sulci Documentation

Release 0.1.alpha0

Yohan Boniface

August 06, 2012

CONTENTS

Sulci is a French text mining tool, initially designed for the analysis of the corpus and thesaurus of Libération, a French newspaper.

This code is "work in progress", but it's yet used in production at Libération.

Therefore, here is a demo page with the frozen 0.1 alpha version:

http://alpha.sulci.dotcloud.com

Sulci provides 4 algorithms, designed to be run in sequence: each algorithm needs the data provided by the previous one :

1. Part of Speech tagging
2. Lemmatization
3. Collocation and key entities extraction
4. Semantical tagging

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Configure the env

Sulci is built on top of Django, and therefore you will need Django installed and an active project to be able to use sulci.

Also, a SQL database (we use Postgresql) is needed.

1.1.2 Import the trained data

Retrieve the three files of the three tables:

1. sulci_descriptor
2. sulci_trigger
3. sulci_triggertodescriptor

Warning: Be careful to load at last the sulci_triggertodescriptor, as it has FK to the others.

Note: You may have to change the table owner handly in the SQL files.

Load these three files into your database.

1.1.3 Install sulci

A easy way is to use pip:

```
pip install git+git@github.com:yohanboniface/sulci.git
```

Note: A very good habit is to use a *virtualenv* to do so.

Or you can retrieve the code, and "python setup.py install" or put the sulci folder in your PYTHONPATH:

```
$ export PYTHONPATH=$PYTHONPATH:`pwd`
```

Then add "sulci" to your INSTALLED_APPS.

1.1.4 Try it from shell

Sample usage:

```
>>> from sulci.textmining import SemanticalTagger
>>> text = u"""«La Russie et la Chine finiront par regretter leur décision qui
             les a vues s'aligner sur un dictateur en fin de vie et qui les a
             mises en porte-à-faux avec le peuple syrien.»"""
>>> s = SemanticalTagger(text)
>>> s.descriptors
[(<Descriptor: Russie>, 100.0),
 (<Descriptor: Chine>, 100.0),
 (<Descriptor: diplomatie>, 14.798308089447328),
 (<Descriptor: Dmitri Medvedev>, 10.337552742616033)]
```

1.1.5 Test Page

A simple view is provided to easily test Sulci. You need to add the url in your urlconfs, for example:

```
url(r'^sulci/demo$', 'sulci.views.demo', name='sulci_demo'),
```

Then you'll have to start the Django runserver and browse the page http://localhost:8000/sulci/demo.

1.1.6 Configure for training

If you plan to train you own Sulci or to use the command line, you have to add these settings (with you own values, of course):

```
SULCI_CLI_CONTENT_MANAGER_METHOD_NAME = 'objects'
SULCI_CLI_CONTENT_APP_NAME = 'libe'
SULCI_CLI_CONTENT_MODEL_NAME = 'article'
SULCI_CLI_KEYWORDS_PROPERTY = 'keywords'
SULCI_CLI_CONTENT_PROPERTY = "content"
```

1.2 Overview

Sulci provides 4 algorithms, designed to be run in sequence: each algorithm needs the data provided by the previous one :

1. Part of Speech tagging
2. Lemmatization
3. Collocation and key entities extraction
4. Semantical tagging

Each algorithm must be *trained* to be operational. Therefore, a trained set of data is provided (see fixtures/). This is the result of a training with the Libération corpus.

Therefore, if you want a text mining most accurate to your own texts, you must train it.

What does "training" mean?

1. You first need to prepare a corpus, for each of the algorithms. A corpus is a set of texts with the final output you want : for example, the corpus for the PosTagger is some texts where each word is pos tagged, with a verified value.

2. You then have to launch the training with the command line.
3. Each algorithm has it's own trainer. This trainer will take the input texts, remove the verified output, and begin to find the best rules to determine the right output. The trainer will analyze its mistakes, and change its processing rules/weights/triggers/... in order to avoid making those mistakes again in the future. The trainer will loop over the corpus until it considers that no more rules could be inferred.

Note: You can also use the algorithm to help you create the corpus : give a text to the algorithm, and correct the output.

Warning: each algorithm needs the previous algorithm to work, so remember to train the algorithms in the order they are called.

Note: The trained data provided with the Sulci alpha version has been made with a corpus of : #. 15500 POS tagged words #. 2000 words in lexicon (lexicon must be smaller than POS corpus) #. FIXME: 2000 lemmatized words #. 28000 semantical tagged words #. 17000 descriptors in thesaurus

---

Before running the first algorithm, the text is split into tokens (words, symbols, punctuation marks, etc.), using simple regular expressions.

1.2.1 Part-Of-Speech tagging

The PosTagger finds out the "POS tag", or "lexical class", or "lexical category" of each word. The algorithm used is similar to the Brill POS-tagging algorithm.

Some possible classes are:

- VCJ:sg (Verbe ConJugé singulier),
- PAR:pl (PARticipe pluriel),
- PREP (PREPosition),
- etc.

To see more available classes, see in base.py the methods named is_<something> or run the following command (it provides the tag stats in corpus):

```
python manage.py sulci_cli -g
```

The format used is a plain text file; tokens are separated by spaces; each token is annotated with its POS tag, separated with a slash. Example:

```
Mon/DTN:sg cher/ADJ:sg Fred/SBP:sg ,/, Je/PRV:sg quitte/VCJ:sg
Paris/SBP:sg demain/ADV matin/SBC:sg ./.
```

This format combines "input" and "output": the input is the token, the output is the POS tag.

