Decomp

Release 0.2.2

Aaron Steven White

Jun 08, 2022

CONTENTS:

Decomp is a toolkit for working with the Universal Decompositional Semantics (UDS) dataset, which is a collection of directed acyclic semantic graphs with real-valued node and edge attributes pointing into Universal Dependencies syntactic dependency trees.

The toolkit is built on top of NetworkX and RDFLib making it straightforward to:

- read the UDS dataset from its native JSON format
- query both the syntactic and semantic subgraphs of UDS (as well as pointers between them) using SPARQL 1.1 queries
- serialize UDS graphs to many common formats, such as Notation3, N-Triples, turtle, and JSON-LD, as well as any other format supported by NetworkX

The toolkit was built by Aaron Steven White and is maintained by the Decompositional Semantics Initiative. The UDS dataset was constructed from annotations collected by the Decompositional Semantics Initiative.

If you use either UDS or Decomp in your research, we ask that you cite the following paper:

> White, Aaron Steven, Elias Stengel-Eskin, Siddharth Vashishtha, Venkata Subrahmanyan Govindarajan, Dee Ann Reisinger, Tim Vieira, Keisuke Sakaguchi, et al. 2020. The Universal Decompositional Semantics Dataset and Decomp Toolkit. *Proceedings of The 12th Language Resources and Evaluation Conference*, 5698–5707. Marseille, France: European Language Resources Association.

```
@inproceedings{white-etal-2020-universal,
    title = "The Universal Decompositional Semantics Dataset and Decomp Toolkit",
    author = "White, Aaron Steven  and
      Stengel-Eskin, Elias  and
      Vashishtha, Siddharth  and
      Govindarajan, Venkata Subrahmanyan  and
      Reisinger, Dee Ann  and
      Vieira, Tim  and
      Sakaguchi, Keisuke  and
      Zhang, Sheng  and
      Ferraro, Francis  and
      Rudinger, Rachel  and
      Rawlins, Kyle  and
      Van Durme, Benjamin",
    booktitle = "Proceedings of The 12th Language Resources and Evaluation Conference",
    month = may,
    year = "2020",
    address = "Marseille, France",
    publisher = "European Language Resources Association",
    url = "https://www.aclweb.org/anthology/2020.lrec-1.699",
    pages = "5698--5707",
    ISBN = "979-10-95546-34-4",
}
```

CHAPTER

ONE

INSTALLATION

The most painless way to get started quickly is to use the included barebones Python 3.6-based Dockerfile. To build the image and start a python interactive prompt, use:

```
git clone git://gitlab.hltcoe.jhu.edu/aswhite/decomp.git
cd decomp
docker build -t decomp .
docker run -it decomp python
```

A jupyter notebook can then be opened in the standard way.

Decomp can also be installed to a local environment using `pip`.

```
pip install git+git://github.com/decompositional-semantics-initiative/decomp.git
```

As an alternative to `pip` you can clone the decomp repository and use the included `setup.py` with the `install` flag.

```
git clone https://github.com/decompositional-semantics-initiative/decomp.git
cd decomp
pip install --user --no-cache-dir -r ./requirements.txt
python setup.py install
```

If you would like to install the package for the purposes of development, you can use the included `setup.py` with the `develop` flag.

```
git clone https://github.com/decompositional-semantics-initiative/decomp.git
cd decomp
pip install --user --no-cache-dir -r ./requirements.txt
python setup.py develop
```

If you have trouble installing via setup.py or pip on OS X Mojave, adding the following environment variables may help.

```
CXXFLAGS=-stdlib=libc++ CFLAGS=-stdlib=libc++ python setup.py install
```

CHAPTER

TWO

TUTORIALS

If you have not already *installed* the decomp package, follow those instructions before continuing the tutorial.

2.1 Quick Start

To read the Universal Decompositional Semantics (UDS) dataset, use:

```
from decomp import UDSCorpus

uds = UDSCorpus()
```

This imports a UDSCorpus object `uds`, which contains all graphs across all splits in the data. If you would like a corpus, e.g., containing only a particular split, see other loading options in *Reading the UDS dataset*.

The first time you read UDS, it will take several minutes to complete while the dataset is built from the Universal Dependencies English Web Treebank, which is not shipped with the package (but is downloaded automatically on import in the background), and the UDS annotations, which are shipped with the package. Subsequent uses will be faster, since the dataset is cached on build.

UDSSentenceGraph objects in the corpus can be accessed using standard dictionary getters or iteration. For instance, to get the UDS graph corresponding to the 12th sentence in `en-ud-train.conllu`, you can use:

```
uds["ewt-train-12"]
```

To access documents (UDSDocument objects, each of which has an associated UDSDocumentGraph), you can use:

```
uds.documents["reviews-112579"]
```

To get the associated document graph, use:

```
uds.documents["reviews-112579"].document_graph
```

More generally, `UDSCorpus` objects behave like dictionaries. For example, to print all the sentence-level graph identifiers in the corpus (e.g. `"ewt-train-12"`), you can use:

```
for graphid in uds:
    print(graphid)
```

To print all the document identifiers in the corpus, which correspond directly to English Web Treebank file IDs (e.g. `"reviews-112579"`), you can use:

```
for documentid in uds.documents:
    print(documentid)
```

Similarly, to print all the sentence-level graph identifiers in the corpus (e.g. `"ewt-train-12"`) along with the corresponding sentence, you can use:

```
for graphid, graph in uds.items():
    print(graphid)
    print(graph.sentence)
```

Likewise, the following will print all document identifiers, along with each document's entire text:

```
for documentid, document in uds.documents.items():
    print(documentid)
    print(document.text)
```

A list of sentence-level graph identifiers can also be accessed via the `graphids` attribute of the UDSCorpus. A mapping from these identifiers and the corresponding graph can be accessed via the `graphs` attribute.

```
# a list of the sentence-level graph identifiers in the corpus
uds.graphids

# a dictionary mapping the sentence-level
# graph identifiers to the corresponding graph
uds.graphs
```

A list of document identifiers can also be accessed via the `document_ids` attribute of the UDSCorpus:

```
uds.document_ids
```

For sentence-level graphs, there are various instance attributes and methods for accessing nodes, edges, and their attributes in the UDS sentence-level graphs. For example, to get a dictionary mapping identifiers for syntax nodes in a sentence-level graph to their attributes, you can use:

```
uds["ewt-train-12"].syntax_nodes
```

To get a dictionary mapping identifiers for semantics nodes in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].semantics_nodes
```

To get a dictionary mapping identifiers for semantics edges (tuples of node identifiers) in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].semantics_edges()
```

To get a dictionary mapping identifiers for semantics edges (tuples of node identifiers) in the UDS graph involving the predicate headed by the 7th token to their attributes, you can use:

```
uds["ewt-train-12"].semantics_edges('ewt-train-12-semantics-pred-7')
```

To get a dictionary mapping identifiers for syntax edges (tuples of node identifiers) in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].syntax_edges()
```

And to get a dictionary mapping identifiers for syntax edges (tuples of node identifiers) in the UDS graph involving the node for the 7th token to their attributes, you can use:

```
uds["ewt-train-12"].syntax_edges('ewt-train-12-syntax-7')
```

There are also methods for accessing relationships between semantics and syntax nodes. For example, you can get a tuple of the ordinal position for the head syntax node in the UDS graph that maps of the predicate headed by the 7th token in the corresponding sentence to a list of the form and lemma attributes for that token, you can use:

```
uds["ewt-train-12"].head('ewt-train-12-semantics-pred-7', ['form', 'lemma'])
```

And if you want the same information for every token in the span, you can use:

```
uds["ewt-train-12"].span('ewt-train-12-semantics-pred-7', ['form', 'lemma'])
```

This will return a dictionary mapping ordinal position for syntax nodes in the UDS graph that make of the predicate headed by the 7th token in the corresponding sentence to a list of the form and lemma attributes for the corresponding tokens.

More complicated queries of a sentence-level UDS graph can be performed using the `query` method, which accepts arbitrary SPARQL 1.1 queries. See *Querying UDS Graphs* for details.

Queries on document-level graphs are not currently supported. However, each UDSDocument does contain a number of useful attributes, including its `genre` (corresponding to the English Web Treebank subcorpus); its `text` (as demonstrated above); its `timestamp`; the `sentence_ids` of its constituent sentences; and the sentence-level graphs (`sentence_graphs`) associated with those sentences. Additionally, one can also look up the semantics node associated with a particular node in the document graph via the semantics_node instance method.

Lastly, iterables for the nodes and edges of a document-level graph may be accessed as follows:

```
uds.documents["reviews-112579"].document_graph.nodes
uds.documents["reviews-112579"].document_graph.edges
```

Unlike the nodes and edges in a sentence-level graph, the ones in a document- level graph all share a common (`document`) domain. By default, document graphs are initialized without edges and with one node for each semantics node in the sentence-level graphs associated with the constituent sentences. Edges may be added by supplying annotations (see *Reading the UDS dataset*).

2.2 Reading the UDS dataset

The most straightforward way to read the Universal Decompositional Semantics (UDS) dataset is to import it.

```
from decomp import UDSCorpus

uds = UDSCorpus()
```

This loads a UDSCorpus object `uds`, which contains all graphs across all splits in the data.

As noted in *Quick Start*, the first time you do read UDS, it will take several minutes to complete while the dataset is built from the Universal Dependencies English Web Treebank (UD-EWT), which is not shipped with the package (but is downloaded automatically on import in the background), and the UDS annotations, which are shipped with the package as package data. Normalized annotations are loaded by default. To load raw annotations, specify `"raw"` as the argument to the UDSCorpus `annotation_format` keyword arugment as follows:

```
from decomp import UDSCorpus

uds = UDSCorpus(annotation_format="raw")
```

(See *Adding annotations* below for more detail on annotation types.) Subsequent uses of the corpus will be faster after the initial build, since the built dataset is cached.

2.2.1 Standard splits

If you would rather read only the graphs in the training, development, or test split, you can do that by specifying the `split` parameter of `UDSCorpus`.

