rnaglib

Release 0.0.1

Vincent Mallet, Carlos Oliver, Jonathan Broadbent, William L. Ham

Sep 15, 2023

GET STARTED

CHAPTER ONE

PYPI INSTALL

```
$ pip install rnaglib
```

CHAPTER TWO

LATEST VERSION

```
$ git clone https://github.com/cgoliver/rnaglib.git
$ cd rnaglib
$ pip install .
```

Warning: RNAglib does not install downstream deep learning frameworks which you may need for model training and data loading. See installation instructions for dgl and pytorch-geometric.

CHAPTER

THREE

QUICKSTART

3.1 Get the data

Once you have *installed* RNAglib, you can fetch a pre-built dataset of RNA structures using the command line:

```
rnaglib_download
```

By default you will get non-redundant RNA structures saved to `~/.rnaglib`.

To obtain different versions or larger sets of RNAs have a look at the command line options `rnaglib_download --help`.

3.2 Load single RNA

Annotations for each RNA are accessed through networkx graph objects. You can load one RNA using `graph_from_pdbid()`

```
>>> from rnaglib.utils import available_pdbids
>>> from rnaglib.utils import graph_from_pdbid

>>> pdbids = available_pdbids()
>>> rna = graph_from_pdbid(pdbids[0])
>>> rna
DiGraph with 69 nodes and 194 edges
>>> rna.graph
{'dbn': {'all_chains': {'num_nts': 143, 'num_chars': 144, 'bseq':
↪'GCCCGGAUAGCUCAGUCGGUAGAGCAGGGGAUUGAAAAUCCCCGUGUCCUUGGUUCGAUUCCGAGUCUGGGCAC&
↪CGGAUAGCUCAGUCGGUAGAGCAGGGGAUUGAAAAUCCCCGUGUCCUUGGUUCGAUUCCGAGUCCGGGC', 'sstr':
↪'(((((((..((((.....[..)))).(((((.......))))).....(((((..]....))))))))))))..&((((..((((.
↪....[..)))).(((((.......))))).....(.(((..]....))).)))))...', 'form': 'AAAAAA...AA...A..
↪.....AAA.AAAA.......A.AAA......AAAAA..A....AAAAAAAAAAAA.-&.AA...AA...A.......AAA.AAAA..
↪.....A.AAA......AAAAA..A....A...AAAA.A.-'}...,
```

See the data *tutorial* for more on the data.

3.3 Load an RNA Dataset

For machine learning purposes, we often want a collection of data objects in one place. For that we have the `RNADataset` object.:

```
from rnaglib.data_loading import RNADataset

dataset = RNADataset()
```

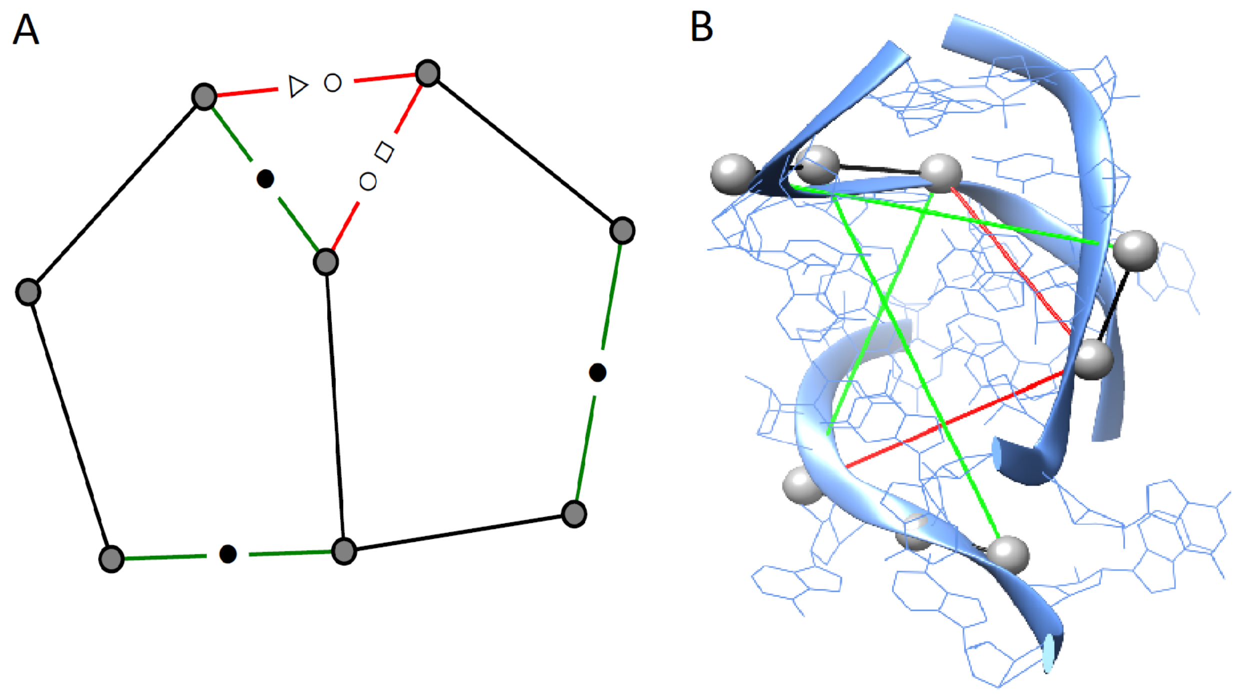
This object holds the same objects as above but also supports ML functionalities such as converting the RNAs to different representations (graphs, point clouds, voxels) and to different frameworks (dgl, torch, pytorch geometric) See the ML *tutorial* for more on model training and tasks.

CHAPTER

FOUR

WHAT IS AN RNA 2.5D STRUCTURE?

RNA 2.5D structures are discrete graph-based representations of atomic coordinates derived from techniques such as X-ray crystallography and NMR. This type of representation encodes all possible base pairing interactions which are known to be crucial for understanding RNA function.



Why use RNA 2.5D data?

The benefit is twofold. When dealing with RNA 3D data, a representation centered on base pairing is a very natural prior which has been shown to carry important signals for complex interactions, and can be directly interpreted. Second, adopting graph representations lets us take advantage of many powerful algorithmic tools such as graph neural networks and graph kernels.

What type of functional data is included?

The graphs are annotated with graph, node, and edge-level attributes. These include, but are not limited to:

- Secondary structure
- Protein binding
- Small molecule binding
- Chemical modifications

- 3-D coordinates
- Leontis-westhof base pair geometry classification

We provide a visualization of what the graphs in this database contain. A more detailed description of the data is presented in rnaglib.data.

CHAPTER

FIVE

BUILDING RNA DATABASES WITH RNAGLIB

This module (*prepare_data*) contains all the necessary code to build databases of annotated RNA 3D structures, and the user interfaces with it through the *rnaglib_prepare_data* command line script. Dataset creation follows the following steps:

- **Fetching the raw RNA structures from either:**
  - RCSB PDB Databank (accepts the *–nr* flag to only use structures in the [BGSU Representative Set](https://www.bgsu.edu/research/rna/databases/non-redundant-list.html)
  - A local user-defined folder
- For each structure, run x3dna-dssr
- Store x3dna-dssr output in a networkx Graph object
- **If the *–annotate* flag is passed for pre-training:**
  - Chop the whole RNAs into smaller chunks
  - Pre-compute local neighbourhoods
  - Extract all graphlets

5.1 Quickstart

Print the help message:

```
$ rnaglib_prepare_data -h
```

To run a quick debug build with default values:

```
$ rnaglib_prepare_data -s structures/ --tag first_build -o builds/ -d
```

5.2 Data versioning

The optional argument *–tag* is used to name the folder containing the final output. For our distributions we use *rnaglib-<'all' or 'nr'><'-annotated' or ''><'-chopped' or ''>-<version>* depending on the build options. We distribute data builds with all available RNAs and assign *all* to the tag, and non-redundant structures according to the [BGSU Representative Set](https://www.bgsu.edu/research/rna/databases/non-redundant-list.html). For each of these two choices, we also provide versions pre-processed for [graphlet kernel](https://rnaglib.cs.mcgill.ca/static/docs/html/rnaglib.kernels.html) computations used to compute node similarity and assign the *annot* value to the tag.

5.3 Output

After running the *–debug* test run above, your *./builds/* folder will contain a single sub-folder called *./builds/graphs* with 10 *.json* files and a file *./builds/graphs/errors.csv*. Each of these JSONs contains the annotated RNAs and the CSV contains a list of RNAs that failed to build and the failure reason.

5.4 Data building options

- *–nr* only outputs RNAs from the non-redundant set from BGSU
- *–chop* creates a sub-folder in the build called *chops* which contains chunked RNAs for even batch sizes
- *–annot* builds necessary annotations for computing node similarities

CHAPTER

SIX

CITING

If you use this tool in your work, we thank you for citing:

```
@article{mallet2022rnaglib,
  title={RNAglib: a python package for RNA 2.5 D graphs},
  author={Mallet, Vincent and Oliver, Carlos and Broadbent, Jonathan and Hamilton,␣
↪William L and Waldisp{\"u}hl, J{\'e}r{\^o}me},
  journal={Bioinformatics},
  volume={38},
  number={5},
  pages={1458--1459},
  year={2022},
  publisher={Oxford University Press}
}
```

CHAPTER

SEVEN

WORKING WITH 2.5D GRAPHS

Now that we know *what is an RNA 2.5D graph* we can inspect the graph using *rnaglib*.

7.1 Fetching hosted graphs

The libray ships with some pre-built datasets which you can download with the following command line:

```
$ rnaglib_download
```

This will download the default data distribution to *~/.rnaglib*

To see the list of available PDBs you downloaded, use:

```
from rnaglib.utils import available_pdbids
# returns a list of PDBIDs
pdbids = available_pdbids()
# get the first RNA by PDBID
rna = graph_from_pdbid(pdbids[0])
```

Warning: The list of available PDBs depends on which data build you want to use. See *preparing data* for more info on versioning and data build arguments. You can pass these arguments to the *available_pdbids(redundancy='all', version='0.0.0', annotated=True)* for non-default builds.

