PyGraphistry Documentation

Graphistry, Inc.

Apr 07, 2023

CONTENTS

PyGraphistry is a Python visual graph AI library to extract, transform, analyze, model, and visualize big graphs, and especially alongside Graphistry end-to-end GPU server sessions. Installing optional graphistry[ai] dependencies adds graph autoML, including automatic feature engineering, UMAP, and graph neural net support. Combined, PyGraphistry reduces your time to graph for going from raw data to visualizations and AI models down to three lines of code. Here in our docstrings you can find useful packages, modules, and commands to maximize your graph AI experience with PyGraphistry. In the navbar you can find an overview of all the packages and modules we provided and a few useful highlighted ones as well. You can search for them on our Search page. For a full tutorial, refer to our PyGraphistry repo.

For self-hosting and access to a free API key, refer to our Graphistry Hub.

CHAPTER

ONE

PLOTTER

1.1 Plotter Base

class graphistry.PlotterBase.**PlotterBase**(**args*, ***kwargs*)

Bases: graphistry.Plottable.Plottable

Graph plotting class.

Created using `Graphistry.bind()`.

Chained calls successively add data and visual encodings, and end with a plot call.

To streamline reuse and replayable notebooks, Plotter manipulations are immutable. Each chained call returns a new instance that derives from the previous one. The old plotter or the new one can then be used to create different graphs.

When using memoization, for .register(api=3) sessions with .plot(memoize=True), Pandas/cudf arrow coercions are memoized, and file uploads are skipped on same-hash dataframes.

The class supports convenience methods for mixing calls across Pandas, NetworkX, and IGraph.

add_style(*fg=None*, *bg=None*, *page=None*, *logo=None*)

Set general visual styles

See .bind() and .settings(url_params={}) for additional styling options, and style() for another way to set the same attributes.

To facilitate reuse and replayable notebooks, the addStyle() call is chainable. Invocation does not effect the old style: it instead returns a new Plotter instance with the new styles added to the existing ones. Both the old and new styles can then be used for different graphs.

addStyle() will extend the existing style settings, while style() will replace any in the same group

Parameters

- **fg** (*dict*) – Dictionary {'blendMode': str} of any valid CSS blend mode
- **bg** (*dict*) – Nested dictionary of page background properties. {'color': str, 'gradient': {'kind': str, 'position': str, 'stops': list }, 'image': { 'url': str, 'width': int, 'height': int, 'blendMode': str }
- **logo** (*dict*) – Nested dictionary of logo properties. { 'url': str, 'autoInvert': bool, 'position': str, 'dimensions': { 'maxWidth': int, 'maxHeight': int }, 'crop': { 'top': int, 'left': int, 'bottom': int, 'right': int }, 'padding': { 'top': int, 'left': int, 'bottom': int, 'right': int}, 'style': str}
- **page** (*dict*) – Dictionary of page metadata settings. { 'favicon': str, 'title': str }

Returns Plotter

Return type Plotter

Example: Chained merge - results in color, blendMode, and url being set

```
g2 =  g.addStyle(bg={'color': 'black'}, fg={'blendMode': 'screen'})
g3 = g2.addStyle(bg={'image': {'url': 'http://site.com/watermark.png'}})
```

Example: Overwrite - results in blendMode multiply

```
g2 =  g.addStyle(fg={'blendMode': 'screen'})
g3 = g2.addStyle(fg={'blendMode': 'multiply'})
```

Example: Gradient background

```
g.addStyle(bg={'gradient': {'kind': 'linear', 'position': 45, 'stops': [[
↪'rgb(0,0,0)', '0%'], ['rgb(255,255,255)', '100%']]}})
```

Example: Page settings

```
g.addStyle(page={'title': 'Site - {{ name }}', 'favicon': 'http://site.
↪com/logo.ico'})
```

bind(*source=None*, *destination=None*, *node=None*, *edge=None*, *edge_title=None*, *edge_label=None*, *edge_color=None*, *edge_weight=None*, *edge_size=None*, *edge_opacity=None*, *edge_icon=None*, *edge_source_color=None*, *edge_destination_color=None*, *point_title=None*, *point_label=None*, *point_color=None*, *point_weight=None*, *point_size=None*, *point_opacity=None*, *point_icon=None*, *point_x=None*, *point_y=None*)

Relate data attributes to graph structure and visual representation. To facilitate reuse and replayable notebooks, the binding call is chainable. Invocation does not effect the old binding: it instead returns a new Plotter instance with the new bindings added to the existing ones. Both the old and new bindings can then be used for different graphs.

Parameters

- **source** (*str*) – Attribute containing an edge's source ID
- **destination** (*str*) – Attribute containing an edge's destination ID
- **node** (*str*) – Attribute containing a node's ID
- **edge** (*str*) – Attribute containing an edge's ID
- **edge_title** (*str*) – Attribute overriding edge's minimized label text. By default, the edge source and destination is used.
- **edge_label** (*str*) – Attribute overriding edge's expanded label text. By default, scrollable list of attribute/value mappings.
- **edge_color** (*str*) – Attribute overriding edge's color. rgba (int64) or int32 palette index, see palette definitions for values. Based on Color Brewer.
- **edge_source_color** (*str*) – Attribute overriding edge's source color if no edge_color, as an rgba int64 value.
- **edge_destination_color** (*str*) – Attribute overriding edge's destination color if no edge_color, as an rgba int64 value.
- **edge_weight** (*str*) – Attribute overriding edge weight. Default is 1. Advanced layout controls will relayout edges based on this value.
- **point_title** (*str*) – Attribute overriding node's minimized label text. By default, the node ID is used.

- **point_label** (*str*) – Attribute overriding node's expanded label text. By default, scrollable list of attribute/value mappings.
- **point_color** (*str*) – Attribute overriding node's color.rgba (int64) or int32 palette index, see palette definitions for values. Based on Color Brewer.
- **point_size** (*str*) – Attribute overriding node's size. By default, uses the node degree. The visualization will normalize point sizes and adjust dynamically using semantic zoom.
- **point_x** (*str*) – Attribute overriding node's initial x position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **point_y** (*str*) – Attribute overriding node's initial y position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout

Returns Plotter

Return type Plotter

Example: Minimal

```
import graphistry
g = graphistry.bind()
g = g.bind(source='src', destination='dst')
```

Example: Node colors

```
import graphistry
g = graphistry.bind()
g = g.bind(source='src', destination='dst',
           node='id', point_color='color')
```

Example: Chaining

```
import graphistry
g = graphistry.bind(source='src', destination='dst', node='id')

g1 = g.bind(point_color='color1', point_size='size1')

g.bind(point_color='color1b')

g2a = g1.bind(point_color='color2a')
g2b = g1.bind(point_color='color2b', point_size='size2b')

g3a = g2a.bind(point_size='size3a')
g3b = g2b.bind(point_size='size3b')
```

In the above **Chaining** example, all bindings use src/dst/id. Colors and sizes bind to:

```
g: default/default
g1: color1/size1
g2a: color2a/size1
g2b: color2b/size2b
g3a: color2a/size3a
g3b: color2b/size3b
```

bolt(*driver*)

compute_cugraph(*alg*, *out_col=None*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*)

Run cugraph algorithm on graph. For algorithm parameters, see cuGraph docs.

Parameters

- **alg** (*str*) – algorithm name
- **out_col** (*Optional[str]*) – node table output column name, defaults to alg param
- **params** (*dict*) – algorithm parameters passed to cuGraph as kwargs
- **kind** (*CuGraphKind*) – kind of cugraph to use
- **directed** (*bool*) – whether graph is directed
- **G** (*Optional[cugraph.Graph]*) – cugraph graph to use; if None, use self

Returns Plottable

Return type Plottable

Example: Pagerank

```
g2 = g.compute_cugraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

Example: Katz centrality with rename

```
g2 = g.compute_cugraph('katz_centrality', out_col='katz_centrality_renamed
↪')
assert 'katz_centrality_renamed' in g2._nodes.columns
```

Example: Pass params to cugraph

```
g2 = g.compute_cugraph('k_truss', params={'k': 2})
assert 'k_truss' in g2._nodes.columns
```

compute_igraph(*alg*, *out_col=None*, *directed=None*, *use_vids=False*, *params={}*)

Enrich or replace graph using igraph methods

Parameters

- **alg** (*str*) – Name of an igraph.Graph method like *pagerank*
- **out_col** (*Optional[str]*) – For algorithms that generate a node attribute column, *out_col* is the desired output column name. When *None*, use the algorithm's name. (default None)
- **directed** (*Optional[bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – During the to_igraph conversion, whether to interpret IDs as igraph vertex IDs (non-negative integers) or arbitrary values (False, default)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method

Returns Plotter

Return type Plotter

Example: Pagerank

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
```

(continues on next page)

(continued from previous page)

```
g2 = g.compute_igraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

Example: Pagerank with custom name

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank', out_col='my_pr')
assert 'my_pr' in g2._nodes.columns
```

Example: Pagerank on an undirected

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank', directed=False)
assert 'pagerank' in g2._nodes.columns
```

Example: Pagerank with custom parameters

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank', params={'damping': 0.85})
assert 'pagerank' in g2._nodes.columns
```

copy()

Return type Plottable

cypher(*query*, *params={}*)

description(*description*)

Upload description

Parameters **description** (*str*) – Upload description

edges(*edges*, *source=None*, *destination=None*, *edge=None*, **args*, ***kwargs*)

Specify edge list data and associated edge attribute values. If a callable, will be called with current Plotter and whatever positional+named arguments

Parameters **edges** (*Pandas dataframe, NetworkX graph, or IGraph graph*) – Edges and their attributes, or transform from Plotter to edges

Returns Plotter

Return type Plotter

Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .edges(df)
    .plot()
```

Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0], 'id': [0, 1, 2]})
graphistry
    .bind(source='src', destination='dst', edge='id')
    .edges(df)
    .plot()
```

Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .edges(df, 'src', 'dst')
    .plot()
```

Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0], 'id': [0, 1, 2]})
graphistry
    .edges(df, 'src', 'dst', 'id')
    .plot()
```

Example

```
import graphistry

def sample_edges(g, n):
    return g._edges.sample(n)

df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})

graphistry
    .edges(df, 'src', 'dst')
    .edges(sample_edges, n=2)
    .edges(sample_edges, None, None, None, 2)  # equivalent
    .plot()
```

encode_axis(*rows=[]*)

Render radial and linear axes with optional labels

Parameters **rows** – List of rows - { label: Optional[str],?r: float, ?x: float, ?y: float, ?internal: true, ?external: true, ?space: true }

Returns Plotter

Return type Plotter

Example: Several radial axes

```
g.encode_axis([
  {'r': 14, 'external': True, 'label': 'outermost'},
  {'r': 12, 'external': True},
  {'r': 10, 'space': True},
  {'r': 8, 'space': True},
  {'r': 6, 'internal': True},
  {'r': 4, 'space': True},
```

(continues on next page)

(continued from previous page)

```
  {'r': 2, 'space': True, 'label': 'innermost'}
])
```

Example: Several horizontal axes

```
g.encode_axis([
  {"label": "a",  "y": 2, "internal": True },
  {"label": "b",  "y": 40, "external": True, "width": 20, "bounds": {"min
↪": 40, "max": 400}},
])
```

encode_edge_badge(*column*, *position='TopRight'*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *color=None*, *bg=None*, *fg=None*, *for_current=False*, *for_default=True*, *as_text=None*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

encode_edge_color(*column*, *palette=None*, *as_categorical=None*, *as_continuous=None*, *categorical_mapping=None*, *default_mapping=None*, *for_default=True*, *for_current=False*)

Set edge color with more control than bind()

Parameters

- **column** (*str*) – Data column name
- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: [“black, “#FF0”, “rgb(255,255,255)”]. Used as a gradient for continuous and round-robin for categorical.
- **as_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {“car”: “red”, “truck”: #000”}
- **default_mapping** (*Optional[str]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=”gray”.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

Returns Plotter

Return type Plotter

Example: See encode_point_color

encode_edge_icon(*column*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

Set edge icon with more control than bind() Values from Font Awesome 4 such as “laptop”: https://fontawesome.com/v4.7.0/icons/ , image URLs (http://...), and data URIs (data:...). When as_text=True is enabled, values are instead interpreted as raw strings.

Parameters

- **column** (*str*) – Data column name
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": 'car', "ford": 'truck'}
- **default_mapping** (*Optional[Union[int,float]]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=50.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)

Returns Plotter

Return type Plotter

Example: Set a string column of icons for the edge icons, same as bind(edge_icon='my_column')

```
g2a = g.encode_edge_icon('my_icons_column')
```

Example: Map specific values to specific icons, including with a default

```
g2a = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'})
g2b = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'}, default_mapping='question')
```

Example: Map countries to abbreviations

```
g2a = g.encode_edge_icon('country_abbrev', as_text=True)
g2b = g.encode_edge_icon('country', as_text=True, categorical_mapping={
↪'England': 'UK', 'America': 'US'}, default_mapping='')
```

Example: Border

```
g2b = g.encode_edge_icon('country', border={'width': 3, color: 'black',
↪'stroke': 'dashed'}, 'categorical_mapping={'England': 'UK', 'America':
↪'US'})
```

encode_point_badge(*column*, *position='TopRight'*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *color=None*, *bg=None*, *fg=None*, *for_current=False*, *for_default=True*, *as_text=None*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

encode_point_color(*column*, *palette=None*, *as_categorical=None*, *as_continuous=None*, *categorical_mapping=None*, *default_mapping=None*, *for_default=True*, *for_current=False*)

Set point color with more control than bind()

Parameters

- **column** (*str*) – Data column name

- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: [“black, “#FF0”, “rgb(255,255,255)”]. Used as a gradient for continuous and round-robin for categorical.
- **as_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {“car”: “red”, “truck”: #000”}
- **default_mapping** (*Optional[str]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=”gray”.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

Returns Plotter

Return type Plotter

Example: Set a palette-valued column for the color, same as bind(point_color='my_column')

```
g2a = g.encode_point_color('my_int32_palette_column')
g2b = g.encode_point_color('my_int64_rgb_column')
```

Example: Set a cold-to-hot gradient of along the spectrum blue, yellow, red

```
g2 = g.encode_point_color('my_numeric_col', palette=["blue", "yellow",
↪"red"], as_continuous=True)
```

Example: Round-robin sample from 5 colors in hex format

```
g2 = g.encode_point_color('my_distinctly_valued_col', palette=["#000", "
↪#00F", "#0F0", "#0FF", "#FFF"], as_categorical=True)
```

Example: Map specific values to specific colors, including with a default

```
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪ 'ford': 'blue'})
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪ 'ford': 'blue'}, default_mapping='gray')
```

encode_point_icon(*column*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

Set node icon with more control than bind(). Values from Font Awesome 4 such as “laptop”: https://fontawesome.com/v4.7.0/icons/ , image URLs (http://...), and data URIs (data:...). When as_text=True is enabled, values are instead interpreted as raw strings.

Parameters

- **column** (*str*) – Data column name

- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": 'car', "ford": 'truck'}
- **default_mapping** (*Optional[Union[int,float]]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=50.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)
- **blend_mode** (*Optional[str]*) – CSS blend mode
- **style** (*Optional[dict]*) – CSS filter properties - opacity, saturation, luminosity, grayscale, and more
- **border** (*Optional[dict]*) – Border properties - 'width', 'color', and 'storke'

Returns Plotter

Return type Plotter

Example: Set a string column of icons for the point icons, same as bind(point_icon='my_column')

```
g2a = g.encode_point_icon('my_icons_column')
```

Example: Map specific values to specific icons, including with a default

```
g2a = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'})
g2b = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'}, default_mapping='question')
```

Example: Map countries to abbreviations

```
g2b = g.encode_point_icon('country_abbrev', as_text=True)
g2b = g.encode_point_icon('country', as_text=True, categorical_mapping={
↪'England': 'UK', 'America': 'US'}, default_mapping='')
```

Example: Border

```
g2b = g.encode_point_icon('country', border={'width': 3, color: 'black',
↪'stroke': 'dashed'}, 'categorical_mapping={'England': 'UK', 'America':
↪'US'})
```

encode_point_size(*column*, *categorical_mapping=None*, *default_mapping=None*, *for_default=True*, *for_current=False*)

Set point size with more control than bind()

Parameters

- **column** (*str*) – Data column name
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to numbers. Ex: {"car": 100, "truck": 200}

- **default_mapping** (*Optional[Union[int,float]]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=50.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

Returns Plotter

Return type Plotter

Example: Set a numerically-valued column for the size, same as bind(point_size='my_column')

```
g2a = g.encode_point_size('my_numeric_column')
```

Example: Map specific values to specific colors, including with a default

```
g2a = g.encode_point_size('brands', categorical_mapping={'toyota': 100,
↪'ford': 200})
g2b = g.encode_point_size('brands', categorical_mapping={'toyota': 100,
↪'ford': 200}, default_mapping=50)
```

from_cugraph(*G*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

If bound IDs, use the same IDs in the returned graph.

If non-empty nodes/edges, instead of returning G's topology, use existing topology and merge in G's attributes

Parameters

- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **load_nodes** (bool) –
- **load_edges** (bool) –
- **merge_if_existing** (bool) –

from_igraph(*ig*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

Convert igraph object into Plotter

If base g has _node, _source, _destination definitions, use them

When merge_if_existing with preexisting nodes/edges df and shapes match ig, combine attributes

For merge_if_existing to work with edges, must set g._edge and have corresponding edge index attribute in igraph.Graph

Parameters

- **ig** (*igraph*) – Source igraph object
- **node_attributes** (*Optional[List[str]]*) – Subset of node attributes to load; None means all (default)
- **edge_attributes** (*Optional[List[str]]*) – Subset of edge attributes to load; None means all (default)

- **load_nodes** (*bool*) – Whether to load nodes dataframe (default True)
- **load_edges** (*bool*) – Whether to load edges dataframe (default True)
- **merge_if_existing** – Whether to merge with existing node/edge dataframes (default True)
- **merge_if_existing** – bool

Returns Plotter

Example: Convert from igraph, including all node/edge properties

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e
↪'], 'v': [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degrees()
assert 'degree' in g._nodes.columns
g2 = g.from_igraph(g.to_igraph())
assert len(g2._nodes.columns) == len(g._nodes.columns)
```

Example: Enrich from igraph, but only load in 1 node attribute

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e
↪'], 'v': [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degree()
assert 'degree' in g._nodes
ig = g.to_igraph(include_nodes=False)
assert 'degree' not in ig.vs
ig.vs['pagerank'] = ig.pagerank()
g2 = g.from_igraph(ig, load_edges=False, node_attributes=[g._node,
↪'pagerank'])
assert 'pagerank' in g2._nodes
asssert 'degree' in g2._nodes
```

graph(*ig*)

Specify the node and edge data.

Parameters **ig** (*Any*) – NetworkX graph or an IGraph graph with node and edge attributes.

