared.semantic.Person
Place               org.u_compare.shared.semantic.Place
ProperName          org.u_compare.shared.semantic.ProperName
Paragraph           org.u_compare.shared.document.text.Paragraph
Title               org.u_compare.shared.document.text.Title
TextBody            org.u_compare.shared.document.text.TextBody
AbstractText        org.u_compare.shared.document.text.AbstractText
```

Output data format

The E-txt2db component generator produces two output files when it is executed: *(i)* the E-txt2db model; and *(ii)* the XML file that corresponds to the component descriptor.

None of these files is supposed to be interpreted by humans. Instead, they are supposed to be used loaded and executed in U-Compare.

Integration with external tools

The outputs of this tool are the E-txt2db model and the XML file that corresponds to the component descriptor. In order to use them, you need to load them into U-Compare. Assuming you run the E-txt2db component generator and save the results in the folder /path/to/results/ , the process to use the component in U-Compare goes as follows:

1. Go to *Library > Register External Componends (Edit classpath)*.
2. Press *Add Jar File(s) to Classpath* and browse to the jar file of the E-txt2db Component Generator. Press *Open*. (This step only needs to be performed once)
3. Press *Add Directory to Classpath* and browse to the /path/to/results/ . Press *Open*. (This step needs to be done for each new output folder of the E-txt2db Component Generator. If you use the same output folder more than once you do not need to repeat this step)
4. Check the box next to the /path/to/results/ entry and press the *Search for Component Descriptors* button.

5. You will see a list of all the components that are present in that folder. Check the box next to the ones you want to load and press the *Add Selected Components button*.
6. Your component will appear in the main U-Compare window, under *Custom Components* (at the bottom of the list). You can now use it in your workflows.

3. CONTENT INFORMATION

Our distribution of the E-txt2db component generator contains a folder called *example* that can be used to perform some tests and familiarize with this software. Inside this folder, you will find three folders:

data: Contains a sample of the CMU seminar announcements dataset. The data is divided into train and test data. The train data is used by the component generator to produce the model while the test data can be used by U-Compare to perform tests with the component.

specification: Contains three examples of specifications: *(i)* to train an HMM model using the default splitter and tokenizer; *(ii)* to train an HMM model using a splitter based on Sentence annotations from U-Compare and a tokenizer based on Token annotations from U-Compare; and *(iii)* to train a SVM using a splitter based on Sentence annotations from U-Compare, a tokenizer based on Token annotations from U-Compare and some features that include POS tokens from U-Compare.

workflows: Contains two workflows that can be used by the component generator.

4. ADMINISTRATIVE INFORMATION

Contact

For further information and technical support installing and/or running this tool, please email to Gonçalo Simões: goncalo.simoes@ist.utl.pt.

5. REFERENCES

[1] Simões, G., *E-txt2db: Giving structure to unstructured data*; Master Thesis from Instituto Superior Técnico – Technical University of Lisbon, 2009, https://dspace.ist.utl.pt/bitstream/2295/570030/1/dissertacao.pdf.

[2] A. Baronchelli, E. Caglioti, V. Loreto, and E. Pizzi. *Dictionary based methods for information extraction*. In Physica A: Statistical Mechanics and its Applications 300, 2004.

[3] A. McCallum and W. Li. *Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons*. In CONLL, 2003.

[4] Cohen, William W. *MinorThird: Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data*, http://minorthird.sourceforge.net, 2004.

[5] Alias-i. *LingPipe 4.1.0*. http://alias-i.com/lingpipe, 2008.