7.2 Overview of the 2.5D Graphs

First, let us have a look at the 2.5d graph object from a code perspective. We use networkx to store the RNA information in a *nx.DiGraph* directed graph object. Once the graphs are downloaded, they can be fetched directly using their PDBID.

Since nodes represent nucleotides, the node data dictionary will include features such as nucleotide type, position, 3D coordinates, etc… Nodes are assigned an ID in the form <`pdbid.chain.position`>. Using node IDs we can access node and edge attributes as dictionary keys.

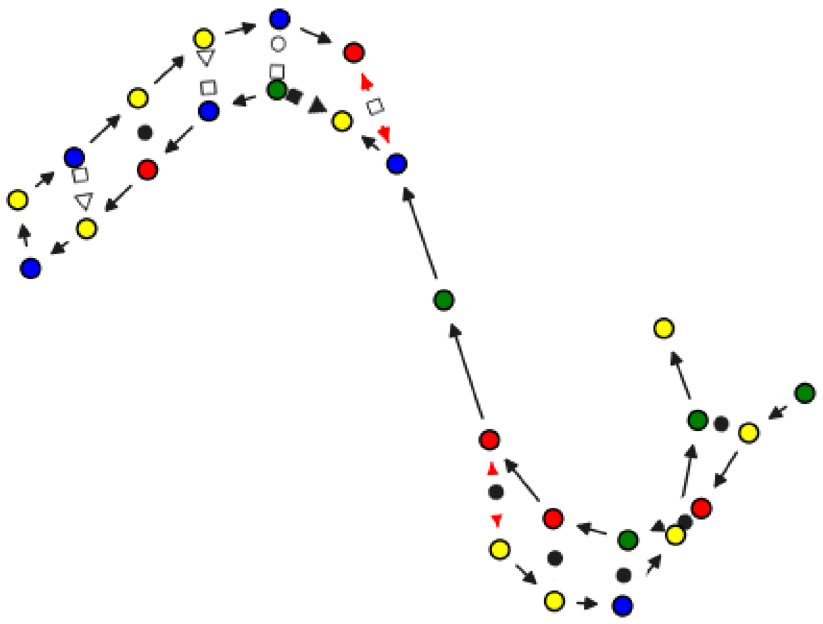
```
>>> from rnaglib.utils import graph_from_pdbid
>>> G = graph_from_pdbid("4nlf")
>>> G.nodes['4nlf.A.2647']
 {'index': 1, 'index_chain': 1, 'chain_name': 'A', 'nt_resnum': 2647, 'nt_name': 'U',
↪'nt_code': 'U',
  'binding_protein': None, 'binding_ion': None, 'binding_small-molecule': None}
```

The RNA 2.5D graph contains a rich set of annotations. For a complete list of these annotations see *this page*.

7.3 Visualization

To visualize the 2.5D graphs in the format described above, we have implemented a drawing toolbox with several functions. The easiest way to use it in your application is to call `rnaglib.drawing.draw(graph, show=True)`. A functioning installation of Latex is needed for correct plotting of the graphs. If no installation is detected, the graphs will be plotted using the LaTex reduced features that ships with matplotlib.

```
>>> from rnaglib.drawing import rna_draw
>>> rna_draw(G, show=True, layout="spring")
```



In the next two examples we will show how you can make use of these annotations to study chemical modifications and RNA-protein binding sites.

7.4 Analyzing RNA-small molecule binding sites

In this short example we will compute some statistics to describe the kinds of structural features around RNA-small molecule binding pockets using RNAGlib.

Let's get our graphs. We are using the default data build which contains whole non-redundant RNA structures. We will iterate over all available non-redundant RNAs and extract residues near small molecules.

```
from rnaglib.utils import available_pdbids
from rnaglib.utils import graph_from_pdbid

pockets = []
for i,G in enumerate(graphs):
        try:
            pocket = [n for n, data in G.nodes(data=True) if data['binding_small-molecule
↪'] is not None]
            # sample same number of random nucleotides
            non_pocket = random.sample(list(G.nodes()), k=len(pocket))
        except KeyError as e:
            continue
        if pocket:
            pockets.append((pocket, non_pocket, G))
        else:
            # no pocket found
            pass
```

Now we have a list of pockets where each is a thruple of a list of pocket nodes, a list of non-pocket nodes, and the parent graph. Let's collect some stats about these residues. Namely, what base pair types and secondary structure elements they are involved in.

```
bps, sses = [], []

for pocket, non_pocket, G in pockets:
    for nt in pocket:
        # add edge type of all base pairs in pocket
        bps.extend([{'bp_type': data['LW'],
                     'is_pocket': True} for _,data in G[nt].items()])
        # sse key is format '<sse type>_<id>'
        node_data = G.nodes[nt]
        if node_data['sse']['sse'] is None:
            continue
        sses.append({'sse_type': node_data['sse']['sse'].split("_")[0],
                     'is_pocket': True})

    # do the same for non-pocket
    for nt in non_pocket:
        # add edge type of all base pairs in pocket
        bps.extend([{'bp_type': data['LW'],
                     'is_pocket': False} for _,data in G[nt].items()])
        # sse key is format '<sse type>_<id>'
        node_data = G.nodes[nt]
        if node_data['sse']['sse'] is None:
            continue
```

(continues on next page)

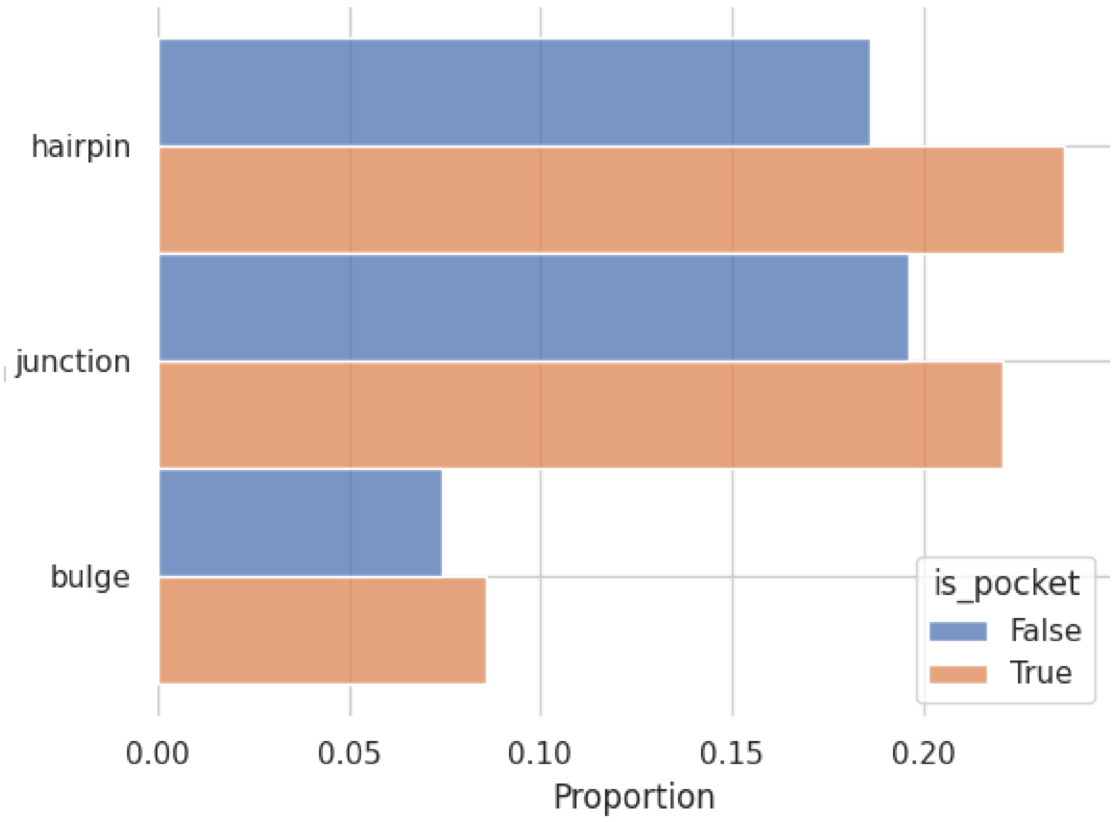
(continued from previous page)

```
        sses.append({'sse_type': node_data['sse']['sse'].split("_")[0],
                     'is_pocket':False})


# for convenience convert to dataframe
bp_df = pd.DataFrame(bps)
sse_df = pd.DataFrame(sses)
```

