pypolibox Documentation

Release 1.0.2

Arne Neumann

Jan 20, 2018

Contents

Contents:

CHAPTER 1

pypolibox

pypolibox is a database-to-text generation (NLG) software built on Python 2.7, *NLTK* and Nicholas FitzGerald's *pydocplanner*.

Using a database of technical books and some user input, pypolibox generates sentences descriptions. These descriptions are then used by the *OpenCCG* surface realiser to generate written sentences in German.

1.1 Installation

1.1.1 Prerequisites

In order to generate sentences (instead of abstract sentence descriptions), you will need to install OpenCCG (tested with version 0.9.5). Make sure that you can call `tccg` from the command line, e.g. by adding the `openccg/bin` directory to your `$PATH`.

Under Linux, you'd have to add something like this to your `.bashrc`:

```
export PATH=/home/username/bin/openccg/bin:$PATH

export OPENCCG_HOME=/home/username/bin/openccg
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64
```

Under Windows, you'll have to set the environment variables `OPENCCG_HOME`, `JAVA_HOME` and add the full path of your `openccg/bin` directory to the `PATH` variable.

pywin32 also needs to be installed under Windows.

1.1.2 Install from PyPI

```
pip install pypolibox
```

Under Linux, you might have to prepend that command with `sudo` or execute it as root. Under Windows, you'll need to run this command in a console with administrator rights.

1.1.3 Install from source

You might also need superuser/admin rights for this (see above).

```
git clone https://github.com/arne-cl/pypolibox.git
cd pypolibox
python setup.py install
```

1.2 Usage

1.2.1 Command line usage

`pypolibox` can be used from the command line or from within a Python interpreter. To see all the available options, enter:

```
pypolibox -h
```

To find books that are written in German and use the programming language Prolog, type:

```
pypolibox --language German --proglang Prolog
```

or, if you prefer short but cryptic commands:

```
pypolibox -l German -p Prolog
```

You can choose between several output formats using the `-o` or `--output-format` argument.

- `openccg` generates sentences using OpenCCG (default option)
- `textplan-xml` generates an XML representation of the textplans
- `textplan-featstruct` generates a feature structure representation (`nltk.featstruct`)
- `hlds` generates an HLDS XML representations of all the sentences.

In future versions, you will be able to choose between several output natural languages the `-d` or `--output-language` argument (currently only German is supported).

The following example query will generate HLDS XML snippets describing books about Prolog written in German:

```
pypolibox --language German --proglang Prolog --output-format hlds
```

Further usage examples can be found in the `pypolibox.database.Query` class documentation.

1.2.2 Library usage

If you'd like to access `pypolibox` from within a Python interpreter, you can simply use the same arguments. Instead of a string like *-l German -p Prolog*, you will have to provide your arguments as a list of strings:

```
Query(["-l", "German", "-p", "Prolog"])
```

This query would be equivalent to the command line queries above. `pypolibox` is built as a pipeline, where each important step is represented by a class. Each of these classes function as the input of the next class in the pipeline, e.g.:

```
query = Query(["-l", "German", "-p", "Prolog"])
Results(query)
Books(Results(query))
...
TextPlans(AllMessages(AllPropositions(AllFacts(Books(Results(query))))))
```

If you instanciate a Query with your query arguments, you can use this `Query` instance as the input of a `Results` instance (which contains the data that the database provided for your query), which in turn can be used as the input of a `Books` instance etc.

Of course, you wouldn't want to chain all those classes just to retrieve textplans. To do so, simply use one of the functions provided in the `debug` module, either by running the `debug.py` file in the interpreter or by importing it:

```
import debug
debug.gen_textplans(["-l", "German", "-p", "Prolog"])
```

This function call would return the same results as the aforementioned command line calls. For further testing, try `debug.testqueries` and `debug.error_testqueries`, which basically are lists of predefined valid and invalid query arguments and which can be used to query the database (and see how errors are handled).

1.3 Documentation

The documentation is available online, but you can always get an up-to-date local copy using Sphinx.

You can generate an HTML or PDF version by running these commands in pypolibox's `docs` directory:

```
make latexpdf
```

to produce a PDF (`docs/_build/latex/pypolibox.pdf`) and

```
make html
```

to produce a set of HTML files (`docs/_build/html/index.html`).

1.4 Package Overview

The pypolibox package contains the following modules:

- The `pypolibox` module is the main module, which is invoked from the command line.
- The `database` module handles the user input, queries the database and returns the results.
- `facts` converts those results into attribute value matrices.
- The `propositions` module evaluates those facts (positive, negative, neutral).
- The `textplan` module takes those propositions and turns them into messages. In contrast to propositions, messages do not contain duplicates and add comparative information. Rules will be used to combine those message into constituent sets and ultimately into one text plan. The `textplan` module also allows exporting those text plans in XML format.

- The `rules` module contains the rules used by be the `textplan` module to combine messages into constituent sets and textplans, respectively.
- The `messages` module generates messages from propositions, which will be used by the `textplan` module.
- The `lexicalize_messageblocks` is the "main" module of the lexicalization. For each message block in a textplan, it generates one or more possible lexicalizations which are then realized by the `realization` module.
- The `lexicalization` module generates lexicalizations (in HLDS-XML format) for each message, which are used by the `lexicalize_messageblocks` module to form lexicalizations of complete message blocks.
- **A note on terminology**: A message block in `pypolibox` is basically an instance of the `Message` class, e.g an "id" message block. This "id" message block in turn consists of several messages, e.g. an "authors" message and a "title" message.
- The `realization` module takes a lexicalized phrase or sentence (in HLDS-XML format) and converts it into a surface realization (with the help of OpenCCGs `tccg` executable).
- The `hlds` module allows to convert textplans from a `nltk.featstruct`-based format to HLDS-XML and vice versa. In addition, the module can produce attribute-value matrices of these textplans as LaTeX/PDF files.

1.5 Licence

1.6 Contributors

Arne Neumann (original author), Pablo Duboue

1.7 Acknowledgements

This software reimplements parts of the Java-based *JPolibox* text-generation software written by Alexandra Strelakova, Felix Dombek, Mathias Langer and Till Kolter. pypolibox also includes a heavily modified version of Nicholas FitzGerald's *pydocplanner*, which he released under a Creative Commons license (not specified further). The German OpenCCG grammar fragment that comes with pypolibox was written by Martin Oltmann.