Returns Plotter

Return type Plotter

gsql(*query*, *bindings={}*, *dry_run=False*)

Run Tigergraph query in interpreted mode and return transformed Plottable

param query Code to run

type query str

param bindings Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList

type bindings Optional[dict]

param dry_run Return target URL without running

type dry_run bool

returns Plotter

rtype Plotter

Example: Minimal

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do

    Start = select t from Start:s-(:e)-:t
    where e.goUpper == TRUE
    accum @@edgeList += e
    having t.type != "Service";
    end;

    print @@edgeList;
}
""").plot()
```

Example: Full

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do

    Start = select t from Start:s-(:e)-:t
    where e.goUpper == TRUE
    accum @@edgeList += e
    having t.type != "Service";
    end;

    print @@my_edge_list;
}
""", {'edges': 'my_edge_list'}).plot()
```

gsql_endpoint(*method_name*, *args={}*, *bindings={}*, *db=None*, *dry_run=False*)

Invoke Tigergraph stored procedure at a user-definend endpoint and return transformed Plottable

Parameters

- **method_name** (*str*) – Stored procedure name
- **args** (*Optional[dict]*) – Named endpoint arguments
- **bindings** (*Optional[dict]*) – Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList
- **db** (*Optional[str]*) – Name of the database, defaults to value set in .tigergraph(...)
- **dry_run** (*bool*) – Return target URL without running

Returns Plotter

Return type Plotter

Example: Minimal

```
import graphistry
tg = graphistry.tigergraph(db='my_db')
tg.gsql_endpoint('neighbors').plot()
```

Example: Full

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql_endpoint('neighbors', {'k': 2}, {'edges': 'my_edge_list'}, 'my_db
↪').plot()
```

Example: Read data

```
import graphistry
tg = graphistry.tigergraph()
out = tg.gsql_endpoint('neighbors')
(nodes_df, edges_df) = (out._nodes, out._edges)
```

hypergraph(*raw_events*, *entity_types=None*, *opts={}*, *drop_na=True*, *drop_edge_attrs=False*, *verbose=True*, *direct=False*, *engine='pandas'*, *npartitions=None*, *chunksize=None*)

Transform a dataframe into a hypergraph.

Parameters

- **raw_events** (*pandas.DataFrame*) – Dataframe to transform (pandas or cudf).
- **entity_types** (*Optional[list]*) – Columns (strings) to turn into nodes, None signifies all
- **opts** (*dict*) – See below
- **drop_edge_attrs** (*bool*) – Whether to include each row's attributes on its edges, defaults to False (include)
- **verbose** (*bool*) – Whether to print size information
- **direct** (*bool*) – Omit hypernode and instead strongly connect nodes in an event
- **engine** (*bool*) – String (pandas, cudf, ...) for engine to use
- **npartitions** (*Optional[int]*) – For distributed engines, how many coarse-grained pieces to split events into
- **chunksize** (*Optional[int]*) – For distributed engines, split events after chunksize rows

Create a graph out of the dataframe, and return the graph components as dataframes, and the renderable result Plotter. Hypergraphs reveal relationships between rows and between column values. This transform is useful for lists of events, samples, relationships, and other structured high-dimensional data.

Specify local compute engine by passing *engine='pandas'*, 'cudf', 'dask', 'dask_cudf' (default: 'pandas'). If events are not in that engine's format, they will be converted into it.

The transform creates a node for every unique value in the entity_types columns (default: all columns). If direct=False (default), every row is also turned into a node. Edges are added to connect every table cell to its originating row's node, or if direct=True, to the other nodes from the same row. Nodes are given the attribute 'type' corresponding to the originating column name, or in the case of a row, 'EventID'. Options further control the transform, such column category definitions for controlling whether values reocurring in different columns should be treated as one node, or whether to only draw edges between certain column type pairs.

Consider a list of events. Each row represents a distinct event, and each column some metadata about an event. If multiple events have common metadata, they will be transitively connected through those metadata values. The layout algorithm will try to cluster the events together. Conversely, if an event has unique metadata, the unique metadata will turn into nodes that only have connections to the event node, and the clustering algorithm will cause them to form a ring around the event node.

Best practice is to set EVENTID to a row's unique ID, SKIP to all non-categorical columns (or entity_types to all categorical columns), and CATEGORY to group columns with the same kinds of values.

To prevent creating nodes for null values, set drop_na=True. Some dataframe engines may have undesirable null handling, and recommend replacing None values with np.nan .

The optional `opts={...}` configuration options are:

- 'EVENTID': Column name to inspect for a row ID. By default, uses the row index.
- 'CATEGORIES': Dictionary mapping a category name to inhabiting columns. E.g., {'IP': ['srcAddress', 'dstAddress']}. If the same IP appears in both columns, this makes the transform generate one node for it, instead of one for each column.
- 'DELIM': When creating node IDs, defines the separator used between the column name and node value
- 'SKIP': List of column names to not turn into nodes. For example, dates and numbers are often skipped.
- 'EDGES': For direct=True, instead of making all edges, pick column pairs. E.g., {'a': ['b', 'd'], 'd': ['d']} creates edges between columns a->b and a->d, and self-edges d->d.

Returns {'entities': DF, 'events': DF, 'edges': DF, 'nodes': DF, 'graph': Plotter}

Return type dict

Example: Connect user<-row->boss

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df)
g = h['graph'].plot()
```

Example: Connect user->boss

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
```

(continues on next page)

(continued from previous page)

```
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph'].plot()
```

Example: Connect user<->boss

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, direct=True, opts={'EDGES': {'user
↪': ['boss'], 'boss': ['user']}})
g = h['graph'].plot()
```

Example: Only consider some columns for nodes

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, entity_types=['boss'])
g = h['graph'].plot()
```

Example: Collapse matching user::<id> and boss::<id> nodes into one person::<id> node

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, opts={'CATEGORIES': {'person': [
↪'user', 'boss']}})
g = h['graph'].plot()
```

Example: Use cudf engine instead of pandas

```
import cudf, graphistry
users_gdf = cudf.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x',
↪'y']})
h = graphistry.hypergraph(users_gdf, engine='cudf')
g = h['graph'].plot()
```

Parameters

- **entity_types** (Optional[List[str]]) –
- **opts** (dict) –
- **drop_na** (bool) –
- **drop_edge_attrs** (bool) –
- **verbose** (bool) –
- **direct** (bool) –
- **engine** (str) –
- **npartitions** (Optional[int]) –
- **chunksize** (Optional[int]) –

igraph2pandas(*ig*)

Under current bindings, transform an IGraph into a pandas edges dataframe and a nodes dataframe.

Deprecated in favor of *.from_igraph()*

Example

```
import graphistry
g = graphistry.bind()

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = g.bind(source='src', destination='dst').edges(es)

ig = g.pandas2igraph(es)
ig.vs['community'] = ig.community_infomap().membership

(es2, vs2) = g.igraph2pandas(ig)
g.nodes(vs2).bind(point_color='community').plot()
```

infer_labels()

Returns Plotter w/neo4j

- Prefers point_title/point_label if available
- Fallback to node id
- Raises exception if no nodes available, no likely candidates, and no matching node id fallback

Example

```
import graphistry
g = graphistry.nodes(pd.read_csv('nodes.csv'), 'id_col').infer_labels()
g.plot()
```

layout_cugraph(*layout='force_atlas2'*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*)

Layout the grpah using a cuGraph algorithm. For a list of layouts, see cugraph documentation (currently just force_atlas2).

Parameters

- **layout** (*str*) – Name of an cugraph layout method like *force_atlas2*
- **params** (*dict*) – Any named parameters to pass to the underlying cugraph method
- **kind** (*CuGraphKind*) – The kind of cugraph Graph
- **directed** (*bool*) – During the to_cugraph conversion, whether to be directed. (default True)
- **G** (*Optional[Any]*) – The cugraph graph (G) to layout. If None, the current graph is used.
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)
- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional[str]*) – If defined, set settings(url_params={'play': play}). (default 0)

Returns Plotter

Return type Plotter

Example: ForceAtlas2 layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_cugraph().plot()
```

Example: Change which column names are generated

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('force_atlas2', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

Example: Pass parameters to layout methods

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('forceatlas_2', params={'lin_log_mode': True,
↪'prevent_overlapping': True})
g2.plot()
```

layout_igraph(*layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Compute graph layout using igraph algorithm. For a list of layouts, see layout_algs or igraph documentation.

Parameters

- **layout** (*str*) – Name of an igraph.Graph.layout method like *sugiyama*
- **directed** (*Optional[bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – Whether to use igraph vertex ids (non-negative integers) or arbitary node ids (False, default)
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)
- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional[str]*) – If defined, set settings(url_params={'play': play}). (default 0)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method

Returns Plotter

Return type Plotter

Example: Sugiyama layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama')
```

(continues on next page)

(continued from previous page)

```
assert 'x' in g2._nodes
g2.plot()
```

Example: Change which column names are generated

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

Example: Pass parameters to layout methods - Sort nodes by degree

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.get_degrees()
assert 'degree' in g._nodes.columns
g3 = g.layout_igraph('sugiyama', params={'layers': 'degree'})
g3.plot()
```

layout_settings(*play=None*, *locked_x=None*, *locked_y=None*, *locked_r=None*, *left=None*, *top=None*, *right=None*, *bottom=None*, *lin_log=None*, *strong_gravity=None*, *dissuade_hubs=None*, *edge_influence=None*, *precision_vs_speed=None*, *gravity=None*, *scaling_ratio=None*)

Set layout options. Additive over previous settings.

Corresponds to options at https://hub.graphistry.com/docs/api/1/rest/url/#urloptions

Example: Animated radial layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'boss': ['c','c','e','e
↪']})
nodes = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'y': [1,   1,   2,   3,   4],
    'x': [1,   1,   0,   0,   0],
})
g = (graphistry
    .edges(edges, 's', 'd')
    .nodes(nodes, 'n')
    .layout_settings(locked_r=True, play=2000)
g.plot()
```

Parameters

- **play** (Optional[int]) –
- **locked_x** (Optional[bool]) –
- **locked_y** (Optional[bool]) –
- **locked_r** (Optional[bool]) –
- **left** (Optional[float]) –

- **top** (Optional[float]) –
- **right** (Optional[float]) –
- **bottom** (Optional[float]) –
- **lin_log** (Optional[bool]) –
- **strong_gravity** (Optional[bool]) –
- **dissuade_hubs** (Optional[bool]) –
- **edge_influence** (Optional[float]) –
- **precision_vs_speed** (Optional[float]) –
- **gravity** (Optional[float]) –
- **scaling_ratio** (Optional[float]) –

name(*name*)

Upload name

Parameters **name** (*str*) – Upload name

networkx2pandas(*g*)

networkx_checkoverlap(*g*)

nodes(*nodes*, *node=None*, **args*, ***kwargs*)

Specify the set of nodes and associated data. If a callable, will be called with current Plotter and whatever positional+named arguments

Must include any nodes referenced in the edge list.

Parameters **nodes** (*Pandas dataframe or Callable*) – Nodes and their attributes.

Returns Plotter

Return type Plotter

Example

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry
    .bind(source='src', destination='dst')
    .edges(es)

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.bind(node='v').nodes(vs)

g.plot()
```

Example

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry.edges(es, 'src', 'dst')

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.nodes(vs, 'v)
```

(continues on next page)

(continued from previous page)

```
g.plot()
```

Example

```
import graphistry

def sample_nodes(g, n):
    return g._nodes.sample(n)

df = pandas.DataFrame({'id': [0,1,2], 'v': [1,2,0]})

graphistry
    .nodes(df, 'id')
    ..nodes(sample_nodes, n=2)
    ..nodes(sample_nodes, None, 2)  # equivalent
    .plot()
```

nodexl(*xls_or_url*, *source='default'*, *engine=None*, *verbose=False*)

pandas2igraph(*edges*, *directed=True*)

Convert a pandas edge dataframe to an IGraph graph.

Uses current bindings. Defaults to treating edges as directed.

Example

```
import graphistry
g = graphistry.bind()

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = g.bind(source='src', destination='dst')

ig = g.pandas2igraph(es)
ig.vs['community'] = ig.community_infomap().membership
g.bind(point_color='community').plot(ig)
```

pipe(*graph_transform*, **args*, ***kwargs*)

Create new Plotter derived from current

Parameters **graph_transform** (*Callable*) –

Example: Simple

```
import graphistry

def fill_missing_bindings(g, source='src', destination='dst'):
    return g.bind(source=source, destination=destination)

graphistry
    .edges(pandas.DataFrame({'src': [0,1,2], 'd': [1,2,0]}))
    .pipe(fill_missing_bindings, destination='d')  # binds 'src'
    .plot()
```

Return type Plottable

plot(*graph=None*, *nodes=None*, *name=None*, *description=None*, *render=None*, *skip_upload=False*, *as_files=False*, *memoize=True*, *extra_html=''*, *override_html_style=None*)

Upload data to the Graphistry server and show as an iframe of it.

Uses the currently bound schema structure and visual encodings. Optional parameters override the current bindings.

When used in a notebook environment, will also show an iframe of the visualization.

Parameters

- **graph** (*Any*) – Edge table (pandas, arrow, cudf) or graph (NetworkX, IGraph).
- **nodes** (*Any*) – Nodes table (pandas, arrow, cudf)
- **name** (*str*) – Upload name.
- **description** (*str*) – Upload description.
- **render** (*bool*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL
- **skip_upload** (*bool*) – Return node/edge/bindings that would have been uploaded. By default, upload happens.
- **as_files** (*bool*) – Upload distinct node/edge files under the managed Files PI. Default off, will switch to default-on when stable.
- **memoize** (*bool*) – Tries to memoize pandas/cudf->arrow conversion, including skipping upload. Default on.
- **extra_html** (*Optional[str]*) – Allow injecting arbitrary HTML into the visualization iframe.
- **override_html_style** (*Optional[str]*) – Set fully custom style tag.

Example: Simple

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .edges(es)
    .plot()
```

Example: Shorthand

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .plot(es)
```

privacy(*mode=None*, *notify=None*, *invited_users=None*, *message=None*)

Set local sharing mode

Parameters

- **mode** (*Optional[str]*) – Either “private”, “public”, or inherit from global privacy()
- **notify** (*Optional[bool]*) – Whether to email the recipient(s) upon upload, defaults to global privacy()

- **invited_users** (*Optional[List]*) – List of recipients, where each is {"email": str, "action": str} and action is "10" (view) or "20" (edit), defaults to global privacy()
- **message** (Optional[str]) – Email to send when notify=True

Requires an account with sharing capabilities.

Shared datasets will appear in recipients' galleries.

If mode is set to "private", only accounts in invited_users list can access. Mode "public" permits viewing by any user with the URL.

Action "10" (view) gives read access, while action "20" (edit) gives edit access, like changing the sharing mode.

When notify is true, uploads will trigger notification emails to invitees. Email will use visualization's ".name()"

When settings are not specified, they are inherited from the global graphistry.privacy() defaults

Example: Limit visualizations to current user

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
g = g.privacy()  # default uploads to mode="private"
g.plot()
```

Example: Default to publicly viewable visualizations

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
#g = g.privacy(mode="public")  # can skip calling .privacy() for this␣
↪default
g.plot()
```

Example: Default to sharing with select teammates, and keep notifications opt-in

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
g = g.privacy(
    mode="private",
    invited_users=[
```

(continues on next page)

(continued from previous page)

```
        {"email": "friend1@acme.org", "action": "10"}, # view
        {"email": "friend2@acme.org", "action": "20"}, # edit
    ],
    notify=False)
g.plot()
```

Example: Keep visualizations public and email notifications upon upload

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
g = g.name('my cool viz')  # For friendlier invitations
g = g.privacy(
    mode="public",
    invited_users=[
        {"email": "friend1@acme.org", "action": "10"}, # view
        {"email": "friend2@acme.org", "action": "20"}, # edit
    ],
    notify=True)
g.plot()
```

reset_caches()

Reset memoization caches

scene_settings(*menu=None*, *info=None*, *show_arrows=None*, *point_size=None*, *edge_curvature=None*, *edge_opacity=None*, *point_opacity=None*)

Set scene options. Additive over previous settings.

Corresponds to options at https://hub.graphistry.com/docs/api/1/rest/url/#urloptions

Example: Hide arrows and straighten edges

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'boss': ['c','c','e','e
↪']})
nodes = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'y': [1,   1,   2,   3,   4],
    'x': [1,   1,   0,   0,   0],
})
g = (graphistry
    .edges(edges, 's', 'd')
    .nodes(nodes, 'n')
    .scene_settings(show_arrows=False, edge_curvature=0.0)
g.plot()
```

Parameters

- **menu** (Optional[bool]) –
- **info** (Optional[bool]) –
- **show_arrows** (Optional[bool]) –

- **point_size** (Optional[float]) –
- **edge_curvature** (Optional[float]) –
- **edge_opacity** (Optional[float]) –
- **point_opacity** (Optional[float]) –

settings(*height=None*, *url_params={}*, *render=None*)

Specify iframe height and add URL parameter dictionary.

The library takes care of URI component encoding for the dictionary.

Parameters

- **height** (*int*) – Height in pixels.
- **url_params** (*dict*) – Dictionary of querystring parameters to append to the URL.
- **render** (*bool*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL

style(*fg=None*, *bg=None*, *page=None*, *logo=None*)

Set general visual styles

See .bind() and .settings(url_params={}) for additional styling options, and addStyle() for another way to set the same attributes.

To facilitate reuse and replayable notebooks, the style() call is chainable. Invocation does not effect the old style: it instead returns a new Plotter instance with the new styles added to the existing ones. Both the old and new styles can then be used for different graphs.

style() will fully replace any defined parameter in the existing style settings, while addStyle() will merge over previous values

Parameters

- **fg** (*dict*) – Dictionary {'blendMode': str} of any valid CSS blend mode
- **bg** (*dict*) – Nested dictionary of page background properties. { 'color': str, 'gradient': {'kind': str, 'position': str, 'stops': list }, 'image': { 'url': str, 'width': int, 'height': int, 'blendMode': str }
- **logo** (*dict*) – Nested dictionary of logo properties. { 'url': str, 'autoInvert': bool, 'position': str, 'dimensions': { 'maxWidth': int, 'maxHeight': int }, 'crop': { 'top': int, 'left': int, 'bottom': int, 'right': int }, 'padding': { 'top': int, 'left': int, 'bottom': int, 'right': int}, 'style': str}
- **page** (*dict*) – Dictionary of page metadata settings. { 'favicon': str, 'title': str }

Returns Plotter

Return type Plotter

Example: Chained merge - results in url and blendMode being set, while color is dropped

```
g2 =  g.style(bg={'color': 'black'}, fg={'blendMode': 'screen'})
g3 = g2.style(bg={'image': {'url': 'http://site.com/watermark.png'}})
```

Example: Gradient background

```
g.style(bg={'gradient': {'kind': 'linear', 'position': 45, 'stops': [[
↪'rgb(0,0,0)', '0%'], ['rgb(255,255,255)', '100%']]}})
```

Example: Page settings

```
g.style(page={'title': 'Site - {{ name }}', 'favicon': 'http://site.com/
↪logo.ico'})
```

tigergraph(*protocol='http'*, *server='localhost'*, *web_port=14240*, *api_port=9000*, *db=None*, *user='tigergraph'*, *pwd='tigergraph'*, *verbose=False*)

Register Tigergraph connection setting defaults

Parameters

- **protocol** (*Optional[str]*) – Protocol used to contact the database.
- **server** (*Optional[str]*) – Domain of the database
- **web_port** (*Optional[int]*) –
- **api_port** (*Optional[int]*) –
- **db** (*Optional[str]*) – Name of the database
- **user** (*Optional[str]*) –
- **pwd** (*Optional[str]*) –
- **verbose** (*Optional[bool]*) – Whether to print operations

Returns Plotter

Return type Plotter

Example: Standard

```
import graphistry
tg = graphistry.tigergraph(protocol='https', server='acme.com', db='my_db
↪', user='alice', pwd='tigergraph2')
```

to_cugraph(*directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *kind='Graph'*)

Convert current graph to a cugraph.Graph object

To assign an edge weight, use *g.bind(edge_weight='some_col').to_cugraph()*

Load from pandas, cudf, or dask_cudf DataFrames

Parameters

- **directed** (bool) –
- **include_nodes** (bool) –
- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **kind** (Literal['Graph', 'MultiGraph', 'BiPartiteGraph']) –

to_igraph(*directed=True*, *use_vids=False*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*)

Convert current item to igraph Graph . See examples in from_igraph.