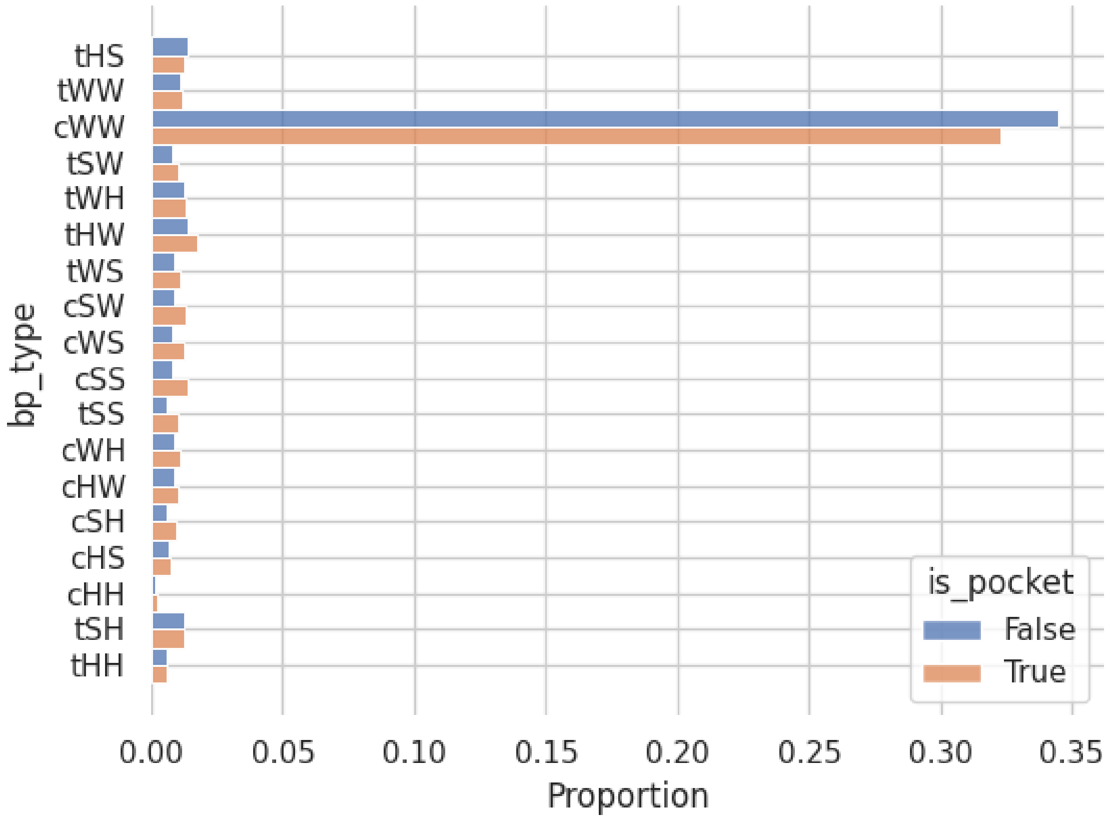
Finally we can draw some plots of the base pair type and secondary structure element distribution around small molecule binding sites.

```
# remove backbones
bp_df = bp_df.loc[~bp_df['bp_type'].isin(['B35', 'B53'])]

sns.histplot(y='bp_type', hue='is_pocket', multiple='dodge', stat='proportion', data=bp_
↪df)
sns.despine(left=True, bottom=True)
plt.savefig("bp.png")
plt.clf()

sns.histplot(y='sse_type', hue='is_pocket', multiple='dodge', stat='proportion',␣
↪data=sse_df)
sns.despine(left=True, bottom=True)
plt.savefig("sse.png")
plt.clf()
```

This is the distribution of secondary structures in binding pockets and in a random sample of residues:



And the same but for the different LW base pair geometries:



From this small experiment we confirm a property of RNA binding sites which is that they tend to occur in looping regions with a slight tendency towards non-canonical (non-CWW) base pair geometries.

`Download source code for this example.`

7.5 Aligning two RNA graphs: Graph Edit Distance (GED)

GED is the gold standard of graph comparisons. We have put our ged implementation as a part of networkx, and offer in *rnaglib.ged* the weighting scheme we propose to compare 2.5D graphs. One can call `rnaglib.ged.ged()` on two graphs to compare them. However, due to the exponential complexity of the comparison, the maximum size of the graphs should be around ten nodes, making it more suited for comparing graphlets or subgraphs.

```
>>> from rnaglib.ged import graph_edit_distance
>>> from rnaglib.utils import graph_from_pdbid
>>> G = graph_from_pdbid("4nlf")
>>> graph_edit_distance(G, G)
... 0.0
```

7.6 Using your own local RNA structures

If you have an mmCIF containing RNA stored locally and you wish to build a 2.5D graph that can be used in RNAglib you can use the `prepare_data` module. To do so, you need to have `x3dna-dssr` executable in your `$PATH` which requires a *license <http://x3dna.org/>*. The first option is to use the library from a python script, following the example :

```
>>> from rnaglib.prepare_data import cif_to_graph

>>> pdb_path = '../data/1aju.cif'
>>> graph_nx = cif_to_graph(pdb_path)
```

Another possibility is to use the shell function that ships with rnaglib.

```
$ rnaglib_prepare_data  --one_mmcif $PATH_TO_YOUR_MMCIF -O /path/to/output
```

CHAPTER

EIGHT

MACHINE LEARNING TUTORIAL

8.1 RNAGlib data structures

We have introduced the 2.5D graph format in another tutorial. RNAGlib provides access to collections of RNAs for machine learning with PyTorch. It revolves around the usual Dataset and Dataloader objects, along with a Representation object :

- *RNADatasets* objects are an iterable of RNA data, that returns representations of this data
- The *Representation* object will return our data in a certain representation (e.g. graphs, voxels, point clouds) as well as cast to different data science and ML frameworks (DGL, pytorch-geometric, networkx).
- The *get_loader* function encapsulates automatic data splitting and collating and returns appropriate PyTorch data loaders.

8.1.1 Datasets

The *rnaglib.data_loading.RNADataset* object builds and provides access to collections of RNAs. When using the Dataset class, our standard data distribution should be downloaded automatically. Alternatively, you can choose to provide your own annotated RNAs by providing a *data_path*.

To create a dataset using our hosted data simply instantiate the *RNADataset* object.

```
from rnaglib.data_loading import RNADataset

dataset = RNADataset()
```

Different datasets can be specified using the following options:

- *version='x.x.x'*: which data build to load
- *nr=False*: by default, we only load non-redundant structures, if you want all structures in the PDB, set this flag to *False*
- *all_graphs*: a specific list of pdb ids to iterate through

Datasets can be indexed like a list or you can inspect an individual RNA by its PDBID.

```
rna_1 = dataset[3]
pdbid = dataset.available_pdbids[3]
rna_2 = dataset.get_pdbid(pdbid)
```

The returned object is a dictionnary with three entries :

- rna : The raw 2.5D graph in the form of a networkx object which you can inspect as shown in *this tutorial*.

- rna_name : the name of the PDB being returned
- path : the path to the pdb files

8.1.2 Representations

The next important object for RNAGlib is the *representation*. Previously, our return only included the raw data. One can add a *Representation* object with arguments to post-process this raw data into a more usable data format. The most trivial one is to ask for a *GraphRepresentation*. One can choose either networkx, DGL or PyTorch Geometric as a return type.

By default, this 2.5D graph only includes the connectivity of the graphs. The user can ask for input nucleotide features and nucleotide targets. As an example, we use nucleotide identity ('nt_code') as input and the binding of an ion ('binding_ion') as output. These two additions are exemplified below :

```
from rnaglib.representations import GraphRepresentation

graph_rep = GraphRepresentation(framework='dgl')
nt_features = ['nt_code']
nt_targets = ['binding_ion']
dataset = RNADataset(nt_features=nt_features, nt_targets=nt_targets,␣
↪representations=[graph_rep])
print(dataset[0]['graph'])

>>> {Graph(num_nodes=24, num_edges=58,
        ndata_schemes={'nt_features': Scheme(shape=(4,), dtype=torch.float32),
                       'nt_targets': Scheme(shape=(1,), dtype=torch.float32)}
        edata_schemes={'edge_type': Scheme(shape=(), dtype=torch.int64)})}
```

We currently support two other data representations : *PointCloudRepresentation* and *VoxelRepresentation* More generally, *rnaglib.representations.Representation* class holds the logic for converting a dataset to one of the above representations and users can easily sub-class this to create their own representations.

These classes come with their own set of attributes. Users can use several representations at the same time.

```
from rnaglib.representations import PointCloudRepresentation, VoxelRepresentation

pc_rep = PointCloudRepresentation()
voxel_rep = VoxelRepresentation(spacing=2)

dataset.add_representation(voxel_rep)
dataset.add_representation(pc_rep)
print(dataset[0].keys())

>>> dict_keys(['rna_name', 'rna', 'path', 'graph', 'voxel', 'point_cloud'])
```

As can be seen, we now have additional keys in the returned dictionnary corresponding to the data represented as voxels or point clouds. In our case, the RNA has 24 nucleotides and is approximately 12 Angrstroms wide. Hence, dataset[0]['point_cloud'] is a dictionnary that contains two grids in the PyTorch order :

- `voxel_feats : torch.Size([4, 6, 5, 6])`
- `voxel_target : torch.Size([1, 6, 5, 6])`

While dataset[0]['point_cloud'] is a dictionnary that contains one list and three tensors :

- `point_cloud_coords torch.Size([24, 3])`

- point_cloud_feats torch.Size([24, 4])
- point_cloud_targets torch.Size([24, 1])
- point_cloud_nodes ['1a9n.Q.0', '1a9n.Q.1',... '1a9n.Q.9']

8.1.3 Dataloader

The missing piece is utilities to efficiently load our dataset for machine learning. The first task is to split our data in a principled way. To enhance reproducibility, we offer automatic random splitting procedure that avoid loading useless graphs (for instance graphs with no positive nodes for node classification) and balance the train/test proportions in the multi-task setting.

The other problematic step is to batch our data automatically, as the batching procedure depends on the representations that are used. These two functionalities are implemented in a straightforward manner :

```
from torch.utils.data import DataLoader
from rnaglib.data_loading import split_dataset, Collater

train_set, valid_set, test_set = split_dataset(dataset, split_train=0.7, split_valid=0.
↪85)
collater = Collater(dataset=dataset)
train_loader = DataLoader(dataset=train_set, shuffle=True, batch_size=2, num_workers=0,␣
↪collate_fn=collater.collate)

for batch in train_loader:
    ...
```

will yield a dictionnary with the same keys and structure as above, for batches of two graphs.