Check "corpus/*.crp" to see more examples of "valid output".

1.2.2 Lemmatization

The Lemmatizer tries to find the *Lemma* of each word. The lemma of a word is almost similar to its stem. A few examples:

- "mangerons" lemma is "manger" (infinitive form of the verb)

- "bonnes" lemma is "bon" (masculine singular form of the adjective)
- "filles" lemma is "fille" (singular form of the noun)

The format used is similar to the one of the PosTagger, but each token is annotated by both its POS tag and its lemma. Example:

```
«/« Ce/PRV:sg/ce n'/ADV/ne est/ECJ:sg/être pas/ADV à/PREP moi/PRO:sg
de/PREP partir/VNCFF ./. Je/PRV:sg/je me/PRV:sg battrai/VCJ:sg/battre
jusqu'/PREP/jusque au/DTC:sg bout/SBC:sg ./. »/»
```

If a word and its lemma are identical, the lemma is omitted. Note that this is case-sensitive (as you can see on the first word of the above excerpt).

Check "corpus/*.lem.lxc.crp" to see more examples of "valid output".

1.2.3 Semantical tagging (Collocation and key entities extraction)

The Semantical Tagger tries to find "collocations" – i.e., sequence of tokens that have a higher chance of appearing together – and key entities – i.e. words that may help to find the significance of the text : proper nouns, for example, or ones with many occurrences in the text, etc. A few examples:

- Président de la République
- voiture électrique
- Barack Obama

The Semantical Tagger will actually use two different algorithms:

- a purely statistical algorithm, scoring n-grams according to their relatives frequencies in the corpus.
- an heuristics-based algorithm, scoring n-grams (sequences of words) with hand-crafted rules;

The statistical algorithm uses Point-wise mutual information.

The first one is mainly used do determine whether or not a sequence of words is a collocation ; the second one, to determine whether or not a word or a collocation is representative of the text.

1.2.4 Semantical training

- each text of the semantical corpus is processed by the previous algorithm, to find the key entities (triggers)
- each of the triggers found are linked with a weight to the descriptors to the text processed
- finally, the more a trigger was linked to a descriptors, the more this trigger will trigger the descriptors in the tagging process.

1.2.5 After the training phase...

Once all algorithms have been trained to a satisfactory level, they are ready to analyze new texts without your guidance (i.e., you won't have to pre-tag those texts, indeed).

Steps 1 to 4 are run in sequence, and trigger to descriptors relations are used to extract the must pertinent descriptors.

1.3 Example of full training

Warning: the training of Sulci is a hard North face, be sure to have the minimum of French knowledge, some time, some pre-categorized texts, some fast computer, before taking this way

Warning: each algorithm needs the previous algorithm to work, so remember to train the algorithms in the order they are called.

Note: The trained data provided with the Sulci alpha version has been made with a corpus of :

1. 30000 POS tagged words
2. 3500 words in lexicon (lexicon must be smaller than POS corpus)
3. FIXME: 2000 lemmatized words
4. 40000 semantical tagged texts
5. 17000 descriptors in thesaurus

1.3.1 Lexical training

First, we need to create some text corpus, in two groups:

- one group with texts where only the POS tag for each word is set. Example:

  ```
  Tout/PRV:sg était/ECJ:sg tellement/ADV absurde/ADJ:sg et/COO compliqué/ADJ:sg
  ```

  These texts need to have the *.crp* extension ; this group must be bigger.

- one other with texts where both the POS tag and the lemme are set. Example:

  ```
  Dans/PREP/dans les/DTN:pl/le faits/SBC:pl/fait ,/, la/DTN:sg/le répression/SBC:sg
  est/ECJ:sg/être contrebalancée/PAR:sg/contrebalancer
  ```

  These texts will be used to build the lexicon ; the valid extension is *.lxc.lem.crp* ; this group must be smaller.

Note: You can also use the algorithm to help you create the corpus : give a text to the algorithm, and correct the output.

Then, we can build the lexicon:

```
./manage.py sulci_train -x
```

This will write the new lexicon in temporary *.pdg* (pending) file. For now, we have to manually rename it in *lexicon.lxc* if the result is ok for us.

Now, we can launch the lexical training:

```
./manage.py sulci_train -e
```

or, to load-balance the work in more than one process (using zmq), here one master and 4 slaves subprocesses:

```
./manage.py sulci_train -e -s 4
```

Another time, we have to manually rename the file generated in */corpus/* from *lexical_rules.pdg* to *lexical_rules.rls*.

Then, we can launch the contextual training (remember to rename the file after):

```
./manage.py sulci_train -c -s 4
```

1.3.2 Lemmatization

Now, the lemmatizer trainer:

```
./manage.py sulci_train -r -s 4
```

1.3.3 Semantical training

Now, the last step, but the bigger : the semantical training. Here a big corpus of categorized texts is needed. For example, in Libération we are using now a corpus of 35000 texts.

Make sure you have configured the needed settings (see Installation below).