Parameters

- **directed** (*bool*) – Whether to create a directed graph (default True)
- **include_nodes** (*bool*) – Whether to ingest the nodes table, if it exists (default True)

- **node_attributes** (*Optional[List[str]]*) – Which node attributes to load, None means all (default None)
- **edge_attributes** (*Optional[List[str]]*) – Which edge attributes to load, None means all (default None)
- **use_vids** (*bool*) – Whether to interpret IDs as igraph vertex IDs, which must be non-negative integers (default False)

graphistry.PlotterBase.**maybe_cudf**()

graphistry.PlotterBase.**maybe_dask_cudf**()

graphistry.PlotterBase.**maybe_dask_dataframe**()

graphistry.PlotterBase.**maybe_spark**()

1.2 Plotter Modules

class graphistry.Plottable.**Plottable**(**args*, ***kwargs*)

Bases: object

DGL_graph: Optional[Any]

bind(*source=None*, *destination=None*, *node=None*, *edge=None*, *edge_title=None*, *edge_label=None*, *edge_color=None*, *edge_weight=None*, *edge_size=None*, *edge_opacity=None*, *edge_icon=None*, *edge_source_color=None*, *edge_destination_color=None*, *point_title=None*, *point_label=None*, *point_color=None*, *point_weight=None*, *point_size=None*, *point_opacity=None*, *point_icon=None*, *point_x=None*, *point_y=None*)

chain(*ops*)

ops is List[ASTObject]

Parameters **ops** (List[Any]) –

Return type Plottable

collapse(*node*, *attribute*, *column*, *self_edges=False*, *unwrap=False*, *verbose=False*)

Parameters

- **node** (Union[str, int]) –
- **attribute** (Union[str, int]) –
- **column** (Union[str, int]) –
- **self_edges** (bool) –
- **unwrap** (bool) –
- **verbose** (bool) –

Return type Plottable

compute_cugraph(*alg*, *out_col=None*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*)

Parameters

- **alg** (str) –
- **out_col** (Optional[str]) –
- **params** (dict) –

- **kind** (Literal['Graph', 'MultiGraph', 'BiPartiteGraph']) –
- **G** (Optional[Any]) –

copy()

drop_nodes(*nodes*)

Parameters **nodes** (Any) –

Return type Plottable

edges(*edges*, *source=None*, *destination=None*, *edge=None*, **args*, ***kwargs*)

Parameters

- **edges** (Union[Callable, Any]) –
- **source** (Optional[str]) –
- **destination** (Optional[str]) –
- **edge** (Optional[str]) –

Return type Plottable

filter_edges_by_dict(*filter_dict=None*)

Parameters **filter_dict** (Optional[dict]) –

Return type Plottable

filter_nodes_by_dict(*filter_dict=None*)

Parameters **filter_dict** (Optional[dict]) –

Return type Plottable

from_cugraph(*G*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

Parameters

- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **load_nodes** (bool) –
- **load_edges** (bool) –
- **merge_if_existing** (bool) –

from_igraph(*ig*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

Parameters

- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **load_nodes** (bool) –
- **load_edges** (bool) –
- **merge_if_existing** (bool) –

get_degrees(*col='degree'*, *degree_in='degree_in'*, *degree_out='degree_out'*)

Parameters

- **col** (str) –
- **degree_in** (str) –
- **degree_out** (str) –

Return type Plottable

get_indegrees(*col='degree_in'*)

Parameters **col** (str) –

Return type Plottable

get_outdegrees(*col='degree_out'*)

Parameters **col** (str) –

Return type Plottable

get_topological_levels(*level_col='level'*, *allow_cycles=True*, *warn_cycles=True*, *remove_self_loops=True*)

Parameters

- **level_col** (str) –
- **allow_cycles** (bool) –
- **warn_cycles** (bool) –
- **remove_self_loops** (bool) –

Return type Plottable

hop(*nodes*, *hops=1*, *to_fixed_point=False*, *direction='forward'*, *edge_match=None*, *source_node_match=None*, *destination_node_match=None*, *return_as_wave_front=False*)

Parameters

- **nodes** (Optional[DataFrame]) –
- **hops** (Optional[int]) –
- **to_fixed_point** (bool) –
- **direction** (str) –
- **edge_match** (Optional[dict]) –
- **source_node_match** (Optional[dict]) –
- **destination_node_match** (Optional[dict]) –
- **return_as_wave_front** (bool) –

Return type Plottable

keep_nodes(*nodes*)

Parameters **nodes** (Union[List, Any]) –

Return type Plottable

layout_cugraph(*layout='force_atlas2'*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*)

Parameters

- **layout** (str) –

- **params** (dict) –
- **kind** (Literal['Graph', 'MultiGraph', 'BiPartiteGraph']) –
- **G** (Optional[Any]) –
- **bind_position** (bool) –
- **x_out_col** (str) –
- **y_out_col** (str) –
- **play** (Optional[int]) –

layout_settings(*play=None*, *locked_x=None*, *locked_y=None*, *locked_r=None*, *left=None*, *top=None*, *right=None*, *bottom=None*, *lin_log=None*, *strong_gravity=None*, *dissuade_hubs=None*, *edge_influence=None*, *precision_vs_speed=None*, *gravity=None*, *scaling_ratio=None*)

Parameters

- **play** (Optional[int]) –
- **locked_x** (Optional[bool]) –
- **locked_y** (Optional[bool]) –
- **locked_r** (Optional[bool]) –
- **left** (Optional[float]) –
- **top** (Optional[float]) –
- **right** (Optional[float]) –
- **bottom** (Optional[float]) –
- **lin_log** (Optional[bool]) –
- **strong_gravity** (Optional[bool]) –
- **dissuade_hubs** (Optional[bool]) –
- **edge_influence** (Optional[float]) –
- **precision_vs_speed** (Optional[float]) –
- **gravity** (Optional[float]) –
- **scaling_ratio** (Optional[float]) –

materialize_nodes(*reuse=True*, *engine='auto'*)

Parameters

- **reuse** (bool) –
- **engine** (Union[Engine, Literal['auto']]) –

Return type Plottable

nodes(*nodes*, *node=None*, **args*, ***kwargs*)

Parameters

- **nodes** (Union[Callable, Any]) –
- **node** (Optional[str]) –

Return type Plottable

pipe(*graph_transform*, **args*, ***kwargs*)

Parameters **graph_transform** (Callable) –

Return type Plottable

prune_self_edges()

Return type Plottable

to_cugraph(*directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *kind='Graph'*)

Parameters

- **directed** (bool) –
- **include_nodes** (bool) –
- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **kind** (Literal['Graph', 'MultiGraph', 'BiPartiteGraph']) –

Return type Any

to_igraph(*directed=True*, *use_vids=False*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*)

Parameters

- **directed** (bool) –
- **use_vids** (bool) –
- **include_nodes** (bool) –
- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –

Return type Any

CHAPTER

TWO

PLUGINS

2.1 iGraph

graphistry.plugins.igraph.**compute_igraph**(*self*, *alg*, *out_col=None*, *directed=None*, *use_vids=False*, *params={}*)

Enrich or replace graph using igraph methods

Parameters

- **alg** (*str*) – Name of an igraph.Graph method like *pagerank*
- **out_col** (*Optional[str]*) – For algorithms that generate a node attribute column, *out_col* is the desired output column name. When *None*, use the algorithm's name. (default None)
- **directed** (*Optional[bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – During the to_igraph conversion, whether to interpret IDs as igraph vertex IDs (non-negative integers) or arbitrary values (False, default)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method

Returns Plotter

Return type Plotter

Example: Pagerank

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

Example: Pagerank with custom name

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank', out_col='my_pr')
assert 'my_pr' in g2._nodes.columns
```

Example: Pagerank on an undirected

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank', directed=False)
assert 'pagerank' in g2._nodes.columns
```

Example: Pagerank with custom parameters

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_igraph('pagerank', params={'damping': 0.85})
assert 'pagerank' in g2._nodes.columns
```

Parameters **self** (Plottable) –

graphistry.plugins.igraph.**from_igraph**(*self*, *ig*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

Convert igraph object into Plotter

If base g has _node, _source, _destination definitions, use them

When merge_if_existing with preexisting nodes/edges df and shapes match ig, combine attributes

For merge_if_existing to work with edges, must set g._edge and have corresponding edge index attribute in igraph.Graph

Parameters

- **ig** (*igraph*) – Source igraph object
- **node_attributes** (*Optional[List[str]]*) – Subset of node attributes to load; None means all (default)
- **edge_attributes** (*Optional[List[str]]*) – Subset of edge attributes to load; None means all (default)
- **load_nodes** (*bool*) – Whether to load nodes dataframe (default True)
- **load_edges** (*bool*) – Whether to load edges dataframe (default True)
- **merge_if_existing** (bool) – Whether to merge with existing node/edge dataframes (default True)
- **merge_if_existing** – bool

Return type Plottable

Returns Plotter

Example: Convert from igraph, including all node/edge properties

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e'], 'v
↪': [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degrees()
assert 'degree' in g._nodes.columns
g2 = g.from_igraph(g.to_igraph())
assert len(g2._nodes.columns) == len(g._nodes.columns)
```

Example: Enrich from igraph, but only load in 1 node attribute

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e'], 'v
↪': [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degree()
assert 'degree' in g._nodes
ig = g.to_igraph(include_nodes=False)
assert 'degree' not in ig.vs
ig.vs['pagerank'] = ig.pagerank()
g2 = g.from_igraph(ig, load_edges=False, node_attributes=[g._node, 'pagerank
↪'])
assert 'pagerank' in g2._nodes
asssert 'degree' in g2._nodes
```

graphistry.plugins.igraph.**layout_igraph**(*self*, *layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Compute graph layout using igraph algorithm. For a list of layouts, see layout_algs or igraph documentation.

Parameters

- **layout** (*str*) – Name of an igraph.Graph.layout method like *sugiyama*
- **directed** (*Optional[bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – Whether to use igraph vertex ids (non-negative integers) or arbitary node ids (False, default)
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)
- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional[str]*) – If defined, set settings(url_params={'play': play}). (default 0)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method

Returns Plotter

Return type Plotter

Example: Sugiyama layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama')
assert 'x' in g2._nodes
g2.plot()
```

Example: Change which column names are generated

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
```

(continues on next page)

(continued from previous page)

```
assert g2._point_x == 'my_x'
g2.plot()
```

Example: Pass parameters to layout methods - Sort nodes by degree

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.get_degrees()
assert 'degree' in g._nodes.columns
g3 = g.layout_igraph('sugiyama', params={'layers': 'degree'})
g3.plot()
```

Parameters **self** (Plottable) –

graphistry.plugins.igraph.**to_igraph**(*self*, *directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *use_vids=False*)

Convert current item to igraph Graph . See examples in from_igraph.

Parameters

- **directed** (*bool*) – Whether to create a directed graph (default True)
- **include_nodes** (*bool*) – Whether to ingest the nodes table, if it exists (default True)
- **node_attributes** (*Optional[List[str]]*) – Which node attributes to load, None means all (default None)
- **edge_attributes** (*Optional[List[str]]*) – Which edge attributes to load, None means all (default None)
- **use_vids** (*bool*) – Whether to interpret IDs as igraph vertex IDs, which must be non-negative integers (default False)
- **self** (Plottable) –

2.2 CuGraph

graphistry.plugins.cugraph.**compute_cugraph**(*self*, *alg*, *out_col=None*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*)

Run cugraph algorithm on graph. For algorithm parameters, see cuGraph docs.

Parameters

- **alg** (*str*) – algorithm name
- **out_col** (*Optional[str]*) – node table output column name, defaults to alg param
- **params** (*dict*) – algorithm parameters passed to cuGraph as kwargs
- **kind** (*CuGraphKind*) – kind of cugraph to use
- **directed** (*bool*) – whether graph is directed
- **G** (*Optional[cugraph.Graph]*) – cugraph graph to use; if None, use self

Returns Plottable

Return type Plottable

Example: Pagerank

```
g2 = g.compute_cugraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

Example: Katz centrality with rename

```
g2 = g.compute_cugraph('katz_centrality', out_col='katz_centrality_renamed')
assert 'katz_centrality_renamed' in g2._nodes.columns
```

Example: Pass params to cugraph

```
g2 = g.compute_cugraph('k_truss', params={'k': 2})
assert 'k_truss' in g2._nodes.columns
```

Parameters **self** (Plottable) –

graphistry.plugins.cugraph.**df_to_gdf**(*df*)

Parameters **df** (Any) –

graphistry.plugins.cugraph.**from_cugraph**(*self*, *G*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

If bound IDs, use the same IDs in the returned graph.

If non-empty nodes/edges, instead of returning G's topology, use existing topology and merge in G's attributes

Parameters

- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **load_nodes** (bool) –
- **load_edges** (bool) –
- **merge_if_existing** (bool) –

Return type Plottable

graphistry.plugins.cugraph.**layout_cugraph**(*self*, *layout='force_atlas2'*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*)

Layout the grpah using a cuGraph algorithm. For a list of layouts, see cugraph documentation (currently just force_atlas2).

Parameters

- **layout** (*str*) – Name of an cugraph layout method like *force_atlas2*
- **params** (*dict*) – Any named parameters to pass to the underlying cugraph method
- **kind** (*CuGraphKind*) – The kind of cugraph Graph
- **directed** (*bool*) – During the to_cugraph conversion, whether to be directed. (default True)
- **G** (*Optional[Any]*) – The cugraph graph (G) to layout. If None, the current graph is used.
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)

- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional[str]*) – If defined, set settings(url_params={'play': play}). (default 0)

Returns Plotter

Return type Plotter

Example: ForceAtlas2 layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_cugraph().plot()
```

Example: Change which column names are generated

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('force_atlas2', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

Example: Pass parameters to layout methods

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('forceatlas_2', params={'lin_log_mode': True, 'prevent_
↪overlapping': True})
g2.plot()
```

Parameters **self** (Plottable) –

graphistry.plugins.cugraph.**to_cugraph**(*self*, *directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *kind='Graph'*)

Convert current graph to a cugraph.Graph object

To assign an edge weight, use *g.bind(edge_weight='some_col').to_cugraph()*

Load from pandas, cudf, or dask_cudf DataFrames

Parameters

- **self** (Plottable) –
- **directed** (bool) –
- **include_nodes** (bool) –
- **node_attributes** (Optional[List[str]]) –
- **edge_attributes** (Optional[List[str]]) –
- **kind** (Literal['Graph', 'MultiGraph', 'BiPartiteGraph']) –

CHAPTER

THREE

COMPUTE

3.1 ComputeMixin module

class graphistry.compute.ComputeMixin.**ComputeMixin**(**args*, ***kwargs*)

Bases: object

chain(**args*, ***kwargs*)

Experimental: Chain a list of operations

Return subgraph of matches according to the list of node & edge matchers If any matchers are named, add a correspondingly named boolean-valued column to the output

Parameters **ops** – List[ASTObject] Various node and edge matchers

Returns Plotter

Return type Plotter

Example: Find nodes of some type

```
from graphistry.ast import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
```

Example: Find 2-hop edge sequences with some attribute

```
from graphistry.ast import e_forward

g_2_hops = g.chain([ e_forward({"interesting": True}, hops=2) ])
g_2_hops.plot()
```

Example: Find any node 1-2 hops out from another node, and label each hop

```
from graphistry.ast import n, e_undirected

g_2_hops = g.chain([ n({g._node: "a"}), e_undirected(name="hop1"), e_
↪undirected(name="hop2") ])
print('# first-hop edges:', len(g_2_hops._edges[ g_2_hops._edges.hop1 == True␣
↪]))
```

Example: Transaction nodes between two kinds of risky nodes

```
from graphistry.ast import n, e_forward, e_reverse

g_risky = g.chain([
```

(continues on next page)

(continued from previous page)

```
    n({"risk1": True}),
    e_forward(to_fixed=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
print('# hits:', len(g_risky._nodes[ g_risky._nodes.hit ]))
```

collapse(*node*, *attribute*, *column*, *self_edges=False*, *unwrap=False*, *verbose=False*)
Topology-aware collapse by given column attribute starting at *node*

Traverses directed graph from start node *node* and collapses clusters of nodes that share the same property so that topology is preserved.

Parameters

- **node** (Union[str, int]) – start *node* to begin traversal
- **attribute** (Union[str, int]) – the given *attribute* to collapse over within *column*
- **column** (Union[str, int]) – the *column* of nodes DataFrame that contains *attribute* to collapse over
- **self_edges** (bool) – whether to include self edges in the collapsed graph
- **unwrap** (bool) – whether to unwrap the collapsed graph into a single node
- **verbose** (bool) – whether to print out collapse summary information

:returns:A new Graphistry instance with nodes and edges DataFrame containing collapsed nodes and edges given by column attribute – nodes and edges DataFrames contain six new columns *collapse_{node | edges}* and *final_{node | edges}*, while original (node, src, dst) columns are left untouched :rtype: Plottable

drop_nodes(*nodes*)
return g with any nodes/edges involving the node id series removed

filter_edges_by_dict(**args*, ***kwargs*)
filter edges to those that match all values in filter_dict

filter_nodes_by_dict(**args*, ***kwargs*)
filter nodes to those that match all values in filter_dict

get_degrees(*col='degree'*, *degree_in='degree_in'*, *degree_out='degree_out'*)
Decorate nodes table with degree info

Edges must be dataframe-like: pandas, cudf, ...

Parameters determine generated column names

Warning: Self-cycles are currently double-counted. This may change.

Example: Generate degree columns

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
print(g._nodes)  # None
g2 = g.get_degrees()
print(g2._nodes)  # pd.DataFrame with 'id', 'degree', 'degree_in',
↪'degree_out'
```

Parameters

- **col** (str) –
- **degree_in** (str) –
- **degree_out** (str) –

get_indegrees(*col='degree_in'*)
See get_degrees

Parameters **col** (str) –

get_outdegrees(*col='degree_out'*)
See get_degrees

Parameters **col** (str) –

get_topological_levels(*level_col='level'*, *allow_cycles=True*, *warn_cycles=True*, *remove_self_loops=True*)
Label nodes on column level_col based on topological sort depth Supports pandas + cudf, using parallelism within each level computation Options: * allow_cycles: if False and detects a cycle, throw ValueException, else break cycle by picking a lowest-in-degree node * warn_cycles: if True and detects a cycle, proceed with a warning * remove_self_loops: preprocess by removing self-cycles. Avoids allow_cycles=False, warn_cycles=True messages.