8.2 More advanced functionalities

8.2.1 Additional inputs and outputs

Adding more input features to the graphs is straightforward, as you simply have to specify more items in the features list. A full description of the input features that can be used is available in rnaglib.data. Similarly, you can seamlessly switch to a multi-task setting by adding more targets. However, doing this affects the splitting procedure. A side effect could be a slight deviation in the train/validation/test fractions. The tasks currently implemented are in the set : {'node_binding_small-molecule', 'node_binding_protein', 'node_binding_ion', "node_is_modified"}. An example of a variation is provided below, the rest of the code is unaffected.

```
nt_features = ['nt_code', "alpha", "C5prime_xyz", "is_modified"]
nt_targets = ['binding_ion', 'binding_protein']
```

8.2.2 Unsupervised pre-training

Due to a relatively scarse data, we have found useful to pretrain our networks. The semi-supervised setting was found to work well, where node embeddings are asked to approximate a similarity function over subgraphs. More precisely, given two subgraphs g1 and g2, a similarity function K, and a neural embedding function f, we want to approximate K(sg1,sg2) ~ <f(sg1), f(sg2)> . This was described more precisely in VeRNAl .

The datasets and dataloaders natively support the computation of many comparison functions, factored in the SimFunctionNode object. We also offer the possibility to compute this comparison on a fixed number of sampled nodes from the batch, using the max_size_kernel argument. To use this functionality, we packaged into an additional Representation. The loader will then return an additional field in the batch, with a 'ring' key that represents the values of the similarity function over subgraphs.

```
from rnaglib.kernels import node_sim
from rnaglib.representations import RingRepresentation


node_simfunc = node_sim.SimFunctionNode(method='R_1', depth=2)
ring_rep = RingRepresentation(node_simfunc=node_simfunc, max_size_kernel=100)
da.add_representation(ring_rep)
train_loader, _, _ = graphloader.get_loader(dataset=unsupervised_dataset)
```

The coordinated use of these functionalities is illustrated in the *rnaglib.examples*: section.

CHAPTER

NINE

RNAGLIB.EXAMPLES

9.1 Protein Binding

This script just shows a first very basic example : learn binding protein preference from the nucleotide types and the graph structure

To do so, we choose our data, create a data loader around it, build a RGCN model and train it.

```
from rnaglib.learning import models, learn
from rnaglib.data_loading import graphloader

# Choose the data, features and targets to use and GET THE DATA GOING
node_features = ['nt_code']
node_target = ['binding_protein']
supervised_dataset = loader.SupervisedDataset(node_features=node_features,
                                              node_target=node_target)
train_loader, validation_loader, test_loader = loader.Loader(dataset=supervised_dataset).
↪get_data()

# Define a model, we first embed our data in 10 dimensions, and then add one␣
↪classification
input_dim, target_dim = supervised_dataset.input_dim, supervised_dataset.output_dim
embedder_model = models.Embedder(dims=[10, 10], infeatures_dim=input_dim)
classifier_model = models.Classifier(embedder=embedder_model, classif_dims=[target_dim])

# Finally get the training going
optimizer = torch.optim.Adam(classifier_model.parameters(), lr=0.001)
learn.train_supervised(model=classifier_model,
                       optimizer=optimizer,
                       train_loader=train_loader)
```

9.2 Small Molecule Binding

This script shows a second more complicated example : learn binding protein preferences as well as small molecules binding from the nucleotide types and the graph structure We also add a pretraining phase based on the R_graphlets kernel

```
from rnaglib.learning import models, learn
from rnaglib.data_loading import graphloader
from rnaglib.benchmark import evaluate
from rnaglib.kernels import node_sim

# Choose the data, features and targets to use
node_features = ['nt_code']
node_target = ['binding_protein']

###### Unsupervised phase : ######
# Choose the data and kernel to use for pretraining
print('Starting to pretrain the network')
node_sim_func = node_sim.SimFunctionNode(method='R_graphlets', depth=2)
unsupervised_dataset = loader.UnsupervisedDataset(node_simfunc=node_sim_func,
                                                  node_features=node_features)
train_loader = loader.Loader(dataset=unsupervised_dataset, split=False,
                             num_workers=0, max_size_kernel=100).get_data()

# Then choose the embedder model and pre_train it, we dump a version of this pretrained␣
↪model
embedder_model = models.Embedder(infeatures_dim=unsupervised_dataset.input_dim,
                                 dims=[64, 64])
optimizer = torch.optim.Adam(embedder_model.parameters())
learn.pretrain_unsupervised(model=embedder_model,
                            optimizer=optimizer,
                            train_loader=train_loader,
                            learning_routine=learn.LearningRoutine(num_epochs=10),
                            rec_params={"similarity": True, "normalize": False, "use_
↪graph": True, "hops": 2})
# torch.save(embedder_model.state_dict(), 'pretrained_model.pth')
print()

###### Now the supervised phase : ######
print('We have finished pretraining the network, let us fine tune it')
# GET THE DATA GOING, we want to use precise data splits to be able to use the benchmark.
train_split, test_split = evaluate.get_task_split(node_target=node_target)
supervised_train_dataset = loader.SupervisedDataset(node_features=node_features,
                                                    redundancy='NR',
                                                    node_target=node_target,
                                                    all_graphs=train_split)
train_loader = loader.Loader(dataset=supervised_train_dataset, split=False).get_data()

# Define a model and train it :
# We first embed our data in 64 dimensions, using the pretrained embedder and then add␣
↪one classification
# Then get the training going
classifier_model = models.Classifier(embedder=embedder_model, classif_dims=[supervised_
```

(continues on next page)

(continued from previous page)

```
↪train_dataset.output_dim])
optimizer = torch.optim.Adam(classifier_model.parameters(), lr=0.001)
learn.train_supervised(model=classifier_model,
                       optimizer=optimizer,
                       train_loader=train_loader,
                       learning_routine=learn.LearningRoutine(num_epochs=10))

# torch.save(classifier_model.state_dict(), 'final_model.pth')
# embedder_model = models.Embedder(infeatures_dim=4, dims=[64, 64])
# classifier_model = models.Classifier(embedder=embedder_model, classif_dims=[1])
# classifier_model.load_state_dict(torch.load('final_model.pth'))

# Get a benchmark performance on the official uncontaminated test set :
metric = evaluate.get_performance(node_target=node_target, node_features=node_features,␣
↪model=classifier_model)
print('We get a performance of :', metric)
```

9.3 Link Prediction

This is a very basic example of link prediction applied to RNA base pairs. We use our Embedder object along with the nucleotide ID as features. This is passed to an edge loader and a base pair predictor model.

```
from rnaglib.learning import models, learn
from rnaglib.data_loading import graphloader
from rnaglib.benchmark import evaluate

 # Get loader for link prediction,
 # use nucleotide identity as input features and base our fixed train/test split
 # on the binding protein one for reproducibility
 node_features = ['nt_code']
 node_target = ['binding_protein']
 train_split, test_split = evaluate.get_task_split(node_target=node_target)

 train_dataset = loader.GraphDataset(node_features=['nt_code'], all_graphs=train_split)
 test_dataset = loader.GraphDataset(node_features=['nt_code'], all_graphs=test_split)
 train_loader = loader.EdgeLoaderGenerator(loader.Loader(train_dataset, split=False).get_
↪data())
 test_loader = loader.EdgeLoaderGenerator(loader.Loader(test_dataset, split=False).get_
↪data())

 # Choose the data, features and targets to use and GET THE DATA GOING
 embedder_model = models.Embedder(dims=[10, 10], infeatures_dim=train_dataset.input_dim)
 linkpred_model = models.BasePairPredictor(embedder_model)

 # Finally get the training going
 optimizer = torch.optim.Adam(linkpred_model.parameters(), lr=0.001)
 learn.train_linkpred(linkpred_model, optimizer, train_loader, test_loader)
```

CHAPTER

TEN

DATA REFERENCE

10.1 Graph Format

Each graph contains structure information for one model of a PDB entry containing at least one RNA chain.

Graphs are stored in *JSON* node-link format which can be loaded by [networkx](https://networkx.org/documentation/stable/reference/readwrite/generated/networkx.readwrite.json_graph.node_link_data.html#networkx.readwrite.json_graph.node_link_data). All data comes from the output of *x3dna-dssr* which can be downloaded [here](https://x3dna.org/) and our custom interface extraction tools.

Graphs are dumped as JSONs in the node-link format.

```
import json
from networkx.readwrite.json_graph import node_link_graph

G  = node_link_graph(open('path/to/graph', 'r'))
```

10.2 Nodes

10.2.1 Node IDs

Node IDs are strings in the form *[pdb id].[chain name].[residue number]*.