Then launch the command line:

```
./manage.py sulci_train -n -s 4
```

1.3.4 Postprocessing

Finally, we can clean manually to reduce noise and remove useless rows, for example, removing all synapses that have been seen just one time (triggertodescriptor.weight == 1) or those where the pondered_weight is too low (triggertodescriptor.pondered_weight < 0.01 for example). And after that, triggers with no synapse can be also deleted.

1.4 How can I help?

- You're a python killer: there is many optimizations to do in the actual code
- You're a language expert: all the algorithm can be optimized
- You know well French language: you can add texts in the POS corpus, or make a proof read of the actual texts (in corpus/*.crp)
- You're an enthusiast: you can play with the demo, with the debug, and make tickets for the bug seen ; you can help for making the doc, etc.
- in any case, Meet us for IRC chats: #sulci on irc.freenode.net

1.5 Management Commands

Sulci comes with three management commands. They are tools useful when you work **on** Sulci.

Note: You need do define the Sulci settings to be able to use the management commands.

1.5.1 sulci_cli

The `sulci_cli` command lets you run the semantical tagger. In addition to the standard management command options, you must provide the following argument:

```
``--pk``:
    The value is the pk of the content you want to process.
```

Ex.:

```
$ ./manage.py sulci_cli --pk=746007
```

1.5.2 sulci_monit

The `sulci_monit` command lets you inspect the corpus and lexicon data. In addition to the standard management command options, you can provide the following arguments:

```
``--check_corpus,-u``:
    The action will be run on the corpus.
``--check_lexicon,-x``:
    The action will be run on the lexicon.
``--count,-c``:
    Run a count on the selected set of data.
``--word,-w``:
    Search for a word.
``--tags_stats,-g``:
    Display tags stats on the selected set of data.
``--lemme,-M``:
    Specify a lemme.
``--tag,-t``:
    Specify a tag.
``--path,-p``:
    Specify a path.
``--case_insensitive,-i``:
    Case insensitive.
```

Count words in corpus:

```
./manage.py sulci_monit -uc
```

Count entries in lexicon:

```
./manage.py sulci_monit -xc
```

Search for occurrences of 'bateau' in corpus:

```
./manage.py sulci_monit -uw bateau
```

Search for occurrences of "été" in corpus where tag is "SBC:sg":

```
./manage.py sulci_monit -uw été -t SBC:sg
```

Search for tag stats of word "révolutionnaires":

```
./manage.py sulci_monit -uw révolutionnaires -g
```

Search for occurrences of "relève" where lemme is "relever":

```
./manage.py sulci_monit -uw relève -M relever
```

Search for occurrences of "tout" case insensitive:

```
./manage.py sulci_monit -uiw tout
```

1.5.3 sulci_train

The `sulci_train` command lets you train Sulci. In addition to the standard management command options, you can provide the following arguments:

```
``--lexicon,-x``:
    Build the lexicon.
``--lexical,-e``:
    Launch the lexical trainer.
``--contextual,-c``:
    Launch the contextual trainer.
``--lemmatizer,-r``:
    Launch the lemmatizer trainer.
``--semantical,-n``:
    Launch the semantical trainer.
``--subprocesses,-s``:
    Launch trainer with w subprocesses (using zeromq).
``--add_candidate,-a``:
    Prepare a content to manual POS tagging (before adding it in corpus).
``--add_lemmes,-a``:
    When preparing a content to POS tagging, prepare lemmes to.
```

Build lexicon:

```
./manage.py sulci_train -x
```

Launch lexical training with 4 subprocesses:

```
./manage.py sulci_train -e -s 4
```

Add a content for POS tagging

./manage.py sulci_train -a –pk=123456

1.6 Sulci Package

1.6.1 base Module

class sulci.base.**RetrievableObject**

Bases: object

Simple abstract class to manage RAM stored and retrievable objects.

classmethod get_or_create(*ref*, *parent_container*, ***kwargs*)

Here, objects are created within a parent container. For exemple, the text, or a sample, or a lexicon, ecc. The store field is build from the name of the class.

classmethod make_key(*expression*)

Make a standardization in the expression to return a tuple who maximise maching possibilities. expression must be a list or tuple, or string or unicode

classmethod sort(*seq*, *attr*, *reverse=True*)

class sulci.base.**Sample**(*pk*, *parent=None*, ***kwargs*)
Bases: sulci.base.RetrievableObject

A sentence of the text.

append(*item*)

get_errors(*attr='tag'*)
Retrieve errors, comparing attr and verified_attr. Possible values are : tag, lemme.

has_position(*pos*)

is_token(*stemm*, *position*)
Check if there is stemm "stemm" in position "position".

meaning_words_count()

reset_trainer_status()
This method has to be called by the trainer each time a token of this sample is modified.

set_trained_position(*pos*)
This method has to be called by trainer each time a token is processed but not corrected.

show_context(*position*)
Returns a string of tokens around some positin of the sample.

class sulci.base.**TextManager**
Bases: object

This is an abstract class for all the "text", i.e. collection of samples and tokens.

PENDING_EXT = None

VALID_EXT = None

get_files(*kind*)

instantiate_text(*text*)
return samples and tokens. text is tokenized each token is : original + optionnal verified_tag (for training)

load_valid_files()

pending_files

tokenize(*text*)

valid_files

class sulci.base.**Token**(*pk*, *original*, *parent=None*, *position=0*, ***kwargs*)
Bases: sulci.base.RetrievableObject

Simplest element of a text.

begin_of_sample(*previous_token*)

get_neighbors(**args*)
Returns tokens neighbors in sample in positions passed as args, if available.