Example:

edges_df = gpd.DataFrame({'s': ['a', 'b', 'c', 'd'],'d': ['b', 'c', 'e', 'e']}) g = graphistry.edges(edges_df, 's', 'd') g2 = g.get_topological_levels() g2._nodes.info() # pd.DataFrame with | 'id' , 'level' |

Parameters

- **level_col** (str) –
- **allow_cycles** (bool) –
- **warn_cycles** (bool) –
- **remove_self_loops** (bool) –

Return type Plottable

hop(**args*, ***kwargs*)
Given a graph and some source nodes, return subgraph of all paths within k-hops from the sources

g: Plotter nodes: dataframe with id column matching g._node. None signifies all nodes (default). hops: how many hops to consider, if any bound (default 1) to_fixed_point: keep hopping until no new nodes are found (ignores hops) direction: 'forward', 'reverse', 'undirected' edge_match: dict of kv-pairs to exact match (see also: filter_edges_by_dict) source_node_match: dict of kv-pairs to match nodes before hopping destination_node_match: dict of kv-pairs to match nodes after hopping (including intermediate) return_as_wave_front: Only return the nodes/edges reached, ignoring past ones (primarily for internal use)

keep_nodes(*nodes*)
Limit nodes and edges to those selected by parameter nodes For edges, both source and destination must be in nodes Nodes can be a list or series of node IDs, or a dictionary When a dictionary, each key corresponds to a node column, and nodes will be included when all match

materialize_nodes(*reuse=True*, *engine='auto'*)
Generate g._nodes based on g._edges

Uses g._node for node id if exists, else 'id'

Edges must be dataframe-like: cudf, pandas, ...

When reuse=True and g._nodes is not None, use it

Example: Generate nodes

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
print(g._nodes)  # None
g2 = g.materialize_nodes()
print(g2._nodes)  # pd.DataFrame
```

Parameters

- **reuse** (bool) –
- **engine** (Union[Engine, Literal['auto']]) –

Return type Plottable

prune_self_edges()

3.2 Chain

graphistry.compute.chain.**chain**(*self*, *ops*)

Experimental: Chain a list of operations

Return subgraph of matches according to the list of node & edge matchers If any matchers are named, add a correspondingly named boolean-valued column to the output

Parameters **ops** (List[ASTObject]) – List[ASTObject] Various node and edge matchers

Returns Plotter

Return type Plotter

Example: Find nodes of some type

```
from graphistry.ast import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
```

Example: Find 2-hop edge sequences with some attribute

```
from graphistry.ast import e_forward

g_2_hops = g.chain([ e_forward({"interesting": True}, hops=2) ])
g_2_hops.plot()
```

Example: Find any node 1-2 hops out from another node, and label each hop

```
from graphistry.ast import n, e_undirected

g_2_hops = g.chain([ n({g._node: "a"}), e_undirected(name="hop1"), e_
↪undirected(name="hop2") ])
print('# first-hop edges:', len(g_2_hops._edges[ g_2_hops._edges.hop1 == True ]))
```

Example: Transaction nodes between two kinds of risky nodes

```
from graphistry.ast import n, e_forward, e_reverse

g_risky = g.chain([
    n({"risk1": True}),
```

(continues on next page)

(continued from previous page)

```
    e_forward(to_fixed=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
print('# hits:', len(g_risky._nodes[ g_risky._nodes.hit ]))
```

Parameters **self** (Plottable) –

graphistry.compute.chain.**combine_steps**(*g*, *kind*, *steps*)

Collect nodes and edges, taking care to deduplicate and tag any names

Parameters

- **g** (Plottable) –
- **kind** (str) –
- **steps** (List[Tuple[ASTObject, Plottable]]) –

Return type DataFrame

3.3 Cluster

class graphistry.compute.cluster.**ClusterMixin**(**args*, ***kwargs*)

Bases: object

dbscan(*min_dist=0.2*, *min_samples=1*, *cols=None*, *kind='nodes'*, *fit_umap_embedding=True*, *target=False*, *verbose=False*, **args*, ***kwargs*)

DBSCAN clustering on cpu or gpu infered automatically. Adds a *_dbscan* column to nodes or edges.

Examples:

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')

# cluster by UMAP embeddings
kind = 'nodes' | 'edges'
g2 = g.umap(kind=kind).dbscan(kind=kind)
print(g2._nodes['_dbscan']) | print(g2._edges['_dbscan'])

# dbscan in umap or featurize API
g2 = g.umap(dbscan=True, min_dist=1.2, min_samples=2, **kwargs)
# or, here dbscan is infered from features, not umap embeddings
g2 = g.featurize(dbscan=True, min_dist=1.2, min_samples=2, **kwargs)

# and via chaining,
g2 = g.umap().dbscan(min_dist=1.2, min_samples=2, **kwargs)

# cluster by feature embeddings
g2 = g.featurize().dbscan(**kwargs)

# cluster by a given set of feature column attributes, or with target=True
g2 = g.featurize().dbscan(cols=['ip_172', 'location', 'alert'], target=False,␣
↪**kwargs)

# equivalent to above (ie, cols != None and umap=True will still use features␣
↪dataframe, rather than UMAP embeddings)
```

(continues on next page)

(continued from previous page)

```
g2 = g.umap().dbscan(cols=['ip_172', 'location', 'alert'], umap=True | False,␣
↪**kwargs)

g2.plot() # color by `_dbscan` column
```

Useful: Enriching the graph with cluster labels from UMAP is useful for visualizing clusters in the graph by color, size, etc, as well as assessing metrics per cluster, e.g. https://github.com/graphistry/pygraphistry/blob/master/demos/ai/cyber/cyber-redteam-umap-demo.ipynb

Args:

min_dist float The maximum distance between two samples for them to be considered as in the same neighborhood.

kind str 'nodes' or 'edges'

cols list of columns to use for clustering given *g.featurize* has been run, nice way to slice features or targets by fragments of interest, e.g. ['ip_172', 'location', 'ssh', 'warnings']

fit_umap_embedding bool whether to use UMAP embeddings or features dataframe to cluster DBSCAN

min_samples The number of samples in a neighborhood for a point to be considered as a core point. This includes the point itself.

target whether to use the target column as the clustering feature

Parameters

- **min_dist** (float) –
- **min_samples** (int) –
- **cols** (Union[List, str, None]) –
- **kind** (str) –
- **fit_umap_embedding** (bool) –
- **target** (bool) –
- **verbose** (bool) –

transform_dbscan(*df*, *y=None*, *min_dist='auto'*, *infer_umap_embedding=False*, *sample=None*, *n_neighbors=None*, *kind='nodes'*, *return_graph=True*, *verbose=False*)

Transforms a minibatch dataframe to one with a new column '_dbscan' containing the DBSCAN cluster labels on the minibatch and generates a graph with the minibatch and the original graph, with edges between the minibatch and the original graph inferred from the umap embedding or features dataframe. Graph nodes | edges will be colored by '_dbscan' column.

Examples:

```
fit:
    g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')
    g2 = g.featurize().dbscan()

predict:
::

    emb, X, _, ndf = g2.transform_dbscan(ndf, return_graph=False)
    # or
```

(continues on next page)

(continued from previous page)

```
    g3 = g2.transform_dbscan(ndf, return_graph=True)
    g3.plot()
```

likewise for umap:

```
fit:
    g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')
    g2 = g.umap(X=.., y=..).dbscan()

predict:
::

    emb, X, y, ndf = g2.transform_dbscan(ndf, ndf, return_graph=False)
    # or
    g3 = g2.transform_dbscan(ndf, ndf, return_graph=True)
    g3.plot()
```

Args:

df dataframe to transform

y optional labels dataframe

min_dist The maximum distance between two samples for them to be considered as in the same neighborhood. smaller values will result in less edges between the minibatch and the original graph. Default 'auto', infers min_dist from the mean distance and std of new points to the original graph

fit_umap_embedding whether to use UMAP embeddings or features dataframe when inferring edges between the minibatch and the original graph. Default False, uses the features dataframe

sample number of samples to use when inferring edges between the minibatch and the original graph, if None, will only use closest point to the minibatch. If greater than 0, will sample the closest *sample* points in existing graph to pull in more edges. Default None

kind 'nodes' or 'edges'

return_graph whether to return a graph or the (emb, X, y, minibatch df enriched with DBSCAN labels), default True infered graph supports kind='nodes' only.

verbose whether to print out progress, default False

Parameters

- **df** (DataFrame) –
- **y** (Optional[DataFrame]) –
- **min_dist** (Union[float, str]) –
- **infer_umap_embedding** (bool) –
- **sample** (Optional[int]) –
- **n_neighbors** (Optional[int]) –
- **kind** (str) –
- **return_graph** (bool) –
- **verbose** (bool) –

graphistry.compute.cluster.**dbscan_fit**(*g*, *dbscan*, *kind='nodes'*, *cols=None*, *use_umap_embedding=True*, *target=False*, *verbose=False*)

Fits clustering on UMAP embeddings if umap is True, otherwise on the features dataframe or target dataframe if target is True.

Args:

g graphistry graph

kind 'nodes' or 'edges'

cols list of columns to use for clustering given *g.featurize* has been run

use_umap_embedding whether to use UMAP embeddings or features dataframe for clustering (default: True)

Parameters

- **g** (Any) –
- **dbscan** (Any) –
- **kind** (str) –
- **cols** (Union[List, str, None]) –
- **use_umap_embedding** (bool) –
- **target** (bool) –
- **verbose** (bool) –

graphistry.compute.cluster.**dbscan_predict**(*X*, *model*)

DBSCAN has no predict per se, so we reverse engineer one here from https://stackoverflow.com/questions/27822752/scikit-learn-predicting-new-points-with-dbscan

Parameters

- **X** (DataFrame) –
- **model** (Any) –

graphistry.compute.cluster.**get_model_matrix**(*g*, *kind*, *cols*, *umap*, *target*)

Allows for a single function to get the model matrix for both nodes and edges as well as targets, embeddings, and features

Args:

g graphistry graph

kind 'nodes' or 'edges'

cols list of columns to use for clustering given *g.featurize* has been run

umap whether to use UMAP embeddings or features dataframe

target whether to use the target dataframe or features dataframe

Returns: pd.DataFrame: dataframe of model matrix given the inputs

Parameters

- **kind** (str) –
- **cols** (Union[List, str, None]) –

graphistry.compute.cluster.**lazy_dbscan_import_has_dependency**()

graphistry.compute.cluster.**resolve_cpu_gpu_engine**(*engine*)

Parameters **engine** (Literal[typing_extensions.Literal['cuml', 'umap_learn'], 'auto']) –

Return type Literal['cuml', 'umap_learn']

3.4 Collapse

graphistry.compute.collapse.**check_default_columns_present_and_coerce_to_string**(*g*)
Helper to set COLLAPSE columns to nodes and edges dataframe, while converting src, dst, node to dtype(str)
:type g: Plottable :param g: graphistry instance

Returns graphistry instance

graphistry.compute.collapse.**check_has_set**(*ndf*, *parent*, *child*)

graphistry.compute.collapse.**collapse_algo**(*g*, *child*, *parent*, *attribute*, *column*, *seen*)
Basically candy crush over graph properties in a topology aware manner

Checks to see if child node has desired property from parent, we will need to check if (start_node=parent: has_attribute , children nodes: has_attribute) by case (T, T), (F, T), (T, F) and (F, F),we start recursive collapse (or not) on the children, reassigning nodes and edges.

if (T, T), append children nodes to start_node, re-assign the name of the node, and update the edge table with new name,

if (F, T) start k-(potentially new) super nodes, with k the number of children of start_node. Start node keeps k outgoing edges.

if (T, F) it is the end of the cluster, and we keep new node as is; keep going

if (F, F); keep going

Parameters

- **seen** (dict) –
- **g** (Plottable) – graphistry instance
- **child** (Union[str, int]) – child node to start traversal, for first traversal, set child=parent or vice versa.
- **parent** (Union[str, int]) – parent node to start traversal, in main call, this is set to child.
- **attribute** (Union[str, int]) – attribute to collapse by
- **column** (Union[str, int]) – column in nodes dataframe to collapse over.

Returns graphistry instance with collapsed nodes.

graphistry.compute.collapse.**collapse_by**(*self*, *parent*, *start_node*, *attribute*, *column*, *seen*, *self_edges=False*, *unwrap=False*, *verbose=True*)

Main call in collapse.py, collapses nodes and edges by attribute, and returns normalized graphistry object.

Parameters

- **self** (Plottable) – graphistry instance

- **parent** (Union[str, int]) – parent node to start traversal, in main call, this is set to child.
- **start_node** (Union[str, int]) –
- **attribute** (Union[str, int]) – attribute to collapse by
- **column** (Union[str, int]) – column in nodes dataframe to collapse over.
- **seen** (dict) – dict of previously collapsed pairs – {n1, n2) is seen as different from (n2, n1)
- **verbose** (bool) – bool, default True

:returns graphistry instance with collapsed and normalized nodes.

Parameters

- **self_edges** (bool) –
- **unwrap** (bool) –

Return type Plottable

graphistry.compute.collapse.**collapse_nodes_and_edges**(*g*, *parent*, *child*)

Asserts that parent and child node in ndf should be collapsed into super node. Sets new ndf with COLLAPSE nodes in graphistry instance g

this asserts that we SHOULD merge parent and child as super node # outside logic controls when that is the case # for example, it assumes parent is already in cluster keys of COLLAPSE node

Parameters

- **g** (Plottable) – graphistry instance
- **parent** (Union[str, int]) – *node* with *attribute* in *column*
- **child** (Union[str, int]) – *node* with *attribute* in *column*

Returns graphistry instance

graphistry.compute.collapse.**get_children**(*g*, *node_id*, *hops=1*)

Helper that gets children at k-hops from node *node_id*

:returns graphistry instance of hops

Parameters

- **g** (Plottable) –
- **node_id** (Union[str, int]) –
- **hops** (int) –

graphistry.compute.collapse.**get_cluster_store_keys**(*ndf*, *node*)

Main innovation in finding and adding to super node. Checks if node is a segment in any collapse_node in COLLAPSE column of nodes DataFrame

Parameters

- **ndf** (DataFrame) – node DataFrame
- **node** (Union[str, int]) – node to find

Returns DataFrame of bools of where *wrap_key(node)* exists in COLLAPSE column

graphistry.compute.collapse.**get_edges_in_out_cluster**(*g*, *node_id*, *attribute*, *column*, *directed=True*)

Traverses children of *node_id* and separates them into incluster and outcluster sets depending if they have *attribute* in node DataFrame *column*

Parameters

- **g** (Plottable) – graphistry instance
- **node_id** (Union[str, int]) – *node* with *attribute* in *column*
- **attribute** (Union[str, int]) – *attribute* to collapse in *column* over
- **column** (Union[str, int]) – *column* to collapse over
- **directed** (bool) –

graphistry.compute.collapse.**get_edges_of_node**(*g*, *node_id*, *outgoing_edges=True*, *hops=1*)

Gets edges of node at k-hops from node

Parameters

- **g** (Plottable) – graphistry instance
- **node_id** (Union[str, int]) – *node* to find edges from
- **outgoing_edges** (bool) – bool, if true, finds all outgoing edges of *node*, default True
- **hops** (int) – the number of hops from *node* to take, default = 1

Returns DataFrame of edges

graphistry.compute.collapse.**get_new_node_name**(*ndf*, *parent*, *child*)

If child in cluster group, melts name, else makes new parent_name from parent, child

Parameters

- **ndf** (DataFrame) – node DataFrame
- **parent** (Union[str, int]) – *node* with *attribute* in *column*
- **child** (Union[str, int]) – *node* with *attribute* in *column*

:returns new_parent_name

Return type str

graphistry.compute.collapse.**has_edge**(*g*, *n1*, *n2*, *directed=True*)

Checks if *n1* and *n2* share an (directed or not) edge

Parameters

- **g** (Plottable) – graphistry instance
- **n1** (Union[str, int]) – *node* to check if has edge to *n2*
- **n2** (Union[str, int]) – *node* to check if has edge to *n1*
- **directed** (bool) – bool, if True, checks only outgoing edges from *n1*->`n2`, else finds undirected edges

Return type bool

Returns bool, if edge exists between *n1* and *n2*

graphistry.compute.collapse.**has_property**(*g*, *ref_node*, *attribute*, *column*)

Checks if ref_node is in node dataframe in column with attribute :type attribute: Union[str, int] :param attribute: :type column: Union[str, int] :param column: :type g: Plottable :param g: graphistry instance :type ref_node: Union[str, int] :param ref_node: *node* to check if it as *attribute* in *column*

Return type bool

Returns bool

graphistry.compute.collapse.**in_cluster_store_keys**(*ndf*, *node*)

checks if node is in collapse_node in COLLAPSE column of nodes DataFrame

Parameters

- **ndf** (DataFrame) – nodes DataFrame
- **node** (Union[str, int]) – node to find

Return type bool

Returns bool

graphistry.compute.collapse.**melt**(*ndf*, *node*)

Reduces node if in cluster store, otherwise passes it through. ex:

node = "4" will take any sequence from get_cluster_store_keys, "1 2 3", "4 3 6" and returns "1 2 3 4 6" when they have a common entry (3).

:param ndf, node DataFrame :type node: Union[str, int] :param node: node to melt :returns new_parent_name of super node

Parameters **ndf** (DataFrame) –

Return type str

graphistry.compute.collapse.**normalize_graph**(*g*, *self_edges=False*, *unwrap=False*)

Final step after collapse traversals are done, removes duplicates and moves COLLAPSE columns into respective(node, src, dst) columns of node, edges dataframe from Graphistry instance g.

Parameters

- **g** (Plottable) – graphistry instance
- **self_edges** (bool) – bool, whether to keep duplicates from ndf, edf, default False
- **unwrap** (bool) – bool, whether to unwrap node text with ~, default True

Return type Plottable

Returns final graphistry instance

graphistry.compute.collapse.**reduce_key**(*key*)

Takes "1 1 2 1 2 3" -> "1 2 3

Parameters **key** (Union[str, int]) – node name

Return type str

Returns new node name with duplicates removed

graphistry.compute.collapse.**unpack**(*g*)

Helper method that unpacks graphistry instance

ex:

ndf, edf, src, dst, node = unpack(g)

Parameters **g** (Plottable) – graphistry instance

Returns node DataFrame, edge DataFrame, source column, destination column, node column

graphistry.compute.collapse.**unwrap_key**(*name*)
Unwraps node name: ~name~ -> name

Parameters **name** (Union[str, int]) – node to unwrap

Return type str

Returns unwrapped node name

graphistry.compute.collapse.**wrap_key**(*name*)
Wraps node name -> ~name~

Parameters **name** (Union[str, int]) – node name

Return type str

Returns wrapped node name

3.5 Conditional

class graphistry.compute.conditional.**ConditionalMixin**(**args*, ***kwargs*)
Bases: object

conditional_graph(*x*, *given*, *kind='nodes'*, **args*, ***kwargs*)
conditional_graph – p(x|given) = p(x, given) / p(given)

Useful for finding the conditional probability of a node or edge attribute

returned dataframe sums to 1 on each column

Parameters

- **x** – target column
- **given** – the dependent column
- **kind** – 'nodes' or 'edges'
- **args/kwargs** – additional arguments for g.bind(...)

Returns a graphistry instance with the conditional graph edges weighted by the conditional probability. edges are between *x* and *given*, keep in mind that g._edges.columns = [given, x, _probs]

conditional_probs(*x*, *given*, *kind='nodes'*, *how='index'*)
Produces a Dense Matrix of the conditional probability of x given y

Args: x: the column variable of interest given the column y=given given : the variabe to fix constant df pd.DataFrame: dataframe how (str, optional): One of 'column' or 'index'. Defaults to 'index'. kind (str, optional): 'nodes' or 'edges'. Defaults to 'nodes'.