10.2.2 Node Data

To access node data dictionary:

```
G.nodes[<node_id>]
```

These are the keys in the node data dictionary:

- *'index'*: (int), relative index along chain starting at 1 (e.g. *1*)
- *'index_chain'*: (int) 26 (e.g. *26*)
- *'chain_name'*: (str), name of chain. (e.g. *A*)
- *'nt_resnum'*: (int), residue number according to PDB. (e.g. *101*)
- *'nt_name'*: (str), nucleotide name (e.g. *G*)
- *'nt_code'*: (str), (e.g. *'U'*)

- *'nt_id'*: (str) unique nucleotide ID generated by DSSR. <*chain name*>.<*nt name*><*nt_resnum*> (e.g. *'A.U42'*),
- *'nt_type'*: (str) molecule type of residue (e.g. *'RNA'*)
- *'dbn':* (str) dot-bracket notation for the residue (e.g. *')'*)
- *'summary'*: (str) additional residue info (e.g. *"anti,~C3'-endo,BI,canonical,non-pair-contact,helix,stem,coaxial-stack"*)
- *'alpha'*: (float) base angle in degrees *[-180, 180].*
- *'beta'*: (float) base angle
- *'gamma'*: (float) base angle
- *'delta'*: (float) base angle
- *'epsilon'*: (float)
- *'zeta'*: (float)
- *'epsilon_zeta'*: (float)
- *'bb_type':* (str) Backbone type 'BI',
- *'chi'*: (float)
- *'glyco_bond'*: str (e.g. *'anti'*
- *'C5prime_xyz': (list), 5' Carbon xyz coordinates (e.g. `[-1.343, 8.453, 1.288]*)
- *'P_xyz'*: (list) Phosphate coordinates.
- *'form'*: (str) (e.g. *'A'*) classification of a dinucleotide step comprising the bp above the given designation and the bp that follows it. Types include 'A', 'B' or 'Z' for the common A-, B- and Z- form helices, '.' for an unclassified step , and 'x' for a step without a continuous backbone.
- *'ssZp'*: (float) (e.g. *4.41*),
- *'Dp'*: (float) (e.g. *4.404*)
- *'splay_angle'*: (float) (e.g. *21.6*),
- *'splay_distance'*: (float) (e.g. *3.612*)
- *'splay_ratio':* (float) (e.g. *0.199*)
- *'eta'*: (float) (e.g. *169.652*),
- *'theta':* -167.457,
- *'eta_prime'*: (float) (e.g. *-176.189*)
- *'theta_prime':* (float) (e.g. *-167.27*)
- *'eta_base'*: (float) (e.g. *-135.681*)
- *'theta_base'*: (float) (e.g. *-141.003*)
- *'v0'*: (float) (e.g *8.194*)
- *'v1'*: (float) (e.g. *-28.393*),
- *'v2'*: (float)
- *'v3'*: (float)
- *'v4'*: (float)
- *'amplitude'*: (float)

- *'phase_angle'*: (float)
- *'puckering'*: (str) (e.g. *"C3'-endo"*)
- *'sugar_class':* (str) (e.g. *"~C3'-endo"*)
- *'bin'*: (str) (e.g. *'33t'*) (name of the 12 bins based on [delta (i -1) , delta , gamma], where delta (i -1) and delta can be either 3 (for C3 '- endo sugar) or 2 (for C2 '- endo) and gamma can be p/t/ m (for gauche +/ trans / gauche - conformations , respectively) (2 x2x3 =12 combinations : *33p* , *33t* , … *22m*); *'inc'* refers to incomplete cases (i .e., with missing torsions) , and *'trig'* to triages (i.e., with torsion angle outliers),[1]
- *'cluster'*: (str) (e.g. *'1c'*) (2-char suite name, for one of 53 reported clusters (46 certain and 7 wannabes) , '__' for incomplete cases , and *'!!'* for outliers),[1]
- *'suiteness'*: (float) (measure of conformer - match quality (low to high in range 0 to 1)) [1]
- *'filter_rmsd'*: (float)
- *'frame':* (dict) e.g. (*{'rmsd': 0.006, 'origin': [-4.856, 8.564, -1.171], 'x_axis': [0.922, 0.386, -0.006], 'y_axis': [0.098, -0.25, -0.963], 'z_axis': [-0.374, 0.888, -0.269], 'quaternion': [0.592, -0.781, -0.155, 0.122]}*
- *'sse'*: (dict) Secondary structure info (e.g. residue inside third hairpin *{'sse': 'hairpin_3'}*)
- *'binding_protein'*: (dict) RNA-Protein interface. If no interface found, *None*. Else, dictionary (e.g. *{'nt-aa': 'C-arg', 'nt': 'A.C37', 'aa': 'A.ARG47', 'Tdst': '6.62', 'Rdst': '-114.00', 'Tx': '-1.15', 'Ty': '1.89', 'Tz': '6.23', 'Rx': '-53.57', 'Ry': '19.41', 'Rz': '-103.42', 'sse': 'a-helix'}*)
- *'binding_ion'*: (string) molecule ID of ion if residue is at a binding site (otherwise *None*) (e.g. *'Ca'*)
- *'binding_small-molecule'*: (string) molecule ID of small molecule if residue is at a binding site (otherwise *None*) (e.g. *'SAM'*)

10.3 Edge data

Each edge also has an attribute dictionary:

```
G.edges[(<node_1>, <node_2>)]
```

- *'index'*: (int) Index of edge in DSSR ordering.
- *'nt1'*: (str) DSSR nucleotide ID of first base (e.g. *'A.G17'*)
- *'nt2'*: (str) DSSR nucleotide ID of second base (e.g. *'A.G29'*)
- *'bp'* (str): Nucleotide identity of paired residues (e.g. *'G-C'*)
- *'name'*: (str) (e.g. *'WC'*)
- *'Saenger'*: (str) Saenger base pairing category (e.g. *'19-XIX'*),
- *'LW'*: (str) Leontis-Westhof base pair geometry category (e.g. *'cWW'*)
- *'DSSR'*: (str) Custom DSSR base pair geometry category (e.g. *'cW-W'*)

10.4 Graph-level data

Each graph also has an attribute dictionary:

```
G.graph
```

- *'dbn'*: a dict containing information on the chains contained in the graph, such as the sequences or their length
- *'resolution_{low,high}'*: bounds on the resolution, present in most (~80%) of the graphs
- *'proteins'*: A list of the residues in contact with a protein
- *'ligands'*: A list of the ligands interacting with the graph nucleotides. Each ligand is a dict with the ligand Biopython id, its name, and the rna residues it is bound to
- *'ions'*: same thing with ions

## Graph creation pipeline * *dssr_to_graphs.py* : runs dssr on the cif file to get the first networkx graph object. It moreover computes the RNA/protein interfaces (since it uses dssr) and annotates at the level of the node. * *annotations.py'*: Completes the graph using the mmcif file to include resolution and interaction with small molecules and ions * *main.py* : The script to call to build or update the data releases

10.5 References

[1] [Richardson et al. (2008): "RNA backbone: consensus all-angle conformers and modular string nomenclature (an RNA Ontology Consortium contribution). RNA, 14(3):465-481](https://rnajournal.cshlp.org/content/14/3/465.short)

CHAPTER

ELEVEN

RNAGLIB.PREPARE_DATA

Functions to build data releases from raw PDBs.

`rnaglib.prepare_data.`**`filter_dot_edges`**`(`*graph*`)`

Remove edges with a '.' in the LW annotation. This happens in place.

Parameters
graph – networkx graph

`rnaglib.prepare_data.`**`filter_all`**`(`*graph_dir*, *output_dir*, *filters=['NR']*, *min_nodes=20*`)`

Apply filters to a graph dataset.

Parameters

- **graph_dir** – where to read graphs from
- **output_dir** – where to dump the graphs
- **filters** – list of which filters to apply ('NR', 'Ribo', 'NonRibo')
- **min_nodes** – skip graphs with fewer than *min_nodes* nodes (default=20)

`rnaglib.prepare_data.`**`one_rna_from_cif`**`(`*cif*`)`

Build 2.5d graph for one cif using dssr

Parameters
cif – path to mmCIF

Returns
2.5d graph

`rnaglib.prepare_data.`**`cif_to_graph`**`(`*cif*, *output_dir=None*, *min_nodes=20*, *return_graph=False*`)`

Build DDSR graphs for one mmCIF. Requires x3dna-dssr to be in PATH.

Parameters

- **cif** – path to CIF
- **output_dir** – where to dump
- **min_nodes** – smallest RNA (number of residue nodes)
- **return_graph** – Boolean to include the graph in the output

Returns
networkx graph of structure.

`rnaglib.prepare_data.`**`add_graph_annotations`**`(`*g*, *cif*`)`

Adds information at the graph level and on the small molecules partner of an RNA molecule

Parameters

- `g` – the nx graph created from dssr output
- `cif` – the path to a .mmcif file

Returns
the annotated graph, actually the graph is mutated in place

`rnaglib.prepare_data.`**`hariboss_filter`**(*lig*, *cif_dict*, *mass_lower_limit=160*, *mass_upper_limit=1000*)

Sorts ligands into ion / ligand / None
Returns ions for a specific list of ions, ligands if the hetatm has the right atoms and mass and None otherwise

Parameters

- `lig` – A biopython ligand residue object
- `cif_dict` – The output of the biopython MMCIF2DICT object
- `mass_lower_limit` –
- `mass_upper_limit` –

`rnaglib.prepare_data.`**`chop_all`**(*graph_path*, *dest*, *n_jobs=4*, *parallel=True*)

Chop and dump all the rglib graphs in the dataset.

Parameters

- `graph_path` – path to graphs for chopping
- `dest` – path where chopped graphs will be dumped

N_jobs
number of workers to use

Paralle
whether to use multiprocessing

`rnaglib.prepare_data.`**`annotate_all`**(*dump_path='../data/annotated/sample_v2'*, *graph_path='../data/chunks_nx'*, *parallel=True*, *do_hash=True*, *wl_hops=3*, *graphlet_size=1*, *re_annotate=False*)

Routine for all files in a folder

Parameters

- `dump_path` –
- `graph_path` –
- `parallel` –

Returns

CHAPTER TWELVE

RNAGLIB.DATA_LOADING

class `rnaglib.data_loading.`**`RNADataset`**(*data_path=None*, *version='1.0.0'*, *download_dir=None*, *redundancy='nr'*, *all_graphs=None*, *representations=()*, *rna_features=None*, *nt_features=None*, *bp_features=None*, *rna_targets=None*, *nt_targets=None*, *bp_targets=None*, *annotated=False*, *verbose=False*)

This class is the main object to hold the core RNA data annotations. The `RNAglibDataset.all_rnas` object is a generator networkx objects that hold all the annotations for each RNA in the dataset. You can also access individual RNAs on-disk with `RNAGlibDataset()[idx]` or `RNAGlibDataset().get_pdbid('1b23')`

Parameters

- **`representations`** – List of *rnaglib.Representation* objects to apply to each item.
- **`data_path`** – The path to the folder containing the graphs. If node_sim is not None, this data should be annotated
- **`version`** – Version of the dataset to use (default='0.0.0')
- **`redundancy`** – To use all graphs or just the non redundant set.
- **`all_graphs`** – In the given directory, one can choose to provide a list of graphs to use

`subset`(*list_of_graphs*)

Create another dataset with only the specified graphs

Parameters
`list_of_graphs` – a list of graph names

Returns
A graphdataset

`get_pdbid`(*pdbid*)

Grab an RNA by its pdbid

`get_nt_encoding`(*g*, *encode_feature=True*)

Get targets for graph g for every node get the attribute specified by self.node_target output a mapping of nodes to their targets

Parameters

- **`g`** – a nx graph
- **`encode_feature`** – A boolean as to whether this should encode the features or targets

Returns
A dict that maps nodes to encodings

`compute_dim`(*node_parser*)

Based on the encoding scheme, we can compute the shapes of the in and out tensors

Returns

`compute_features`(*rna_dict*)

Add 3 dictionaries to the *rna_dict* wich maps nts, edges, and the whole graph to a feature vector each. The final converter uses these to include the data in the framework-specific object.