CHAPTER 2

pypolibox

2.1 pypolibox Package

2.1.1 pypolibox Package

2.1.2 database Module

The database module is responsible for parsing the user's requirements (both from command line options, as well as interactively from the Python interpreter), transforming these requirements into an SQL query, querying the sqlite database and returning the results.

class pypolibox.database.**Book**(*db_item*, *db_columns*, *query_args*)
a Book instance represents one book from a database query

get_number_of_book_matches()
calculates the number of query parameters that a book matches

Return type int

class pypolibox.database.**Books**(*results*)
a Books instance stores all books that were found by a database query as a list of Book instances in self.books

get_book_ranks(*possible_matches*)
ranks 'OR query' results according to the number of query parameters they match.

Parameters **possible_matches** (*int*) – the number of (meaningful) parameters of the query.

Returns **book_ranks** – a list of tuples, where each tuple consists of the score of a book and its index in self.books

Return type list of (float, int) tuples

class pypolibox.database.**Query**(*argv*)
a Query instance represents one user query to the database

Queries can be made from the command line, as well as from the Python interpreter. From the command line, queries can be made using either abbreviated or long parameters. The following examples both query the database for books that contain code examples and deal with both semantics and parsing:

```
python pypolibox.py -k semantics, parsing -c 1
python pypolibox.py --keywords semantics, parsing --codeexamples 1
```

When calling `pypolibox.py` from within the Python interpreter, the same query can be made using the following command:

```
Query(["-k", "semantics", "parsing", "-c", "1"])
```

If you print the `Query` instance (by using the `print` command), it will return the SQL query that was constructed from the user input:

```
SELECT * FROM books WHERE keywords like '%semantics%' AND keywords
like '%parsing%' AND examples = 1
```

TODO: This module talks directly to the database. To make it easier to adapt pypolibox to a different domain, an SQL abstraction layer (e.g. SQL Alchemy) should be used.

class pypolibox.database.**Results**(*query*)

A `Results` instance sends queries to the database, retrieves and stores the results.

get_number_of_possible_matches()

Counts the number of query paramters that `could` be matched by books from the results set. The actual scoring of books takes place in `Books.get_book_ranks()`.

For example, if a query contains the parameters:

```
keywords = pragmatics, keywords = semantics, language = German
```

it means that a book could possible match 3 parameters (possible_matches = 3).

Returns the number of possible matches

Return type `int`

get_table_header(*table_name*)

get the column names (e.g. title, year, authors) and their index from the books table of the db and return them as a dictionary.

Parameters **table_name** (`str`) – name of a database table, e.g. 'books'

Returns a dictionary, which contains the names of the table columns

as keys and their index as values :rtype: `dict`, with `str` keys and `int` values

pypolibox.database.**get_column**(*column_name*)

debugging: primitive db query that returns all the values stored in a column, e.g. get_column("title") will return all book titles stored in the database

Return type `list` of `str`

2.1.3 debug Module

The `debug` module contains a number of functions, which can be used to test the behaviour of pypolibox' classes, test its error handling or simply provides short cuts to generate frequently needed data.

pypolibox.debug.**abbreviate_textplan**(*textplan*)
recursive helper function that prints only the skeletton of a textplan (message types and RST relations but not the actual message content)

Parameters **textplan** (TextPlan or ConstituentSet or Message) – a text plan, a constituent set or a message

Returns a message (without the attribute value pairs stored in it)

Return type Message

pypolibox.debug.**apply_rule**(*messages*, *rule_name*)
debugging: take a rule and apply it to your list of messages.

the resulting ConstituentSet will be added to the list, while the messages involved in its construction will be removed. repeat this step until you've found an erroneous/missing rule.

pypolibox.debug.**compare_hlds_variants**()
TODO: kill bugs

BUG1: sentence001-original-test contains 2(!) <item> sentences.

pypolibox.debug.**compare_textplans**()
helps to find out how many different text plan structures there are.

pypolibox.debug.**enumprint**(*obj*)
prints every item of an iterable on its own line, preceded by its index

pypolibox.debug.**find_applicable_rules**(*messages*)
debugging: find out which rules are directly (i.e. without forming ConstituentSets first) applicable to your messages

pypolibox.debug.**findrule**(*ruletype=''*, *attribute=''*, *value=''*)
debugging: find rules that have a certain ruleType and some attribute-value pair

Example: findrule("Concession", "nucleus", "usermodel_match") finds rules of type 'Concession' where rule.nucleus == 'usermodel_match'.

pypolibox.debug.**gen_all_messages_of_type**(*msg_type*)
generate all messages for all books from all testqueries, but return only those which match the given message type, e.g. 'id' or 'extra'.

pypolibox.debug.**gen_all_textplans**()
generates all text plans for each query in the predefined list of test queries.

Return type list of ''TextPlan''s or ''str''s

Returns

pypolibox.debug.**gen_textplans**(*query*)
debug function: generates all text plans for a query.

Parameters **query** (int or list of str) – can be the index of a test query (e.g. 4) OR a list of query parameters (e.g. ["-k", "phonology", "-l", "German"])

Return type TextPlans

Returns a TextPlans instance, containing a number of text plans

pypolibox.debug.**genallmessages**(*query*)
debug function: generates all messages plans for a query.

Parameters **query** (int or list of str) – can be the index of a test query (e.g. 4) OR a list of query parameters (e.g. ["-k", "phonology", "-l", "German"])

Return type AllMessages

Returns all messages that could be generated for the query

pypolibox.debug.**genmessages**(*booknumber=0*, *querynumber=10*)
generates all messages for a book regarding a specific database query.

Parameters **booknumber** (int) – the index of the book from the results list ("0" would be the first book with the highest score)

Parameters **querynumber** (int) – the index of a query from the predefined list of test queries (named 'testqueries')

Return type list of ''Message''s

pypolibox.debug.**genprops**(*querynumber=10*)
generates all propositions for all books in the database concerning a specific query.