Returns: pd.DataFrame: the conditional probability of x given the column y as dense array like dataframe

graphistry.compute.conditional.**conditional_probability**(*x*, *given*, *df*)

conditional probability function over categorical variables p(x | given) = p(x, given)/p(given)

Args: x: the column variable of interest given the column 'given' given: the variabe to fix constant df: dataframe with columns [given, x]

Returns: pd.DataFrame: the conditional probability of x given the column 'given'

Parameters **df** (DataFrame) –

graphistry.compute.conditional.**probs**(*x*, *given*, *df*, *how='index'*)
Produces a Dense Matrix of the conditional probability of x given *y=given*

Args: x: the column variable of interest given the column 'y' given : the variabe to fix constant df pd.DataFrame: dataframe how (str, optional): One of 'column' or 'index'. Defaults to 'index'.

Returns: pd.DataFrame: the conditional probability of x given the column 'y' as dense array like dataframe

Parameters **df** (DataFrame) –

3.6 Filter by Dictionary

graphistry.compute.filter_by_dict.**filter_by_dict**(*df*, *filter_dict=None*)
return df where rows match all values in filter_dict

Parameters **filter_dict** (Optional[dict]) –

Return type DataFrame

graphistry.compute.filter_by_dict.**filter_edges_by_dict**(*self*, *filter_dict*)
filter edges to those that match all values in filter_dict

Parameters

- **self** (Plottable) –
- **filter_dict** (dict) –

Return type Plottable

graphistry.compute.filter_by_dict.**filter_nodes_by_dict**(*self*, *filter_dict*)
filter nodes to those that match all values in filter_dict

Parameters

- **self** (Plottable) –
- **filter_dict** (dict) –

Return type Plottable

3.7 Hop

graphistry.compute.hop.**hop**(*self*, *nodes=None*, *hops=1*, *to_fixed_point=False*, *direction='forward'*, *edge_match=None*, *source_node_match=None*, *destination_node_match=None*, *return_as_wave_front=False*)
Given a graph and some source nodes, return subgraph of all paths within k-hops from the sources

g: Plotter nodes: dataframe with id column matching g._node. None signifies all nodes (default). hops: how many hops to consider, if any bound (default 1) to_fixed_point: keep hopping until no new nodes are found (ignores hops) direction: 'forward', 'reverse', 'undirected' edge_match: dict of kv-pairs to exact match (see also: filter_edges_by_dict) source_node_match: dict of kv-pairs to match nodes before hopping destination_node_match: dict of kv-pairs to match nodes after hopping (including intermediate) return_as_wave_front: Only return the nodes/edges reached, ignoring past ones (primarily for internal use)

Parameters

- **self** (Plottable) –

- **nodes** (Optional[DataFrame]) –
- **hops** (Optional[int]) –
- **to_fixed_point** (bool) –
- **direction** (str) –
- **edge_match** (Optional[dict]) –
- **source_node_match** (Optional[dict]) –
- **destination_node_match** (Optional[dict]) –

Return type Plottable

CHAPTER

FOUR

LAYOUTS

4.1 edge Module

class graphistry.layout.graph.edge.**Edge**(*x*, *y*, *w=1*, *data=None*, *connect=False*)

Bases: *graphistry.layout.graph.edgeBase.EdgeBase*

A graph edge.

Attributes

- data (object): an optional payload
- w (int): an optional weight associated with the edge (default 1) used by Dijkstra to find min-flow paths.
- feedback (bool): whether the Tarjan algorithm has inverted this edge to de-cycle the graph.

attach()

Attach this edge to the edge collections of the vertices.

data: object

detach()

Removes this edge from the edge collections of the vertices.

feedback: bool

w: int

4.2 edgeBase Module

class graphistry.layout.graph.edgeBase.**EdgeBase**(*x*, *y*)

Bases: object

Base class for edges.

Attributes

- degree (int): degree of the edge (number of unique vertices).
- v (list[Vertex]): list of vertices associated with this edge.

degree: int

Is 0 if a loop, otherwise 1.

4.3 graph Module

class graphistry.layout.graph.graph.**Graph**(*vertices=None*, *edges=None*, *directed=True*)

Bases: object

N(*v*, *f_io=0*)

add_edge(*e*)
add edge e and its vertices into the Graph possibly merging the associated graph_core components

add_edges(*edges*)

Parameters **edges** (List) –

add_vertex(*v*)
add vertex v into the Graph as a new component

component_class
alias of *graphistry.layout.graph.graphBase.GraphBase*

connected()
returns the list of components

deg_avg()
the average degree of vertices

deg_max()
the maximum degree of vertices

deg_min()
the minimum degree of vertices

edges()

eps()
the graph epsilon value (norm/order), average number of edges per vertex.

get_vertex_from_data(*data*)

get_vertices_count()

norm()
the norm of the graph (number of edges)

order()
the order of the graph (number of vertices)

path(*x*, *y*, *f_io=0*, *hook=None*)

remove_edge(*e*)
remove edge e possibly spawning two new cores if the graph_core that contained e gets disconnected.

remove_vertex(*x*)
remove vertex v and all its edges.

vertices()
see graph_core

4.4 graphBase Module

class graphistry.layout.graph.graphBase.**GraphBase**(*vertices=None*, *edges=None*, *directed=True*)

Bases: object

A connected graph of Vertex/Edge objects. A GraphBase is a *component* of a Graph that contains a connected set of Vertex and Edges.

Attributes: verticesPoset (Poset[Vertex]): the partially ordered set of vertices of the graph. edgesPoset (Poset[Edge]): the partially ordered set of edges of the graph. loops (set[Edge]): the set of *loop* edges (of degree 0). directed (bool): indicates if the graph is considered *oriented* or not.

N(*v*, *f_io=0*)

add_edge(*e*)
add edge e. At least one of its vertex must belong to the graph, the other being added automatically.

add_single_vertex(*v*)
allow a GraphBase to hold a single vertex.

complement(*G*)

constant_function(*value*)

contract(*e*)

deg_avg()
the average degree of vertices

deg_max()
the maximum degree of vertices

deg_min()
the minimum degree of vertices

dft(*start_vertex=None*)

dijkstra(*x*, *f_io=0*, *hook=None*)
shortest weighted-edges paths between x and all other vertices by dijkstra's algorithm with heap used as priority queue.

edges(*cond=None*)
generates an iterator over edges, with optional filter

eps()
the graph epsilon value (norm/order), average number of edges per vertex.

get_scs_with_feedback(*roots=None*)

Minimum FAS algorithm (feedback arc set) creating a DAG. Returns the set of strongly connected components ("scs") by using Tarjan algorithm. These are maximal sets of vertices such that there is a path from each vertex to every other vertex. The algorithm performs a DFS from the provided list of root vertices. A cycle is of course a strongly connected component,but a strongly connected component can include several cycles. The Feedback Acyclic Set of edge to be removed/reversed is provided by marking the edges with a "feedback" flag. Complexity is O(V+E).

Parameters **roots** –

Returns

leaves()
returns the list of *leaves* (vertices with no outward edges).

matrix(*cond=None*)

This associativity matrix is like the adjacency matrix but antisymmetric. Returns the associativity matrix of the graph component

Parameters **cond** – same a the condition function in vertices().

Returns array

norm()
The size of the edge poset (number of edges).

order()
the order of the graph (number of vertices)

partition()

path(*x*, *y*, *f_io=0*, *hook=None*)
shortest path between vertices x and y by breadth-first descent, contrained by f_io direction if provided. The path is returned as a list of Vertex objects. If a *hook* function is provided, it is called at every vertex added to the path, passing the vertex object as argument.

remove_edge(*e*)
remove Edge e, asserting that the resulting graph is still connex.

remove_vertex(*x*)
remove Vertex x and all associated edges.

roots()
returns the list of *roots* (vertices with no inward edges).

spans(*vertices*)

union_update(*G*)

vertices(*cond=None*)
generates an iterator over vertices, with optional filter

4.5 vertex Module

class graphistry.layout.graph.vertex.**Vertex**(*data=None*)
Bases: *graphistry.layout.graph.vertexBase.VertexBase*

Vertex class enhancing a VertexBase with graph-related features.

Attributes component (GraphBase): the component of connected vertices that contains this vertex. By default, a vertex belongs no component but when it is added in a graph, c points to the connected component in this graph. data (object) : an object associated with the vertex.

property index

4.6 vertexBase Module

class graphistry.layout.graph.vertexBase.**VertexBase**
Bases: object

Base class for vertices.

Attributes e (list[Edge]): list of edges associated with this vertex.

degree()
degree() : degree of the vertex (number of edges).

detach()
removes this vertex from all its edges and returns this list of edges.

e_dir(*dir*)
either e_in, e_out or all edges depending on provided direction parameter (>0 means outward).

e_from(*x*)
returns the Edge from vertex v directed toward this vertex.

e_in()
e_in() : list of edges directed toward this vertex.

e_out()
e_out(): list of edges directed outward this vertex.

e_to(*y*)
returns the Edge from this vertex directed toward vertex v.

e_with(*v*)
return the Edge with both this vertex and vertex v

neighbors(*direction=0*)
Returns the neighbors of this vertex. List of neighbor vertices in all directions (default) or in filtered f_io direction (>0 means outward).

Parameters **direction** –

- 0: parent and children
- -1: parents
- +1: children

Returns list of vertices

4.7 Module contents

CHAPTER

FIVE

FEATURIZE

class graphistry.feature_utils.**Embedding**(*df*)
Bases: object

Generates random embeddings of a given dimension that aligns with the index of the dataframe

Parameters **df** (DataFrame) –

fit(*n_dim*)

Parameters **n_dim** (int) –

fit_transform(*n_dim*)

Parameters **n_dim** (int) –

transform(*ids*)

Return type DataFrame

class graphistry.feature_utils.**FastEncoder**(*df*, *y=None*, *kind='nodes'*)
Bases: object

fit(*src=None*, *dst=None*, **args*, ***kwargs*)

fit_transform(*src=None*, *dst=None*, **args*, ***kwargs*)

scale(*X=None*, *y=None*, *return_pipeline=False*, **args*, ***kwargs*)
Fits new scaling functions on df, y via args-kwargs

Example:

```
from graphisty.features import SCALERS, SCALER_OPTIONS
print(SCALERS)
g = graphistry.nodes(df)
# set a scaling strategy for features and targets -- umap uses those and␣
↪produces different results depending.
g2 = g.umap(use_scaler='standard', use_scaler_target=None)

# later if you want to scale new data, you can do so
X, y = g2.transform(df, df, scaled=False)  # unscaled transformer output
# now scale with new settings
X_scaled, y_scaled = g2.scale(X, y, use_scaler='minmax', use_scaler_
↪target='kbins', n_bins=5)
# fit some other pipeline
clf.fit(X_scaled, y_scaled)
```

args:

```
;X: pd.DataFrame of features
:y: pd.DataFrame of target features
:kind: str, one of 'nodes' or 'edges'
*args, **kwargs: passed to smart_scaler pipeline
```

returns: scaled X, y

transform(*df*, *ydf=None*)
Raw transform, no scaling.

transform_scaled(*df*, *ydf=None*, *scaling_pipeline=None*, *scaling_pipeline_target=None*)

class graphistry.feature_utils.**FastMLB**(*mlb*, *in_column*, *out_columns*)
Bases: object

fit(*X*, *y=None*)

get_feature_names_in()

get_feature_names_out()

transform(*df*)

class graphistry.feature_utils.**FeatureMixin**(**args*, ***kwargs*)
Bases: object

FeatureMixin for automatic featurization of nodes and edges DataFrames. Subclasses UMAPMixin for umaping of automatic features.

Usage:

```
g = graphistry.nodes(df, 'node_column')
g2 = g.featurize()
```

or for edges,

```
g = graphistry.edges(df, 'src', 'dst')
g2 = g.featurize(kind='edges')
```

or chain them for both nodes and edges,

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node_column')
g2 = g.featurize().featurize(kind='edges')
```

featurize(*kind='nodes'*, *X=None*, *y=None*, *use_scaler=None*, *use_scaler_target=None*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=12*, *multilabel=False*, *embedding=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=4.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *impute=True*, *n_quantiles=100*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *similarity=None*, *categories='auto'*, *keep_n_decimals=5*, *remove_node_column=True*, *inplace=False*, *feature_engine='auto'*, *dbscan=False*, *min_dist=0.5*, *min_samples=1*, *memoize=True*, *verbose=False*)
Featurize Nodes or Edges of the underlying nodes/edges DataFrames.

Parameters

- **kind** (str) – specify whether to featurize *nodes* or *edges*. Edge featurization includes a pairwise src-to-dst feature block using a MultiLabelBinarizer, with any other columns being treated the same way as with *nodes* featurization.

- **X** (Union[List[str], str, DataFrame, None]) – Optional input, default None. If symbolic, evaluated against self data based on kind. If None, will featurize all columns of DataFrame
- **y** (Union[List[str], str, DataFrame, None]) – Optional Target(s) columns or explicit DataFrame, default None
- **use_scaler** (Optional[str]) – selects which scaler (and automatically imputes missing values using mean strategy) to scale the data. Options are; "minmax", "quantile", "standard", "robust", "kbins", default None. Please see scikits-learn documentation https://scikit-learn.org/stable/modules/preprocessing.html Here 'standard' corresponds to 'StandardScaler' in scikits.
- **cardinality_threshold** (int) – dirty_cat threshold on cardinality of categorical labels across columns. If value is greater than threshold, will run GapEncoder (a topic model) on column. If below, will one-hot_encode. Default 40.
- **cardinality_threshold_target** (int) – similar to cardinality_threshold, but for target features. Default is set high (400), as targets generally want to be one-hot encoded, but sometimes it can be useful to use GapEncoder (ie, set threshold lower) to create regressive targets, especially when those targets are textual/softly categorical and have semantic meaning across different labels. Eg, suppose a column has fields like ['Application Fraud', 'Other Statuses', 'Lost-Target scaling using/Stolen Fraud', 'Investigation Fraud', ...] the GapEncoder will concentrate the 'Fraud' labels together.
- **n_topics** (int) – the number of topics to use in the GapEncoder if cardinality_thresholds is saturated. Default is 42, but good rule of thumb is to consult the Johnson-Lindenstrauss Lemma https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma or use the simplified *random walk* estimate => n_topics_lower_bound ~ (pi/2) * (N-documents)**(1/4)
- **n_topics_target** (int) – the number of topics to use in the GapEncoder if cardinality_thresholds_target is saturated for the target(s). Default 12.
- **min_words** (float) – sets threshold on how many words to consider in a textual column if it is to be considered in the text processing pipeline. Set this very high if you want any textual columns to bypass the transformer, in favor of GapEncoder (topic modeling). Set to 0 to force all named columns to be encoded as textual (embedding)
- **model_name** (str) – Sentence Transformer model to use. Default Paraphrase model makes useful vectors, but at cost of encoding time. If faster encoding is needed, *average_word_embeddings_komninos* is useful and produces less semantically relevant vectors. Please see sentence_transformer (https://www.sbert.net/) library for all available models.
- **multilabel** (bool) – if True, will encode a *single* target column composed of lists of lists as multilabel outputs. This only works with y=['a_single_col'], default False
- **embedding** (bool) – If True, produces a random node embedding of size *n_topics* default, False. If no node features are provided, will produce random embeddings (for GNN models, for example)
- **use_ngrams** (bool) – If True, will encode textual columns as TfIdf Vectors, default, False.
- **ngram_range** (tuple) – if use_ngrams=True, can set ngram_range, eg: tuple = (1, 3)
- **max_df** (float) – if use_ngrams=True, set max word frequency to consider in vocabulary eg: max_df = 0.2,

- **min_df** (int) – if use_ngrams=True, set min word count to consider in vocabulary eg: min_df = 3 or 0.00001
- **categories** (Optional[str]) – Optional[str] in ["auto", "k-means", "most_frequent"], decides which category to select in Similarity Encoding, default 'auto'
- **impute** (bool) – Whether to impute missing values, default True
- **n_quantiles** (int) – if use_scaler = 'quantile', sets the quantile bin size.
- **output_distribution** (str) – if use_scaler = 'quantile', can return distribution as ["normal", "uniform"]
- **quantile_range** – if use_scaler = 'robust'|'quantile', sets the quantile range.
- **n_bins** (int) – number of bins to use in kbins discretizer, default 10
- **encode** (str) – encoding for KBinsDiscretizer, can be one of *onehot*, *onehot-dense*, *ordinal*, default 'ordinal'
- **strategy** (str) – strategy for KBinsDiscretizer, can be one of *uniform*, *quantile*, *kmeans*, default 'quantile'
- **n_quantiles** – if use_scaler = "quantile", sets the number of quantiles, default=100
- **output_distribution** – if use_scaler="quantile"|"robust", choose from ["normal", "uniform"]
- **dbscan** (bool) – whether to run DBSCAN, default False.
- **min_dist** (float) – DBSCAN eps parameter, default 0.5.
- **min_samples** (int) – DBSCAN min_samples parameter, default 5.
- **keep_n_decimals** (int) – number of decimals to keep
- **remove_node_column** (bool) – whether to remove node column so it is not featurized, default True.
- **inplace** (bool) – whether to not return new graphistry instance or not, default False.
- **memoize** (bool) – whether to store and reuse results across runs, default True.
- **use_scaler_target** (Optional[str]) –
- **similarity** (Optional[str]) –
- **feature_engine** (Literal[typing_extensions.Literal['none', 'pandas', 'dirty_cat', 'torch'], 'auto']) –
- **verbose** (bool) –

Returns graphistry instance with new attributes set by the featurization process.

get_matrix(*columns=None*, *kind='nodes'*, *target=False*)

Returns feature matrix, and if columns are specified, returns matrix with only the columns that contain the string *column_part* in their name.`X = g.get_matrix(['feature1', 'feature2'])` will retrieve a feature matrix with only the columns that contain the string *feature1* or *feature2* in their name. Most useful for topic modeling, where the column names are of the form *topic_0: descriptor*, *topic_1: descriptor*, etc. Can retrieve unique columns in original dataframe, or actual topic features like [ip_part, shoes, preference_x, etc]. Powerful way to retrieve features from a featurized graph by column or (top) features of interest.

Example:

```
# get the full feature matrices
X = g.get_matrix()
y = g.get_matrix(target=True)

# get subset of features, or topics, given topic model encoding
X = g2.get_matrix(['172', 'percent'])
X.columns
    => ['ip_172.56.104.67', 'ip_172.58.129.252', 'item_percent']
# or in targets
y = g2.get_matrix(['total', 'percent'], target=True)
y.columns
    => ['basket_price_total', 'conversion_percent', 'CTR_percent',
↪'CVR_percent']

# not as useful for sbert features.
```

Caveats:

- if you have a column name that is a substring of another column name, you may get unexpected results.

Args:

columns (Union[List, str]) list of column names or a single column name that may exist in columns of the feature matrix. If None, returns original feature matrix

kind (str, optional) Node or Edge features. Defaults to 'nodes'.

target (bool, optional) If True, returns the target matrix. Defaults to False.

Returns: pd.DataFrame: feature matrix with only the columns that contain the string *column_part* in their name.

Parameters

- **columns** (Union[List, str, None]) –
- **kind** (str) –
- **target** (bool) –

Return type DataFrame

scale(*df=None*, *y=None*, *kind='nodes'*, *use_scaler=None*, *use_scaler_target=None*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *return_scalers=False*)

Scale data using the same scalers as used in the featurization step.

Example