`rnaglib.data_loading.get_loader`(*dataset*, *batch_size=5*, *num_workers=0*, *split=True*, *split_train=0.7*, *split_valid=0.85*, *verbose=False*, *framework='dgl'*)

Fetch a loader object for a given dataset.

Parameters

- **dataset** (`rnaglib.data_loading.RNADataset`) – Dataset for loading.
- **batch_size** (*int*) – number of items in batch
- **split** (*bool*) – whether to compute splits
- **split_train** (*float*) – proportion of dataset to keep for training
- **split_valid** (*float*) – proportion of dataset to keep for validation
- **verbose** (*bool*) – print updates
- **framework** (*str*) – learning framework to use ('dgl')

Returns

torch.utils.data.DataLoader

`class rnaglib.data_loading.Collater`(*dataset*)

Wrapper for collate function, so we can use different node similarities. We cannot use functools.partial as it is not picklable so incompatible with Pytorch loading

Initialize a Collater object.

Parameters

node_simfunc – A node comparison function as defined in kernels, to optionally return a pair-wise

comparison of the nodes in the batch :param max_size_kernel: If the node comparison is not None, optionnaly only return a pairwise comparison between a subset of all nodes, of size max_size_kernel :param hstack: If True, hstack point cloud return

Returns

a picklable python function that can be called on a batch by Pytorch loaders

`collate`(*samples*)

New format that iterates through the possible keys returned by get_item

The graphs are batched, the rings are compared with self.node_simfunc and the features are just put into a list. :param samples: :return: a dict

CHAPTER

THIRTEEN

RNAGLIB.REPRESENTATIONS

`class rnaglib.representations.`**`Representation`**

Callable object that accepts a raw RNA networkx object along with features and target vector representations and returns a representation of it (e.g. graph, voxel, point cloud)

`property name`

Just return the name of the representation

Returns
A string

`batch`(*samples*)

Batch a list of voxel samples

Parameters
`samples` – A list of the output from this representation

Returns
a batched version of it.

`class rnaglib.representations.`**`GraphRepresentation`**(*clean_edges=True*, *framework='nx'*, *edge_map={'B35': 19, 'B53': 0, 'cHH': 1, 'cHS': 2, 'cHW': 3, 'cSH': 4, 'cSS': 5, 'cSW': 6, 'cWH': 7, 'cWS': 8, 'cWW': 9, 'tHH': 10, 'tHS': 11, 'tHW': 12, 'tSH': 13, 'tSS': 14, 'tSW': 15, 'tWH': 16, 'tWS': 17, 'tWW': 18}*, *etype_key='LW'*, ***kwargs*)

Converts RNA into a graph

`property name`

Just return the name of the representation

Returns
A string

`batch`(*samples*)

Batch a list of graph samples

Parameters
`samples` – A list of the output from this representation

Returns
a batched version of it.

`class rnaglib.representations.`**`VoxelRepresentation`**(*spacing=2*, *padding=3*, *sigma=1.0*, ***kwargs*)

Converts RNA into a voxel based representation `property name`

Just return the name of the representation

Returns
A string

`batch`(*samples*)

Batch a list of voxel samples

Parameters
`samples` – A list of the output from this representation

Returns
a batched version of it.

`class rnaglib.representations.`**`PointCloudRepresentation`**(*hstack=True*, *sorted_nodes=True*, ***kwargs*)

Converts RNA into a point cloud based representation

`property name`

Just return the name of the representation

Returns
A string

`batch`(*samples*)

Batch a list of point cloud samples

Parameters
`samples` – A list of the output from this representation

Returns
a batched version of it.

`class rnaglib.representations.`**`RingRepresentation`**(*node_simfunc=None*, *max_size_kernel=None*, *hash_path=None*, ***kwargs*)

Converts RNA into a ring based representation

`property name`

Just return the name of the representation

Returns
A string

`batch`(*samples*)

Batch a list of ring samples

Parameters
`samples` – A list of the output from this representation

Returns
a batched version of it.

CHAPTER

FOURTEEN

RNAGLIB.DRAWING

14.1 Draw RNA

`rnaglib.drawing.rna_draw.`**`make_label`**`(`*s*`)`

`rnaglib.drawing.rna_draw.`**`process_axis`**`(`*axis*, *g*, *subtitle=None*, *highlight_edges=None*, *node_color=None*, *node_labels=None*, *node_ids=False*, *layout='spring'*, *label='LW'*`)`

Draw a graph on a given axis.

Parameters

- **`axis`** – matplotlib axis to draw on
- **`g`** – networkx graph to draw
- **`subtitle`** – string to use as a subtitle on this axis
- **`highlight_edges`** – A list of edges to highlight on the drawing
- **`node_color`** –
- **`node_labels`** –
- **`node_ids`** –
- **`label`** –

Returns

`rnaglib.drawing.rna_draw.`**`rna_draw`**`(`*g*, *title=''*, *node_ids=False*, *highlight_edges=None*, *node_labels=None*, *node_colors=None*, *num_clusters=None*, *pos=None*, *pos_offset=(0, 0)*, *scale=1*, *ax=None*, *show=False*, *alpha=1*, *save=False*, *node_size=250*, *fontsize=12*, *format='pdf'*, *seed=None*, *layout='circular'*`)`

Draw an RNA with the edge labels used by Leontis Westhof

Parameters

- **`nx_g`** –
- **`title`** –
- **`highlight_edges`** –
- **`node_colors`** –
- **`num_clusters`** –

Returns

`rnaglib.drawing.rna_draw.`**`rna_draw_pair`**`(`*graphs*, *subtitles=None*, *highlight_edges=None*, *node_colors=None*, *save=None*, *show=False*, *node_ids=False*`)`

Plot a line of plots of graphs along with a value for each graph. Useful for graph comparison vizualisation

Parameters

- **`graphs`** – iterable nx graphs
- **`estimated_value`** – iterable of values of comparison (optional)
- **`iihighlight_edges`** –
- **`node_colors`** – iterable of node colors

Returns

`rnaglib.drawing.rna_draw.`**`rna_draw_grid`**`(`*graphs*, *subtitles=None*, *highlight_edges=None*, *node_colors=None*, *row_labels=None*, *save=None*, *show=False*, *format='png'*, *grid_shape=None*`)`

Plot a line of plots of graphs along with a value for each graph. Useful for graph comparison vizualisation

Parameters

- **`graphs`** – list of lists containing nx graphs all lists must have the same dimension along axis 1. To skip a cell, add a None instead of graph.
- **`estimated_value`** – iterable of values of comparison (optional)
- **`highlight_edges`** –
- **`node_colors`** – iterable of node colors

Returns

14.2 Draw RNA layout

`rnaglib.drawing.rna_layout.`**`rescale_layout`**`(`*pos*, *scale=1*`)`

Return scaled position array to (-scale, scale) in all axes.

The function acts on NumPy arrays which hold position information. Each position is one row of the array. The dimension of the space equals the number of columns. Each coordinate in one column.

To rescale, the mean (center) is subtracted from each axis separately. Then all values are scaled so that the largest magnitude value from all axes equals *scale* (thus, the aspect ratio is preserved). The resulting NumPy Array is returned (order of rows unchanged).

Parameters

- **`pos`** (*`numpy array`*) – positions to be scaled. Each row is a position.
- **`scale`** (*`number (default: 1)`*) – The size of the resulting extent in all directions.

Returns
pos – scaled positions. Each row is a position.

Return type
numpy array

`rnaglib.drawing.rna_layout.`**`circular_layout`**`(`*G*, *scale=1*, *center=None*, *dim=2*`)`

Position nodes on a circle.

Parameters

- **G** (*NetworkX graph or list of nodes*) – A position will be assigned to every node in G.
- **scale** (*number (default: 1)*) – Scale factor for positions.
- **center** (*array-like or None*) – Coordinate pair around which to center the layout.
- **dim** (*int*) – Dimension of layout. If dim>2, the remaining dimensions are set to zero in the returned positions.

Returns
pos – A dictionary of positions keyed by node

Return type
dict

Examples

```
>>> G = nx.path_graph(4)
>>> pos = nx.circular_layout(G)
```

Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

CHAPTER

FIFTEEN

RNAGLIB.GED

This submodule is simply a collection of wrapper functions to invoke the above GED functions on our RNA graphs. Refer to the list below for function input parameters.

`rnaglib.ged.`**`ged`**`(`*g1*, *g2*, *roots=None*, *upper_bound=None*, *timeout=None*`)`

Compute the graph edit distance on RNA graphs (default weighting scheme is adapted to RNA)

Parameters

- **`g1`** – A networkx graph to compare
- **`g2`** – A networkx graph to compare
- **`roots`** – Whether to match rooted subgraphs (forced pairing betweeen these nodes)
- **`upper_bound`** – Maximum edit distance to consider.
- **`timeout`** – Time after which we want to stop

Returns

The GED value

`rnaglib.ged.`**`ged_approx`**`(`*g1*, *g2*, *upper_bound=None*`)`

Compute a faster version of the ged on RNA graphs

Parameters

- **`g1`** – A networkx graph to compare
- **`g2`** – A networkx graph to compare
- **`upper_bound`** – Maximum edit distance to consider.

