Grafeno Documentation

Release 0.1

Antonio F. G. Sevilla <afgs@ucm.es>

May 14, 2018

Contents

Python library for concept graph extraction from text, operation, and linearization. An integrated web service is provided.

This library is still a work in progress, but it has shown to be already useful for a number of applications, for example extractive text summarization.

Contents:

CHAPTER 1

Introduction

Grafeno is a python library for working with semantic graphs.

The normal workflow has three steps:

1. Creating a graph
2. Operating with the created graph
3. Linearizing the graph

Creating a graph

A number of transformers are used.

Operating with a graph

There are many different operations.

Linearizing a graph

A number of linearizers can be used.

Using a pre-built pipeline

Either programatically, or more easily, in YAML.

CHAPTER 2

Examples

Summarization

This pipeline takes a text and produces a short extractive summary.

```
In [1]: from grafeno import pipeline
```

```
In [2]: import yaml
        config = yaml.load('''
        %YAML 1.2
        ---
        # Summarizes a text by extracting the most relevant sentences.
        transformers:
            - pos_extract
            - sim_link
            - extend
            - unique
            - sentences
        transformer_args:
            sempos: { noun: n }
            unique_gram: { hyper: [ True ] }
            extended_sentence_edges: [ HYP ]
        operations:
            - op: cluster
              hubratio: 0.2
        #    - op: markov_cluster
        #      expand_factor: 10
        #      inflate_factor: 2
        #      max_loop: 10
        #      mult_factor: 1
        #    - op: louvain_cluster
        linearizers:
            - cluster_extract
        linearizer_args:
            summary_length: 50
            summary_margin: 10
            normalize_sentence_scores: True
        ''')
```

```
In [3]: document = '''
        Hurricane Gilbert swept toward the Dominican Republic Sunday, and the Civil Defense
```

```
        '''
In [4]: res = pipeline.run({ **config, 'text': document })
        print(res)
```

```
Tropical Storm Gilbert formed in the eastern Caribbean and strengthened into a hurricane Sa
San Juan, on the north coast, had heavy rains and gusts Saturday, but they subsided during
The first, Debby, reached minimal hurricane strength briefly before hitting the Mexican coa
```

Fig. 2.1: Grafeno logo

Interactive visualization of grafeno Graphs

We use the great D3 library: https://d3js.org/

```
In [1]: from grafeno import Graph
        from grafeno.transformers import get_pipeline
        from grafeno.jupyter import visualize
```

```
<IPython.core.display.Javascript object>
```

```
In [2]: import yaml
        semantic_pipeline = yaml.load(open('../../configs/semantic.yaml'))
        T = get_pipeline(['spacy_parse']+semantic_pipeline.get('transformers'))
```

One sentence

```
In [3]: sentence = """
        John writes a short program that works correctly and he comments his code like a go
        """
```

```
In [4]: G1 = Graph(text=sentence, transformer=T)
```

```
In [5]: visualize(G1)
```

```
Out[5]: <IPython.core.display.HTML object>
```

Bigger graph (from the simple.wikipedia page of AI)

```
In [6]: text = """
        An extreme goal of AI research is to create computer programs that can learn, solve
        In practice, however, most applications have picked on problems which computers can
        Searching data bases and doing calculations are things computers do better than peo
        On the other hand, "perceiving its environment" in any real sense is way beyond pre
        """
```

```
In [7]: G2 = Graph(text=text, transformer=T)
```

```
In [8]: visualize(G2)
```

```
Out[8]: <IPython.core.display.HTML object>
```

CHAPTER 3

Pre-built Pipelines

Some pre-built pipelines come with the library's source code. They are stored under the directory `configs`. The script `test.py` can load them with the `-c` flag, and `server.py` automatically finds them and serves them in the web service.

Summarization

This pipeline is used for extracting short summaries out of news documents.

```
%YAML 1.2
---
# Summarizes a text by extracting the most relevant sentences.
transformers:
    - pos_extract
    - sim_link
    - extend
    - unique
    - sentences
transformer_args:
    sempos: { noun: n }
    unique_gram: { hyper: [ True ] }
    extended_sentence_edges: [ HYP ]
operations:
    - op: cluster
      hubratio: 0.2
#    - op: markov_cluster
#      expand_factor: 10
#      inflate_factor: 2
#      max_loop: 10
#      mult_factor: 1
#    - op: louvain_cluster
linearizers:
    - cluster_extract
linearizer_args:
    summary_length: 100
    summary_margin: 10
    normalize_sentence_scores: True
```

Concept maps

This pipeline generates concept maps useful for conceptual blending. Additionally, it linearizes them into a prolog triplet format.

```
%YAML 1.2
---
# Extracts a concept map from a text.
transformers:
    - pos_extract
    - wordnet
    - numerals
    - adjectives
    - negation
    - genitive
    - prepositions
    - attr_class
    - verb_collapse
    - specific_edges
    - unique
    - lenient
transformer_args:
    sempos:
        noun: n
        adjective: j
    attach_adjectives: True
    keep_attached_adj: True
operations:
    - op: filter_edges
      remove:
        - isa
      rename:
        be: is
      frequency:
        max: 15
        min: 0
    - op: spot_domain
linearizers:
    - prolog
```