Parameters **querynumber** (int) – the index of a query from the predefined list of test queries (named 'testqueries')

Return type AllPropositions

pypolibox.debug.**msgtypes**(*messages*)
print message types / rst relation types, no matter which data structure is used to represent them

pypolibox.debug.**printeach**(*obj*)
prints every item of an iterable on its own line

pypolibox.debug.**test_cli**(*query_arguments=[[], ['-k', 'pragmatics'], ['-k', 'pragmatics', '-r', '4'], ['-k', 'pragmatics', 'semantics'], ['-k', 'pragmatics', 'semantics', '-r', '7'], ['-l', 'German'], ['-l', 'German', '-p', 'Lisp'], ['-l', 'German', '-p', 'Lisp', '-k', 'parsing'], ['-l', 'English', '-s', '0', '-c', '1'], ['-l', 'English', '-s', '0', '-e', '1', '-k', 'discourse'], ['-k', 'syntax', 'parsing', '-l', 'German', '-p', 'Prolog', 'Lisp', '-s', '2', '-t', '0', '-e', '1', '-c', '1', '-r', '7']]*)
run several complex queries and print their results to stdout

2.1.4 facts Module

The facts module takes the information stored in Book instances and converts them into attribute value matrices (Facts). Furthermore, the module compares each book with its predecessor (e.g. book A is newer than book B and has code examples, while B is shorter and targets beginners ...). The insights gathered from these comparisons are also stored in Facts instances.

class pypolibox.facts.**AllFacts**(*b*)
Simply speaking, an AllFacts instance contains all facts about all books that were returned by a database query. More formally, it contains a Facts instance for each Book in a Books instance.

In a Books instance, all books returned by a database query are sorted by the number of query parameters they match ('user model match') in descending order. This means, that AllFacts will contain facts about the best-matching book, followed by facts about the second-best matching book (including a comparison to the best matching one), followed by facts about the third-best matching book (including a comparison to the second one) etc.

class pypolibox.facts.**Facts**(*book*, *book_score*, *index=0*, *preceding_book=False*)

A Facts instance represents facts about a single book, but also contains a comparison of that particular book with its predecessor.

generate_extra_facts(*index*, *book*)

generates extra_facts, if the current book is very new/old or very short/long.

Parameters

- **index** (int) – the index of the book in the Books list of books
- **book** (Book) – a Book instance

Returns a dictionary that contains information about the recency and

length of a book :rtype: dict

generate_id_facts(*index*, *book*)

generates a dictionary of id facts about the current book which will be stored in self.facts["id_facts"]. In contrast to other facts, id_facts are those kind of facts that can be directly retrieved from the database (i.e. there is no comparison between books or reasoning involved). The id_facts dictionary contains the following keys:

```
id_facts keys       database book table columns

'authors'
'codeexamples'      'examples'
'exercises'
'keywords'
'language'          'lang'
'pages'
'proglang'          'plang'
'target'
'title'
'year'
```

The key names should be self-exlanatory. In those cases where they do not exactly match their counterparts in the database, the corresponding database table column name is given in the table above.

Parameters

- **index** (int) – the index of the book in the Books list of books
- **book** (Book) – a Book instance

Returns a dictionary with the keys described above

Return type dict

generate_lastbook_facts(*index*, *book*, *preceding_book*)

generates facts that compare the current book with the preceding one. A typical example of a lastbook_facts dictionary would look like this:

```
lastbook_facts:
    lastbook_nomatch:
        {'language': 'German',
        'keywords_preceding_book_only':
            set(['pragmatics', 'chart parsing']),
        'keywords_current_book_only':
            set([' ', 'grammar', 'language hierarchy', 'corpora',
                'syntax', 'morphology', 'left associative
                grammar']),
```

```
        'codeexamples': 0,
        'proglang': set(['Lisp']),
        'newer': 11,
        'keywords':
            set([' ', 'grammar', 'language hierarchy', 'corpora',
            'syntax', 'left associative grammar', 'morphology',
            'chart parsing', 'pragmatics']),
        'proglang_preceding_book_only':
            set(['Lisp'])}
    lastbook_match:
        {'exercises': 1, 'keywords': set(['semantics',
        'parsing']), 'target': 0, 'pagerange': 1}
```

This method will calculate if is newer/older/shorter/longer than its predecessor (if so, it will store the difference as an integer). For keys that have sets as their values (`keywords` and `proglang`), the resulting dictionary will list which values differed and which were only present in either the preceding or the current book.

Parameters

- **index** (`int`) – the index of the book in the `Books` list of books
- **book** (`Book`) – a `Book` instance
- **preceding_book** – if True, there is a book preceding this one

and both books will be compared :type preceding_book: `bool`

Returns a dictionary with two keys: `lastbook_match` and

`lastbook_nomatch`, which in turn are dictionaries themselves and contain facts that are shared between the two books (lastbook_match) or that differ between the two (lastbook_nomatch).

generate_query_facts (*index*, *book*, *book_score*)

generates facts that describes if a book matches (parts of) the query (a.k.a the user model). a typical query_facts dictionary will look like this:

```
query_facts:
    usermodel_nomatch: {'codeexamples': 0}
    usermodel_match: {'exercises': 1, 'keywords':
                      set(['semantics', 'parsing']), 'language':
                      'German'}
    book_score: 0.8
```

The book described in this examples matches 80 % of the user requirements (it contains exercises and deals with semantics and parsing and is written in German) but does not contain code examples (as was asked for by the user).