```
g = graphistry.nodes(df)
X, y = g.featurize().scale(kind='nodes', use_scaler='robust', use_scaler_
↪target='kbins', n_bins=3)

# or
g = graphistry.nodes(df)
# set a scaling strategy for features and targets -- umap uses those and␣
↪produces different results depending.
g2 = g.umap(use_scaler='standard', use_scaler_target=None)
```

(continues on next page)

(continued from previous page)

```
# later if you want to scale new data, you can do so
X, y = g2.transform(df, df, scale=False)
X_scaled, y_scaled = g2.scale(X, y, use_scaler='minmax', use_scaler_target=
↪'kbins', n_bins=5)
# fit some other pipeline
clf.fit(X_scaled, y_scaled)
```

Args:

df pd.DataFrame, raw data to transform, if None, will use data from featurization fit

y pd.DataFrame, optional target data

kind str, one of *nodes*, *edges*

use_scaler str, optional, one of *minmax*, *robust*, *standard*, *kbins*, *quantile*

use_scaler_target str, optional, one of *minmax*, *robust*, *standard*, *kbins*, *quantile*

impute bool, if True, will impute missing values

n_quantiles int, number of quantiles to use for quantile scaler

output_distribution str, one of *normal*, *uniform*, *lognormal*

quantile_range tuple, range of quantiles to use for quantile scaler

n_bins int, number of bins to use for KBinsDiscretizer

encode str, one of *ordinal*, *onehot*, *onehot-dense*, *binary*

strategy str, one of *uniform*, *quantile*, *kmeans*

keep_n_decimals int, number of decimals to keep after scaling

return_scalers bool, if True, will return the scalers used to scale the data

Returns:

(X, y) transformed data if return_graph is False or a graph with inferred edges if return_graph is True, or (X, y, scaler, scaler_target) if return_scalers is True

Parameters

- **df** (Optional[DataFrame]) –
- **y** (Optional[DataFrame]) –
- **kind** (str) –
- **use_scaler** (Optional[str]) –
- **use_scaler_target** (Optional[str]) –
- **impute** (bool) –
- **n_quantiles** (int) –
- **output_distribution** (str) –
- **n_bins** (int) –
- **encode** (str) –
- **strategy** (str) –

- **keep_n_decimals** (int) –
- **return_scalers** (bool) –

transform(*df*, *y=None*, *kind='nodes'*, *min_dist='auto'*, *n_neighbors=7*, *merge_policy=False*, *sample=None*, *return_graph=True*, *scaled=True*, *verbose=False*)

Transform new data and append to existing graph, or return dataframes

args:

df pd.DataFrame, raw data to transform

ydf pd.DataFrame, optional

kind str # one of *nodes*, *edges*

return_graph bool, if True, will return a graph with inferred edges.

merge_policy bool, if True, adds batch to existing graph nodes via nearest neighbors. If False, will infer edges only between nodes in the batch, default False

min_dist float, if return_graph is True, will use this value in NN search, or 'auto' to infer a good value. min_dist represents the maximum distance between two samples for one to be considered as in the neighborhood of the other.

sample int, if return_graph is True, will use sample edges of existing graph to fill out the new graph

n_neighbors int, if return_graph is True, will use this value for n_neighbors in Nearest Neighbors search

scaled bool, if True, will use scaled transformation of data set during featurization, default True

verbose bool, if True, will print metadata about the graph construction, default False

Returns:

X, y: pd.DataFrame, transformed data if return_graph is False or a graphistry Plottable with inferred edges if return_graph is True

Parameters

- **df** (DataFrame) –
- **y** (Optional[DataFrame]) –
- **kind** (str) –
- **min_dist** (Union[str, float, int]) –
- **n_neighbors** (int) –
- **merge_policy** (bool) –
- **sample** (Optional[int]) –
- **return_graph** (bool) –
- **scaled** (bool) –
- **verbose** (bool) –

graphistry.feature_utils.**assert_imported**()

graphistry.feature_utils.**assert_imported_text**()

class graphistry.feature_utils.**callThrough**(*x*)
Bases: object

graphistry.feature_utils.**check_if_textual_column**(*df*, *col*, *confidence=0.35*, *min_words=2.5*)
Checks if *col* column of df is textual or not using basic heuristics

Parameters

- **df** (DataFrame) – DataFrame
- **col** – column name
- **confidence** (float) – threshold float value between 0 and 1. If column *col* has *confidence* more elements as type *str* it will pass it onto next stage of evaluation. Default 0.35
- **min_words** (float) – mean minimum words threshold. If mean words across *col* is greater than this, it is deemed textual. Default 2.5

Return type bool

Returns bool, whether column is textual or not

graphistry.feature_utils.**concat_text**(*df*, *text_cols*)

graphistry.feature_utils.**encode_edges**(*edf*, *src*, *dst*, *mlb*, *fit=False*)
edge encoder – creates multilabelBinarizer on edge pairs.

Args: edf (pd.DataFrame): edge dataframe src (string): source column dst (string): destination column mlb (sklearn): multilabelBinarizer fit (bool, optional): If true, fits multilabelBinarizer. Defaults to False.

Returns tuple: pd.DataFrame, multilabelBinarizer

graphistry.feature_utils.**encode_multi_target**(*ydf*, *mlb=None*)

graphistry.feature_utils.**encode_textual**(*df*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*)

Parameters

- **df** (DataFrame) –
- **min_words** (float) –
- **model_name** (str) –
- **use_ngrams** (bool) –
- **ngram_range** (tuple) –
- **max_df** (float) –
- **min_df** (int) –

Return type Tuple[DataFrame, List, Any]

graphistry.feature_utils.**features_without_target**(*df*, *y=None*)
Checks if y DataFrame column name is in df, and removes it from df if so

Parameters

- **df** (DataFrame) – model DataFrame
- **y** (Union[List, str, DataFrame, None]) – target DataFrame

Return type DataFrame

Returns DataFrames of model and target

graphistry.feature_utils.**find_bad_set_columns**(*df*, *bad_set=['[]']*)
Finds columns that if not coerced to strings, will break processors.

Parameters

- **df** (DataFrame) – DataFrame
- **bad_set** (List) – List of strings to look for.

Returns list

graphistry.feature_utils.**fit_pipeline**(*X*, *transformer*, *keep_n_decimals=5*)
Helper to fit DataFrame over transformer pipeline. Rounds resulting matrix X by keep_n_digits if not 0, which helps for when transformer pipeline is scaling or imputer which sometime introduce small negative numbers, and umap metrics like Hellinger need to be positive :type X: DataFrame :param X: DataFrame to transform. :param transformer: Pipeline object to fit and transform :type keep_n_decimals: int :param keep_n_decimals: Int of how many decimal places to keep in rounded transformed data

Return type DataFrame

graphistry.feature_utils.**get_cardinality_ratio**(*df*)
Calculates the ratio of unique values to total number of rows of DataFrame

Parameters **df** (DataFrame) – DataFrame

graphistry.feature_utils.**get_dataframe_by_column_dtype**(*df*, *include=None*, *exclude=None*)

graphistry.feature_utils.**get_matrix_by_column_part**(*X*, *column_part*)
Get the feature matrix by column part existing in column names.

Parameters

- **X** (DataFrame) –
- **column_part** (str) –

Return type DataFrame

graphistry.feature_utils.**get_matrix_by_column_parts**(*X*, *column_parts*)
Get the feature matrix by column parts list existing in column names.

Parameters

- **X** (DataFrame) –
- **column_parts** (Union[list, str, None]) –

Return type DataFrame

graphistry.feature_utils.**get_numeric_transformers**(*ndf*, *y=None*)

graphistry.feature_utils.**get_preprocessing_pipeline**(*use_scaler='robust'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='quantile'*)
Helper function for imputing and scaling np.ndarray data using different scaling transformers.

Parameters

- **X** – np.ndarray
- **impute** (bool) – whether to run imputing or not

- **use_scaler** (str) – string in None or ["minmax", "quantile", "standard", "robust", "kbins"], selects scaling transformer, default None
- **n_quantiles** (int) – if use_scaler = 'quantile', sets the quantile bin size.
- **output_distribution** (str) – if use_scaler = 'quantile', can return distribution as ["normal", "uniform"]
- **quantile_range** – if use_scaler = 'robust'/'quantile', sets the quantile range.
- **n_bins** (int) – number of bins to use in kbins discretizer
- **encode** (str) – encoding for KBinsDiscretizer, can be one of *onehot*, *onehot-dense*, *ordinal*, default 'ordinal'
- **strategy** (str) – strategy for KBinsDiscretizer, can be one of *uniform*, *quantile*, *kmeans*, default 'quantile'

Return type Any

Returns scaled array, imputer instances or None, scaler instance or None

graphistry.feature_utils.**get_text_preprocessor**(*ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*)

graphistry.feature_utils.**get_textual_columns**(*df*, *min_words=2.5*)

Collects columns from df that it deems are textual.

Parameters

- **df** (DataFrame) – DataFrame
- **min_words** (float) –

Return type List

Returns list of columns names

graphistry.feature_utils.**group_columns_by_dtypes**(*df*, *verbose=True*)

Parameters

- **df** (DataFrame) –
- **verbose** (bool) –

Return type Dict

graphistry.feature_utils.**identity**(*x*)

graphistry.feature_utils.**impute_and_scale_df**(*df*, *use_scaler='robust'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*)

Parameters

- **df** (DataFrame) –
- **use_scaler** (str) –
- **impute** (bool) –
- **n_quantiles** (int) –
- **output_distribution** (str) –

- **n_bins** (int) –
- **encode** (str) –
- **strategy** (str) –
- **keep_n_decimals** (int) –

Return type Tuple[DataFrame, Any]

graphistry.feature_utils.**is_dataframe_all_numeric**(*df*)

Parameters **df** (DataFrame) –

Return type bool

graphistry.feature_utils.**lazy_import_has_dependancy_text**()

graphistry.feature_utils.**lazy_import_has_min_dependancy**()

graphistry.feature_utils.**make_array**(*X*)

graphistry.feature_utils.**passthrough_df_cols**(*df*, *columns*)

graphistry.feature_utils.**process_dirty_dataframes**(*ndf*, *y*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *similarity=None*, *categories='auto'*, *multilabel=False*)

Dirty_Cat encoder for record level data. Will automatically turn inhomogeneous dataframe into matrix using smart conversion tricks.

Parameters

- **ndf** (DataFrame) – node DataFrame
- **y** (Optional[DataFrame]) – target DataFrame or series
- **cardinality_threshold** (int) – For ndf columns, below this threshold, encoder is OneHot, above, it is GapEncoder
- **cardinality_threshold_target** (int) – For target columns, below this threshold, encoder is OneHot, above, it is GapEncoder
- **n_topics** (int) – number of topics for GapEncoder, default 42
- **use_scaler** – None or string in [‘minmax’, ‘standard’, ‘robust’, ‘quantile’]
- **similarity** (Optional[str]) – one of ‘ngram’, ‘levenshtein-ratio’, ‘jaro’, or’jaro-winkler’}) – The type of pairwise string similarity to use. If None or False, uses a SuperVectorizer
- **n_topics_target** (int) –
- **categories** (Optional[str]) –
- **multilabel** (bool) –

Return type Tuple[DataFrame, Optional[DataFrame], Any, Any]

Returns Encoded data matrix and target (if not None), the data encoder, and the label encoder.

graphistry.feature_utils.**process_edge_dataframes**(*edf*, *y*, *src*, *dst*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *use_scaler=None*, *use_scaler_target=None*, *multilabel=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *similarity=None*, *categories='auto'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *feature_engine='pandas'*)

Custom Edge-record encoder. Uses a MultiLabelBinarizer to generate a src/dst vector and then process_textual_or_other_dataframes that encodes any other data present in edf, textual or not.

Parameters

- **edf** (DataFrame) – pandas DataFrame of edge features
- **y** (DataFrame) – pandas DataFrame of edge labels
- **src** (str) – source column to select in edf
- **dst** (str) – destination column to select in edf
- **use_scaler** (Optional[str]) – None or string in [‘minmax’, ‘standard’, ‘robust’, ‘quantile’]
- **cardinality_threshold** (int) –
- **cardinality_threshold_target** (int) –
- **n_topics** (int) –
- **n_topics_target** (int) –
- **use_scaler_target** (Optional[str]) –
- **multilabel** (bool) –
- **use_ngrams** (bool) –
- **ngram_range** (tuple) –
- **max_df** (float) –
- **min_df** (int) –
- **min_words** (float) –
- **model_name** (str) –
- **similarity** (Optional[str]) –
- **categories** (Optional[str]) –
- **impute** (bool) –

- **n_quantiles** (int) –
- **output_distribution** (str) –
- **n_bins** (int) –
- **encode** (str) –
- **strategy** (str) –
- **keep_n_decimals** (int) –
- **feature_engine** (Literal['none', 'pandas', 'dirty_cat', 'torch']) –

Return type Tuple[DataFrame, DataFrame, DataFrame, DataFrame, List[Any], Any, Optional[Any], Optional[Any], Any, List[str]]

Returns Encoded data matrix and target (if not None), the data encoders, and the label encoder.

graphistry.feature_utils.**process_nodes_dataframes**(*df*, *y*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *use_scaler='robust'*, *use_scaler_target='kbins'*, *multilabel=False*, *embedding=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *similarity=None*, *categories='auto'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *feature_engine='pandas'*)

Automatic Deep Learning Embedding/ngrams of Textual Features, with the rest of the columns taken care of by dirty_cat

Parameters

- **df** (DataFrame) – pandas DataFrame of data
- **y** (DataFrame) – pandas DataFrame of targets
- **use_scaler** (Optional[str]) – None or string in ['minmax', 'standard', 'robust', 'quantile']
- **n_topics** (int) – number of topics in Gap Encoder
- **use_scaler** –
- **confidence** – Number between 0 and 1, will pass column for textual processing if total entries are string like in a column and above this relative threshold.
- **min_words** (float) – Sets the threshold for average number of words to include column for textual sentence encoding. Lower values means that columns will be labeled textual and sent to sentence-encoder. Set to 0 to force named columns as textual.
- **model_name** (str) – SentenceTransformer model name. See available list at https://www.sbert.net/docs/pretrained_models. html#sentence-embedding-models

- **cardinality_threshold** (int) –
- **cardinality_threshold_target** (int) –
- **n_topics_target** (int) –
- **use_scaler_target** (Optional[str]) –
- **multilabel** (bool) –
- **embedding** (bool) –
- **use_ngrams** (bool) –
- **ngram_range** (tuple) –
- **max_df** (float) –
- **min_df** (int) –
- **similarity** (Optional[str]) –
- **categories** (Optional[str]) –
- **impute** (bool) –
- **n_quantiles** (int) –
- **output_distribution** (str) –
- **n_bins** (int) –
- **encode** (str) –
- **strategy** (str) –
- **keep_n_decimals** (int) –
- **feature_engine** (Literal['none', 'pandas', 'dirty_cat', 'torch']) –

Return type Tuple[DataFrame, Any, DataFrame, Any, Any, Any, Optional[Any], Optional[Any], Any, List[str]]

Returns X_enc, y_enc, data_encoder, label_encoder, scaling_pipeline, scaling_pipeline_target, text_model, text_cols,

graphistry.feature_utils.**prune_weighted_edges_df_and_relabel_nodes**(*wdf*, *scale=0.1*, *index_to_nodes_dict=None*)

Prune the weighted edge DataFrame so to return high fidelity similarity scores.

Parameters

- **wdf** (DataFrame) – weighted edge DataFrame gotten via UMAP
- **scale** (float) – lower values means less edges > (max - scale * std)
- **index_to_nodes_dict** (Optional[Dict]) – dict of index to node name; remap src/dst values if provided

Return type DataFrame

Returns pd.DataFrame

graphistry.feature_utils.**remove_internal_namespace_if_present**(*df*)

Some tranformations below add columns to the DataFrame, this method removes them before featurization Will not drop if suffix is added during UMAP-ing

Parameters **df** (DataFrame) – DataFrame

Returns DataFrame with dropped columns in reserved namespace

graphistry.feature_utils.**remove_node_column_from_symbolic**(*X_symbolic*, *node*)

graphistry.feature_utils.**resolve_X**(*df*, *X*)

Parameters

- **df** (Optional[DataFrame]) –
- **X** (Union[List[str], str, DataFrame, None]) –

Return type DataFrame

graphistry.feature_utils.**resolve_feature_engine**(*feature_engine*)

Parameters **feature_engine** (Literal[typing_extensions.Literal['none', 'pandas', 'dirty_cat', 'torch'], 'auto']) –

Return type Literal['none', 'pandas', 'dirty_cat', 'torch']

graphistry.feature_utils.**resolve_y**(*df*, *y*)

Parameters

- **df** (Optional[DataFrame]) –
- **y** (Union[List[str], str, DataFrame, None]) –

Return type DataFrame

graphistry.feature_utils.**reuse_featurization**(*g*, *memoize*, *metadata*)

Parameters

- **g** (Plottable) –
- **memoize** (bool) –
- **metadata** (Any) –

graphistry.feature_utils.**safe_divide**(*a*, *b*)

graphistry.feature_utils.**set_currency_to_float**(*df*, *col*, *return_float=True*)

Parameters

- **df** (DataFrame) –
- **col** (str) –
- **return_float** (bool) –

graphistry.feature_utils.**set_to_bool**(*df*, *col*, *value*)

Parameters

- **df** (DataFrame) –
- **col** (str) –
- **value** (Any) –

graphistry.feature_utils.**set_to_datetime**(*df*, *cols*, *new_col*)

Parameters

- **df** (DataFrame) –

- **cols** (List) –
- **new_col** (str) –

graphistry.feature_utils.**set_to_numeric**(*df*, *cols*, *fill_value=0.0*)

Parameters

- **df** (DataFrame) –
- **cols** (List) –
- **fill_value** (float) –

graphistry.feature_utils.**smart_scaler**(*X_enc*, *y_enc*, *use_scaler*, *use_scaler_target*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*)

Parameters

- **impute** (bool) –
- **n_quantiles** (int) –
- **output_distribution** (str) –
- **n_bins** (int) –
- **encode** (str) –
- **strategy** (str) –
- **keep_n_decimals** (int) –

graphistry.feature_utils.**transform**(*df*, *ydf*, *res*, *kind*, *src*, *dst*)

Parameters

- **df** (DataFrame) –
- **ydf** (DataFrame) –
- **res** (List) –
- **kind** (str) –

Return type Tuple[DataFrame, DataFrame]

graphistry.feature_utils.**transform_dirty**(*df*, *data_encoder*, *name=''*)

Parameters

- **df** (DataFrame) –
- **data_encoder** (Any) –
- **name** (str) –

Return type DataFrame

graphistry.feature_utils.**transform_text**(*df*, *text_model*, *text_cols*)

Parameters

- **df** (DataFrame) –
- **text_model** (Any) –
- **text_cols** (Union[List, str]) –

Return type DataFrame

graphistry.feature_utils.**where_is_currency_column**(*df*, *col*)

Parameters

- **df** (DataFrame) –
- **col** (str) –

CHAPTER

SIX

UMAP

class graphistry.umap_utils.**UMAPMixin**(**args*, ***kwargs*)

Bases: object

UMAP Mixin for automagic UMAPing

filter_weighted_edges(*scale=1.0*, *index_to_nodes_dict=None*, *inplace=False*, *kind='nodes'*)

Filter edges based on _weighted_edges_df (ex: from .umap())

Parameters

- **scale** (float) –
- **index_to_nodes_dict** (Optional[Dict]) –
- **inplace** (bool) –
- **kind** (str) –

transform_umap(*df*, *y=None*, *kind='nodes'*, *min_dist='auto'*, *n_neighbors=7*, *merge_policy=False*, *sample=None*, *return_graph=True*, *fit_umap_embedding=True*, *verbose=False*)

Transforms data into UMAP embedding

Args:

df Dataframe to transform

y Target column

kind One of *nodes* or *edges*

min_dist Epsilon for including neighbors in infer_graph

n_neighbors Number of neighbors to use for contextualization

merge_policy if True, use previous graph, adding new batch to existing graph's neighbors useful to contextualize new data against existing graph. If False, *sample* is irrelevant.

sample Sample number of existing graph's neighbors to use for contextualization – helps make denser graphs

return_graph Whether to return a graph or just the embeddings

fit_umap_embedding Whether to infer graph from the UMAP embedding on the new data

verbose Whether to print information about the graph inference

Parameters

- **df** (DataFrame) –
- **y** (Optional[DataFrame]) –

- **kind** (str) –
- **min_dist** (Union[str, float, int]) –
- **n_neighbors** (int) –
- **merge_policy** (bool) –
- **sample** (Optional[int]) –
- **return_graph** (bool) –
- **fit_umap_embedding** (bool) –
- **verbose** (bool) –

Return type Union[Tuple[DataFrame, DataFrame, DataFrame], Plottable]

umap(*X=None*, *y=None*, *kind='nodes'*, *scale=1.0*, *n_neighbors=12*, *min_dist=0.1*, *spread=0.5*, *local_connectivity=1*, *repulsion_strength=1*, *negative_sample_rate=5*, *n_components=2*, *metric='euclidean'*, *suffix=''*, *play=0*, *encode_position=True*, *encode_weight=True*, *dbscan=False*, *engine='auto'*, *feature_engine='auto'*, *inplace=False*, *memoize=True*, *verbose=False*, ***featurize_kwargs*)

UMAP the featurized nodes or edges data, or pass in your own X, y (optional) dataframes of values

Example