CHAPTER 4

grafeno package

This is the main package for the grafeno library. If you want to use grafeno from your code, import it like this:

```
import grafeno
```

If you only need specific functionality, you may want to import it like this:

```
from grafeno import Graph as CG, pipeline
```

Subpackages

grafeno.jupyter package

Submodules

grafeno.jupyter.visualize module

grafeno.linearizers package

grafeno.linearizers. **get_pipeline** (*modules*)
Takes a list of linearizers and returns a linearizer which subclasses them all

Submodules

grafeno.linearizers.all_concepts module

class grafeno.linearizers.all_concepts. **Linearizer** (***kwds*)
Bases: *grafeno.linearizers.base.Linearizer*

Methods

get_root_nodes ( )

grafeno.linearizers.analyzer module

class grafeno.linearizers.analyzer. **Linearizer** (*form='form'*, *attrs=[]*, ***kwds*)
Bases: *grafeno.linearizers.base.Linearizer*

Methods

get_root_nodes ( )

process_node (*n*)

grafeno.linearizers.base module

class grafeno.linearizers.base. **Linearizer** (*header=''*, *separator=''*, *footer=''*, *graph=None*)
Bases: object

Methods

apply_boundaries (*words*, *nodes*)

boundary (*left*, *n*, *word*, *right*)

concat (*nodes*)

expand_node (*n*)

expand_node_list (*nodes*)

get_root_nodes ( )

linearize ( )

process_node (*n*)

grafeno.linearizers.cluster_extract module

class grafeno.linearizers.cluster_extract. **Linearizer** (*hub_score=2*, *nonhub_score=1*, ***kwds*)
Bases: *grafeno.linearizers.extract.Linearizer*

Methods

score_sentence (*sentence_nodes*)

grafeno.linearizers.cypher_base module

class grafeno.linearizers.cypher_base. **Linearizer** (*node_gram_blacklist='id'*, *node_gram_whitelist=None*, *edge_gram_blacklist='functor'*, *edge_gram_whitelist=None*, *sempos_map={'v': 'VERB', 'j': 'ADJECTIVE', 'r': 'ADVERB', 'n': 'NOUN'}*, *cypher_extra_params={}*, ***kwds*)

Bases: *grafeno.linearizers.node_edges.Linearizer*

Methods

cypher_format_edge (*head*, *child*, *edge*, *labels*, *gram*)

cypher_format_node (*node*, *labels*, *gram*)

cypher_get_edge_gram (*edge*)

cypher_get_edge_labels (*edge*)

cypher_get_node_gram (*node*)

cypher_get_node_labels (*node*)

cypher_print_edge (*id*, *labels*, *gram*)

cypher_print_node (*id*, *labels*, *gram*)

process_edge (*n*, *m*, *edge*)

process_node (*node*)

grafeno.linearizers.cypher_base. **cypher_gram** (*gram*)

grafeno.linearizers.cypher_base. **cypher_labels** (*labels*)

grafeno.linearizers.cypher_create module

class grafeno.linearizers.cypher_create. **Linearizer** (*footer=''*, ***kwds*)

Bases: *grafeno.linearizers.cypher_base.Linearizer*

Methods

cypher_format_edge (*head*, *child*, *edge*, *labels*, *gram*)

cypher_format_node (*node*, *labels*, *gram*)

grafeno.linearizers.cypher_query module

class grafeno.linearizers.cypher_query. **Linearizer** (***kwds*)

Bases: *grafeno.linearizers.cypher_base.Linearizer*

This linearizer converts the graph into a Cypher query, suitable for running against a Neo4J database.

The created query matches against subgraphs in the database with **at least** the same nodes and relations. If there are more nodes in the graph database, it also matches. If some node or relation in the grafeno graph is not present in the database subgraph, the whole subgraph doesn't match.

If there are any question nodes (`concept = '?'`) in the graph, the query finds the equivalent node in the database, and returns it and its full directed subcomponent. To reconstruct it, see *reconstruct_graphs()* .

If there are no question nodes in the query, the number of matches is returned.

Methods

cypher_format_edge (*head*, *child*, *edge*, *labels*, *gram*)

cypher_format_node (*node*, *labels*, *gram*)

filter_node (*node*)

grafeno.linearizers.cypher_query. **reconstruct_graphs** (*results*)

This function can be used to reconstruct a grafeno concept graph from the results returned by a query from Neo4J created with the *cypher_query linearizer* .

```
from grafeno.linearizers.cypher_query import Linearizer as graph_to_cypher_query,␣
↪reconstruct_graphs
query = query_graph.linearize(linearizer=graph_to_cypher_query)

from neo4j.v1 import GraphDatabase
driver = GraphDatabase.driver(**connection_params)
results = driver.session().run(query)

for graph in reconstruct_graphs(results):
    do_something_with(graph)
```

grafeno.linearizers.example_nlg module

class grafeno.linearizers.example_nlg. **Linearizer** (*header=''*, *separator=''*, *footer=''*, *graph=None*)

Bases: *grafeno.linearizers.base.Linearizer*

Methods

boundary (*left*, *n*, *word*, *right*)

expand_node (*n*)

get_root_nodes ( )

process_node (*n*)

grafeno.linearizers.extract module

class grafeno.linearizers.extract. **Linearizer** (*summary_length=100*, *summary_margin=10*, *normalize_sentence_scores=False*, *graph=None*)

Bases: object

Methods

linearize ( )

score_sentence (*sentence_nodes*)

grafeno.linearizers.node_edges module

class grafeno.linearizers.node_edges. **Linearizer** (*node_header=''*, *node_sep='n'*, *edge_header='n'*, *edge_sep='n'*, *footer=''*, *graph=None*)

Bases: object

This linearizer outputs the nodes first, and then the edges.