```
from decomp import UDSCorpus

# read the train split of the UDS corpus
uds_train = UDSCorpus(split='train')
```

2.2.2 Adding annotations

Additional annotations beyond the standard UDS annotations can be added using this method by passing a list of UDSAnnotation objects. These annotations can be added at two levels: the sentence level and the document level. Sentence-level annotations contain attributes of UDSSentenceGraph nodes or edges. Document-level annotations contain attributes for UDSDocumentGraph nodes or edges. Document-level edge annotations may relate nodes associated with different sentences in a document, although they are added as annotations only to the the appropriate UDSDocumentGraph.

Sentence-level and document-level annotations share the same two in-memory representations: `RawUDSDataset` and `NormalizedUDSDataset`. The former may have multiple annotations for the same node or edge attribute, while the latter must have only a single annotation. Both are loaded from JSON-formatted files, but differ in the expected format (see the from_json methods of each class for formatting guidelines). For example, if you have some additional *normalized* sentence-level annotations in a file `new_annotations.json`, those can be added to the existing UDS annotations using:

```
from decomp import NormalizedUDSDataset

# read annotations
new_annotations = [NormalizedUDSDataset.from_json("new_annotations.json")]

# read the train split of the UDS corpus and append new annotations
uds_train_plus = UDSCorpus(split='train', sentence_annotations=new_annotations)
```

If instead you wished to add *raw* annotations (and supposing those annotations were still in "new_annotations.json"), you would do the following:

```
from decomp import RawUDSDataset

# read annotations
new_annotations = [RawUDSDataset.from_json("new_annotations.json")]

# read the train split of the UDS corpus and append new annotations
uds_train_plus = UDSCorpus(split='train', sentence_annotations=new_annotations,
                           annotation_format="raw")
```

If `new_annotations.json` contained document-level annotations you would pass `new_annotations.json` to the constructor keyword argument `document_annotations` instead of to `sentence_annotations`. Importantly, these annotations are added *in addition* to the existing UDS annotations that ship with the toolkit. You do not need to add these manually.

Finally, it should be noted that querying is currently **not** supported for document-level graphs or for sentence-level graphs containing raw annotations.

2.2.3 Reading from an alternative location

If you would like to read the dataset from an alternative location—e.g. if you have serialized the dataset to JSON, using the to_json instance method—this can be accomplished using `UDSCorpus` class methods (see *Serializing the UDS dataset* for more information on serialization). For example, if you serialize `uds_train` to the files `uds-ewt-sentences-train.json` (for sentences) and `uds-ewt-documents-train.json` (for the documents), you can read it back into memory using:

```
# serialize uds_train to JSON
uds_train.to_json("uds-ewt-sentences-train.json", "uds-ewt-documents-train.json")

# read JSON serialized uds_train
uds_train = UDSCorpus.from_json("uds-ewt-sentences-train.json", "uds-ewt-documents-train.
↪json")
```

2.2.4 Rebuilding the corpus

If you would like to rebuild the corpus from the UD-EWT CoNLL files and some set of JSON-formatted annotation files, you can use the analogous from_conll class method. Importantly, unlike the standard instance initialization described above, the UDS annotations are *not* automatically added. For example, if `en-ud-train.conllu` is in the current working directory and you have already loaded `new_annotations` as above, a corpus containing only those annotations (without the UDS annotations) can be loaded using:

```
# read the train split of the UD corpus and append new annotations
uds_train_annotated = UDSCorpus.from_conll("en-ud-train.conllu", sentence_
↪annotations=new_annotations)
```

This also means that if you only want the semantic graphs as implied by PredPatt (without annotations), you can use the `from_conll` class method to load them.

```
# read the train split of the UD corpus
ud_train = UDSCorpus.from_conll("en-ud-train.conllu")
```

Note that, because PredPatt is used for predicate-argument extraction, only versions of UD-EWT that are compatible with PredPatt can be used here. Version 1.2 is suggested.

Though other serialization formats are available (see *Serializing the UDS dataset*), these formats are not yet supported for reading.

2.3 Querying UDS Graphs

Decomp provides a rich array of methods for querying UDS graphs: both pre-compiled and user-specified. Arbitrary user-specified graph queries can be performed using the UDSSentenceGraph.query instance method. This method accepts arbitrary SPARQL 1.1 queries, either as strings or as precompiled Query objects built using RDFlib's prepareQuery.

NOTE: Querying is not currently supported for document-level graphs (UDSDocumentGraph objects) or for sentence-level graphs that contain raw annotations (RawUDSDataset).

2.3.1 Pre-compiled queries

For many use cases, the various instance attributes and methods for accessing nodes, edges, and their attributes in the UDS graphs will likely be sufficient; there is no need to use `query`. For example, to get a dictionary mapping identifiers for syntax nodes in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].syntax_nodes
```

To get a dictionary mapping identifiers for semantics nodes in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].semantics_nodes
```

To get a dictionary mapping identifiers for semantics edges (tuples of node identifiers) in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].semantics_edges()
```

To get a dictionary mapping identifiers for semantics edges (tuples of node identifiers) in the UDS graph involving the predicate headed by the 7th token to their attributes, you can use:

```
uds["ewt-train-12"].semantics_edges('ewt-train-12-semantics-pred-7')
```

To get a dictionary mapping identifiers for syntax edges (tuples of node identifiers) in the UDS graph to their attributes, you can use:

```
uds["ewt-train-12"].syntax_edges()
```

And to get a dictionary mapping identifiers for syntax edges (tuples of node identifiers) in the UDS graph involving the node for the 7th token to their attributes, you can use:

```
uds["ewt-train-12"].syntax_edges('ewt-train-12-syntax-7')
```

There are also methods for accessing relationships between semantics and syntax nodes. For example, you can get a tuple of the ordinal position for the head syntax node in the UDS graph that maps of the predicate headed by the 7th token in the corresponding sentence to a list of the form and lemma attributes for that token, you can use:

```
uds["ewt-train-12"].head('ewt-train-12-semantics-pred-7', ['form', 'lemma'])
```

And if you want the same information for every token in the span, you can use:

```
uds["ewt-train-12"].span('ewt-train-12-semantics-pred-7', ['form', 'lemma'])
```

This will return a dictionary mapping ordinal position for syntax nodes in the UDS graph that make of the predicate headed by the 7th token in the corresponding sentence to a list of the form and lemma attributes for the corresponding tokens.

2.3.2 Custom queries

Where the above methods generally turn out to be insufficient is in selecting nodes and edges on the basis of (combinations of their attributes). This is where having the full power of SPARQL comes in handy. This power comes with substantial slow downs in the speed of queries, however, so if you can do a query without using SPARQL you should try to.

For example, if you were interested in extracting only predicates referring to events that likely happened and likely lasted for minutes, you could use:

```
querystr = """
           SELECT ?pred
           WHERE { ?pred <domain> <semantics> ;
                         <type> <predicate> ;
                         <factual> ?factual ;
                         <dur-minutes> ?duration
                         FILTER ( ?factual > 0 && ?duration > 0 )
                 }
           """

results = {gid: graph.query(querystr, query_type='node', cache_rdf=False)
           for gid, graph in uds.items()}
```

Or more tersely (but equivalently):

```
results = uds.query(querystr, query_type='node', cache_rdf=False)
```

Note that the `query_type` parameter is set to `'node'`. This setting means that a dictionary mapping node identifiers to node attribute values will be returned. If no such query type is passed, an RDFLib Result object will be returned, which you will need to postprocess yourself. This is necessary if, for instance, you are making a `CONSTRUCT`, `ASK`, or `DESCRIBE` query.

Also, note that the `cache_rdf` parameter is set to `False`. This is a memory-saving measure, as `UDSSentenceGraph.query` implicitly builds an RDF graph on the backend, and these graphs can be quite large. Leaving `cache_rdf` at its defaults of `True` will substantially speed up later queries at the expense of sometimes substantial memory costs.

Constraints can also make reference to node and edge attributes of other nodes. For instance, if you were interested in extracting all predicates referring to events that are likely spatiotemporally delimited and have at least one spatiotemporally delimited participant that was volitional in the event, you could use:

```
querystr = """
           SELECT DISTINCT ?node
           WHERE { ?node ?edge ?arg ;
                         <domain> <semantics> ;
                         <type>   <predicate> ;
                         <pred-particular> ?predparticular
                         FILTER ( ?predparticular > 0 ) .
                   ?arg  <domain> <semantics> ;
                         <type>   <argument>  ;
                         <arg-particular> ?argparticular
                         FILTER ( ?argparticular > 0 ) .
                   ?edge <volition> ?volition
                         FILTER ( ?volition > 0 ) .
                 }
           """
```

(continues on next page)

(continued from previous page)

```
results = uds.query(querystr, query_type='node', cache_rdf=False)
```

Disjunctive constraints are also possible. For instance, for the last query, if you were interested in either volitional or sentient arguments, you could use:

```
querystr = """
           SELECT DISTINCT ?node
           WHERE { ?node ?edge ?arg ;
                         <domain> <semantics> ;
                         <type>   <predicate> ;
                         <pred-particular> ?predparticular
                         FILTER ( ?predparticular > 0 ) .
                   ?arg  <domain> <semantics> ;
                         <type>   <argument>  ;
                         <arg-particular> ?argparticular
                         FILTER ( ?argparticular > 0 ) .
                   { ?edge <volition> ?volition
                           FILTER ( ?volition > 0 )
                   } UNION
                   { ?edge <sentient> ?sentient
                           FILTER ( ?sentient > 0 )
                   }
                 }
           """

results = uds.query(querystr, query_type='node', cache_rdf=False)
```

Beyond returning node attributes based on complex constraints, you can also return edge attributes. For instance, for the last query, if you were interested in all the attributes of edges connecting predicates and arguments satisfying the constraints of the last query, you could simply change which variable is bound by `SELECT` and set `query_type` to `'edge'`.

```
querystr = """
           SELECT ?edge
           WHERE { ?node ?edge ?arg ;
                         <domain> <semantics> ;
                         <type>   <predicate> ;
                         <pred-particular> ?predparticular
                         FILTER ( ?predparticular > 0 ) .
                   ?arg  <domain> <semantics> ;
                         <type>   <argument>  ;
                         <arg-particular> ?argparticular
                         FILTER ( ?argparticular > 0 ) .
                   { ?edge <volition> ?volition
                           FILTER ( ?volition > 0 )
                   } UNION
                   { ?edge <sentient> ?sentient
                           FILTER ( ?sentient > 0 )
                   }
                 }
           """
```

(continues on next page)

(continued from previous page)

```
results = uds.query(querystr, query_type='edge', cache_rdf=False)
```

2.4 Serializing the UDS dataset

The canonical serialization format for the Universal Decompositional Semantics (UDS) dataset is JSON. Sentence- and document-level graphs are serialized separately. For example, if you wanted to serialize the entire UDS dataset to the files `uds-sentence.json` (for sentences) and `uds-document.json` (for documents), you would use:

```
from decomp import uds

uds.to_json("uds-sentence.json", "uds-document.json")
```

The particular format is based directly on the adjacency_data method implemented in NetworkX

For the sentence-level graphs only, in addition to this JSON format, any serialization format supported by RDFLib can also be used by accessing the rdf attribute of each UDSSentenceGraph object. This attribute exposes an rdflib.graph.Graph object, which implements a serialize method. By default, this method outputs rdf/xml. The `format` parameter can also be set to `'n3'`, `'turtle'`, `'nt'`, `'pretty-xml'`, `'trix'`, `'trig'`, or `'nquads'`; and additional formats, such as JSON-LD, can be supported by installing plugins for RDFLib.