```
>>> import graphistry
>>> g = graphistry.nodes(pd.DataFrame({'node': [0,1,2], 'data': [1,2,3], 'meta
↪': ['a', 'b', 'c']}))
>>> g2 = g.umap(n_components=3, spread=1.0, min_dist=0.1, n_neighbors=12,␣
↪negative_sample_rate=5, local_connectivity=1, repulsion_strength=1.0,␣
↪metric='euclidean', suffix='', play=0, encode_position=True, encode_
↪weight=True, dbscan=False, engine='auto', feature_engine='auto',␣
↪inplace=False, memoize=True, verbose=False)
>>> g2.plot()
```

Parameters

X either a dataframe ndarray of features, or column names to featurize

y either an dataframe ndarray of targets, or column names to featurize targets

kind *nodes* or *edges* or None. If None, expects explicit X, y (optional) matrices, and will Not associate them to nodes or edges. If X, y (optional) is given, with kind = [nodes, edges], it will associate new matrices to nodes or edges attributes.

scale multiplicative scale for pruning weighted edge DataFrame gotten from UMAP, between [0, ..) with high end meaning keep all edges

n_neighbors UMAP number of nearest neighbors to include for UMAP connectivity, lower makes more compact layouts. Minimum 2

min_dist UMAP float between 0 and 1, lower makes more compact layouts.

spread UMAP spread of values for relaxation

local_connectivity UMAP connectivity parameter

repulsion_strength UMAP repulsion strength

negative_sample_rate UMAP negative sampling rate

n_components number of components in the UMAP projection, default 2

metric UMAP metric, default 'euclidean'. see (UMAP-LEARN)[https://umap-learn.readthedocs.io/ en/latest/parameters.html] documentation for more.

suffix optional suffix to add to x, y attributes of umap.

play Graphistry play parameter, default 0, how much to evolve the network during clustering. 0 preserves the original UMAP layout.

encode_weight if True, will set new edges_df from implicit UMAP, default True.

encode_position whether to set default plotting bindings – positions x,y from umap for .plot(), default True

dbscan whether to run DBSCAN on the UMAP embedding, default False.

engine selects which engine to use to calculate UMAP: default "auto" will use cuML if available, otherwise UMAP-LEARN.

feature_engine How to encode data ("none", "auto", "pandas", "dirty_cat", "torch")

inplace bool = False, whether to modify the current object, default False. when False, returns a new object, useful for chaining in a functional paradigm.

memoize whether to memoize the results of this method, default True.

verbose whether to print out extra information, default False.

Returns self, with attributes set with new data

Parameters

- **X** (Union[List[str], str, DataFrame, None]) –
- **y** (Union[List[str], str, DataFrame, None]) –
- **kind** (str) –
- **scale** (float) –
- **n_neighbors** (int) –
- **min_dist** (float) –
- **spread** (float) –
- **local_connectivity** (int) –
- **repulsion_strength** (float) –
- **negative_sample_rate** (int) –
- **n_components** (int) –
- **metric** (str) –
- **suffix** (str) –
- **play** (Optional[int]) –
- **encode_position** (bool) –
- **encode_weight** (bool) –
- **dbscan** (bool) –
- **engine** (Literal[typing_extensions.Literal['cuml', 'umap_learn'], 'auto']) –
- **feature_engine** (str) –

- **inplace** (bool) –
- **memoize** (bool) –
- **verbose** (bool) –

umap_fit (*X*, *y=None*, *verbose=False*)

Parameters

- **X** (DataFrame) –
- **y** (Optional[DataFrame]) –

umap_lazy_init (*res*, *n_neighbors=12*, *min_dist=0.1*, *spread=0.5*, *local_connectivity=1*, *repulsion_strength=1*, *negative_sample_rate=5*, *n_components=2*, *metric='euclidean'*, *engine='auto'*, *suffix=''*, *verbose=False*)

Parameters

- **n_neighbors** (int) –
- **min_dist** (float) –
- **spread** (float) –
- **local_connectivity** (int) –
- **repulsion_strength** (float) –
- **negative_sample_rate** (int) –
- **n_components** (int) –
- **metric** (str) –
- **engine** (Literal[typing_extensions.Literal['cuml', 'umap_learn'], 'auto']) –
- **suffix** (str) –
- **verbose** (bool) –

graphistry.umap_utils.**assert_imported**()

graphistry.umap_utils.**assert_imported_cuml**()

graphistry.umap_utils.**is_legacy_cuml**()

graphistry.umap_utils.**lazy_cuml_import_has_dependancy**()

graphistry.umap_utils.**lazy_umap_import_has_dependancy**()

graphistry.umap_utils.**resolve_umap_engine** (*engine*)

Parameters **engine** (Literal[typing_extensions.Literal['cuml', 'umap_learn'], 'auto']) –

Return type Literal['cuml', 'umap_learn']

graphistry.umap_utils.**reuse_umap** (*g*, *memoize*, *metadata*)

Parameters

- **g** (Plottable) –
- **memoize** (bool) –
- **metadata** (Any) –

graphistry.umap_utils.**umap_graph_to_weighted_edges**(*umap_graph*, *engine*, *is_legacy*, *cfg=<module 'graphistry.constants' from '/home/docs/checkouts/readthedocs.org/user_builds/pygrap fixes/graphistry/constants.py'>*)

CHAPTER

SEVEN

SEMANTIC SEARCH

class graphistry.text_utils.**SearchToGraphMixin**(**args*, ***kwargs*)

Bases: object

assert_features_line_up_with_nodes()

assert_fitted()

build_index(*angular=False*, *n_trees=None*)

classmethod load_search_instance(*savepath*)

save_search_instance(*savepath*)

search(*query*, *cols=None*, *thresh=5000*, *fuzzy=True*, *top_n=10*)

Natural language query over nodes that returns a dataframe of results sorted by relevance column "distance".

If node data is not yet feature-encoded (and explicit edges are given), run automatic feature engineering:

```
g2 = g.featurize(kind='nodes', X=['text_col_1', ..],
min_words=0 # forces all named columns are textually encoded
)
```

If edges do not yet exist, generate them via

```
g2 = g.umap(kind='nodes', X=['text_col_1', ..],
min_words=0 # forces all named columns are textually encoded
)
```

If an index is not yet built, it is generated *g2.build_index()* on the fly at search time. Otherwise, can set *g2.build_index()* and then subsequent *g2.search(...)* calls will be not rebuilt index.

Args:

query (str) natural language query.

cols (list or str, optional) if fuzzy=False, select which column to query. Defaults to None since fuzzy=True by defaul.

thresh (float, optional) distance threshold from query vector to returned results. Defaults to 5000, set large just in case, but could be as low as 10.

fuzzy (bool, optional) if True, uses embedding + annoy index for recall, otherwise does string matching over given *cols* Defaults to True.

top_n (int, optional) how many results to return. Defaults to 100.

Returns: pd.DataFrame, vector_encoding_of_query: rank ordered dataframe of results matching query

vector encoding of query via given transformer/ngrams model if fuzzy=True else None

Parameters

- **query** (str) –
- **thresh** (float) –
- **fuzzy** (bool) –
- **top_n** (int) –

search_graph(*query*, *scale=0.5*, *top_n=100*, *thresh=5000*, *broader=False*, *inplace=False*)

Input a natural language query and return a graph of results. See help(g.search) for more information

Args:

query (str) query input eg "coding best practices"

scale (float, optional) edge weigh threshold, Defaults to 0.5.

top_n (int, optional) how many results to return. Defaults to 100.

thresh (float, optional) distance threshold from query vector to returned results. Defaults to 5000, set large just in case, but could be as low as 10.

broader (bool, optional) if True, will retrieve entities connected via an edge that were not necessarily bubbled up in the results_dataframe. Defaults to False.

inplace (bool, optional) whether to return new instance (default) or mutate self. Defaults to False.

Returns: graphistry Instance: g

Parameters

- **query** (str) –
- **scale** (float) –
- **top_n** (int) –
- **thresh** (float) –
- **broader** (bool) –
- **inplace** (bool) –

CHAPTER

EIGHT

DBSCAN

class graphistry.compute.cluster.**ClusterMixin**(**args*, ***kwargs*)

Bases: object

dbscan(*min_dist=0.2*, *min_samples=1*, *cols=None*, *kind='nodes'*, *fit_umap_embedding=True*, *target=False*, *verbose=False*, **args*, ***kwargs*)

DBSCAN clustering on cpu or gpu infered automatically. Adds a *_dbscan* column to nodes or edges.

Examples:

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')

# cluster by UMAP embeddings
kind = 'nodes' | 'edges'
g2 = g.umap(kind=kind).dbscan(kind=kind)
print(g2._nodes['_dbscan']) | print(g2._edges['_dbscan'])

# dbscan in umap or featurize API
g2 = g.umap(dbscan=True, min_dist=1.2, min_samples=2, **kwargs)
# or, here dbscan is infered from features, not umap embeddings
g2 = g.featurize(dbscan=True, min_dist=1.2, min_samples=2, **kwargs)

# and via chaining,
g2 = g.umap().dbscan(min_dist=1.2, min_samples=2, **kwargs)

# cluster by feature embeddings
g2 = g.featurize().dbscan(**kwargs)

# cluster by a given set of feature column attributes, or with target=True
g2 = g.featurize().dbscan(cols=['ip_172', 'location', 'alert'], target=False,␣
↪**kwargs)

# equivalent to above (ie, cols != None and umap=True will still use features␣
↪dataframe, rather than UMAP embeddings)
g2 = g.umap().dbscan(cols=['ip_172', 'location', 'alert'], umap=True | False,␣
↪**kwargs)

g2.plot() # color by `_dbscan` column
```

Useful: Enriching the graph with cluster labels from UMAP is useful for visualizing clusters in the graph by color, size, etc, as well as assessing metrics per cluster, e.g. https://github.com/graphistry/pygraphistry/blob/master/demos/ai/cyber/cyber-redteam-umap-demo.ipynb

Args:

min_dist float The maximum distance between two samples for them to be considered as in the same neighborhood.

kind str 'nodes' or 'edges'

cols list of columns to use for clustering given *g.featurize* has been run, nice way to slice features or targets by fragments of interest, e.g. ['ip_172', 'location', 'ssh', 'warnings']

fit_umap_embedding bool whether to use UMAP embeddings or features dataframe to cluster DBSCAN

min_samples The number of samples in a neighborhood for a point to be considered as a core point. This includes the point itself.

target whether to use the target column as the clustering feature

Parameters

- **min_dist** (float) –
- **min_samples** (int) –
- **cols** (Union[List, str, None]) –
- **kind** (str) –
- **fit_umap_embedding** (bool) –
- **target** (bool) –
- **verbose** (bool) –

transform_dbscan(*df*, *y=None*, *min_dist='auto'*, *infer_umap_embedding=False*, *sample=None*, *n_neighbors=None*, *kind='nodes'*, *return_graph=True*, *verbose=False*)

Transforms a minibatch dataframe to one with a new column '_dbscan' containing the DBSCAN cluster labels on the minibatch and generates a graph with the minibatch and the original graph, with edges between the minibatch and the original graph inferred from the umap embedding or features dataframe. Graph nodes | edges will be colored by '_dbscan' column.

Examples:

```
fit:
    g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')
    g2 = g.featurize().dbscan()

predict:
::

    emb, X, _, ndf = g2.transform_dbscan(ndf, return_graph=False)
    # or
    g3 = g2.transform_dbscan(ndf, return_graph=True)
    g3.plot()
```

likewise for umap:

```
fit:
    g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')
    g2 = g.umap(X=.., y=..).dbscan()

predict:
::
```

(continues on next page)

(continued from previous page)

```
    emb, X, y, ndf = g2.transform_dbscan(ndf, ndf, return_graph=False)
    # or
    g3 = g2.transform_dbscan(ndf, ndf, return_graph=True)
    g3.plot()
```

Args:

df dataframe to transform

y optional labels dataframe

min_dist The maximum distance between two samples for them to be considered as in the same neighborhood. smaller values will result in less edges between the minibatch and the original graph. Default 'auto', infers min_dist from the mean distance and std of new points to the original graph

fit_umap_embedding whether to use UMAP embeddings or features dataframe when inferring edges between the minibatch and the original graph. Default False, uses the features dataframe

sample number of samples to use when inferring edges between the minibatch and the original graph, if None, will only use closest point to the minibatch. If greater than 0, will sample the closest *sample* points in existing graph to pull in more edges. Default None

kind 'nodes' or 'edges'

return_graph whether to return a graph or the (emb, X, y, minibatch df enriched with DBSCAN labels), default True infered graph supports kind='nodes' only.

verbose whether to print out progress, default False

Parameters

- **df** (DataFrame) –
- **y** (Optional[DataFrame]) –
- **min_dist** (Union[float, str]) –
- **infer_umap_embedding** (bool) –
- **sample** (Optional[int]) –
- **n_neighbors** (Optional[int]) –
- **kind** (str) –
- **return_graph** (bool) –
- **verbose** (bool) –

graphistry.compute.cluster.**dbscan_fit**(*g*, *dbscan*, *kind='nodes'*, *cols=None*, *use_umap_embedding=True*, *target=False*, *verbose=False*)

Fits clustering on UMAP embeddings if umap is True, otherwise on the features dataframe or target dataframe if target is True.

Args:

g graphistry graph

kind 'nodes' or 'edges'

cols list of columns to use for clustering given *g.featurize* has been run

use_umap_embedding whether to use UMAP embeddings or features dataframe for clustering (default: True)

Parameters

- **g** (Any) –
- **dbscan** (Any) –
- **kind** (str) –
- **cols** (Union[List, str, None]) –
- **use_umap_embedding** (bool) –
- **target** (bool) –
- **verbose** (bool) –

graphistry.compute.cluster.**dbscan_predict**(*X*, *model*)

DBSCAN has no predict per se, so we reverse engineer one here from https://stackoverflow.com/questions/27822752/scikit-learn-predicting-new-points-with-dbscan

Parameters

- **X** (DataFrame) –
- **model** (Any) –

graphistry.compute.cluster.**get_model_matrix**(*g*, *kind*, *cols*, *umap*, *target*)

Allows for a single function to get the model matrix for both nodes and edges as well as targets, embeddings, and features

Args:

g graphistry graph

kind 'nodes' or 'edges'

cols list of columns to use for clustering given *g.featurize* has been run

umap whether to use UMAP embeddings or features dataframe

target whether to use the target dataframe or features dataframe

Returns: pd.DataFrame: dataframe of model matrix given the inputs

Parameters

- **kind** (str) –
- **cols** (Union[List, str, None]) –

graphistry.compute.cluster.**lazy_dbscan_import_has_dependency**()

graphistry.compute.cluster.**resolve_cpu_gpu_engine**(*engine*)

Parameters **engine** (Literal[typing_extensions.Literal['cuml', 'umap_learn'], 'auto']) –

Return type Literal['cuml', 'umap_learn']

CHAPTER

NINE

ARROW UPLOADER MODULE

class graphistry.arrow_uploader.**ArrowUploader**(*server_base_path='http://nginx'*, *view_base_path='http://localhost'*, *name=None*, *description=None*, *edges=None*, *nodes=None*, *node_encodings=None*, *edge_encodings=None*, *token=None*, *dataset_id=None*, *metadata=None*, *certificate_validation=True*, *org_name=None*)

Bases: object

Parameters **org_name** (Optional[str]) –

arrow_to_buffer(*table*)

Parameters **table** (Table) –

cascade_privacy_settings(*mode=None*, *notify=None*, *invited_users=None*, *mode_action=None*, *message=None*)

Cascade:

- local (passed in)
- global
- hard-coded

Parameters

- **mode** (Optional[str]) –
- **notify** (Optional[bool]) –
- **invited_users** (Optional[List]) –
- **mode_action** (Optional[str]) –
- **message** (Optional[str]) –

property certificate_validation

create_dataset(*json*)

property dataset_id

Return type str

property description

Return type str

property edge_encodings

property edges

Return type Table

g_to_edge_bindings(*g*)

g_to_edge_encodings(*g*)

g_to_node_bindings(*g*)

g_to_node_encodings(*g*)

login(*username*, *password*, *org_name=None*)

maybe_bindings(*g*, *bindings*, *base={}*)

maybe_post_share_link(*g*)
Skip if never called .privacy() Return True/False based on whether called

Return type bool

property metadata

property name

Return type str

property node_encodings

property nodes

Return type Table

property org_name

Return type Optional[str]

pkey_login(*personal_key_id*, *personal_key_secret*, *org_name=None*)

post(*as_files=True*, *memoize=True*)
Note: likely want to pair with self.maybe_post_share_link(g)

Parameters

- **as_files** (bool) –
- **memoize** (bool) –

post_arrow(*arr*, *graph_type*, *opts=''*)

Parameters

- **arr** (Table) –
- **graph_type** (str) –
- **opts** (str) –

post_arrow_generic(*sub_path*, *tok*, *arr*, *opts=''*)

Parameters

- **sub_path** (str) –
- **tok** (str) –
- **arr** (Table) –

Return type Response

post_edges_arrow(*arr=None*, *opts=''*)

post_edges_file(*file_path*, *file_type='csv'*)

post_file(*file_path*, *graph_type='edges'*, *file_type='csv'*)

post_g(*g*, *name=None*, *description=None*)
Warning: main post() does not call this

post_nodes_arrow(*arr=None*, *opts=''*)

post_nodes_file(*file_path*, *file_type='csv'*)

post_share_link(*obj_pk*, *obj_type='dataset'*, *privacy=None*)
Set sharing settings. Any settings not passed here will cascade from PyGraphistry or defaults

Parameters

- **obj_pk** (str) –
- **obj_type** (str) –
- **privacy** (Optional[dict]) –

refresh(*token=None*)

property server_base_path

Return type str

sso_get_token(*state*)
Koa, 04 May 2022 Use state to get token

sso_login(*org_name=None*, *idp_name=None*)
Koa, 04 May 2022 Get SSO login auth_url or token

property token

Return type str

verify(*token=None*)

Return type bool

property view_base_path

Return type str

CHAPTER

TEN

ARROW FILE UPLOADER MODULE

class graphistry.ArrowFileUploader.**ArrowFileUploader**(*uploader*)

Bases: object

Implement file API with focus on Arrow support

Memoization in this class is based on reference equality, while plotter is based on hash. That means the plotter resolves different-identity value matches, so by the time ArrowFileUploader compares, identities are unified for faster reference-based checks.