Parameters **node_header** : string

A string to print before all other content.

node_sep : string

A string to print between nodes.

edge_header : string

A string to print after the nodes, and before the edges.

edge_sep : string

A string to print between edges.

footer : string

A string to print after all other content.

graph : *Graph*

The graph to linearize.

Attributes

graph	(*Graph*) The graph to linearize.

Methods

filter_edge (*n*, *m*, *edge*)

Override this method to exclude some edges from the output.

Parameters **n** : int

The id of the head of the edge

m : int

The id of the child of the edge

edge : dict

The grammatemes of the edge to filter.

Returns bool

Whether to include this edge in further processing.

filter_node (*node*)
Override this method to exclude some nodes from the output.

Parameters **node** : dict

The grammatemes of the node to filter.

Returns bool

Whether to include this node in further processing.

linearize ( )

process_edge (*n*, *m*, *edge*)
This method generates a string representation of an edge. Override to customize.

Parameters **n** : int

Id of the head node.

m : int

Id of the dependent node.

edge : dict

Grammatemes of the edge between 'n' and 'm'.

Returns string

A string representation of the edge.

process_node (*node*)
This method generates a string representation of a node. Override to customize.

Parameters **node** : dict

The grammatemes of the node to transform.

Returns string

A string representation of the node.

grafeno.linearizers.prolog module

class grafeno.linearizers.prolog. **Linearizer** (***kwds*)
Bases: *grafeno.linearizers.triplets.Linearizer*

Methods

process_node (*n*)

grafeno.linearizers.semtriplets module

class grafeno.linearizers.semtriplets. **Linearizer** (*make_comp_triplets=False*, ***kwds*)
Bases: *grafeno.linearizers.base.Linearizer*

Methods

expand_node (*n*)

get_root_nodes ( )

process_node (*n*)

grafeno.linearizers.simplenlg module

grafeno.linearizers.triplets module

class grafeno.linearizers.triplets. **Linearizer** (***kwds*)
Bases: *grafeno.linearizers.base.Linearizer*

Methods

expand_node (*n*)

get_root_nodes ( )

process_node (*n*)

grafeno.operations package

grafeno.operations. **operate** (*graph*, *operation*, ***args*)

Submodules

grafeno.operations.cluster module

grafeno.operations.cluster. **cluster** (*cgraph*, *hubratio=0.2*)

grafeno.operations.cluster. **operate** (*graph*, ***args*)

grafeno.operations.clustering module

Created on 3 de mar. de 2016

@author: fiutten

class grafeno.operations.clustering. **Clustering** (*G*, *num_percentage_vertexes*)
Bases: object

Methods

assignNonHubToClusters ( )

computeClusters ( )

computeHVSs ( )

createHubs ( )

extractNodesWithOneVertex ( )

find (*node1*, *node2*)

getConnectionWithHVS2 (*id*, *vertexes*)

getInterSimilarity (*hvs1*, *hvs2*)

getIntraSimilarity (*vertexes*)

getMaxConnectionWithHVSs2 (*id*, *intraconnection*)

getMoreSimilarHVS (*id*)

getNodeFromId (*id*)

getSalienceRanking ( )

grafeno.operations.clustering. **cluster** (*cgraph*, *hubratio=0.2*)

class grafeno.operations.clustering. **salience_node** (*id*, *neighbors*)
Bases: object

Methods

getid ( )

getneighbors ( )

grafeno.operations.filter_edges module

grafeno.operations.filter_edges. **filter_edges** (*cgraph*, *remove=[]*, *rename={}*, *frequency=None*)

grafeno.operations.filter_edges. **operate** (*graph*, ***args*)

grafeno.operations.generalize module

grafeno.operations.generalize. **concept_equal** (*a*, *b*)

grafeno.operations.generalize. **functor_equal** (*a*, *b*)

grafeno.operations.generalize. **generalize** (*a*, *b*, *node_generalize=<function concept_equal>*, *edge_generalize=<function functor_equal>*)
Take two concept graphs and return a new one which generalizes them

grafeno.operations.generalize. **wordnet_generalize** (*a*, *b*)

grafeno.operations.graft module

grafeno.operations.graft. **graft** (*stem*, *locus*, *bud*, *root*)
This operation inserts a whole semantic graph (the *bud*) in place of a node in another graph (the *stem*).

It could be used to replace interrogative nodes in a question graph with the answer graph, or to reify exophoric relations.

Note: If the *bud* is not connected, all components will be inserted into *stem*, but only the *locus* and *root* nodes will be merged.

Warning: This operation is destructive. If you want to keep a non-modified version of *stem*, copy it first.

Parameters **stem** : Graph

The concept graph into which the *bud* is going to be inserted.

locus : int

ID of the node in *stem* to be replaced with *bud*.

bud : Graph

The concept graph to insert into *stem*.

root : int

ID of the node in *bud* that is going to replace the locus, taking with it all its sub-graph.

grafeno.operations.hits module

grafeno.operations.hits. **hits** (*graph*, *epsilon=1e-05*, *max_its=100*)

grafeno.operations.louvain_cluster module

grafeno.operations.markov_cluster module

grafeno.operations.markov_cluster. **add_diag** (*A*, *mult_factor*)

grafeno.operations.markov_cluster. **cluster** (*graph*, *expand_factor=2*, *inflate_factor=2*, *max_loop=10*, *mult_factor=1*)

grafeno.operations.markov_cluster. **expand** (*A*, *expand_factor*)

grafeno.operations.markov_cluster. **get_clusters** (*A*)

grafeno.operations.markov_cluster. **inflate** (*A*, *inflate_factor*)

grafeno.operations.markov_cluster. **mcl** (*M*, *expand_factor=2*, *inflate_factor=2*, *max_loop=10*, *mult_factor=1*)

grafeno.operations.markov_cluster. **normalize** (*A*)

grafeno.operations.markov_cluster. **operate** (*graph*, ***args*)

grafeno.operations.markov_cluster. **stop** (*M*, *i*)

grafeno.operations.rename_concepts module

grafeno.operations.rename_concepts. **operate** (*graph*, ***args*)

grafeno.operations.spot_domain module

grafeno.operations.spot_domain. **operate** (*graph*, ***args*)

grafeno.operations.spot_domain. **spot_domain** (*cgraph*)

grafeno.transformers package

Transformers are one of the key objects of the *grafeno* library. They are in charge of converting the dependency parse of a sentence, extracted by an external tool, into a *grafeno* semantic graph.

```
from grafeno import Graph as CG
from grafeno.transformers import get_pipeline

T = get_pipeline(['pos_extract', 'wordnet', 'unique'])
g = CG(transformer=T, transformer_args={}, text="Fish fish fish fish fish fish fish.")
```

This process happens in stages. First, morphological nodes are transformed into semantic ones:

Semantic nodes

Semantic nodes are dictionaries with the following attributes:

- `concept` : if present, the node will be added to the semantic graph. It represents the main idea, or meaning, of the node. If there is no *concept*, no semantic node will be produced corresponding to the morphological one.
- `id` : a temporal identifier for the node while it is being processed, and hasn't thus been added to the graph yet. When the node is finally added to the graph, it will be changed to the proper graph ID.

Other attributes in the dictionary are also added to the semantic graph node, and are referred to as *grammatemes*.

After the nodes have been processed, the dependency relations are transformed into semantic edges:

Semantic edges

Each semantic edge is a dictionary with the following attributes:

- `parent` : the (temporal or otherwise) id of the source node.
- `child` : the (temporal or otherwise) id of the target node.
- `functor` : if present, the edge will be added to the semantic graph. The *functor* represents the semantic relation between the *parent* and *child* nodes.

Other attributes in the dictionary will be also added to the semantic graph edge, and are referred to as *grammatemes*.

Apart from the main operations of node and edge transformations, there are additional stages in the process where previous or further processing can happen. In order to construct this collection of processing stages, a *Transformer* object has to be created. For this, a *base* class is provided, which has methods for the different stages and is in charge of calling them at the right time and with the appropriate arguments.

The way to construct a pipeline is thus to inherit from this *base* class, and extend the appropriate methods. See its documentation for more information on them.

Additionally, the idea behind transformer classes is that each is supposed to perform a specific operation. This way, a transforming pipeline can be constructed by mixing and matching the desired transformers, by way of creating a class which inherits from them. In order to make this operation easier, a convenience function is provided:

grafeno.transformers.get_pipeline, which takes a list of transformers to use, and constructs the appropriate class which inherits from them all in the correct order.

grafeno.transformers. **get_pipeline** (*modules*)

Takes a list of transformers and returns a transformer class which subclasses them all

Submodules

grafeno.transformers.adjectives module

class grafeno.transformers.adjectives. **Transformer** (*attach_adjectives=False*, *attached_adjective_hyper=True*, *keep_attached_adj=False*, ***kwds*)

Bases: *grafeno.transformers.pos_extract.Transformer*, grafeno.transformers.__utils.Transformer

Processes adjectives. Adds an ATTR functor relation to the head noun.

Parameters **attach_adjectives** : bool

Attaches the adjectival concept to the head noun concept. Useful to distinguish nominal nodes when specified by modifiers.

attached_adjective_hyper : bool

If both *attach_adjectives* and *attached_adjective_hyper* are true, an hypernym node is added to the head with the original nominal concept.

keep_attached_adj : bool

If *attach_adjectives* is True and *keep_attached_adj* is False, adjectival nodes are dropped after being attached.

Methods

transform_dep (*dep*, *parent*, *child*)

grafeno.transformers.adverbs module

class grafeno.transformers.adverbs. **Transformer** (*sempos={'adv': 'r', 'noun': 'n', 'verb': 'v', 'propn': 'n', 'adjective': 'j', 'adj': 'j', 'adverb': 'r'}*, ***kwds*)

Bases: *grafeno.transformers.pos_extract.Transformer*

Processes adverbial modification as ATTR .