Before considering serialization to such a format, be aware that only the JSON format mentioned above can be read by the toolkit. Additionally, note that if your aim is to query the graphs in the corpus, this can be done using the query instance method in `UDSSentenceGraph`. See *Querying UDS Graphs* for details.

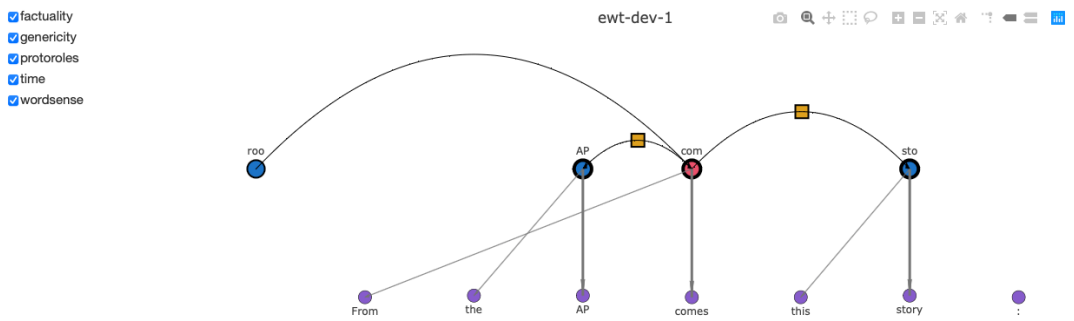
2.5 Visualizing UDS Graphs

Decomp comes with a built-in interactive visualization tool using the UDSVisualization object. This object visualizes a UDSSentenceGraph.

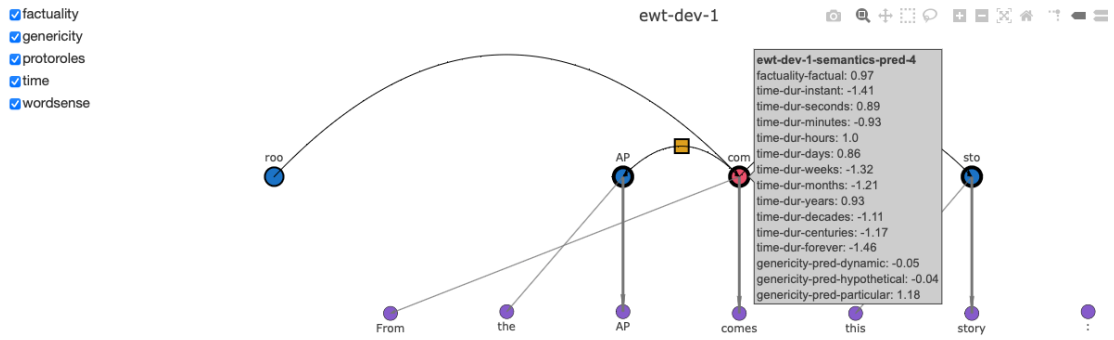
A visualization (which is based on Dash) is served to your local browser via port 8050 (e.g. *http://localhost:8050*). The following snippet visualizes the first graph in the dev split:

```
graph = uds["ewt-dev-1"]
vis = UDSVisualization(graph)
vis.serve()
```

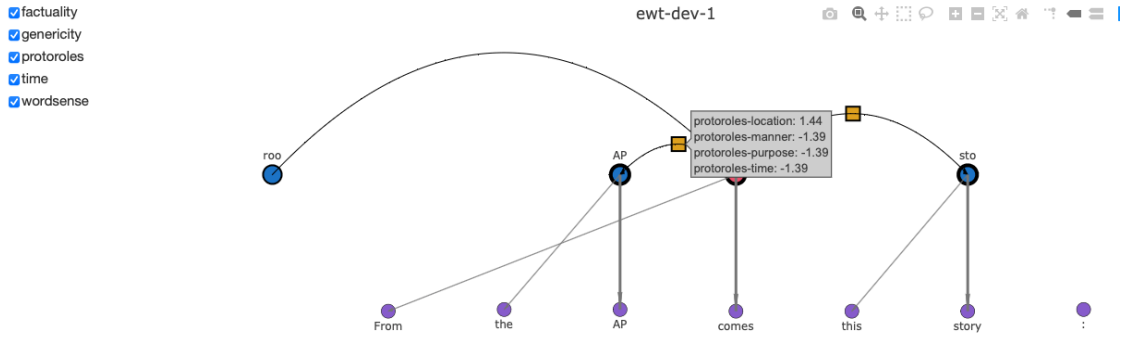
The browser window will look like this:



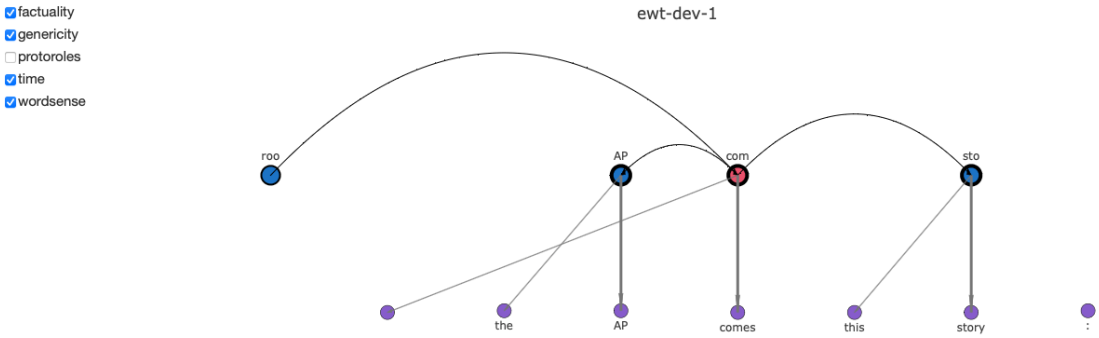
Black edges indicate edges in the semantic graph, while gray arrows are instance edges between semantics and syntax nodes. Thick gray arrows indicate the syntactic head of a semantic argument or predicate. Semantics nodes have a thick outline when they are annotated with decomp properties. Hovering over such a node will reveal the annotations in a pop-out window.



Similarly, yellow boxes on edges indicate protorole annotations, and can be hovered over to reveal their values.



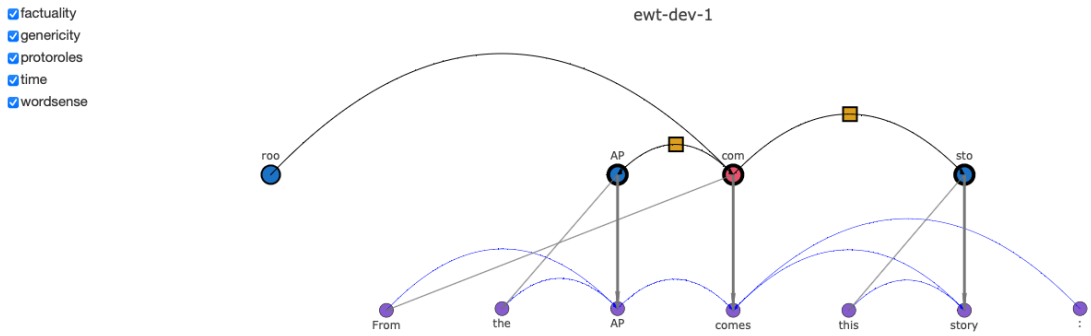
Using the checkboxes at the top left, annotation subspaces can be selected and de-selected. If all the annotations for a node or edge are de-selected, it will become non-bolded or disappear



Several options can be supplied to a visualization via arguments. For example, we can visualize the syntactic parse along with the semantic parse by setting

```
vis = UDSVisualization(graph, add_syntax_edges = True)
```

which results in the following visualization.


factuality
genericity
protoroles
time
wordsense
ewt-dev-1
roo
AP
com
sto
From
the
AP
comes
this
story
:

CHAPTER

THREE

DATASET REFERENCE

The Universal Decompositional Semantics (UDS) dataset consists of four layers of annotations built on top of the English Web Treebank (EWT).

3.1 Universal Dependencies Syntactic Graphs

The syntactic graphs that form the first layer of annotation in the dataset come from gold UD dependency parses provided in the UD-EWT treebank, which contains sentences from the Linguistic Data Consortium's constituency parsed EWT. UD-EWT has predefined training (`train`), development (`dev`), and test (`test`) data in corresponding files in CoNLL-U format: `en_ewt-ud-train.conllu`, `en_ewt-ud-dev.conllu`, and `en_ewt-ud-test.conllu`. Henceforth, `SPLIT` ranges over `train`, `dev`, and `test`.

In UDS, each dependency parsed sentence in UD-EWT is represented as a rooted directed graph (digraph). Each graph's identifier takes the form `ewt-SPLIT-SENTNUM`, where `SENTNUM` is the ordinal position (1-indexed) of the sentence within `en_ewt-ud-SPLIT.conllu`.

Each token in a sentence is associated with a node with identifier `ewt-SPLIT-SENTNUM-syntax-TOKNUM`, where `TOKNUM` is the token's ordinal position within the sentence (1-indexed, following the convention in UD-EWT). At minimum, each node has the following attributes.

- `position` (`int`): the ordinal position (`TOKNUM`) of that node as an integer (again, 1-indexed)
- `domain` (`str`): the subgraph this node is part of (always `syntax`)
- `type` (`str`): the type of the object in the particular domain (always `token`)
- `form` (`str`): the actual token
- `lemma` (`str`): the lemma corresponding to the actual token
- `upos` (`str`): the UD part-of-speech tag
- `xpos` (`str`): the Penn TreeBank part-of-speech tag
- any attribute found in the features column of the CoNLL-U

For information about the values `upos`, `xpos`, and the attributes contained in the features column can take on, see the UD Guidelines.

Each graph also has a special root node with identifier `ewt-SPLIT-SENTNUM-root-0`. This node always has a `position` attribute set to `0` and `domain` and `type` attributes set to `root`.

Edges within the graph represent the grammatical relations (dependencies) annotated in UD-EWT. These dependencies are always represented as directed edges pointing from the head to the dependent. At minimum, each edge has the following attributes.

- `domain` (`str`): the subgraph this node is part of (always `syntax`)

- `type` (`str`): the type of the object in the particular domain (always `dependency`)
- `deprel` (`str`): the UD dependency relation tag

For information about the values `deprel` can take on, see the UD Guidelines.

3.2 PredPatt Sentence Graphs

The semantic graphs that form the second layer of annotation in the dataset are produced by the PredPatt system. PredPatt takes as input a UD parse for a single sentence and produces a set of predicates and set of arguments of each predicate in that sentence. Both predicates and arguments are associated with a single head token in the sentence as well as a set of tokens that make up the predicate or argument (its span). Predicate or argument spans may be trivial in only containinig the head token.