Returns

The GED value

`rnaglib.ged.`**`ged`**`(`*g1*, *g2*, *roots=None*, *upper_bound=None*, *timeout=None*`)`

Compute the graph edit distance on RNA graphs (default weighting scheme is adapted to RNA)

Parameters

- **`g1`** – A networkx graph to compare
- **`g2`** – A networkx graph to compare
- **`roots`** – Whether to match rooted subgraphs (forced pairing betweeen these nodes)
- **`upper_bound`** – Maximum edit distance to consider.
- **`timeout`** – Time after which we want to stop

Returns
The GED value

`rnaglib.ged.ged_approx`(*g1*, *g2*, *upper_bound=None*)

Compute a faster version of the ged on RNA graphs

Parameters

- **g1** – A networkx graph to compare
- **g2** – A networkx graph to compare
- **upper_bound** – Maximum edit distance to consider.

Returns
The GED value

CHAPTER

SIXTEEN

RNAGLIB.KERNELS

16.1 Node Similarity

Functions for comparing node similarity.

class rnaglib.kernels.node_sim.**SimFunctionNode**(*method*, *depth*, *decay=0.5*, *idf=False*, *normalization=None*, *hash_init_path='/home/docs/checkouts/readthedocs.org/user_builds/rnag*

Bases: `object`

Factory object to compute all node similarities. These methods take as input an annotated pair of nodes and compare them.

These methods are detailed in the supplemental of the paper, but include five methods. These methods frequently rely on the hungarian algorithm, an algorithm that finds optimal matches according to a cost function.

Three of them compare the edges :

- R_1 compares the histograms of each ring, possibly with an idf weighting (to emphasize differences in rare edges)
- R_iso compares each ring with the best matching based on the isostericity values
- hungarian compares the whole annotation, with the rings being differentiated with an additional 'depth' field.

Then all the nodes are compared based on isostericity and this depth field.

Two of them compare the graphlets. The underlying idea is that just comparing lists of edges does not constraint the graph structure, while the assembling of graphlet does it more (exceptions can be found but for most graphs, knowing all graphlets at each depth enables recreating the graph) :

- R_graphlets works like R_iso except that the isostericity is replaced by the GED
- graphlet works like the hungarian except that the isostericity is replaced by the GED

Parameters

- **method** – a string that identifies which of these method to use
- **depth** – The depth to use in the annotations rings
- **decay** – When using rings comparison function, the weight decay of importance based on the depth (the

closest rings weigh more as they are central to the comparison) :param idf: Whether to use IDF weighting on the frequency of labels. :param normalization: We experiment with three normalization scheme, the basal one is just a division of the score by the maximum value, 'sqrt' denotes using the square root of the ratio as a power of the raw value and 'log' uses the log. The underlying idea is that we want to put more emphasis on the long matches than on just matching a few nodes :param hash_init_path: For the graphlets comparisons, we need to supply a hashing path to be able to store the values of ged and reuse them based on the hash.

`add_hashtable`(*hash_init_path*)

Parameters
`hash_init_path` – A string with the full path to a pickled hashtable

Returns
None, modify self.

`compare`(*rings1*, *rings2*)

Compares two nodes represented as their rings.

The edge list for the first hop (centered around a node) is None, so it gets skipped, when we say depth=3, we want rings[1:4], hence range(1, depth+1) Need to take this into account for normalization

(see class constructor)

param rings1
A list of rings at each depth. Rings contain a list of node, edge or graphlet information at a

given distance from a central node. :param rings2: Same as above for another node. :return: Normalized comparison score between the nodes

`normalize`(*unnormalized*, *max_score*)

We want our normalization to be more lenient to longer matches

Parameters

- `unnormalized` – a score in [0, max_score]
- `max_score` – the best possible matching score of the sequences we are given

Returns
a score in [0,1]

`get_length`(*ring1*, *ring2*, *graphlets=False*)

This is meant to return an adapted 'length' that represents the optimal score obtained when matching all the elements in the two rings at hands

param rings1
A list of rings at each depth. Rings contain a list of node, edge or graphlet information at a

given distance from a central node. :param rings2: Same as above for another node. :param graphlets: Whether we use graphlets instead of edges. Then no isostericity can be used to compute length

Returns
a float that represents the score of a perfect match

`static delta_indices_sim`(*i*, *j*, *distance=False*)

We need a scoring related to matching different depth nodes. Returns a positive score in [0,1]

Parameters

- `i` – pos of the first node
- `j` – pos of the second node

Returns
A normalized value of their distance (exp(abs(i-j))

`get_cost_nodes`(*node_i*, *node_j*, *bb=False*, *pos=False*)

Compare two nodes and returns a cost.

Returns a positive number that has to be negated for minimization

:param node_i : This either just contains a label to be compared with isostericity, or a tuple that also includes distance from the root node :param node_j : Same as above :param bb : Check if what is being compared is backbone (no isostericity then) :param pos: if pos is true, nodes are expected to be (edge label, distance from root) else just a edge label. pos is True when used from a comparison between nodes from different rings :return: the cost of matching those two nodes

`R_1`(*ring1*, *ring2*)

Compute R_1 function over lists of features by counting intersect and normalise by the number

Parameters

- **`ring1`** – list of features
- **`ring2`** – Same as above for other node

Returns
Score

`R_iso`(*list1*, *list2*)

Compute R_iso function over lists of features by matching each ring with the hungarian algorithm on the iso values

We do a separate computation for backbone.

Parameters

- **`list1`** – list of features
- **`list2`** – ”

Returns
Score

`hungarian`(*rings1*, *rings2*)

Compute hungarian function over lists of features by adding a depth field into each ring (based on its index in rings). Then we try to match all nodes together, to deal with bulges for instances.

We do a separate computation for backbone.

Parameters

- **`list1`** – list of features
- **`list2`** – ”

Returns
Score

`graphlet_cost_nodes`(*node_i*, *node_j*, *pos=False*, *similarity=False*)

Returns a node distance between nodes represented as graphlets
Compare two nodes and returns a cost.

Returns a positive number that has to be negated for minimization

:param node_i : This either just contains a label to be compared with isostericity, or a tuple that also includes distance from the root node :param node_j : Same as above :param pos: if pos is true, nodes are expected to be (graphlet, distance from root) else just a graphlet. pos is True when used from a comparison between nodes from different rings :return: the cost of matching those two nodes

`R_graphlets`(*ring1*, *ring2*)

Compute R_graphlets function over lists of features.

Parameters

- `ring1` – list of list of graphlets
- `ring2` – Same as above for other node

Returns
Score

`graphlet`(*rings1*, *rings2*)

This function performs an operation similar to the hungarian algorithm using ged between graphlets instead of isostericity.

We also add a distance to root node attribute to each graphlet and then match them optimally

Parameters

- `ring1` – list of list of graphlets
- `ring2` – Same as above for other node

Returns
Score

`rnaglib.kernels.node_sim.graph_edge_freqs`(*graphs*, *stop=False*)

Get IDF for each edge label over whole dataset. First get a total frequency dictionnary :{'CWW': 110, 'TWW': 23} Then compute IDF and returns the value.

Parameters

- `graphs` – The graphs over which to compute the frequencies, a list of nx graphs
- `stop` – Set to True for just doing it on a subset

Returns
A dict with the idf values.

`rnaglib.kernels.node_sim.pdist_list`(*rings*, *node_sim*)

Defines the block creation using a list of rings at the graph level (should also ultimately include trees) Creates a SIMILARITY matrix.

Parameters

- `rings` – a list of rings, dictionnaries {node : (nodelist, edgelist)}
- `node_sim` – the pairwise node comparison function

Returns
the upper triangle of a similarity matrix, in the form of a list

`rnaglib.kernels.node_sim.k_block_list`(*rings*, *node_sim*)

Defines the block creation using a list of rings at the graph level (should also ultimately include trees) Creates a SIMILARITY matrix.

Parameters

- `rings` – a list of rings, dictionnaries {node : (nodelist, edgelist)}
- `node_sim` – the pairwise node comparison function

Returns

A whole similarity matrix in the form of a numpy array that follows the order of rings

`rnaglib.kernels.node_sim.`**`simfunc_time`**(*simfuncs*, *graph_path*, *batches=1*, *batch_size=5*, *names=None*)

Do time benchmark on a list of simfunc.

Parameters

- **`simfuncs`** –
- **`graph_path`** –
- **`batches`** –
- **`batch_size`** –
- **`names`** –

Returns

CHAPTER

SEVENTEEN

RNAGLIB.LEARNING

17.1 Models

17.2 Learning

CHAPTER

EIGHTEEN

RNAGLIB.UTILS

General utilities for handling RNA structures and graphs.

rnaglib.utils.**download_graphs**(*redundancy='nr'*, *version='1.0.0'*, *annotated=False*, *chop=False*, *overwrite=False*, *data_root=None*, *verbose=False*)

Based on the options, get the right data from the latest release and put it in download_dir.

Parameters

- **redundancy** – Whether to include all RNAs or just a non-redundant set as defined by BGSU
- **annotated** – Whether to include graphlet annotations in the graphs. This will also create a hashing directory and table
- **overwrite** – To overwrite existing data
- **download_dir** – Where to save this data. Defaults to ~/.rnaglib/

Returns

the path of the data along with its hashing.

rnaglib.utils.**get_rna_list**(*nr_only=False*)

Fetch a list of PDBs containing RNA from RCSB API.

rnaglib.utils.**graph_from_pdbid**(*pdbid*, *graph_dir=None*, *version='1.0.0'*, *annotated=False*, *chop=False*, *redundancy='nr'*, *graph_format='json'*)

Fetch an annotated graph with a PDBID.