Example: Upload files with per-session memoization uploader : ArrowUploader arr : pa.Table afu = ArrowFileUploader(uploader)

file1_id = afu.create_and_post_file(arr)[0] file2_id = afu.create_and_post_file(arr)[0]

assert file1_id == file2_id # memoizes by default (memory-safe: weak refs)

Example: Explicitly create a file and upload data for it uploader : ArrowUploader arr : pa.Table afu = ArrowFileUploader(uploader)

file1_id = afu.create_file() afu.post_arrow(arr, file_id)

file2_id = afu.create_file() afu.post_arrow(arr, file_id)

assert file1_id != file2_id

create_and_post_file(*arr*, *file_id=None*, *file_opts={}*, *upload_url_opts='erase=true'*, *memoize=True*)

Create file and upload data for it.

Default upload_url_opts='erase=true' throws exceptions on parse errors and deletes upload.

Default memoize=True skips uploading 'arr' when previously uploaded in current session

See File REST API for file_opts (file create) and upload_url_opts (file upload)

Parameters

- **arr** (Table) –
- **file_id** (Optional[str]) –
- **file_opts** (dict) –
- **upload_url_opts** (str) –
- **memoize** (bool) –

Return type Tuple[str, dict]

create_file(*file_opts={}*)

Creates File and returns file_id str.

Defauls:

- file_type: 'arrow'

See File REST API for file_opts

Parameters **file_opts** (dict) –

Return type str

post_arrow(*arr*, *file_id*, *url_opts='erase=true'*)
Upload new data to existing file id

Default url_opts='erase=true' throws exceptions on parse errors and deletes upload.

See File REST API for url_opts (file upload)

Parameters

- **arr** (Table) –
- **file_id** (str) –
- **url_opts** (str) –

Return type dict

uploader: Any = None

graphistry.ArrowFileUploader.**DF_TO_FILE_ID_CACHE: weakref.WeakKeyDictionary = <WeakKeyDict**

NOTE: Will switch to pa.Table -> ... when RAPIDS upgrades from pyarrow, which adds weakref support

class graphistry.ArrowFileUploader.**MemoizedFileUpload**(*file_id*, *output*)
Bases: object

Parameters

- **file_id** (str) –
- **output** (dict) –

file_id: str

output: dict

class graphistry.ArrowFileUploader.**WrappedTable**(*arr*)
Bases: object

Parameters **arr** (Table) –

arr: pyarrow.lib.Table

graphistry.ArrowFileUploader.**cache_arr**(*arr*)
Hold reference to most recent memoization entries Hack until RAPIDS supports Arrow 2.0, when pa.Table becomes weakly referenceable

CHAPTER

ELEVEN

VERSIONEER

Git implementation of _version.py.

exception graphistry._version.**NotThisMethod**
Bases: Exception

Exception raised if a method is not valid for the current scenario.

class graphistry._version.**VersioneerConfig**
Bases: object

Container for Versioneer configuration parameters.

graphistry._version.**get_config**()
Create, populate and return the VersioneerConfig() object.

graphistry._version.**get_keywords**()
Get the keywords needed to look up the version information.

graphistry._version.**get_versions**()
Get version information or return default if unable to do so.

graphistry._version.**git_get_keywords**(*versionfile_abs*)
Extract version information from the given file.

graphistry._version.**git_pieces_from_vcs**(*tag_prefix*, *root*, *verbose*, *run_command=<function run_command>*)
Get version from 'git describe' in the root of the source tree.

This only gets called if the git-archive 'subst' keywords were *not* expanded, and _version.py hasn't already been rewritten with a short version string, meaning we're inside a checked out source tree.

graphistry._version.**git_versions_from_keywords**(*keywords*, *tag_prefix*, *verbose*)
Get version information from git keywords.

graphistry._version.**plus_or_dot**(*pieces*)
Return a + if we don't already have one, else return a .

graphistry._version.**register_vcs_handler**(*vcs*, *method*)
Create decorator to mark a method as the handler of a VCS.

graphistry._version.**render**(*pieces*, *style*)
Render the given version pieces into the requested style.

graphistry._version.**render_git_describe**(*pieces*)
TAG[-DISTANCE-gHEX][-dirty].

Like 'git describe –tags –dirty –always'.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

graphistry._version.**render_git_describe_long**(*pieces*)
TAG-DISTANCE-gHEX[-dirty].

Like 'git describe –tags –dirty –always -long'. The distance/hash is unconditional.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

graphistry._version.**render_pep440**(*pieces*)
Build up version string, with post-release "local version identifier".

Our goal: TAG[+DISTANCE.gHEX[.dirty]] . Note that if you get a tagged build and then dirty it, you'll get TAG+0.gHEX.dirty

Exceptions: 1: no tags. git_describe was just HEX. 0+untagged.DISTANCE.gHEX[.dirty]

graphistry._version.**render_pep440_old**(*pieces*)
TAG[.postDISTANCE[.dev0]] .

The ".dev0" means dirty.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

graphistry._version.**render_pep440_post**(*pieces*)
TAG[.postDISTANCE[.dev0]+gHEX] .

The ".dev0" means dirty. Note that .dev0 sorts backwards (a dirty tree will appear "older" than the corresponding clean one), but you shouldn't be releasing software with -dirty anyways.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

graphistry._version.**render_pep440_pre**(*pieces*)
TAG[.post0.devDISTANCE] – No -dirty.

Exceptions: 1: no tags. 0.post0.devDISTANCE

graphistry._version.**run_command**(*commands*, *args*, *cwd=None*, *verbose=False*, *hide_stderr=False*, *env=None*)
Call the given command(s).

graphistry._version.**versions_from_parentdir**(*parentdir_prefix*, *root*, *verbose*)
Try to determine the version from the parent directory name.

Source tarballs conventionally unpack into a directory that includes both the project name and a version string. We will also support searching up two directory levels for an appropriately named parent directory

CHAPTER

TWELVE

MODULES

12.1 graphistry.plugins_types package

12.1.1 graphistry.layout.utils package

graphistry.layout.graph package

Submodules

graphistry.layout.graph.edge module

class graphistry.layout.graph.edge.**Edge**(*x*, *y*, *w=1*, *data=None*, *connect=False*)
Bases: *graphistry.layout.graph.edgeBase.EdgeBase*

A graph edge.

Attributes

- data (object): an optional payload
- w (int): an optional weight associated with the edge (default 1) used by Dijkstra to find min-flow paths.
- feedback (bool): whether the Tarjan algorithm has inverted this edge to de-cycle the graph.

attach()
Attach this edge to the edge collections of the vertices.

data: object

degree: int
Is 0 if a loop, otherwise 1.

detach()
Removes this edge from the edge collections of the vertices.

feedback: bool

w: int

graphistry.layout.graph.edgeBase module

class graphistry.layout.graph.edgeBase.**EdgeBase**(*x*, *y*)
Bases: object

Base class for edges.

Attributes

- degree (int): degree of the edge (number of unique vertices).
- v (list[Vertex]): list of vertices associated with this edge.

degree: int
Is 0 if a loop, otherwise 1.

graphistry.layout.graph.graph module

class graphistry.layout.graph.graph.**Graph**(*vertices=None*, *edges=None*, *directed=True*)
Bases: object

N(*v*, *f_io=0*)

add_edge(*e*)
add edge e and its vertices into the Graph possibly merging the associated graph_core components

add_edges(*edges*)

Parameters **edges** (List) –

add_vertex(*v*)
add vertex v into the Graph as a new component

component_class
alias of *graphistry.layout.graph.graphBase.GraphBase*

connected()
returns the list of components

deg_avg()
the average degree of vertices

deg_max()
the maximum degree of vertices

deg_min()
the minimum degree of vertices

edges()

eps()
the graph epsilon value (norm/order), average number of edges per vertex.

get_vertex_from_data(*data*)

get_vertices_count()

norm()
the norm of the graph (number of edges)

order()
the order of the graph (number of vertices)

path(*x*, *y*, *f_io=0*, *hook=None*)

remove_edge(*e*)
remove edge e possibly spawning two new cores if the graph_core that contained e gets disconnected.

remove_vertex(*x*)
remove vertex v and all its edges.

vertices()
see graph_core

graphistry.layout.graph.graphBase module

class graphistry.layout.graph.graphBase.**GraphBase**(*vertices=None*, *edges=None*, *directed=True*)

Bases: object

A connected graph of Vertex/Edge objects. A GraphBase is a *component* of a Graph that contains a connected set of Vertex and Edges.

Attributes: verticesPoset (Poset[Vertex]): the partially ordered set of vertices of the graph. edgesPoset (Poset[Edge]): the partially ordered set of edges of the graph. loops (set[Edge]): the set of *loop* edges (of degree 0). directed (bool): indicates if the graph is considered *oriented* or not.

N(*v*, *f_io=0*)

add_edge(*e*)
add edge e. At least one of its vertex must belong to the graph, the other being added automatically.

add_single_vertex(*v*)
allow a GraphBase to hold a single vertex.

complement(*G*)

constant_function(*value*)

contract(*e*)

deg_avg()
the average degree of vertices

deg_max()
the maximum degree of vertices

deg_min()
the minimum degree of vertices

dft(*start_vertex=None*)

dijkstra(*x*, *f_io=0*, *hook=None*)
shortest weighted-edges paths between x and all other vertices by dijkstra's algorithm with heap used as priority queue.

edges(*cond=None*)
generates an iterator over edges, with optional filter

eps()
the graph epsilon value (norm/order), average number of edges per vertex.

get_scs_with_feedback(*roots=None*)

Minimum FAS algorithm (feedback arc set) creating a DAG. Returns the set of strongly connected components ("scs") by using Tarjan algorithm. These are maximal sets of vertices such that there is a path from each vertex to every other vertex. The algorithm performs a DFS from the provided list of root vertices. A cycle is of course a strongly connected component,but a strongly connected

component can include several cycles. The Feedback Acyclic Set of edge to be removed/reversed is provided by marking the edges with a "feedback" flag. Complexity is O(V+E).

Parameters **roots** –

Returns

leaves()
returns the list of *leaves* (vertices with no outward edges).

matrix(*cond=None*)

This associativity matrix is like the adjacency matrix but antisymmetric. Returns the associativity matrix of the graph component

Parameters **cond** – same a the condition function in vertices().

Returns array

norm()
The size of the edge poset (number of edges).

order()
the order of the graph (number of vertices)

partition()

path(*x*, *y*, *f_io=0*, *hook=None*)
shortest path between vertices x and y by breadth-first descent, contrained by f_io direction if provided. The path is returned as a list of Vertex objects. If a *hook* function is provided, it is called at every vertex added to the path, passing the vertex object as argument.

remove_edge(*e*)
remove Edge e, asserting that the resulting graph is still connex.

remove_vertex(*x*)
remove Vertex x and all associated edges.

roots()
returns the list of *roots* (vertices with no inward edges).

spans(*vertices*)

union_update(*G*)

vertices(*cond=None*)
generates an iterator over vertices, with optional filter

graphistry.layout.graph.vertex module

class graphistry.layout.graph.vertex.**Vertex**(*data=None*)
Bases: *graphistry.layout.graph.vertexBase.VertexBase*

Vertex class enhancing a VertexBase with graph-related features.

Attributes component (GraphBase): the component of connected vertices that contains this vertex. By default, a vertex belongs no component but when it is added in a graph, c points to the connected component in this graph. data (object) : an object associated with the vertex.

property index

graphistry.layout.graph.vertexBase module

class graphistry.layout.graph.vertexBase.**VertexBase**
Bases: object

Base class for vertices.

Attributes e (list[Edge]): list of edges associated with this vertex.

degree()
degree() : degree of the vertex (number of edges).

detach()
removes this vertex from all its edges and returns this list of edges.

e_dir(*dir*)
either e_in, e_out or all edges depending on provided direction parameter (>0 means outward).

e_from(*x*)
returns the Edge from vertex v directed toward this vertex.

e_in()
e_in() : list of edges directed toward this vertex.

e_out()
e_out(): list of edges directed outward this vertex.

e_to(*y*)
returns the Edge from this vertex directed toward vertex v.

e_with(*v*)
return the Edge with both this vertex and vertex v

neighbors(*direction=0*)
Returns the neighbors of this vertex. List of neighbor vertices in all directions (default) or in filtered f_io direction (>0 means outward).

Parameters **direction** –

- 0: parent and children
- -1: parents
- +1: children

Returns list of vertices

Module contents

graphistry.layout.gib

Submodules

graphistry.layout.utils.dummyVertex module

class graphistry.layout.utils.dummyVertex.**DummyVertex**(*r=None*)
Bases: *graphistry.layout.utils.layoutVertex.LayoutVertex*

A DummyVertex is used for edges that span over several layers, it's inserted in every inner layer.

Attributes

- view (viewclass): since a DummyVertex is acting as a Vertex, it must have a view.
- ctrl (list[_sugiyama_attr]): the list of associated dummy vertices.

inner(*direction*)
True if a neighbor in the given direction is *dummy*.

neighbors(*direction*)
Reflect the Vertex method and returns the list of adjacent vertices (possibly dummy) in the given direction. :type direction: int :param direction: +1 for the next layer (children) and -1 (parents) for the previous

graphistry.layout.utils.geometry module

graphistry.layout.utils.geometry.**angle_between_vectors**(*p1*, *p2*)

graphistry.layout.utils.geometry.**lines_intersection**(*xy1*, *xy2*, *xy3*, *xy4*)
Returns the intersection of two lines.

graphistry.layout.utils.geometry.**new_point_at_distance**(*pt*, *distance*, *angle*)

graphistry.layout.utils.geometry.**rectangle_point_intersection**(*rec*, *p*)
Returns the intersection point between the Rectangle (w,h) that characterize the rec object and the line that goes from the recs' object center to the 'p' point.

graphistry.layout.utils.geometry.**set_round_corner**(*e*, *pts*)

graphistry.layout.utils.geometry.**setcurve**(*e*, *pts*, *tgs=None*)

Returns the spline curve that path through the list of points P. The spline curve is a list of cubic bezier curves (nurbs) that have matching tangents at their extreme points. The method considered here is taken from "The NURBS book" (Les A. Piegl, Wayne Tiller, Springer, 1997) and implements a local interpolation rather than a global interpolation.

Args: e: pts: tgs:

Returns:

graphistry.layout.utils.geometry.**size_median**(*recs*)

graphistry.layout.utils.geometry.**tangents**(*P*, *n*)

graphistry.layout.utils.layer module

class graphistry.layout.utils.layer.**Layer**(*iterable=()*, */*)
Bases: list

Layer is where Sugiyama layout organises vertices in hierarchical lists. The placement of a vertex is done by the Sugiyama class, but it highly relies on the *ordering* of vertices in each layer to reduce crossings. This ordering depends on the neighbors found in the upper or lower layers.

Attributes: layout (SugiyamaLayout): a reference to the sugiyama layout instance that contains this layer upper (Layer): a reference to the *upper* layer (layer-1) lower (Layer): a reference to the *lower* layer (layer+1) crossings (int) : number of crossings detected in this layer

Methods: setup (layout): set initial attributes values from provided layout nextlayer(): returns *next* layer in the current layout's direction parameter. prevlayer(): returns *previous* layer in the current layout's direction parameter. order(): compute *optimal* ordering of vertices within the layer.

crossings = None

layout = None

lower = None

neighbors(*v*)
neighbors refer to upper/lower adjacent nodes. Note that v.neighbors() provides neighbors of v in the graph, while this method provides the Vertex and DummyVertex adjacent to v in the upper or lower layer (depending on layout.dirv state).

nextlayer()

order()

prevlayer()

setup(*layout*)

upper = None

graphistry.layout.utils.layoutVertex module

class graphistry.layout.utils.layoutVertex.**LayoutVertex**(*layer=None*, *is_dummy=0*)
Bases: object

The Sugiyama layout adds new attributes to vertices. These attributes are stored in an internal _sugimyama_vertex_attr object.

Attributes: layer (int): layer number dummy (0/1): whether the vertex is a dummy pos (int): the index of the vertex within the layer x (list(float)): the list of computed horizontal coordinates of the vertex bar (float): the current barycenter of the vertex

Parameters **layer** (Optional[int]) –

graphistry.layout.utils.poset module

class graphistry.layout.utils.poset.**Poset**(*collection=[]*)
Bases: object

Poset class implements a set but allows to integrate over the elements in a deterministic way and to get specific objects in the set. Membership operator defaults to comparing __hash__ of objects but Poset allows to check for __cmp__/__eq__ membership by using contains__cmp__(obj)

add(*obj*)

contains__cmp__(*obj*)

copy()

deepcopy()

difference(**args*)

get(*obj*)

index(*obj*)

intersection(**args*)

issubset(*other*)

issuperset(*other*)

remove(*obj*)

symmetric_difference(**args*)

union(*other*)

update(*other*)

graphistry.layout.utils.rectangle module

class graphistry.layout.utils.rectangle.**Rectangle**(*w=1*, *h=1*)
Bases: object

Rectangular region.

graphistry.layout.utils.routing module

class graphistry.layout.utils.routing.**EdgeViewer**
Bases: object

setpath(*pts*)

graphistry.layout.utils.routing.**route_with_lines**(*e*, *pts*)
Basic edge routing with lines. The layout pass has already provided to list of points through which the edge shall be drawn. We just compute the position where to adjust the tail and head.

graphistry.layout.utils.routing.**route_with_rounded_corners**(*e*, *pts*)

graphistry.layout.utils.routing.**route_with_splines**(*e*, *pts*)
Enhanced edge routing where 'corners' of the above polyline route are rounded with a Bezier curve.

Module contents

12.1.2 Submodules

12.1.3 graphistry.plugins_types.cugraph_types module

12.1.4 Module contents

CHAPTER

THIRTEEN

ARTICLES

- Graphistry: Visual Graph AI Interactive demo
- PyGraphistry + Databricks
- PyGraphistry + UMAP

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W