For example, given the dependency parse for the sentence *Chris gave the book to Pat* ., PredPatt produces the following.

```
?a gave ?b to ?c
    ?a: Chris
    ?b: the book
    ?c: Pat
```

Assuming UD's 1-indexation, the single predicate in this sentence (*gave…to*) has a head at position 2 and a span over positions {2, 5}. This predicate has three arguments, one headed by *Chris* at position 1, with span over position {1}; one headed by *book* at position 4, with span over positions {3, 4}; and one headed by *Pat* at position 6, with span over position {6}.

See the PredPatt documentation tests for examples.

Each predicate and argument produced by PredPatt is associated with a node in a digraph with identifier `ewt-SPLIT-SENTNUM-semantics-TYPE-HEADTOKNUM`, where `TYPE` is always either `pred` or `arg` and `HEADTOKNUM` is the ordinal position of the head token within the sentence (1-indexed, following the convention in UD-EWT). At minimum, each such node has the following attributes.

- `domain` (`str`): the subgraph this node is part of (always `semantics`)
- `type` (`str`): the type of the object in the particular domain (either `predicate` or `argument`)
- `frompredpatt` (`bool`): whether this node is associated with a predicate or argument output by PredPatt (always `True`)

Predicate and argument nodes produced by PredPatt furthermore always have at least one outgoing *instance* edge that points to nodes in the syntax domain that correspond to the associated span of the predicate or argument. At minimum, each such edge has the following attributes.

- `domain` (`str`): the subgraph this node is part of (always `interface`)
- `type` (`str`): the type of the object in the particular domain (either `head` or `nonhead`)
- `frompredpatt` (`bool`): whether this node is associated with a predicate or argument output by PredPatt (always `True`)

Because PredPatt produces a unique head for each predicate and argument, there is always exactly one instance edge of type `head` from any particular node in the semantics domain. There may or may not be instance edges of type `nonhead`.

In addition to instance edges, predicate nodes always have exactly one outgoing edge connecting them to each of the nodes corresponding to their arguments. At minimum, each such edge has the following attributes.

- `domain` (`str`): the subgraph this node is part of (always `semantics`)
- `type` (`str`): the type of the object in the particular domain (always `dependency`)

- frompredpatt (bool): whether this node is associated with a predicate or argument output by PredPatt (always True)

There is one special case where an argument nodes has an outgoing edge that points to a predicate node: clausal subordination.

For example, given the dependency parse for the sentence *Gene thought that Chris gave the book to Pat .*, PredPatt produces the following.

```
?a thinks ?b
    ?a: Gene
    ?b: SOMETHING := that Chris gave the book to Pat

?a gave ?b to ?c
    ?a: Chris
    ?b: the book
    ?c: Pat
```

In this case, the second argument of the predicate headed by *thinks* is the argument *that Chris gave the book to Pat*, which is headed by *gave*. This argument is associated with a node of type argument with span over positions {3, 4, 5, 6, 7, 8, 9} and identifier ewt-SPLIT-SENTNUM-semantics-arg-5. In addition, there is a predicate headed by *gave*. This predicate is associated with a node with span over positions {5, 8} and identifier ewt-SPLIT-SENTNUM-semantics-pred-5. Node ewt-SPLIT-SENTNUM-semantics-arg-5 then has an outgoing edge pointing to ewt-SPLIT-SENTNUM-semantics-pred-5. At minimum, each such edge has the following attributes.

- domain (str): the subgraph this node is part of (always semantics)
- type (str): the type of the object in the particular domain (always head)
- frompredpatt (bool): whether this node is associated with a predicate or argument output by PredPatt (always True)

The type attribute in this case has the same value as instance edges, but crucially the domain attribute is distinct. In the case of instance edges, it is interface and in the case of clausal subordination, it is semantics. This matters when making queries against the graph.

If the frompredpatt attribute has value True, it is guaranteed that the only semantics edges of type head are ones that involve clausal subordination like the above. This is not guaranteed for nodes for which the frompredpatt attribute has value False.

Every semantic graph contains at least four additional *performative* nodes that are note produced by PredPatt (and thus, for which the frompredpatt attribute has value False).

- ewt-SPLIT-SENTNUM-semantics-arg-0: an argument node representing the entire sentence in the same way complement clauses are represented
- ewt-SPLIT-SENTNUM-semantics-pred-root: a predicate node representing the author's production of the entire sentence directed at the addressee
- ewt-SPLIT-SENTNUM-semantics-arg-speaker: an argument node representing the author
- ewt-SPLIT-SENTNUM-semantics-arg-addressee: an argument node representing the addressee

All of these nodes have a domain attribute with value semantics. Unlike nodes associated with PredPatt predicates and arguments, ewt-SPLIT-SENTNUM-semantics-pred-root, ewt-SPLIT-SENTNUM-semantics-arg-speaker, and ewt-SPLIT-SENTNUM-semantics-arg-addressee have no instance edges connecting them to syntactic nodes. In contrast, ewt-SPLIT-SENTNUM-semantics-arg-0 has an instance head edge to ewt-SPLIT-SENTNUM-root-0.

The ewt-SPLIT-SENTNUM-semantics-arg-0 node has semantics head edges to each of the predicate nodes in the graph that are not dominated by any other semantics node. This node, in addition to

ewt-SPLIT-SENTNUM-semantics-arg-speaker and ewt-SPLIT-SENTNUM-semantics-arg-addressee, has a dependency edge to ewt-SPLIT-SENTNUM-semantics-pred-root.

These nodes are included for purposes of forward compatibility. None of them currently have attributes, but future releases of decomp will include annotations on either them or their edges.

3.3 Universal Decompositional Document Graphs

The semantic graphs that form the third layer of annotation represent document-level relations. These graphs contain a node for each node in the document's constituent sentence-level graphs along with a pointer from the document-level node to the sentence-level node. Unlike the sentence-level graphs, they are not produced by PredPatt, so whether any two nodes in a document-level graph are joined by an edge is determined by whether the relation between the two nodes is annotated in some UDS dataset.

At minimum, each of these nodes has the following attributes:

- domain (str): the subgraph this node is part of (always document)
- type (str): the type of object corresponding to this node in the semantics domain (either predicate or argument)
- frompredpatt (bool): whether this node is associated with a predicate or argument output by PredPatt (always False, although the corresponding semantics node will have this set as True)
- semantics (dict): a two-item dictionary containing information about the corresponding semantics node. The first item, graph, indicates the sentence-level graph that the semantics node comes from. The second item, node, contains the name of the node.

Document graphs are initialized without edges, which are created dynamically when edge attribute annotations are added. These edges may span nodes associated with different sentences within a document and may connect not only predicates to arguments, but predicates to predicates and arguments to arguments. Any annotations that are provided that cross document boundaries will be automatically filtered out. Finally, beyond the attributes provided by annotations, each edge will also contain all but the last of the core set of node attributes listed above.

The UDSDocumentGraph object is wrapped by a UDSDocument, which holds additional metadata associated with the document, data relating to its constituent sentences (and their graphs), and methods for interacting with it. Finally, it should be noted that querying on document graphs is not currently supported.

3.4 Universal Decompositional Semantic Types

PredPatt makes very coarse-grained typing distinctions—between predicate and argument nodes, on the one hand, and between dependency and head edges, on the other. UDS provides ultra fine-grained typing distinctions, represented as collections of real-valued attributes. The union of all node and edge attributes defined in UDS determines the *UDS type space*; any proper subset determines a *UDS type subspace*.

UDS attributes are derived from crowd-sourced annotations of the heads or spans corresponding to predicates and/or arguments and are represented in the dataset as node and/or edge attributes. It is important to note that, though all nodes and edges in the semantics domain have a type attribute, UDS does not afford any special status to these types. That is, the only thing that UDS "sees" are the nodes and edges in the semantics domain. The set of nodes and edges visible to UDS is a superset of those associated with PredPatt predicates and their arguments.

There are currently four node type subspaces annotated on nodes in sentence-level graphs.

- *Factuality* (factuality)
- *Genericity* (genericity)

- *Time* (`time`)
- *Entity type* (`wordsense`)
- *Event structure* (`event_structure`)

There is currently one edge type subspace annotated on edges in sentence-level graphs.

- Semantic Proto-Roles (`protoroles`)
- *Event structure* (`event_structure`)

There is currently (starting in UDS2.0) one edge type subspace annotated on edges in document-level graphs.

- *Time* (`time`)
- *Event structure* (`event_structure`)

Each subspace key lies at the same level as the `type` attribute and maps to a dictionary value. This dictionary maps from attribute keys (see *Attributes* in each section below) to dictionaries that always have two keys `value` and `confidence`. See the below paper for information on how the these are derived from the underlying dataset.

Two versions of these annotations are currently available: one containing the raw annotator data (`"raw"`) and the other containing normalized data (`"normalized"`). In the former case, both the `value` and `confidence` fields described above map to dictionaries keyed on (anonymized) annotator IDs, where the corresponding value contains that annotator's response (for the `value` dictionary) or confidence (for the `confidence` dictionary). In the latter case, the `value` and `confidence` fields map to single, normalized value and confidence scores, respectively.

For more information on the normalization used to produce the normalized annotations, see:

> White, Aaron Steven, Elias Stengel-Eskin, Siddharth Vashishtha, Venkata Subrahmanyan Govindarajan, Dee Ann Reisinger, Tim Vieira, Keisuke Sakaguchi, et al. 2020. The Universal Decompositional Semantics Dataset and Decomp Toolkit. *Proceedings of The 12th Language Resources and Evaluation Conference*, 5698–5707. Marseille, France: European Language Resources Association.

```
@inproceedings{white-etal-2020-universal,
    title = "The Universal Decompositional Semantics Dataset and Decomp Toolkit",
    author = "White, Aaron Steven  and
      Stengel-Eskin, Elias  and
      Vashishtha, Siddharth  and
      Govindarajan, Venkata Subrahmanyan  and
      Reisinger, Dee Ann  and
      Vieira, Tim  and
      Sakaguchi, Keisuke  and
      Zhang, Sheng  and
      Ferraro, Francis  and
      Rudinger, Rachel  and
      Rawlins, Kyle  and
      Van Durme, Benjamin",
    booktitle = "Proceedings of The 12th Language Resources and Evaluation Conference",
    month = may,
    year = "2020",
    address = "Marseille, France",
    publisher = "European Language Resources Association",
    url = "https://www.aclweb.org/anthology/2020.lrec-1.699",
    pages = "5698--5707",
    ISBN = "979-10-95546-34-4",
}
```

Information about each subspace can be found below. Unless otherwise specified the properties in a particular subspace remain constant across the raw and normalized formats.

3.4.1 Factuality

Project page

http://decomp.io/projects/factuality/

Sentence-level attributes

`factual`

First UDS version

1.0

References

White, A.S., D. Reisinger, K. Sakaguchi, T. Vieira, S. Zhang, R. Rudinger, K. Rawlins, & B. Van Durme. 2016. Universal Decompositional Semantics on Universal Dependencies. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1713–1723, Austin, Texas, November 1-5, 2016.

Rudinger, R., White, A.S., & B. Van Durme. 2018. Neural models of factuality. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 731–744. New Orleans, Louisiana, June 1-6, 2018.

```
@inproceedings{white-etal-2016-universal,
    title = "Universal Decompositional Semantics on {U}niversal {D}ependencies",
    author = "White, Aaron Steven  and
      Reisinger, Dee Ann  and
      Sakaguchi, Keisuke  and
      Vieira, Tim  and
      Zhang, Sheng  and
      Rudinger, Rachel  and
      Rawlins, Kyle  and
      Van Durme, Benjamin",
    booktitle = "Proceedings of the 2016 Conference on Empirical Methods in Natural␣
↪Language Processing",
    month = nov,
    year = "2016",
    address = "Austin, Texas",
    publisher = "Association for Computational Linguistics",
    url = "https://www.aclweb.org/anthology/D16-1177",
    doi = "10.18653/v1/D16-1177",
    pages = "1713--1723",
}

@inproceedings{rudinger-etal-2018-neural-models,
    title = "Neural Models of Factuality",
    author = "Rudinger, Rachel  and
      White, Aaron Steven  and
      Van Durme, Benjamin",
    booktitle = "Proceedings of the 2018 Conference of the North {A}merican Chapter of␣
↪the Association for Computational Linguistics: Human Language Technologies, Volume 1␣
↪(Long Papers)",
```

(continues on next page)

(continued from previous page)

```
    month = jun,
    year = "2018",
    address = "New Orleans, Louisiana",
    publisher = "Association for Computational Linguistics",
    url = "https://www.aclweb.org/anthology/N18-1067",
    doi = "10.18653/v1/N18-1067",
    pages = "731--744",
}
```

3.4.2 Genericity

Project page

http://decomp.io/projects/genericity/

Sentence-level attributes

`arg-particular`, `arg-kind`, `arg-abstract`, `pred-particular`, `pred-dynamic`, `pred-hypothetical`

First UDS version

1.0

References

Govindarajan, V.S., B. Van Durme, & A.S. White. 2019. Decomposing Generalization: Models of Generic, Habitual, and Episodic Statements. Transactions of the Association for Computational Linguistics.

```
@article{govindarajan-etal-2019-decomposing,
    title = "Decomposing Generalization: Models of Generic, Habitual, and Episodic␣
↪Statements",
    author = "Govindarajan, Venkata  and
      Van Durme, Benjamin  and
      White, Aaron Steven",
    journal = "Transactions of the Association for Computational Linguistics",
    volume = "7",
    month = mar,
    year = "2019",
    url = "https://www.aclweb.org/anthology/Q19-1035",
    doi = "10.1162/tacl_a_00285",
    pages = "501--517"
}
```

3.4.3 Time

Project page

http://decomp.io/projects/time/

Sentence-level attributes

normalized

`dur-hours`, `dur-instant`, `dur-forever`, `dur-weeks`, `dur-days`, `dur-months`, `dur-years`, `dur-centuries`, `dur-seconds`, `dur-minutes`, `dur-decades`

raw

`duration`

Document-level attributes

raw

`rel-start1`, `rel-start2`, `rel-end1`, `rel-end2`

First UDS version

1.0 (sentence-level), 2.0 (document-level)

References

Vashishtha, S., B. Van Durme, & A.S. White. 2019. Fine-Grained Temporal Relation Extraction. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL 2019)*, 2906—2919. Florence, Italy, July 29-31, 2019.

```
@inproceedings{vashishtha-etal-2019-fine,
    title = "Fine-Grained Temporal Relation Extraction",
    author = "Vashishtha, Siddharth  and
      Van Durme, Benjamin  and
      White, Aaron Steven",
    booktitle = "Proceedings of the 57th Annual Meeting of the Association for␣
↪Computational Linguistics",
    month = jul,
    year = "2019",
    address = "Florence, Italy",
    publisher = "Association for Computational Linguistics",
    url = "https://www.aclweb.org/anthology/P19-1280",
    doi = "10.18653/v1/P19-1280",
    pages = "2906--2919"
}
```

Notes

1. The Time dataset has different formats for raw and normalized annotations. The duration attributes from the normalized version are each assigned an ordinal value in the raw version (in ascending order of duration), which is assigned to the single attribute `duration`.
2. The document-level relation annotations are *only* available in the raw format and only starting in UDS2.0.

3.4.4 Entity type

Project page

http://decomp.io/projects/word-sense/

Sentence-level attributes

`supersense-noun.shape`, `supersense-noun.process`, `supersense-noun.relation`, `supersense-noun.communication`, `supersense-noun.time`, `supersense-noun.plant`, `supersense-noun.phenomenon`, `supersense-noun.animal`, `supersense-noun.state`, `supersense-noun.substance`, `supersense-noun.person`, `supersense-noun.possession`, `supersense-noun.Tops`, `supersense-noun.object`, `supersense-noun.event`, `supersense-noun.artifact`, `supersense-noun.act`, `supersense-noun.body`, `supersense-noun.attribute`, `supersense-noun.quantity`, `supersense-noun.motive`, `supersense-noun.location`, `supersense-noun.cognition`, `supersense-noun.group`, `supersense-noun.food`, `supersense-noun.feeling`

First UDS version

1.0

Notes

1. The key is called `wordsense` because the normalized annotations come from UDS-Word Sense (v1.0).

References

White, A.S., D. Reisinger, K. Sakaguchi, T. Vieira, S. Zhang, R. Rudinger, K. Rawlins, & B. Van Durme. 2016. Universal Decompositional Semantics on Universal Dependencies. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1713–1723, Austin, Texas, November 1-5, 2016.

```
@inproceedings{white-etal-2016-universal,
    title = "Universal Decompositional Semantics on {U}niversal {D}ependencies",
    author = "White, Aaron Steven  and
      Reisinger, Dee Ann  and
      Sakaguchi, Keisuke  and
      Vieira, Tim  and
      Zhang, Sheng  and
      Rudinger, Rachel  and
      Rawlins, Kyle  and
      Van Durme, Benjamin",
    booktitle = "Proceedings of the 2016 Conference on Empirical Methods in Natural␣
↪Language Processing",
    month = nov,
    year = "2016",
    address = "Austin, Texas",
    publisher = "Association for Computational Linguistics",
    url = "https://www.aclweb.org/anthology/D16-1177",
    doi = "10.18653/v1/D16-1177",
    pages = "1713--1723",
}
```

3.4.5 Semantic Proto-Roles

Project page

http://decomp.io/projects/semantic-proto-roles/

Sentence-level attributes

`was_used`, `purpose`, `partitive`, `location`, `instigation`, `existed_after`, `time`, `awareness`, `change_of_location`, `manner`, `sentient`, `was_for_benefit`, `change_of_state_continuous`, `existed_during`, `change_of_possession`, `existed_before`, `volition`, `change_of_state`

References

Reisinger, D., R. Rudinger, F. Ferraro, C. Harman, K. Rawlins, & B. Van Durme. (2015). Semantic Proto-Roles. *Transactions of the Association for Computational Linguistics 3*:475–488.

White, A.S., D. Reisinger, K. Sakaguchi, T. Vieira, S. Zhang, R. Rudinger, K. Rawlins, & B. Van Durme. 2016. Universal Decompositional Semantics on Universal Dependencies. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1713–1723, Austin, Texas, November 1-5, 2016.

```
@article{reisinger-etal-2015-semantic,
    title = "Semantic Proto-Roles",
    author = "Reisinger, Dee Ann  and
      Rudinger, Rachel  and
      Ferraro, Francis  and
      Harman, Craig  and
      Rawlins, Kyle  and
      Van Durme, Benjamin",
    journal = "Transactions of the Association for Computational Linguistics",
    volume = "3",
    year = "2015",
    url = "https://www.aclweb.org/anthology/Q15-1034",
    doi = "10.1162/tacl_a_00152",
    pages = "475--488",
}

@inproceedings{white-etal-2016-universal,
    title = "Universal Decompositional Semantics on {U}niversal {D}ependencies",
    author = "White, Aaron Steven  and
      Reisinger, Dee Ann  and
      Sakaguchi, Keisuke  and
      Vieira, Tim  and
      Zhang, Sheng  and
      Rudinger, Rachel  and
      Rawlins, Kyle  and
      Van Durme, Benjamin",
    booktitle = "Proceedings of the 2016 Conference on Empirical Methods in Natural␣
↪Language Processing",
    month = nov,
    year = "2016",
    address = "Austin, Texas",
    publisher = "Association for Computational Linguistics",
    url = "https://www.aclweb.org/anthology/D16-1177",
    doi = "10.18653/v1/D16-1177",
    pages = "1713--1723",
}
```

3.4.6 Event structure

Project page

http://decomp.io/projects/event-structure/

Sentence-level attributes

normalized

distributive, dynamic, natural_parts, part_similarity, telic, avg_part_duration_lbound-centuries, avg_part_duration_ubound-centuries, situation_duration_lbound-centuries, situation_duration_ubound-centuries, avg_part_duration_lbound-days, avg_part_duration_ubound-days, situation_duration_lbound-days, situation_duration_ubound-days, avg_part_duration_lbound-decades, avg_part_duration_ubound-decades, situation_duration_lbound-decades, situation_duration_ubound-decades, avg_part_duration_lbound-forever, avg_part_duration_ubound-forever,

situation_duration_lbound-forever, situation_duration_ubound-forever, avg_part_duration_lbound-fractions_of_a_second, avg_part_duration_ubound-fractions_of_a_second, situation_duration_lbound-fractions_of_a_second, situation_duration_ubound-fractions_of_a_second, avg_part_duration_lbound-hours, avg_part_duration_ubound-hours, situation_duration_lbound-hours, situation_duration_ubound-hours, avg_part_duration_lbound-instant, avg_part_duration_ubound-instant, situation_duration_lbound-instant, situation_duration_ubound-instant, avg_part_duration_lbound-minutes, avg_part_duration_ubound-minutes, situation_duration_lbound-minutes, situation_duration_ubound-minutes, avg_part_duration_lbound-months, avg_part_duration_ubound-months, situation_duration_lbound-months, situation_duration_ubound-months, avg_part_duration_lbound-seconds, avg_part_duration_ubound-seconds, situation_duration_lbound-seconds, situation_duration_ubound-seconds, avg_part_duration_lbound-weeks, avg_part_duration_ubound-weeks, situation_duration_lbound-weeks, situation_duration_ubound-weeks, avg_part_duration_lbound-years, avg_part_duration_ubound-years, situation_duration_lbound-years, situation_duration_ubound-years

raw

dynamic, natural_parts, part_similarity, telic, avg_part_duration_lbound, avg_part_duration_ubound, situation_duration_lbound, situation_duration_ubound

Document-level attributes

pred1_contains_pred2, pred2_contains_pred1

First UDS version

2.0

Notes

1. Whether dynamic, situation_duration_lbound, and situation_duration_ubound are answered or part_similarity, avg_part_duration_lbound, and avg_part_duration_ubound are answered is dependent on the answer an annotator gives to natural_parts. Thus, not all node attributes will necessarily be present on all nodes.

References

Gantt, W., L. Glass, & A.S. White. 2021. Decomposing and Recomposing Event Structure. arXiv:2103.10387 [cs.CL].

```
@misc{gantt2021decomposing,
    title={Decomposing and Recomposing Event Structure},
    author={William Gantt and Lelia Glass and Aaron Steven White},
    year={2021},
    eprint={2103.10387},
    archivePrefix={arXiv},
    primaryClass={cs.CL}
}
```

Each layer contains pointers directly to the previous layer.

CHAPTER

FOUR

PACKAGE REFERENCE

4.1 decomp.syntax

Module for representing CoNLL dependency tree corpora

This module provides readers for corpora represented using conll-formatted dependency parses. All dependency parses are read in as networkx graphs. These graphs become subgraphs of the PredPatt and UDS graphs in the semantics module.

4.1.1 decomp.syntax.dependency

Module for building/containing dependency trees from CoNLL

class decomp.syntax.dependency.**CoNLLDependencyTreeCorpus**(*graphs_raw*)

Class for building/containing dependency trees from CoNLL-U

graphs

trees constructed from annotated sentences

graphids

ids for trees constructed from annotated sentences

ngraphs

number of graphs in corpus

class decomp.syntax.dependency.**DependencyGraphBuilder**

A dependency graph builder

classmethod from_conll(*conll*, *treeid=''*, *spec='u'*)

Build DiGraph from a CoNLL representation

Parameters

- **conll** (List[List[str]]) – conll representation
- **treeid** (str) – a unique identifier for the tree
- **spec** (str) – the specification to assume of the conll representation (“u” or “x”)

Return type
DiGraph

4.2 decomp.semantics

Module for representing PredPatt and UDS graphs

This module represents PredPatt and UDS graphs using networkx. It incorporates the dependency parse-based graphs from the syntax module as subgraphs.

4.2.1 decomp.semantics.predpatt

Module for converting PredPatt objects to networkx digraphs

class decomp.semantics.predpatt.**PredPattCorpus**(*graphs_raw*)

Container for predpatt graphs

classmethod from_conll(*corpus*, *name='ewt'*, *options=None*)

Load a CoNLL dependency corpus and apply predpatt

Parameters

- **corpus** (Union[str, TextIO]) – (path to) a .conllu file
- **name** (str) – the name of the corpus; used in constructing treeids
- **options** (Optional[PredPattOpts]) – options for predpatt extraction

Return type
PredPattCorpus

class decomp.semantics.predpatt.**PredPattGraphBuilder**

A predpatt graph builder

classmethod from_predpatt(*predpatt*, *depgraph*, *graphid=''*)

Build a DiGraph from a PredPatt object and another DiGraph

Parameters

- **predpatt** (PredPatt) – the predpatt extraction for the dependency parse
- **depgraph** (DiGraph) – the dependency graph
- **graphid** (str) – the tree indentifier; will be a prefix of all node identifiers

Return type
DiGraph

4.2.2 decomp.semantics.uds

Module for representing UDS corpora, documents, graphs, and annotations.

decomp.semantics.uds.corpus

Module for representing UDS corpora.

class decomp.semantics.uds.corpus.**UDSCorpus**(*sentences=None*, *documents=None*, *sentence_annotations=[]*, *document_annotations=[]*, *version='2.0'*, *split=None*, *annotation_format='normalized'*)

A collection of Universal Decompositional Semantics graphs

Parameters

- **sentences** (Optional[*PredPattCorpus*]) – the predpatt sentence graphs to associate the annotations with
- **documents** (Optional[Dict[str, *UDSDocument*]]) – the documents associated with the predpatt sentence graphs
- **sentence_annotations** (List[*UDSAnnotation*]) – additional annotations to associate with predpatt nodes on sentence-level graphs; in most cases, no such annotations will be passed, since the standard UDS annotations are automatically loaded
- **document_annotations** (List[*UDSAnnotation*]) – additional annotations to associate with predpatt nodes on document-level graphs
- **version** (str) – the version of UDS datasets to use
- **split** (Optional[str]) – the split to load: "train", "dev", or "test"
- **annotation_format** (str) – which annotation type to load ("raw" or "normalized")

add_annotation(*sentence_annotation*, *document_annotation*)

Add annotations to UDS sentence and document graphs

Parameters

- **sentence_annotation** (*UDSAnnotation*) – the annotations to add to the sentence graphs in the corpus
- **document_annotation** (*UDSAnnotation*) – the annotations to add to the document graphs in the corpus

Return type
None

add_document_annotation(*annotation*)

Add annotations to UDS documents

Parameters
annotation (*UDSAnnotation*) – the annotations to add to the documents in the corpus

Return type
None

add_sentence_annotation(*annotation*)

Add annotations to UDS sentence graphs

Parameters
annotation (*UDSAnnotation*) – the annotations to add to the graphs in the corpus

Return type
None

property document_edge_subspaces: Set[str]

The UDS document edge subspaces in the corpus

Return type
Set[str]

property document_node_subspaces: Set[str]

The UDS document node subspaces in the corpus

Return type
Set[str]

document_properties(*subspace=None*)

The properties in a document subspace

Return type
Set[str]

document_property_metadata(*subspace*, *prop*)

The metadata for a property in a document subspace

Parameters

- **subspace** (str) – The subspace the property is in
- **prop** (str) – The property in the subspace

Return type
UDSPropertyMetadata

property document_subspaces: Set[str]

The UDS document subspaces in the corpus

Return type
Set[str]

property documentids

The document ID for each document in the corpus

property documents: Dict[str, *UDSDocument*]

The documents in the corpus

Return type
Dict[str, *UDSDocument*]

classmethod from_conll(*corpus*, *sentence_annotations=[]*, *document_annotations=[]*, *annotation_format='normalized'*, *version='2.0'*, *name='ewt'*)

Load UDS graph corpus from CoNLL (dependencies) and JSON (annotations)

This method should only be used if the UDS corpus is being (re)built. Otherwise, loading the corpus from the JSON shipped with this package using UDSCorpus.__init__ or UDSCorpus.from_json is suggested.

Parameters

- **corpus** (Union[str, TextIO]) – (path to) Universal Dependencies corpus in conllu format
- **sentence_annotations** (List[Union[str, TextIO]]) – a list of paths to JSON files or open JSON files containing sentence-level annotations
- **document_annotations** (List[Union[str, TextIO]]) – a list of paths to JSON files or open JSON files containing document-level annotations
- **annotation_format** (str) – Whether the annotation is raw or normalized

- **version** (str) – the version of UDS datasets to use
- **name** (str) – corpus name to be appended to the beginning of graph ids

Return type
UDSCorpus

classmethod from_json(*sentences_jsonfile*, *documents_jsonfile*)

Load annotated UDS graph corpus (including annotations) from JSON

This is the suggested method for loading the UDS corpus.

Parameters

- **sentences_jsonfile** (Union[str, TextIO]) – file containing Universal Decompositional Semantics corpus sentence-level graphs in JSON format
- **documents_jsonfile** (Union[str, TextIO]) – file containing Universal Decompositional Semantics corpus document-level graphs in JSON format

Return type
UDSCorpus

property ndocuments

The number of IDs in the corpus

query(*query*, *query_type=None*, *cache_query=True*, *cache_rdf=True*)

Query all graphs in the corpus using SPARQL 1.1

Parameters

- **query** (Union[str, Query]) – a SPARQL 1.1 query
- **query_type** (Optional[str]) – whether this is a 'node' query or 'edge' query. If set to None (default), a Results object will be returned. The main reason to use this option is to automatically format the output of a custom query, since Results objects require additional postprocessing.
- **cache_query** (bool) – whether to cache the query. This should usually be set to True. It should generally only be False when querying particular nodes or edges–e.g. as in precompiled queries.
- **clear_rdf** – whether to delete the RDF constructed for querying against. This will slow down future queries but saves a lot of memory

Return type
Union[Result, Dict[str, Dict[str, Any]]]

sample_documents(*k*)

Sample k documents without replacement

Parameters
k (int) – the number of documents to sample

Return type
Dict[str, *UDSDocument*]

property sentence_edge_subspaces: Set[str]

The UDS sentence edge subspaces in the corpus

Return type
Set[str]

property sentence_node_subspaces: Set[str]

The UDS sentence node subspaces in the corpus

Return type
Set[str]

sentence_properties(*subspace=None*)

The properties in a sentence subspace

Return type
Set[str]

sentence_property_metadata(*subspace*, *prop*)

The metadata for a property in a sentence subspace

Parameters

- **subspace** (str) – The subspace the property is in
- **prop** (str) – The property in the subspace

Return type
UDSPropertyMetadata

property sentence_subspaces: Set[str]

The UDS sentence subspaces in the corpus

Return type
Set[str]

to_json(*sentences_outfile=None*, *documents_outfile=None*)

Serialize corpus to json

Parameters

- **sentences_outfile** (Union[str, TextIO, None]) – file to serialize sentence-level graphs to
- **documents_outfile** (Union[str, TextIO, None]) – file to serialize document-level graphs to

Return type
Optional[str]

decomp.semantics.uds.document

Module for representing UDS documents.

class decomp.semantics.uds.document.**UDSDocument**(*sentence_graphs*, *sentence_ids*, *name*, *genre*, *timestamp=None*, *doc_graph=None*)

A Universal Decompositional Semantics document

Parameters

- **sentence_graphs** (Dict[str, *UDSSentenceGraph*]) – the UDSSentenceGraphs associated with each sentence in the document
- **sentence_ids** (Dict[str, str]) – the UD sentence IDs for each graph
- **name** (str) – the name of the document (i.e. the UD document ID)
- **genre** (str) – the genre of the document (e.g. *weblog*)

- **timestamp** (Optional[str]) – the timestamp of the UD document on which this UDSDocument is based
- **doc_graph** (Optional[*UDSDocumentGraph*]) – the NetworkX DiGraph for the document. If not provided, this will be initialized without edges from sentence_graphs

add_annotation(*node_attrs*, *edge_attrs*)

Add node or edge annotations to the document-level graph

Parameters

- **node_attrs** (Dict[str, Dict[str, Any]]) – the node annotations to be added
- **edge_attrs** (Dict[str, Dict[str, Any]]) – the edge annotations to be added

Return type
None

add_sentence_graphs(*sentence_graphs*, *sentence_ids*)

Add additional sentences to a document

Parameters

- **sentence_graphs** (Dict[str, *UDSSentenceGraph*]) – a dictionary containing the sentence-level graphs for the sentences in the document
- **sentence_ids** (Dict[str, str]) – a dictionary containing the UD sentence IDs for each graph
- **name** – identifier to append to the beginning of node ids

Return type
None

classmethod from_dict(*document*, *sentence_graphs*, *sentence_ids*, *name='UDS'*)

Construct a UDSDocument from a dictionary

Since only the document graphs are serialized, the sentence graphs must also be provided to this method call in order to properly associate them with their documents.