Parameters

- **pdbid** – PDB id to fetch
- **graph_dir** – path containing annotated graphs
- **graph_format** – which format to load (JSON, or networkx)

rnaglib.utils.**load_graph**(*filename*)

This is a utility function that supports loading from json or pickle Sometimes, the pickle also contains rings in the form of a node dict, in which case the rings are added into the graph

Parameters

filename – json or pickle filename

Returns

networkx DiGraph object

rnaglib.utils.**dump_json**(*filename*, *graph*)

Just a shortcut to dump a json graph more compactly

Parameters

- **filename** – The dump name
- **graph** – The graph to dump

`rnaglib.utils.`**`load_json`**`(`*filename*`)`

Just a shortcut to load a json graph more compactly

Parameters
filename – The dump name

Returns
The loaded graph

`rnaglib.utils.`**`reorder_nodes`**`(`*g*`)`

Reorder nodes in graph

Parameters
g (*networkx.DiGraph*) – Pass a graph for node reordering.

Return h
(nx DiGraph)

`rnaglib.utils.`**`update_RNApdb`**`(`*pdir*, *nr_only=True*`)`

Download a list of RNA containing structures from the PDB overwrite exising files

Parameters
pdbdir – path containing downloaded PDBs

Returns rna
list of PDBIDs that were fetched.

`rnaglib.utils.`**`fix_buggy_edges`**`(`*graph*, *label='LW'*, *strategy='remove'*, *edge_map={'B35': 19, 'B53': 0, 'cHH': 1, 'cHS': 2, 'cHW': 3, 'cSH': 4, 'cSS': 5, 'cSW': 6, 'cWH': 7, 'cWS': 8, 'cWW': 9, 'tHH': 10, 'tHS': 11, 'tHW': 12, 'tSH': 13, 'tSS': 14, 'tSW': 15, 'tWH': 16, 'tWS': 17, 'tWW': 18}*`)`

Sometimes some edges have weird names such as t.W representing a fuzziness. We just remove those as they don't deliver a good information

Parameters

- **graph** –
- **strategy** – How to deal with it : for now just remove them.

In the future maybe add an edge type in the edge map ? :return:

`rnaglib.utils.`**`dangle_trim`**`(`*graph*`)`

Recursively remove dangling nodes from graph, with in place modification

Parameters
graph – Nx graph

Returns
trimmed graph

`rnaglib.utils.`**`gap_fill`**`(`*original_graph*, *graph_to_expand*`)`

If we subgraphed, get rid of all degree 1 nodes by completing them with one more hop

Parameters

- **original_graph** – nx graph
- **graph_to_expand** – nx graph that needs to be expanded to fix dangles

Returns
the expanded graph

`rnaglib.utils.`**`extract_graphlet`**(*graph*, *n*, *size=1*, *label='LW'*)

Small util to extract a graphlet around a node

Parameters

- **`graph`** – Nx graph
- **`n`** – a node in the graph
- **`size`** – The depth to consider

Returns
The graphlet as a copy

rnaglib.utils.**build_node_feature_parser**(asked_features=None, node_feature_map={'C5prime_xyz': <rnaglib.utils.feature_maps.ListEncoder object>, 'Dp': <rnaglib.utils.feature_maps.FloatEncoder object>, 'P_xyz': <rnaglib.utils.feature_maps.ListEncoder object>, 'alpha': <rnaglib.utils.feature_maps.FloatEncoder object>, 'amplitude': <rnaglib.utils.feature_maps.FloatEncoder object>, 'bb_type': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'beta': <rnaglib.utils.feature_maps.FloatEncoder object>, 'bin': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'binding_ion': <rnaglib.utils.feature_maps.BoolEncoder object>, 'binding_protein': <rnaglib.utils.feature_maps.BoolEncoder object>, 'binding_protein_Rdst': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Rx': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Ry': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Rz': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Tdst': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Tx': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Ty': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_Tz': <rnaglib.utils.feature_maps.FloatEncoder object>, 'binding_protein_aa': None, 'binding_protein_id': None, 'binding_protein_nt': None, 'binding_protein_nt-aa': None, 'binding_small-molecule': <rnaglib.utils.feature_maps.BoolEncoder object>, 'chain_name': None, 'chi': <rnaglib.utils.feature_maps.FloatEncoder object>, 'cluster': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'dbn': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'delta': <rnaglib.utils.feature_maps.FloatEncoder object>, 'epsilon': <rnaglib.utils.feature_maps.FloatEncoder object>, 'epsilon_zeta': <rnaglib.utils.feature_maps.FloatEncoder object>, 'eta': <rnaglib.utils.feature_maps.FloatEncoder object>, 'eta_base': <rnaglib.utils.feature_maps.FloatEncoder object>, 'eta_prime': <rnaglib.utils.feature_maps.FloatEncoder object>, 'filter_rmsd': <rnaglib.utils.feature_maps.FloatEncoder object>, 'form': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'frame_origin': <rnaglib.utils.feature_maps.ListEncoder object>, 'frame_quaternion': <rnaglib.utils.feature_maps.ListEncoder object>, 'frame_rmsd': <rnaglib.utils.feature_maps.FloatEncoder object>, 'frame_x_axis': <rnaglib.utils.feature_maps.ListEncoder object>, 'frame_y_axis': <rnaglib.utils.feature_maps.ListEncoder object>, 'frame_z_axis': <rnaglib.utils.feature_maps.ListEncoder object>, 'gamma': <rnaglib.utils.feature_maps.FloatEncoder object>, 'glyco_bond': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'index': None, 'index_chain': None, 'is_broken': <rnaglib.utils.feature_maps.BoolEncoder object>, 'is_modified': <rnaglib.utils.feature_maps.BoolEncoder object>, 'nt_code': <rnaglib.utils.feature_maps.OneHotEncoder object>, 'nt_id': None, 'nt_name': None, 'nt_resnum': None, 'nt_type': None, 'phase_angle': <rnaglib.utils.feature_maps.FloatEncoder object>, 'puckering':

This function will load the predefined feature maps available globally. Then for each of the features in 'asked feature', it will return an encoder object for each of the asked features in the form of a dict {asked_feature : EncoderObject}

If some keys don't exist, will raise an Error. However if some keys are present but problematic, this will just cause a printing of the problematic keys :param asked_features: A list of string keys that are present in the encoder :return: A dict {asked_feature : EncoderObject}

`rnaglib.utils.`**`build_hash_table`**(*graph_dir*, *hasher*, *graphlets=True*, *max_graphs=0*, *graphlet_size=1*, *mode='count'*, *label='LW'*, *directed=True*)

Iterates over nodes of the graphs in graph dir and fill a hash table with their graphlets hashes

Parameters

- **`graph_dir`** –
- **`hasher`** –
- **`graphlets`** –
- **`max_graphs`** –
- **`graphlet_size`** –
- **`mode`** –
- **`label`** –

Returns

RNAGLIB.CONFIG

19.1 Build Isostericity Matrix

`rnaglib.config.build_iso_mat.`**`get_undirected_iso`**`(`*bpa*, *bpb*`)`

Given two directed edges, get the values from the undirected isostericity matrix

Parameters

- **bpa** (*str*) – LW edge code
- **bpb** (*str*) – LW edge code

Returns
isostericty value

:rtype float

`rnaglib.config.build_iso_mat.`**`build_iso`**`()`

This function builds a directed isostericity matrix

The heuristic is as follows : - It has a diagonal of ones : max similarity is self - Backbone is set aside, and has a little cost for reversing the direction - Different edges types are computed to have the associated undirected isostericity value

Returns
A np matrix that yields the isostericity values, ordered as EDGE_MAP

CHAPTER

TWENTY

RNAGLIB OFFICIAL DOCUMENTATION

`RNAGlib` (RNA Geometric Library) is a Python package for studying models of RNA 3D structures.

20.1 Core Features

- Quick access to all available RNA 3D structures with annotations
- Rich functionality for 2.5D RNA graphs, point clouds, and voxels
- RNA graph visualization
- Machine Learning benchmarking tasks

20.2 Get started with RNAGlib

- *Install*
- *Quickstart*
- *Learn about RNA 2.5D Graphs*
- *Annotation reference*

20.3 Tutorials

- *Working with 2.5D graphs datasets*
- *Training machine learning models*

20.4 Package Structure

- *rnaglib.prepare_data*: processes raw PDB structures and builds a database of 2.5D graphs with full structural annotation
- *rnaglib.data_loading*: custom PyTorch dataloader and dataset implementations
- *rnaglib.representations*: graph, voxel, point cloud representations
- *rnaglib.learning*: learning routines and pre-built GCN models for the easiest use of the package.
- *rnaglib.drawing*: utilities for visualizing 2.5D graphs

- *rnaglib.ged*: custom graph similarity functions
- *rnaglib.kernels*: custom local neighbourhood similarity functions

20.5 Source Code and Contact

- RNAglib homepage.
- Source Code.
- Contact rnaglib@cs.mcgill.ca

  Associated Repositories

RNAmigos : a research paper published in Nucleic Acid Research that demonstrates the usefulness of 2.5D graphs for machine learning tasks, exemplified onto the drug discovery application.

VeRNAl : a research paper published in Bioinformatics that uses learnt vector representations of RNA subgraphs to mine structural motifs in RNA.

20.6 References

1. Leontis, N. B., & Zirbel, C. L. (2012). Nonredundant 3D Structure Datasets for RNA Knowledge Extraction and Benchmarking. In RNA 3D Structure Analysis and Prediction N. Leontis & E. Westhof (Eds.), (Vol. 27, pp. 281–298). Springer Berlin Heidelberg. doi:10.1007/978-3-642-25740-7_13

20.7 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX