Segmenteur Étiqueteur Markovien (SEM)

Contents

1 Preface

1.1 SEM presentation

Segmenteur Étiqueteur Markovien (SEM) [10] is a free syntactic annotation software for French.

It allows raw text segmentation in tokens and sentences, but it can also take presegmented text as an input. Multiword entities can be handled two different ways: either as a single token where every subtoken is linked by the character '_' or as multiple word tagged with the same syntactic category.

SEM offers three levels of annotations. The first one is Part-Of-Speech using the tagset defined by [3]. The second one is a chunking annotation using IOB2 tagging scheme. Chunking can be complete or partial (only noun phrases). The third one is named entity recognition (NER) using the [8] tagset.

2 Installation

You can find all the useful informations to install SEM at the following page:
https://github.com/YoannDupont/SEM

SEM has to be downloaded to be installed. The installation can be launched using the following command:

```
python setup.py install --user
```

It will install SEM with all its prerequisites.
There are two ways to download the latest version.

2.1 If GIT is installed

Open a terminal and launch the following command:

```
git clone https://github.com/YoannDupont/SEM.git
```

It will create the SEM folder into the current working directory.

It is the GIT branch, used to handle the different versions of the software. You should not modify the content of this folder to avoid problems when updating your SEM version.

The interest here is to easily update the software by launching the following command:

```
git pull
```

2.2 If GIT is not installed

On https://github.com/YoannDupont/SEM you can click on "clone or download" then "Download ZIP".

The advantage of this method is that files are not versioned. It is not necessary to be cautious about modifying the folder's content. The drawback is that you need to download the whole zip again if you ant to update your software.

2.3 Wapiti

Wapiti [7] is a free software implementing CRFs, it allows to train tagger from annotated corpora.

The latest Wapiti version that is compatible with SEMis available in the ext folder. It contains installation instructions. SEM will work with this Wapiti specificaly. You need to compile it. Wapiti is now done automatically when installing SEM.

3 Corpus, annotations and linguistic ressources

3.1 French Treebank (FTB)

SEM was trained on the French Treebank (FTB) [1].

3.2 PoS tagset

POS tagging uses the tagset defined in [3] :

ADJ : adjective

ADJWH : "wh question" adjective

ADV : adverb

ADVWH : "wh question" adverb

CC : conjonction de coordination

CLO : object clitic

CLR : reflected clitic

CLS : subject clitic

CS : conjonction de subordination

DET : determiner

DETWH : "wh question" determiner

ET : foreign word

I : interjection

NC : common noun

NPP : proper name

P : preposition

P+D : preposition + determiner

P+PRO : preposition + pronoun

PONCT : ponctuation

PREF : prefix

PRO : pronoun

PROREL : relative pronoun

PROWH : "wh question" pronoun

VINF : infinitive

VPR : present participle

VPP : past participle

V : indicative

VS : subjunctive

VIMP : imperative

3.3 Chunking tagset

Chunking uses the tagset defined in [11] :

AP : adjectival phrase

AdP : adverbial phrase

NP : noun phrase

VN : verbal node

CONJ : conjunction

UNKNOWN : unknown chunk

PP : prepositional phrase

3.4 Named Entity Recognition tagset

For NER SEM uses the tagset defined in [8] :

Person : people (without their titles)

Location : countries, town, regions, etc.

Organization : non profit organizations

Company : companies

POI : Point Of Interet (example : The Opera)

FictionCharacter : fictional characters

Product : branded products

3.5 Lexique des Formes Fléchies du Français (LeFFF)

The Lexique des Formes Fléchies du Français (LeFFF) [2] is a rich french lexicon providing morphological and syntactic information. SEM uses the LeFFF as a gazetteer to improve the quality of the POS tagging.

4 File formats

SEM allows to process two kind of files: linear files and vectorized files.

4.1 Linear files

A linear file is a file in which word are (usually) separated by a space. SEMconsiders that an empty line is the end of a sentence.

4.1.1 Examples

example 1 : raw text
Le chat dort.

example 2 : POS tagged text
Le/DET chat/NC dort/V ./PONCT

example 3 : text annotated with POS and chunking
(NP Le/DET chat/NC) (VN dort/V) ./PONCT

4.2 Vectorised files

A vectorized file is a file where each token in on one line, sentences are separated by an empty line. In this kind of file, each token can have multiple informations (descriptors), each descriptor is separated by a tabulation. Each descriptor is on a specific column.

4.2.1 Examples

example 1 : vectorised raw text

Le
chat
dort
.

example 2 : vectorised text enriched with the information "does the word start with an uppercase?"

Le	oui
chat	non
dort	non
.	non

example 3 : vectorised text with POS annotation

Le	DET
chat	NC
dort	V
.	PONCT

example 4 : vectorised text with POS and chunking

Le	DET	B-NP
chat	NC	I-NP
dort	V	B-VN
.	PONCT	0

4.3 SEM files

SEM can use two formats: and XML one and a JSON one. The two of them give the same informations and represent a Document type used in the code.

In these kind of files, we find various informations such as the name of the document, its content and its metadata. Other informations may be found, such as tokenization (in tokens, sentences, etc.) and annotations (POS, chunks, NER, etc.).

4.3.1 Examples

A SEM-XML example is given in the figure 1. The same file in JSON format will not be provided as they have the same informations.

5 Usage

SEM has independent modules, the main program can be used as a dispatcher to the right module to launch.

```xml
<? xml version="1.0" encoding="UTF-8" ?>
<document name="exemple.txt">
  <metadata encoding="utf-8" />
  <content>SEM est un programme bien documenté.

SEM est écrit par Yoann Dupont.</content>
  <segmentation name="tokens">
    <s s="0" l="3" />
    <s s="4" l="3" />
    <s s="8" l="2" />
    <s s="11" l="9" />
    <s s="21" l="4" />
    <s s="26" l="9" />
    <s s="35" l="1" />
    <s s="38" l="3" />
    <s s="42" l="3" />
    <s s="46" l="2" />
    <s s="49" l="9" />
    <s s="59" l="5" />
    <s s="65" l="3" />
    <s s="69" l="5" />
    <s s="75" l="6" />
    <s s="81" l="1" />
  </segmentation>
  <segmentation name="sentences" reference="tokens">
    <s s="0" l="7" />
    <s s="7" l="9" />
  </segmentation>
  <segmentation name="paragraphs" reference="sentences">
    <s s="0" l="1" />
    <s s="1" l="1" />
  </segmentation>
</document>
```

Figure 1: exemple de fichier XML-SEM.

To get the list of available modules, launch:

```
python -m sem (--help ou -h)
```

To get the SEM version:

```
python -m sem (--version ou -v)
```

To get the informations about the last release of SEM:

```
python -m sem (--informations ou -i)
```

To launch a specific module, the overall syntax is:

```
python -m sem <module\_name> <module\_arguments\_and\_optic
```

The different modules will be explained one by one.

5.1 annotate

description

Annotate using the annotator passed in argument.

arguments

infile

the input file.

outfile

the output file.

annotator

le name of the annotator.

location

the path where every useful information are to be found (model, folder with lexica, etc).

token_field

the name of the column where tokens are located (not always useful)

field

the name of the output column

options

–help ou -h: switch

displays help

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.2 chunking_fscore

description

Compute the f-score over data tagged using IOB scheme. Gives f-score over each class, a micro- and a macro- global f-score.

arguments

infile

The input file. Should follow the CoNLL format.

options

–help ou -h: switch

displays help

–reference-column ou -r: int

the index of the column where reference tags are located (default: -2).

–tagging-column ou -t: int

the index of the column where hypothesis tags given by the system are located (default: -1).

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.3 clean

description

clean allows to remove columns that are not useful.

arguments

infile: file

The input file. Follows the CoNLL-2003 format.

outfile: file

The output file.

ranges: string

columns to keep. The user can give either a number or a range. A range is a couple of number separated by a colon. Multiple ranges can be given, they have to be separated by a comma.

options

–help ou -h: switch

displays help

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.4 enrich

description

adds descriptors to a vectorized file. Informations to add are declared in an XML configuration file. Features are explained in section 6.1.

arguments

infile: file

the input file, in vectorized format.

infofile: file

configuration file where features are declared, XML format.

outfile: file.

The output file, in vectorized format.

options

–help ou -h: switch

displays help

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.5 export

description

Convert CoNLL file to another format.

arguments

infile

the input file, vectorized format.

exporter_name

the name of the exporter.

outfile

the output file.

options

–help ou -h: switch

displays help

–pos-column ou -p

the column where POS tags are located.

–chunk-column ou -c

the column where chunking tags are located.

–ner-column ou -n

the column where NER tags are located.

–lang

the language of the document (default: fr)

–lang-style ou -s

the CSS stylesheet to use for HTML export (default: default.css)

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.6 label_consistency

description

Improves consistency of the annotations by broadcasting the system's annotations in the whole document. Unannotated elements that are identical to tagged elements will have the most common category.

arguments

infile

input file, vectorized format.

outfile

output file, vectorized format.

options

–help ou -h: switch

displays help

–token-column ou -t

the column where tokens are located.

–tag-column ou -c

the column where tags are located.

–label-consistency (choice: non-overriding, overriding)

broadcasting heuristic. "non-overriding" keeps system output in case of conflict. "overriding" overrides system annotations if a longer one is found.

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.7 tagging

description

arguments

options

–help ou -h: switch

displays help

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.8 segmentation

description

Takes a linear file and segments its content in tokens and sentences.

arguments

infile: file

the input file. Raw text format.

outfile: file

the output file. Vectorized format.

options

–help ou -h: switch

displays help

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.9 compile

description

Serializes a gazeteer file that that can be used as a resource by SEM.

arguments

input: file

The gazeteer to compile.

output: file

The serialized gazeteer.

options

–help ou -h: switch

displays help

-k ou –kind: enumeration {token, multiword}

The kind of dictionary. token: each entry is a word. multiword: each entry is a sequence of words.

–input-encoding: string

The encoding of the input file (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.10 decompile

description

Deserializes a gazeteer file. The resource can be easily modified.

arguments

input: file

The serialized gazeteer.

output: file

The deserialized gazeteer.

options

–help ou -h: switch

displays help

–input-encoding: string

The encoding of the input file. Has priority over –encoding (default: –encoding).

–output-encoding: string

The encoding of the output file. Has priority over –encoding (default: –encoding).

–encoding: string

Encoding of both the input and the output files. Does not have priority (default: UTF-8).

–log ou -l: string

the log level: info, warn or critical (default: critical).

–log-file: file

the file where to log (default: terminal).

5.11 tagger

description

it is the main SEM module. It allows to process files using a pipeline. Pipes are processes made by either a module or Wapiti. The modules to use and their order is given in an XML configuration file called the master file.

arguments

master: XML file

the master configuration file. Defines the pipeline and the options.

input_file: file

the input file.

options

–help ou -h: switch

displays help

–output-directory ou -o: directory

the output directory (default: current working directory).

5.12 gui

description

GUI used to annotate with SEM or train a new model.

arguments

resources (optional)

resource file for SEM containing models, lexica, etc.

options

–help ou -h: switch

displays help

5.13 annotation_gui

description

The GUI used to manually annotate documents.

arguments

no arguments.

options

–help ou -h: switch

displays help

–log ou -l: string

the log level: info, warn or critical (default: critical).

6 Configuration files

6.1 For the enrich module

The enrich module configuration file allows to add informations to a vectorised file. It first describres entries that are present then the informations to add.

It is and XML file of the document type "enrich". It has two parts: an "entries" defining entries that are already present and a "features" one to enrich the file.

Each entry (present or added) has to have a name (using the *name* attribute). Two different entries cannot have the same name. An example of an enrich module configuration file is given in the figure 2. Each entry has a mode allowing SEM to only consider it in certain contexts. The *train* mode allows to use an entry only when the aim is to train the model. The default mode, *label*, considers the entry no matter the context.

Most features have the following attributes:

- name="string": the name of the feature. Mandatory for root features.
- action="string": for features of the same nature (eg: evaluating a regular expression), select the kind of result or a different computation.
- x="integer": the shift according to the current token (default: 0).
- entry="string" : the entry used by the feature (default: word or token, otherwise the first feature defined in entries).

- display="(yes|no)": whether the feature should be displayed or not. It is possible to not display a feature used as a temporary result.

There are multiple kinds features according to the kind of results they compute or the arguments they take into account:

- *token features* : they process each token independently. There are two kinds of token features:
  - *string features*: they return a string
  - *boolean features*: they return a boolean
- *sequence features*: they process the whole sequence and return a sequence

```xml
<? xml version="1.0" encoding="UTF-8" ?>
<information>
  <entries>
    <before>
      <entry name="word" />
      <entry name="POS" />
    </before>
    <after>
      <entry name="NE" mode="train" />
    </after>
  </entries>
  <features>
    <nullary name="lower" action="lower" display="no" />
    <ontology name="NER-ontology" path="dictionaries/NER-ontology" display="no" />
    <fill name="NER-ontology-POS" entry="NER-ontology" filler-entry="POS">
      <string action="equal">O</string>
    </fill>
    <find name="NounBackward" action="backward" return_entry="word">
      <regexp action="check" entry="POS">^N</regexp>
    </find>
    <find name="NounForward" action="forward" return_entry="word">
      <regexp action="check" entry="POS">^N</regexp>
    </find>
  </features>
</information>
```

Figure 2: enrich module configuration file example. It allows to add descriptors that will be used for machine leaning.

6.1.1 *Arity* features

Arity features are defined based on the number of arguments they take as input. There are multiple kinds of *arity* features.

The first one *arity token feature* is *nullary*. The kind of feature takes no arguments. Examples of *Nullary* feature are given in figures 3, 4, 5 and 6. The available actions are:

- BOS (*boolean feature*): is the word at the Beginning Of Sequence ?
- EOS (*boolean feature*): is the word at the End Of Sequence ?
- lower (*string feature*): transforms the token in lowercase;
- substring (*string feature*): Provide a substring of the token. Defines the following options (xml attributes):
  - from="[int]": the beginning index for the substring (default: 0)
  - to="[int]": the end index for the substring. If 0 is given, "to" is the end of the string (default: 0)

```
<nullary name="IsFirstWord?" action="BOS" />
```

Figure 3: example of the *nullary* feature "BOS".

```
<nullary name="IsLastWord?" action="EOS" />
```

Figure 4: example of the *nullary* feature "EOS".

```
<nullary name="lower" action="lower" />
```

Figure 5: example of the *nullary* feature "lower".

```
<nullary name="radical-3" action="substring" to="-3" />
```

Figure 6: example of the *nullary* feature "substring".

The second kind of *token feature* is *unary*. They take a single argument. An example is provided in figure 7. The available actions are:

- isUpper: checks if the letter at the given index is in upper case.

```
<unary name="StartsWithUpper?" action="isUpper">0</unary>
```

Figure 7: example of the *unary* feature "isUpper".

The third *token feature* is *binary*. It takes two arguments. An example is given in figure 8. The available actions are:

```
<binary name="To-Greek-Alpha" action="substitute">
    <pattern>alpha</pattern>
    <replace>α</replace>
</binary>
```

Figure 8: example of the *binary* feature "substitue".

- substitute: substitute a string by another one. The first argument is *pattern* and the second is *replace*.

The fourth kind of *token feature* is *n-ary*. It takes an arbitrary number of elements. An example is given in figure 9. The available actions are:

- sequencer: performs a sequence of processes. They are always *string features*, the last one can be either a *string feature* or a *boolean feature*.

```
<nary name="CharacterClass" action="sequencer">
    <binary action="substitute">
        <pattern>[A-Z]</pattern>
        <replace>A</replace>
    </binary>
    <binary action="substitute">
        <pattern>[a-z]</pattern>
        <replace>a</replace>
    </binary>
    <binary action="substitute">
        <pattern>[0-9]</pattern>
        <replace>0</replace>
    </binary>
    <binary action="substitute">
        <pattern>[^Aa0]</pattern>
        <replace>x</replace>
    </binary>
</nary>
```

Figure 9: exmaple of the *n-ary* feature "sequencer". The example here implements character classes (called *word shapes* in [6]).

6.1.2 *Boolean* features

Les *boolean features boolean* définissent des expressions booléennes. Un exemple de feature *boolean* est donné dans la figure 10. Trois actions sont disponibles:

- and: logical and. Takes two *boolean features* arguments.
- or: logical or. Take two *boolean features* arguments.
- not: logical not. Takes one *boolean feature* argument.

```
<boolean name="StartsWithUpper-AndNot-FirstWordOfSentence" action="and">
    <unary action="isUpper">0</unary>
    <boolean action="not">
        <nullary action="BOS" />
    </boolean>
</boolean>
```

Figure 10: example of a *boolean* feature.

6.1.3 *Dictionary* features

Dictionary *features* are features absed on lexica. The available actions are:

- token (*boolean feature*): checks whether a token belongs to a lexicon or not;
- multiword (*sequence feature*): looks for sequences of tokens belonging to the lexicon.

```
<dictionary name="token-dictionary" action="token" path="path/to/token-dictionary" />
```

Figure 11: example of the "token" *dictionary* feature.

```
<dictionary name="multiword-dictionary" action="multiword" path="path/to/multiword-dictionary" />
```

Figure 12: exmaple of the "multiword" *dictionary* feature.

6.1.4 *Directory* features

Directory features allow to use a directory of lexica as described in [5]. Two features are defined:

- directory (*sequence feature*): apply a directory of features. Unmatched tokens are replaced by "O". This *feature* expects a "path" argument that is the path to the folder where all the lexica are located;
- fill (*string feature*): replace elements by those given in the entry "filler-entry" if and only if it is matched by a *boolean feature* given in argument.

```
<directory name="NER-ontology" path="../../dictionaries/fr/NER-ontology" />
```

Figure 13: Example of a *directory* feature.

```
<fill name="NER-ontology-POS" entry="NER-ontology" filler-entry="POS">
    <string action="equal">O</string>
</fill>
```

Figure 14: example of a *directory* feature.

```
<list name="top-of-hierarchy" action="some">
    <string action="equal">PDG</string>
    <regexp action="check" entry="lower">^(président|directeur)$</regexp>
</list>
```

Figure 15: Exmaple of the *list* feature "some".

6.1.5 *List* features

The *list* feature is a boolean feature that allows to define a list of boolean features that will be evaluated. The following actions are available: *none* (all features should evaluate to false), *some* (at least one feature should evaluate to true) et *all* (all features should evaluate to true).

6.1.6 *Matcher* features

Regexp token features evaluate regular expressions. The following actions, illustrated on figures 16, 17 and 18 are available:

- action="check" (*boolean feature*): check whether the regexp is matched on the token or not.
- action="subsequence" (*string feature*): check whether the regexp is matched on the token or not and return the matched substring.
- action="token" (*string feature*): check whether the regexp is matched on the token or not and return the token if it is the case.

```
<regexp name="only-first-upper" action="check">^[A-Z][^A-Z]*$</regexp>
```

Figure 16: example of the *matcher* feature "check".

```
<regexp name="after-hyphen" action="subsequence">-.+$</regexp>
```

Figure 17: example of the *matcher* feature "subsequence".

```
<regexp name="ends-with-isme" action="token">isme$</regexp>
```

Figure 18: example of the *matcher* feature "token".

6.1.7 *Rule* features

The *sequence feature* "rule" allows to add simple rules as features. Arguments of a rule feature are always *boolean features* and all have a "card" field to indicare their cardinality. Different values for "card" are:

- ?: 0 or once
- *: 0 or an unlimited number of times
- +: 1 or an unlimited number of times

int : exactly [int] times

- "min,max": at least min times and at most max times.

a specific argument for rule features is "orrule" that allows to match a rule within a set of rules. An example of a rule feature is given in the figure 19.

```
<rule name="amount">
    <list action="some">
        <regexp action="check" entry="word">^[0-9]+$</regexp>
        <regexp action="check" entry="lower">^(une?|deux|trois|quatre|cinq|six|sept|huit|neuf|dix
|onze|douze|treize|quatorze|quinze|seize|dix-sept|dix-huit|dix-neuf|vingt|trente|quarante|cinquante|soixante
|soixante-dix|quatre-vingt|quatre-vingt-dix|cents?|mille|millions?|milliards?)$</regexp>
    </list>
    <orrule>
        <regexp action="check" entry="chunking" card="+">-NP$</regexp>
        <regexp action="check" entry="chunking" card="+">-PP$</regexp>
    </orrule>
</rule>
```

Figure 19: example of the *rule* feature.

6.1.8 *String* features

String token features define operations on strings. The following actions are available:

- equal (*boolean feature*): checks the equality between the string and the one given in argument. Defines the following options:
  - casing="(sensitive|s|insensitive|i)": defines case sensitivity of the operation (default: "sensitive").

```
<string action="equal" casing="sensitive">O</string>
```

Figure 20: example of the "equal" *string* feature.

6.1.9 *Trigger* features

The *trigger string feature* allows to define a trigger before evaluating another feature. It has two children: *trigger* which is the *boolean feature* to check beforehand and then any *token feature*. An example is given in figure 21.

```
<trigger name="Substitute-Numbers-Triggered">
    <regexp action="check">[0-9]</regexp>
    <binary action="substitute">
        <pattern>[0-9]+</pattern>
        <replace>0</replace>
    </binary>
</trigger>
```

Figure 21: example of the *trigger* feature.

6.2 For the tagger module

The tagger module configuration file is called the master file. It allows to defined pipelines and global options that apply to the input files and modules.

The master file is an XML file of document type "master". It has two parts: a *pipeline* that is a list of modules and an "option" part that allows to define global options. An example of such a file is given in figure 22.

```
<? xml version="1.0" encoding="UTF-8" ?>
<master>
  <pipeline>
    <segmentation tokeniser="fr" />
    <enrich informations="pos.xml" mode="label" />
    <annotate model="models/POS" field="POS" />
    <clean to-keep="word,POS" />
    <enrich informations="NER.xml" mode="label" />
    <annotate model="models/NER" field="NER" />
    <clean to-keep="word,POS,NER" />
  </pipeline>

  <options>
    <encoding input-encoding="utf-8" output-encoding="utf-8" />
    <log log_level="INFO" />
    <export format="html" pos="POS" ner="NER" lang="fr" lang_style="default.css" />
  </options>
</master>
```

Figure 22: Specification of a SEM pipeline.

7 Retrain SEM

Default SEM models may not be appropriate for every use case. To adapt SEM, it offers to train new models that it will be able to use later.

In the following, the step-by-step procedure to train new SEM models will be given. It will assume a very simple use case: recognize persons and softwares. To this end, we will follow two paths: one where the user has an already annotated BRAT document and the other one where the document will have to be manually annotated.

7.1 Retrain SEM from annotated files

7.1.1 Launch SEM GUI

To start SEM's GUI, launch in a terminal:

```
python -m sem gui
```

The SEM GUI should look like the one in the figure 23.

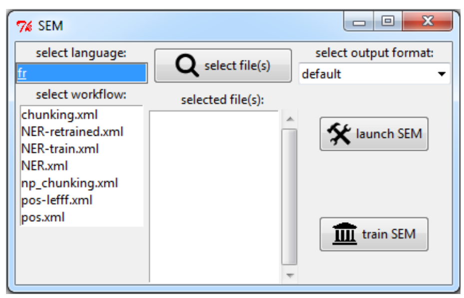

Figure 23: The GUI that allows to retrain SEM

7.1.2 Select data and preprocesses

Once launched, the user needs to select the following things:

- data used for training;
- the workflow to preprocess data.

To select the data that will be used for training, see figures 24 and 25. The different steps are :

1. click on the *select file(s)* button;
2. select annotated files;

3. click on the *open* button.

When they are selected, files will be listed as shown in figure 25. It is not necessary that selected files have the same format. They can have any format that is readable by SEM. Currently, the supported formats are:

- XML SEM, the internal XML format of SEM;
- json SEM, the internal json format of SEM;
- BRAT [9];
- GATE [4];

When annotated files are selected, a workflow has to be selected before processing documents. SEM offer an example of a workflow to retrain NER in figure 24: *NER-train.xml*.

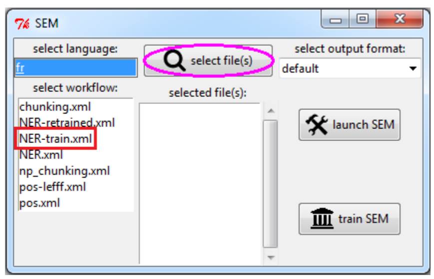

Figure 24: Circled in purple : the button to select documents for training. Framed in red : the workflow to use to train models.

7.1.3 Launch training

When the workflow and the training files are selected, click on the button *train SEM* as shown on figure 25. It will open the window allowing to set the parameters of the CRF to retrain SEM, as shown in figure 25. In this window, it is also possible to select a pattern file to train Wapiti. If none are selected, it will be automatically generated from the entries and features computed in the workflow. When all parameters are configured, click on the *train* button to launch the training of a new SEM model. When the training is finished, SEM will display where files are located on the computer, as shown in the figure 28. To use the model, copy the file *model.txt* in the folder *${SEM_DATA}/resources/models/fr/NER*.

7.2 Retrain SEM from unannotated files

When only unannotated files are available, it is necessary to first tag them. SEMallows the user to manually annotate raw text files.

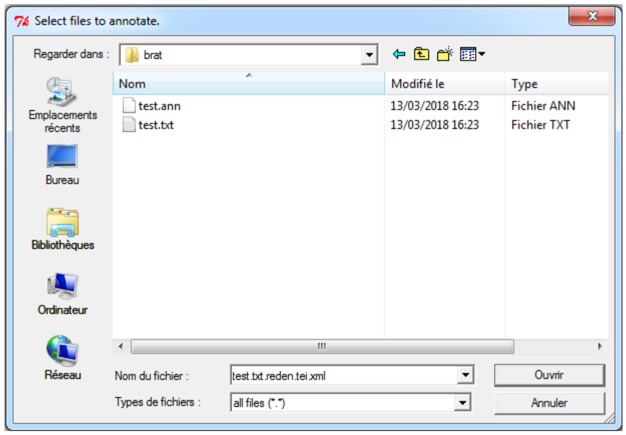

Figure 25: Examples of annotated files in BRAT format. Select ".ann" or ".txt" for training.

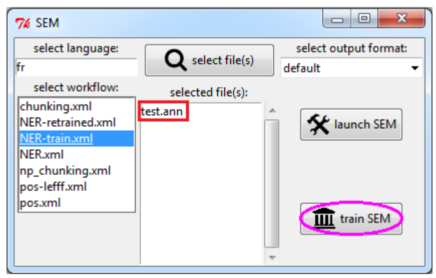

Figure 26: framed in red : documents used for training. Circled in purple : the button to retrain SEM.

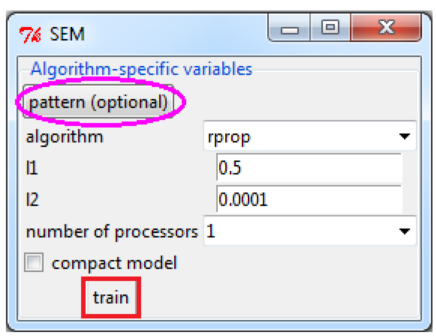

Figure 27: Circled in purple: the button to select a model. Framed in red: the button to launch the training.

7.2.1 Launch SEM GUI for manual annotation

To launch the SEM's manual annotation GUI, launch in a terminal:

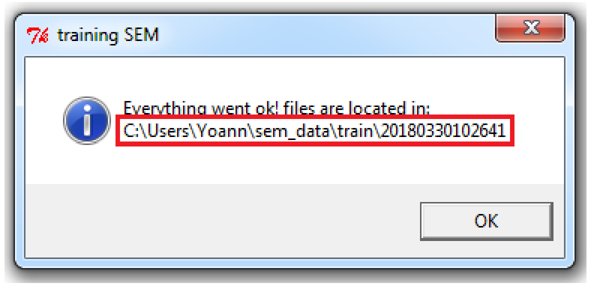

Figure 28: Framed in red : the path where to find the files on the computer.

```
python -m sem annotation_gui
```

The interface shown in figure 29 should display.

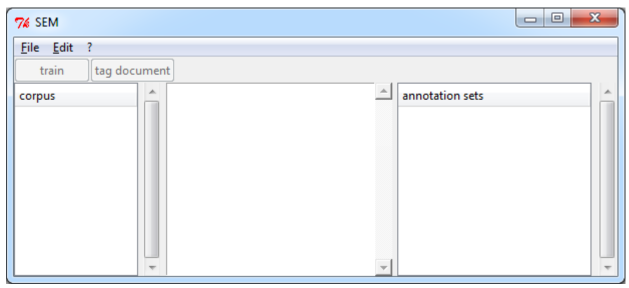

Figure 29: SEM's manual annotation GUI at launch.

7.2.2 Manually annotate with SEM annotation GUI

To make process simpler, SEM disallows the user to modify the content of the files or modify a loaded tagset. To annotate a document, the following should be done:

- load the document from a file;
- load the tagset from a file;

SEM will generate keyboard shortcuts from the loaded tagset. A tagset is a text file containing one tag per line, such as the following:

```
tag1

# a comment
tag2.subtag1
```

```
tag2.subtag2 # another comment

tag3
```

SEM handles hierarchical tags and considers the caracter '.' as the separator between levels. Empty lines are ignored and the character "#" allows to write comment that will be ignored. In our case, we wish to handle the tags *software* ("logiciel" in the figures) and *person* ("personne" in the figures). The tagset file will then have the following content :

```
software
person
```

Figures 30 and 31 show how to load a document and a tagset.

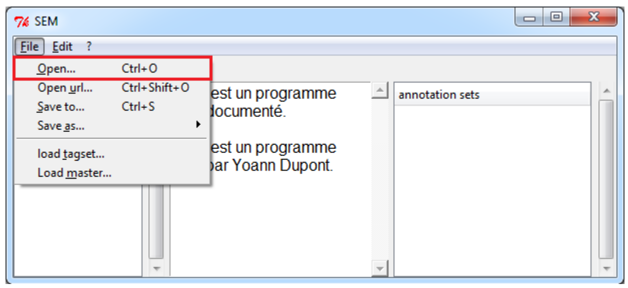

Figure 30: Framed in red : the menu element to load a document.

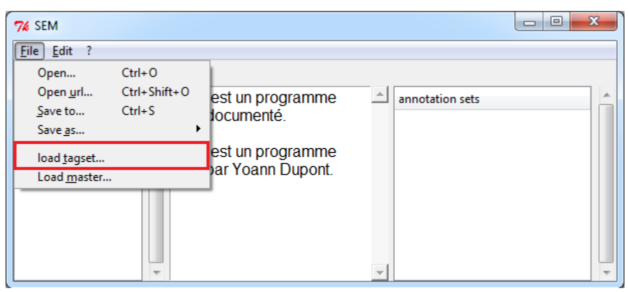

Figure 31: Framed in red : the button to load a tagset.

Figures 32, 33 and 34 how to manually annotate a corpus then retrain a model SEM can use with this corpus.

7.3 Use the new model

Pour annoter des documents avec le nouveau modèle, il faut alors sélectionner la chaîne de prétraitement « NER-retrained.xml » puis cliquer sur le bouton « launch SEM » comme

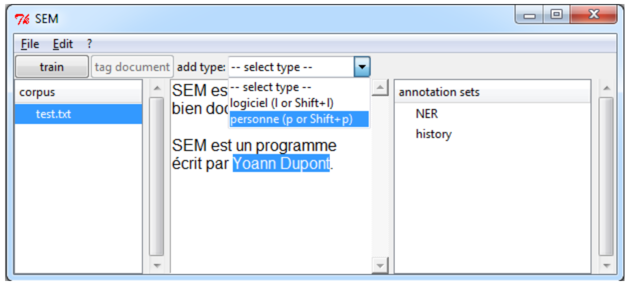

Figure 32: To annotate an element: select the text and choose the tag in the list (which gives the keyboard shortcuts).

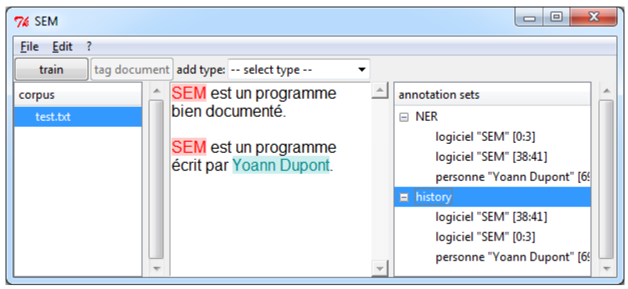

Figure 33: To annotate all occurrences in the document : use keyboard shortcut $< Shift + \_ >$ where "_" is the keyboard shortcut for the tag. If we look at figure 32, $< Shift + l >$ allows to annotate all occurrences of "SEM" as "software".

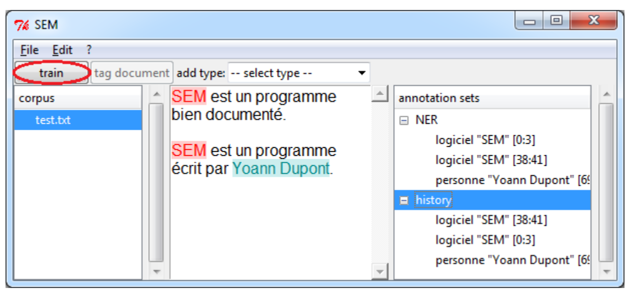

Figure 34: Framed in red : the button to train SEM with the corpus and its current annotations. The interface is identical to the one shown in section 7.1.3.

illustré dans la figure 35. Une fois le traitement effectué, SEM indiquera où trouver les fichiers annotés comme illustré dans la figure 36. To annotate documents with a new model, you need to select the workflow called *NER-retrained.xml* then click on the button *launch SEM* as illustrated in figure 35. Once the processing is done, SEM will tell where to find the newly annotated files as shown in figure 36.

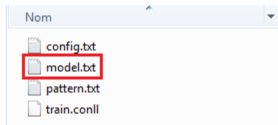

Figure 35: framed in red: the model file to copy in the folder ${SEM_DATA}/resources/models/fr/NER

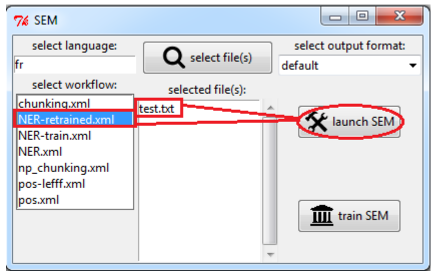

Figure 36: framed in red : the workflow and the file to annotate. Circled in red : the button to launch SEM.

References

[1] A. Abeillé, L. Clément, and F. Toussenel. "Building a treebank for French". In: *Treebanks*. Ed. by A. Abeillé. Dordrecht: Kluwer, 2003.

[2] Lionel Clément, Benoît Sagot, and Bernard Lang. "Morphology Based Automatic Acquisition of Large-coverage Lexica". In: *Proceedings of the Fourth International Conference on Language Resources and Evaluation, LREC 2004*. Lisbon, Portugal: European Language Resources Association, May 2004.

[3] B. Crabbé and M. -H. Candito. "Expériences d'analyse syntaxique statistique du français". In: *Actes de TALN'08*. 2008.

[4] Hamish Cunningham et al. "GATE: an architecture for development of robust HLT applications". In: *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics. 2002, pp. 168–175.

[5] Yoann Dupont. "Exploration de traits pour la reconnaissance d'entités nommées du Français par apprentissage automatique". In: *24e Conférence sur le Traitement Automatique des Langues Naturelles (TALN)*. 2017, p. 42.

[6] Jenny Finkel et al. "Exploiting context for biomedical entity recognition: From syntax to the web". In: *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications*. Association for Computational Linguistics. 2004, pp. 88–91.

[7] Thomas Lavergne, Olivier Cappé, and François Yvon. "Practical Very Large Scale CRFs". In: *Proceedings the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*. Uppsala, Sweden: Association for Computational Linguistics, July 2010, pp. 504–513. URL: http://www.aclweb.org/anthology/P10-1052.

[8] Benoît Sagot, Marion Richard, and Rosa Stern. "Annotation référentielle du Corpus Arboré de Paris 7 en entités nommées". In: *Actes de la 19e conférence sur le Traitement Automatique des Langues Naturelles*. Grenoble, France: Association pour le Traitement Automatique des Langues, June 2012, pp. 535–542. URL: http://www.atala.org/taln_archives/TALN/TALN-2012/taln-2012-court-026.

[9] Pontus Stenetorp et al. "BRAT: a web-based tool for NLP-assisted text annotation". In: *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. 2012, pp. 102–107.

[10] I. Tellier, Y. Dupont, and A. Courmet. "Un segmenteur-étiqueteur et un chunker pour le français". In: *Actes de TALN'12, session démo*. 2012.

[11] I. Tellier et al. "Apprentissage automatique d'un chunker pour le français". In: *Actes de TALN'12, papier court (poster)*. 2012.