Parameters

- **index** (`int`) – the index of the book in the `Books` list of books
- **book** (`Book`) – a `Book` instance
- **book_score** – the score of the book that was calculated in

`Books.get_book_ranks()` :type book_score: `float`

Returns a dictionary that contains three keys, the `book_score`,

the `usermodel_match` as well as the `usermodle_nomatch`. 'usermodel_match' contains all the features that were requested by the user and are present in the book. 'usermodle_nomatch' contains all features that were requested but are missing from the book. :rtype: `dict`

2.1.5 `hlds` Module

HLDS (Hybrid Logic Dependency Semantics) is the format internally used by the OpenCCG realizer. This module shall allow the conversion between HLDS-XML files and NLTK feature structures. In addition, it can also be used as a commandline to convert HLDS-XML files in printable versions of ‘‘nltk.FeatStruct‘‘s. The following command produces a LaTeX file that can be compiled into a PDF:

```
python hlds.py --format latex --outfile output.tex input1.xml input2.xml
```

Alternatively, you can also produce ‘ASCII art’ with this command:

```
python hlds.py --format nltk --outfile output.tex input1.xml input2.xml
```

This way, the phrase ‘das Buch’ can be transformed from this HLDS-XML representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
  <lf>
    <satop nom="b1:artefaktum">
      <prop name="Buch"/>
      <diamond mode="NUM">
        <prop name="sing"/>
      </diamond>
      <diamond mode="ART">
        <nom name="d1:sem-obj"/>
        <prop name="def"/>
      </diamond>
    </satop>
  </lf>
  <target>das Buch</target>
</xml>
```

To this attribute-value matrix (LaTeX):

```
\begin{avm}
    \[ $*$nom$*$  & `b1:artefaktum' \\
       $*$prop$*$ & `Buch' \\
       $*$text$*$ & `das Buch' \\
       NUM        & \[ prop & `sing' \] \\
       ART        & \[ nom  & `d1:sem-obj' \\
                       prop & `def' \] \\
    \]
\end{avm}
```

or this one (plain text):

```
[ *root_nom*        = 'b1:artefaktum'           ]
[ *root_prop*       = 'Buch'                    ]
[ *text*            = 'das Buch'                ]
[                                               ]
[ 00__NUM           = [ *mode* = 'NUM'   ]      ]
[                     [ prop   = 'sing' ]       ]
[                                               ]
[                     [ *mode* = 'ART'        ] ]
[ 01__ART           = [ nom    = 'd1:sem-obj' ] ]
[                     [ prop   = 'def'        ] ]
```

class pypolibox.hlds.**Diamond**(*features=None*, ***morefeatures*)

Bases: nltk.featstruct.FeatDict

A {Diamond} represents an HLDS diamond in form of a (nested) feature structure containing the elements nom? prop? diamond*

```
<diamond mode="AGENS">
    <nom name="s1:addition"/>
    <prop name="sowohl"/>
    <diamond mode="NP1">
        <nom name="h1:nachname"/>
        <prop name="Hausser"/>
    </diamond>
    ...
</diamond>
```

append_subdiamond(*subdiamond*, *mode=None*)

appends a subdiamond structure to an existing diamond structure, while allowing to change the mode of the subdiamond

Parameters **mode** (str or NoneType) – the mode that the subdiamond shall have. this will

also be used to determine the subdiamonds identifier. if the diamond already has two subdiamonds (e.g. "00__AGENS" and "01__PATIENS") and add a third subdiamond with mode "TEMP", its identifier will be "02__TEMP". if mode is None, the subdiamonds mode will be left untouched.

change_mode(*mode*)

changes the mode of a Diamond, which is sometimes needed when embedding it into another Diamond or Sentence.

insert_subdiamond(*index*, *subdiamond_to_insert*, *mode=None*)

insert a Diamond into this one before the index, while allowing to change the mode of the subdiamond.

Parameters **mode** (str or NoneType) – the mode that the subdiamond shall have. this will

also be used to determine the subdiamonds identifier. if the diamond already has two subdiamonds (e.g. "00__AGENS" and "01__PATIENS") and we'll insert a third subdiamond at index '1' with mode "TEMP", its identifier will be "01__TEMP", while the remaining two subdiamond identifiers will will be changed accordingly, e.g. "00__AGENS" and "02__PATIENS". if mode is None, the subdiamonds mode will be left untouched.

prepend_subdiamond(*subdiamond_to_prepend*, *mode=None*)

prepends a subdiamond structure to an existing diamond structure, while allowing to change the mode of the subdiamond

Parameters **mode** (str or NoneType) – the mode that the subdiamond shall have. this will

also be used to determine the subdiamonds identifier. if the diamond already has two subdiamonds (e.g. "00__AGENS" and "01__PATIENS") and we'll prepend a third subdiamond with mode "TEMP", its identifier will be "00__TEMP", while the remaining two subdiamond identifiers will will be incremented by 1, e.g. "01__AGENS" and "02__PATIENS". if mode is None, the subdiamonds mode will be left untouched.

class pypolibox.hlds.**HLDSReader**(*hlds*, *input_format='file'*)

represents a list of sentences (as NLTK feature structures) parsed from an HLDS XML testbed file.

parse_sentence(*sentence*, *single_sent=True*)

parse_sentences(*tree*)
Parses all sentences (represented as HLDS XML structures) into feature structures. These structures are saved as a list of ‘‘Sentence‘‘s in self.sentences.

If there’s only one sentence in a file, it’s root element is <xml>. If there’s more than one, they are each <xml> sentence “roots” is wrapped in an <item>. . . </item> (and <regression> becomes the root tag of the document).

Parameters **tree** (etree._ElementTree) – an etree tree element

class pypolibox.hlds.**Sentence**(*features=None*, ***morefeatures*)
Bases: nltk.featstruct.FeatDict

represents an HLDS sentence as an NLTK feature structure.

create_sentence(*sent_str*, *expected_parses*, *root_nom*, *root_prop*, *diamonds*)
wraps all ‘‘Diamond‘‘s that were already constructed by HLDSReader.parse_sentences() plus some meta data (root verb etc.) into a NLTK feature structure that represents a complete sentence.

Parameters

- **sent_str** (str) – the text that should be generated
- **expected_parses** (int) – the expected number of parses
- **root_prop** (str) – the root element of that text (in case we’re

actually generating a sentence: the main verb)

Parameters

- **root_nom** (str) – the root (element/verb) category, e.g. “b1:handlung”
- **diamonds** (list of ‘‘Diamond‘‘s) – a list of the diamonds that are contained in the

sentence

pypolibox.hlds.**add_mode_suffix**(*diamond*, *mode='N'*)

pypolibox.hlds.**add_nom_prefixes**(*diamond*)
Adds a prefix/index to the name attribute of every <nom> tag of a Diamond or Sentence structure. Without this, ccg-realize will only produce gibberish.

Every <nom> tag has a ‘name’ attribute, which contains a category/type-like description of the corresponding <prop> tag’s name attribute, e.g.:

```
<diamond mode="PRÄP">
    <nom name="v1:zugehörigkeit"/>
    <prop name="von"/>
</diamond>
```

Here ‘zugehörigkeit’ is the name of a category that the preposition ‘von’ belongs to. usually, the nom prefix is the first character of the prop name attribute with an added index. index iteration is done by a depth-first walk through all diamonds contained in the given feature structure. In this example ‘v1:zugehörigkeit’ means, that “von” is the first diamond in the structure that starts with ‘v’ and belongs to the category ‘zugehörigkeit’.

pypolibox.hlds.**convert_diamond_xml2fs**(*etree*)
transforms a HLDS XML <diamond>. . . </diamond> structure (that was parsed into an etree element) into an NLTK feature structure.

Parameters **etree_or_tuple** (etree._Element) – a diamond etree element

Return type Diamond

pypolibox.hlds.**create_diamond**(*mode*, *nom*, *prop*, *nested_diamonds_list*)
creates an HLDS feature structure from scratch (in contrast to convert_diamond_xml2fs, which converts an HLDS XML structure into its corresponding feature structure representation)

NOTE: I'd like to simply put this into __init__, but I don't know how to subclass FeatDict properly. FeatDict.__new__ complains about Diamond.__init__(self, mode, nom, prop, nested_diamonds_list) having too many arguments.

pypolibox.hlds.**create_hlds_file**(*sent_or_sent_list*, *mode='test'*, *output='etree'*)
this function transforms ‘‘Sentence‘‘s into a a valid HLDS XML testbed file

Parameters

- **sent_or_sent_list** (Sentence or list of ‘‘Sentence‘‘s) – a Sentence or a list of ‘‘Sentence‘‘s
- **mode** (str) – “test”, if the sentence will be part of a (regression)

testbed file (ccg-test). “realize”, if the sentence will be put in a file on its own (ccg-realize).

Parameters **output** (str) – “etree” (etree element) or “xml” (formatted, valid xml

document as a string)

Return type str

pypolibox.hlds.**diamond2sentence**(*diamond*)
Converts a Diamond feature structure into a Sentence feature structure. This becomes necessary whenever we want to realize a short utterance, e.g. “die Autoren” or “die Themen Syntax und Pragmatik”.

Note: OpenCCG does not really distinguish between a sentence and smaller units of meaning. It simply assigns the <sentence> tag to every HLDS structure it realizes, whereas each substructure of this “sentence” (no matter how complex) is labelled as a <diamond>.

Return type Sentence

pypolibox.hlds.**etreeprint**(*element*, *debug=True*, *raw=False*)
pretty print function for etree trees or elements

Parameters **debug** – if True: not only return the XML string, but also print it to

stdout. if False: only return the XML string

Parameters **raw** – if True: just transform the etree (element) into a string,

don't add or prettify anything. if False: add an XML declaration and use pretty print to make the output more readable for humans.

pypolibox.hlds.**featstruct2avm**(*featstruct*, *mode='non-recursive'*)
converts an NLTK feature structure into an attribute-value matrix that can be printed with LaTeX's avm environment.

Return type str

pypolibox.hlds.**hlds2xml**(*featstruct*)
debug function that returns the string representation of a feature structure (Diamond or Sentence) and its HLDS XML equivalent.

Return type str

pypolibox.hlds.**last_diamond_index**(*featstruct*)
Returns the highest index currently used withing a given Diamond or Sentence. E.g., if this structure contains three diamonds ("00__ART", "01__NUM" and "02__TEMP"), the return value will be 2. The return value is -1, if the feature structure doesn't contain any ``Diamond``s.

Return type int

pypolibox.hlds.**main**()
parse command line args and do the conversions

pypolibox.hlds.**remove_nom_prefixes**(*diamond*)

pypolibox.hlds.**test_conversion**()
tests HLDS XML <-> NLTK feature structures conversions. converts an HLDS XML testbed file into a list of sentences in NLTK feature structure. picks one of these sentences randomly and converts it back to HLDS XML. prints boths versions of this sentence. returns an HLDSReader instance (containing a list of ``Sentence``s in NLTK feature structure notation) and a HLDS XML testbed file (as a string) created from those feature structures.

Return type tuple of (HLDSReader, str)

Returns a tuple containing an HLDSReader instance and a string

representation of an HLDS XML testbed file

2.1.6 lexicalization Module

2.1.7 lexicalize_messageblocks Module

2.1.8 messages Module

The messages module contains the Message class and related classes.

Message``s contain propositions about books. The text planner applies ``Rule``s to these ``Message``s to form ``ConstituentSet``s. ``Rule``s will also be applied to ``ConstituentSet``s, ultimately forming one ``TextPlan that contains all the information to be realized.

class pypolibox.messages.**AllMessages**(*allpropositions*)
represents all Messages generated from AllPropositions about all Books() that were returned by a query

class pypolibox.messages.**Message**(*msgType=None*)
Bases: nltk.featstruct.FeatDict

A Message combines and stores knowledge about an object (here: books) in a logical structure. Messages are constructed during content selection (taking the user's requirements, querying a database and processing its results), which precedes text planning.

Each Message has a msgType which describes the kind of information it includes. For example, the msgType 'id' specifies information that is needed to distinguish a book from other books:

```
[ *msgType*    = 'id'                                ]
[ authors      = frozenset(['Roland Hausser'])       ]
[ codeexamples = 0                                   ]
[ language     = 'German'                            ]
[ pages        = 572                                 ]
[ proglang     = frozenset([])                       ]
[ target       = 0                                   ]
```

```
[ title        = 'Grundlagen der Computerlinguistik' ]
[ year         = 2000                                ]
```

class pypolibox.messages.**Messages**(*propositions*)
represents all Message instances generated from Propositions about a Book.

add_identification_to_message(*message*)
Adds special 'reference_title' and 'reference_authors' attributes to messages other than the id_message.

In contrast to the id_message, other messages will not be used to produce sentences that contain their content (i.e. no statement of the 'author X wrote book Y in 1979' generated from an 'extra_message' or a 'lastbook_nomatch' message). Nevertheless, they will need to make reference to the title and the authors of the book (e.g. 'Y is a rather short book'). As an example, look at this 'usermodel_match' message:

```
[ *msgType*           = 'usermodel_match'                    ]
[ *reference_authors* = frozenset(['Ulrich Schmitz'])        ]
[ *reference_title*   = 'Computerlinguistik. Eine Einführung' ]
[ language            = 'German'                             ]
[ proglang            = frozenset(['Lisp'])                  ]
```

The message contains two bits of information (the language and programming language used), which both have regular strings as keys. The 'referential' keys on the other hand are nltk.Feature instances and not strings. This distinction should be regarded as a syntactic trick used to emphasize a semantic difference (READ: if you have a better solution, please change it).

generate_extra_message(*proposition_dict*)
generates a Message from an 'extra' Proposition. Extra propositions only exist if a book is remarkably new / old or very short / long.

generate_lastbook_nomatch_message(*proposition_dict*)
generates a Message from a 'lastbook_nomatch' Proposition. A lastbook_nomatch propositions states which differences exist between two books.

generate_message(*proposition_type*)
generates a Message from a 'simple' Proposition. Simple propositions are those kinds of propostions that only give information about one item (i.e. describe one book) but don't compare two items (e.g. book A is 12 years older than book B).

2.1.9 propositions Module

The propositions module evaluates the facts generated by the *pypolibox.facts* module and stores its results as nested dictionaries.

class pypolibox.propositions.**AllPropositions**(*allfacts*)
contains propositions about ALL the books that were listed in a query result

class pypolibox.propositions.**Propositions**(*facts*)
represents propositions (positive/negative/neutral ratings) of a single book. Propositions() are generated from Facts() about a Book().

2.1.10 pypolibox Module

The pypolibox module is the 'main' module of the pypolibox package. It's the module you'd usually call from the command line or load into your Python interpreter. It just imports all the important modules and runs some demo code in case it is run from the command line without any arguments.

pypolibox.pypolibox.**check_and_realize_textplan**(*openccg*, *textplan*, *lexicalize_message_block*, *phrase2sentence*)
realizes a text plan and warns about message blocks that cannot be realized due to current restrictions in the OpenCC grammar.

Parameters

- **openccg** (OpenCCG) – a running OpenCCG instance
- **textplan** (TextPlan) – text plan to be realized

pypolibox.pypolibox.**generate_textplans**(*query*)
generates all text plans for a database query

pypolibox.pypolibox.**initialize_openccg**(*lang='de'*)
starts OpenCCG's tccg realizer as a server in the background (ca. 20s).

pypolibox.pypolibox.**main**()
This is the pypolibox commandline interface. It allows you to query the database and generate book recommendatins, which will either be handed to OpenCCG for generating sentences or printed to stdout in an XML format representing the text plans.

pypolibox.pypolibox.**test**()
test and realize all text plans for all test queries

2.1.11 realization Module

The realization module shall take HLDS XML structures, realize them with the OpenCCG surface realizer and parse its output string.

class pypolibox.realization.**OpenCCG**(*grammar_dir='/home/docs/checkouts/readthedocs.org/user_builds/pypolibox/envpackages/pypolibox-1.0.2-py2.7.egg/pypolibox/grammar'*, *lang='de'*)
Bases: object

command-line interaction with OpenCCG's tccg parser/generator, which can either be run as a JSON-RPC server or simply imported as a Python module.

parse(*text*, *verbose=True*, *raw_output=True*)
This is the core interaction with the parser.

It returns a Python data-structure, while the parse() function returns a JSON object

Returns if raw_output=True, the raw response string from the server

will be returned. otherwise, a list of dictionaries will be returned (one for each input sentence). :rtype: str OR list of ‘‘dict‘‘s

realize(*featstruct*, *raw_output=True*)
converts a Diamond or Sentence feature structure into HLDS-XML, write it to a temporary file, realizes this file with tccg and parses the output it returns.

realize_hlds(*hlds_xml_filename*)

terminate()

pypolibox.realization.**parse_tccg_generator_output**(*tccg_output*)
parses the output string returned from tccg's interactive generator shell.

2.1.12 rules Module

The rules module contains rules, which are used by the text planner to combine messages into constituent sets and ultimately form one TextPlan.

class pypolibox.rules.**ConstituentSet**(*relType=None*, *nucleus=None*, *satellite=None*)
Bases: nltk.featstruct.FeatDict

ConstituentSet is the contstuction built up by applying Rules to a set of ConstituentSet``s and ``Message``s. Each ``ConstituentSet is of a specific relType, and has two constituents, one which is designated the nucleus and one which is designated aux. These ‘‘ConstituentSet‘‘s can then be combined with other ‘‘ConstituentSet‘‘s or ‘‘Message‘‘s.

ConstituentSet is based on nltk.featstruct.FeatDict.

class pypolibox.rules.**Rule**(*name*, *ruleType*, *nucleus*, *satellite*, *conditions*, *heuristic*)
Bases: object

Rules are the elements which specify relationships which hold between elements of the document. These elements can be ‘‘Message‘‘s or ‘‘ConstituentSet‘‘s.

Each Rule specifies a list of inputs, which are is a minimal specification of a Message or ConstituentSet. To be a valid input to this Rule, a given Message or ConstituentSet must subsume one of the specified ‘‘input‘‘s.

Each Rule can also specify a set of conditions which must be met in order for the Rule to hold between the inputs.

Each Rule specifies a heuristic, which will be evaluated to provide a score by which to rank the order in which rules should be applied.

Each Rule specifies which of the inputs will be the nucleus and which will be the aux of the output ConstituentSet.

find_message_candidates(*messages*, *message_prototype*)
takes a list of messages and returns only those with the right message type (as specified in Rule.inputs)

Parameters **messages** (list of ‘‘Message‘‘s) – a list of Message objects, each containing one

message about a book

Parameters **message_prototype** – a tuple consisting of a message name and a

Message or ConstituentSet :type message_prototype: tuple of (string, Message or ConstituentSet)

Return type list of tuple``s of (string, ``Message)

Returns a list containing all (name, message) tuples which are

subsumed by the input message type (self.nucleus or self.satellite). If a rule should only be applied to UserModelMatch and UserModelNoMatch messages, the return value contains a list of messages with these types.

get_conditions(*group*)
applies __name_eval to all conditions a Rule has, i.e. checks if a group meets all conditions

ConstituentSet) :param group: a list of message tuples of the form (message name, message)

Return type list of bool

Returns a list of truth values, each of which tells if a group met

all conditions specified in self.conditions

get_options(*messages*)
this is the main method used for document planning

From the list of `Messages`, `get_options` selects all possible ways the Rule could be applied.

The planner can then select with the `textplan.__bottom_up_search` function one of these possible applications of the Rule to use.

`non_empty_message_combinations` is a list of combinations, where each combination is a (nucleus, satellite)-tuple. both the nucleus and the satellite each consist of a (name, message) tuple.

The method returns an empty list if `get_options` can't find a way to apply the `Rule`.

Parameters **messages** (list of `Message` objects) – a list of `Message` objects, each containing one

message about a book

Return type empty list or a list containing one `tuple` of (`int`,

`ConstituentSet`, `list`), where `list` consists of `Message` or `ConstituentSet` objects :return: a list containing one 3-tuple (score, `ConstituentSet`, inputs) where:

- score is the evaluated heuristic score for this application of

the Rule - ConstituentSet is the new `ConstituentSet` instance returned by the application of the Rule - inputs is the list of inputs (`Message``s or ``ConstituentSets` used in this application of the rule

get_satisfactory_groups(*groups*)

`Message` or `ConstituentSet`) :param groups: a list of group elements. each group contains a list which contains one or more message tuples of the form (message name, message)

Return type `list` of `list`'s of `tuple`'s of (`str`, `Message`

or `ConstituentSet`) :return: a list of group elements. contains only those groups which meet all the conditions specified in self.conditions

class `pypolibox.rules.`**Rules**
creates Rule() instances

Each rule of the form Rule(ruleType, inputs, conditions, nucleus, aux, heuristic) is generated by its own method. Important note: these methods have to adhere to a naming convention, i.e. begin with '**genrule_**'; otherwise, self.__init__ will fail!

genrule_book_differences()
Contrast({id, id_extra_sequence}, lastbook_nomatch)

Meaning: id/id_extra_sequence. In contrast to book X, this book is in German, targets advanced users and ... Condition: There are differences between the two books

genrule_book_similarities()
Elaboration(id_usermodelmatch, lastbook_match)

Meaning: 'id_usermodelmatch' mentions that the books matches ALL requirements. In addition, the book shares many features with its predecessor. Condition: There are both differences and commonalities (>=50%) between the two books.

genrule_compare_eval()
Sequence(concession_books, {pos_eval, neg_eval, usermodel_match, usermodel_nomatch})

Meaning: 'concession_books' describes common and diverging features of the books. 'pos_eval/neg_eval/usermodel_match/usermodel_nomatch' explains how many user requirements they meet

genrule_concession_book_differences_usermodelmatch()
Concession(book_differences, usermodel_match)

Meaning: 'book_differences' explains the differences between both books. Nevertheless, this book meets ALL your requirements ... Condition: All user requirements are met.

genrule_concession_books()
Concession(book_differences, lastbook_match)

Meaning: After 'book_differences' explains the differences between both books, their common features are explained.

genrule_contrast_books_posneg_eval()
Sequence(book_differences, {pos_eval, neg_eval})

Meaning: book_differences mentions the differences between the books, pos_eval/neg_eval explains how many user requirements they meet Conditions: matches some of the requirements

genrule_id_extra_sequence()
Sequence(id_complete, extra), if 'extra' exists:

adds an additional "sentence" about extra facts after the id messages

genrule_id_usermodelmatch()
Elaboration({id, id_extra_sequence}, usermodel_match), if there's no usermodel_nomatch

Meaning: This book fulfills ALL your requirments. It was written in ..., contains these features ... and ... etc

genrule_neg_eval()
Concession(usermodel_nomatch, usermodel_match)

Meaning: Although this book fulfills some of your requirements, it doesn't match most of them. Therefore, this book might not be the best choice.

genrule_no_similarities_concession()
Concession({id, id_extra_sequence}, lastbook_nomatch)

Meaning: Book X has these features BUT share none of them with its predecessor. Condition: There is a predecessor to this book, but they don't share ANY features.

genrule_pos_eval()
Concession(usermodel_match, usermodel_nomatch)

Meaning: Book matches many (>= 50%) of the requirements, but not all of them

genrule_single_book_complete()
Sequence({id, id_extra_sequence}, {pos_eval, neg_eval})

Meaning: The nucleus mentions all the (remaining) facts (that aren't mentioned in the evaluation), while the satellite evaluates the book (in terms of usermodel matches)

genrule_single_book_complete_usermodelmatch()
Sequence({id, id_extra_sequence}, usermodel_match)

Meaning: The satellite states that the book matches ALL the user's requirements. The nucleus mentions the remaining facts about the book. Condition: there's no preceding book and there are only usermodel matches.

genrule_single_book_complete_usermodelnomatch()
Sequence({id, id_extra_sequence}, usermodel_nomatch)

Meaning: The satellite states that the book matches NONE of the user's requirements. The nucleus mentions the remaining facts about the book. Condition: there's no preceding book and there are no usermodel matches.

2.1.13 `textplan` Module

The `textplan` module is based on Nicholas FitzGerald's `py_docplanner``[1], in particular on his idea to represent RST trees as attribute value matrices by using the ``nltk.featstruct` data structure.

`textplan` converts `Proposition` instances into `Message``s (using attribute value notation). Via a set of ``Rule``s, these messages are combined into ``ConstituentSet``s. Rules are applied bottom-up, via a recursive best-first search (cf. ``__bottom_up_search`).

Not only messages, but also constituent sets can be combined via rules. If all messages present can be combined into one large `ConstituentSet`, this constituent set is called a `TextPlan`. A `TextPlan` represents a complete text plan in form of an attribute value matrix.

[1] Fitzgerald, Nicholas (2009). Open-Source Implementation of Document Structuring Algorithm for NLTK.

class pypolibox.textplan.**TextPlan**(*book_score=None*, *dtype='TextPlan'*, *text=None*, *children=None*)
Bases: `nltk.featstruct.FeatDict`

`TextPlan` is the output of Document Planning. A TextPlan consists of an optional title and text, and a child `ConstituentSet`.

TODO: append __str__ method: should describe verbally if a TP is describing one book or comparing two books

class pypolibox.textplan.**TextPlans**(*allmessages*, *debug=False*)
Bases: `object`

generates all `TextPlan``s for an ``AllMessages` instance, i.e. one DocumentPlan for each book that is returned as a result of the user's database query

pypolibox.textplan.**generate_textplan**(*messages, rules=[<pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>, <pypolibox.rules.Rule object>], book_score=None, dtype='TextPlan', text=''*)
The main method implementing the Bottom-Up document structuring algorithm from "Building Natural Language Generation Systems" figure 4.17, p. 108.

The method takes a list of `Message``s and a set of ``Rule``s and creates a document plan by repeatedly applying the highest-scoring Rule-application (according to the Rule's heuristic score) until a full tree is created. This is returned as a ``TextPlan` with the tree set as `children`.

If no plan is reached using bottom-up, `None` is returned.

Parameters **messages** – a list of ‘‘Message‘‘s which have been selected during

content selection for inclusion in the TextPlan :type messages: list of Message``s :param rules: a list of ``Rule``s specifying relationships which can hold between the messages :type rules: list of ``Rule``s :param dtype: an optional type for the document :type dtype: string :param text: an optional text string describing the document :type text: string :return: a document plan. if no plan could be created: return None :rtype: ``TextPlan or NoneType

pypolibox.textplan.**linearize_textplan**(*textplan*)
takes a text plan (an RST tree represented as a NLTK.featstruct data structure) and returns an ordered list of ‘‘Message‘‘s for surface generation.

Return type list of ‘‘Message‘‘s

pypolibox.textplan.**test_textplan2xml_conversion**()
test text plan to XML conversion with all the text plans that were generated for all test queries with debug.gen_all_textplans().

pypolibox.textplan.**textplan2xml**(*textplan*)
converts one TextPlan into an XML structure representing it.

Return type etree._ElementTree

pypolibox.textplan.**textplans2xml**(*textplans*)
converts several ‘‘TextPlan‘‘s into an XML structure representing these text plans.

Return type etree._ElementTree

2.1.14 util Module

The util module contains a number of ‘bread and butter’ functions that are needed to run pypolibox, but are not particularly interesting (e.g. format converters, existence checks etc.).

There shouldn’t be any code in this module that require loading other modules from pypolibox!

pypolibox.util.**ensure_unicode**(*string_or_int*)
ensures that a string does use unicode instead of UTF8. converts integer input to a unicode string.

pypolibox.util.**ensure_utf8**(*string_or_int*)
ensures that a string does not use unicode but UTF8. converts integer input to a string.

pypolibox.util.**exists**(*thing*, *namespace*)
checks if a variable/object/instance exists in the given namespace

Return type bool

pypolibox.util.**flatten**(*nested_list*)
flattens a list, where each list element is itself a list

Parameters **nested_list** (*list*) – the nested list

Returns flattened list

pypolibox.util.**freeze_all_messages**(*message_list*)
makes all messages (‘‘FeatDict‘‘s) immutable, which is necessary for turning them into sets

pypolibox.util.**msgs_instance_to_list_of_msgs**(*messages_instance*)
converts a Messages instance into a list of Message instances

pypolibox.util.**sql_array_to_list**(*sql_array*)
converts SQL string "arrays" into a list of strings

Our book database uses '[' and ']' to handle attributes w/ more than one value: e.g. authors = '[Noam Chomsky][Alan Touring]'. This function turns those multi-value strings into a set with separate values.

Parameters **sql_array** (str) – a string from the database that represents one or

more items delimited by '[' and ']', e.g. "[Noam Chomsky]" or "[Noam Chomsky][Alan Touring]"

Return type list of str

Returns a list of strings, where each string represents one item from

the database, e.g. ["Noam Chomsky", "Alan Touring"]

pypolibox.util.**sql_array_to_set**(*sql_array*)
converts SQL string "arrays" into a set of strings

our book database uses '[' and ']' to handle attributes w/ more than one value: e.g. authors = '[Noam Chomsky][Alan Touring]'

this function turns those multi-value strings into a set with separate values

Parameters **sql_array** (str) – a string from the database that represents one or

more items delimited by '[' and ']', e.g. "[Noam Chomsky]" or "[Noam Chomsky][Alan Touring]"

Return type set of str

Returns a set of strings, where each string represents one item from

the database, e.g. ["Noam Chomsky", "Alan Touring"]

pypolibox.util.**write_to_file**(*str_or_obj*, *file_path*)
takes a string and writes it to a file or takes any other object, pickles it and writes it to a file

CHAPTER 3

News

3.1 1.0.2 (2014-05-17)

Release data: 17-May-2014

- added Windows-specific requirements to setup.py (`winpexpect` vs. `pexpect`)
- README now covers installation prerequisites

3.2 1.0.1 (2014-05-13)

Release date: 13-May-2014

- installation via `pip` or `python setup.py install` now adds two programs to your path: `pypolibox` and `hlds-converter`
- added new output formats (`--output-format` parameter): textplan featstructs, HLDS XML
- documentation is now hosted at readthedocs.org
- converted documentation from epydoc to sphinx
- added make file, license file

3.3 1.0.0 (2014-30-04)

Release date: 30-Apr-2014

- pypolibox is now licensed under GPLv3
- OpenCCG grammar fragment (CC-BY-NC-SA 4.0 licensed) now shipped with code
- first release via PyPI

- got rid of configuration file
- fixed some errors in the documentation

CHAPTER 4

To-do list

- **Theory/Structure**: Rewrite rules for the textplanner. RST relations should combine messages, not message blocks. (A message should be something that can be expressed in a single sentence.)
- **Coverage**: Update the lexicalization module once the grammar fragment is "completed".
- **Consistency**: Make keys unique. Instead of three different "recency" keys, there should be a regular one, an "extra_recency" key ('This book is particularly recent/old') and a "relative_recency" key ('This book is 20 years older than the other one').
- **Consistency**: In "extra recency" messages, "values" are called "descriptions".
- **Unicode**: If NLTK becomes available for Python 3, switch to that branch. Otherwise, evaluate if porting nltk.featstruct to Python 3 is feasible (e.g. with the help of python2to3).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

H

I

L

M

O

P

Q

R

S

T

W