Methods

transform_dep (*dep*, *parent*, *child*)

grafeno.transformers.all module

class grafeno.transformers.all. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

This transformer carries over all morphological nodes and syntactic dependencies to the semantic level. It is good for developing/debugging purposes, since it directly translates the dependency tree into the semantic graph.

Methods

transform_dep (*dependency*, *parent*, *child*)
The functor is the dependency name.

transform_node (*msnode*)
The concept is the lemma of the morphological node.

grafeno.transformers.attr_class module

class grafeno.transformers.attr_class. **Transformer** (*attribute_class_keywords={'appearance': 'appearance', 'shape': 'shape', 'size': 'size', 'times': 'time', 'time': 'time', 'color': 'color', 'colour': 'color'}*, ***kwds*)
Bases: *grafeno.transformers.wordnet.Transformer*

Specifies `ATTR` edges by trying to find the specific property name and adding it as a `class` grammateme. It relies on WordNet definitions.

For example, if there is a `SWAN -- ATTR --> BLUE` edge, the `class` attribute with value `color` will be added to it.

Parameters **atribute_class_keywords** : dict

Specifies a non-default mapping from keywords to property class names. The class name with most keywords found in the WordNet definitions will be chosen.

Methods

post_process ( )

grafeno.transformers.base module

class grafeno.transformers.base. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: `object`

This class is the basic transformer class. Other transformers should inherit from it either directly or indirectly.

Transformer composition in *grafeno* uses cooperative inheritance. When a new module is written, it should extend the base class, or it can extend one or more other transformers which provide some required functionality. This is a way of managing dependencies, since the base classes will be inserted by Python into the inheritance chain.

The new module can then extend the methods it is interested in, adding some processing to that stage. However, for the chaining to work, every extended method has to make sure to:

1. Call `super()` with the correct (original) arguments at the very beggining of the function body.
2. Return the appropriate value, either modified, or untouched as returned from `super()`.

Some attributes are present in the transformer during processing. They can be used and modified in the appropriate stages.

Parameters **graph** : *Graph*

The graph to which all transformed text will be added.

lang : string

Language code to use for parsing, and available to any subclassing transformers.

Attributes

graph	(*Graph*) The graph to which all transformed text will be added.
stage	(string) Useful for debugging or checking, this string denotes what stage the processing is currently in.
nodes	(dict of *semantic nodes*) Available from pre_process up to post_process. Map of semantic nodes obtained for the sentence being processed, indexed by id.
edges	(list of *semantic edges*) Available from pre_process up to post_process. List of semantic edges obtained for the sentence being processed.

Methods

after_all ( )
Called at the end of processing a full text, after all sentences.

before_all ( )
Called at the beggining of processing a full text, before any sentences.

merge (*a*, *b*)
TODO: maybe this should be moved to the utils transformer.

Combine two nodes by id. Update all outgoing and incoming edges. All properties of b are lost, the ones in a are kept. a can be an existing graph node, b should be a node currently being processed.

Note: Can be done only during post_process.

---

parse_text (*text*)
Return a list of dependency treees.

post_insertion (*sentence_nodes*)
Called after the processed nodes and edges are added to the semantic graph. It is useful if some processing needs the real graph ids of the new nodes.

Parameters **sentence_nodes** : list of ids

The definitive (graph) ids of the nodes that were produced by analyzing the current sentence.

post_process ( )
This method is called after all nodes and dependency relations are processed. Sentence-level processing should be done here, as well as any node or edge merging or destruction.

Even though there are no parameters or return values, extenders should still call `super()` at the beggining. Semantic nodes and edges are available in the transformer's (self) `nodes` and `edges` properties.

pre_process (*tree*)
Prepares the transformer for processing a new sentence. Transformers can extend this method to initialize per-sentence variables.

Parameters **tree** : dict

The dependency parse of the sentence.

transform_dep (*dependency*, *parent*, *child*)
Transforms a dependency relation into a semantic edge.

Transformers should extend this module if any processing should occur for each dependency relation.

Parameters **dependency** : string

Name of the dependency relation.

parent, child : int

Temporary ids of the source and target semantic nodes. Note that these provisional nodes might not turn into true semantic nodes in the graph, if they don't have a *concept* attribute by the end of processing.

Returns *semantic edge*

transform_node (*msnode*)
Transform a morphosyntactic node to a semantic one.

Transformers should extend this module if any processing should occur for individual nodes.

The parser module should add to it an *id* property with the temporary id to use to refer to it.

Parameters **msnode** : dict

Dictionary of morphosyntactic tags

Returns *semantic node*

transform_text (*text*)
Transforms a list of sentences into the semantic graph.

It shouldn't be overriden.

grafeno.transformers.concept_class module

class grafeno.transformers.concept_class. **Transformer** (*concept_class_hypernyms=True*, ***kwds*)
Bases: *grafeno.transformers.wordnet.Transformer*

Finds the wordnet-defined 'class' of a concept.

Parameters **concept_class_hypernyms** : bool

If True, a new node is added with the class concept, related to the original node by an ''HYP" edge.

Methods

post_process ( )

grafeno.transformers.conjunction module

class grafeno.transformers.conjunction. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

post_process ( )

transform_dep (*dep*, *parent*, *child*)

grafeno.transformers.copula module

class grafeno.transformers.copula. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Processes copulative verbs, changing the functor of all its arguments to the same value: 'COP'. This reflects the simmetry of copulative relations, so the resulting graph is independent of the surface expression.

Methods

transform_dep (*dep*, *pid*, *cid*)

grafeno.transformers.edge_reverse module

class grafeno.transformers.edge_reverse. **Transformer** (*reversed_edges={'AGENT'}*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Reverses the direction of some edges.

Parameters **reversed_edges** : set

Set of functors which should have reverse orientation from the syntactic dependency.

Methods

transform_dep (*dep*, *pid*, *cid*)

grafeno.transformers.extend module

class grafeno.transformers.extend. **Transformer** (*extend_min_depth=4*, ***kwds*)
Bases: *grafeno.transformers.wordnet.Transformer*, grafeno.transformers.__utils.Transformer

Adds to the graph all WordNet hypernyms of every possible concept node.

The hypernyms are added as nodes with grammateme ''hyper = True'', and related by edges with functor ''HYP''.

Parameters **extend_min_depth** : int

Minimum depth of hypernyms to add. This depth is defined as the shortest path from the synset to the root of the WordNet hypernym hierarchy.

Methods

post_process ( )

grafeno.transformers.freeling_parse module

class grafeno.transformers.freeling_parse. **Transformer** (***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

parse_text (*text*)
Calls the freeling process to obtain the dependency parse of a text.

transform_node (*msnode*)

transform_tree (*tree*)

grafeno.transformers.genitive module

class grafeno.transformers.genitive. **Transformer** (*attach_genitive=False*, *add_genitive_class=True*, ***kwds*)
Bases: grafeno.transformers.__utils.Transformer

Processes genitive relations. Does two main things:

1. Turns saxon genitive ('s) into the preposition of with the correct dependencencies. This means that it must appear before preposition processing nodes in the transformer chain. 2. If enabled, collapses of edges, adding the information to the parent node.

Parameters **attach_genitive** : bool

If True, the concept is attached to the parent concept. For example, john's father turns into a single node father_of_john , instead of a father node with an of edge to a john node.

add_genitive_class : bool

If both *attach_genitive* and *add_genitive_class* are True, a HYP edge is added with the original dependent concept.

Methods

transform_dep (*dep*, *parent*, *child*)

transform_node (*ms*)

grafeno.transformers.index module

class grafeno.transformers.index. **Transformer** (***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

grafeno.transformers.interrogative module

class grafeno.transformers.interrogative. **Transformer** (***kwds*)
Bases: *grafeno.transformers.pos_extract.Transformer*

Methods

post_insertion (*sentence_nodes*)

transform_node (*msnode*)

grafeno.transformers.keep_deps module

class grafeno.transformers.keep_deps. **Transformer** (*dep_translate={'dobj': 'THEME', 'ncsubj': 'AGENT', 'iobj': 'ARG'}, unknown_dep_translate=''*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Converts syntactic dependency relations directly into semantic edges. It uses a translation table to find the appropriate functor given a syntactic dependency.

Parameters **dep_translate** : dict

A map from syntactic function to functor.

unknown_dep_translate : functor

Functor to use for unknown dependencencies.

Methods

transform_dep (*dep*, *pid*, *cid*)

grafeno.transformers.lenient module

class grafeno.transformers.lenient. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Removes edges where parent or child node don't have a concept.

This might necessary because otherwise these edges would give an error when trying to be added to the graph. Ideally, this situation should never happen, but sometimes nodes get dropped after the edges have already been processed.

Methods

post_process ( )

grafeno.transformers.lesk_link module

class grafeno.transformers.lesk_link. **Transformer** (*sim_threshold=100*, *sim_weight=1*, ***kwds*)

Bases: *grafeno.transformers.sim_link.Transformer*

Methods

get_similarity (*a*, *b*)

grafeno.transformers.negation module

class grafeno.transformers.negation. **Transformer** (*polarity_grammateme='polarity'*, *positive_polarity='+'*, *negative_polarity='-'*, ***kwds*)

Bases: *grafeno.transformers.base.Transformer*

Processes negation and its scope, setting the polarity of the affected verb.

Parameters **polarity_grammateme** : string

Name of the grammateme to store polarity

positive_polarity : string

Value for the polarity grammateme when affirmative/positive

negative_polarity : string

Value for the polarity grammateme when negative

Methods

transform_dep (*dep*, *pid*, *cid*)

Rise negation until a verb is found, which is then marked negative. Modal negative particles are also processed.

transform_node (*ms*)

Find negative particles, and by default mark all verbs as affirmative.

grafeno.transformers.nouns module

class grafeno.transformers.nouns. **Transformer** (*sempos={'adv': 'r', 'noun': 'n', 'verb': 'v', 'propn': 'n', 'adjective': 'j', 'adj': 'j', 'adverb': 'r'}*, ***kwds*)

Bases: *grafeno.transformers.pos_extract.Transformer*

Processes noun grammatemes and noun-noun modifications, such as apposition.