Eg. token.get_neighbors(1, 2) will return the next and next again tokens.

has_meaning()
What about isdigit ?

has_meaning_alone()
Do we take it in count if alone?

has_verified_tag(*tag*)

is_avoir()

is_closing_quote()

is_etre()

is_neighbor(*candidates*)
Return true if word appears with right neighbours. False otherwise. candidates is tuple (Stemm object, distance)

is_opening_quote()

is_strong_punctuation()

is_tagged(*tag*)

is_tool_word()
Try to define if this word is a "mot outil".

is_verb()
We don't take in count the verbs Etre and Avoir.

istitle()
Determine if the token is a title, using its tag.

lower()

next_bigram
Return the two next token, or None if there is not two tokens after.

previous_bigram
Return the two previous token, or None if there is not two tokens before.

sample
For retrocompatibility.

show_context()

1.6.2 textmining Module

class sulci.textmining.**KeyEntity**(*pk*, ***kwargs*)
Bases: sulci.base.RetrievableObject

collocation_confidence

compute_confidence()
Compute scores that will be used to order, select, deduplicate keytentities.

confidence

count = 0

frequency_confidence()
Lets define that a ngram of 10 for a text of 100 words means 1 of confidence, so 0.1

frequency_relative_pmi_confidence

heuristical_mutual_information_confidence()
Return the probability of all the terms of the ngram to appear together. The matter is to understand the dependance or independance of the terms. If just some terms appears out of this context, it may be normal (for exemple, a name, which appaers sometimes with both firstname and lastname and sometimes with just lastname). And if these terms appears many many times, but some others appears just in this context,

the number doesn't count. If NO term appears out of this context, with have a good probability for a collocation. If each term appears out of this context, and specialy if this occurs often, we can doubt of this collocation candidate. Do we may consider the stop_words ? This may affect negativly and positivly the main confidence.

index(*key*)

is_duplicate(*KeyEntity*)
Say two keyentities are duplicate if one is contained in the other.

is_equal(*other*)
This is for confidence and length comparison. NOT for content comparison.

istitle()
A keyEntity is a title when all its stemms are title.

keyconcept_confidence

merge(*other*)
Other is merged in self. Merging equal to say that other and self are the same KeyEntity, and self is the "delegate" of other. So (this is the case if other is smaller than self) each time other appears without the specific terms of self, we concider that is the same concept. So, we keep the highest frequency_confidence.

nrelative_frequency_confidence()
This is the frequency of the entity relatively to the possible entity of its length.

pos_confidence()
Give a score linked to the POS of the subelements.

statistical_mutual_information_confidence()
Number of occurrences of the ngram / number of ngrams possible / probability of each member of the ngram.

thesaurus_confidence()
Try to find a descriptor in thesaurus, calculate levenshtein distance, and make a score. This may not be < 1, because if there is a descriptor, is a good point for the collocation, but if not, is doesn't means that this is not a real collocation.

title_confidence()
Define the probability of a ngram to be a title. Factor is for the confidence coex max. This may not have a negative effect, just positive : a title is a good candidate to be a collocation but this doesn't means that if it's not a title it's not a collocation. Two things have importance here : the proportion of title AND the number of titles. Ex. : - "Jérôme Bourreau" is "more" title than "Bourreau" - "projet de loi Création et Internet" is "less" title than "loi Création et Internet"

trigger_score
Score used by trigger, may be the final confidence ?

class sulci.textmining.**SemanticalTagger**(*text*, *thesaurus=None*, *pos_tagger=None*, *lemmatizer=None*, *lexicon=None*)

Bases: object

Main class.

debug()

deduplicate_keyentities()
If a KeyEntity is contained in an other (same stemms in same place) longuer delete the one with the smaller confidence, or the shortest if same confidence We have to begin from the shortest ones.

descriptors

filter_ngram(*candidate*)
Here we try to keep the right ngrams to make keyentities.

get_descriptors(*min_score=10*)
Final descriptors for the text.

Only descriptors triggered up to min_score will be returned.

keyentities_for_trainer()

keystems(*min_count=3*)

make_keyentities(*min_length=2*, *max_length=10*, *min_count=2*)

ngrams(*min_length=2*, *max_length=15*, *min_count=2*)

triggers
Select triggers available for the current keyentities.

class sulci.textmining.**Stemm**(*pk*, ***kwargs*)
Bases: sulci.base.RetrievableObject

Subpart of text, grouped by meaning (stem). This try to be the *core* meaning of a word, so many tokens can point to the same stemm. Should be renamed in Lemm, because we are talking about lemmatisation, not stemmatisation.

count
Number of occurrences of this stemm.

has_interest()
Do we take it in count as potential KeyEntity? If count is less than x, but main_occurrence is a title, we try to keep it

has_interest_alone()
Do we take it in count if alone ?? If count is less than x, but main_occurrence is a title, we try to keep it

is_valid()

is_valid_alone()

istitle()

main_occurrence
Returns the "main" one from the linked tokens.

tag

class sulci.textmining.**StemmedText**(*text*, *pos_tagger=None*, *lemmatizer=None*, *lexicon=None*)
Bases: sulci.base.TextManager

Basic text class, with tokens, samples, etc.

create_stemm()

distinct_words()

distincts_meaning_words()

make()
Text is expected to be tokenized. And filtered ?

meaning_words

meaning_words_count()
Return the number of words in the text.

medium_word_count

stemms

words

words_count()
Return the number of words in the text.

1.6.3 corpus Module

class sulci.corpus.**Corpus**(*extension='.crp'*, *tagger=None*)
Bases: sulci.corpus.CorpusMonitor

The corpus is a collection of manualy categorised texts.

We have different kind of categorised texts :

•.crp => just POS tag

•.lem... => also manualy lemmatized

•.lcx... => will be used to make the Lexicon

When loading a Corpus, you'll need to specify the kind of texts to load.

LEXICON_EXT = '.lxc'

NEW_EXT = '.new'

PATH = 'corpus'

PENDING_EXT = '.pdg'

VALID_EXT = '.crp'

files
Return a list of files for the corpus extension.

samples

texts

tokens

class sulci.corpus.**CorpusMonitor**
Bases: object

Convenience class to store common methors between Corpus and TextCorpus.

check(*lexicon*, *check_lemmes=False*)
Check the text of the corpus, and try to determine if there are some errors. Compare with lexicon.

check_usage(*word=None*, *tag=None*, *lemme=None*, *case_insensitive=False*)
Find occurrences of a word or tag or both in the corpus loaded.

tags_stats(*word=None*, *case_insensitive=None*)
Display tags usage stats.

class sulci.corpus.**TextCorpus**(*path=None*)
Bases: sulci.base.TextManager, sulci.corpus.CorpusMonitor

One single text of the corpus.

This is not a raw text, but a manualy categorized text.

The normalisation is : word/TAG/lemme word2/TAG2/lemme2, etc.

LEXICON_EXT = '.lxc.lem.crp'

PATH = 'corpus'

PENDING_EXT = '.pdg'

VALID_EXT = '.crp'

export(*name*, *force=False*, *add_lemmes=False*)
Export tokens in a file.

force for export in the valid extension, otherwise it use the pending.

has_verified_lemmes
Returns True if the text is supposed to contains verified lemmes.

load()

prepare(*text*, *tagger*, *lemmatizer*)
Given a raw text, clean it, and make tokens and samples.

(Maybe this method should be in the TextManager class.)

samples

tokens

1.6.4 thesaurus Module

class sulci.thesaurus.**Descriptor**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Entries of the Thesaurus.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Descriptor.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Descriptor.**aliases**

Descriptor.**children**

Descriptor.**is_alias_of**

Descriptor.**max_weight**

Descriptor.**objects = <django.db.models.manager.Manager object at 0x282d250>**

Descriptor.**original**

Descriptor.**parent**

Descriptor.**primeval**
Returns the primeval descriptor when self is alias of another.

Descriptor.**trigger_set**

Descriptor.**triggertodescriptor_set**

class sulci.thesaurus.**Thesaurus**(*path='thesaurus.txt'*)
Bases: object

load_triggers()

normalize_item(*item*)

classmethod **reset_triggers**()
For full training, we need to remove previous triggers.

triggers

class sulci.thesaurus.**Trigger**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

The trigger is a keyentity who suggest some descriptors when in a text. It is linked to one or more descriptors, and the distance of the link between the trigger and a descriptor is stored in the relation. This score is populated during the sementical training.

exception **DoesNotExist**
Bases: django.core.exceptions.ObjectDoesNotExist

exception Trigger.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

classmethod Trigger.**clean_all_connections**()

Trigger.**clean_connections**()
Remove the negative connections.

Trigger.**connect**(*descriptor*, *score*)
Create a connection with the descriptor if doesn't yet exists. In each case, update the connection weight. Delete the connection if the score is negative.

Trigger.**descriptors**

Trigger.**export**()
Return a string for file storage.

Trigger.**items**()

Trigger.**max_weight**

Trigger.**objects** = <django.db.models.manager.Manager object at 0x282de10>

Trigger.**triggertodescriptor_set**

class sulci.thesaurus.**TriggerToDescriptor**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

This is the "synapse" of the trigger to descriptor relation.

exception **DoesNotExist**
Bases: django.core.exceptions.ObjectDoesNotExist

exception TriggerToDescriptor.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

TriggerToDescriptor.**descriptor**

TriggerToDescriptor.**objects** = <django.db.models.manager.Manager object at 0x282d890>

TriggerToDescriptor.**pondered_weight**
Give the weight of the relation, relative to the max weight of the trigger and the max weight of the descriptor.

TriggerToDescriptor.**trigger**

1.6.5 trainers Module

class sulci.trainers.**ContextualTrainer**(*tagger*, *corpus*, *mode='full'*)
Bases: sulci.trainers.POSTrainer

pretrain()
Tag the tokens, but not using the contextual rules, as we are training it.

template_generator
alias of ContextualTemplateGenerator

class sulci.trainers.**LemmatizerTrainer**(*lemmatizer*, *mode='full'*)
Bases: sulci.trainers.RuleTrainer

Train the Lemmatizer.

attr_name = 'lemme'

get_template_instance(*tpl*)

log_error(*token*)

pretrain()
We need to have the right tags, here

select_one_rule(*rules*)
Having a set of rules candidate for correcting some error, select the one correcting the more case, and creating the less errors.

template_generator
alias of LemmatizerTemplateGenerator

class sulci.trainers.**LexicalTrainer**(*tagger*, *corpus*, *mode='full'*)
Bases: sulci.trainers.POSTrainer

get_errors()
We don't care about token in Lexicon, for lexical trainer.

pretrain()
Tag the tokens, but not using POS rules, as we are training it.

template_generator
alias of LexicalTemplateGenerator

class sulci.trainers.**POSTrainer**(*tagger*, *corpus*, *mode='full'*)
Bases: sulci.trainers.RuleTrainer

Pos Tagger trainer.

attr_name = 'tag'

get_template_instance(*tpl*)

log_error(*token*)

pretrain()

select_one_rule(*rules*)

class sulci.trainers.**RuleTrainer**
Bases: object

Main trainer class for rules based, for factorisation.

display_errors()
Display errors in current step.

do()

get_errors()
Retrieve token where tag !== verified_tag.

get_template_instance(*tpl*)

log_error(*token*)

pretrain()
Trainer specific training session preparation.

select_one_rule(*rules*)

setup_socket_master()
Configure the sockets for the master trainer.

setup_socket_slave()
Configure sockets for the workers (slaves).

slave()

test_rule(*rule*)

test_rules(*rules_candidates*)

train()
Main factorized train method.

class sulci.trainers.**SemanticalTrainer**(*thesaurus*, *pos_tagger*, *mode='full'*)
Bases: object

Create and update triggers. And make triggertodescription ponderation.

PENDING_EXT = '.pdg'

VALID_EXT = '.trg'

begin()
Make one trigger for each descriptor of the thesaurus. Have to be called one time at the begining, and that's all.

clean_connections()
Delete all the connection where score < 0.

do(**args*)

setup_socket_master()
Configure the sockets for the master trainer.

setup_socket_slave()
Configure sockets for the workers (slaves).

slave()

train(*inst*)
For the moment, human defined descriptors are a string with "," separator.

1.6.6 rules_templates Module

class sulci.rules_templates.**CHANGESUFFIX**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LemmatizerBaseTemplate

Make the original lower, if the tag is x.

apply_rule(*tokens*, *rule*)

compile_rule(*tag*, *to_delete*, *to_add*)

is_candidate(*token*, *rule*)

make_rules(*token*)
We make one rule for each possible transformation making verified_lemme from token.original.

test_rule(*token*, *rule*)

uncompile_rule(*rule*)

class sulci.rules_templates.**CURWD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

The word is X. I have doubt on the interest of this template...

get_target()

class sulci.rules_templates.**ContextualBaseTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.RuleTemplate

Base class for the contextual rules.

compile_rule(*from_tag*, *to_tag*, *complement*)
Make the final rule string. complement must be an iterable

classmethod uncompile_rule(*rule*)

class sulci.rules_templates.**ContextualTemplateGenerator**
Bases: type

classmethod export(*rules*)
Export rules to the provisory config file.

rules are tuples (rule, score).

classmethod get_instance(*s*, ***kwargs*)
Returns and instance of a rule, from a template name or a rule string.

s can be template name or rule.

classmethod load()
Load rules from config file.

register = {'NEXTWD': <class 'sulci.rules_templates.NEXTWD'>, 'WDAND2AFT': <class 'sulci.rules_templates.WD

class sulci.rules_templates.**FORCELEMME**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LemmatizerBaseTemplate

Give lemme y, if the tag is x.

apply_rule(*tokens*, *rule*)

compile_rule(*tag*, *lemme*)

is_candidate(*token*, *rule*)

make_rules(*token*)

test_rule(*token*, *rule*)

class sulci.rules_templates.**LBIGRAM**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**LemmatizerBaseTemplate**(*pk*, ***kwargs*)
Bases: sulci.base.RetrievableObject

For the Lemmatizer training, the is just one template : it create as many possible rules as letters in the token tested. MAKELOWER GIVELEMME CHANGESUFFIX

compile_rule()

is_candidate(*token*, *rule*)

make_rules(*token*)

test_rule(*token*, *rule*)

uncompile_rule(*rule*)

class sulci.rules_templates.**LemmatizerTemplateGenerator**
Bases: type

classmethod export(*rules*)
Rules are tuples (rule, score)

classmethod get_instance(*s*, ***kwargs*)
s can be template name or rule.

classmethod load()

register = {'FORCELEMME': <class 'sulci.rules_templates.FORCELEMME'>, 'MAKELOWER': <class 'sulci.rules

class sulci.rules_templates.**LexicalBaseTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.RuleTemplate

Base class for the lexical rules.

compile_rule(*from_tag*, *to_tag*, *complement*)

test_complement(*token*, *complement*)

classmethod uncompile_rule(*rule*)

class sulci.rules_templates.**LexicalTemplateGenerator**
Bases: type

classmethod export(*rules*)
Rules are tuples (rule, score)

classmethod get_instance(*s*, *lexicon*)
s can be template name or rule.

classmethod load()

register = {'faddsuf': <class 'sulci.rules_templates.faddsuf'>, 'haspref': <class 'sulci.rules_templates.haspref'>, 'fdele

class sulci.rules_templates.**LexiconCheckTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexicalBaseTemplate

Base templates for those who have to check lexicon.

make_rules(*token*)

test_complement(*token*, *complement*)
For the Lexicon Check rules, we need to check if modified word is in lexicon.

class sulci.rules_templates.**MAKELOWER**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LemmatizerBaseTemplate

Make the original lower, if the tag is x.

apply_rule(*tokens*, *rule*)

compile_rule(*tag*)

make_rules(*token*)

test_rule(*token*, *rule*)

class sulci.rules_templates.**NEXT1OR2OR3TAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.OrTemplate, sulci.rules_templates.TagBasedTemplate

One of the next three words is tagged X.

get_target()

class sulci.rules_templates.**NEXT1OR2TAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.OrTemplate, sulci.rules_templates.TagBasedTemplate

One of the next three token is tagged X.

get_target()

class sulci.rules_templates.**NEXT1OR2WD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.OrTemplate, sulci.rules_templates.WordBasedTemplate

One of the next two token is word X.

get_target()

class sulci.rules_templates.**NEXT2TAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagBasedTemplate

The token after next token is tagged X.

get_target()

class sulci.rules_templates.**NEXT2WD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**NEXTBIGRAM**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

The next two words are X and Y.

get_target()

class sulci.rules_templates.**NEXTTAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagBasedTemplate

The next token is tagged X.

get_target()

class sulci.rules_templates.**NEXTWD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**NoLexiconCheckTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexicalBaseTemplate

make_rules(*token*)

class sulci.rules_templates.**OrTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.ContextualBaseTemplate

Abstract class for template where we check not specific position.

make_rules(*token*)

test_complement(*token*, *complement*)

class sulci.rules_templates.**PREV1OR2OR3TAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.OrTemplate, sulci.rules_templates.TagBasedTemplate

One of the next three token is tagged X.

get_target()

class sulci.rules_templates.**PREV1OR2TAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.OrTemplate, sulci.rules_templates.TagBasedTemplate

One of the next three token is tagged X.

get_target()

class sulci.rules_templates.**PREV1OR2WD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.OrTemplate, sulci.rules_templates.WordBasedTemplate

One of the next two token is word X.

get_target()

class sulci.rules_templates.**PREV2TAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagBasedTemplate

The token after next token is tagged X.

get_target()

class sulci.rules_templates.**PREV2WD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**PREVBIGRAM**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

The previous two words are X and Y.

get_target()

class sulci.rules_templates.**PREVTAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagBasedTemplate

The next token is tagged X.

get_target()

class sulci.rules_templates.**PREVWD**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**ProximityCheckTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexicalBaseTemplate

compile_rule(*from_tag*, *to_tag*, *complement*)
No len...

class sulci.rules_templates.**RBIGRAM**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**RuleTemplate**(*pk*, ***kwargs*)
Bases: sulci.base.RetrievableObject

Class for managing rules creation and analysis.

apply_rule(*tokens*, *rule*)
Apply rule to candidates in a set of tokens.

get_to_tag(*rule*)

is_candidate(*token*, *rule*)

make_rules(*token*)

classmethod select_one(*rules*, *MAX*, *minval=2*)
Select one rule between a set of tested rules.

rules is a iterable of tuples : (rule, good, bad), where good is the number of errors corrected, and bad the number of error generated.

test_complement(*token*, *complement*)

test_rule(*token*, *rule*)

class sulci.rules_templates.**SURROUNDTAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagBasedTemplate

The preceding word is tagged x and the following word is tagged y.

get_target()

class sulci.rules_templates.**TagBasedTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.ContextualBaseTemplate

Abastract Class for tags based template.

get_complement(*token*)

class sulci.rules_templates.**TagWordBasedTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.ContextualBaseTemplate

Abastract Class for mixed based template : tag, than word.

get_complement(*token*)

class sulci.rules_templates.**WDAND2AFT**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**WDAND2BFR**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordBasedTemplate

One of the next three token is word X.

get_target()

class sulci.rules_templates.**WDAND2TAGAFT**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordTagBasedTemplate

Current word, and tag of two token after.

get_target()

class sulci.rules_templates.**WDAND2TAGBFR**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagWordBasedTemplate

Current word, and tag of two token before.

get_target()

class sulci.rules_templates.**WDNEXTTAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.WordTagBasedTemplate

Current word, and tag of token after.

get_target()

class sulci.rules_templates.**WDPREVTAG**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.TagWordBasedTemplate

Current word, and tag of token before.

get_target()

class sulci.rules_templates.**WordBasedTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.ContextualBaseTemplate

Abstract Class for words based template.

get_complement(*token*)

class sulci.rules_templates.**WordTagBasedTemplate**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.ContextualBaseTemplate

Abastract Class for mixed based template : word, than tag.

get_complement(*token*)

class sulci.rules_templates.**addpref**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexiconCheckTemplate

Change current tag to tag X, if adding prefix Y lead in a entry of the lexicon.

Prefix Y lenght from 1 to 4 (Y < 4) : Syntax : Y addpref len(Y) X Ex. : er addpref 2 VNCFF

get_complement(*token*)

modified_token(*token*, *complement*)

class sulci.rules_templates.**addsuf**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexiconCheckTemplate

Change current tag to tag X, if adding suffix Y lead in a entry of the lexicon.

Suffix Y lenght from 1 to 4 (Y < 4) : Syntax : Y addsuf len(Y) X Ex. : re addsuf 2 VNCFF

get_complement(*token*)

modified_token(*token*, *complement*)

class sulci.rules_templates.**deletepref**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexiconCheckTemplate

Change current tag to tag X, if removing prefix Y lead in a entry of the lexicon.

Prefix Y lenght from 1 to 4 (Y < 4) : Syntax : Y deletepref len(Y) X Ex. : re deletepref 2 VNCFF

get_complement(*token*)

test_complement(*token*, *complement*)
Tests if token has the right prefix, and if deleting it result in a word in the lexicon

class sulci.rules_templates.**deletesuf**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.LexiconCheckTemplate

Change current tag to tag X, if removing suffix Y lead in a entry of the lexicon.

get_complement(*token*)
Return a tuple of afix, ceased_token.

test_complement(*token*, *complement*)
Test if token has the right suffix, and if deleting it result in a word in the lexicon

class sulci.rules_templates.**faddpref**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.addpref

Change current tag to tag X, if adding prefix Y lead in a entry of the lexicon and if current tag is Z.

Prefix Y lenght from 1 to 4 (Y < 4) : Syntax : Z Y faddpref len(Y) X Ex. : SBC:sg re faddpref 2 VNCFF

class sulci.rules_templates.**faddsuf**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.addsuf

Change current tag to tag X, if removing prefix Y lead in a entry of the lexicon and current tag is Z.

Suffix Y lenght from 1 to 4 (Y <= 4) : Syntax : Z Y faddsuf len(Y) X Ex. : SBC:sg re faddsuf 2 VNCFF

class sulci.rules_templates.**fdeletepref**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.deletepref

Change current tag to tag X, if removing prefix Y lead in a entry of the lexicon and if current tag is Z.

Prefix Y lenght from 1 to 4 (Y < 4) : Syntax : Z Y fdeletepref len(Y) X Ex. : ADV re fdeletepref 2 VNCFF

class sulci.rules_templates.**fdeletesuf**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.deletesuf

Change current tag to tag X, if removing suffix Y lead in a entry of lexicon and if current tag is Z.

class sulci.rules_templates.**fgoodleft**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.goodleft

class sulci.rules_templates.**fgoodright**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.goodright

class sulci.rules_templates.**fhaspref**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.haspref

Change current tag to tag X, if prefix is Y and current tag is Z.

Prefix Y is length from 1 to 4 (y <= 4) Syntax: Z Y hassuf len(Y) X Ex. : ADV bla haspref 3 DTC:sg

class sulci.rules_templates.**fhassuf**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.hassuf

Change current tag to tag X, if suffix is Y and current tag is Z.

Suffix Y is length from 1 to 4 (y <= 4) Syntax: Z Y hassuf len(Y) X Ex. : SBC:sg ment hassuf 4 ADV

class sulci.rules_templates.**goodleft**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.NoLexiconCheckTemplate, sulci.rules_templates.ProximityCheckT

The current word is at the right of the word x.

get_complement(*token*)

class sulci.rules_templates.**goodright**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.NoLexiconCheckTemplate, sulci.rules_templates.ProximityCheckT

The current word is at the right of the word X.

get_complement(*token*)

class sulci.rules_templates.**haspref**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.NoLexiconCheckTemplate

Change current tag to tag X, if prefix is Y.

Prefix Y is length from 1 to 4 (y <= 4) Syntax: Z Y haspref len(Y) X Ex. : pro haspref 3 SBC:sg

get_complement(*token*)

class sulci.rules_templates.**hassuf**(*pk*, ***kwargs*)
Bases: sulci.rules_templates.NoLexiconCheckTemplate

Change current tag to tag X, if suffix is Y.

Suffix Y is length from 1 to 4 (y <= 4) Syntax: Y hassuf len(Y) X Ex. : ment hassuf 4 ADV

get_complement(*token*)
Return a tuple of afix, ceased_token.

1.6.7 lexicon Module

Define the Lexicon class.

For now, the lexicon is stored in a flat file, with special syntax :

- word[TAB]POStag1/lemme1[TAB]POStag2/lemme2

class sulci.lexicon.**Lexicon**(*path='corpus'*)
Bases: sulci.base.TextManager

The lexicon is a list of unique words and theirs possible POS tags.

add_factors(*token*)
Build the list of factors (pieces of word).

These factors are used by the POStagger, to determine if an unnown word could be a derivate of another.

check()
Util method to try to individuate errors in the Lexicon. For this, we display the entries with several tags, in case they are wrong duplicate.

create_afixes()
We determinate here the most frequent prefixes and suffixes.

get_entry(*entry*)

items()

loaded
Load lexicon in RAM, from file.

The representation will be a dict {"word1": [{tag1 : lemme1}]}

make(*force=False*)
Build the lexicon.

prefixes

suffixes

class sulci.lexicon.**LexiconEntity**(*raw_data*, ***kwargs*)
Bases: object

One word of a lexicon.

1.6.8 lemmatizer Module

class sulci.lemmatizer.**Lemmatizer**(*lexicon*)
Bases: sulci.base.TextManager

This class give a lemma for a token, using his tag.

PATH = 'corpus'

VALID_EXT = '.lem.crp'

content

do(*token*)
A Token object or a list of token objects is expected. Return the token or the list.

samples

tokens

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S