Parameters

- **document** (Dict[str, Dict]) – a dictionary constructed by networkx.adjacency_data, containing the graph for the document
- **sentence_graphs** (Dict[str, *UDSSentenceGraph*]) – a dictionary containing (possibly a superset of) the sentence-level graphs for the sentences in the document
- **sentence_ids** (Dict[str, str]) – a dictionary containing (possibly a superset of) the UD sentence IDs for each graph
- **name** (str) – identifier to append to the beginning of node ids

Return type
UDSDocument

semantics_node(*document_node*)

The semantics node for a given document node

Parameters
document_node (str) – the document domain node whose semantics node is to be retrieved

Return type
Dict[str, Dict]

property text: str

The document text

Return type
str

to_dict()

Convert the graph to a dictionary

Return type
Dict

decomp.semantics.uds.graph

Module for representing UDS sentence and document graphs.

class decomp.semantics.uds.graph.**UDSDocumentGraph**(*graph*, *name*)

A Universal Decompositional Semantics document-level graph

Parameters

- **graph** (DiGraph) – the NetworkX DiGraph from which the document-level graph is to be constructed
- **name** (str) – the name of the graph

add_annotation(*node_attrs*, *edge_attrs*, *sentence_ids*)

Add node and or edge annotations to the graph

Parameters

- **node_attrs** (Dict[str, Dict[str, Any]]) – the node annotations to be added
- **edge_attrs** (Dict[str, Dict[str, Any]]) – the edge annotations to be added
- **sentence_ids** (Dict[str, str]) – the IDs of all sentences in the document

Return type
None

class decomp.semantics.uds.graph.**UDSGraph**(*graph*, *name*)

Abstract base class for sentence- and document-level graphs

Parameters

- **graph** (DiGraph) – a NetworkX DiGraph
- **name** (str) – a unique identifier for the graph

property edges

All the edges in the graph

classmethod from_dict(*graph*, *name='UDS'*)

Construct a UDSGraph from a dictionary

Parameters

- **graph** (Dict[str, Any]) – a dictionary constructed by networkx.adjacency_data
- **name** (str) – identifier to append to the beginning of node ids

Return type
UDSGraph

property nodes

All the nodes in the graph

to_dict()

Convert the graph to a dictionary

Return type
Dict

class decomp.semantics.uds.graph.**UDSSentenceGraph**(*graph*, *name*, *sentence_id=None*, *document_id=None*)

A Universal Decompositional Semantics sentence-level graph

Parameters

- **graph** (DiGraph) – the NetworkX DiGraph from which the sentence-level graph is to be constructed
- **name** (str) – the name of the graph
- **sentence_id** (Optional[str]) – the UD identifier for the sentence associated with this graph
- **document_id** (Optional[str]) – the UD identifier for the document associated with this graph

add_annotation(*node_attrs*, *edge_attrs*, *add_heads=True*, *add_subargs=False*, *add_subpreds=False*, *add_orphans=False*)

Add node and or edge annotations to the graph

Parameters

- **node_attrs** (Dict[str, Dict[str, Any]]) –
- **edge_attrs** (Dict[str, Dict[str, Any]]) –
- **add_heads** (bool) –
- **add_subargs** (bool) –
- **add_subpreds** (bool) –
- **add_orphans** (bool) –

Return type
None

argument_edges(*nodeid=None*)

The edges between predicates and their arguments

Parameters
nodeid (Optional[str]) – The node that must be incident on an edge

Return type
Dict[Tuple[str, str], Dict[str, Any]]

argument_head_edges(*nodeid=None*)

The edges between nodes and their semantic heads

Parameters
nodeid (Optional[str]) – The node that must be incident on an edge

Return type
Dict[Tuple[str, str], Dict[str, Any]]

property argument_nodes: Dict[str, Dict[str, Any]]

The argument (semantics) nodes in the graph

Return type
Dict[str, Dict[str, Any]]

head(*nodeid*, *attrs=['form']*)

The head corresponding to a semantics node

Parameters

- **nodeid** (str) – the node identifier for a semantics node
- **attrs** (List[str]) – a list of syntax node attributes to return

Return type
Tuple[int, List[Any]]

Returns

- *a pairing of the head position and the requested*
- *attributes*

instance_edges(*nodeid=None*)

The edges between syntax nodes and semantics nodes

Parameters
nodeid (Optional[str]) – The node that must be incident on an edge

Return type
Dict[Tuple[str, str], Dict[str, Any]]

maxima(*nodeids=None*)

The nodes in nodeids not dominated by any other nodes in nodeids

Return type
List[str]

minima(*nodeids=None*)

The nodes in nodeids not dominating any other nodes in nodeids

Return type
List[str]

property predicate_nodes: Dict[str, Dict[str, Any]]

The predicate (semantics) nodes in the graph

Return type
Dict[str, Dict[str, Any]]

query(*query*, *query_type=None*, *cache_query=True*, *cache_rdf=True*)

Query graph using SPARQL 1.1

Parameters

- **query** (Union[str, Query]) – a SPARQL 1.1 query
- **query_type** (Optional[str]) – whether this is a 'node' query or 'edge' query. If set to None (default), a Results object will be returned. The main reason to use this option is to automatically format the output of a custom query, since Results objects require additional postprocessing.

- **cache_query** (bool) – whether to cache the query; false when querying particular nodes or edges using precompiled queries
- **clear_rdf** – whether to delete the RDF constructed for querying against. This will slow down future queries but saves a lot of memory

Return type
Union[Result, Dict[str, Dict[str, Any]]]

property rdf: Graph

The graph as RDF

Return type
Graph

property rootid

The ID of the graph's root node

semantics_edges(*nodeid=None*, *edgetype=None*)

The edges between semantics nodes

Parameters

- **nodeid** (Optional[str]) – The node that must be incident on an edge
- **edgetype** (Optional[str]) – The type of edge ("dependency" or "head")

Return type
Dict[Tuple[str, str], Dict[str, Any]]

property semantics_nodes: Dict[str, Dict[str, Any]]

The semantics nodes in the graph

Return type
Dict[str, Dict[str, Any]]

property semantics_subgraph: DiGraph

The part of the graph with only semantics nodes

Return type
DiGraph

property sentence: str

The sentence annotated by this graph

Return type
str

span(*nodeid*, *attrs=['form']*)

The span corresponding to a semantics node

Parameters

- **nodeid** (str) – the node identifier for a semantics node
- **attrs** (List[str]) – a list of syntax node attributes to return

Return type
Dict[int, List[Any]]

Returns

- *a mapping from positions in the span to the requested*

- *attributes in those positions*

syntax_edges(*nodeid=None*)

The edges between syntax nodes

Parameters
nodeid (Optional[str]) – The node that must be incident on an edge

Return type
Dict[Tuple[str, str], Dict[str, Any]]

property syntax_nodes: Dict[str, Dict[str, Any]]

The syntax nodes in the graph

Return type
Dict[str, Dict[str, Any]]

property syntax_subgraph: DiGraph

The part of the graph with only syntax nodes

Return type
DiGraph

decomp.semantics.uds.annotation

Module for representing UDS property annotations.

class decomp.semantics.uds.annotation.**NormalizedUDSAnnotation**(*metadata*, *data*)

A normalized Universal Decompositional Semantics annotation

Properties in a NormalizedUDSAnnotation may have only a single str, int, or float value and a single str, int, or float confidence.

Parameters

- **metadata** (*UDSAnnotationMetadata*) – The metadata for the annotations
- **data** (Dict[str, Dict[str, Dict[str, Dict[str, Dict[str, Union[str, int, bool, float]]]]]]) – A mapping from graph identifiers to node/edge identifiers to property subspaces to property to value and confidence. Edge identifiers must be represented as NODEID1%%NODEID2, and node identifiers must not contain %%.

classmethod from_json(*jsonfile*)

Generates a dataset of normalized annotations from a JSON file

For node annotations, the format of the JSON passed to this class method must be:

```
{GRAPHID_1: {NODEID_1_1: DATA,
             ...},
 GRAPHID_2: {NODEID_2_1: DATA,
             ...},
 ...
}
```

Edge annotations should be of the form:

```
{GRAPHID_1: {NODEID_1_1%%NODEID_1_2: DATA,
             ...},
 GRAPHID_2: {NODEID_2_1%%NODEID_2_2: DATA,
```

(continues on next page)

(continued from previous page)

```
              ...},
 ...
}
```

Graph and node identifiers must match the graph and node identifiers of the predpatt graphs to which the annotations will be added.

DATA in the above is assumed to have the following structure:

```
{SUBSPACE_1: {PROP_1_1: {'value': VALUE,
                         'confidence': VALUE},
              ...},
 SUBSPACE_2: {PROP_2_1: {'value': VALUE,
                         'confidence': VALUE},
              ...},
}
```

VALUE in the above is assumed to be unstructured.

Return type
NormalizedUDSAnnotation

class decomp.semantics.uds.annotation.**RawUDSAnnotation**(*metadata*, *data*)

A raw Universal Decompositional Semantics dataset

Unlike `decomp.semantics.uds.NormalizedUDSAnnotation`, objects of this class may have multiple annotations for a particular attribute. Each annotation is associated with an annotator ID, and different annotators may have annotated different numbers of items.

Parameters
annotation – A mapping from graph identifiers to node/edge identifiers to property subspaces to property to value and confidence for each annotator. Edge identifiers must be represented as NODEID1%%NODEID2, and node identifiers must not contain %%.

annotators(*subspace=None*, *prop=None*)

Annotator IDs for a subspace and property

If neither subspace nor property are specified, all annotator IDs are returned. IF only the subspace is specified, all annotators IDs for the subspace are returned.

Parameters

- **subspace** (`Optional[str]`) – The subspace to constrain to
- **prop** (`Optional[str]`) – The property to constrain to

Return type
`Set[str]`

classmethod from_json(*jsonfile*)

Generates a dataset for raw annotations from a JSON file

For node annotations, the format of the JSON passed to this class method must be:

```
{GRAPHID_1: {NODEID_1_1: DATA,
             ...},
 GRAPHID_2: {NODEID_2_1: DATA,
             ...},
```

(continues on next page)

(continued from previous page)

```
 ...
}
```

Edge annotations should be of the form:

```
{GRAPHID_1: {NODEID_1_1%%NODEID_1_2: DATA,
             ...},
 GRAPHID_2: {NODEID_2_1%%NODEID_2_2: DATA,
             ...},
 ...
}
```

Graph and node identifiers must match the graph and node identifiers of the predpatt graphs to which the annotations will be added.

DATA in the above is assumed to have the following structure:

```
{SUBSPACE_1: {PROP_1_1: {'value': {
                            ANNOTATOR1: VALUE1,
                            ANNOTATOR2: VALUE2,
                            ...
                                  },
                         'confidence': {
                            ANNOTATOR1: CONF1,
                            ANNOTATOR2: CONF2,
                            ...
                                      }
                        },
              PROP_1_2: {'value': {
                            ANNOTATOR1: VALUE1,
                            ANNOTATOR2: VALUE2,
                            ...
                                  },
                         'confidence': {
                            ANNOTATOR1: CONF1,
                            ANNOTATOR2: CONF2,
                            ...
                                      }
                        },
               ...},
 SUBSPACE_2: {PROP_2_1: {'value': {
                            ANNOTATOR3: VALUE1,
                            ANNOTATOR4: VALUE2,
                            ...
                                  },
                         'confidence': {
                            ANNOTATOR3: CONF1,
                            ANNOTATOR4: CONF2,
                            ...
                                      }
                        },
              ...},
...}
```

VALUEi and CONFi are assumed to be unstructured.

Return type
RawUDSAnnotation

items(*annotation_type=None*, *annotator_id=None*)

Dictionary-like items generator for attributes

This method behaves exactly like UDSAnnotation.items, except that, if an annotator ID is passed, it generates only items annotated by the specified annotator.

Parameters

- **annotation_type** (Optional[str]) – Whether to return node annotations, edge annotations, or both (default)
- **annotator_id** (Optional[str]) – The annotator whose annotations will be returned by the generator (defaults to all annotators)

Raises
ValueError – If both annotation_type and annotator_id are passed and the relevant annotator gives no annotations of the relevant type, and exception is raised

class decomp.semantics.uds.annotation.**UDSAnnotation**(*metadata*, *data*)

A Universal Decompositional Semantics annotation

This is an abstract base class. See its RawUDSAnnotation and NormalizedUDSAnnotation subclasses.

The __init__ method for this class is abstract to ensure that it cannot be initialized directly, even though it is used by the subclasses and has a valid default implementation. The from_json class method is abstract to force the subclass to define more specific constraints on its JSON inputs.

Parameters

- **metadata** (*UDSAnnotationMetadata*) – The metadata for the annotations
- **data** (Dict[str, Dict[str, Any]]) – A mapping from graph identifiers to node/edge identifiers to property subspaces to properties to annotations. Edge identifiers must be represented as NODEID1%%NODEID2, and node identifiers must not contain %%.

property edge_attributes

The edge attributes

property edge_graphids: Set[str]

The identifiers for graphs with edge annotations

Return type
Set[str]

property edge_subspaces: Set[str]

The subspaces for edge annotations

Return type
Set[str]

abstract classmethod from_json(*jsonfile*)

Load Universal Decompositional Semantics dataset from JSON

For node annotations, the format of the JSON passed to this class method must be:

```
{GRAPHID_1: {NODEID_1_1: DATA,
             ...},
 GRAPHID_2: {NODEID_2_1: DATA,
             ...},
 ...
}
```

Edge annotations should be of the form:

```
{GRAPHID_1: {NODEID_1_1%%NODEID_1_2: DATA,
             ...},
 GRAPHID_2: {NODEID_2_1%%NODEID_2_2: DATA,
             ...},
 ...
}
```

Graph and node identifiers must match the graph and node identifiers of the predpatt graphs to which the annotations will be added. The subclass determines the form of DATA in the above.

Parameters
jsonfile (Union[str, TextIO]) – (path to) file containing annotations as JSON

Return type
UDSAnnotation

property graphids: Set[str]

The identifiers for graphs with either node or edge annotations

Return type
Set[str]

items(*annotation_type=None*)

Dictionary-like items generator for attributes

If annotation_type is specified as "node" or "edge", this generator yields a graph identifier and its node or edge attributes (respectively); otherwise, this generator yields a graph identifier and a tuple of its node and edge attributes.

property metadata: *UDSAnnotationMetadata*

All metadata for this annotation

Return type
UDSAnnotationMetadata

property node_attributes

The node attributes

property node_graphids: Set[str]

The identifiers for graphs with node annotations

Return type
Set[str]

property node_subspaces: Set[str]

The subspaces for node annotations

Return type
Set[str]

properties(*subspace=None*)

The properties in a subspace

Return type
Set[str]

property_metadata(*subspace*, *prop*)

The metadata for a property in a subspace

Parameters

- **subspace** (str) – The subspace the property is in
- **prop** (str) – The property in the subspace

Return type
UDSPropertyMetadata

property subspaces: Set[str]

The subspaces for node and edge annotations

Return type
Set[str]

decomp.semantics.uds.metadata

Classes for representing UDS annotation metadata.

class decomp.semantics.uds.metadata.**UDSAnnotationMetadata**(*metadata*)

The metadata for UDS properties by subspace

Parameters
metadata (Dict[str, Dict[str, *UDSPropertyMetadata*]]) – A mapping from subspaces to properties to datatypes and possibly annotators

properties(*subspace=None*)

The properties in a subspace

Parameters
subspace (Optional[str]) – The subspace to get the properties of

Return type
Set[str]

class decomp.semantics.uds.metadata.**UDSCorpusMetadata**(*sentence_metadata=<decomp.semantics.uds.metadata.UDSAnno object>*, *document_metadata=<decomp.semantics.uds.metadata.UDSAnnotat object>*)

The metadata for UDS properties by subspace

This is a thin wrapper around a pair of UDSAnnotationMetadata objects: one for sentence annotations and one for document annotations.

Parameters

- **sentence_metadata** (*UDSAnnotationMetadata*) – The metadata for sentence annotations
- **document_metadata** (*UDSAnnotationMetadata*) – The metadata for document_annotations

document_annotators(*subspace=None*, *prop=None*)

The annotators for a property in a document subspace

Parameters

- **subspace** (Optional[str]) – The subspace to get the annotators of
- **prop** (Optional[str]) – The property to get the annotators of

Return type
Set[str]

document_properties(*subspace=None*)

The properties in a document subspace

Parameters
subspace (Optional[str]) – The subspace to get the properties of

Return type
Set[str]

sentence_annotators(*subspace=None*, *prop=None*)

The annotators for a property in a sentence subspace

Parameters

- **subspace** (Optional[str]) – The subspace to get the annotators of
- **prop** (Optional[str]) – The property to get the annotators of

Return type
Set[str]

sentence_properties(*subspace=None*)

The properties in a sentence subspace

Parameters
subspace (Optional[str]) – The subspace to get the properties of

Return type
Set[str]

class decomp.semantics.uds.metadata.**UDSDataType**(*datatype*, *categories=None*, *ordered=None*, *lower_bound=None*, *upper_bound=None*)

A thin wrapper around builtin datatypes

This class is mainly intended to provide a minimal extension of basic builtin datatypes for representing categorical datatypes. pandas provides a more fully featured version of such a categorical datatype but would add an additional dependency that is heavyweight and otherwise unnecessary.

Parameters

- **datatype** (Union[str, int, bool, float]) – A builtin datatype
- **categories** (Optional[List[Union[str, int, bool, float]]]) – The values the datatype can take on (if applicable)
- **ordered** (Optional[bool]) – If this is a categorical datatype, whether it is ordered
- **lower_bound** (Optional[float]) – The lower bound value. Neither categories nor ordered need be specified for this to be specified, though if both categories and this are specified, the datatype must be ordered and the lower bound must match the lower bound of the categories.

- **upper_bound** (Optional[float]) – The upper bound value. Neither categories nor ordered need be specified for this to be specified, though if both categories and this are specified, the datatype must be ordered and the upper bound must match the upper bound of the categories.

property categories: Union[Set[Union[str, int, bool, float]], List[Union[str, int, bool, float]]]

The categories

A set of the datatype is unordered and a list if it is ordered

Raises
ValueError – If this is not a categorical datatype, an error is raised

Return type
Union[Set[Union[str, int, bool, float]], List[Union[str, int, bool, float]]]

classmethod from_dict(*datatype*)

Build a UDSDataType from a dictionary

Parameters
datatype (Dict[str, Union[str, List[Union[str, int, bool, float]], bool]]) – A dictionary representing a datatype. This dictionary must at least have a "datatype" key. It may also have a "categorical" and an "ordered" key, in which case it must have both.

Return type
UDSDataType

class decomp.semantics.uds.metadata.**UDSPropertyMetadata**(*value*, *confidence*, *annotators=None*)

The metadata for a UDS property

classmethod from_dict(*metadata*)

Parameters
metadata (Dict[str, Union[Set[str], Dict[str, Dict[str, Union[str, List[Union[str, int, bool, float]], bool]]]]]) – A mapping from "value" and "confidence" to *decomp.semantics.uds.metadata.UDSDataType*. This mapping may optionally specify a mapping from "annotators" to a set of annotator identifiers.

Return type
UDSPropertyMetadata

4.3 decomp.corpus

Module for defining abstract corpus readers

4.3.1 decomp.corpus.corpus

Module for defining abstract graph corpus readers

class decomp.corpus.corpus.**Corpus**(*graphs_raw*)

Container for graphs

Parameters
graphs_raw (Iterable[TypeVar(InGraph)]) – a sequence of graphs in a format that the graph-builder for a subclass of this abstract class can process

property graphids: List[Hashable]

The graph ids in corpus

Return type
List[Hashable]

property graphs: Dict[Hashable, OutGraph]

the graphs in corpus

Return type
Dict[Hashable, TypeVar(OutGraph)]

items()

Dictionary-like iterator for (graphid, graph) pairs

Return type
Iterable[Tuple[Hashable, TypeVar(OutGraph)]]

property ngraphs: int

Number of graphs in corpus

Return type
int

sample(*k*)

Sample k graphs without replacement

Parameters
k (int) – the number of graphs to sample

Return type
Dict[Hashable, TypeVar(OutGraph)]

4.4 decomp.graph

Module for converting between NetworkX and RDFLib graphs

4.4.1 decomp.graph.rdf

Module for converting from networkx to RDF

class decomp.graph.rdf.**RDFConverter**(*nxgraph*)

A converter between NetworkX digraphs and RDFLib graphs

Parameters
nxgraph (DiGraph) – the graph to convert

classmethod networkx_to_rdf(*nxgraph*)

Convert a NetworkX digraph to an RDFLib graph

Parameters
nxgraph (DiGraph) – the NetworkX graph to convert

Return type
Graph

4.4.2 decomp.graph.nx

Module for converting from networkx to RDF

class decomp.graph.nx.**NXConverter**(*rdfgraph*)

A converter between RDFLib graphs and NetworkX digraphs

Parameters
graph – the graph to convert

classmethod rdf_to_networkx(*rdfgraph*)

Convert an RDFLib graph to a NetworkX digraph

Parameters
rdfgraph (Graph) – the RDFLib graph to convert

Return type
DiGraph

4.5 decomp.vis

4.5.1 decomp.vis.uds_vis

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

E

F

G

H

I

M

N

U