Methods

transform_dep (*dep*, *pid*, *cid*)

transform_node (*msnode*)

grafeno.transformers.numerals module

class grafeno.transformers.numerals. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

transform_dep (*dep*, *parent*, *child*)

transform_node (*ms*)

grafeno.transformers.phrasal module

class grafeno.transformers.phrasal. **Transformer** (*guess_phrasals=True*, ***kwds*)
Bases: *grafeno.transformers.thematic.Transformer*

Methods

post_process ( )

pre_process (*tree*)

transform_dep (*dep*, *pid*, *cid*)

transform_node (*msnode*)

grafeno.transformers.pos_extract module

class grafeno.transformers.pos_extract. **Transformer** (*sempos={'adv': 'r', 'noun': 'n', 'verb': 'v', 'propn': 'n', 'adjective': 'j', 'adj': 'j', 'adverb': 'r'}*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

transform_node (*msnode*)

grafeno.transformers.prepositions module

class grafeno.transformers.prepositions. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Processes prepositions, trying to turn them into `COMP` edges with the preposition lemma as the `class` grammateme.

These edges join the prepositional phrase nucleus (direct dependent of the preposition, head) with the parent (direct dominating node of the preposition).

Methods

post_process ( )

transform_dep (*dep*, *parent*, *child*)

transform_node (*msnode*)

grafeno.transformers.pronouns module

class grafeno.transformers.pronouns. **Transformer** (***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

transform_dep (*dep*, *pid*, *cid*)

transform_node (*ms*)

grafeno.transformers.relative module

class grafeno.transformers.relative. **Transformer** (***kwds*)
Bases: *grafeno.transformers.interrogative.Transformer*

Methods

post_process ( )

transform_dep (*dep*, *pid*, *cid*)

grafeno.transformers.sentences module

class grafeno.transformers.sentences. **Transformer** (*extended_sentence_edges=None*, ***kwds*)
Bases: *grafeno.transformers.base.Transformer*

Methods

post_insertion (*sentence_nodes*)

pre_process (*tree*)

grafeno.transformers.sim_link module

class grafeno.transformers.sim_link. **Transformer** (*sim_threshold=0.1*, *sim_weight=1*, ***kwds*)

Bases: *grafeno.transformers.wordnet.Transformer*

Methods

get_similarity (*a*, *b*)

post_insertion (*sentence_nodes*)

grafeno.transformers.spacy_parse module

grafeno.transformers.specific_edges module

class grafeno.transformers.specific_edges. **Transformer** (*graph=None*, *lang='en'*, ***kwds*)

Bases: *grafeno.transformers.base.Transformer*

Methods

post_process ( )

grafeno.transformers.thematic module

class grafeno.transformers.thematic. **Transformer** (*sempos={'adv': 'r', 'noun': 'n', 'verb': 'v', 'propn': 'n', 'adjective': 'j', 'adj': 'j', 'adverb': 'r'}*, ***kwds*)

Bases: *grafeno.transformers.pos_extract.Transformer*

Methods

post_process ( )

predication = {'nsubjpass': ('THEME', 1.0, {'n'}), 'nsubj': ('AGENT', 1.0, {'n'}), 'obl': ('ARG', 1.0, None), 'agent':

transform_dep (*dep*, *pid*, *cid*)

transform_node (*msnode*)

grafeno.transformers.unique module

class grafeno.transformers.unique. **Transformer** (*unique_gram=None*, ***kwds*)

Bases: *grafeno.transformers.index.Transformer*

Methods

post_insertion (*sentence_nodes*)

post_process ( )

grafeno.transformers.verb_collapse module

class grafeno.transformers.verb_collapse. **Transformer** (*sempos={}, main_argument=['dobj', 'iobj', 'ncmod'], **kwds*)

Bases: *grafeno.transformers.pos_extract.Transformer*

Methods

post_process ( )

transform_dep (*dep*, *parent*, *child*)

grafeno.transformers.wordnet module

class grafeno.transformers.wordnet. **Transformer** (***kwds*)

Bases: *grafeno.transformers.base.Transformer*

Methods

post_process ( )

Submodules

grafeno.graph module

This module provides the main Graph class. Graph objects are the core of the library, and most operations revolve around manipulating them.

```
from grafeno import Graph as CG

g = CG(transformer = MyTransformer)
print(g.linearize(linearizer = MyLinearizer))
```

class grafeno.graph. **Graph** (*original=None, transformer=None, transformer_args={}, text=None, subgraph=None, from_networkx=None*)

Bases: object

Semantic graph class. Nodes represent concepts, while edges stand for the relations between them.

Parameters **transformer** : *Transformer* , optional

If provided, it will be used to transform all text added to the graph into semantic nodes and edges.

transformer_args : dict, optional

Arguments for the *transformer* class.

text : string, optional

If provided, this text will be added to the graph (transformed with the *transformer* class).

original : Graph, optional

If provided, the new graph will be initialized with the existing information in *original*.

subgraph : bunch of nodes

If *original* and *subgraph* are provided, only the nodes in *subgraph* will be copied over from *original*.

Attributes

gram	(dict) dictionary of parameters global to the conceptual graph.
node	(dict) dictionary of concept nodes, indexed by node id.

Methods

add_edge (*head*, *dependent*, *functor*, ***gram*)
Creates a semantic edge between two concept nodes in the graph.

Parameters **head, dependent** : node_id

The graph ids of the nodes to link. The edge is directed, from head to dependent.

functor : string

The textual representation of the _functor_, the name of the relation between the concepts.

gram : keyword args, optional

Additional 'grammatemes', a free-form python dict of attributes to attach to the edge.

Raises **ValueError**

When the head or dependent id's are not valid.

add_node (*concept*, ***gram*)
Creates a concept node in the graph.

Parameters **concept** : string

The (non-unique) textual representation of the concept node.

gram : keyword args, optional

Additional 'grammatemes', a free-form python dict of attributes to attach to the node.

Returns int

The graph id of the newly created node.

add_text (*text*)
Processes a text, and adds the resulting nodes and edges to the graph.

Parameters **text** : string

A clean text to process and add to the graph.

all_edges ( )
Iterates over all the edges in the graph.

Returns An iterator over all the edges of the graph, in the form of tuples

(head id, dependent id, edge).

draw (*bunch=None*)
Draws the graph on screen.

Note: Requires matplotlib and a compatible configured environment.

Parameters **bunch** : list of nodes

An iterable of node ids to draw, if `None` then all nodes are included.

edges (*nid*)
Returns a dictionary of the dependents of a node.

Parameters **nid** : int

ID of the node

Returns A dictionary of edges, keyed by neighbor id, and with data

the grammatemes of the edge.

linearize (*linearizer=None*, *linearizer_args={}*)
Linearizes a graph into a string.

Parameters **linearizer** : *Linearizer* , optional

If provided, from this point on all linearizations of the graph will use an instance of this class. The linearizer is used to transform the semantic data, nodes and edges, into a string representation.

linearizer_args : dict, optional

Arguments for the *linearizer* class.

Returns A string, the result of running the linearizer on the graph.

neighbours (*node*)
Iterates over the neighbours of a node, giving the edge information for each neighbour.

```
node = graph.node[0]
for neighbour, edge in graph.neighbours(node)
    print('{}-{}->{}'.format(
             node['concept'],
             edge['functor'],
             neighbour['concept']))
```

Parameters **node** : node

The node in the graph to explore

Returns An iterator over the neighbours of the node, in the form of tuples

(node, edge).

nodes ( )
Returns a list of all the nodes in the graph. Each node is represented as a dictionary of concept and further grammatemes.

to_json (*with_labels=True*)
Returns a JSON representation of the graph data.

Parameters **with_labels** : bool

If True, a 'label' attribute is added to nodes and edges with the _concept_ and _functor_, respectively. Useful for further consuming by some libraries.

Returns A string with the graph data encoded in JSON.

grafeno.pipeline module

The pipeline module allows the user to write full pipelines of experiments in a dict, which can then be loaded and run by the library with one function call:

```
from grafeno import pipeline

experiment = {
    'text': 'Colorless green ideas sleep furiously.',
    'parser': 'freeling',
    'transformers': [ 'all' ],
    'linearizers': [ 'triplets' ]
}

result = pipeline.run(experiment)
print(result)
```

Pipeline Formatting

The following attributes for the pipeline dict are supported.

Note: The pipeline is designed so that it can be easily serialized and loaded from a string format such as YAML, making repeatable experiments as easy as writing into a text file what operations to perform, and with what arguments.

Input

Input to the pipeline is required. It can be an already constructed *graph*, otherwise *text*, *parser* and *transformers* will be needed.

- *graph*: a *Graph*
- *text*: a raw natural language text.
- ***parser*: what parser to use to process the text. Possible values are:**
  - `freeling` : http://nlp.lsi.upc.edu/freeling/node/1
  - `spacy` : https://spacy.io

Note: This is just a shortcut for using as first transformer a module named `<parser_type>_parser`. This allows parsers to be changed easily and independently from the rest of the pipeline.

Warning: To use a specific parser, it must be installed and available to grafeno. For *freeling*, the *analyze* executable must be in the path, in the case of *spacy*, the module must be importable.

- *transformers*: list of transformer names to use (see *grafeno.transformers*)
- *transformer_args*: dict of arguments for the *transformers*

Operation

- *operations*: a list of dicts, each with an `op` attribute with the operation name, and the rest of the arguments to be used as parameters for the operation.

Output

A text if a *linearizers* attribute is present, otherwise the raw *graph* obtained is returned.

- *linearizers*: list of linearizer names to use (see *grafeno.linearizers*)
- *linearizer_args*: dict of arguments for the linearizers

See also:

Some pre-built pipelines can be found in the `config` directory, written in YAML: *Pre-built Pipelines*.

`grafeno.pipeline.` **`run`** (*pipeline*)

Run a complete pipeline of graph operations.

Parameters **pipeline** : dict

The pipeline description.

Returns The result from running the pipeline with the provided arguments.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Credits

Authors:

- Antonio F. G. Sevilla <afgs@ucm.es>
- Alberto Díaz <albertodiaz@fdi.ucm.es>

Acknowledgements

The continued development of this library has been possible thanks to a number of different research and development projects, listed below.

- A collaboration with MedWhat (https://medwhat.com/), a company that develops virtual medical assistant bots and other medical artificial intelligence solutions.
- This research is funded by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund (TIN2015-66655-R (MINECO/FEDER)).
- This work is funded by ConCreTe. The project ConCreTe acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 611733.

g

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W